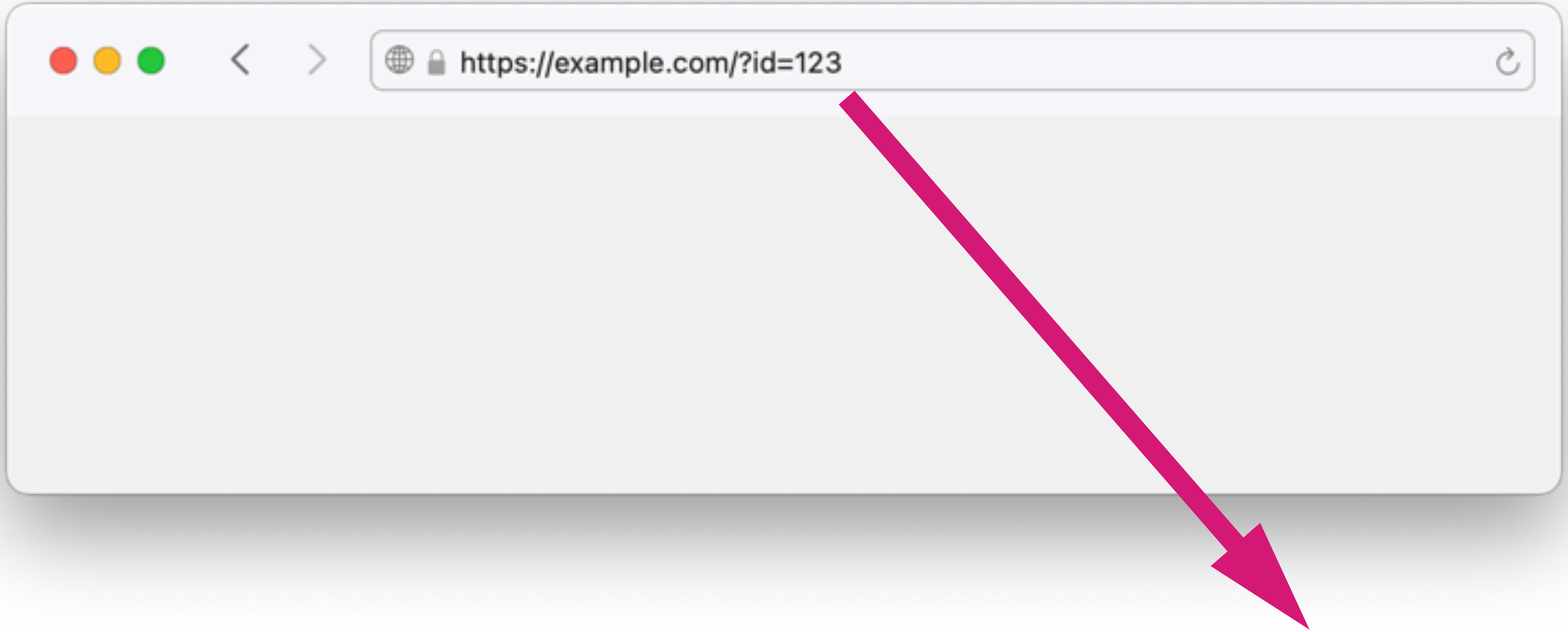
PRESENTED BY: Craig Francis

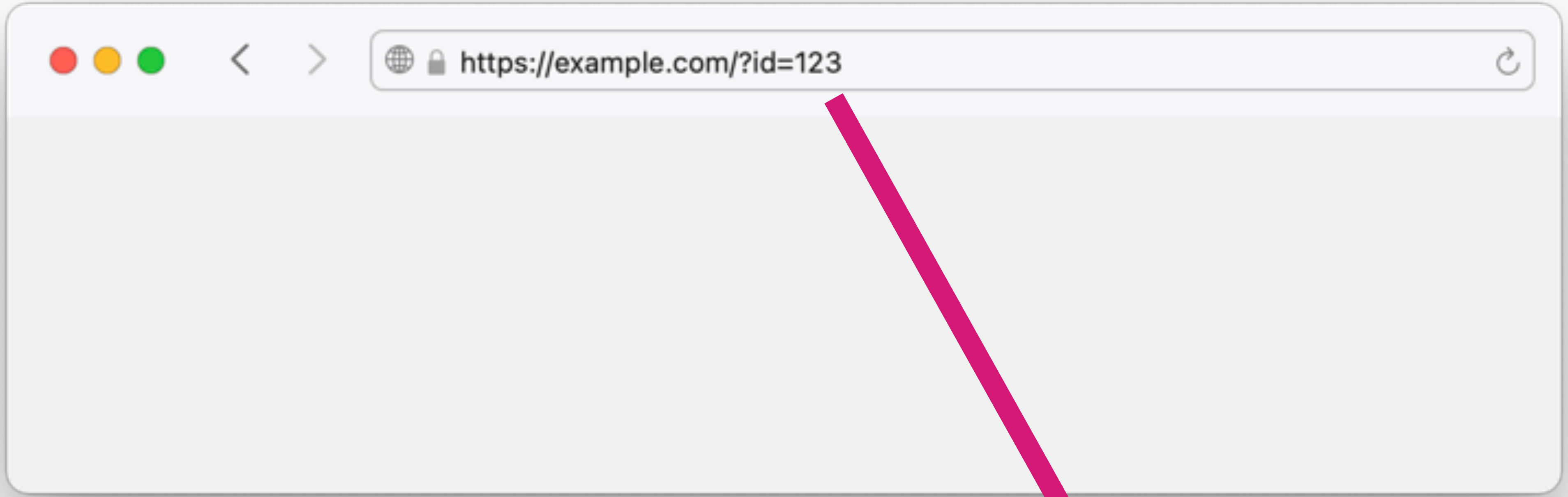
Ending Injection Vulnerabilities,
Using developer defined strings.

https://eiv.dev/



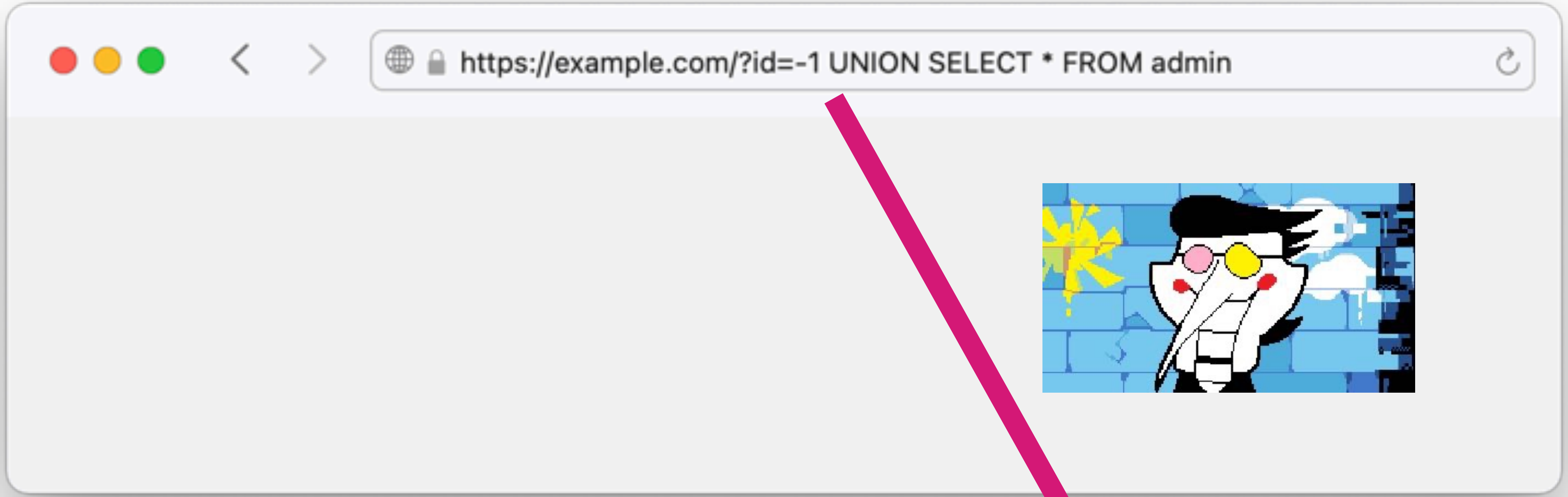
https://example.com/?id=123

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



```
$sql = 'SELECT * FROM user WHERE id = [illegible] $_GET['id'];
```

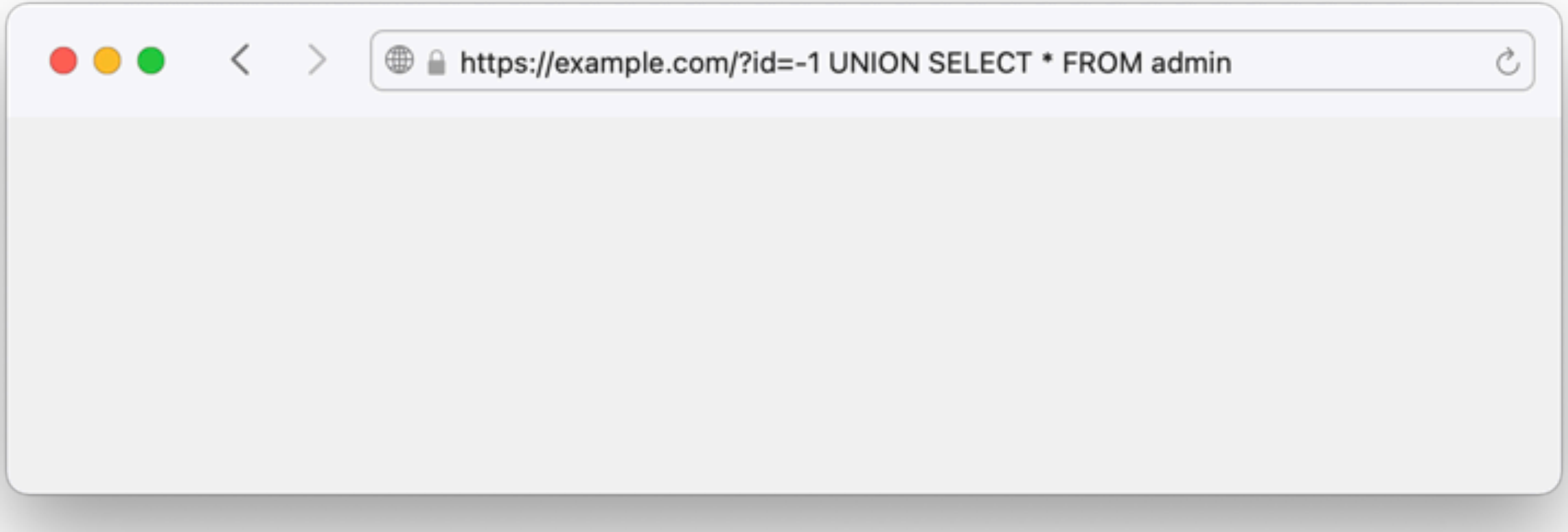
```
SELECT * FROM user WHERE id = 123
```



https://example.com/?id=-1 UNION SELECT * FROM admin

```
$sql = 'SELECT * FROM user WHERE id = $_GET['id'];
```

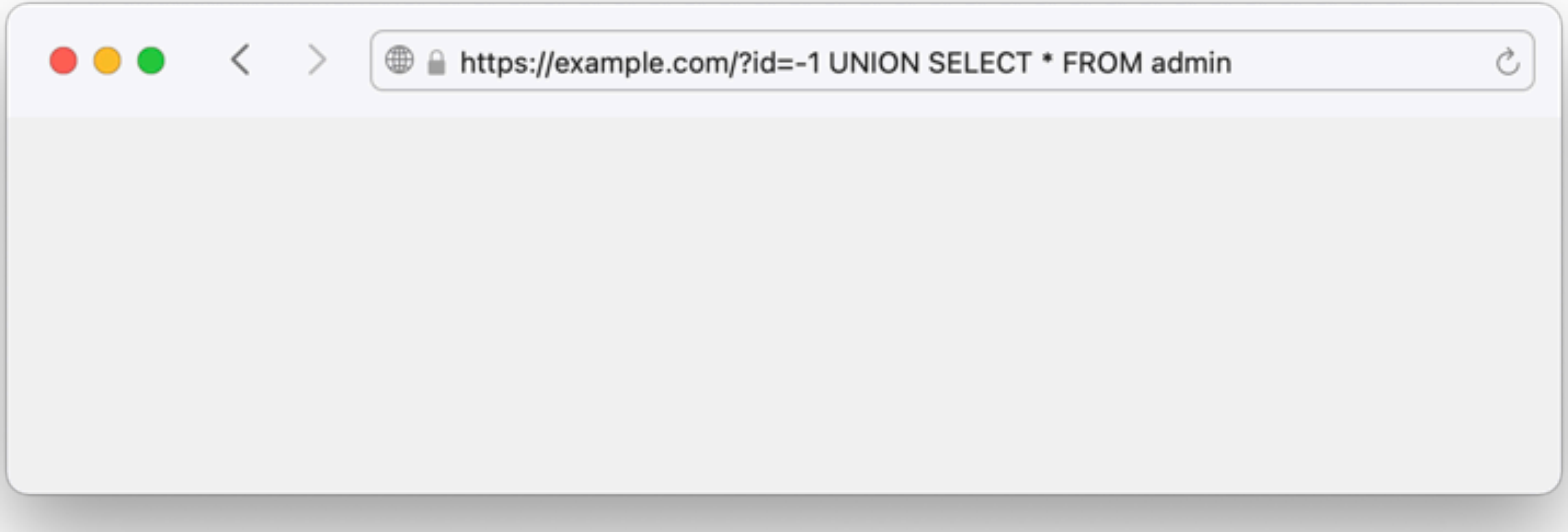
```
SELECT * FROM user WHERE id = -1 UNION SELECT * FROM admin
```



https://example.com/?id=-1 UNION SELECT * FROM admin

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

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

👎



https://example.com/?id=-1 UNION SELECT * FROM admin

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

$db->query($sql, $_GET['id']);
```

👍

QueryBuilders

```
$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'
```

Doctrine QueryBuilder

Doctrine

```
$qb->select('u')
    ->from('User', 'u')
    ->where('u.id = :identifier')
    ->setParameter('identifier', $_GET['id']);
```

Doctrine

```
$qb->select('u')
    ->from('User', 'u')
    ->where('u.id = ' . $_GET['id']); // INSECURE
```

Demo Time

https://laravel.examples.eiv.dev/users/example-a/

Laravel DB

whereRaw()

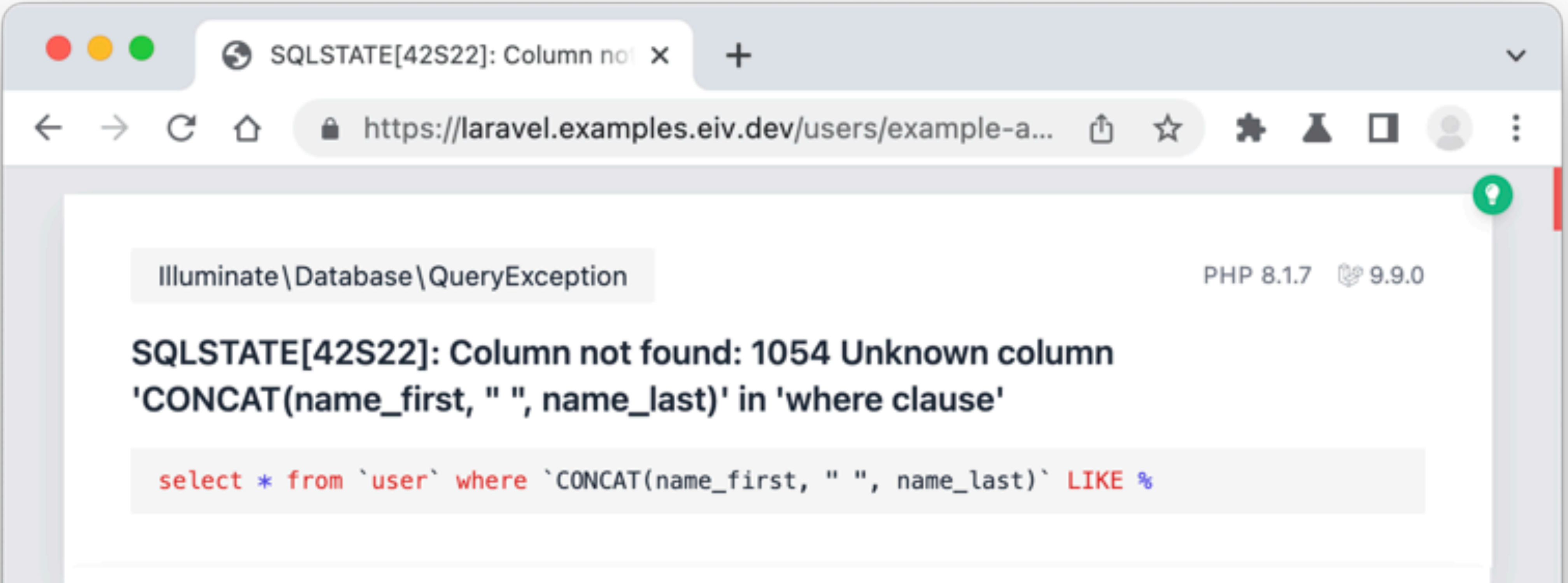
Laravel DB

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

Laravel DB

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

```
DB::table('user')
    ->where('CONCAT(name_first, " ", name_last)', 'LIKE', $search . '%');
```

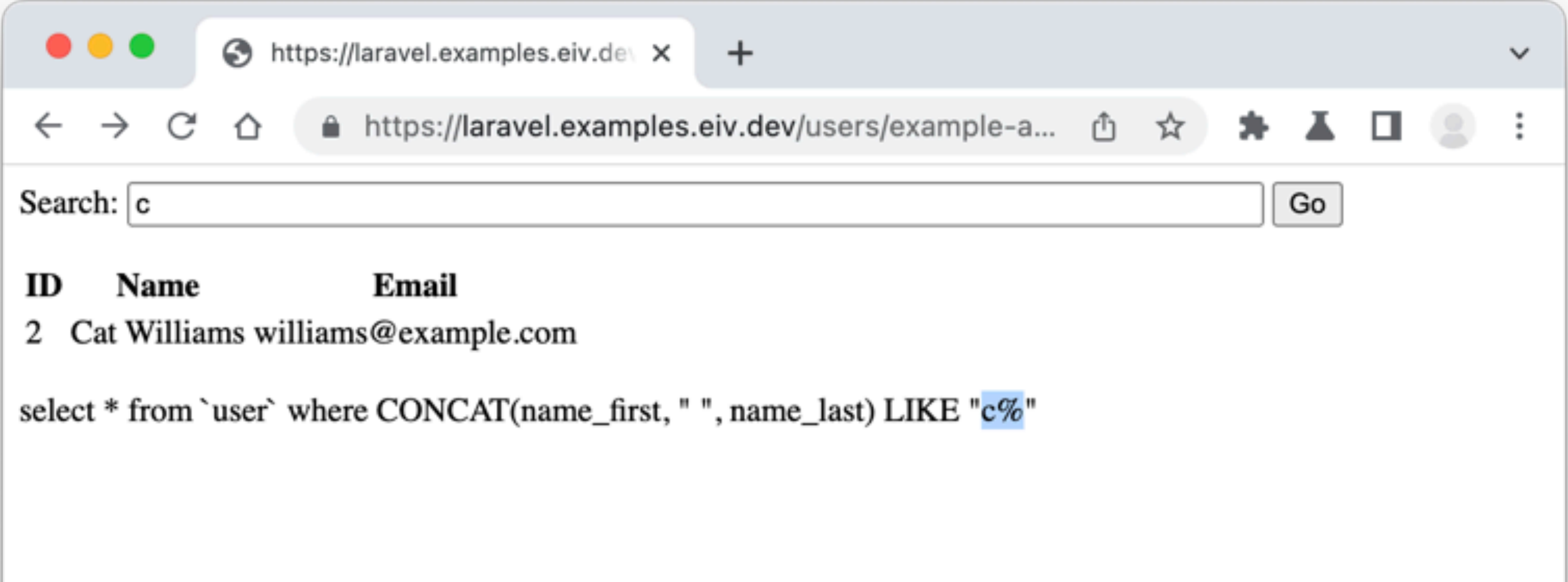

SQLSTATE[42S22]: Column no
https://laravel.examples.eiv.dev/users/example-a...
PHP 8.1.7
9.9.0

Illuminate\Database\QueryException

SQLSTATE[42S22]: Column not found: 1054 Unknown column 'CONCAT(name_first, " ", name_last)' in 'where clause'

```
select * from `user` where `CONCAT(name_first, " ", name_last)` LIKE %
```

```
DB::table('user')
    ->whereRaw('CONCAT(name_first, " ", name_last) LIKE ?', $search . '%');
```



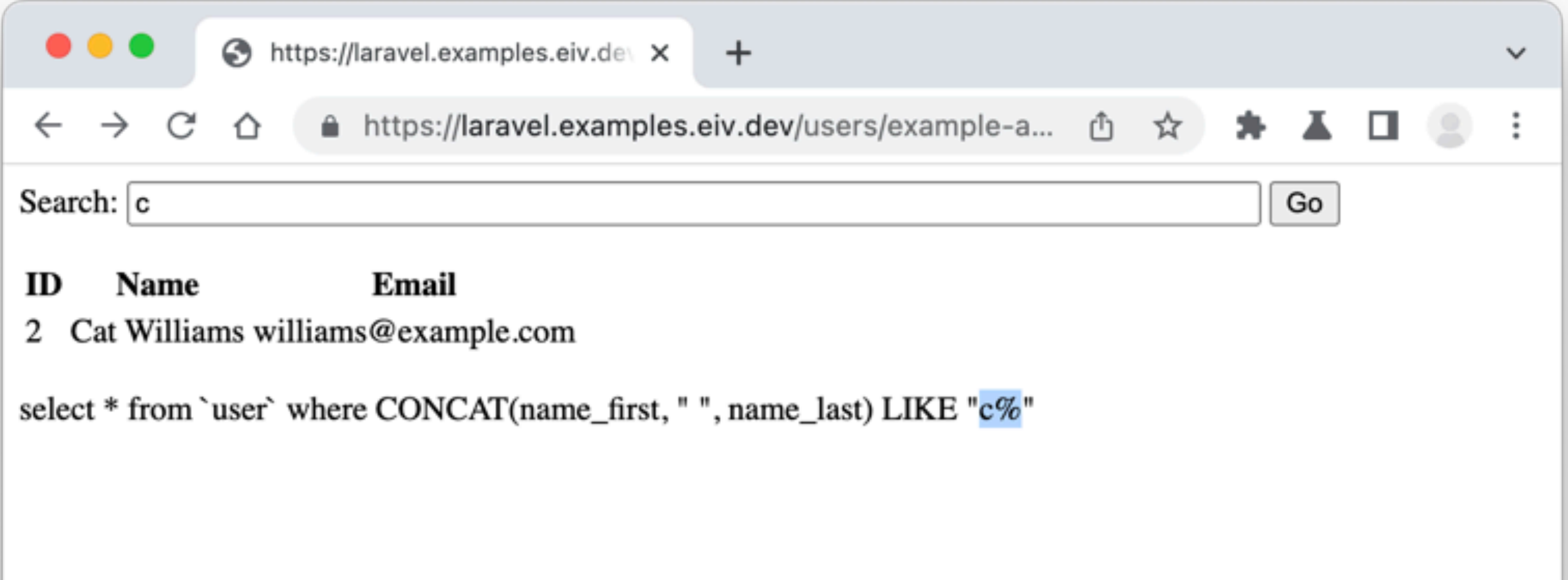
https://laravel.examples.eiv.de
https://laravel.examples.eiv.dev/users/example-a...
Go

Search: c

ID	Name	Email
2	Cat Williams	williams@example.com

```
select * from `user` where CONCAT(name_first, " ", name_last) LIKE "c%"
```

```
DB::table('user')
    ->whereRaw('CONCAT(name_first, " ", name_last) LIKE "' . $search . '%"');
```



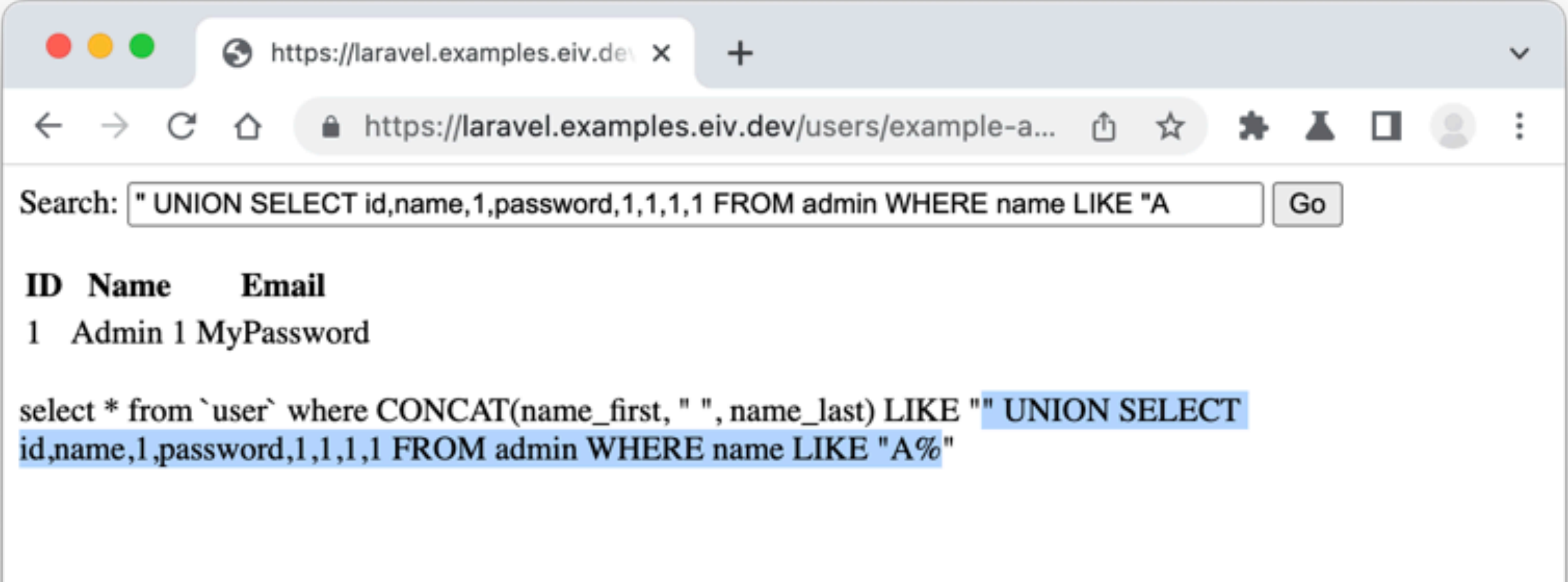
https://laravel.examples.eiv.de
https://laravel.examples.eiv.dev/users/example-a...
Go

Search: c

```
ID    Name           Email
2   Cat Williams williams@example.com

select * from `user` where CONCAT(name_first, " ", name_last) LIKE "c%"
```

Laravel DB

```
DB::table('user')
    ->whereRaw('CONCAT(name_first, " ", name_last) LIKE "' . $search . '%"');
```



https://laravel.examples.eiv.dev
https://laravel.examples.eiv.dev/users/example-a...

Search: " UNION SELECT id,name,1,password,1,1,1,1 FROM admin WHERE name LIKE "A Go

```
ID  Name    Email
1   Admin 1 MyPassword
```

select * from `user` where CONCAT(name_first, " ", name_last) LIKE "" UNION SELECT id,name,1,password,1,1,1,1 FROM admin WHERE name LIKE "A%"

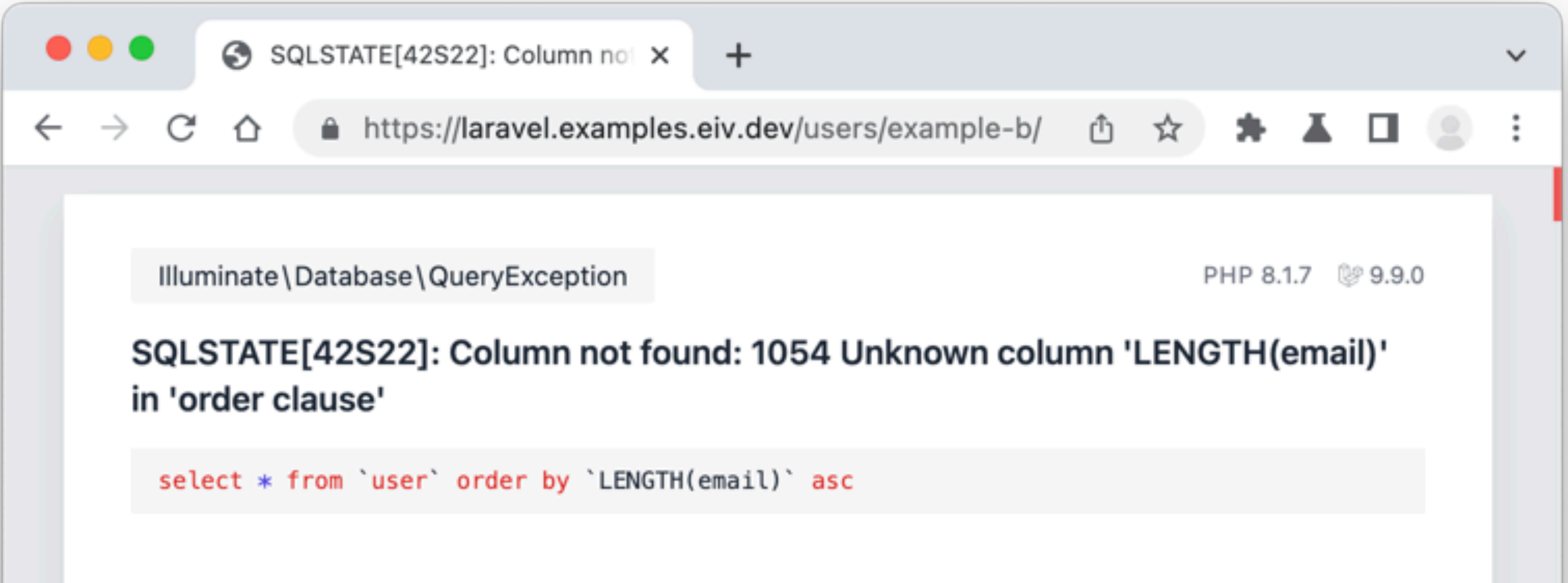
Demo Time

https://laravel.examples.eiv.dev/users/example-b/

Laravel DB

orderByRaw()

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



SQLSTATE[42S22]: Column no
https://laravel.examples.eiv.dev/users/example-b/

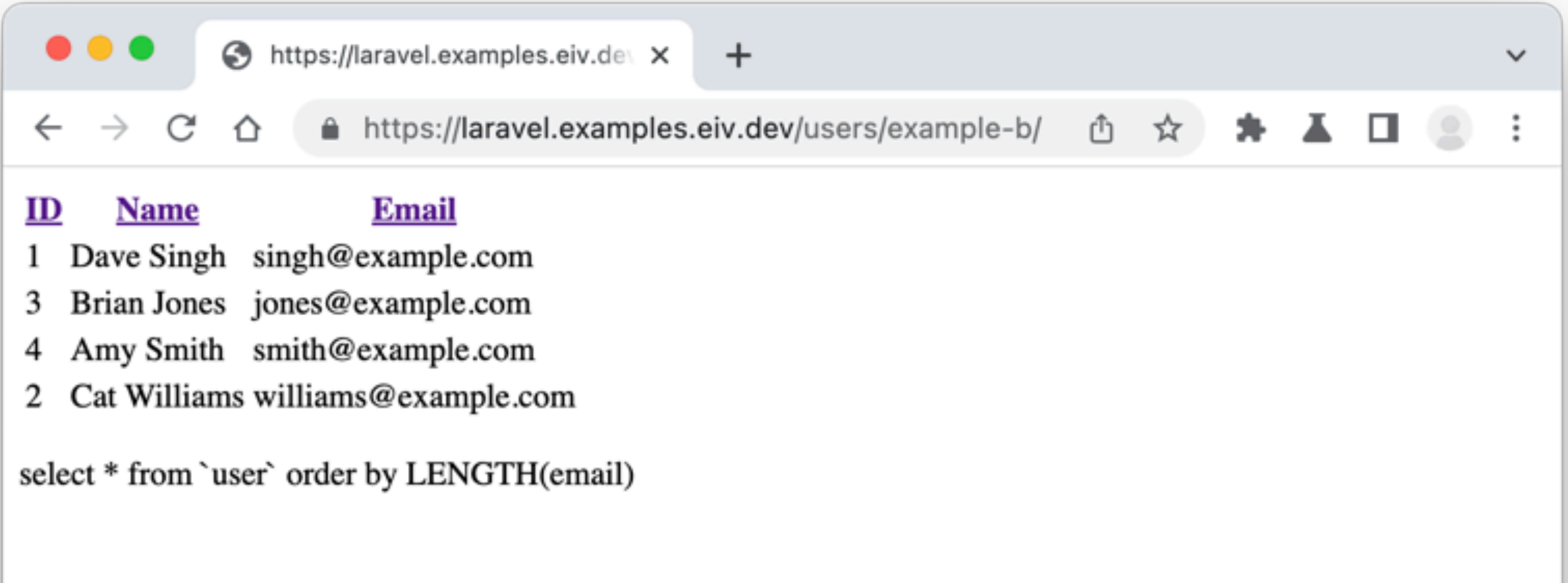
Illuminate\Database\QueryException

PHP 8.1.7 9.9.0

SQLSTATE[42S22]: Column not found: 1054 Unknown column 'LENGTH(email)' in 'order clause'

```
select * from `user` order by `LENGTH(email)` asc
```

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


https://laravel.examples.eiv.de
https://laravel.examples.eiv.dev/users/example-b/

ID	Name	Email
1	Dave Singh	singh@example.com
3	Brian Jones	jones@example.com
4	Amy Smith	smith@example.com
2	Cat Williams	williams@example.com

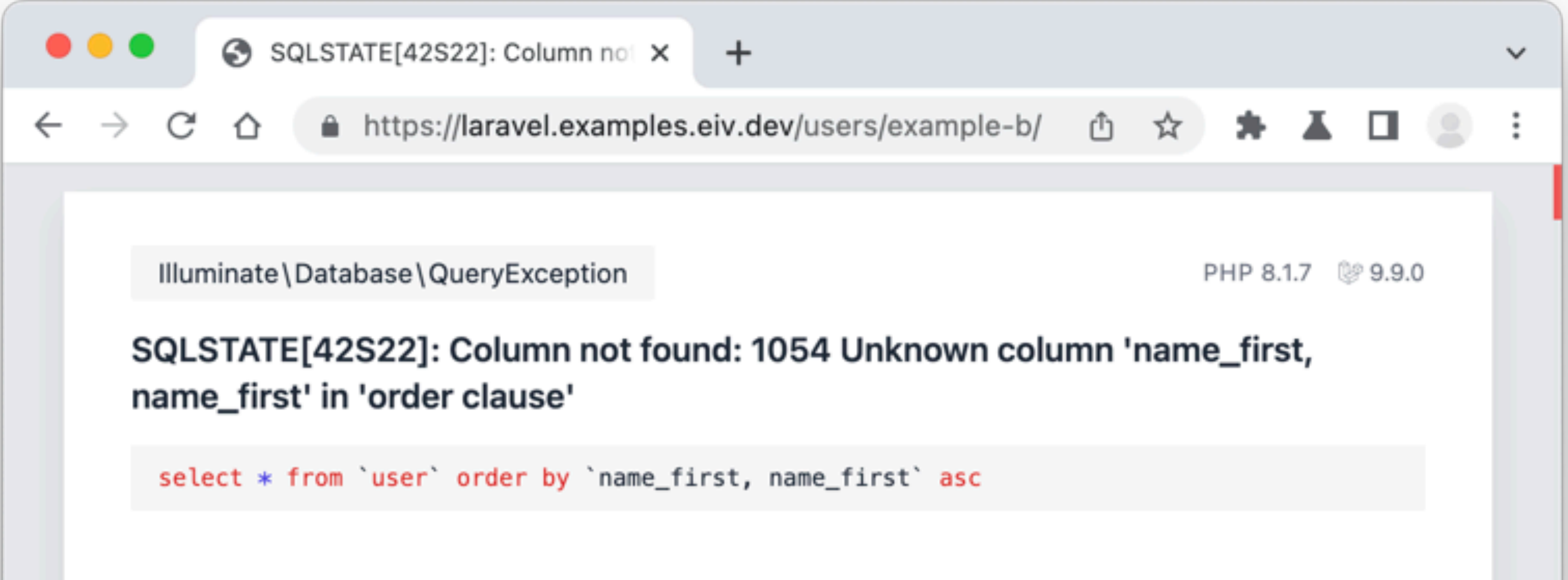
```
select * from `user` order by LENGTH(email)
```

Laravel DB

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

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

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



SQLSTATE[42S22]: Column not
https://laravel.examples.eiv.dev/users/example-b/

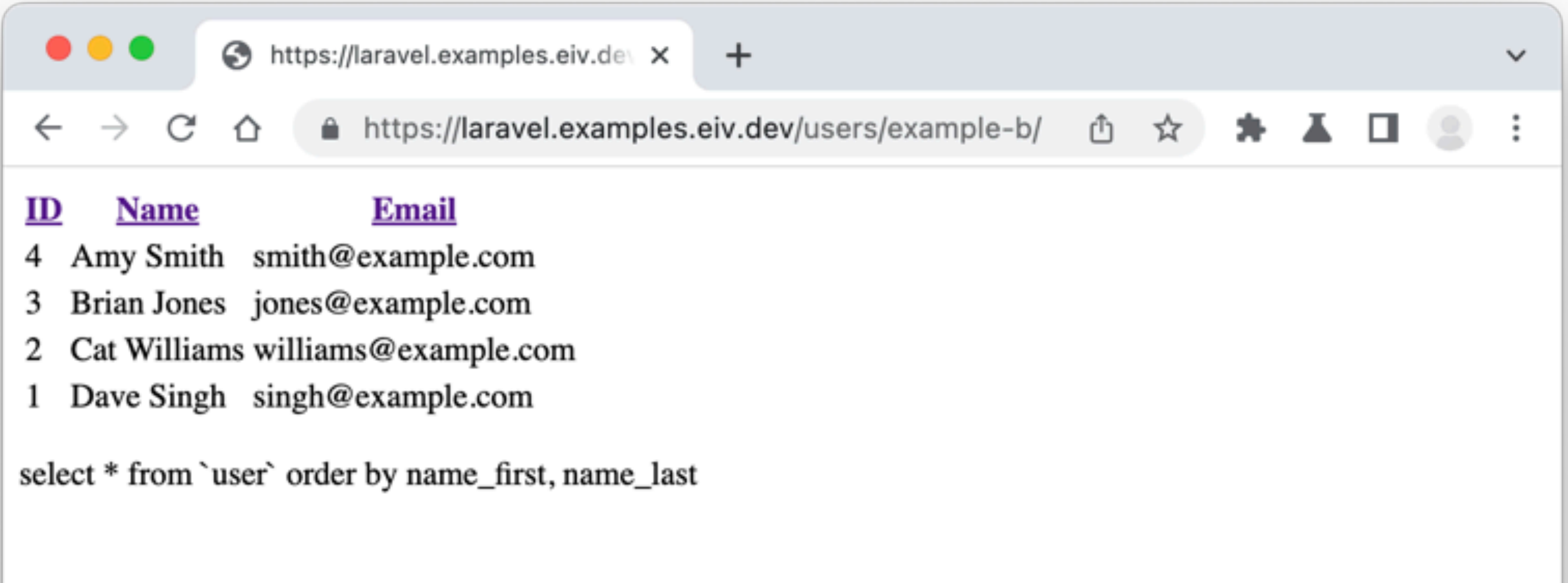
Illuminate\Database\QueryException

PHP 8.1.7 9.9.0

SQLSTATE[42S22]: Column not found: 1054 Unknown column 'name_first, name_first' in 'order clause'

```
select * from `user` order by `name_first, name_first` asc
```

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



https://laravel.examples.eiv.de
https://laravel.examples.eiv.dev/users/example-b/

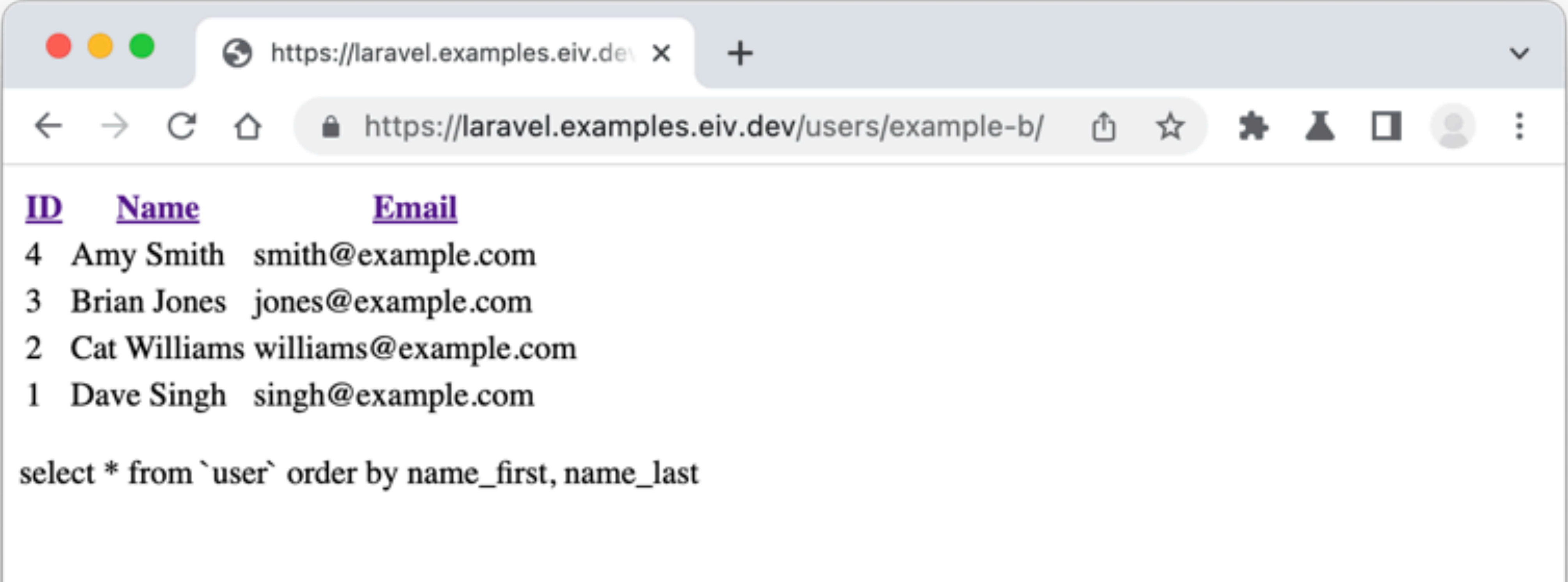
ID	Name	Email
4	Amy Smith	smith@example.com
3	Brian Jones	jones@example.com
2	Cat Williams	williams@example.com
1	Dave Singh	singh@example.com

```
select * from `user` order by name_first, name_last
```

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

Query String "/?sort=id"

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

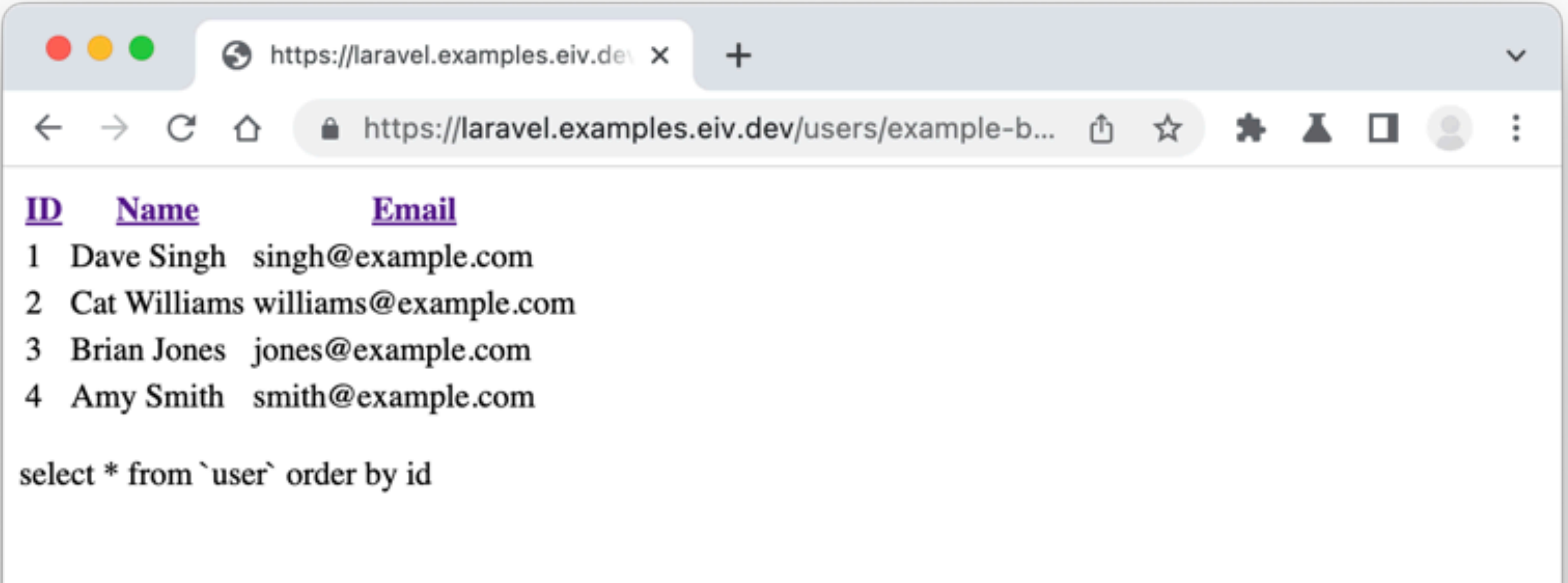

https://laravel.examples.eiv.de
https://laravel.examples.eiv.dev/users/example-b/

ID	Name	Email
4	Amy Smith	smith@example.com
3	Brian Jones	jones@example.com
2	Cat Williams	williams@example.com
1	Dave Singh	singh@example.com

```
select * from `user` order by name_first, name_last
```

Laravel DB

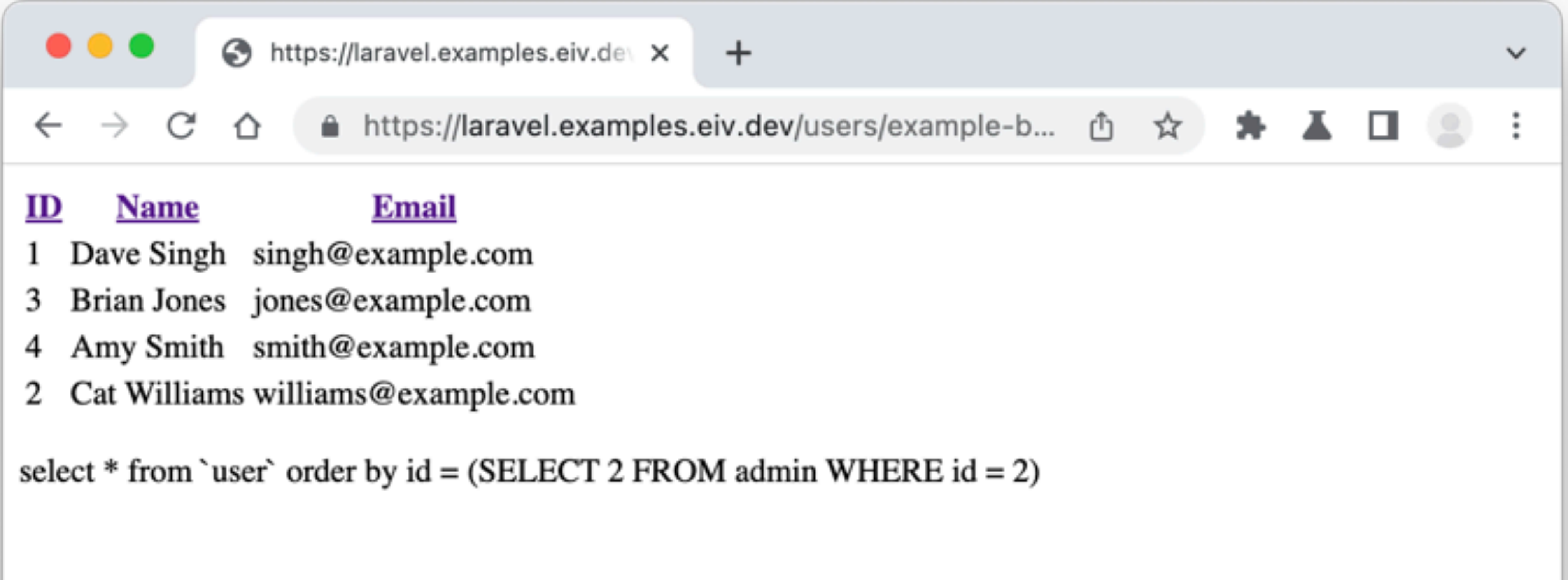
```
$sort = $request->input('sort');

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


https://laravel.examples.eiv.de
https://laravel.examples.eiv.dev/users/example-b...

```
ID  Name        Email
1   Dave Singh  singh@example.com
2   Cat Williams williams@example.com
3   Brian Jones  jones@example.com
4   Amy Smith   smith@example.com

select * from `user` order by id
```

Laravel DB

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

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



https://laravel.examples.eiv.de

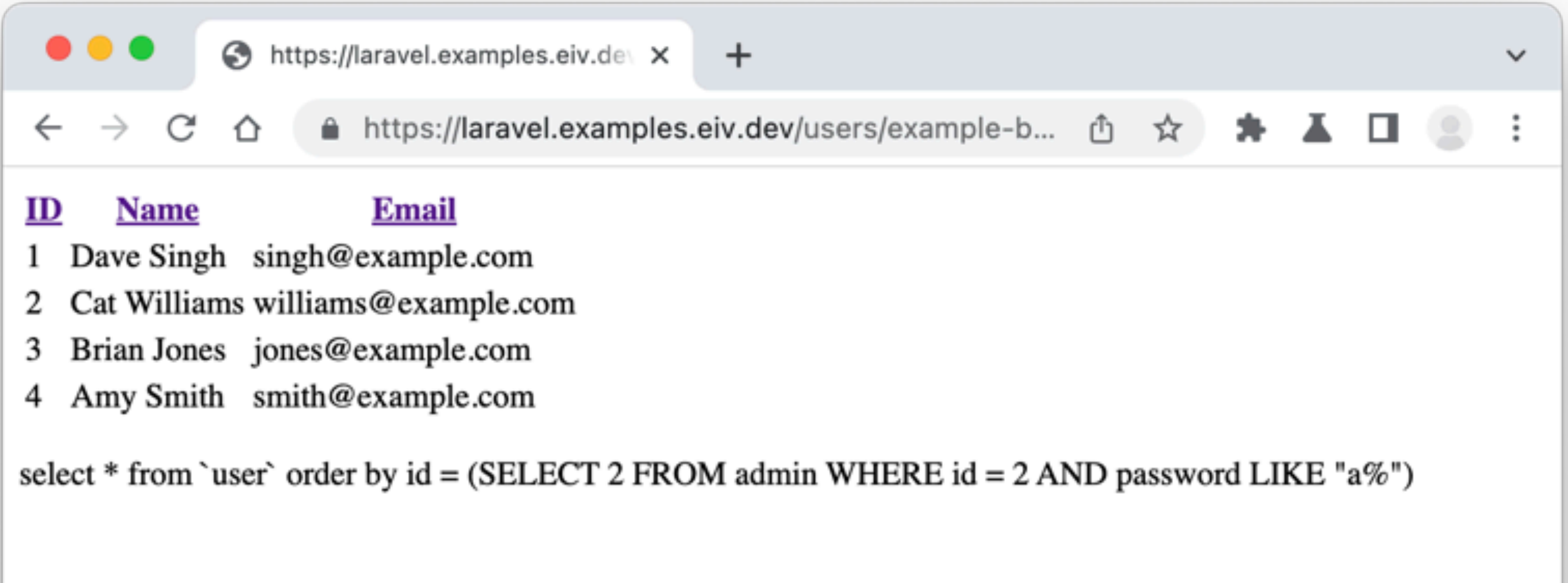
https://laravel.examples.eiv.dev/users/example-b...

ID	Name	Email
1	Dave Singh	singh@example.com
3	Brian Jones	jones@example.com
4	Amy Smith	smith@example.com
2	Cat Williams	williams@example.com

```
select * from `user` order by id = (SELECT 2 FROM admin WHERE id = 2)
```

Laravel DB

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

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

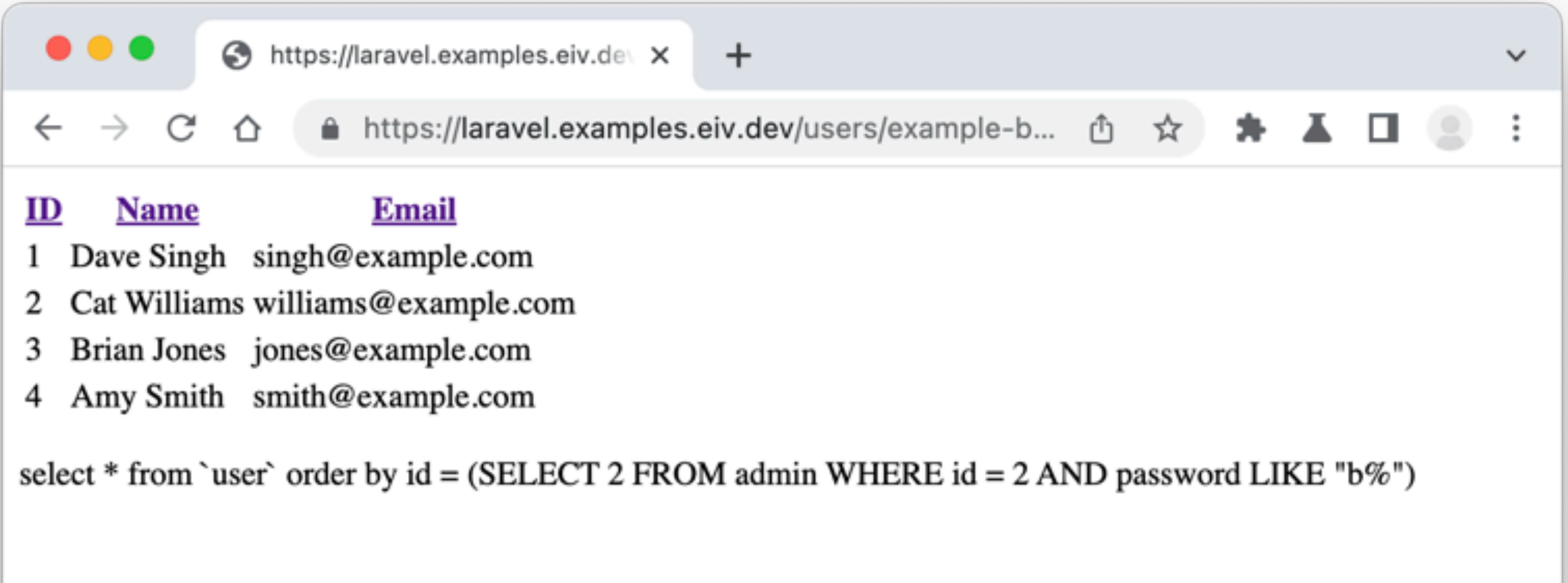

https://laravel.examples.eiv.de
https://laravel.examples.eiv.dev/users/example-b...

ID	Name	Email
1	Dave Singh	singh@example.com
2	Cat Williams	williams@example.com
3	Brian Jones	jones@example.com
4	Amy Smith	smith@example.com

```
select * from `user` order by id = (SELECT 2 FROM admin WHERE id = 2 AND password LIKE "a%")
```

Laravel DB

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

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

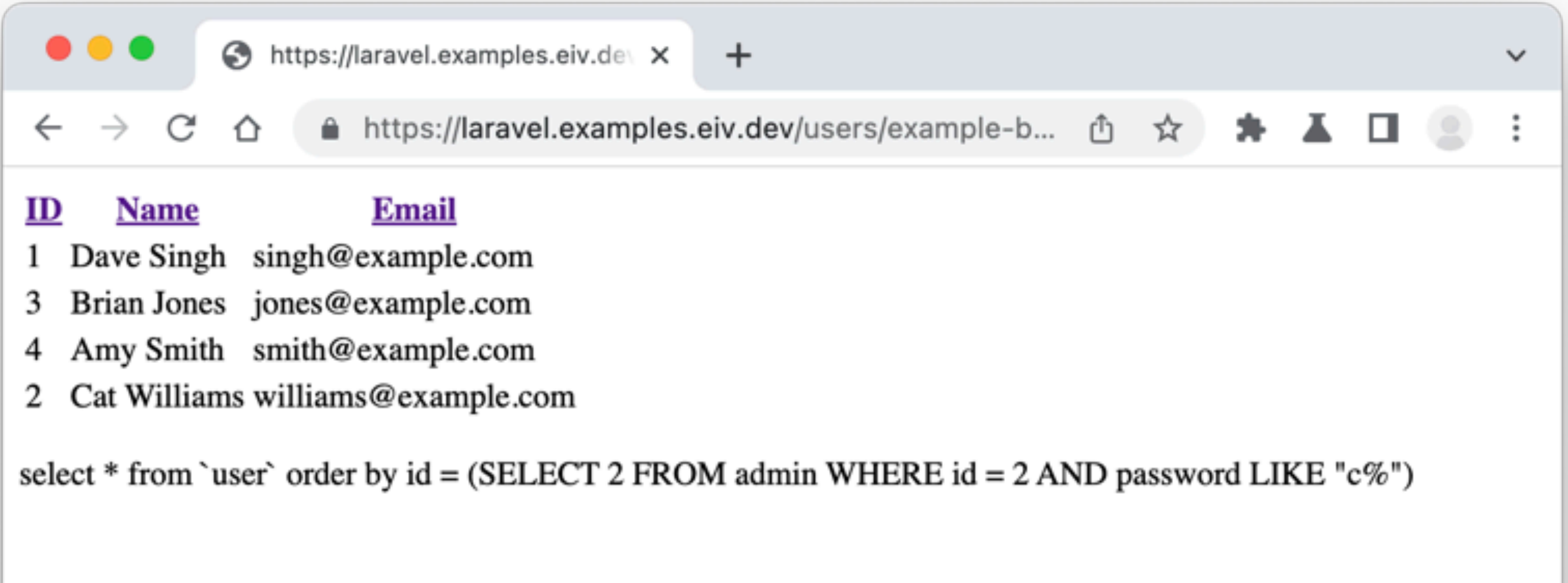

https://laravel.examples.eiv.de
https://laravel.examples.eiv.dev/users/example-b...

ID	Name	Email
1	Dave Singh	singh@example.com
2	Cat Williams	williams@example.com
3	Brian Jones	jones@example.com
4	Amy Smith	smith@example.com

```
select * from `user` order by id = (SELECT 2 FROM admin WHERE id = 2 AND password LIKE "b%")
```

Laravel DB

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

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


https://laravel.examples.eiv.de
https://laravel.examples.eiv.dev/users/example-b...

ID	Name	Email
1	Dave Singh	singh@example.com
3	Brian Jones	jones@example.com
4	Amy Smith	smith@example.com
2	Cat Williams	williams@example.com

```
select * from `user` order by id = (SELECT 2 FROM admin WHERE id = 2 AND password LIKE "c%")
```

Demo Time

https://laravel.examples.eiv.dev/users/example-c/?id=1

Laravel DB

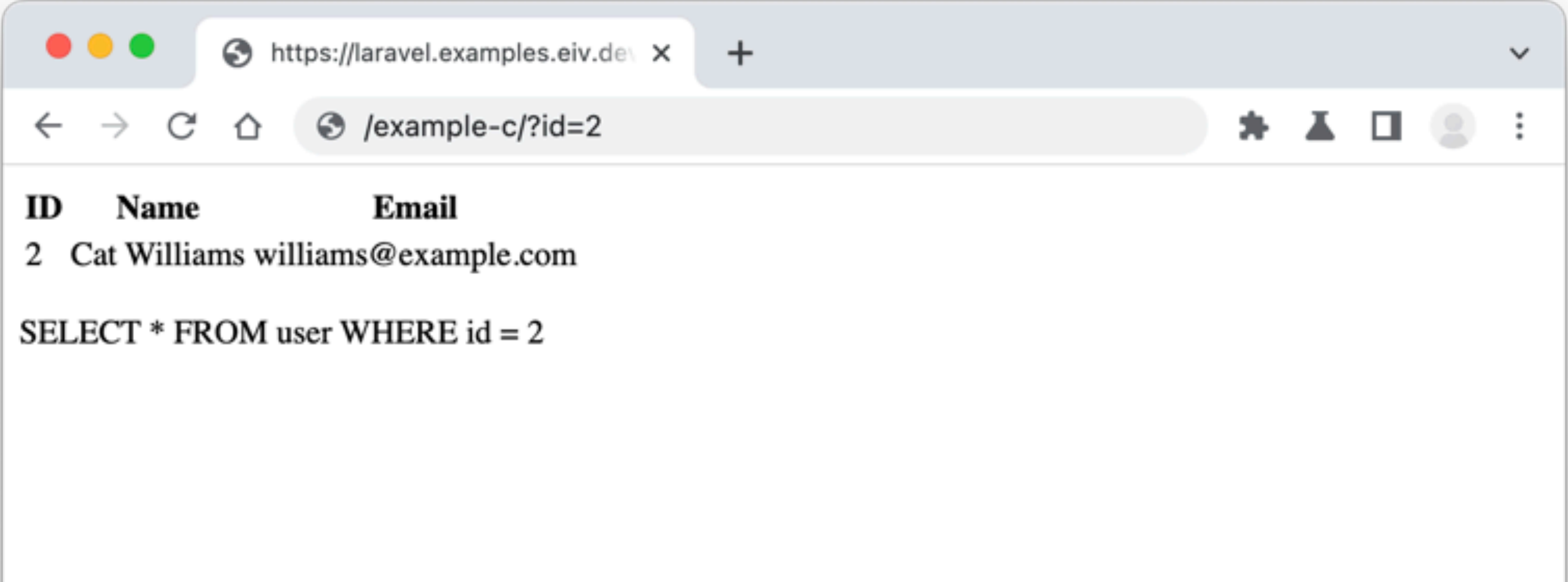
DB::select()

Laravel DB

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

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

Laravel DB

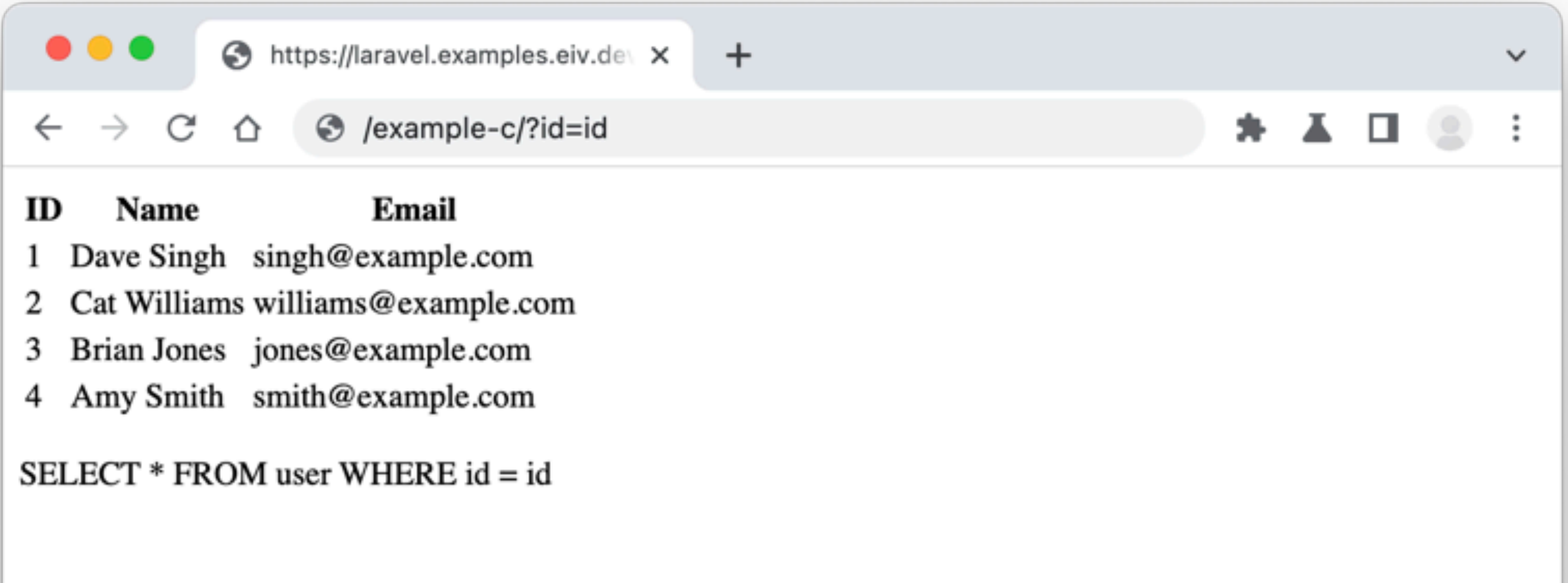
```
$id = $request->input('id');

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



```
ID   Name           Email
2  Cat Williams williams@example.com

SELECT * FROM user WHERE id = 2
```

Laravel DB

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

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



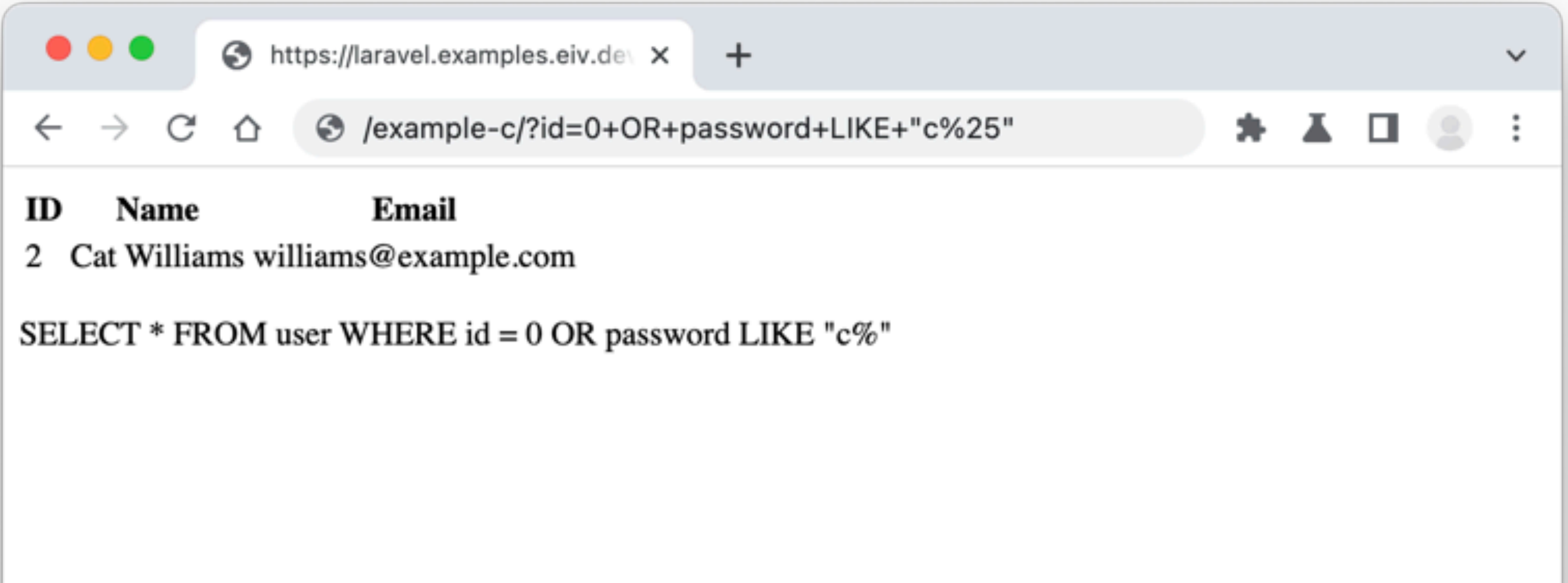
https://laravel.examples.eiv.de

/example-c/?id=id

ID	Name	Email
1	Dave Singh	singh@example.com
2	Cat Williams	williams@example.com
3	Brian Jones	jones@example.com
4	Amy Smith	smith@example.com

SELECT * FROM user WHERE id = id

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

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



https://laravel.examples.eiv.de
/example-c/?id=0+OR+password+LIKE+"c%25"

ID	Name	Email
2	Cat Williams	williams@example.com

```
SELECT * FROM user WHERE id = 0 OR password LIKE "c%"
```

Demo Time

https://twig.examples.eiv.dev/example-a/

Twig

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

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

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

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

https://twig.examples.eiv.dev/example-a/?name=%3Cscript%3Ealert(1)%3C/script%3E

Twig

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

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

👎

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

https://twig.examples.eiv.dev/example-b/?url=javascript:alert(1)

Context aware?

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

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

👎

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

https://twig.examples.eiv.dev/example-c/?url=%2F%20onerror=alert(1)

Missing quotes?

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

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

👎

```
'<img src=/ onerror=alert(1) alt="Alt Text" />'
```

Command Line Injection

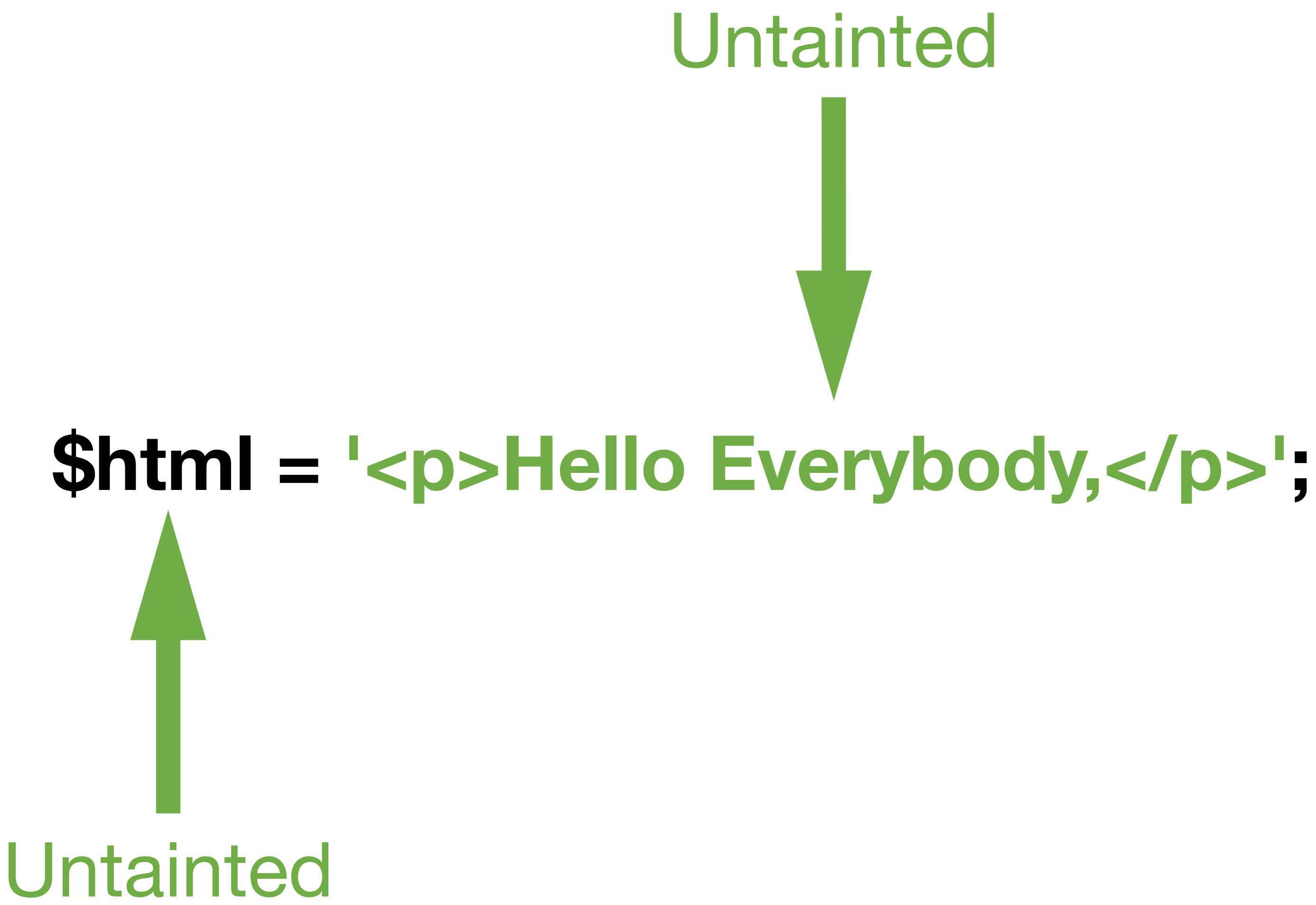
```
$exec = 'grep "' . $search . '" /path/to/file';
```

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

Taint Checking???

Where variables note if they are Tainted, or Untainted.

🤔



Untainted

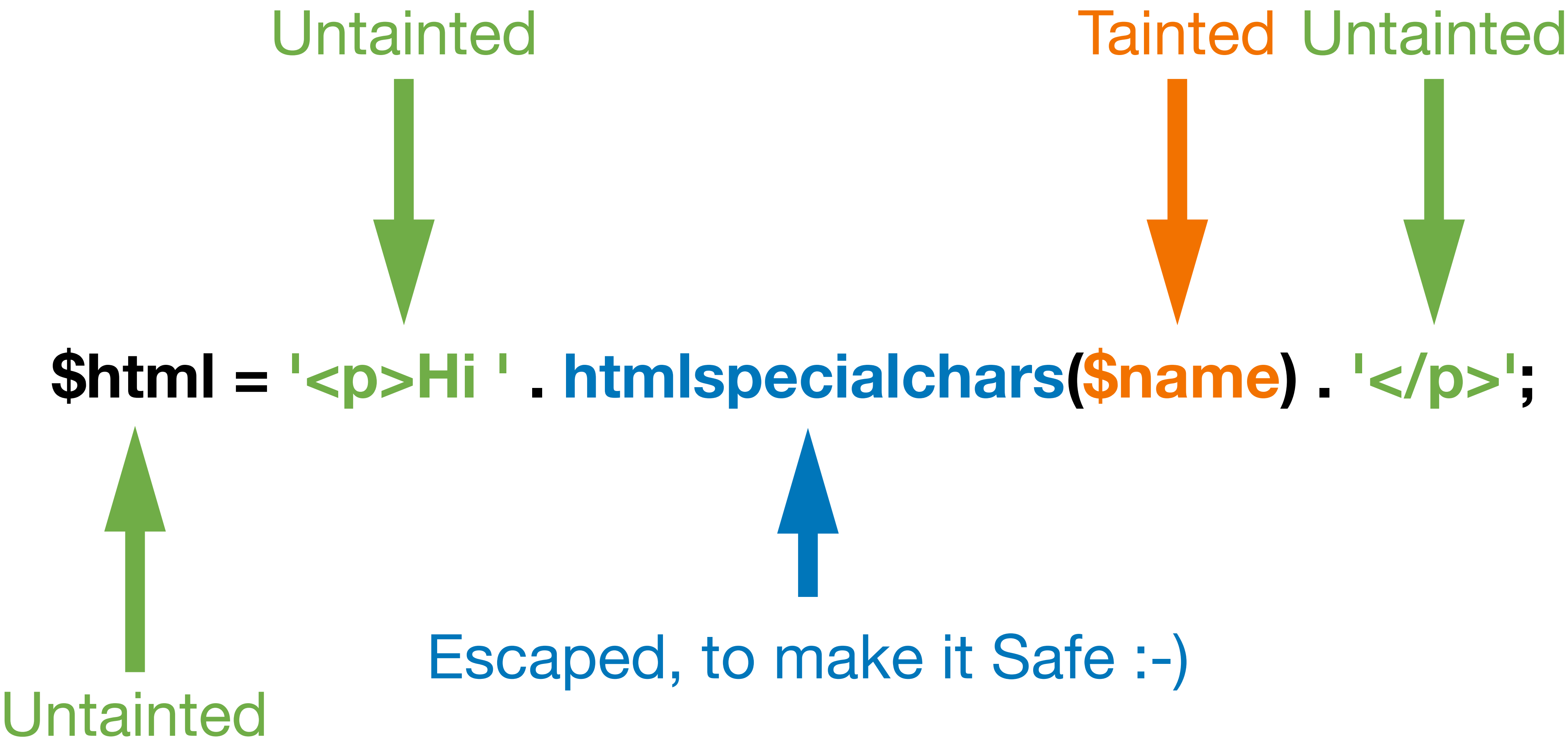
```
$html = '<p>Hello Everybody,</p>';
```

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 &lt;script&gt;alert()&lt;/script&gt;</p>
```

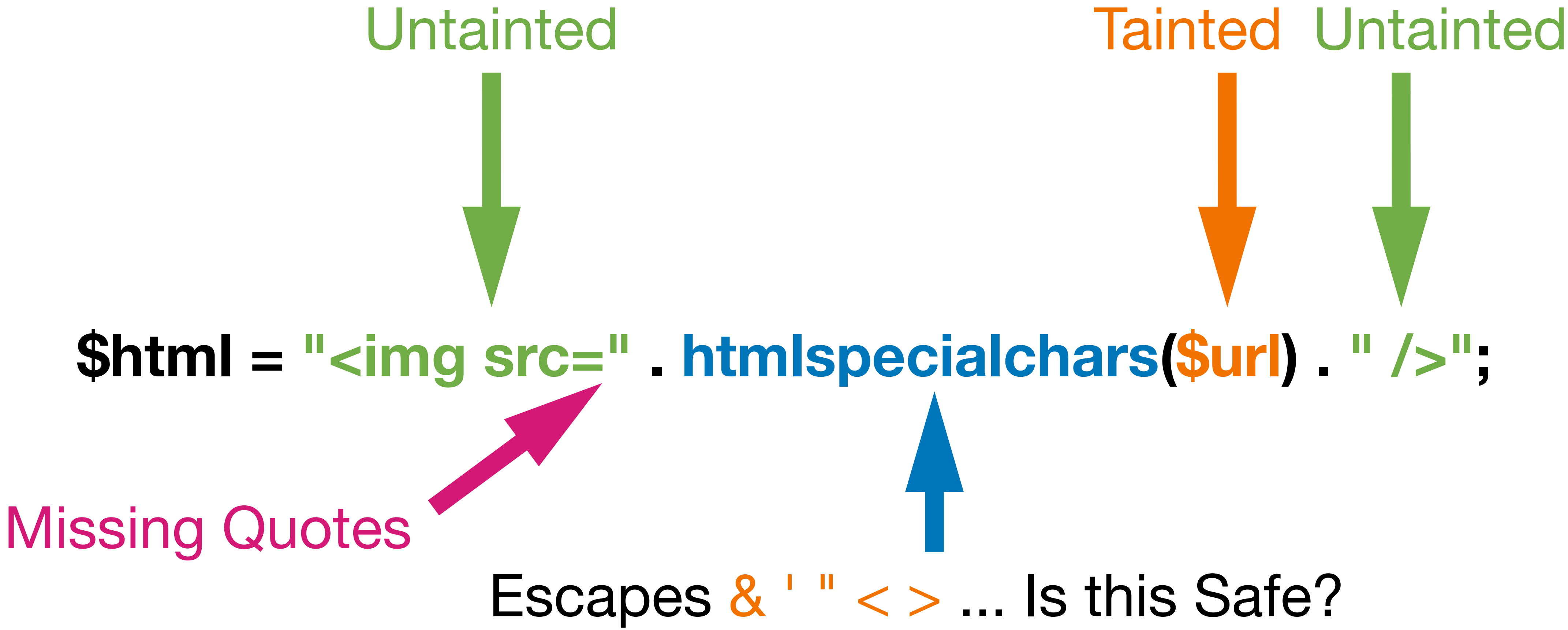
Unfortunately Taint Checking incorrectly assumes escaping makes a value “safe” for *any* context.

Untainted Tainted Untainted

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

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

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



Untainted

Tainted

Untainted

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

Missing Quotes

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

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

Untainted

Tainted

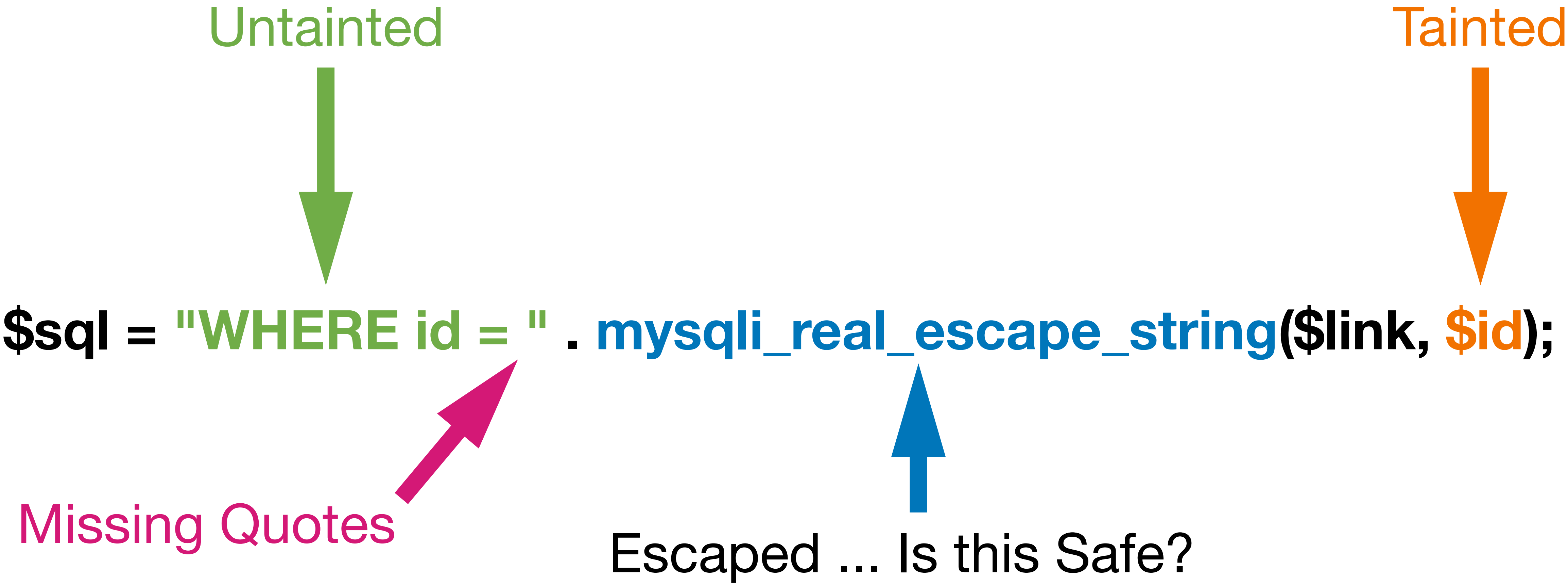
Untainted

```
$html = "<img src='" . htmlspecialchars($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 should be done by a Library.

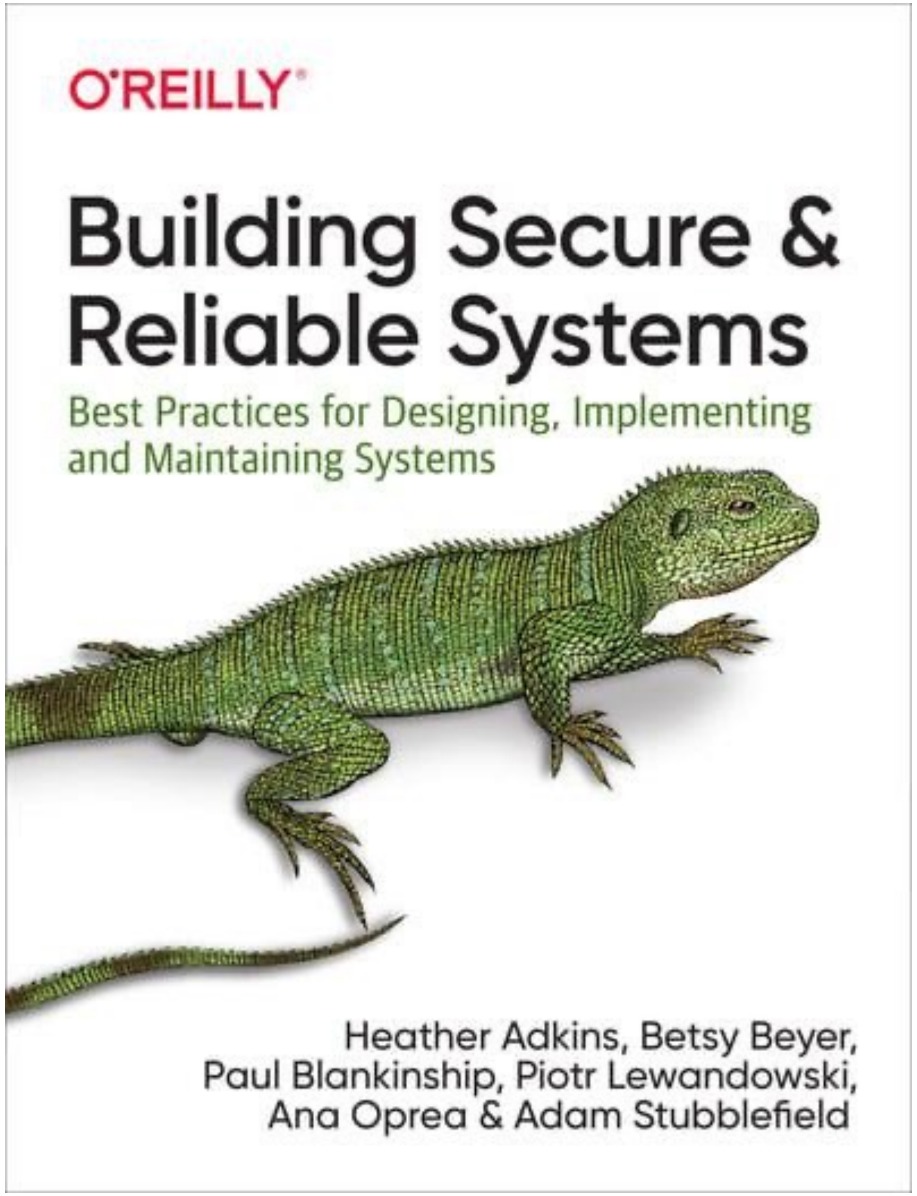
Christoph Kern

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

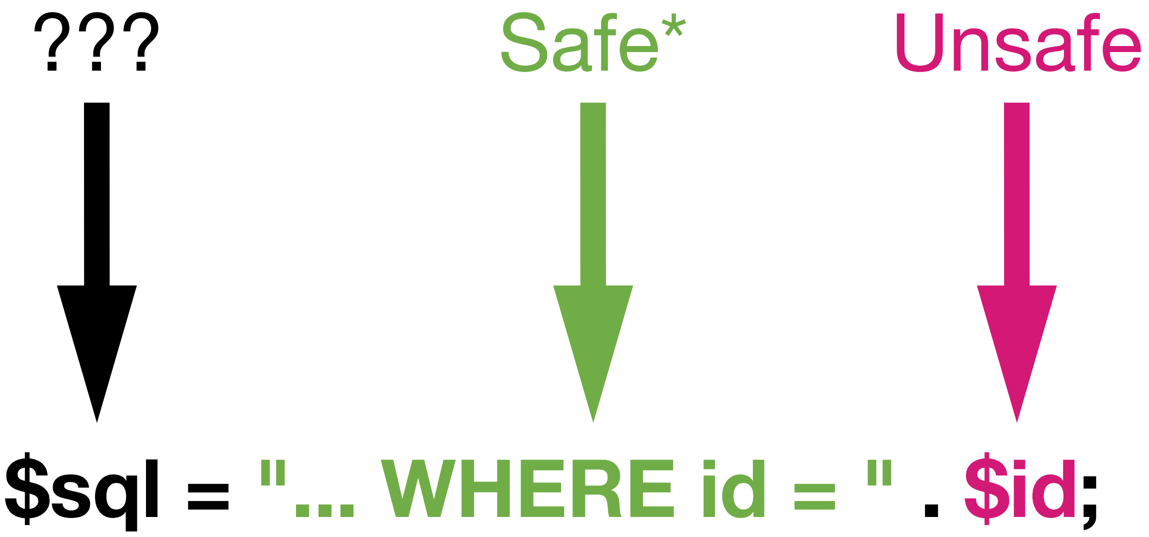
When talking about
Injection Vulnerabilities

Safe*

A developer defined string.
(in the source code)

Unsafe

Everything else.



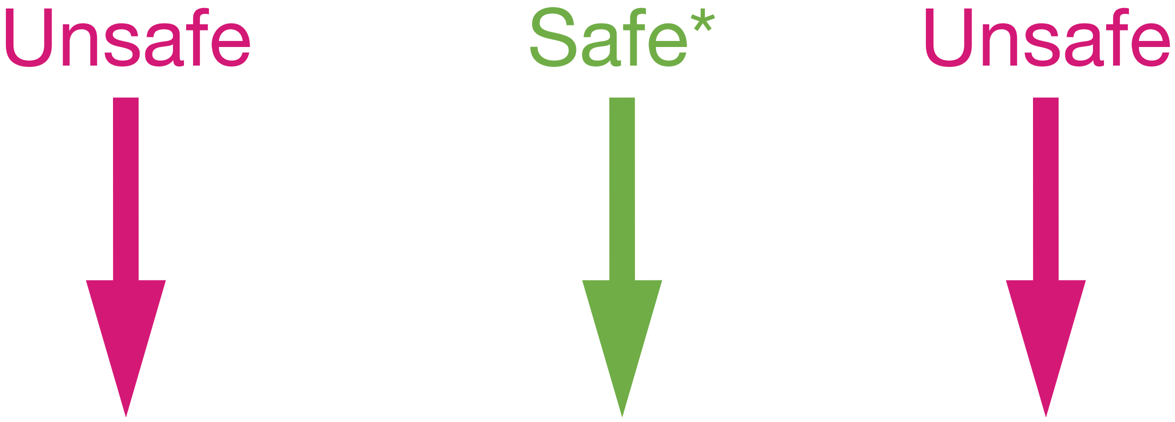
???

Safe*

Unsafe

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

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



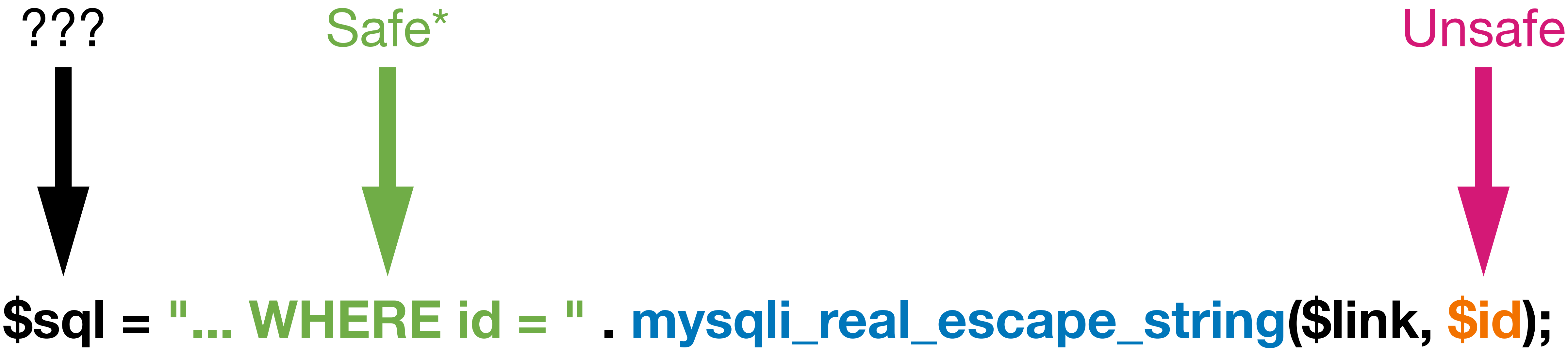
Unsafe

Safe*

Unsafe

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

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



???

Safe*

Unsafe

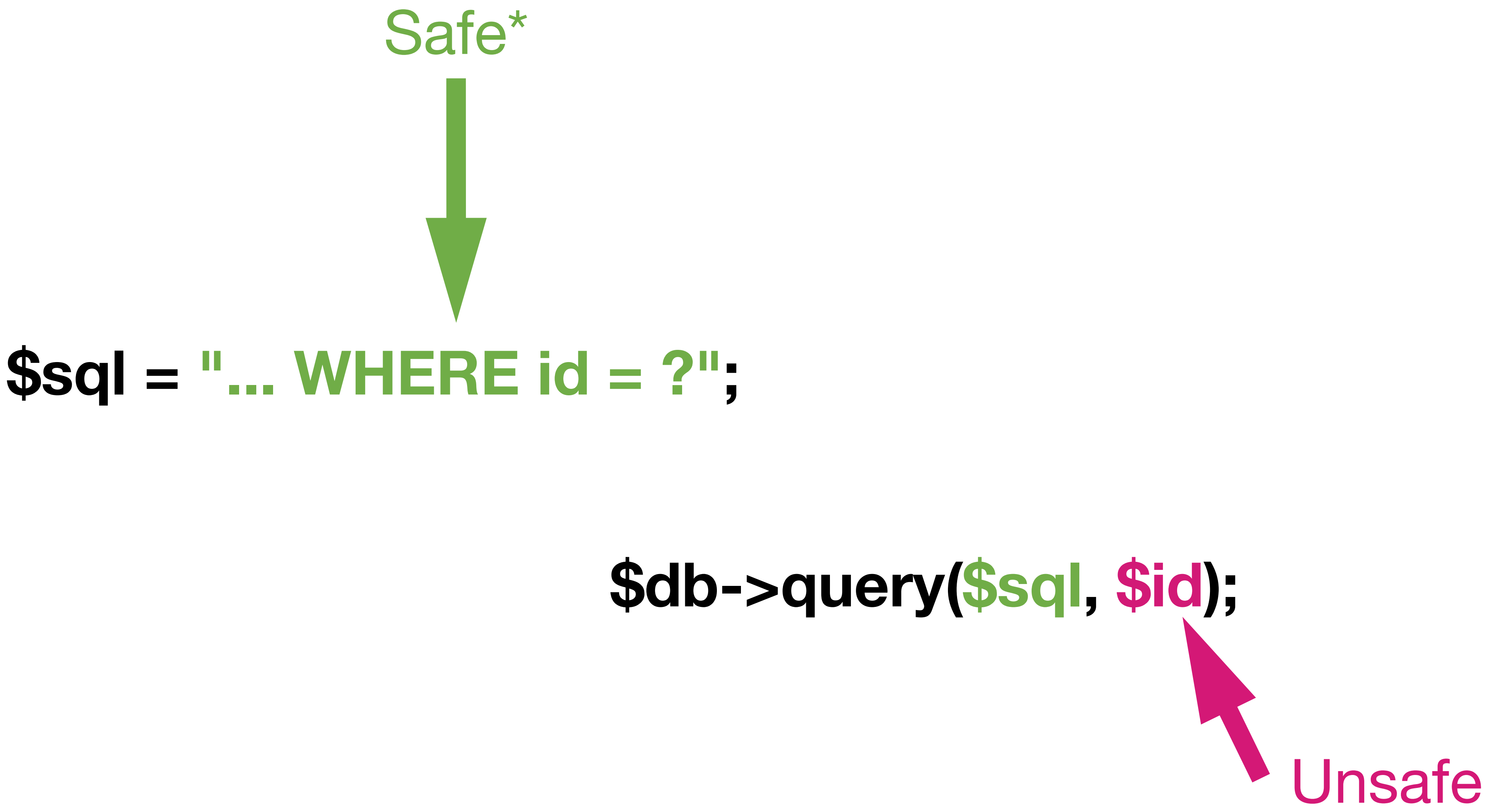
```
$sql = "... WHERE id = " . mysqli_real_escape_string($link, $id);
```

```
$db->query($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

👍

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

Safe*

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

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

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

Unsafe

👍

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

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

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

```
'WHERE id IN (?, ?, ?)'
```

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

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

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

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

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

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

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

List of Allowed fields

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?

If 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]);
```

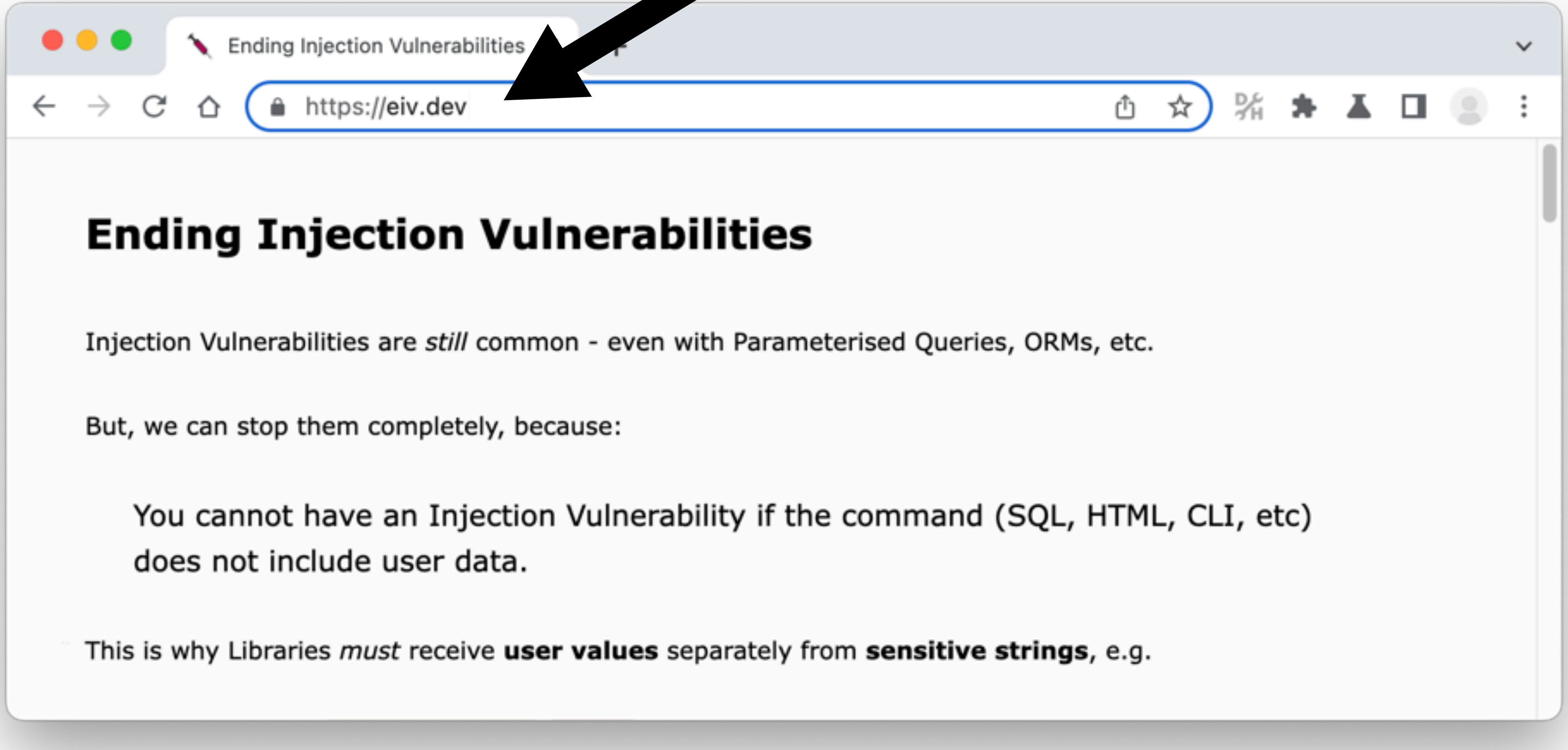
Parameters (as normal)

```
$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



Ending Injection Vulnerabilities

Injection Vulnerabilities are *still* common - even with Parameterised Queries, ORMs, etc.

But, we can stop them completely, because:

> You cannot have an Injection Vulnerability if the command (SQL, HTML, CLI, etc) does not include user data.

This is why Libraries *must* receive **user values** separately from **sensitive strings**, e.g.

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 consteval 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

JavaScript and PHP...

In JavaScript, hopefully one day you will be able to use:

isTemplateObject

or

TrustedHTML.fromLiteral

Thanks to Krzysztof Kotowicz

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

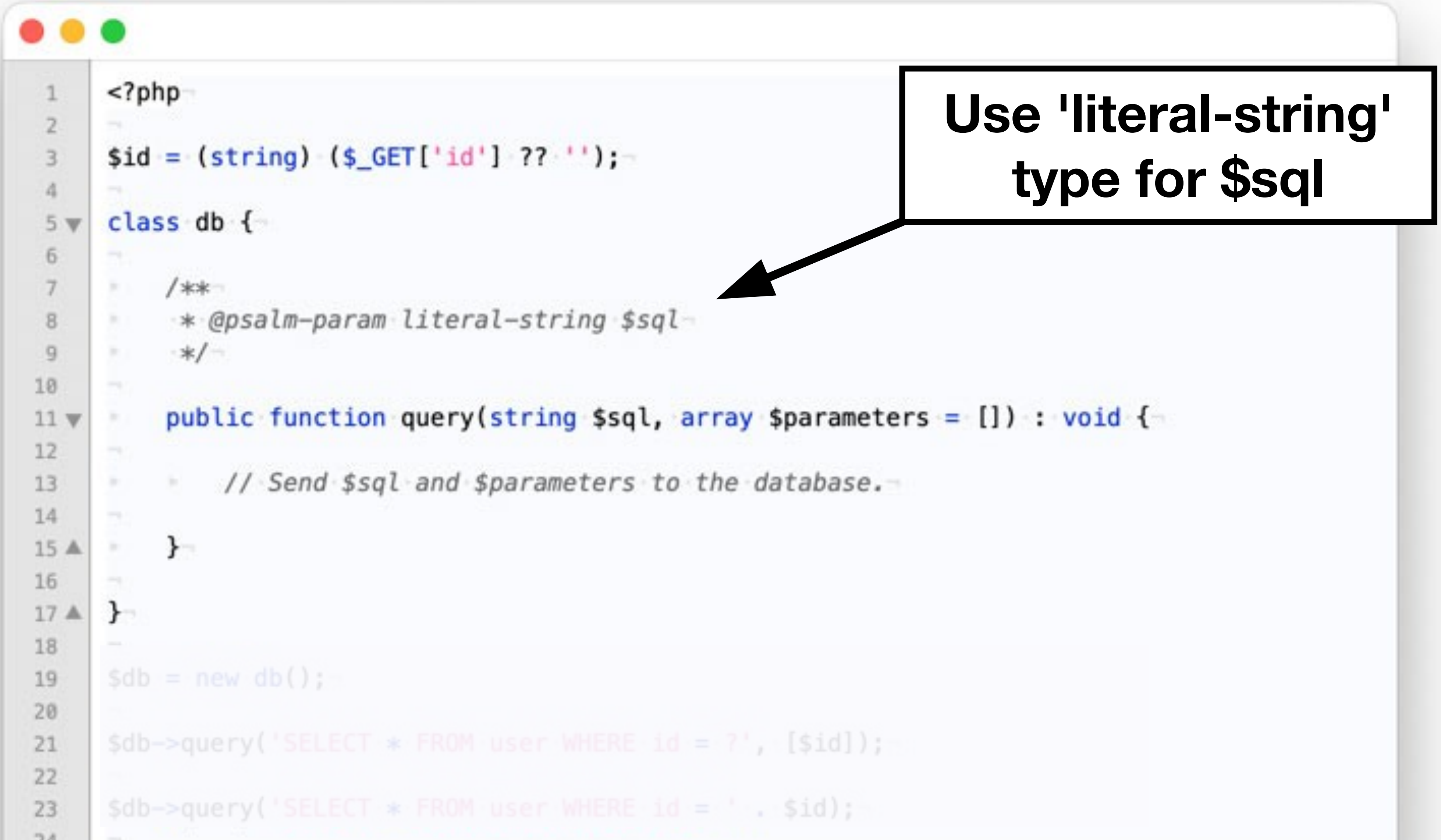
Using Psalm

Thanks to Matthew Brown

```
composer require --dev vimeo/psalm

./vendor/bin/psalm --init
```

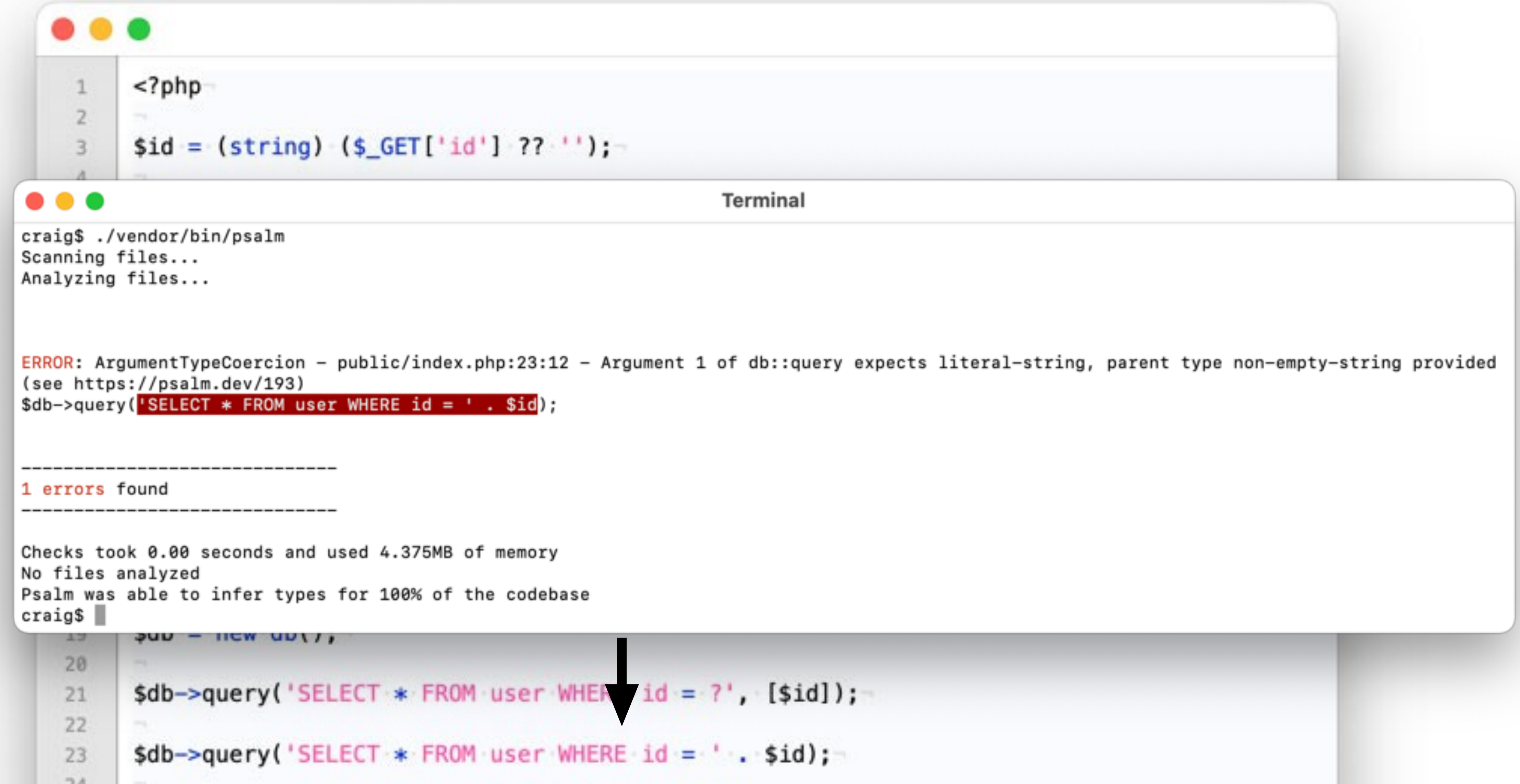
Check Psalm is at level 3 or stricter.

(level 1 is the most strict)



```php
<?php

$id = (string) ($_GET['id'] ?? '');

class db {

    /**
     * @psalm-param literal-string $sql
     */

    public function query(string $sql, array $parameters = []) : void {

        // Send $sql and $parameters to the database.

    }

}

$db = new db();

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

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

Use 'literal-string' type for $sql



```php
<?php

$id = (string) ($_GET['id'] ?? '');
```

```
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$
```

```php
$db = new db();

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

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

Using PHPStan

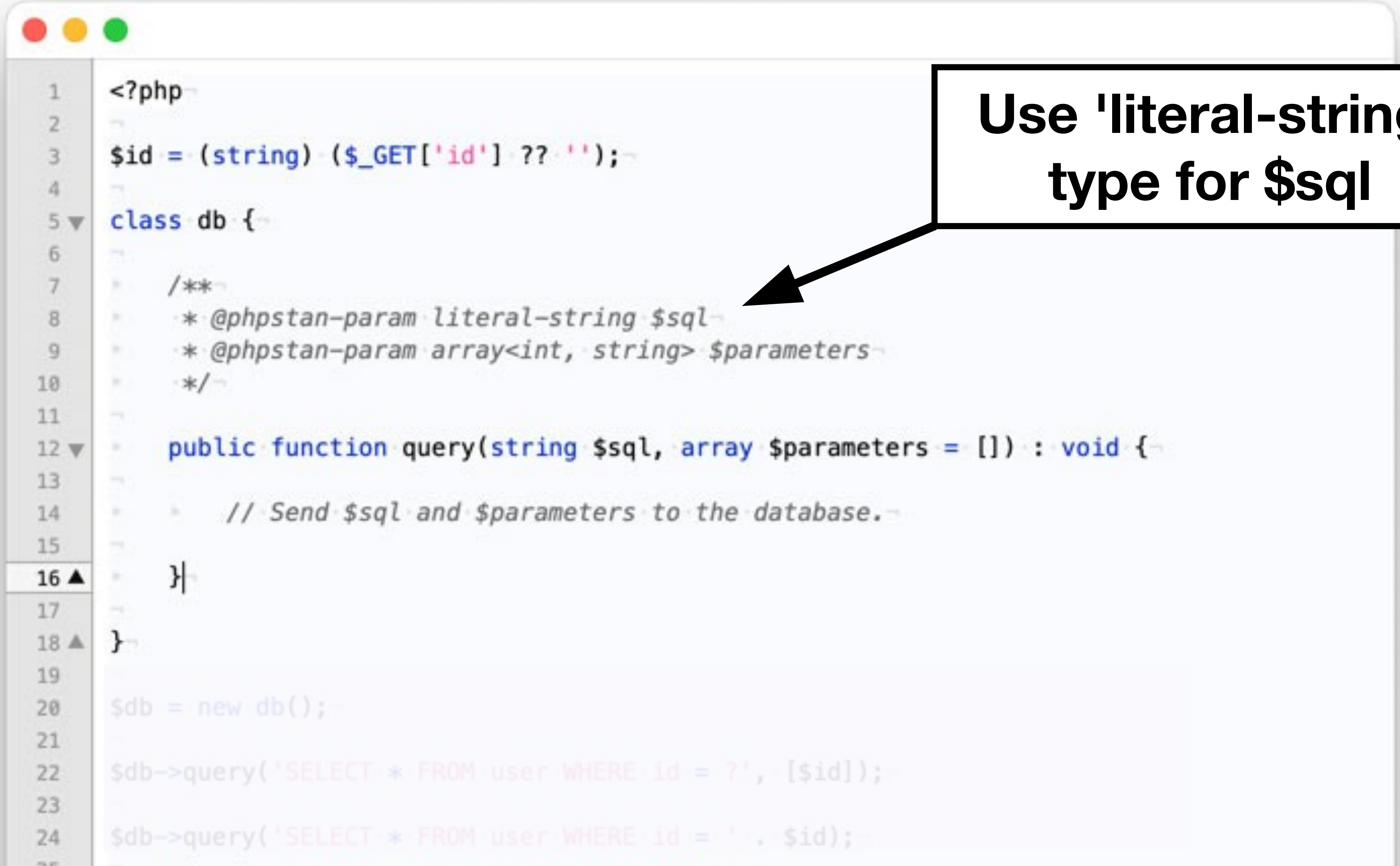
Thanks to Ondřej Mirtes

```
composer require --dev phpstan/phpstan
```

Check PHPStan is at level:

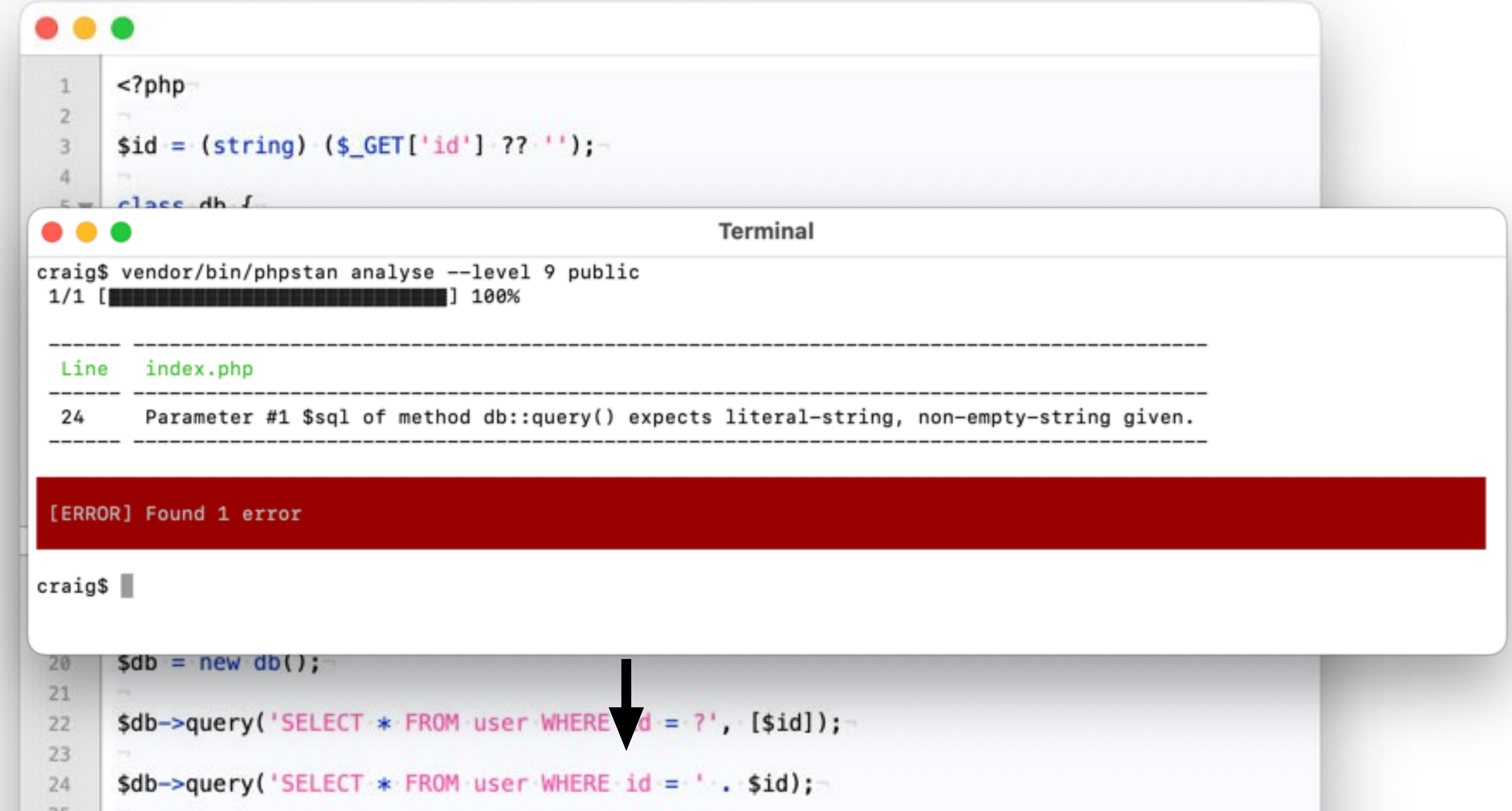
5 or stricter when an argument uses a single type.
7 or stricter when an argument uses multiple types.

(level 9 is the most strict)



```php
<?php

$id = (string) ($_GET['id'] ?? '');

class db {

    /**
     * @phpstan-param literal-string $sql
     * @phpstan-param array<int, string> $parameters
     */

    public function query(string $sql, array $parameters = []) : void {

        // Send $sql and $parameters to the database.

    }

}

$db = new db();

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

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

Use 'literal-string' type for $sql



```php
<?php

$id = (string) ($_GET['id'] ?? '');

class db {
```

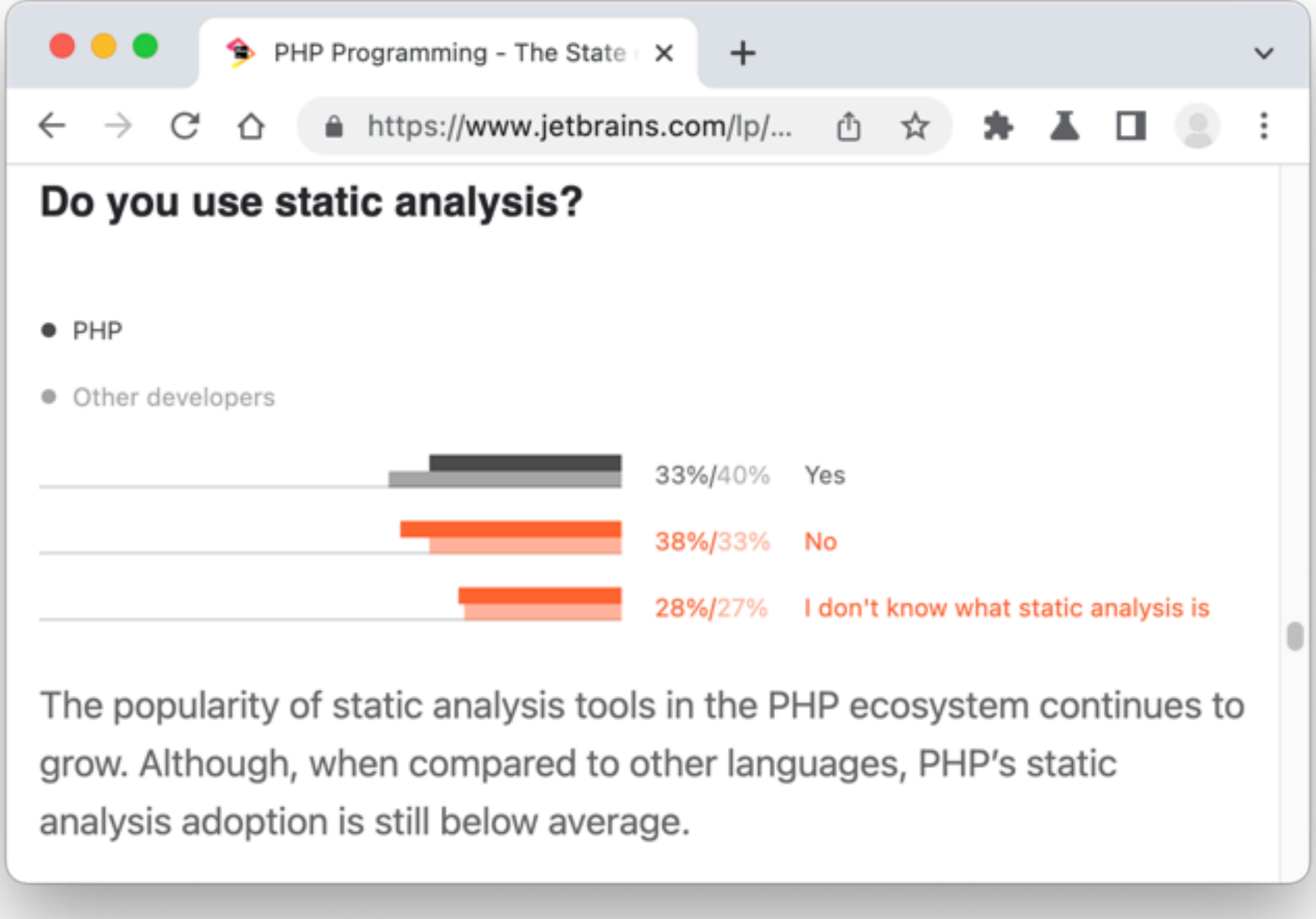
Terminal

```
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$ 
```

```php
$db = new db();

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

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

A Small Problem...

2021



Do you use static analysis?

- ● PHP
- ● Other developers

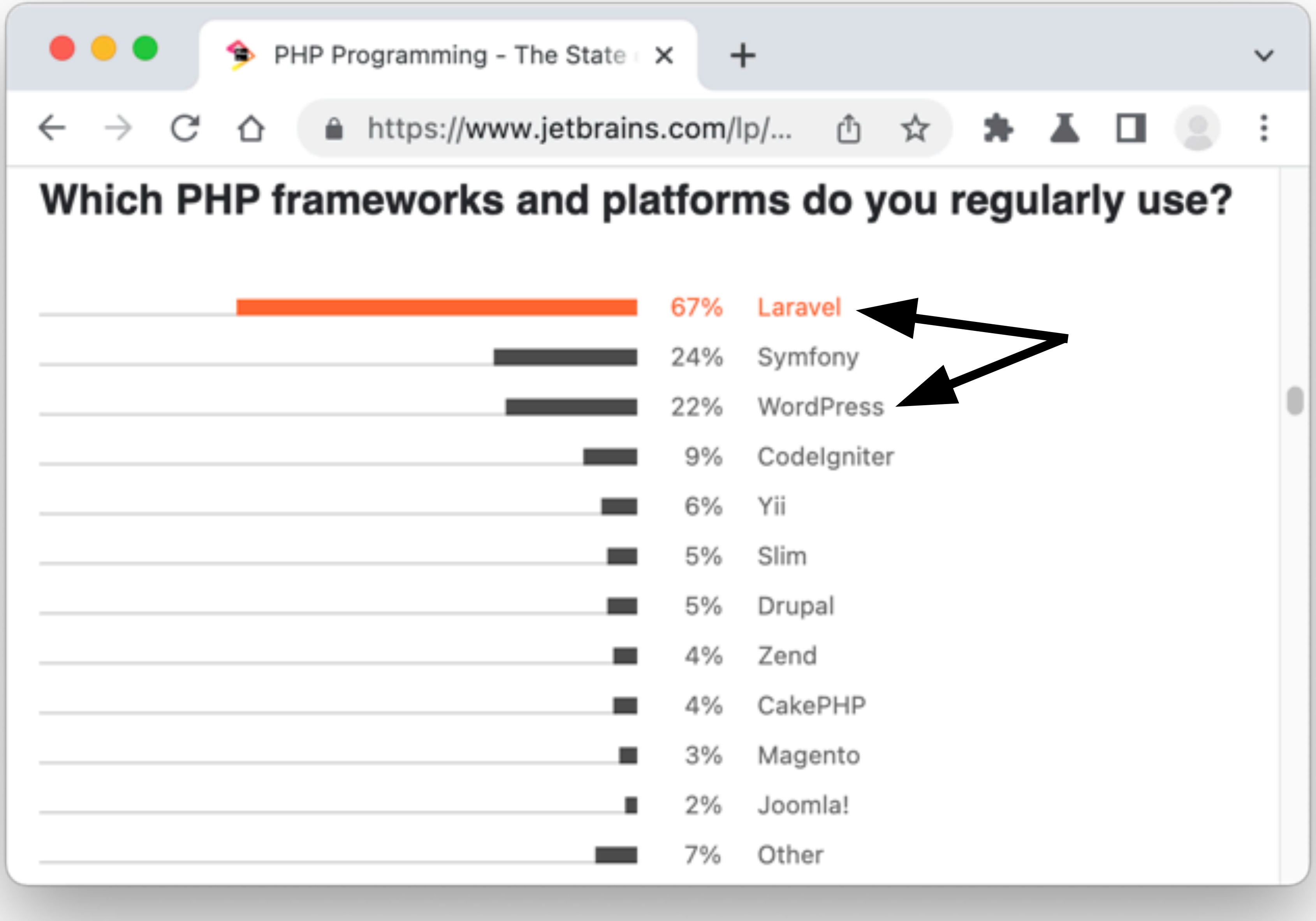
33%/40% Yes

38%/33% No

28%/27% I don't know what static analysis is

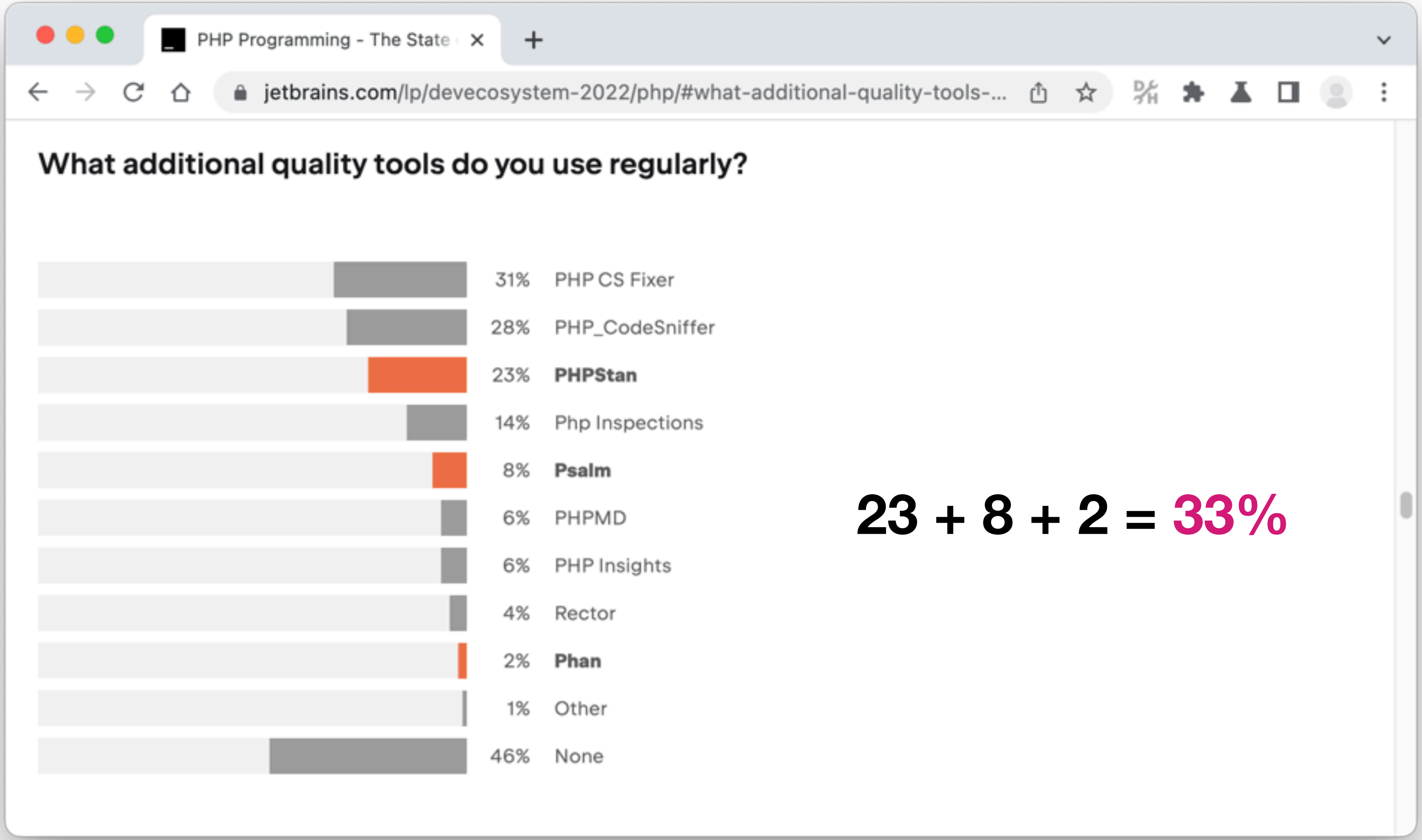
The popularity of static analysis tools in the PHP ecosystem continues to grow. Although, when compared to other languages, PHP's static analysis adoption is still below average.

2021



Which PHP frameworks and platforms do you regularly use?

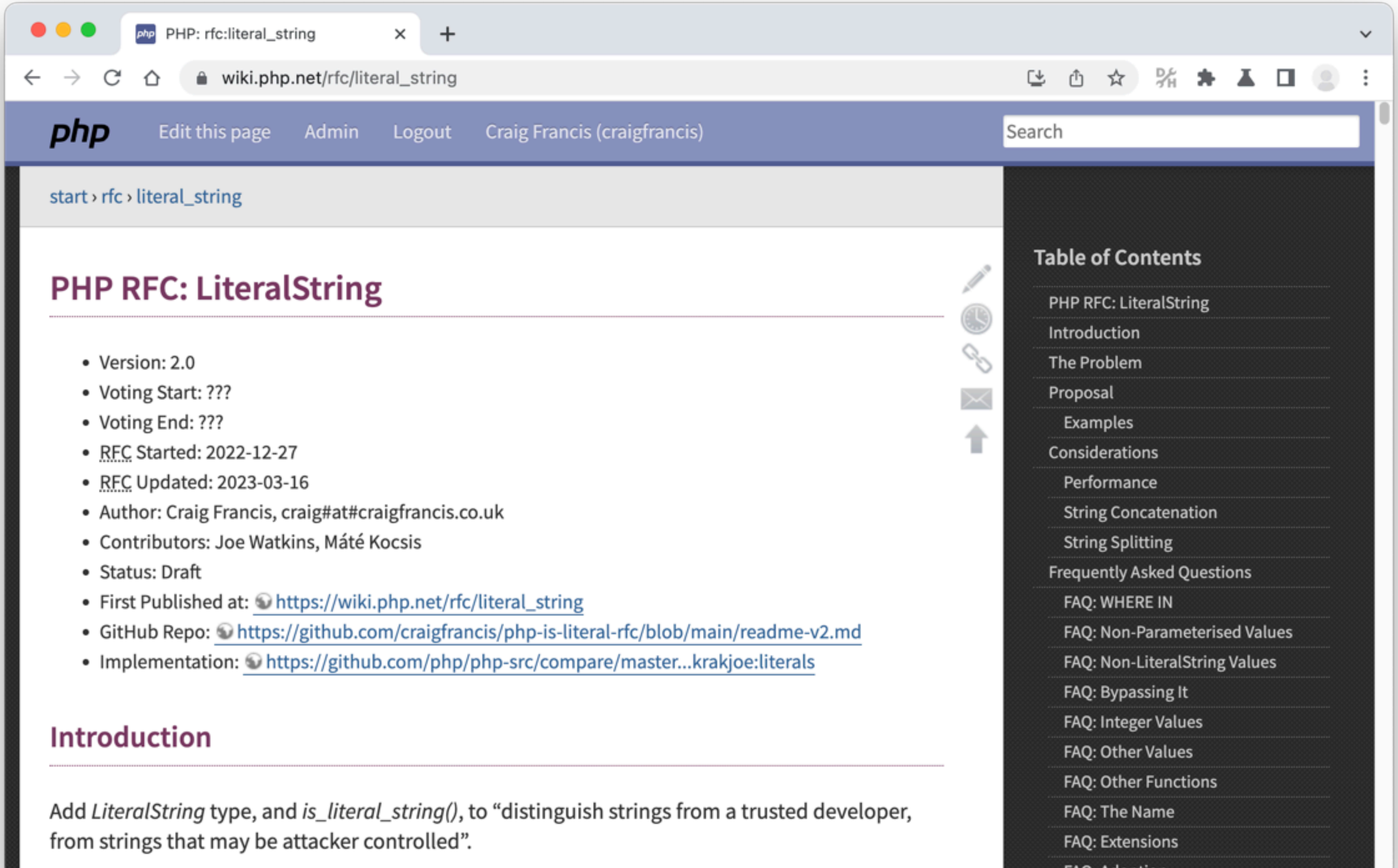
2022


PHP Programming - The State
jetbrains.com/lp/devecosystem-2022/php/#what-additional-quality-tools-...

What additional quality tools do you use regularly?

23 + 8 + 2 = 33%

Make it a key part of the Programming Language



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

"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