Laravel Vs Symfony

# Intro

Laravel en Symfony zijn beiden PHP frameworks. Ze zijn een set aan componenten dat je kan gebruiken om snel en efficiënt een webapplicatie op te starten, of een zeer complexe webapplicatie uit te bouwen maar toch overzichtelijk te houden. Welke van de twee is er dan de “best”? Hieronder vind je de belangrijkste verschillen tussen te twee frameworks. En enkele benchmark tests.

De maker van Laravel is Taylor Otwell - Links (US) (<https://github.com/taylorotwell>).   
De maker van Symfony is Fabien Potencier - Rechts (FR) (<https://github.com/fabpot>). Was dit jaar aanwezig op PHP Benelux.





# Sharing is caring

Zowel Laravel als Symfony zijn open source. Dit wil niet alleen zeggen dat ze gratis zijn, maar dat je ook verder mag bouwen op hun software.

Dit is dan ook wat Laravel heeft gedaan. De core van Laravel is Symfony. Dingen zoals HTTP, routing, sessions, etc. komen allemaal van Symfony bouwblokken. Dit wil zeggen dat wanneer Symfony updates uitbrengt, Laravel hierin volgt.

# Complexiteit

De populariteit van Laravel is erg gegroeid over de laatste jaren. Vroeger was Symfony de nummer 1 maar Laravel heeft enorm aan populariteit gewonnen en Symfony voorbij gestoken in gebruikers. Hoe komt dit?

Wanneer je nieuw bent in PHP, begin je best met Laravel, omdat Laravel heel eenvoudig aan te leren is. Symfony is véél complexer, ingewikkelder en moeilijker om aan te leren. Dit heeft bijvoorbeeld te maken met het feit dat Laravel vooral impliciet werkt, terwijl Symfony uitsluitend expliciet werkt.

Symfony werkt met bouwblokken, Laravel is een volledig pakket op zich. Dit wil wel zeggen dat je Symfony veel harder kan aanpassen naar je noden als developer en vooral naar het te schrijven project. Wat niet wil zeggen dat Laravel geen bouwblokken heeft om eraan toe te voegen.

# View templating engines

Laravel gebruikt de Blade templating engine en Symfony gebruikt Twig. Beide templating engines werken in grote lijnen hetzelfde en bevatten template inheritance, sections, geëscapete output en propere, duidelijke syntax.

# Blade

Blade is de standaard templating engine voor Laravel. De syntax is origineel geïnspireerd door ASP.net Razor. Het grootste voordeel aan Blade is zijn @section feature. Dit zorgt ervoor dat je dingen kan hergebruiken, of enkel een bepaald stuk kan wijzigen op een pagina.

Voorbeeld:

**@extends(**'layouts.master'**)  
@section(**'content'**)  
 @foreach (**$users **as** $user**)** <p>{{ $user->name }}</p>  
 **@endforeach  
@endsection**

# Twig

Deze templating engine zit standaard in Symfony2. Twig kan gebruikt worden in standalone projecten omdat het modulair is. Durpal bijvoorbeeld, maakt gebruik van Twig. Fabien vind security belangrijk. Daarom is heeft Twig een extra laag security en is het bijvoorbeeld niet mogelijk zomaar functies uit te voeren in de view.

Voorbeeld:

{% extends "layouts.master" %}  
{% block content %}  
 {% for user in users %}  
 <p>This is user {{ *user*.*id* }}</p>  
 {% endfor %}  
{% endblock %}

Gelijkenissen:

* Beide compilen naar vanilla PHP voor betere prestaties
* Beide bieden de mogelijkheid simpel variabelen te outputten
* Beide bevatten structure control
* Beide escapen automatisch
* Beide bevatten template inheritance en sections

Verschillen:

* Andere syntax voor control structures
* Behandelen escaping anders
* Behandelen variables en toegang hiertoe op een andere manier
* Andere manier van functionaliteit toe voegen
* Een andere kijk op security

# ORM

Laravel maakt hiervoor gebruik van Eloquent (active record). Symfony gebruikt Doctrine (Data mapper).

## Migrations

Beide systemen maken gebruik van migrations om hun database op te bouwen. Migrations zijn lijsten van aanpassingen aan de database. Dit is handig voor ontwikkeling want wanneer iemand een branch pulled waar een database aanpassing is gemaakt, moet die persoon enkel migraten om de benodigde aanpassing uit te voeren.

Voorbeeld Laravel:

**public function** up()  
{  
 Schema::*create*('flights', **function** (Blueprint $table) {  
 $table->increments('id');  
 $table->string('name');  
 $table->string('airline');  
 $table->timestamps();  
 });  
}

Voorbeeld Symfony:

**public function** up(Schema $schema)  
{  
 $this->abortIf($this->connection->getDatabasePlatform()->getName() != 'mysql', 'Migration can only be executed safely on \'mysql\'.');  
 $this->addSql('CREATE TABLE address (id **INT** AUTO\_INCREMENT NOT NULL, street **VARCHAR**(255) DEFAULT NULL, street\_number **VARCHAR**(255) DEFAULT NULL, postal\_code **VARCHAR**(255) DEFAULT NULL, city **VARCHAR**(255) DEFAULT NULL, country **VARCHAR**(255) DEFAULT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8 COLLATE utf8\_unicode\_ci ENGINE = InnoDB');  
 $this->addSql('ALTER TABLE person ADD address\_id **INT** DEFAULT NULL, DROP address\_street, DROP address\_street\_number, DROP address\_postal\_code, DROP address\_city, DROP address\_country');  
 $this->addSql('ALTER TABLE person ADD CONSTRAINT FK\_34DCD176F5B7AF75 FOREIGN KEY (address\_id) REFERENCES address (id)');  
 $this->addSql('CREATE INDEX IDX\_34DCD176F5B7AF75 ON person (address\_id)');  
}

Zoals je kan zien is het voorbeeld van Symfony weer expliciet terwijl het voorbeeld van Laravel impliciet is.

## Performantie

Omdat Eloquent (Laravel) een active record ORM is, breng het vaak veel queries met zich mee. Dit komt door hoe eloquent haar relaties opbouwt. Voor elke tabel maak je een model aan en geeft je de relaties van die tabel mee in functies. Met deze functies kan je dan bij het ophalen van een object meteen aan alles wat eraan vast hangt.

Bijvoorbeeld:

**class** User **extends** Model  
{  
 **public function** phone()  
 {  
 **return** $this->hasOne('App\Phone');  
 }  
}

Dit zegt dat User één telefoon nummer kan bevatten. Als je dan een user object “$user” hebt kan ge gewoon zeggen “$user->phone” en zorgt eloquent vanzelf voor een queries dat je het “phone” object geeft van die user.

## Entity annotation

Wanneer we een entity gaan ophalen uit de databank met eloquent, krijgen we een eloquent collection object. Dit wil zeggen dat we een aanpassing kunnen maken aan het object en gewoon de save functie callen zoals hieronder:

$user = User::find(3);  
$user->name = “Sam”;  
$user->save();

De save() functie is genoeg om de verandering te persisten naar de databank.

De entity manager in Doctrine geeft gewone php objecten terug. Dit zorgt ervoor dat we niet zomaar de save() method kunnen callen. Doctrine werkt met de EntityManager class. Deze is speciaal geschreven om alle persistance logica te bevatten.

Voorbeeld:

$user = **new** User;  
$user->setName(‘Philip Brown’);  
  
EntityManager::persist($user);  
EntityManager::flush();

## Transactional write-behind

Zoals je hierboven kan zien moet je in Doctrine eerst je object persisten, en dan flushen. Dit komt door de strategie dat Doctrine gebruikt genaamd transactional write-behind. Dit wil zeggen dat Doctrine de interacties gaat bijhouden totdat flush() gecalled wordt. Dit wil zeggen dat Doctrine alle objecten bijhoud waar persist() op gecalled is geweest en ze dan allemaal tegelijk persist wanneer je de flush() functie called.

## Grabbing Entities

Als je een entity uit een active record database wenst te halen kan je gewoon zeggen:  
$user = User::find(4);

Voor doctrine gebruiken we de entity manager in plaats van het model:  
$user = EntityManager::find(‘Cribbb\Entities\User’, 1);

## Repositories

Strikt genomen volgt het gebruik van repositories bij Active Record-stijlobjecten niet het repositorypatroon, omdat elk Active Record-object de database kan manipuleren, ongeacht of deze zich binnen de repository bevindt of niet. Om trouw te blijven aan het Repository-patroon moeten we echt standaard PHP-objecten returnen in plaats van instances van een Eloquent-model.

Wanneer we een entity willen vinden op basis van id, kunnen we het volgende doen:

$user = EntityManager::find(‘Cribbb\Entities\User’, 1);

Op de achtergrond zal her volgende gebeuren:

$user = EntityManager::getRepository(‘Cribbb\Entities\User’)->find($id);

De getRepository () method retourneert een repository waarmee je de database kunt opvragen. Je hoeft deze repository niet echt te maken, aangezien Doctrine 2 een exemplaar van Doctrine \ ORM \ EntityRepository returned. Dit object heeft een aantal nuttige methods om de databank te doorzoeken.

Voorbeeld:

$users = EntityManager::getRepository(‘Cribbb\Entities\User’)->findBy([‘location’ => ‘UK’]); $users = EntityManager::getRepository(‘Cribbb\Entities\User’)->findOneBy([‘username’ => ‘philipbrown’]);

# Doctrine Query Language (DQL)

Grote applicaties in lopen vaak tegen het feit dat zeer complexe database queries niet mogelijk zijn met eloquent. Laravel heeft hiervoor een query builder dat meer controle geeft over je uit te voeren databanktransacties.

DQL is een objct query language. Dit wil zeggen dat er niet wordt gekeken naar queries als een bundel van relaties, maar als een bundel van objecten. DQL is een zéér krachtige manier van queries uit te voeren. Daarom wordt het aangeraden deze te gebruiken bij complexere databank structuren en queries om zéér exacte queries te kunnen schrijven.

# Configuratie

Beide projecten gebruiken config files om belangrijke parameters in te stellen op het project. Bijvoorbeeld welke mail driver er gebruikt moet worden, of in welke taal de applicatie staat. Laravel gebruikt gewone PHP files en Symfony maakt gebruik van yaml files.

Voorbeeld:

<?php  
 **return array**(  
 'host' => 'localhost',  
 'username' => 'root',  
);

VS

# config/packages/acme\_social.yaml  
acme\_social:  
 twitter:  
 client\_id: 123  
 client\_secret: your\_secret

# Benchmarks

Benchmarks Eloquent Vs Doctrine.

Benchmarks zijn uitgevoerd in Laravel. Er is een heel basic database aangemaakt. Hieraan worden 1000 records toevoegen (strings van 1000 woorden lorem ipsum) en te tijd meten. Daarna zijn de records uitlezen en weer te tijd meten. Dit is gebeurd in zowel Eloquent als Doctrine om de verschillen te meten.

## Writing

### Eloquent

**public function** eloquent()  
{  
 Benchmark::*truncate*();  
  
 $start = microtime(**true**);  
 **for** ($i = 0; $i < 1000; $i++) {  
 $benchmark = **new** Benchmark();  
 $benchmark->content = config('benchmark.string');  
 $benchmark->save();  
 }  
 $end = microtime(**true**);  
  
 dd($end-$start);  
}

Resultaat: 13.4577 seconden

### Doctrine

**public function** doctrine()  
{  
 Benchmark::*truncate*();  
  
 $start = microtime(**true**);  
 **for** ($i = 0; $i < 1000; $i++) {  
 $benchmark = **new** BenchmarkDoctrine(config('benchmark.string'));  
 EntityManager::persist($benchmark);  
 EntityManager::*flush*();  
 }  
  
  
 $end = microtime(**true**);  
  
 dd($end-$start);  
}

Resultaat: 14.8035 seconden

In dit voorbeeld wordt er elke keer geflushed en ge’save()’d(effectief de record wegschrijven). Als we flushen nadat we alle records hebben gepersist krijgen we het volgende resultaat:

### Eloquent

**public function** eloquent()  
{  
 Benchmark::*truncate*();  
  
 $records = [];  
 **for** ($i = 0; $i < 1000; $i++) {  
 $records[$i] = ['content' => config('benchmark.string')];  
 }  
   
 $start = microtime(**true**);  
 DB::*table*('benchmarks')->insert($records);  
 $end = microtime(**true**);  
  
 dd($end-$start);  
}

Resultaat: 7.2904 seconden

### Doctrine

**public function** doctrine()  
{  
 Benchmark::*truncate*();  
  
 $start = microtime(**true**);  
 **for** ($i = 0; $i < 1000; $i++) {  
 $benchmark = **new** BenchmarkDoctrine(config('benchmark.string'));  
 EntityManager::persist($benchmark);  
 }  
  
 EntityManager::*flush*();  
  
 $end = microtime(**true**);  
  
 dd($end-$start);  
}

Resultaat: 2.1205 seconden

Dit is een verschil van ruim 12 seconden (85.81% sneller).

## Reading

### Eloquent

**public function** eloquentRead()  
{  
 $start = microtime(**true**);  
  
 $records = Benchmark::*all*();  
  
 $end = microtime(**true**);  
  
 dd(($end-$start)\*1000);  
}

Resultaat: 64.2869 milliseconden

### Doctrine

**public function** doctrineRead()  
{  
 $start = microtime(**true**);  
  
 $repository = EntityManager::*getRepository*(BenchmarkDoctrine::**class**);  
 $record = $repository->findAll();  
   
 $end = microtime(**true**);  
  
 dd(($end-$start)\*1000);  
}

Resultaat: 136.0201 milliseconden

# Results:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Benchmarks | Eloquent | Doctrine | Winner | Faster |
| Write with multiple persists | **13.457 s** | 14.803 s | Eloquent | 9.09 % |
| Write with one persist | 7.290 s | **2.120 s** | Doctrine | 343.86 % |
| Read | **64.286 s** | 136.020 s | Eloquent | 211.58 % |

# Conclusie

Beide frameworks zijn enorm efficiënt in wat ze doen. Het hangt er erg vanaf wat de noden zijn van de developer of het project om te zeggen wat het “beste” framework van de twee is.

Door dit onderzoek heb ik alleszins de smaak te pakken om mijn kennis uit te breiden met het Symfony framework en mezelf te verdiepen in Doctrine.

Bronnen:

* <https://laravel.com/>
* <https://symfony.com/>
* <https://www.valuecoders.com/blog/technology-and-apps/symfony-vs-laravel-php-framework-choose/>
* <https://www.codementor.io/michalwojtasik/what-are-the-best-php-frameworks-in-2018-symfony-laravel-or-else-let-s-compare-those-two-jcvwzpkl3>
* <https://medium.com/@barryvdh/comparing-blade-and-twig-templates-in-laravel-187fde7fcac9>
* <https://www.drupal.org/docs/8/theming/twig>
* <https://sarvendev.com/en/2018/01/active-record-eloquent-vs-data-mapper-doctrine-en/>
* <https://www.culttt.com/2014/07/07/doctrine-2-different-eloquent/>
* <https://medium.com/@shahroznawaz/symfony-laravel-benchmark-c4721d0582f>
* <https://symfony.com/doc/current/bundles/configuration.html>