Laravel 2019

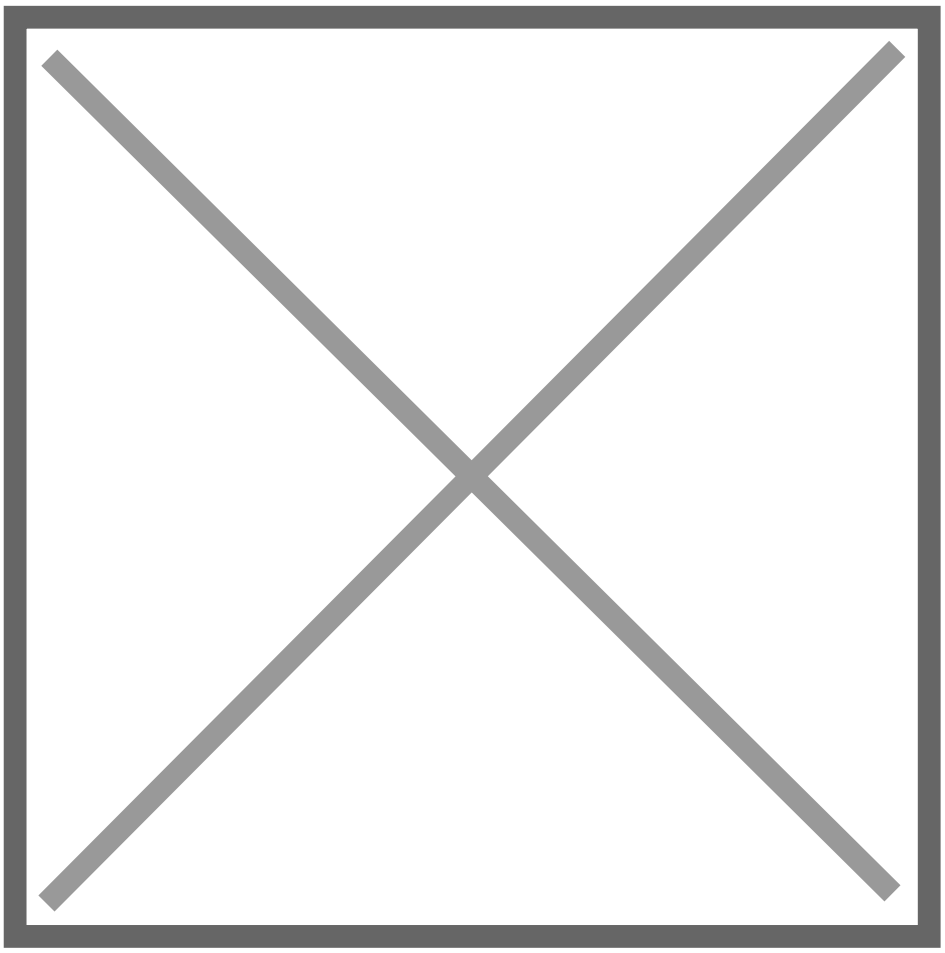
Project Links
Aan de hand van een eenvoudig project gaan we de basics van Laravel leren. Dit is het opzetten van een database, het opzetten van forms en templates en het opzetten van controls. We gaan een volledige CRUD pagina maken.

- Overzicht
- Composer en Laravel
- Database Migration
- Routes en Read
- Create
- Delete
- Refactoring to MVC
- Front End
- Create en Edit
- Opdrachten
- Validation
- Login (Laravel 6)
- Search
- Extra opdracht - count user clicks
- Opdracht
- Toets nabesspreking
- Laravel stappenplan
- Laravel - aanpassen applicatie

Overzicht

Wat gaan we doen met Laravel?

We gaan een applicatie maken waarmee we links kunnen opslaan. We kunnen links toevoegen, veranderen, bekijken en verwijderen.



In deze lessen serie gaan we:

1. Laravel installeren.
2. Een database met één table opzetten met Laravel (artisan).
3. We gaan een virtual host in Apache instellen.
4. We gaan tabellen vullen en een database migratie uitvoeren.
5. We gaan leren wat routing is en gaan dat voor één object opzetten.
6. We gaan de CRUD controls voor één object opzetten.
7. We gaan templates opzetten.

8. We gaan back-end validation opzetten.
9. We gaan een login maken en we gaan verschillende functies achter deze login plaatsen.
10. We gaan een search functie maken.

Uiteindelijk hebben we een werkende Laravel aaplicatie van scratch opgezet met één object en hebben we alle CRUD functies op de juiste manier geïmplementeerd.

> Voor naslag kun je gebruik maken van: https://laravel.com/docs/6.x

Bewaar deze link, want je kunt daar alles over Laravel vinden (ook tijdens je examen)!

--

Composer en Laravel

In deze les leren we wat Composer is, hoe we Laravel installeren en een nieuw Laravel project starten. Aan het eidn van deze les hebben we een 'leeg' Laravel project.

Buckle up....

Composer

Als we een framework als Laravel gaan gebruiken dan installeren we eigenlijk een hele grote doos met allemaal bouwblokken. Deze blokken moeten allemaal samen werken en zijn vaak afhankelijk van elkaar. De blokken worden door verschillende mensen ontwikkeld en op één of andere manier moet er voor worden gezorgd dat de juisten blokken zijn geinstalleerd. Hiervoor dient Composer. Composer is als het ware de installer van PHP tools, deze worden vaak libraries of packages genoemd. Librabries zijn dan de blokken functionaliteit die je kunt gebruiken om een programma te maken.

Formeel kun je zeggen dat Composer een package manager is voor PHP. Je zou dit vrij kunnen vertalen en kunnen zeggen dat Compser de installer voor PHP is.

Installeren Composer

https://getcomposer.org/

Document Root

Je document root is een belangrijke directory. Het is de directory waar jouw web server 'begint'. Het is de directory die je ziet als je naar de webserver gaat. Meestel doe je dit via loclahost of 127.0.0.1.

Open een command window door in windows in je search "command prompt" op te starten. Je kunt ook de gratis tool CMDer downloader die werkt iets prettige. Op Mac moet je terminal window openen.

Ga nu naar je document root. Standaard is dit op windows met XAMPP `c:\xampp\htdocs\`

Install Laravel

Met de 'installer' composer installeren we nu Laravel.

```
composer global require "laravel/installer"
```

New Laravel Project

We maken nu een nieuw laravel poject en noemen dat project 'links'. Dat doen we vanuit de document root van onze webserver.

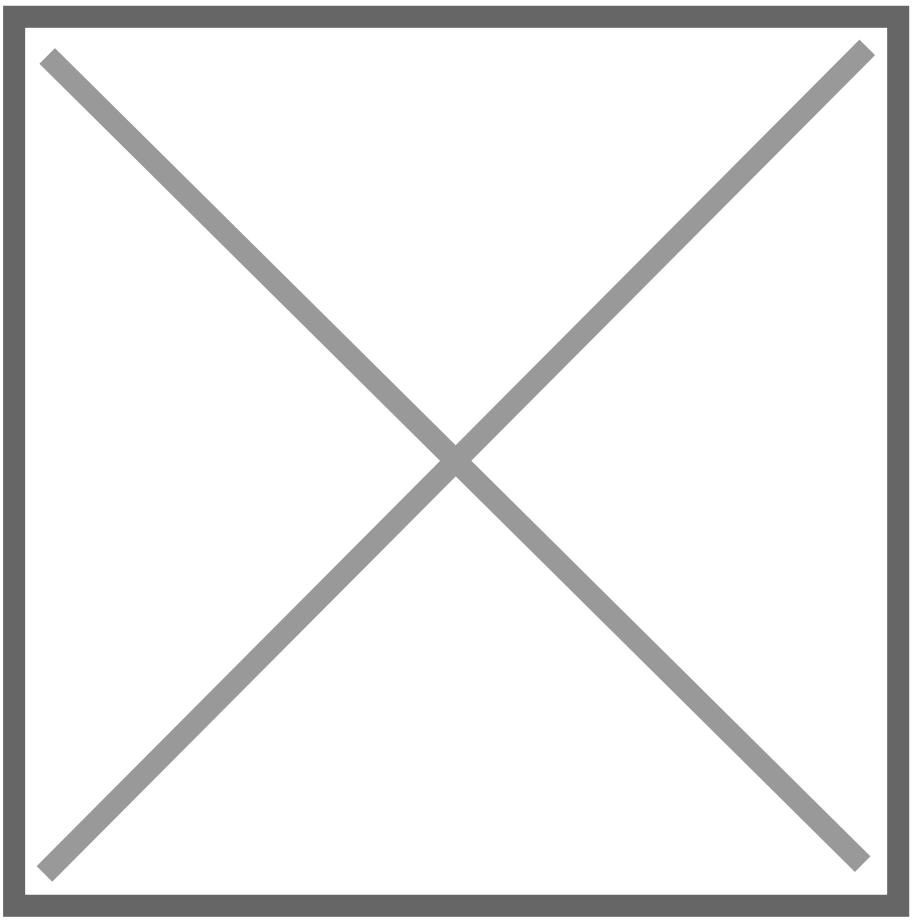
```
laravel new links
```

Aan het eind zie je:

> **Application ready! Build something amazing.**

Ga nu naar de site: http://localhost/links/public/

Als het goed is zie je nu je 'legel' nieuwe laravel site.



We hebben nu een standaard 'lege' Laravel applicatie. Leeg is die eigenlijk niet helemaal want er zitten veel zaken al standaard al in laravel ingebakken.

Versie afhankelijkheden

Laravel is een bibliotheek met veel subsystemen. De versies passen allemaal bij elkaar, maar de PHP en SQL moet ook een bepaalde versie hebben. Dit zijn externe afhankelijkheden. Als dit niet het geval is dan zul je PHP/SQL moeten updaten. Dat kan dan vaak weer door XAMPP te updaten.

```
// Laravel version (Laravel Framework 7.17.2)
php artisan --version

// PHP version (PHP 7.3.11)
php -v
```

Voor versie 7.x van Laravel heb je minimaal PHP 7.2.5 nodig, zie https://en.wikipedia.org/wiki/Laravel

Als je geen gebruik maakt van XAMMP dan moet je controleren of de juiste PHP-extensies zijn geïnstalleerd. Zie https://laravel.com/docs/7.x#server-requirements (in XAMPP zitten deze PHP extensies).

In de volgende les gaan we aan de slag met het maken van een database. Dat gaat in Laravel ook anders dan we gewend zijn.

--

Database Migration

In deze les leren we

- *hoe via Laravel een websever kunnen opstarten en;*
- *hoe we met Laravel een database aanmaken en vullen met testdata.*

Aan het eind van deze les hebben we ee nieuwe database die is gevuld met test data.

Laravel web server

We zagen in de vorige les dat we naar ons nieuw Laravel project gaan via de link 127.0.0.1/links/public. Uiteindelijk als we productie gaan draaien willen we een eigen domain name zoals www.links.com. Voor nu kunnen we een 'virtual' host op een andere poort aanmaken met het commando:

```
php artisan serve
```

Nadat je php artisan server heb gestart zie je het volgende:

```
Laravel development server started: http://127.0.0.1:8000
```

Dit betekent dat je Laravel project draait op `http://127.0.0.1:8000`

Als we naar 127.0.0.1:8000 gaan dan gaan we dus naar een nieuwe document root, namelijk c:/xampp/httpdocs/links/public

Deze directory is de (Laravel) project directory en als we via de Laravel webserver nasar 127.0.0.1:8000 gaan dan komen we rechtstreeks in de directory c:/xampp/httpdocs/links/public

We kunnen XAMPP naast de Laravel server laten draaien. Dat kan ook handig zijn om phpmyadmin te kunnen gebruiken. XAMPP en Laravel serve zitten elkaar niet in de weg, omdat ze beide andere poorten gebruiken.

Artisan

Artisan (ambachtsman) is de command line tool die bij Laravel hoort. Je kunt er allemaal handige dingen mee doen.

>

> Let op het uitvoeren van een artisan command doen we vanuit de project directory.

Database Migratie

Met Artisan maken we een tabel, eerst voeren we het volgende in de command prompt uit.

```
// Check of je in de juiste directory staat
php artisan make:migration create_links_table --create=links
```

Dit commando maakt een file. In de output van dit commando staat de naam van de file die is gemaakt. Deze staat in `database/migrations/` en heet `{{datetime}}_create_links_table.php`

Open deze file.

> tip: ga naar de direcory waar deze file in staan in je command prompt en type *notepad* gevolgd door de naam van de file.

In het script staan twee functies up() and down(). up() wordt gebruikt om een nieuwe database te maken en down kan worden gebuikt om deze database weer te verwijderen.

Migratie scripts kunnen worden gebruikt om een bestaande database aan te passen (migrate) of een nieuwe database aan te maken.

Verander de function up() zodat die er als volgt uit ziet (niet alles kopiëren; alleen de functie/method aanpassen).

```
public function up()
{
    Schema::create('links', function (Blueprint $table) {
        $table->increments('id');
        $table->string('title');
        $table->string('url')->unique();
        $table->text('description');
        $table->timestamps();
    });
}
```

Om het database script te draaien moet Laravel toegang hebben tot de database. Edit hiervoor de `.env` file in de *files* directory.

In de .env file staan configuratie parameters zoals de database naam en userid en password van de database.

In de .env.example file staat een complete voorbeeld config waar alle mogelijke configuratie-parameters in staan. Voor nu beperken we ons even tot de database connectie.

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=links
DB_USERNAME=root
DB_PASSWORD=
```

Maak nu met phpMyAdmin de database `links`.

Voer de migratie uit:

```
php artisan migrate
```

De output moet zijn: Migration table created successfully.

Bij het uitvoeren van een migratie van een product dat je via githib hebt 'ge-cloned' ,moet je mogelijk nog alle vendor packages downloaden en de dependencies controleren, dat kan met:

```
composer update --no-scripts
```

Composer zorgt er dan voor dat alle benodigde software wordt geïnstalleerd. Dit hoeft dus niet bij een 'verse' installatie van Laravel.

Als je een foutmelding krijgt die iets over key lengte zegt, kijk dan naar het commentaar aan het eidne van deze les.

In alle andere gevallen moet je de foutmelding bestuderen en met behulp van Google uitvogelen wat er aan de hand is. Het meest aannemelijke is dat er op één of andere manier geen verbinding met de database kan worden gemaakt.

Kijk nu via phpmyadmin welke of de tabel is aangemaakt.

Nu volgen nog wat tips over de migration, als het niet goed is gegaan bijvoorbeeld, hoe kun je het dan terug draaien. Indien je voor een tweede keer deze les doorloopt kan (hoeft niet) je doorgaan naar 'Tabel Vullen' hieronder.

Draai nu de migratie terug met het commando:

```
php artisan migrate:rollback
```

Kijk nu weer via phpmyadmin wat er is gebeurd.

Voer tenslotte de migratie nog een keer uit (probeer zelf te bedenken met welk commando).

Met het commando:

```
 php artisan help migrate
```

Zie je alle migratie opties. Kijk zelf in de opties hoe je een commando kan geven waarbij er niets wordt uitgevoerd maar waarbij er wel wordt getoond wat er *zou* worden uitgevoerd. Dit kun je de *net-als-of* optie noemen.

met het commando:

```
php artisan migrate:refresh
```

Combineer je een rollback en een migrate.

Opdracht

Bestudeer de help van de migrate en kijk met deze *net-alsof-optie* wat er wordt uitgevoerd bij een migrate en een rollback.

Tabel vullen

De tabel is nu aangemaakt in de database links. NU gaan we de tabel vullen.

Om aan de slag te gaan is het handig als we onze tabel met wat data kunnen vullen. Ook daar zijn handige commando's voor.

```
php artisan make:model --factory Link
```

Dit commando maakt een model (van MVC) voor het object link en het maakt een file in de directory databases/factories die nodig is om de database te vullen..

Nadat je dit commando hebt uitgevoerd, open je de file LinkFactory.php in de genoemde directory.

Pas de code in deze file aan op de volgende manier:

```
$factory->define(Link::class, function (Faker $faker) {
    return [
    'title' => substr($faker->sentence(2), 0, -1),
'url' => $faker->url,
'description' => $faker->paragraph,
    ];
```

```
});
```

De $faker->sentence(2) method maakt een random zin van 2 woorden en met de substring functie halen we het laatste karakter weg, omdat dat een punt is die we er niet bij willen hebben.

De $faker->url genereert een random url.

En de $faker->paragraph, je raadt het al, maakt een stukje tekst, een paragraaf.

Nu moeten we een script maken dat met deze omschrijving de tabellen vult, dat doen we met:

```
php artisan make:seeder LinksTableSeeder
```

(denk eraan dat je in de juiste directory staat als je een Artisan commando uitvoert)

In de database/seeds directory staat de file LinksTableSeeder.php die we net hebben aangemaakt. Open deze en verander deze.

```
public function run()
{
factory(App\Link::class, 5)->create();
}
```

De 5 geeft aan dat we 5 rijen willen vullen met data.

Open vervolgens de file DatabaseSeeder.php indezelfde directory en plaats de volgende code in deze file:

```
public function run()
{
    $this->call(LinksTableSeeder::class);
}
```

De code in deze file zorgt ervoor dat de LinksTableSeeder.php wordt uitgevoerd.

Let op dat de naam *LinksTableSeeder* case sensitivie is en verwijst naar de file die je in de vorige stap hebt gemaakt.

We gaan een nieuwe migration uitvoeren en zorgen er voor dat tijdens deze migratie de tabellen gevuld worden met data op de manier zoals we zojuist hebben gedefinieerd.

```
php artisan migrate:refresh --seed
```

Controleer met phpmyadmin of de tabel links is gevuld.

Ga door met de volgende les, als er data in de tabel staat.

Samengevat

Er zijn nog wat andere handige commando's die tijdens het ontwikkelen van een applicatie goed van pas kunnen komen, alle handige migratie commando's op een rijtje:

Commando	Beschrijving
php artisan migrate	**voer de migratie uit**
php artisan migrate:rollback	maak de migratie weer ongedaan
php artisan migrate:refresh	**doe een rollback en een migrate**
php artisan migrate:refresh --seed	doe een rollback en een migrate en vul de tabellen met data
php artisan migrate:fresh	drop alle tabellen en doe opnieuw een migrate
php artisan help	alle commando's
php artisan migrate help	help over het commando migrate

We zijn de volgende files tegengekomen:

files (vanuit project directorie)	Beschrijving
database/migrations/<create_table>.php	definieer hoe de tabel moet worden gemaakt
.env	file met configuratie opties zoals database opties
database/factories/<table>Factory.php	Definieer hoe tabel te vullen met test data
database/seeds/<table>TableSeeder.php	Roep functie X keer aan om rijen te genereren
database/seeds/DatabaseSeeder.php	Vanuit hier dient de TableSeeder te worden aangeroepen

Comment

(door Rocco)

voor de error met Artisan migrate die met length te maken heeft
ga naar de AppServiceProvider.php file in je project folder en pas dit aan

```
<?php


namespace App\Providers;
```

```
use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\Schema;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Schema::defaultStringLength(191);
    }
}
```

Routes en Read

In deze les leren we wat een virtuele app is en hoe we deze in Apache configureren.

Daarna gaan we een route in ons Laravel project opzetten en laten we de inhoud van onze table links die we in de vorige les hebben gemaakt, met een blade template afdrukken. We gaan de R van CRUD maken.

Document Root

Even eenreminder. De *document root* is de directory waar je webserver begint. De document root is een configuratie parameter in de webserver config file. De parameter verwijst naar een directory op je file systeem en deze directory is waar de webserver zijn web pagina's verwacht.

Virtual App

Het opzetten van een virtuele app onder XAMPP blijkt niet goed te werken. Ik moet nog een keer uitzoeken waarom niet. Voor nu kan dit hoofdstuk over Virtual App worden overgeslagen.

De nieuwe Laravel App staat onder localhost/links/public waarbij links/public in de document root staat.

Op de file: C:\xampp\apache\conf\extra\httpd-vhosts.conf en plaats de volgende configuratie:

```
<VirtualHost *:80>
    ServerName links.local
    DocumentRoot "C:/xampp/htdocs/links/public"
    <Directory C:/xampp/htdocs/links/public>
        Require all granted
        AllowOverride All
    </Directory>
</VirtualHost>
```

Restart de Apache webserver. Wat er nu is ingesteld is dat als de webserver een verzoek krijgt van het domein links.local de webserver's document root verandert naar c:/xampp/htdocs/links.

Nu moeten we alleen nog zorgen dat links.local naar het ip adress 127.0.0.1 resolved. Dat kan door de hosts file onder windows aan te passen. Open notepad in admin mode en open C:\Windows\System32\drivers\etc\hosts

Voeg de volgende regel toe:

```
127.0.0.1 links.local
```

Deze regel verteld jouw computer dat als je links.local in tikt hierbij het ipaddres 127.0.0.1 hoort. Je computer gaat dan niet meer verder naar de DNS van internet om te vragen wat het ip address van links.local is.

Routes

Een route is een manier voor de webserver om te weten welke pagina hij moet laden. In een traditionele web app komt het pas in de URL overeen met het path in je document root.

Bijvoorbeeld als je gaat naar localhost/myproject/b/c/input.php

Dan kun je in je *document root* een directory vinden die myproject heet en daarin staat een directory die b heet en daarin staat dan weer de directory c waarin uiteindelijk de file input.php staat.

In Laravel werkt dit anders.

Via een slimme truuk wordt alles wat naar Laravel verwijst opgevangen en het path uit de url wordt door de webserver en Laravel ontleed. Stel myproject was een Laravel project in het vorige voorbeeld dan wordt de rest van het path b/c/input.php als een parameter aan Laravel doorgegeven en kun je in Laravel door middel van het definiëren van een route aangeven wat er moet gebeuren.

Ga naar de file routes/web.php en open deze. Plaats de volgende code:

```
Route::get('/welcome', function () {
    return view('welcome');
});
```

Ga nu terug naar je Laravel project in de browser en controleer of je (standaard) web pagina nu ander /welcome staat. Je hebt nu de standaard Laravel welcome pagina verhuisd van / naar /welcome .

Laten we een route toevoegen, door het volgende toe te voegen:

```
Route::get('/showlinks', function () {
    $links = \App\Link::all();
    echo "<pre>";
    print_r($links);
    echo "</pre>";
});
```

Je ziet dat $links een (Laravel) object is. Als je goed kijkt herken je de resultaten van de query en van de tabel links.

Maak nu in resources/views een nieuwe view aan. Je ziet daar de welcome.blade.php file staat. Dat is de eerste standaard pagina, die we al zagen. Maak nu een nieuwe template aan en noem die links.blade.php

Plaats nu eerst de volgende route in de web.php file.

```
Route::get('/links', function () {
$links = \App\Link::all();
    return view('links', ['links' => $links]);
});
```

Wat hier staat is het volgende:

1. als je naar de link /links gaat dan
2. haal je alle links op en zet je deze in een object, dan
3. geeft je dit object mee als value van een assiociative array aan de view 'links' en dan
4. return de output van deze view (naar de browser).

We gaan nu naar de zo net aangemaakte pagina resources/view/links.blade.php en maken de volgende code:

```
<!DOCTYPE html>
<html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">

        <title>Laravel</title>
    </head>

@foreach ($links as $link)
  <a href="{{ $link->url }}">{{ $link->title }}</a><br>
@endforeach

</html>
```

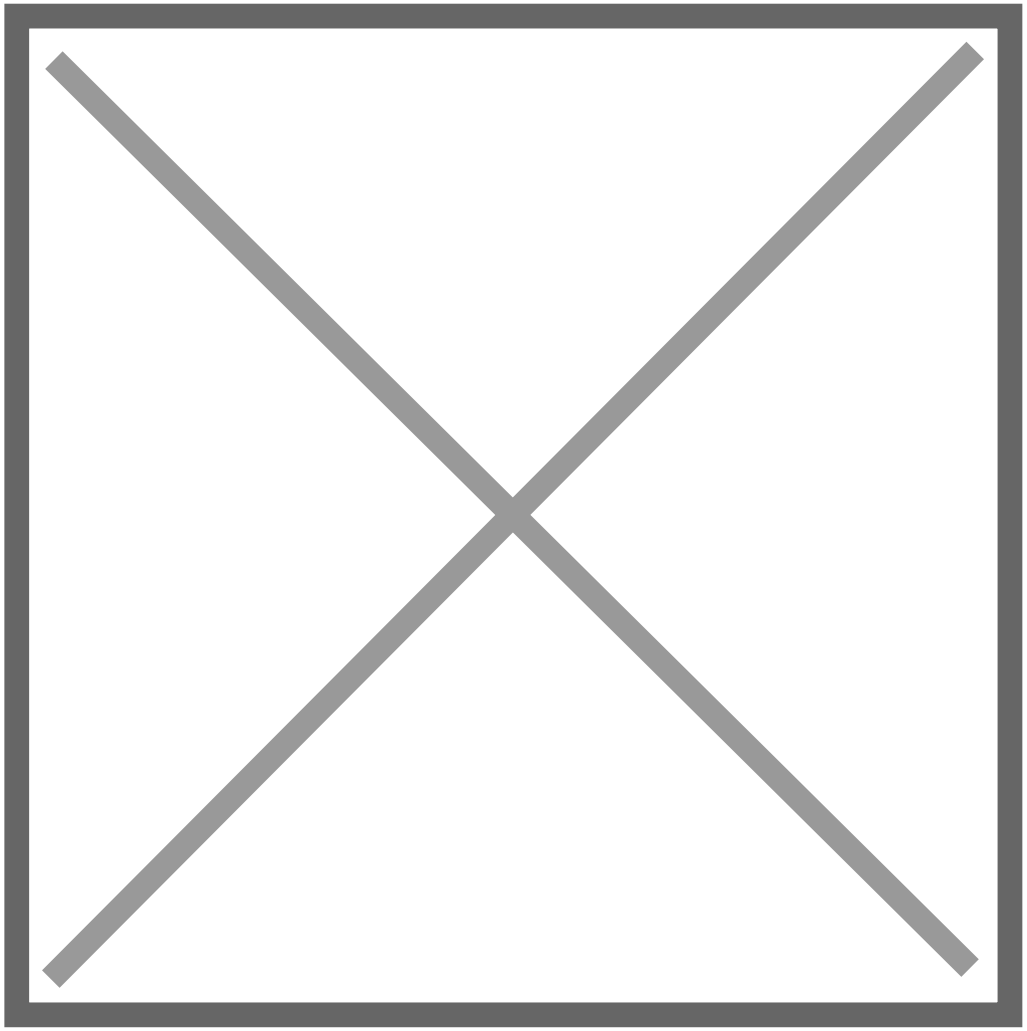
Dit is een Blade template. In principe is het HTML maar er zitten speciale statements in. Zo zie je een @foreach loop die door het object $links loopt. Een beetje zoals we eerder hebben geoefend

met assiociative arrays (2) in een tabel afdrukken.

Ook zien we in de balde template fat de url en title uit het links object worden afgedrukt. We hebben ook nog een description in de links tabel, weet je nog?

Opdracht

Pas nu de code in de blade template aan zodat de links en de description in een tabel worden afgedrukt. De output moet er als volgt komen uit te zien.



--

Create

Via een form gaan we een nieuwe link aanmaken. We gaan dus onze C van CRUD maken.

In deze les hebben we uiteindelijk ook een controler gemaakt (C van MVC) die een nieuwe link aanmaakt. Je moet zel fin de documentatie van Laravel uitzoeken hoe je een controller aanmaakt.

Aan het eind van de les kunnen we een nieuwe link aanmaken.

Nieuwe Link

Laten we eerst een eenvoudige link maken onderaan de pagina die we in de vorige les hebben gemaakt. Open de links.blade.php file en zet de volgende link onder de tabel die je in de vorige les heb gemaakt.

```
<a href="/newlink"> New </a>
```

Uiteindelijk maken we hier een menu van maar dat doen we later.

Maak een nieuwe template file en noem deze newlink.blade.php.

Opdracht 1

Maak een route in de routes/web.php file en verwijs naar de newlink.blade.php op dezelfde manier als we dat bij de welcome pagina hebben gedaan.

Plaats een eenvoudige echo "test"; in de newlink.blade.php en test of de nieuwe routering goed werkt.

Form

in de newlink.blade.php flie maken we nu een form.

```
<form action="/submitlink" method="post">
@csrf <!-- {{ csrf_field() }} -->
Title <input type="text" class="form-control" id="title" name="title" placeholder="Title"><br>
URL <input type="text" class="form-control" id="url" name="url" placeholder="URL"><br>
Description <textarea class="form-control" id="description" name="description"
placeholder="description"></textarea><br>
```

```
<button type="submit" class="btn btn-default">Submit</button>
</form>
```

Opdracht 2

Zet het form in een nette tabel; maak 3 regels en twee kolommen.

Create New

Als het form wordt ge-submit dat moet er een nieuw record komen. We moeten eerst zorgen voor de juiste routering. Het form roept namelijk /submitlink aan en via de routering kunnen we aangeven wat er dan moet gebeuren.

De routering definiëren we als volgt:

```
Route::post('/submitlink', function(Request $request) {
```

Deze routering 'vangt' de /submitlink op en roept de functie aan met als parameter de variabelen die zijn gepost.

Nu maken we een nieuwe link, zetten de waarden en slaan die op.

```
$link = new \App\Link;
$link->title = $request->title;
$link->url = $request->url;
$link->description = $request->description;
$link->save();
```

Deze code hoort eigenlijk niet in de routering file, althans het is netter om deze code ergens anders te plaatsen.

Als we de code in de routing (web.php) plaatsen dan moeten we de request module includen:

```
use Illuminate\Http\Request;
```

Laten we de code testen. Wat gebeurt er als we een nieuwe link hebben toegevoegd? Waarom zien we niets?

Controller

```
Route::post('/submitlink', 'LinksController@newlink');
```

Zoek op hoe je een controller maakt op: https://laravel.com/docs/5.7/controllers

Aan het eind gaan we terug naar de /links pagina met:

```
return redirect('/links');
```

Test de code en maak een paar nieuwe links aan. Let op dat de input nog niet wordt gevalideerd (op bijvoorbeeld maximaal aantal characters). Ook dat doen we later.

--

Delete

We gaan de Delete van de CRUD nu toevoegen.

Opdracht

Ga naar de file link.blade.php waarin we het overzicht van de links in een tabel afdrukken en voeg een kolom toe aan het eind van de regels. Zet in deze kolom:

```
<a href="/links/del/{{ $link->id }}"> Delete </a>
```

of

```
<form action="{{ url('/links/del' , $link->id ) }}" method="GET">
{{ csrf_field() }}
<button>Delete</button>
</form>
```

Beide doen hetzelfde, of moeten hetzelfde gaan doen. De eerste werkt via een delete link en de tweede werkt via een delete button. Het verschil is vooral grafisch (GUI).

In het form en in de link wordt het ID van de regel die moet worden verwijderd meegegeven aan de url.

Routing en destroy()

In de routing krijgen we de url /links/del/12 waarbij 12 het ID is van de regel die verwijderd moet worden. De routering wordt opgezet en het ID wordt als parameter van de functie gedefinieerd. Daarna kunnen we met een eenvoudig Laravel statement de regel uit de tabel halen:

```
Route::get('/links/del/{id}', function ($del_id) {
    \App\Link::destroy($del_id);
    return redirect('/links');
});
```

Je hebt nu een delete (destroy) gemaakt.

Filter

We hebben nu een 'probleem' dat de functie om een regel te verwijderen rechtstreeks vanaf de command url kan worden aangeroepen. Laten we er in ieder geval voor zorgen dat de input wordt gefilterd (=veilig programmeren!).

```
...
...
})->where('number', '[0-9]+');
```

Voeg dit toe aan de bovenstaande routering, zodat je alleen numerieke waarden accepteert. Probeer of het werkt.

Refactoring to MVC

Aan het eind van deze les heb je een control file en heb je de complete routing voor het object links opgezet. De functies Read and Delete lopen aan het eind van deze les via de controller en we hebben het raamwerk opgezet op de ander CRUD functies ook op te zetten via de controller.

We gaan onze code een beetje opnieuw bouwen, dit heet ook wel refacturing. Het doel is dat we onze code opschonen en het MVC model wat duidelijker vorm geven in onze code. Hierdoor wordt code overzichtelijker en sluiten we beter aan bij de Laravel standaard.

Routing and controllers

Tot nu te hebben we onze routing opgezet en in de routing hebben we tevens de controls opgenomen. Bijvoobeeld bij het toevoegen van een nieuwe link hebben we alle code in de routing file gestopt. Dit werkt maar is op den duur, als onze code meer wordt niet meer overzichtelijk. In onze routing houden we alleen de routing bij en de control-code gaat in een aparte file. Hiermee gaan we een duidelijk onderscheid maken tussen de routing en de controls.

We kunnen al onze CRUD routes voor het object Link in één keer aanpassen met één route:

```
Route::resource('links','LinksController');
```

Plaats deze route en plaats de andere routes voorlopig in commentaar.

Deze ene regel zorgt voor routes naar alle CRUD-functies:

Method	URI	Action (function in controller)	functie
GET	/links	index	standaard pagina voor indes
GET	/links/create	create	create form
POST	/links	store	create form post
GET	/links/{id}	show (één item)	laat één item zien
GET	/links/{id}/edit	edit	edit form
PUT/PATCH	/links/{id}	update	edit form post
DELETE	/links/{id}	destroy	delete

zie ook https://laravel.com/docs/6.x/controllers of gebruik het commando:

```
 php artisan route:list
```

In de directory app/http/Controllers staan al onze controlers en we maken een nieuwe file en noemen die LinkController.php

Tip: er is ook een artisan command om een standaard controller file te maken:

```
php artisan make:controller LinksController
```

De routing zal naar deze file verwijzen. In deze php file maken we de volgende 'standaard' code:

```
<?php
namespace App\Http\Controllers;
use App\Link;
use Illuminate\Http\Request;

class LinkController extends Controller
{
    protected function index (){
 //
    }

    protected function create(){
       //
    }

    public function edit($id){
        //
    }

    public function show($id){
    //
    }

    public function update(Request $request, $id) {
//
   }

    protected function store(Request $request){
         //
    }
```

```
    protected function destroy($del_id){
□□//
    }
}
```

Nu hebben we alle standaard methods voor de routing, we moeten nu alleen de code nog per functie invullen.

Plaats in elke control een echo die laat zien dat in welke control je zit. Dus in de index control plaats je `echo "index";`, in de create control plaats je `echo "create";` , etc.

> Check: Test alle GET functies (zie tabel hierboven) en check of de juiste controls worden aangeroepen.

Read

Als alles goed werkt dan kunnen we de code bij de index method in de liinksController invullen. Deze code hadden we al eerder gemaakt, maar die stond in de routing (web.php). Verplaats deze code naar de index method in de linksController. Onze Read van CRUD hebben we op deze manier via de linksController laten lopen

Delete

Voor de delete hebben we nu nog een aparte control in de web.php staan.

Als je in de tabel kijkt dan zie je dat de method destroy in de linksController wordt aangeroepen als je de HTTP header DELETE meesturen. Dat kan niet via een link of via een button. Dat kan alleen als je een form gebruikt. We moeten dus kiezen: bij een button of een link moeten we de route zoals we die al in web.php hebben gemaakt laten staan. Als we de delete via de route::resource willen laten lopen dan moeten we een form gebruiken.

```
<form method="POST" action="/links/{{ $link->id }}">
  {{ csrf_field() }}
  {{ method_field('DELETE') }}
  <input type="submit" class="btn btn-danger" value="Delete">
</form>
```

In dit form wordt de method DELETE in de header van het HTTP request meegestuurd. Hierdoor 'weet' de resource control dat die de method destroy moet aanroepen.

Als dit werkt gaan we de andere controls later invullen; we gaan eerst de templates aanpassen zodat alles er iets beter uit ziet.

--

Front End

In deze les gaan we naar templates kijken. We maken één master template met een standaard lay-out en we definiëren een content sections waarin we met een andere template onze content daarin plaatsen.

Templates

De templates voor het object links zetten we in de (nieuwe) directory /resources/views/links.

In deze directory maken we de volgende template files:

- create.blade.php
- edit.blade.php
- form.blade.php
- index.blade.php
- update.blade.php

We gaan nu eerst testen of alles werkt, ga naar de control index() (in de LinkController file) en verander de return waarde in `view('links.index');`

Deze verwijzing is naar de template index.blade.php in de directory links. De punt zou je dus kunnen zien als een / waarmee je ene directory aangeeft.

Ga naar de file index.blade.php en plaats daar een `echo "Blablabla...";` in . Check of het werkt. Ga hiervoor naar de links url. Deze wordt via de routing naar de controle index() gestuurd en de index() control roept de template aan en in de template wordt blablabla... afgedrukt.

> We gaan dus via de *routing* naar de *control* en vanuit de control wordt de juiste *template* aangeroepen.

Standaard template

Op al onze pagina's willen we een standaard menu, footer en andere zaken. Deze maken we één keer. Deze menu-template kunnen we dan toevoegen aan onze andere templates. In deze template maken we de standaard lay-out van al onze pagina's.

De standaard template zetten we in de (nieuwe) directory resources/vieuws/layouts. In de lay-outs directory maken we een file en noemen deze app.blade.php.

In deze app.blade.php zetten we de volgende code zoals hieronder is weergegeven.

Deze code regelt een aantal standaard zaken. Standaard scripts, fonts en (bootstrap) styles worden ingelezen. Er wordt een soort menu bar gemaakt. En als alles is geregeld dan wordt er met het commando @yield('content') de eigenlijke content geplaatst. Hoe dit werkt leggen we later in deze les uit.

```
<!doctype html>
<html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <!-- CSRF Token -->
    <meta name="csrf-token" content="{{ csrf_token() }}">

    <title>{{ config('app.name', 'Laravel') }}</title>

    <!-- Scripts -->
    <script src="{{ asset('js/app.js') }}" defer></script>

    <!-- Fonts -->
    <link rel="dns-prefetch" href="//fonts.gstatic.com">
    <link href="https://fonts.googleapis.com/css?family=Nunito" rel="stylesheet">

    <!-- Styles -->
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"
integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM"
crossorigin="anonymous"></script>

</head>

<body>
    <div id="app">
```

```
        <nav class="navbar navbar-expand-md navbar-light bg-white shadow-sm">
            <div class="container">
                <a class="navbar-brand" href="{{ url('/') }}">
                    {{ config('app.name', 'Laravel') }}
                </a>
                <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target="#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false"
aria-label="{{ __('Toggle navigation') }}">
                    <span class="navbar-toggler-icon"></span>
                </button>

                <div class="collapse navbar-collapse" id="navbarSupportedContent">
                    <!-- Left Side Of Navbar -->
                    <ul class="navbar-nav mr-auto">
                        <li class="nav-item">
                            <a class="nav-link" href="{{ route('links.index') }}">{{
__('Links') }}</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link" href="{{ url('links/create') }}">New Link</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>

        <main class="py-4">
            @yield('content')
        </main>
    </div>
</body>
</html>
```

Zet nu op de volgende code in de links/index.blade.php file:

```
@extends('layouts.app')

@section('content')
<?php echo "Hello Mr. Blablabla....!"; ?>
@endsection
```

Let op er volgt een lastig stuk, maar zorg dat je het begrijpt (vraag anders de docent).

Wat hier gebeurt is het volgende:
In de links/index.blade.php wordt een soort variabele gedefinieerd en die heet *content*. Alles tussen @section('content') en @endsection staat er in deze variabele.
Verder wordt de layouts/app.blade.php toegoevegd (included met het @extends commando). De lay-out voor de pagina wordt uit deze app.blade.php gehaald en daar waar in deze template @yield('content') staat, wordt de inhoud van de variabele content afgedrukt.

Schematisch



Als je dit begrijpt, gaan we verder en vervangen we de echo in de index.blade.php file door de volgende code. Deze code komt dus in de @section('content')

```
<div class="container">
    <table class="table table-dark table-hover">
        <thead class="thead-light">
            <tr>
                <th>#</th>
                <th>Link</th>
                <th>Description</th>
                <th></th>
                <th></th>
            </tr>
        </thead>
        <tbody>
            @foreach ($links as $link)
                <tr>
                    <td class="align-top">{{ $loop->iteration }}</td>
                    <td class="align-top"><a href="{{ $link->url }}">{{ $link->title
}}</a></td>
                    <td class="align-top">{{ $link->description }}</td>
                    <td class="align-top">
                        <a class="btn btn-warning" href="{{ url('/links/'.$link->id.'/edit')
}}">Edit</a>
                    </td>
                    <td class="align-top">
                        <form method="post" action="{{ url('/links/'.$link->id) }}"
style="display:inline">
```

```
                            {{ csrf_field() }}
                            {{ method_field('DELETE') }}
                            <button class="btn btn-danger" type="submit" onclick="return
confirm('Delete?')">Delete</button>
                        </form>
                    </td>
                </tr>
            @endforeach
        </tbody>
    </table>
</div>
```

In deze template staat veel Laravel en Bootstrap code. Wat belangrijk is, is dat je ziet dat er in een loop regels in een tabel worden gemaakt (een beetje zoals we eerder hebben geoefend met PHP) en dat er in deze regels velden uit het object Links worden afgedrukt. Kijk zelf goed naar de template of je alles begrijpt.

Je ziet dat het links object wordt gebruikt. Als je de template aanroept zonder parameter(s) dan 'weet' de template niets van het links object. Zoals we dat eerder deden zullen we in de controller ons links object moeten maken en mee moeten geven aan de aanroep naar deze template.

Weet je nog hoe we dat deden?

```
$links = \App\Link::all();
return view('links.index', ['links' => $links]);
```

Als het goed is heb je nu een opgemaakte pagina als je naar links gaat. Er zitten knoppen in voor delete en edit. In de volgende les gaan we deze knoppen we ons bezig houden met de edit and create omdat die best weel op elkaar lijken.

--

Create en Edit

We gaan de exit en create controls en bijbehorende templates maken voor onze links applicatie.

We beginnen met de edit en die is lastig. De create hadden we al gemaakt en lijkt qua form heel erg op de edit. Dus als we de edit hebben dan is de create ook bijna vanzelf af.

Aan het eind van deze les heb je een werkende CRUD applicatie.

Create

Voor de create functie gaan we de routing, control en view opzetten.

Routing

Voor de routing verwijzen we naar de view create in de links directory, we plaatsen dus de code:

```
return view('links.create');
```

in de create control.

Wrapper

De view wordt weer een beetje complex, we maken namelijk één form die we gaan gebruiken voor create en voor edit. We maken voor create en voor edit een eigen template maar roepen vanuit deze template deze ene form-template aan. Dit wordt ook wel een *wrapper* genoemd. We verpakken de form template als het ware door een andere template. De verpakking (wrapper) bepaald wat het wordt (edit of create) maar de inhoud van de verpakking (wrapper) is gelijk omdat eht form voor edit en create namelijk hetzelfde zijn.

Edit

We plaatsten in de edit template de volgende code (dit is de wrapper voor edit).

```
@extends('layouts.app')


@section('content')


<div class="container">
```

```
    <form action="{{ url('/links/' . $link->id)  }}" method="POST" enctype="multipart/form-
data">
        {{ csrf_field() }}
        {{ method_field('PATCH') }}
        @include('links.form', ['mode' => 'edit'])
    </form>
</div>

@endsection
```

Belangrijk hier is de @include, daar wordt de template links/form.blade.php included en er wordt een parameter (mode=edit) meegegven zodat het form links/form.blade.php 'weet' dat het aangeroepen is vanuit de create.

In het form.blade.php file beginnen we met:

```
{{ $mode == 'create' ? 'Create Link' : 'Modify Link' }} <br>
```

Dit is gewoon een soort if statement, je kunt dit herschrijven als volgt:

```
if ($mode=='create') {
echo "Create Link";
}else{
echo "Modify Link";
}
```

Onder de eerste regel in form.blade.php plaatsen we de voglende code:

```
<div class="form-group">
    <label for="title" class="control-label">{{'Title'}}</label>
    <input type="text" class="form-control" name="title" id="title" value="{{ isset($link-
>title) ? $link->title : '' }}">
</div>

<div class="form-group">
    <label for="url" class="control-label">{{'URL'}}</label>
    <input type="text" class="form-control" name="url" id="url" value="{{ isset($link->url) ?
$link->url : '' }}">
</div>

<div class="form-group">
```

```
    <label for="description" class="control-label">{{'Description'}}</label>
    <textarea type="text" class="form-control" name="description" id="description">{{
isset($link->description) ? $link->description : '' }}</textarea>
</div>

<input type="submit" class="btn btn-success" value="{{ $mode == 'Create' ? 'Create' : 'Update'
}}">

<a class="btn btn-primary" href="{{ url('links') }}">Cancel</a>
```

> Vraag 1, de php functie isset wordt een paar gebruikt in deze template, wat doet deze en waarom is deze nodig?

> Vraag 2, de button tekst (regel 16) is variabel. Wat kan de button tekst worden en waar hangt dat van af?

Nu gaan we naar de control kijken die wordt aangeroepen als we de edit knop indrukken. In de tabel voor alle routing kun je zien dat de standaard url voor een edit is /link/ID/edit. Zoek in de tabel weer op welke control er wordt aangeroepen. Ga naar deze control en plaats daar de volgende code:

```
$link = Link::findOrFail($id);
// echo print_r(compact('link'));
 return view('links.edit', compact('link'));
```

$link is een object en de view heeft een assiociatieve array nodig met één element link dat verwijst naar het object. Dit is wat de compact functie doet (https://www.quora.com/What-does-compact-do-in-Laravel).

Je kunt met de echo die in commentaar staat zien hoe dat werkt.

We hebben nu het juiste form en kunnen de rows van de link wijzigen, maar wat gebeurt er als je op Update drukt? In het form (de wrapper) staat iets bijzonders namelijk {{ method_field('PATCH') }} Dit zorgt ervoor dat de action van het form geen POSTof GET is maar PATCH. Zoek in de tabel op welke method daarbij hoort in de controller.

Plaats dan deze code bij deze method:

```
//dd($request);
$modifiedLink = request()->except(['_token','_method']);
```

```
//dd($modifiedLink);
Link::where('id', '=', $id)->update($modifiedLink);
return redirect('links');
```

Met de regels die in commantaar staan kan je zelf uitzoeken wat er precies gebeurt.

Als het goed is heb je nu een werkende edit functie.

De Edit Samengevat

Samengevat, de edit in vijf stappent:

Stap	Vanaf hier	Actie	Eindigt hier
1	index.blade.php	/links/{ID}/edit (via knop)	linksController@edit
2	linksController@edit	edit.blade.php	wrapper voor form.blade.php
3	form	/links/{ID} (method PATCH)	linksController@update
4	linksController@update	doet **update** en gaat naar /links	linksController@index
5	linksController@index	index.blade.php	

Create

De create code hadden we al eens gemaakt en zag er als voglt uit:

```
$link = new Link;
$link->title = $request->title;
$link->url = $request->url;
$link->description = $request->description;
$link->save();
```

Zoek in de tabel op waar deze code moet in de controller moet staan en plaats deze code. Let op dat als de code klaar is dat je weer terug wilt naar je overzichtpagina. Plaats hiervoor de juiste code op regel 6, dus na de sav().

We hoeven alleen nog maar een wrapper voor de create te maken, deze lijkt op de wrapper voor de edit, maar is net iets anders:

```
@extends('layouts.app')
```

```
@section('content')

<div class="container">
    <form action="{{ url('/links') }}" class="form-horizontal" method="post"
enctype="multipart/form-data">
        {{csrf_field()}}
        @include('links.form', ['mode' => 'create'])
    </form>
</div>

@endsection
```

De verschillen met de wrapper van de edit zijn:

1. de form action is anders want er wordt geen ID meegegeven; een nieuwe entry heeft namelijk nog geen ID.
2. de 'variabele' mode is anders zodat het form 'weet' dat het een create form en geen edit form is.

Opdracht

In het create form staat een button en de tekst is 'update'. Dit was niet de bedoeling, de tekst moest afhankelijk zijn van de wrapper die het form aanroept: bij een update zou de tekst *Update* moeten zijn en bij create zou de tekst *Create* moeten zijn. Kijk wat er fout is en los dit bugje op.

Opdrachten

We hebben nu een begin van een complete applicatie gemaakt en om ons beter thuis te voelen gaan we onze applicatie op een paar kleine punten zelf aanpassen. We gaan aan de hand van de opdrachten ook nog proberen wat beter begrip te krijgen van wat we allemaal gemaakt hebben.

Opdracht 1

~~Zorg nu zelf dat de delete buton werkt in onze links applicatie. Alle code hebben we al gemaakt we moeten het alleen op de juiste plaats zetten~~. Is al gemaakt in de voorgaande lessen.

Opdracht 2

Maak nu een tabel en zet daarin alle files die we hebben gemaakt of hebben bewerkt. Geef per file aan waar die staat en wat de functie is. Vul onderstaande tabel daarvoor aan:

File en locatie	Functie
/routes/web.php	
/app/Http/Controllers/LinkController.php	
/resources/views/links/index.blade.php	

Opdracht 3

Maak nu een tabel en beschrijf alle controls uit de LinkController.php file.{{ isset($link->description) ? $link->description : '' }}dO

Control	Functie
index()	
create()	

Opdracht 4

Vanuit welke file worden de (bootstrap) stylesheet toegevoegd aan onze code?

Opdracht 5

Waar vind je de volgende code? Leg uit wat de code doet en waarom die tussen {{ .... }} staat.

```
{{ isset($link->description) ? $link->description : '' }}
```

Opdracht 6

Waar vind je de volgende code? Leg uit wat de code doet.

```
@extends('layouts.app')
```

Opdracht 7

Waar vind je de volgende code? Leg uit wat de code doet.

```
@yield('content')
```

Opdracht 8

Maak een nieuwe view; voer de volgende stappen uit:

1. Maak een menu item en noem die 'compact'.
2. Plaats dit item tussen de items *Links* en *New Link*
3. Als je op deze link drukt verschijnt er een compact view van alle links waarbij alleen de links getoond worden (dus zeg maar de 2de kolom van het huidige overzicht.

--

Validation

In deze les leren we hoe we back-end validation kunnen opzetten met Laravel

Front End Validation

Via het keyword required kun je (met Bootstrap) aangeven dat een veld verplicht is. Hiervoor wordt Java Script gebruikt. Omdat een gebruiker Javascript in de browser kan uitzetten, kand de Fornt End Validation worden 'uitgezet'. Het is dus ook van belang dat je back End Validation implementeerd.

Back End Validation

We gaan een request control maken om de back-end validation te implementeren, dat doen we met het volgende CLI commando.

```
php artisan make:request LinkRequest
```

De php file app/Http/Requests/LinkRequests is nu aangemaakt. Open deze.

We gaan eerst naar de rules kijken en laten we de velden title en url verplicht maken. In de function rules voeren we dan het voglende in:

```
return [
  'url' => 'required|url',
  'title' => 'required
 ];
```

Verder moeten we de return value van de funtie autorize() op true zetten. met deze functie kunnen we logica inbouwen die aangeeft of de gebruiker de update mag uitvoeren of niet.

In de LinkController vervangen we de Request in de update() en in de store() in LinkRequest en vervolgens zetten we dit bovenaan in de LinkController:

```
use App\Http\Requests\LinkRequest;
```

We haalden met de functie Request de request values op, nu gebruiken we de LinkRequest functie die ervoor zorgt dat we validatie kunnen toepassen.

Test de validatie.

Fout melding

In onze form kunnen we de user nog feedback geven als de validatie een fout geeft. Dat kunnen we doen door de volgende code aan ons vorm toe te voegen.

```
@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```

(zie ook: https://laravel.com/docs/5.8/validation)

Je kunt de error messages aanpassen door in de LinkRequest file een message class te maken, met (bijvoorbeeld) de volgende code.

```
public function messages()
{
    return [
        'title.required' => 'A title is required',
        'url.required'  => 'A URL is required',
    ];
}
```

Veilig Programmeren

Je hebt nu een sterke back-end validatie toegepast. In het kader van veilig programmeren is dit de manier om je input te valideren omdat je de validatie hierbij onafhanelijk maakt van de de front end waar je als prgrammeur weinig controle over hebt.

Samengevat

Voor back-end validation hebben we de volgende stappen uitgevoerd:

1. php artisan make:request <object>Request
2. in app/Http/Request/<object>Request.php kun je nu je validatie plaatsen

3. Voeg use App\Http\Requests\<Object>Request toe in je <Object>Controller
4. Vervang de aanroep van Request aanroep in store() en Update() in <Object>Request.

-

Login (Laravel 6)

In deze les gaan we leren hoe we een login met laravel kunnen toevoegen.

Authenticatie toevoegen

We gaan in ons project een login toevoegen. Daar heeft Laravel standaard oplossingen voor. Er bestaat in Laravel zogenaamde middleware, deze software zit midden tussen het (HTTP) request en de code.

Bij oude versies van laravel konden we met één commando de authenticatie middleware modules installeren, nu moet dat in meer stappen. Omdat we al een applicatie hebben ontwikkeld zal tijdens stap 2 worden er gevraagd om de home.blade.php en app.blade.php te overschrijven. Maak eerst even kopieen van deze twee files en laat het installatie process daarna de files overschrijven. Voer de drie stappen die hieronder staan uit.

1. Instal UI standaard

```
composer require laravel/ui --dev
```

2. Create authentication UI en DB tabels.

```
 php artisan ui vue --auth
```

```
php artisan migrate
```

3. Install npm

```
npm install
```

```
npm run dev
```

Check nu of je een login pagina hebt als je naar /login gaat. Als het goed is werkt je links pagina ook nog maar moet je wel naar /links gaan. Omdat het login process twee template files heeft overschrijven gaan we die weer "repareren".

Herstellen van menu

Open de nieuwe app.blade.php file en plaats het oude menu terud op de plaats waar <!-- Left Side Of Navbar --> staat in de template.

Test nu de login door t registreren en controleer of de user ook in de users tabel is opgenomen.

Om de home page naar de links pagina te routeren, voegen we een route toe.

```
Route::get('/', 'LinkController@index' );
```

Beveiligen object Links

Door de volgende code toe te voegen aan de controller van links, LinksController.php, kun je alleen nog maar bij de links als je aangelogd bent.

```
public function __construct() {
    $this->middleware('auth');
}
```

Door het toevoegen van de authenticatie middleware aan het object links, zorgt laravel ervoor dat je eerst moet aanloggen om de controls te kunnen gebruiken.

Door de regel aan te passen in:

```
$this->middleware('auth')->except('index');
```

wordt authenticatie voor alle methods, behalve voor de index, verplicht. Je kunt dus zonder ingelogd te zijn de lijst van links zien, maar je kun de andere functies, edit, delete en dergelijke niet gebruiken zonder aan te loggen.

In plaats van *except* kun je ook *only* gebruiken. De authenticatie wordt dan alleen toegpast op de method die je specificeert.

Forms aanpassen

We gaan nu de knoppen *edit* en *delete* weghalen uit het form als de user niet is aangelogd.

Dit gaan we doen door in de index.blade.php alleen de knoppen te laten zien als je bent ingelogd.

Daarvoor hebben we de constructie:

```
@IF(Auth::check())
// laat hier de knoppen edit en delete zien
@ELSE
 // laat de knoppen niet zien, maar zorg er wel voor dat de tabel kolommen nog kloppen
@ENDIF
```

Verander de index.blade.php en laat de knoppen alleen zien als er een gebruiker is aangelogd.

--

Search

In deze les gaan we een search box toevoegen aan onze pagina.

(zie: https://medium.com/justlaravel/search-functionality-in-laravel-a2527282150b)

Search form opnemen in template

```
<form action="/search" method="POST" role="search">
    {{ csrf_field() }}
    <div class="input-group">
        <input type="text" class="form-control" name="q"
            placeholder="Search users"> <span class="input-group-btn">
            <button type="submit" class="btn btn-default">
                <span class="glyphicon glyphicon-search"></span>
            </button>
        </span>
    </div>
</form>
```

We moeten een route opnemen die naar de juiste method in de LinkController wijst.

```
Route::post(..., 'LinkController@search' );
```

Kijk in het form en vul zelf de jusite code in op de plaats waar de ... staan.

Plaats nu de juiste method in de LinkController

```
protected function search(Request $request){
    $q = $request->input('search');
    $links = Link::where('...','LIKE','%'.$q.'%')->orWhere('...','LIKE','%'.$q.'%')-
>orWhere('...','LIKE','%'.$q.'%')->get();
    return view('links.index', ['links' => $links]);
}
```

Vul de code aan door op de plaats van de ... de juiste code te plaatsen. Let op waar moet je opzoeken, welke velden?

(ToDo search moet buiten de beveliging/login komen)

Extra opdracht - count user clicks

We gaan extra functionaliteit toevoegen aan ons project. We gaan bijhouden hoe vaak er op een link wordt geclicked.

Om dit te doen gaan we eerst bedenken hoe we dit gaan implementeren. We hebben twee belangrijke vragen: wat moet er gebeuren en wanneer moet dat gebeuren?

Wat moet er gebeuren? Hoe gaan we het aantal clicks op een link bijhouden en wat moeten we daarvoor doen? Waar leggen we het vast?

Wanneer moeten we de clicks gaan tellen, op welk moment en hoe kunnen we dat doen?

Bedenk hoe je dit wilt gaan uitvoeren en maak een stappenplan. Overleg dit stappenplan met je docent.

In het stappenplan neem al je wijzigingen die je moet gaan maken op. Je beschrijft wat je gaat wijzigen, waarom je dat doet. Het hoe kun je laer uitwerken wnat hiervoor meot je waarschijnlijk wat research op internet uitvoeren.

--

Opdracht

In deze les gaan we zelf van scratch af aan een eigen CRUD app maken.

Herhaling

De stappen die we in grote lijnen hebben gevolgd zijn:

1. Niuew Laravel application maken: `laravel new <app name>`
2. Virtual Host (vhost) aanmaken in Apache `/xampp/apache/conf/extra/httpd-vhosts.conf`
3. notepad as admin `/windows/system32/drivers/etc/hosts` en voeg vhost toe
4. create controller: php artisan `make:controller <Object>Controller --resource`
5. Voeg routes toe in web.php Route:resource('<Object>',<Object>Controller');
6. Create database in phpmyadmin en update .env file met juiste database naam
7. Maak model php artisan make:model <Object> -m
8. Add colums in database/migrations file in function up()
9. Run migratie php artisan migrate en controleer in php myadmin
10. Vul tabel met php artisan tinker of met seeder (tinker is nog niet behandeld, seeder wel)
11. Route:resource wijst naar <Object>Controller, maak de index() method en haal alle elementen op (gebruik dd() om te controleren of je de juiste data krijgt.
12. Stuur de resultaten van 11 naar de template index.
13. Maak de regels uit de index template clickable en toon de details met de show() method in de controls.
14. Haal de gegevens op met de show() en maak een show.blade.php template om het item te laten zien.
15. Maak ene navigatie strucuur (menu's) met Bootstrap. Maak een 'create new' menu item.
16. Maak een create.blade.php template voor het invoeren van een een nieuwe item (Bootstrap Example Form).
17. Maak de controller voor het bewaren van een nieuw item
18. Maak een edit/modify link of button in de index template.

19. Maak een edit.blade.php en gebruik als basis de create.blade.php form dat je al hebt gemaakt.
20. Maak een delete link of button in de het scherm waarin je de details toont (uit stap 13).

ToDo Lijst

Maak een lijst met ToDo's. Een ToDo heeft een titel, een omschrijving en datum.

Toets nabesspreking

Commemtaar in HTML

```
<?php //comments ?>
of
<!-- comments -->
```

Route Toevoegen

```
Route::redirect('/l', '/links');
of
Route::get('/li', 'LinksController@index' );
of
Route::resource('li','LinksController');
```

Zoek de verschillen op.

404 Error

http://127.0.0.1:8000/links123/edit

Kijk naar de Tabel en zie dat de route /links/{id}/edit is gedefinieerd. Waarom dan toch een 404 error?

Delete button verplaatsen

Naar form.blade.php of naar edit.blade.php? Wat is het verschil?

Sorteervolgorde

https://stackoverflow.com/questions/17429427/laravel-eloquent-ordering-results-of-all

Lees dat eens door en probeer het alsnog te implementeren.

New view

1. Routering, web.php
2. Control, linksController
3. View, compact.blade.php
4. Menu item - in welke blade file, kijk nog eens naar: https://www.roc.ovh/link/120#bkmrk-schematisch

Limiet aantal items (in Compact view)

In welke file? Is het onderdeel van de *controller* of van de *view*?

Controller

Zoek de '*take* method op een collection' op in de Laravel documentatie.

View

Zoek hoe je PHP code kun toevoegen in een blade template en gebruik het PHP break commando om vroegtijdig uit een loop te breaken.

Aantal variabel maken

Maak het break commando afhanelijk van een variable die je aan de template meegeeeft.

Je moet hiervoor een aantal stappen nemen:

1. Maak een form in de compact.blade.php waarin je de waarde (laten we deze '*show*' noemen) 'show' opvraagt. Het form doet een post.

2. Maak een routering voor de post van deze form, bijvoorbeeld: Route::post('/compact', 'LinksController@showCompact');

3. Maak de controller *showCompact* en vraag hierbij de posted variabelen op door *Request $request* als parameter mee te geven zoals je dat ook hebt gedaan bij de method *newLink*.

4. De *showcompact* method is hetzelfde als de *index* method maar met dat verschil dat deze een tweede parameter aan de view mee geeft, namelijk *show*.

5. In de compact.blade.php heb je nu twee mogelijkheden: je hebt een *show* variabele meegekregen, in dat geval toon je show lines of je hebt geen *show* variabele meegekregen en je toont de standaard 5 regels. Test hierop in je template en toon het jusite aantal regels.

Count the Clicks

Dit is een lastige maar als we het in kleine stapjes opdelen valt het ook wel weer mee. Let wel dat je elke stap apart test. Het stappenplan is als volgt:

1. **Database aanpassen** met migrate. Voeg een regel toe aan de create_links_table.php en doe een migrate refresh(zie les over migrations). Vergeet de --seed optie niet om je nieuwe tabel te vullen.

   Ik denk dat het ook mogelijk moet zijn om de data te behouden, ik heb dat alleen nog niet gevonden; iemand?

2. **Template aanpassen**. Nu moeten we zorgen dat elke keer als je op een link klikt, je iet naar die link gaat maar dat de tellen clicks in de database één wordt opgehoogd. Verander dus eerst de <a href=... in de template zodat deze niet naar de link gaat maar dat er een naar een nieuwe method in de controller wordt gesprongen. Bijvoorbeeld de method *click*.

   De link geeft ook een parameter mee (via GET) en deze parameter is het id van de link. Deze variabele kan je meegeven door in de URL van de routing {linkid} op te nemen. De variable wordt dan doorgegeven aan de method in de controller.

3. **Routering aanpassen** De volgende stap is de routering. Omdat we de link via een GET gepost hebben, zul je een Route::get moeten gebruiken.
   Zoals hierboven uitgelegd moet je zorgen dat de variabale wordt doorgegeven. Als je {linkid} gebruikt in de routing dan kun je in de method deze variabale als input variabale definiëren.

4. **Controller aanpassen**. $Link is een model. Kijk maar eens in de file app/Link.php. Daar kun je zien dat Link een extentie is van het Model (van MVC). Lees deze pagina: https://stackoverflow.com/questions/33027047/what-is-the-difference-between-find-findorfail-first-firstorfail-get

   Op deze pagina staan methods beschreven van het model. Om een record te zoeken met een bepaald ID gebruiken we $link = Link::findOrFail($id); Dit hebben we ook in een van onze eerdere methods gebruikt.

IN dit artikel staat beschreven hoe we eenvoudig een kolom in de tabel kunnen omhogen: https://stackoverflow.com/questions/29816123/increment-columns-in-laravel

Er zijn meerdere methodes, je kunt ook een query uitvoeren, maar in het artikel staat een makkelijk voorbeeld.

Als de jusite kolom één is opgehoogd dan moeten we alleen nog naar de juiste link **doorlinken**. Dat kan op verschillende manieren. Vie een HTTP redirect (dat is html code die een redirect bevat). of via een Laravel redirect. Daarvoor moet je wel redirect()->away() gebruiken. Zoek zelf op waarom dat zo is en waarom je geen 'kale' redirect kan gebruiken.

5. **Template**. Als laatste stap kunnen we de clicks kolom nog toevoegen op onze index page (en/of compact page) zodat we kunnen zien dat het aantal clicks wordt opgehoogd.

--

Laravel stappenplan

Laravel One to Many Eloquent Relationship Tutorial

https://www.itsolutionstuff.com/post/laravel-one-to-many-eloquent-relationship-tutorialexample.html

1. create DB in phpmySQL
2. update .env en wijs naa rjuiste DB
3. php artisan make:migration create_cat_table --table=cat
4. edit database/migrations files
5. php artisan migrate
6. front end menu
7. routering (van menu naar lege controls CRUD)
8.

Laravel - aanpassen applicatie

Installatie / voorbereiding

Installeer de quiz applicatie door de zip te unzippen.

Laravel-project

Maak een database genaamd en quiz en importeer deze gegevens.

quiz-database

Start de applicatie op.

Login

Om aan te loggen kun je het standaard account gebruiken

- email = admin@example.com
- password = 123