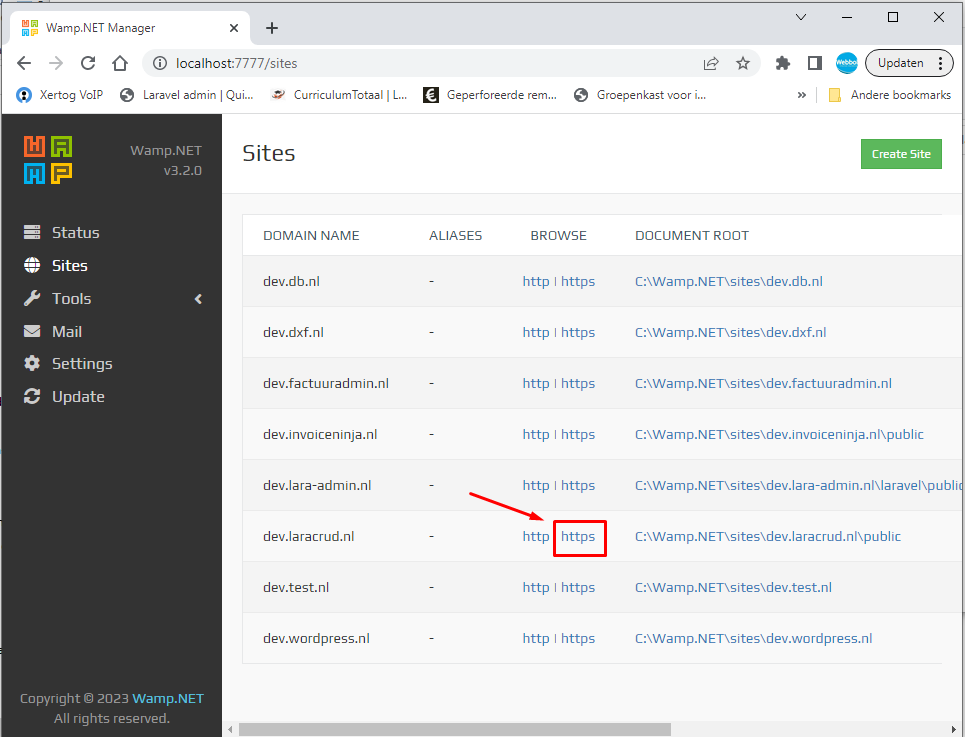
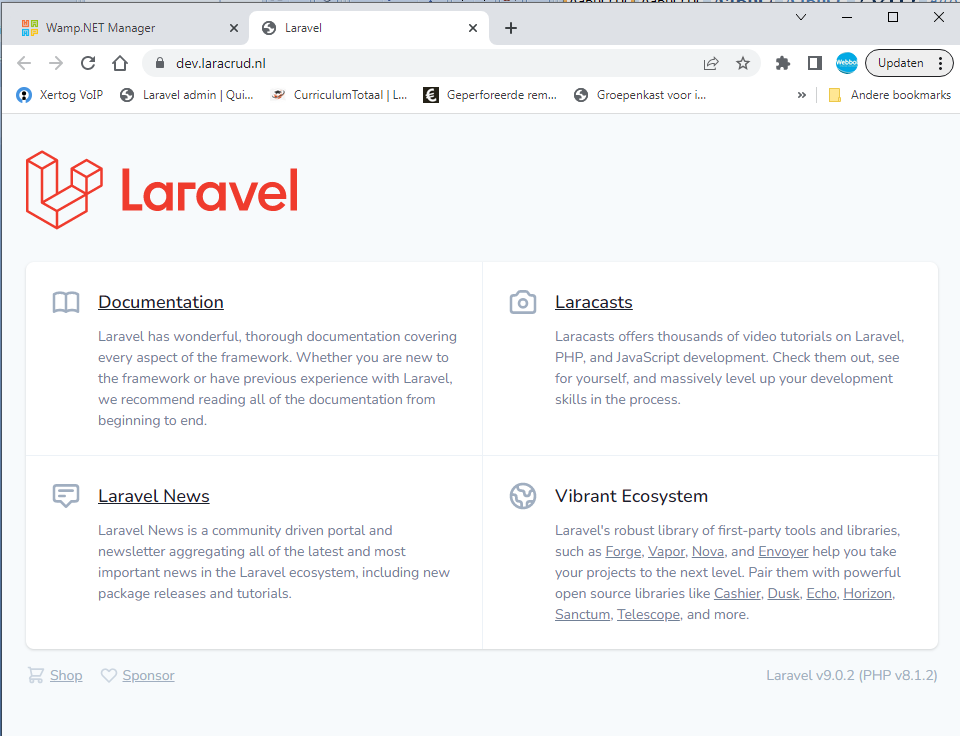
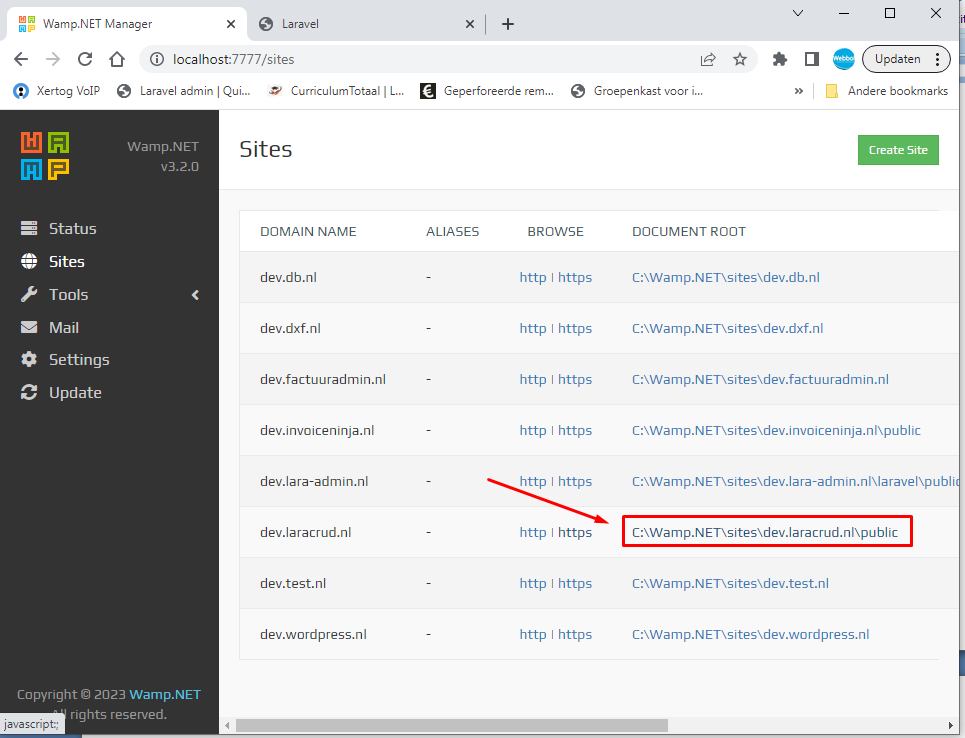
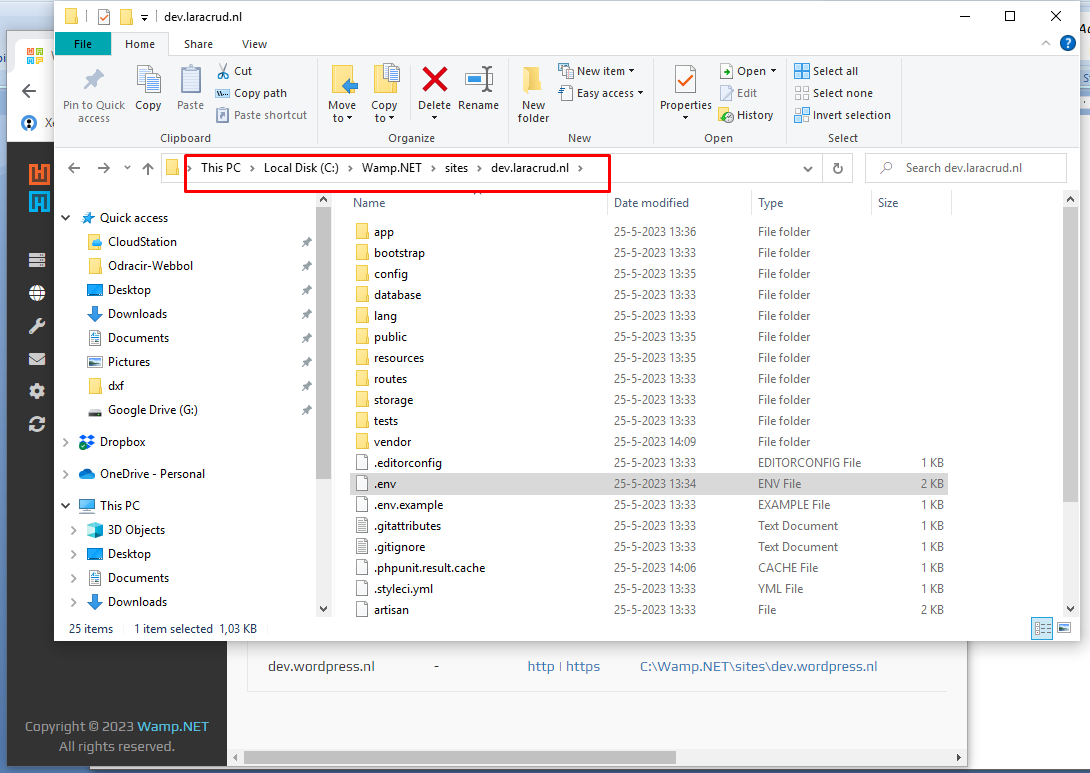
DEBUG BAR INSTALLEREN



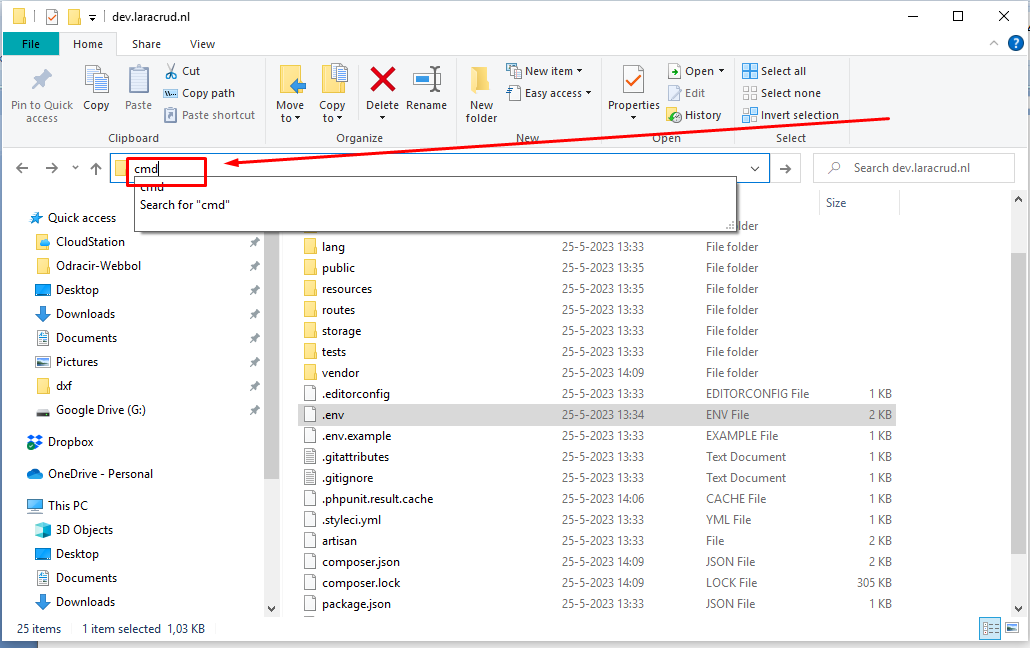


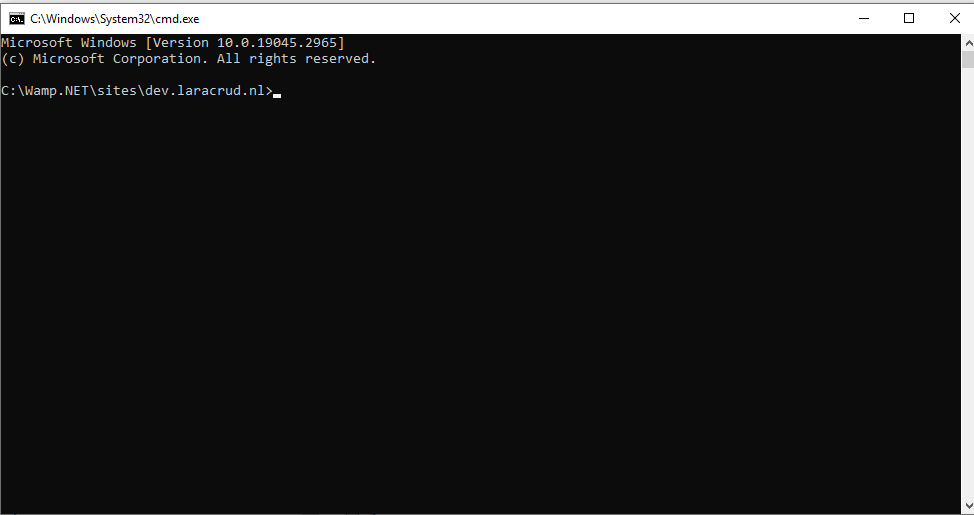


Ga 1 map hoger, dus niet in public map kijken



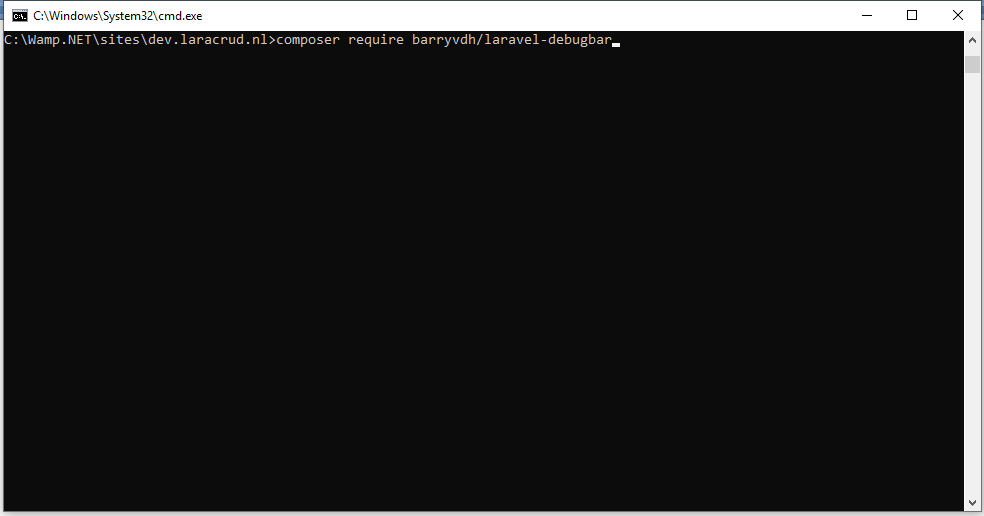
Typ hier “cmd” in en druk op enter

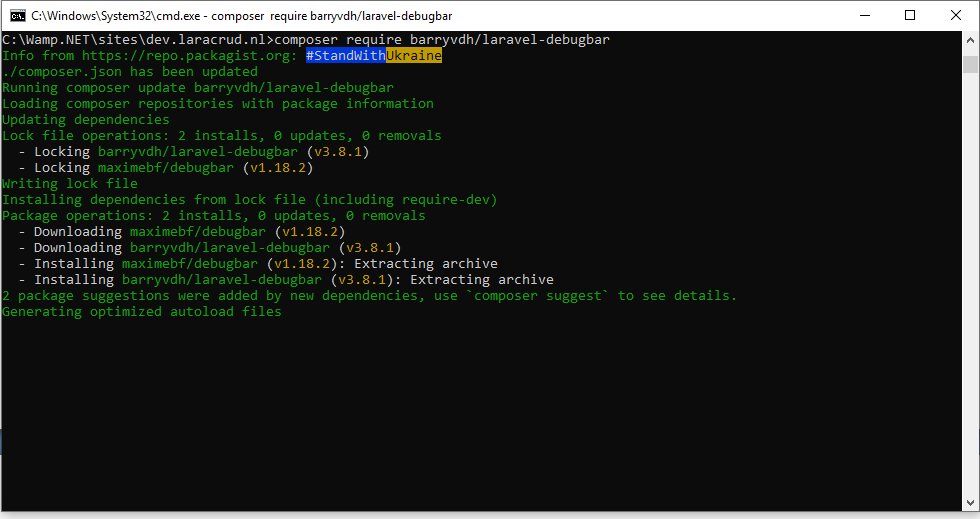


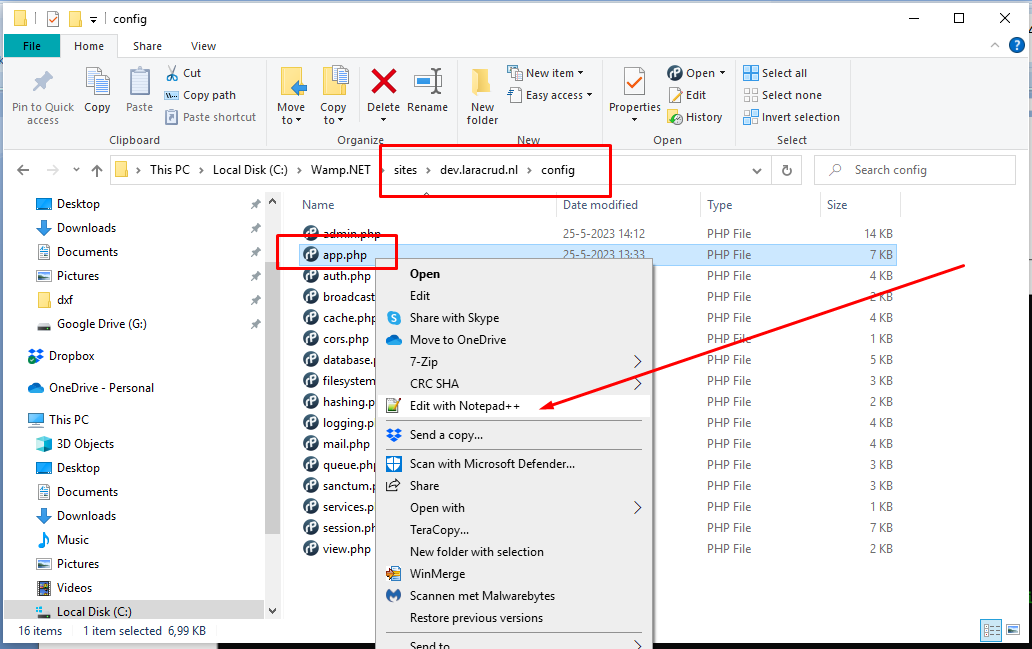


Type in commando:

composer **require** barryvdh/laravel-debugbar

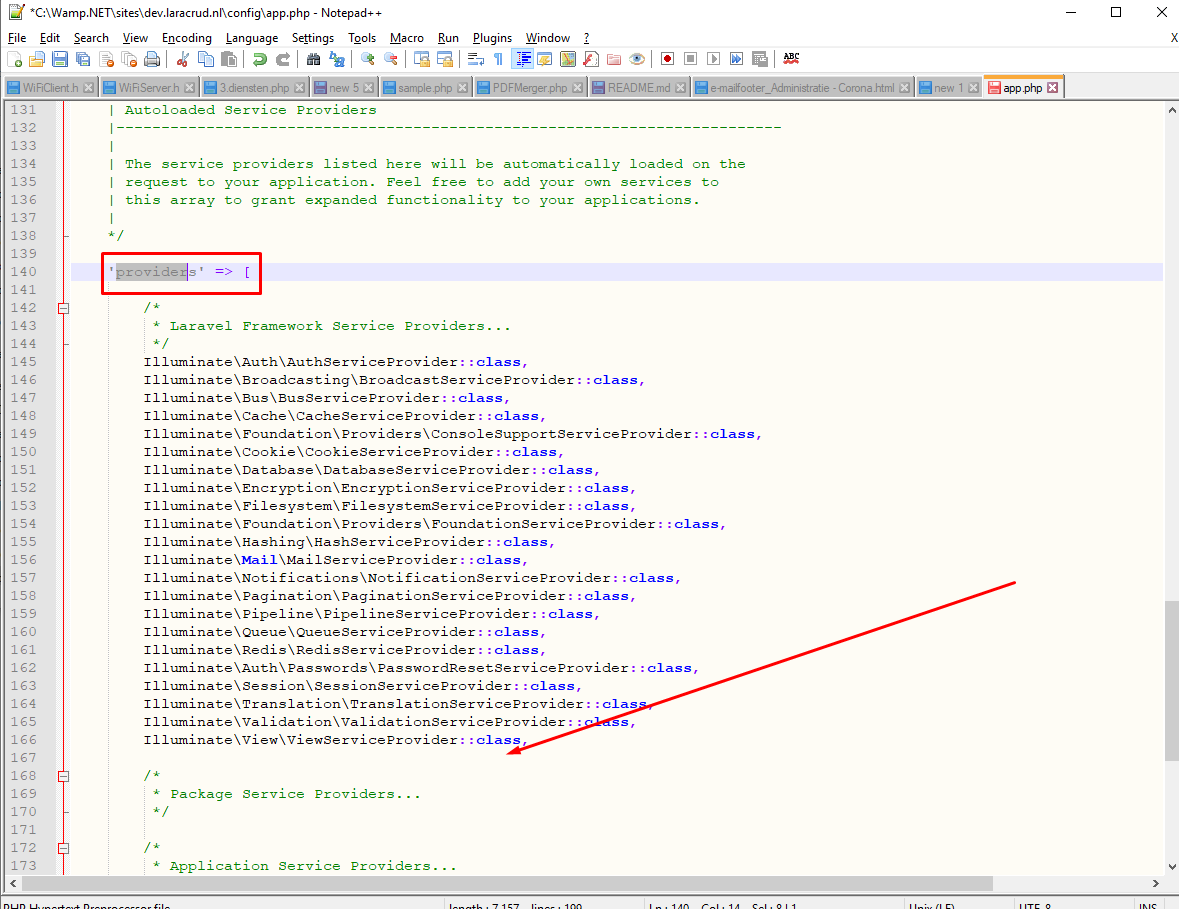


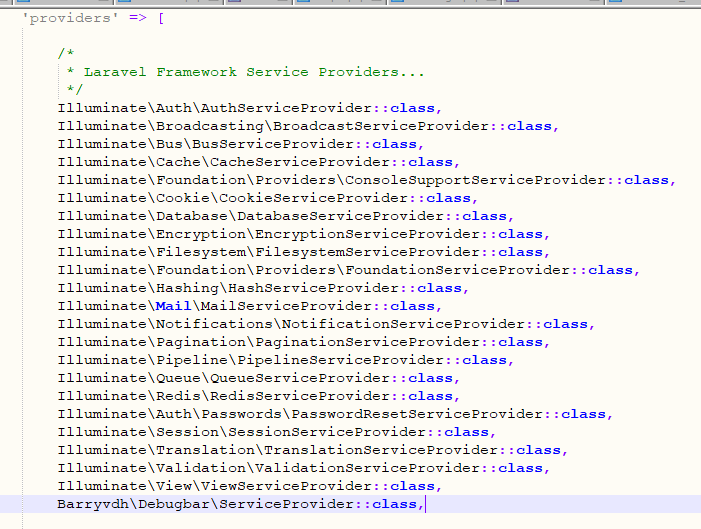


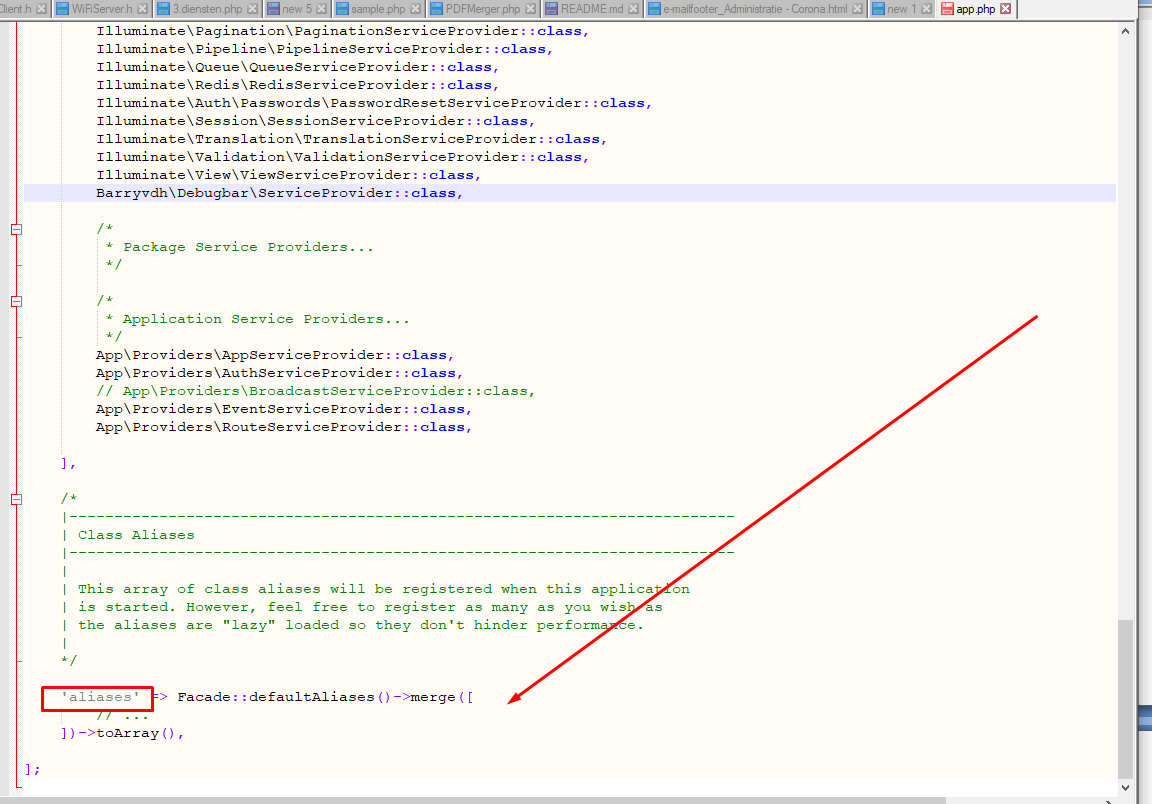


Voeg hieronder toe

'Barryvdh\Debugbar\ServiceProvider'

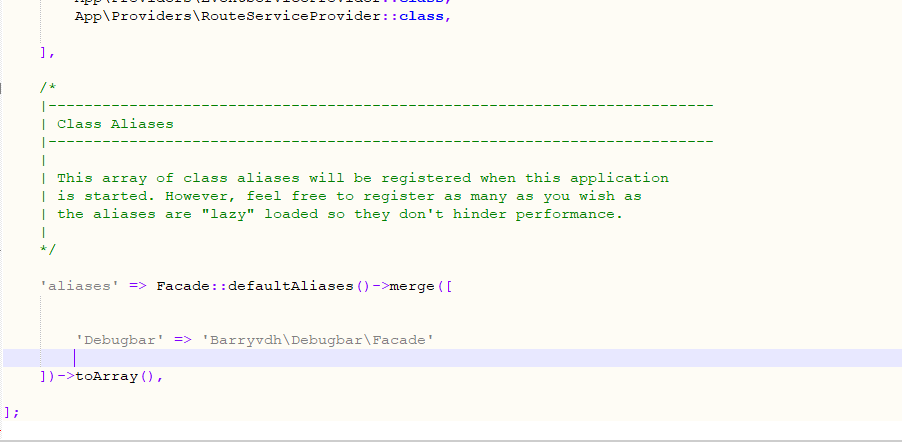


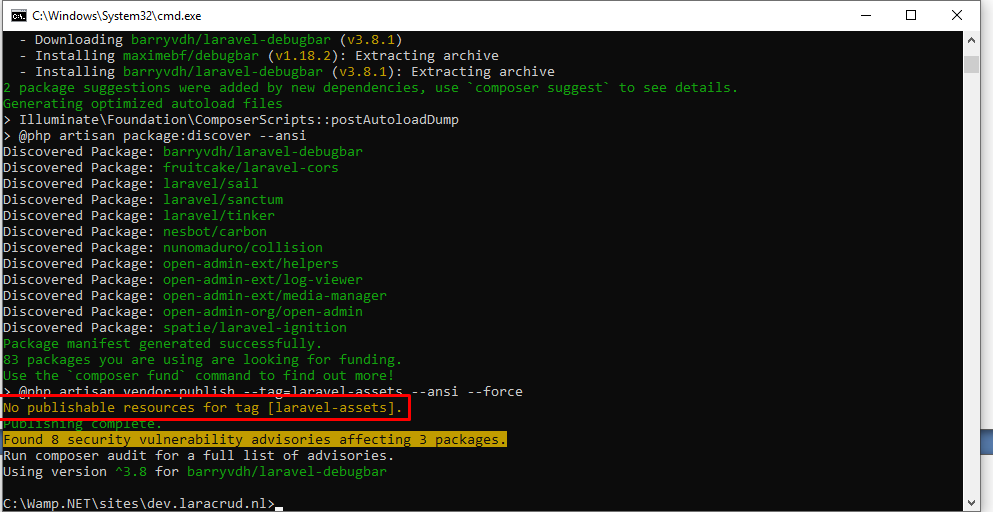




Voeg hier aan toe:

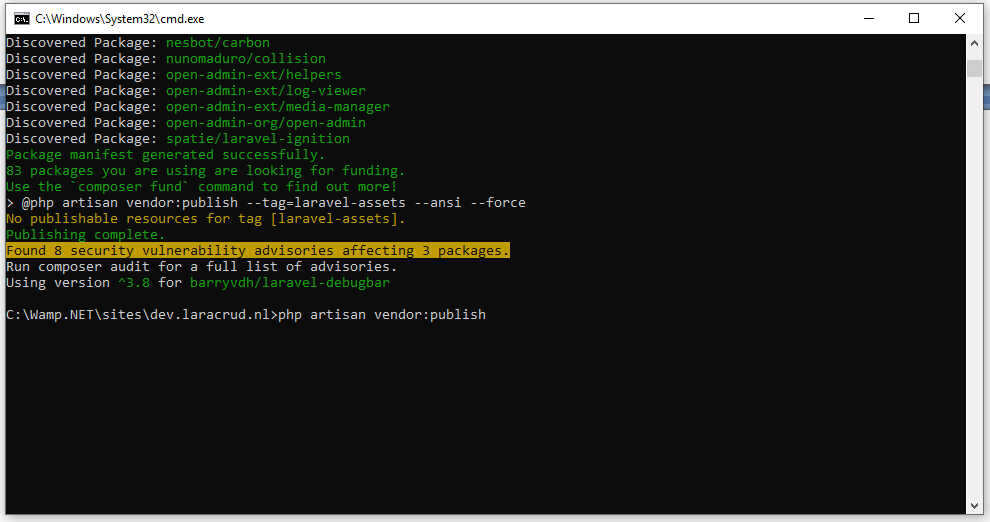
'Debugbar' => 'Barryvdh\Debugbar\Facade'

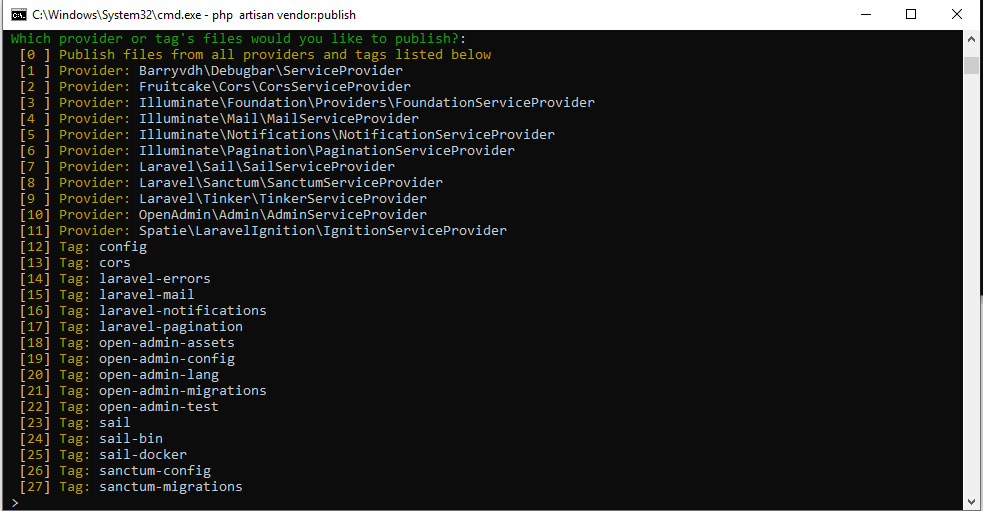




Voer in commando:

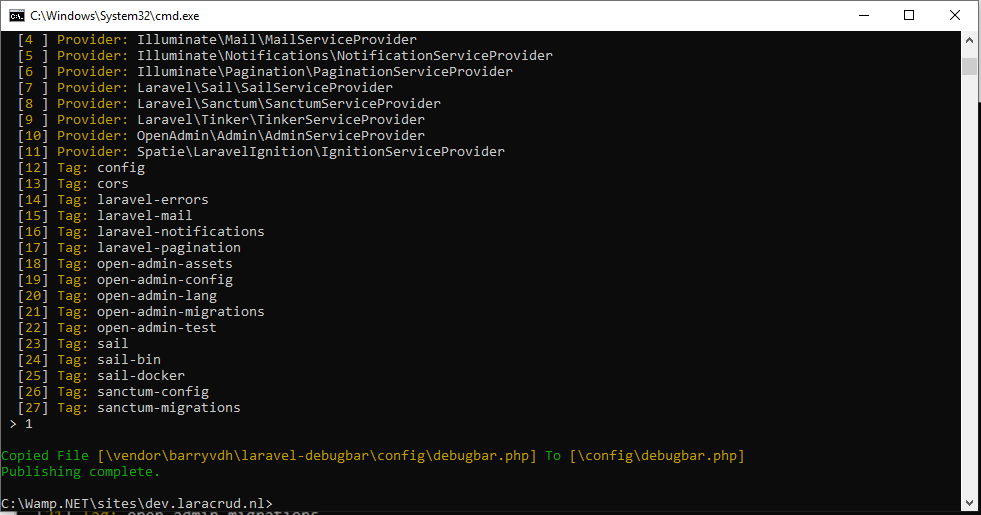
php artisan vendor:publish



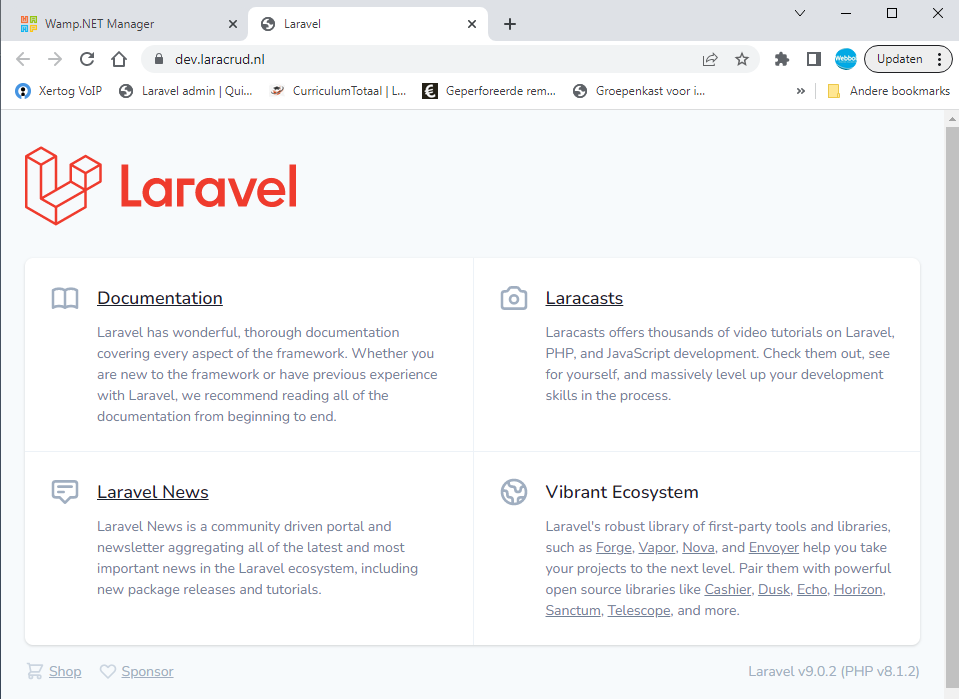


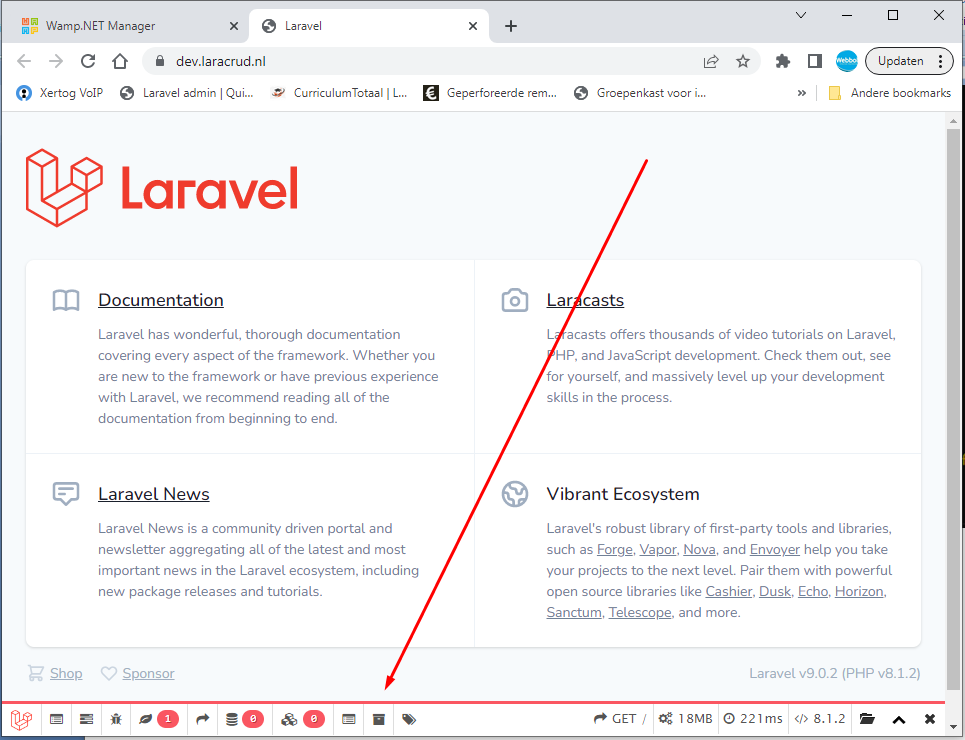
Type in: 1

<Enter>

****

**Refresh je laravel installatie**

****

****

**Hoe kunnen we dit in de code gebruiken?**

**We kunnen SQL queries bekijken, het geheugenstack, etc…**

**Alles van debug en optimalisaties kun je hiermee snel bereiken.**

**We gaan per stap in hoe we hier handig gebruik van kunnen maken.**

**Test 1: Iets loggen**

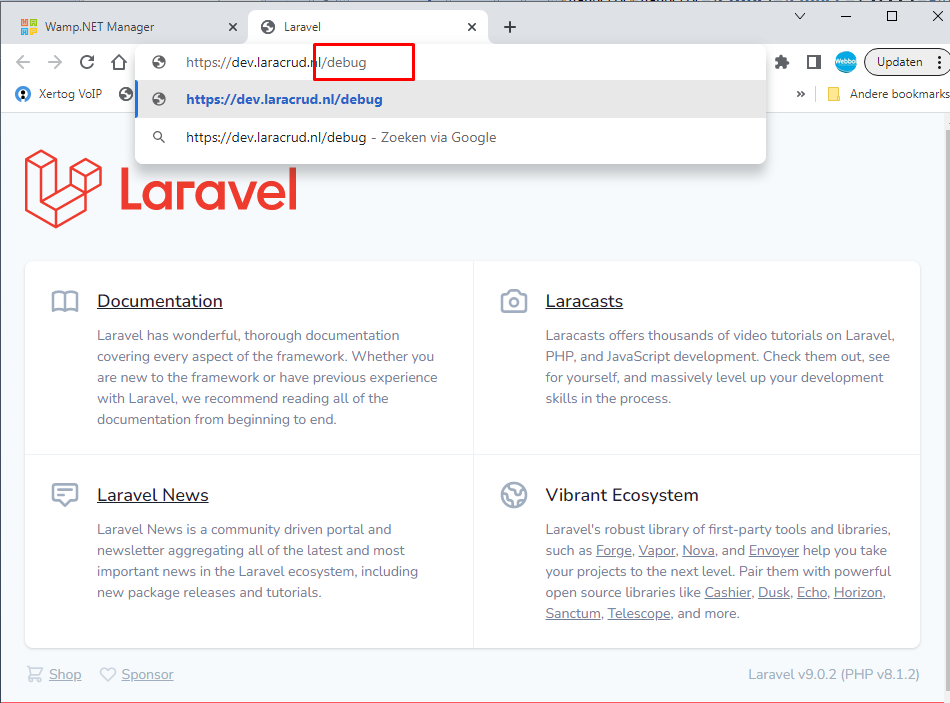
**Je programmeert iets met wil op een bepaalde punt iets naar de debugbar sturen.**

**Hoe doen we dat?**

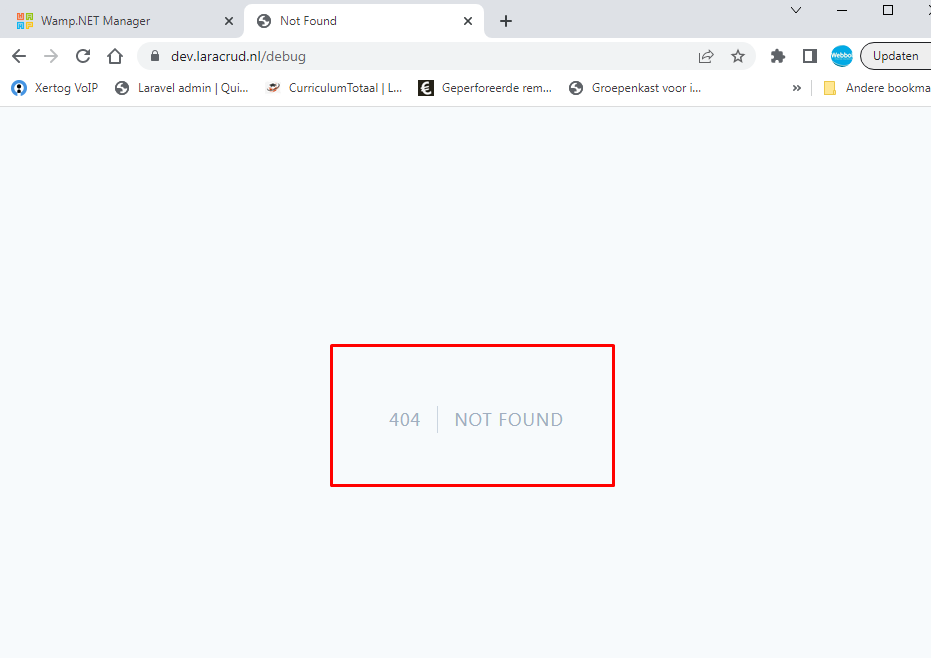
**We maken een testroute aan, iets heel simpels, doet nog niets spannends maar je begrijpt het idee.**

**Open web.php en we maken een publieke route: “/debug” aan**

**Dus: in je browser:**

****

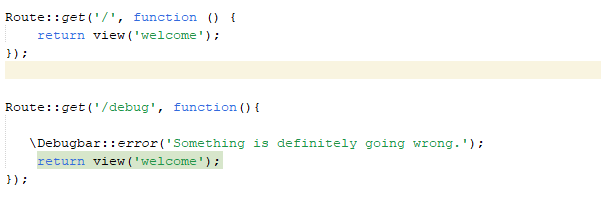
**De route werkt nog niet, kijk maar**

****

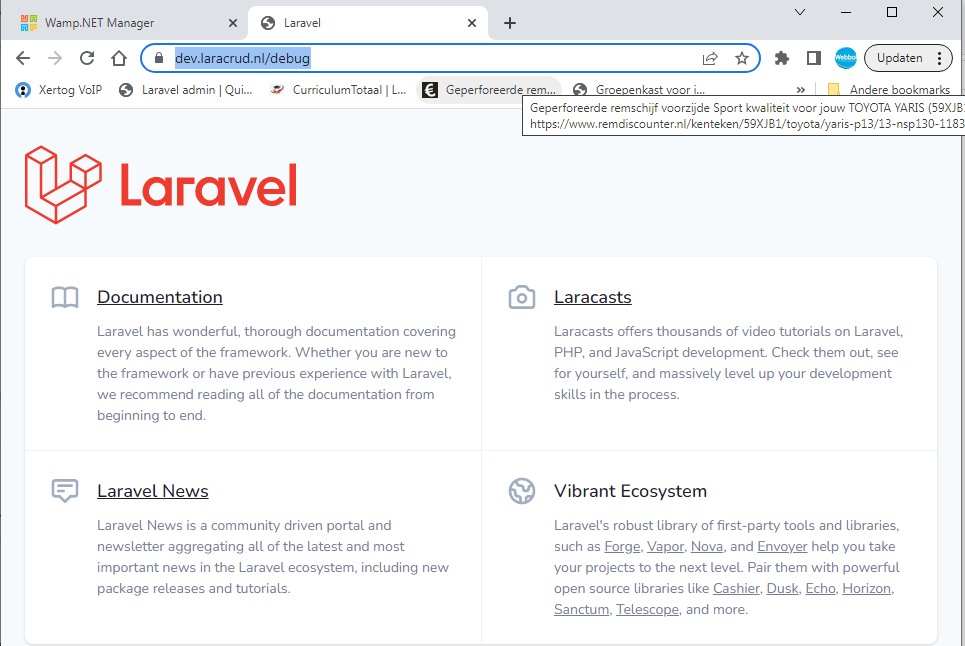
**Web.php**

****

**Vul de route aan met de volgende code:**

****

**Ga naar je browser met je Laravel installatie en bekijk de route “/debug”**

****

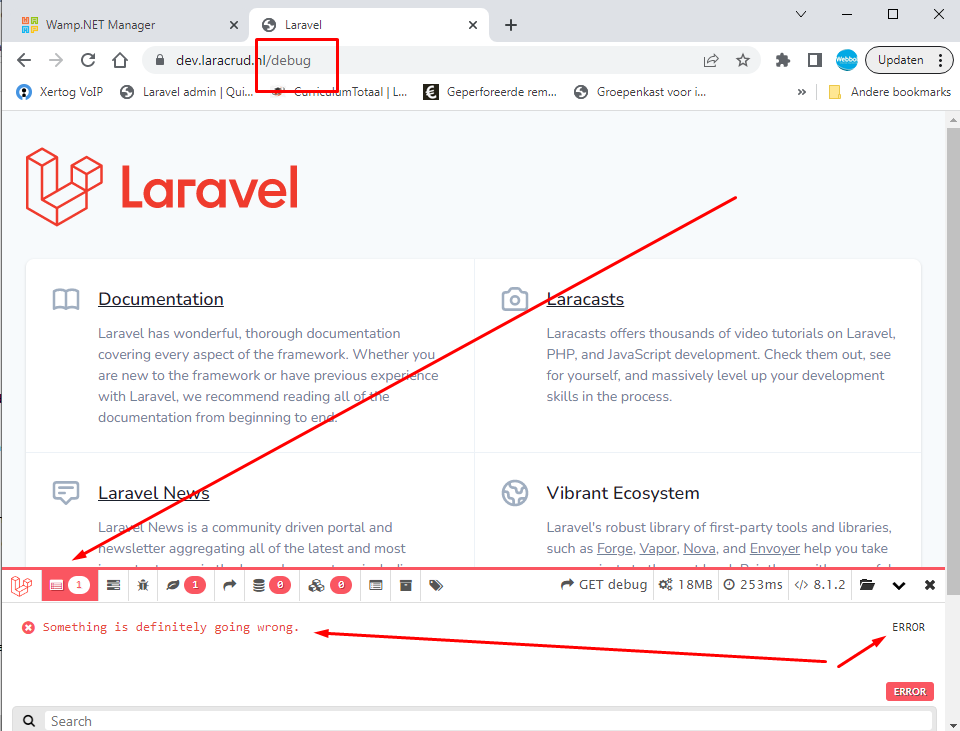
**Zie hier je foutmelding.**

**Cool he….**

**Je kan dus meerdere foutmeldingen ‘dumpen’ zodat je precies kan volgen waar het mis gaat.**

**We hoeven dus geen ‘var\_dump’ of echo in je code te gebruiken (ik raad je dit af!).**

**We gaan steeds meer ‘schone’ code schrijven die we eigenlijk met wat testroutine en een debugbar (waar je nu mee bezig bent) naar wens kunt schrijven.**

****

**We kunnen gebruik maken van kleuren.**

**Om het beter te zeggen, we kunnen een onderscheid maken in een type melding.**

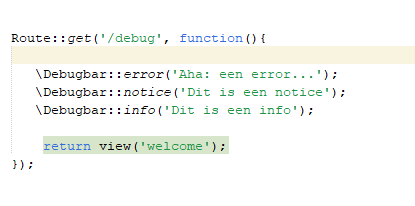
**Niet alles is persé een fout. Soms willen we informatief iets laten zien.**

**Welke soorten meldingen kennen we in basis:**

* **Error**
* **Notice**
* **Info**

**Hoe kunnen we dit gebruiken in onze code?**

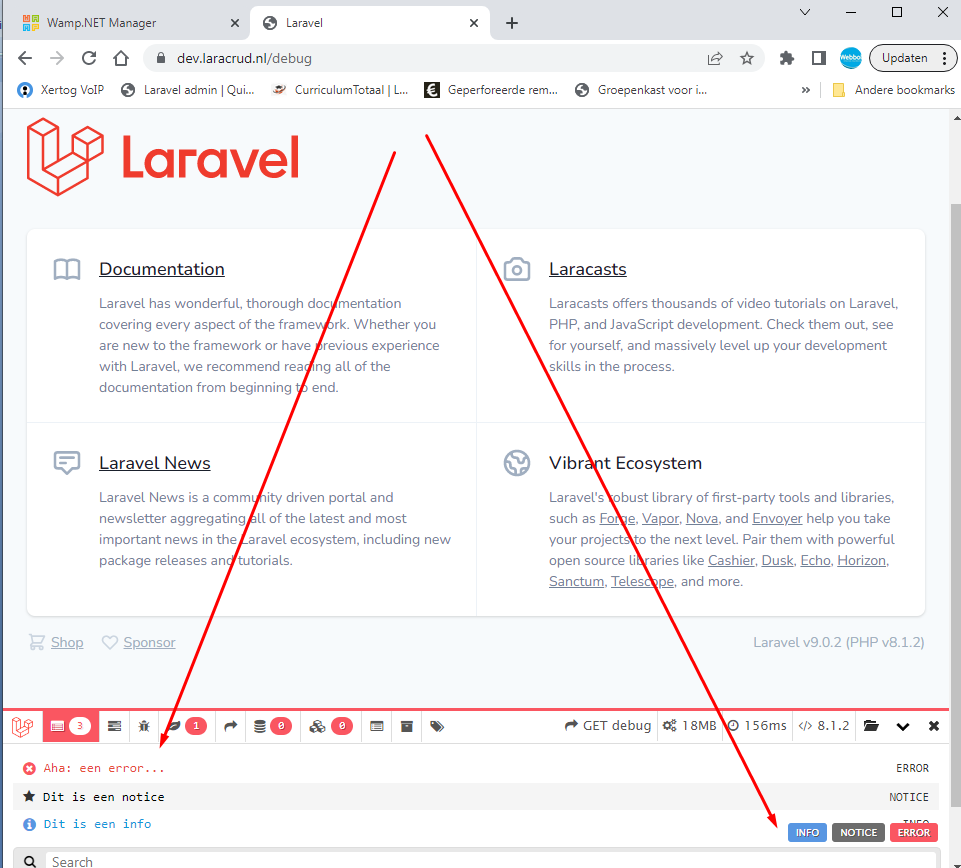
**We pakken voor het gemak dezelfde ‘route’ en dat is “/debug” en voegen hier aan toe:**

****

**Refresh in je browser je scherm:**

**Cool he…**

**Je kan dus vanaf rechts filteren op type melding.**

****

**SQL bekijken**

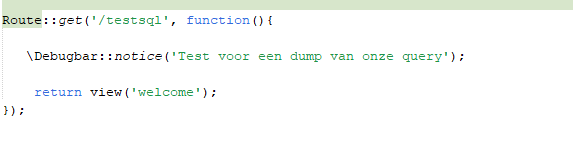
**Hoe kunnen we een SQL bekijken van een query die Laravel via de database engine ‘Eloquent’ maakt?**

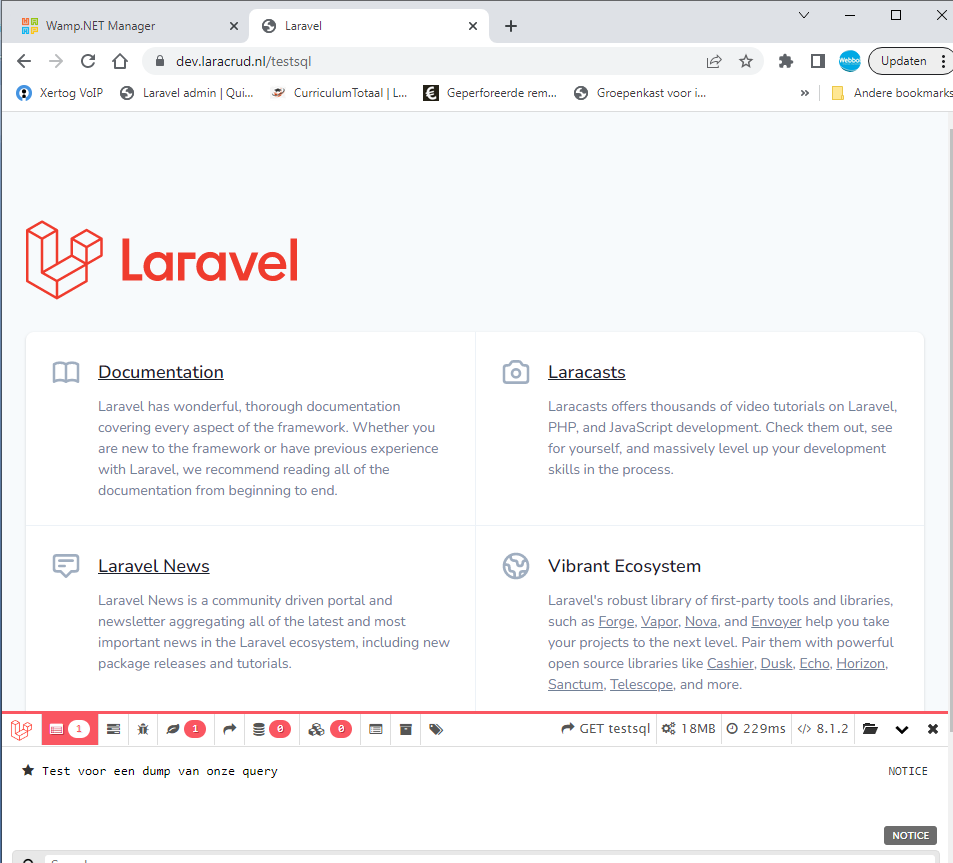
**Binnen Laravel zul je zien dat je eigenlijk nooit meer queries gaat schrijven op de ouderwetse manier “SELECT \* from… bla bla bla bla bl join bla bla ”**

**Nee! We gaan het op een makkelijkere manier uitvoeren en met code die ons veel meer zegt wat er binnen komt of gebeurt in je routine.**

**We gaan nog een testroute aanmaken:**

**Ja dat klopt.. in je web.php gaan we een “/testsql” aanmaken**

****

****

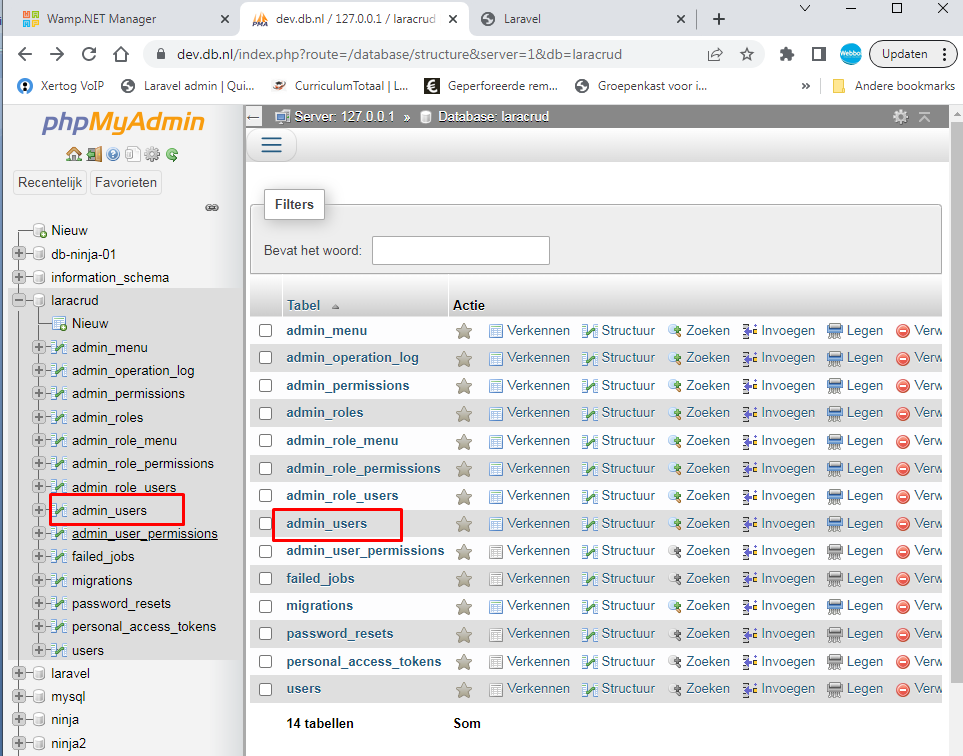
**Een opmerking hier: we hebben de view nodig om het gedeelte van het debugbar zichtbaar te maken.**

**Voor nu gewoon uitvoeren en kijken we later waarom en hoe.**

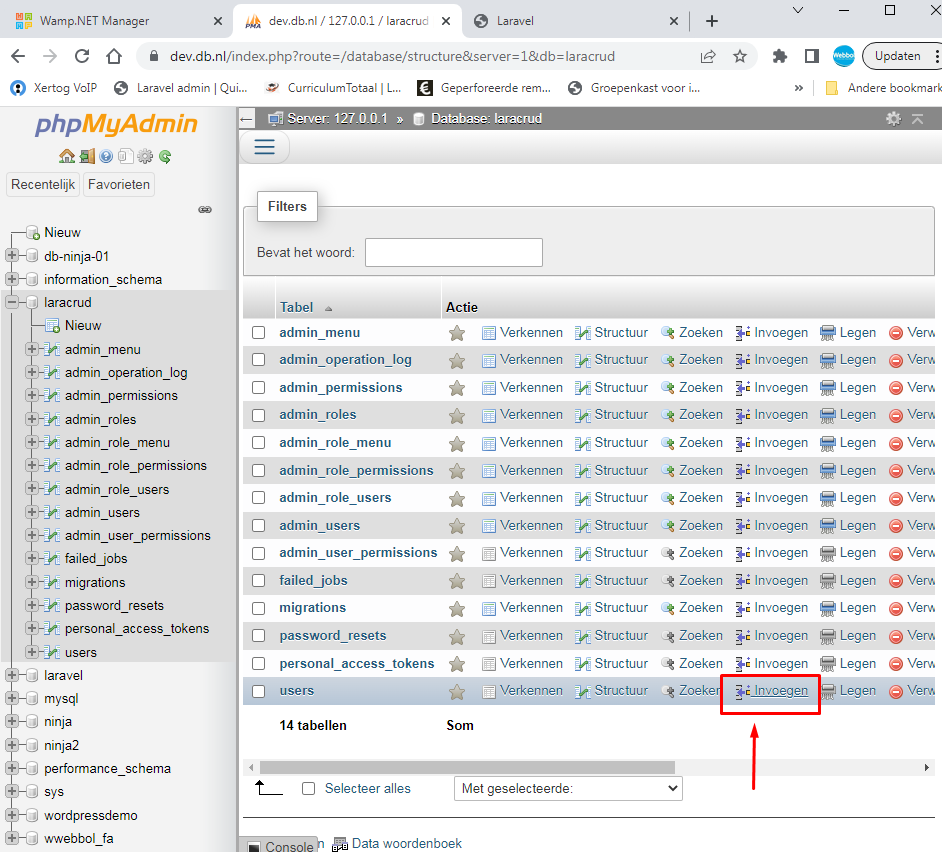
**Enja, hier zit nog geen enkel SQL in om iets met een database**

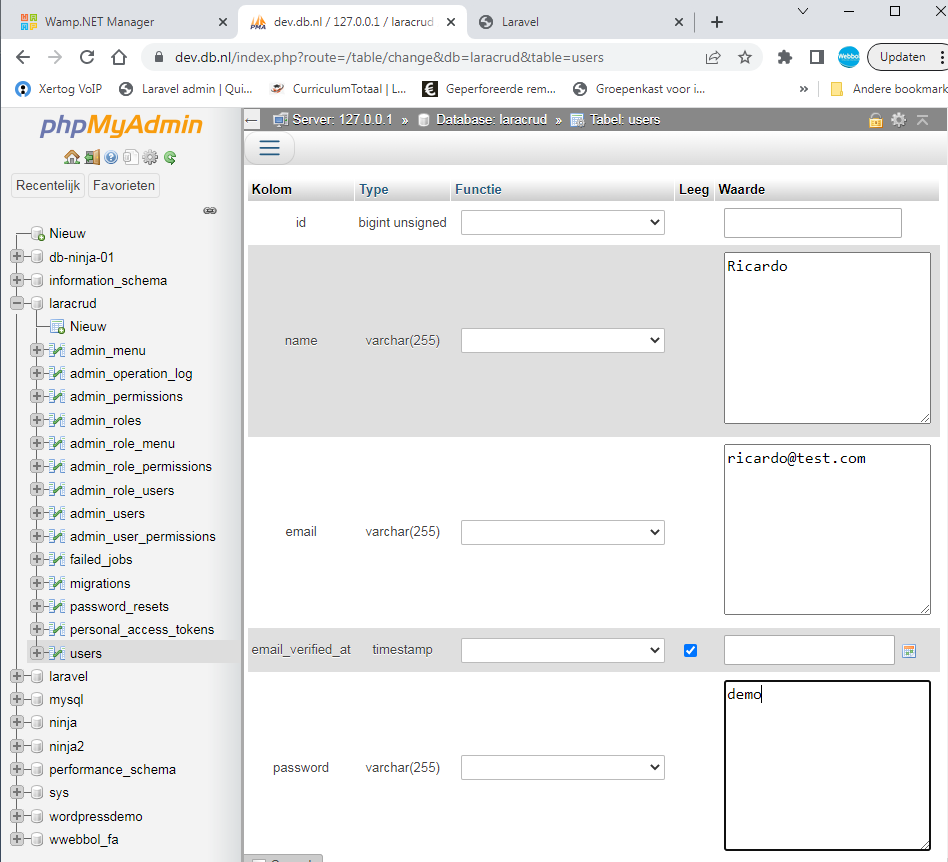
**Kijk in je PhpMyAdmin en voor onze test gaat het er om dat we iets uit onze database ‘pakken’.**

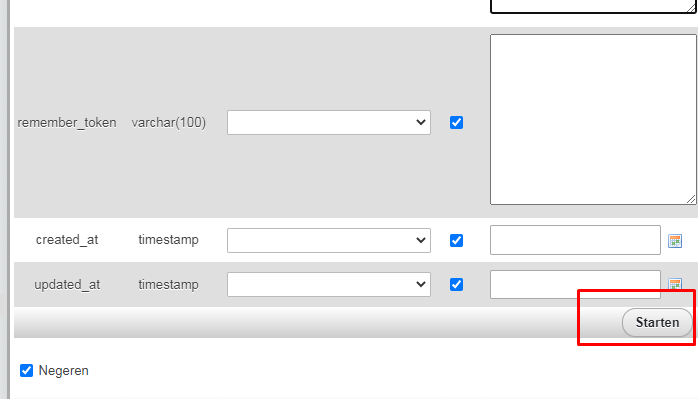
**We gebruiken tabel User voor**

****

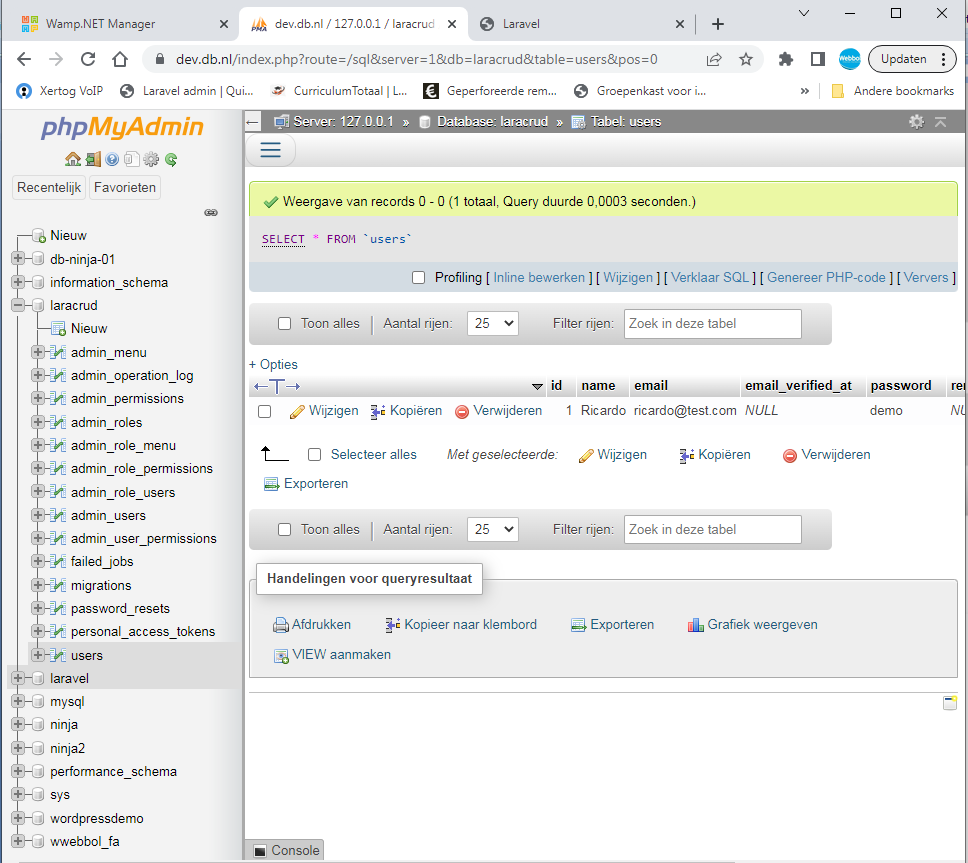
**Insert 1 testrow**

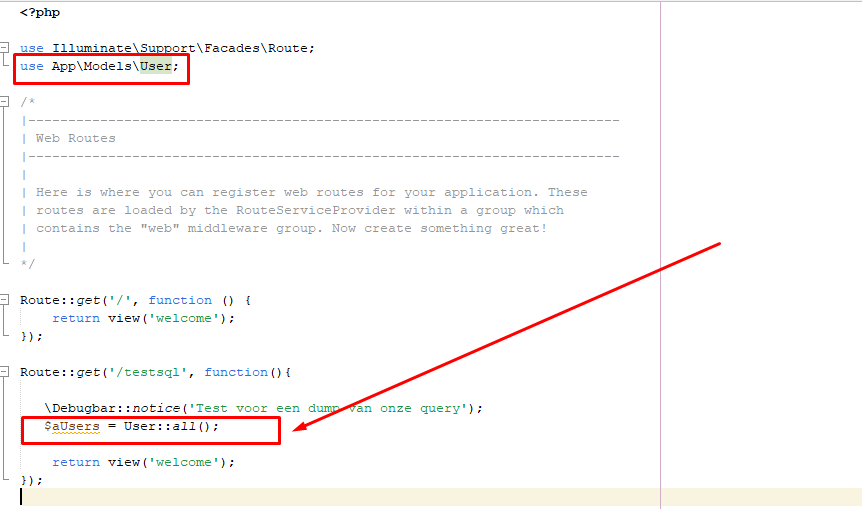
****

****

****

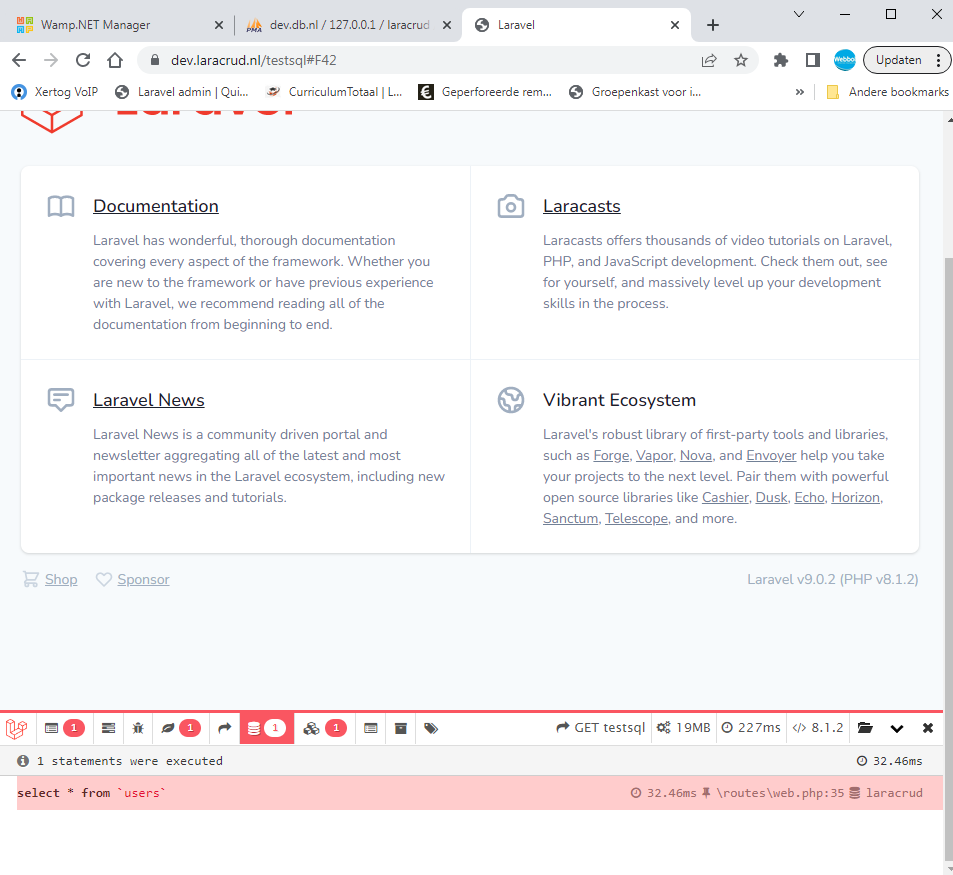
**We hebben een testrecord ☺**

****

****

**Refresh je laravel installatie:**

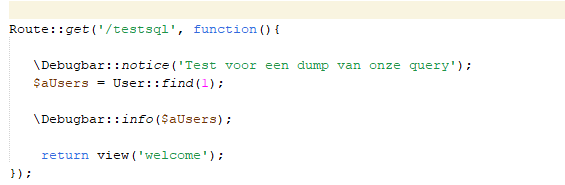
**Zie hier je SQL query**

****

**En hoe kunnen we een sql resultaat ‘debuggen’?**

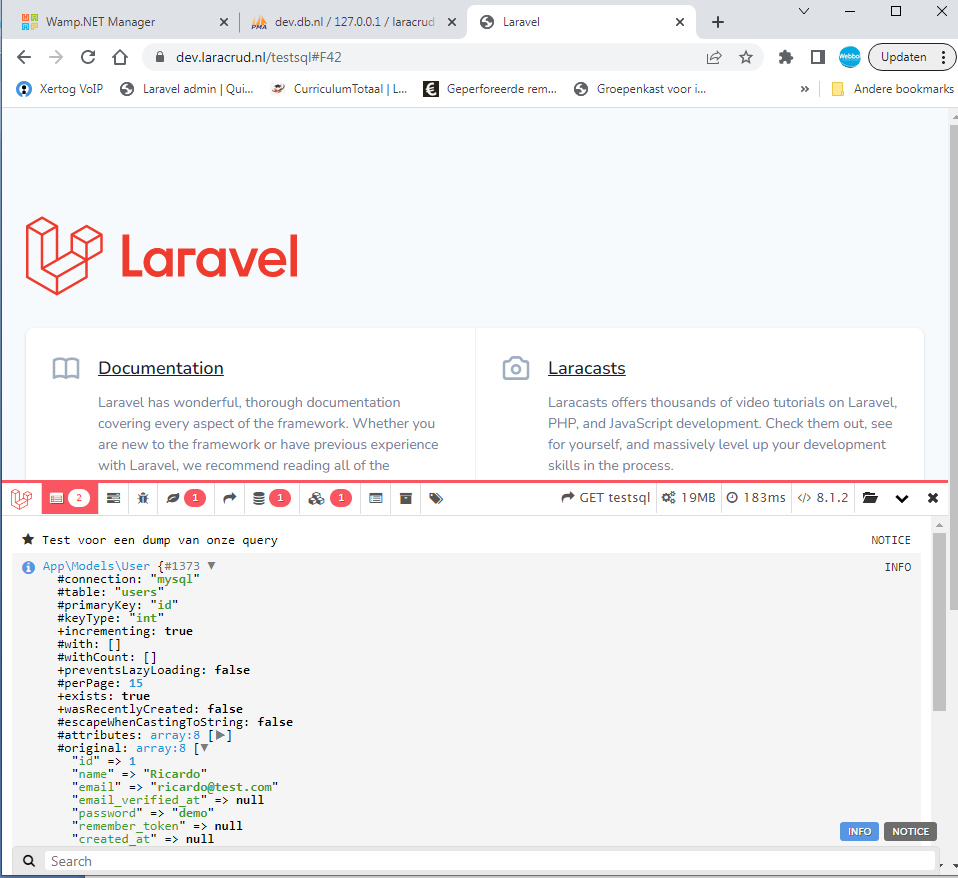
**Dus we hebben een sql query en we willen ook een resultaat hiervan zien.**

**In je web.php**

****

**Het zit wat verstopt in de debugbar maar hier zit het:**

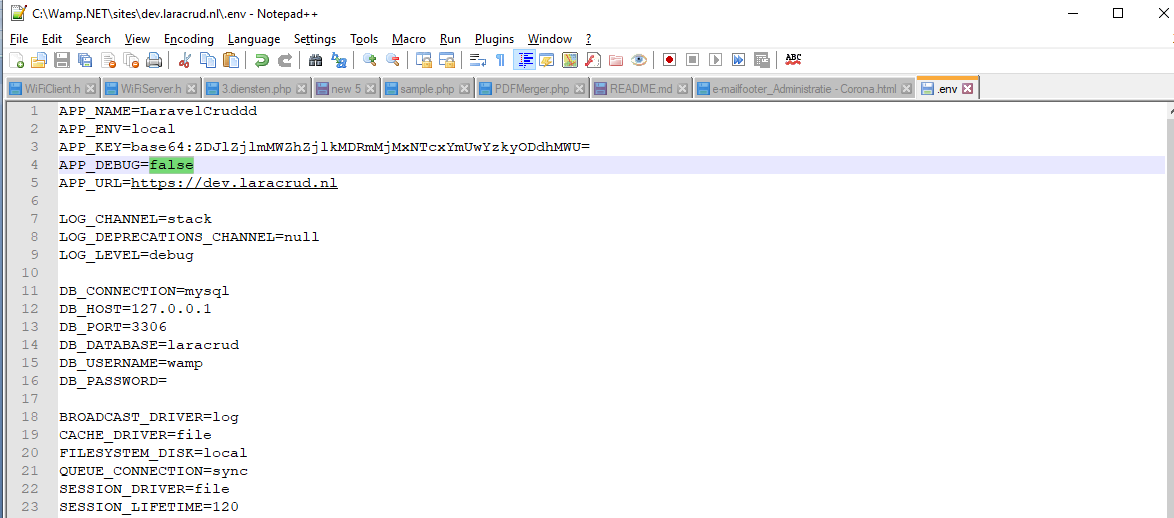
**-uitklappen**

****

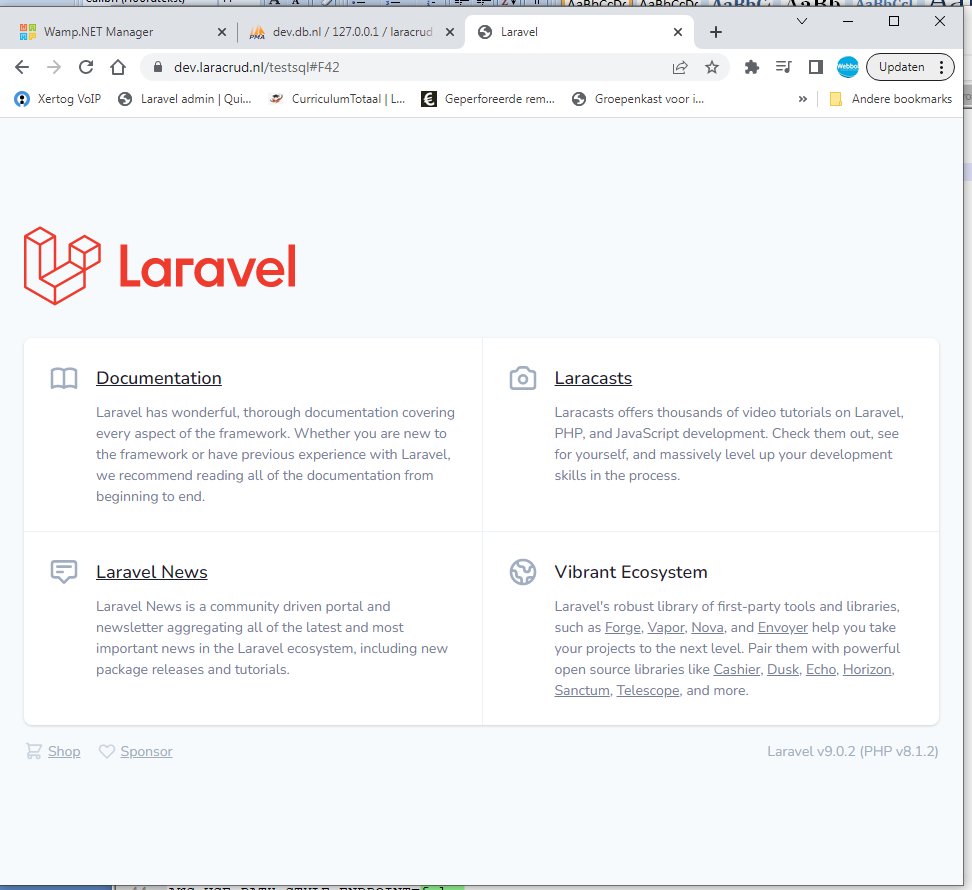
**Hoe kun je de debugbar uitzetten?**

**Open je .env bestand.**

**Set debug = false**

****

**En weg zijn de meldingen.**

****

**Handige links:**

**Git pagina:**

[**https://github.com/barryvdh/laravel-debugbar**](https://github.com/barryvdh/laravel-debugbar)

You can now add messages using the Facade (when added), using the PSR-3 levels (debug, info, notice, warning, error, critical, alert, emergency):

Debugbar::info($object);

Debugbar::error('Error!');

Debugbar::warning('Watch out…');

Debugbar::addMessage('Another message', 'mylabel');

And start/stop timing:

Debugbar::startMeasure('render','Time for rendering');

Debugbar::stopMeasure('render');

Debugbar::addMeasure('now', LARAVEL\_START, microtime(true));

Debugbar::measure('My long operation', function() {

// Do something…

});

Or log exceptions:

try {

throw new Exception('foobar');

} catch (Exception $e) {

Debugbar::addThrowable($e);

}

There are also helper functions available for the most common calls:

// All arguments will be dumped as a debug message

debug($var1, $someString, $intValue, $object);

// `$collection->debug()` will return the collection and dump it as a debug message. Like `$collection->dump()`

collect([$var1, $someString])->debug();

start\_measure('render','Time for rendering');

stop\_measure('render');

add\_measure('now', LARAVEL\_START, microtime(true));

measure('My long operation', function() {

// Do something…

});

If you want you can add your own DataCollectors, through the Container or the Facade:

Debugbar::addCollector(new DebugBar\DataCollector\MessagesCollector('my\_messages'));

//Or via the App container:

$debugbar = App::make('debugbar');

$debugbar->addCollector(new DebugBar\DataCollector\MessagesCollector('my\_messages'));

By default, the Debugbar is injected just before </body>. If you want to inject the Debugbar yourself, set the config option 'inject' to false and use the renderer yourself and follow <http://phpdebugbar.com/docs/rendering.html>

$renderer = Debugbar::getJavascriptRenderer();

Note: Not using the auto-inject, will disable the Request information, because that is added After the response. You can add the default\_request datacollector in the config as alternative.

## Enabling/Disabling on run time

You can enable or disable the debugbar during run time.

\Debugbar::enable();

\Debugbar::disable();

NB. Once enabled, the collectors are added (and could produce extra overhead), so if you want to use the debugbar in production, disable in the config and only enable when needed.

## Twig Integration

Laravel Debugbar comes with two Twig Extensions. These are tested with [rcrowe/TwigBridge](https://github.com/rcrowe/TwigBridge) 0.6.x

Add the following extensions to your TwigBridge config/extensions.php (or register the extensions manually)

'Barryvdh\Debugbar\Twig\Extension\Debug',

'Barryvdh\Debugbar\Twig\Extension\Dump',

'Barryvdh\Debugbar\Twig\Extension\Stopwatch',

The Dump extension will replace the [dump function](http://twig.sensiolabs.org/doc/functions/dump.html) to output variables using the DataFormatter. The Debug extension adds a debug() function which passes variables to the Message Collector, instead of showing it directly in the template. It dumps the arguments, or when empty; all context variables.

{{ debug() }}

{{ debug(user, categories) }}

The Stopwatch extension adds a [stopwatch tag](http://symfony.com/blog/new-in-symfony-2-4-a-stopwatch-tag-for-twig) similar to the one in Symfony/Silex Twigbridge.

{% stopwatch "foo" %}

…some things that gets timed

{% endstopwatch %}