



Opleiding Applicatie Ontwikkelaar

A5 - Leerlijn PHP

MVC en Testen

Auteur: Erik Mast

Datum: 20-1-2020

Crebo: 25187

Versie: 0.8

Inhoudsopgave

Overzicht	4
Voorkennis.....	4
Materialen	4
Bronnen	4
Instructies	4
Beschrijving	5
Doelen	5
Beoordeling	5
Inleiding	6
Wat is een framework?	6
Installatie	7
Configureren van Laravel	8
Werking van Laravel	10
MVC	10
Model en ORM	10
Voorbeeld	10
Controller.....	14
Routing	19
View.....	20
Downloaden	25
Object Relational Mapping.....	26
Simpele queries	26
Relationeel veld toevoegen.....	27
ORM: selecteren van gegevens voor een dropdown	34
ORM: Joins.....	34
Downloaden	35
Login systeem.....	36
Reparatie van app.blade	36
Beschermen van de app d.m.v. het loginsysteem.....	37
De handigste manier	37
Downloaden	38

A5: MVC EN TESTEN

Automatisch Testen.....	39
Simpele tests	40
User Interface Tests.....	46
Downloaden	53
Een project: het grote plaatje	54
Eindopdracht	55
Electronic Press Kit	55
Je eigen project	55
Afspraken.....	56
Bronnen	57
Algemeen.....	57
Laravel tutorials en documentatie	57
Voorbeeldcode	59

Overzicht

Level: Domein A Level 5
Duur: 4 weken
Methode: Weekplanning

Voorkennis

C3 Bootstrap
B3 Relationale Databases
B4 Normalisatie
A4 PHP MVC en testen

Materialen

- Je laptop met;
- Editor, bijvoorbeeld Kladblok, Notepad++, Atom (☺ : aanrader!)
- Webbrowser, bijvoorbeeld Internet Explorer, Firefox, Chrome
- Xampp of andere webserver

Bronnen

- Zie bijlage Bronnen

Instructies

- Hoe start je met Laravel.
- Hoe bouw je een webapp in Laravel
- Hoe test je onderdelen van je programma.

Beschrijving

In deze module leer je werken met een framework in PHP. Je gaat ook je nog meer verdiepen in MVC en testen. Alles wat je leert, ga je ook gebruiken in de eindopdracht.

Doelen

Na afronding van deze module kun je een simpele webapp maken in Laravel

Beoordeling

Deze opdracht wordt beoordeeld aan de hand van de eindopdracht.

Inleiding

Als je een webapp wilt maken in PHP ben je misschien geneigd om te beginnen met programmeren. Waarschijnlijk zul je dan veel dingen doen die je al eerder gedaan hebt, en daar ben je de eerste uren/dagen/weken mee bezig. Je moet dan bijvoorbeeld denken aan de stukken code die je nodig hebt om een menu te maken, of om gegevens te bewaren in de database, of ophalen. Maar dat is saai en misschien zelfs zinloos. Daarvoor heeft men frameworks uitgevonden.

Wat is een framework?

Een framework is een set van stukken code die standaard zaken voor je oplost. Meestal is die OOP geschreven omdat dan de code gemakkelijker te begrijpen is.

Om de uitleg begrijpelijker te maken, gaan we eerst onze eerste app maken in Laravel

Installatie

In onderstaande link staat beschreven hoe je laravel installeert.

<https://laravel.com/docs/6.x#installing-laravel>

Dat zou je bijvoorbeeld in je htdocs map van je webserver kunnen doen. In geval van xampp is dat

C:\xampp\htdocs

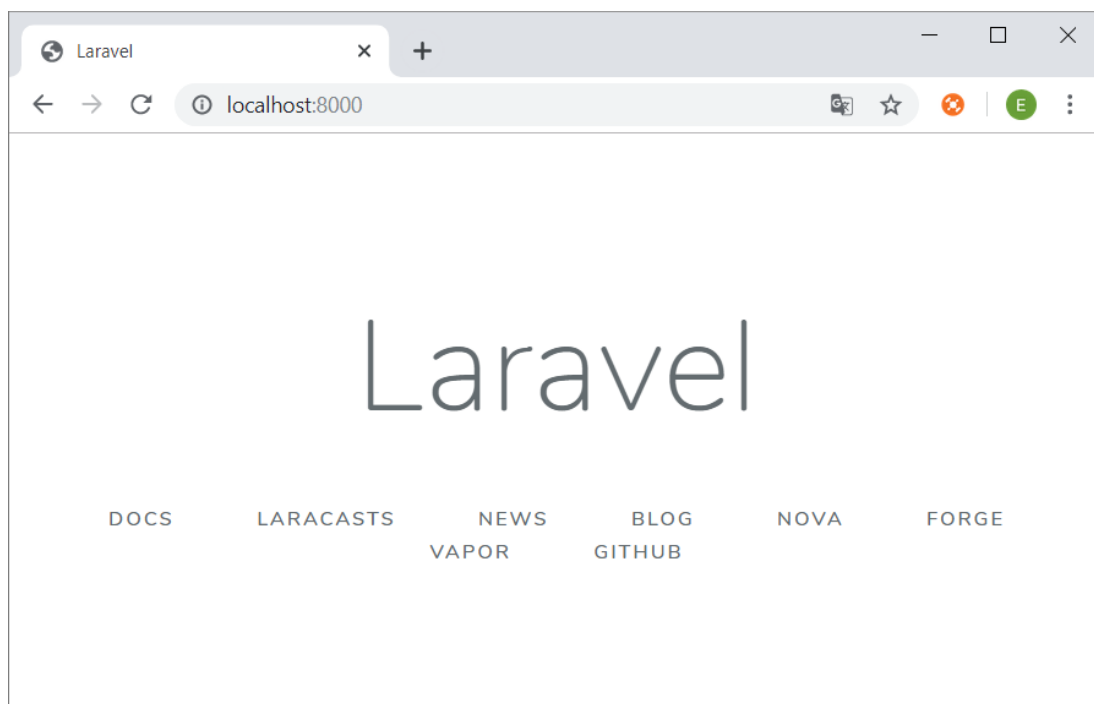
Normaal gesproken kun je dan de webserver starten op de command-prompt met

php artisan serve

Het kan zijn dat dit fout gaat:

1. Je krijgt een serverfout 500: dit betekent waarschijnlijk dat je het bestand **.env** niet hebt gemaakt (daarin zitten alle instellingen voor laravel)
Oplossing: kopieer **.env.example** naar **.env**
2. Je krijgt een melding dat de app-key niet klopt: Als je de installatiehandleiding hebt gelezen weet je wat je moet doen. (**php artisan key:generate**)

We gaan er nu even vanuit dat je een werkende laravel installatie hebt. Als je bijvoorbeeld in Chrome naar <http://localhost:8000> gaat zou je dit moeten zien:



In het CMD scherm kun je de versie zien door dit in te voeren:

php artisan --version

A5: MVC EN TESTEN

Configureren van Laravel

Voor je echt aan de slag kunt gaan met Laravel moet je nog een aantal zaken instellen. Dit gebeurt in het bestand **.env**

In dit bestand staan allerlei instellingen (je hebt hierboven al gezien dat je daar je unieke app-key kunt instellen) die interessant, handig of zelfs noodzakelijk zijn voor je web-app.

In vogelvlucht:

APP_NAME=MijnApp

Hierin kun je de naam van de app neerzetten (zonder spaties) en die kun je dan ook in beeld brengen in de html pagina. Dat kan handig zijn om bijvoorbeeld aangeven of je de test-versie gebruikt. In het hoofdstuk over testen wordt dit verder behandeld.

APP_KEY=base64:d1RGWU4qu9yf6yUcCggYnfFLS6wnf9LUGE0qS1Zk4Y4=

Hier vul je de gegenereerde unieke code in (zie boven).

APP_URL=http://localhost

Het adres van de server waar je je app op draait. Dit verschilt natuurlijk per installatie, als je je app installeert op een webserver zul je deze moeten aanpassen.

Database instellingen

Laravel ondersteunt verschillende databaseservers. In deze reader gaan we uit van MySQL maar er zijn ook nog andere mogelijkheden.

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=a5_laravel
DB_USERNAME=a5_laravel
DB_PASSWORD=a5_laravel
```

Bovenstaande instellingen spreken voor zich, en je kunt deze aanpassen aan je systeem.

Vooruitkijken op problemen

Een van de problemen waar je tegenaan loopt bij het starten van een is dat sommige velden niet goed passen in een standaard laravel installatie. Dit is het moment om dit aan te passen.

Daarvoor kun je het beste nu het bestand **/app/providers/AppServiceProvider.php** aanpassen.

Deze regels moeten worden toegevoegd:

Deze zorgen ervoor dat Strings lang genoeg zijn als je geen lengte opgeeft.

Meer info: <https://laravel-news.com/laravel-5-4-key-too-long-error>

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

public function boot()
{
    Schema::defaultStringLength(191);
}
```


A5: MVC EN TESTEN

Front-end aanpassen

We gaan gebruik maken van Bootstrap voor de vormgeving. Daarvoor moeten nog een paar zaken geregeld worden.

Allereerst moet de volgende opdracht worden uitgevoerd. Deze zal de UI component van Laravel installeren.

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

Daarna moet de volgende opdracht uitgevoerd worden:

```
php artisan ui bootstrap
```

Dit laat weinig aan de fantasie over: bootstrap wordt geïnstalleerd.

```
npm install
```

Deze opdracht installeert npm, dat is een package manager (packages zijn kant en klare pakketjes waarin aanvullende dingen geregeld kunnen worden)

```
npm run dev
```

Onthoud deze opdracht goed, deze opdracht zorgt ervoor dat het gebruik van de packages worden voorbereid (gecompileerd) en bijvoorbeeld de css klaar wordt gezet in de **public** folder.

Op dit moment is Laravel klaar om te gaan programmeren, alles is geconfigureerd en bootstrap geïnstalleerd. Maar eerst nog iets over .env.

Gebruik van .env

Elk systeem (combinatie van webserver en databaseserver) en elke instantie, zoals productie(kopie) of testversie heeft zijn eigen **.env** bestand. Het is gebruikelijk dat je die niet meeneemt in de sourcecode, je zult dit bestand dan ook niet tegenkomen in bijvoorbeeld GIT. Deze zul je dus moeten opnemen in **.gitignore**. Hierdoor kun je dus ook op je eigen computer een configuratie (of meer) maken die gelden voor jou. Dat is praktisch als je in een team werkt! En bedenk ook dat in een .env wachtwoorden staan, dat zou een veiligheidslek kunnen veroorzaken als je die zo op GitHub publiceert.

Dus in het kort, er zijn geen identieke kopieën van **.env**, want elke situatie heeft zijn eigen configuratie. Een voorbeeldbestand kan echter nooit kwaad.

Werking van Laravel

MVC

Zoals je in de vorige module (A4PHP) hebt gezien staat MVC voor Model, View en Controller. Deze MVC structuur zorgt ervoor dat de data (het model) gescheiden wordt van de logica (de controller) van de applicatie en de views (voorkant) van de applicatie.

De basis van de applicatie zijn de controllers. In de controllers worden alle functies van de applicatie afgehandeld zoals bijvoorbeeld het toevoegen, bewerken en inzien van klanten. Elke actie die uitgevoerd kan worden, krijgt zijn eigen functie binnen een controller. Als er bijvoorbeeld een pagina wordt geopend, dan wordt een betreffende controller functie aangesproken. In deze controller wordt de data dan opgehaald, verwerkt en vervolgens getoond in de view aan de gebruiker.

Dit klinkt allemaal indrukwekkend, maar laten we vooral even eerst een voorbeeld gaan bekijken.

Model en ORM

In Laravel zijn databasemodel en datamodel intensief (maar niet noodzakelijk één op één) met elkaar verbonden. Laravel kent een objectmodel voor het Model, je maakt in principe altijd gebruik van overerving in php. De overerving bestaat eruit dat je gebruikt maakt van standaard functionaliteit en alleen het gedeelte wat specifiek is voor jouw model moet je zelf programmeren. Voor meer informatie kijk even naar de link in de bronnen. Of sla module A3 er nog even op na.

Voor de database kant maakt het gebruik van een ORM systeem. ORM staat voor Object Relational Mapping. Door middel van een ORM kan data makkelijker worden uitgelezen uit de database met behulp van de Models uit de MVC structuur. De Models uit de MVC structuur zijn verbonden aan database tabellen.

Laravel maakt gebruik van **migrations**. Dat zijn php-bestanden die ervoor zorgen dat de tabellen aangemaakt worden, en dat je dat eventueel ook kunt terugdraaien. Een migratie zou ook een aanpassing van een tabel kunnen zijn, ook daarvan is het een goed gebruik om het terugdraaien van zo'n wijziging uit te werken.

Voorbeeld

Stel je wilt een webapp maken waarin je contactgegevens wilt bijhouden. Stap 1 zou dan zijn dat je een model maken en een databasetabel. In de cmd prompt in je project voer je dan deze opdracht uit, die in één keer een Model maakt en een migratiebestand klaar zet. Deze moet je dan nog wel zelf verder invullen.

```
php artisan make:model Contact --migration
```

In veel tutorials, met name oudere, zie je dan twee losse opdrachten zoals hieronder, maar de opdracht hierboven doet hetzelfde.

```
php artisan make:model Contact
```

```
php artisan make:migration create_contacts_table
```

A5: MVC EN TESTEN

Het resultaat is dat je twee bestanden hebt gekregen, **/app/Contact.php**

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Contact extends Model
{
    //
}
```

Dit gaat het Model worden.

En (in mijn geval: de datum en tijd zitten in de bestandsnaam)

/database/migrations/2019_11_15_102711_create_contacts_table.php, het bestand dat de migratie gaat worden.

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateContactsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('contacts', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('contacts');
    }
}
```

Deze moeten vervolgens ingevuld worden.

A5: MVC EN TESTEN

Uitwerken van het voorbeeld

Uitleg

In het Model zie je een vermelding van alle velden die vanuit de database ingevuld zouden kunnen worden of in de database gevuld worden vanuit het object. Dit is alles wat je nodig hebt in een simpele situatie!

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Contact extends Model
{
    protected $fillable = [
        'first_name', 'last_name', 'email', 'city', 'country', 'job_title'
    ];
}
```

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateContactsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('contacts', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->timestamps();
            $table->string('first_name');
            $table->string('last_name');
            $table->string('email');
            $table->string('job_title')->nullable();
            $table->string('city')->nullable();
            $table->string('country')->nullable();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('contacts');
    }
}
```

A5: MVC EN TESTEN

De migratie is vergelijkbaar simpel: je geeft aan welke velden je nodig hebt. Opvallend is de timestamp opdracht, die voegt een aantal velden toe, namelijk `created_at` en `updated_at`. Deze twee velden worden automatisch ingevuld op het moment dat je een respectievelijk een rij maakt of aanpast. Dat lijkt hier oninteressante informatie, maar bedenk als je met meer mensen op een systeem werkt, dat dit een manier is om bij te houden of je geen informatie van een andere gebruiker overschrijft. Als bijvoorbeeld `updated_at` nieuwer is dan de datum die in jouw object, is iemand je voor geweest, en moet je misschien actie ondernemen. Veel van dit soort problemen worden door het framework ondervangen, maar uitproberen mag natuurlijk

Doorvoeren van de eerste aanpassing

Uitvoeren van onderstaande regel maakt de tabel in de database.

```
php artisan migrate
```

In theorie kun je nu al gegevens uit de database halen of er in bewaren, maar dat zal dan op een programmeursmanier gebeuren. Gewoon uitprogrammeren. En je gebruikers willen dat vanzelfsprekend gemakkelijker!

Controller

De volgende stap is het maken van een controller. Deze definieert een aantal standaard methodes om schermen te laten zien, maar ook om objecten te bewaren in de database, aan te passen of weg te gooien. Laravel laat qua MVC hier duidelijk een controller zien die alles bepaalt.

De basiscontroller kun je als volgt maken in het Cmd scherm:

```
php artisan make:controller ContactController --resource
```

Deze opdracht maakt een standaard controller voor een CRUD systeem, en genereert alle methodes die je daarvoor nodig hebt. Ook wordt de routes aangepast (als je een “map” in de adresbalk intypt zijn de routes bepalend voor waar je in de code terecht komt, later hierover meer). Het bestand vind je terug in **/app/Http/Controllers/ContactController.php** (commentaar etc. is weggelaten voor het overzicht). Het kan overigens zijn dat de controller er iets anders uit ziet.

Maar wat doen die methodes?

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class ContactController extends Controller
{
    public function index()
    {
    }

    public function create()
    {
    }

    public function store(Request $request)
    {
    }

    public function show(Contact $contact)
    {
    }

    public function edit(Contact $contact)
    {
    }

    public function update(Request $request, Contact $contact)
    {
    }

    public function destroy(Contact $contact)
    {
    }
}
```

A5: MVC EN TESTEN

Uitleg

Van tevoren moet je weten dat als je op bovenstaande manier een CRUD systeem bouwt, er een aantal links ontstaan (verderop meer daarover)

Voor we alle methodes gaan uitwerken moeten we bovenaan het bestand een regel toevoegen. Dat is de vet gedrukte regel hieronder.

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App>Contact;

class ContactController extends Controller
{
    ...
}
```

Die regel zorgt ervoor dat je het object Contact ook kunt gebruiken in de code.

index()

Deze methode wordt aangeroepen als je bijvoorbeeld <http://localhost/contacts> aanroept in de webbrowser. Het is de bedoeling dat je web-app dan een overzicht laat zien van alle contacten.

De controller moet zorgen dat dat overzicht zichtbaar wordt, door de gewenste gegevens op te halen, en deze door te geven naar een view. De uitwerking zie je in het kader hieronder.

```
public function index()
{
    $contacts = Contact::all();

    return view('contacts.index', compact('contacts'));
}
```

De aanroep `Contact::all()` zal uit de database alle contacten halen en in een lijst teruggeven, deze stuur je mee met het scherm (**contacts.index**) dat je teruggeeft aan het systeem.

create()

Deze methode wordt aangeroepen als je <http://localhost/contacts/create> aanroept. De controller zorgt dat het gewenste scherm geopend wordt, daar kan de gebruiker de gegevens invullen en als deze op bv **OK** klinkt worden de gegevens doorgegeven aan de volgende methode.

```
public function create()
{
    return view('contacts.create');
}
```

De controller geeft een scherm (dat **contact.create** heet) terug aan het systeem, en die laat deze zien.

A5: MVC EN TESTEN

store(Request \$request)

Deze methode wordt aangeroepen als de gebruiker op **OK** heeft geklikt, in het request zitten dan de ingevulde gegevens. In het kader staat de uitwerking.

```
public function store(Request $request)
{
    $request->validate([
        'first_name'=>'required',
        'last_name'=>'required',
        'email'=>'required'
    ]);

    Contact::create($request->all());
    return redirect('/contacts')->with('success', 'Contact saved!');
}
```

Hier is te zien dat bovenin de methode dat de velden gevalideerd worden, in dit geval zijn voornaam, achternaam en email verplicht.

Vervolgens worden alle velden (die in \$request zitten) ingevuld in de **create** methode waarmee een nieuw object wordt gemaakt en bewaard in de database.

show(\$id)

Deze methode wordt aangeroepen als je bijvoorbeeld <http://localhost/contacts/3> aanroept. Voor \$id zal dan 3 ingevuld worden. Hier kun je dan de gewenste rij uit de database ophalen (dat doet `Contact::find($id)`) en vervolgens een scherm aanroepen die dat kan laten zien.

```
public function show(Contact $contact)
{
    return view('contacts.show', compact('contact'));
}
```


A5: MVC EN TESTEN

edit(\$id)

Deze methode wordt aangeroepen als je bijvoorbeeld <http://localhost/contacts/3/edit> aanroept.

```
public function edit(Contact $contact)
{
    return view('contacts.edit', compact('contact'));
}
```

Het is de bedoeling dat de gewenste rij (in dit geval natuurlijk met \$id = 3) opgehaald wordt uit de database. Laravel regelt dat.

Vervolgens wordt het scherm teruggegeven aan het systeem, samen met het object dat uit de database gehaald is. In dat scherm kan de gebruiker de gegevens (indien gewenst) in de input-velden van het scherm invullen. Als de gebruiker op bv. **OK** klikt dan wordt de onderstaande methode aangeroepen.

update(Request \$request, \$id)

Hier komt de code terecht als in het edit-scherm op **OK** geklikt is. In \$request zitten de ingevulde velden.

```
public function update(Request $request, Contact $contact)
{
    $request->validate([
        'first_name'=>'required',
        'last_name'=>'required',
        'email'=>'required'
    ]);

    $contact->update($request->all());

    return redirect('/contacts')->with('success', 'Contact updated!');
}
```

Hier is te zien dat bovenin de methode dat de velden gevalideerd worden, net als in store. Daarna wordt met \$contact->update() het object aangepast en bewaard in de database. Tot slot wordt aan het systeem een link teruggegeven aan het systeem, met een melding.

Noot: we komen dus weer in de methode index() terecht, want die luistert naar

<http://localhost/contacts>

A5: MVC EN TESTEN

destroy(\$id)

Deze methode wordt aangeroepen als de gebruiker een rij weg wil gooien. De methode zorgt ervoor dat bijvoorbeeld de rij met \$id=3 uit de database wordt verwijderd. Dit gebeurt op de achtergrond door naar <http://localhost/contacts> een DELETE request te sturen met het id.

```
public function destroy(Contact $contact)
{
    $contact->delete();
    return redirect('/contacts')->with('success', 'Contact deleted!');
}
```

A5: MVC EN TESTEN

Routing

Routing is het proces dat ons systeem ervoor zorgt dat aanroepen vanuit de web-browser (dus bijvoorbeeld naar <http://localhost/contacts> of <http://localhost/3/edit> op de goede plek in de code terecht komen. Dus de routing is min of meer onderdeel van de controller in het MVC.

Om onze gegenereerde controller (die we verder aangevuld hebben daarna) te laten werken in de routing moet er nog een aanpassing gedaan worden. Daarvoor moet je **routes/web.php** aanpassen.

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

Route::resource('contacts', 'ContactController');
```

De vet gedrukte regel moet je toevoegen.

Uitleg

De bovenste regels zorgen ervoor dat als je naar <http://localhost/> navigeert, je altijd het scherm met de naam “welcome” te zien krijgt.

Deze regel zorgt ervoor dat de controller aangemeld wordt in het systeem, maar in dit geval is dat een speciale manier: de controller wordt in één keer aangemeld voor een aantal routes en bovendien snapt het systeem dan ook dat voor {contact} een id ingevuld in de werkelijke route:

Form methode	“map”	Uitleg
GET	/contacts	Wordt doorgestuurd naar de <code>index()</code> methode
GET	/contacts/create	Wordt doorgestuurd naar de <code>create()</code> methode
POST	/contacts	Wordt doorgestuurd naar de <code>store()</code> methode
GET	/contacts/{contact},	Wordt doorgestuurd naar de <code>show()</code> methode. Bv http://localhost/contact/3
GET	/contacts/{contact}/edit	Wordt doorgestuurd naar de <code>edit()</code> methode. Bv http://localhost/3/edit
PUT/PATCH	/contacts/{contact}	Wordt doorgestuurd naar de <code>update()</code> methode
DELETE	/contacts/{contact}	Wordt doorgestuurd naar de <code>destroy()</code> methode

A5: MVC EN TESTEN

View

We hebben nu een half systeem gebouwd, maar we kunnen nog niets zien. Dit is het moment om te focussen op de View van het MVC. Daarvoor heeft Laravel een sjabloon (template) systeem aan boord, die op Blade is gebaseerd. Deze techniek komt in vele andere talen terug, zoals bijvoorbeeld Angular en NodeJS.

Een View in Laravel is relatief simpel, er worden gegevens door het systeem meegegeven en door Blade kun je die gegevens in voegen in de HTML bestanden die doorgegeven worden aan de browser.

Starten met Blade

De eerste view die we gaan bespreken is de hoofdtemplate. Die zorgt ervoor dat het scherm een standaard opbouw krijgt en alle schermen er min of meer hetzelfde uitzien. Deze kun je plaatsen in **/resources/views/layouts/app.blade.php**.

Let op dit is niet de volledige tekst!

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{{ config('app.name', 'Laravel') }}</title>
    <link href="{{ asset('css/app.css') }}" rel="stylesheet" type="text/css" />
  </head>
  <body>
    <div class="container">
      @yield('content')
    </div>
    <script src="{{ asset('js/app.js') }}" type="text/js"></script>
  </body>
</html>
```

Dit is de template van je applicatie. Het hoofdgedeelte komt op de plek van @yield('content') te staan, daar wordt dus straks het 'middengedeelte' van de pagina ingevuld.

Verder is natuurlijk opvallend dat er dubbele haakjes gebruikt worden. Alles wat tussen {{ en }} staat, wordt gezien als een methode aanroep naar het Laravel systeem. Of een php variabele die vanuit de controller komt, of een php variabele die je in dit bestand geïntroduceerd hebt.

Ook zie je hier dat de methode asset('css/app.css') gebruikt wordt. Deze methode zet de bestandsnaam die je invult naar de correcte link voor jouw installatie. Dus dat zou in de webpagina eruit kunnen zien als <http://localhost:8000/css/app.css>. En die kan gevonden worden in de map **public**. Hij wordt gegenereerd door **npm run dev**!

A5: MVC EN TESTEN

De eerste echte pagina die gemaakt wordt, is de pagina voor toevoegen van een contact. De naam van dit bestand is **/resources/views/contacts/create.blade.php**

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

@section('content')
<div class="row">
  <div class="col-sm-8 offset-sm-2">
    <h1 class="display-3">Contact toevoegen</h1>
    <div>
      @if ($errors->any())
        <div class="alert alert-danger">
          <ul>
            @foreach ($errors->all() as $error)
              <li>{{ $error }}</li>
            @endforeach
          </ul>
        </div><br />
      @endif
      <form method="post" action="{{ route('contacts.store') }}">
        @csrf
        <div class="form-group">
          <label for="first_name">Voornaam:</label>
          <input type="text" class="form-control" name="first_name"/>
        </div>

        <!-- hier staat eigenlijk meer -->
        <button type="submit" class="btn btn-primary">Toevoegen</button>
      </form>
    </div>
  </div>
</div>
@endsection
```

Uitleg

@extends('layouts.app')

Dit betekent dat dit scherm overerft van het app scherm (app.blade.php) in de map layouts (resources\views\layouts).

@section('content')

Op de plek waar in app.blade.php @yield('content') staat wordt het stuk ingevuld dat tussen @section('content') en @endsection staat in **create.blade.php**.

@if

Je kunt variabelen gebruiken die meegegeven worden vanuit de controller.

@endif

Dit is het einde van het stuk code dat uitgevoerd wordt. Zie het maar als sluitende haakjes.

@csrf

Een opdracht om in één keer je formulier te beschermen tegen cross-site request forgery. Dus dit is eigenlijk een opdracht die je in elk formulier moet zetten.

@foreach

@endforeach

Alles wat tussen deze twee tags staat, wordt herhaald in een for loop. Dit wordt gedaan totdat alle elementen uit de lijst verwerkt zijn. Dus hier: alle foutmeldingen worden opgehaald en in een element gezet.

View: overzicht van alle contacten

Het volgende scherm dat we gaan toevoegen is het overzichtsscherm voor alle contacten. De code daarvoor komt in `/resources/views/index.blade.php`. Onderstaand bestand is wat ingekort om het overzichtelijk te houden.

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

@section('content')
<div class="row">
<div class="col-sm-12">
    <h1 class="display-3">Contacten</h1>

    <table class="table table-striped">
        <thead>
            <tr>
                <td>ID</td>
                <td>Naam</td>
                <!-- hier stond meer -->
                <td colspan = 2>Actions</td>
            </tr>
        </thead>
        <tbody>
            @foreach($contacts as $contact)
                <tr>
                    <td>{{ $contact->id }}</td>
                    <td>{{ $contact->first_name }} {{ $contact->last_name }}</td>
                    <!-- hier stond meer -->
                    <td>
                        <a href="{{ route('contacts.edit', $contact->id) }}"
                            class="btn btn-primary">Aanpassen</a>
                    </td>
                    <td>
                        <form action="{{ route('contacts.destroy', $contact->id) }}"
                            method="post">
                            @csrf
                            @method('DELETE')
                            <button class="btn btn-danger" type="submit">Verwijderen</button>
                        </form>
                    </td>
                </tr>
            @endforeach
        </tbody>
    </table>
</div>
</div>
@endsection
```

@foreach

In het vorige scherm is dit al besproken. Hier gaan we door de lijst met contacten heen, en zetten we de gewenste gegevens in de tabel. Ook hier geldt weer: alles wat tussen `{{` en `}}` staat, wordt gezien als een variabele of een methode aanroep.

@method('DELETE')

Speciale aanroep voor het weggooien van een rij. Er voor zie je een form staan, waarin je post naar een link die opgehaald met `{{ route('contacts.destroy', $contact->id) }}`. En door deze `@method` wordt de post 'omgebouwd' naar een DELETE.

View: Wijzigen van een contact

De view die we gaan gebruiken voor het wijzigen van een contactpersoon, is in grote lijnen het zelfde als de view voor het toevoegen van een persoon. De view komt te staan in

/resources/views/edit.blade.php

```
@extends('layouts.app')
@section('content')
<div class="row">
    <div class="col-sm-8 offset-sm-2">
        <h1 class="display-3">Contact aanpassen</h1>

        @if ($errors->any())
            <div class="alert alert-danger">
                <ul>
                    @foreach ($errors->all() as $error)
                        <li>{{ $error }}</li>
                    @endforeach
                </ul>
            </div>
            <br />
        @endif
        <form method="post"
            action="{{ route('contacts.update', $contact->id) }}">
            @method('PATCH')
            @csrf
            <div class="form-group">

                <label for="first_name">Voornaam:</label>
                <input type="text" class="form-control" name="first_name"
                    value="{{ $contact->first_name }}" />

            </div>

            <!-- hier stond meer -->

            <button type="submit" class="btn btn-primary">Aanpassen</button>
        </form>
    </div>
</div>
```

Uitleg

Het enige wat hier nog extra uitleg behoeft (de rest staat namelijk ook al bij de uitleg van het toevoegscherm), is hoe de action zich gedraagt. Het ID wordt doorgegeven aan de update methode, de rest van invoervelden komen in die methode terecht via het request-object.

A5: MVC EN TESTEN

View: overzicht van een contact

Tot slot laten we nog even de uitwerking van het overzicht scherm zien. Met de huidige kennis hoeft deze niet meer uitgelegd te worden.

```
@extends('layouts.app')
@section('content')
<div class="row">
    <div class="col-sm-8 offset-sm-2">
        <h1 class="display-3">Contact bekijken</h1>

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

        <div>
            <a style="margin: 19px;" href="{{ route('contacts.index') }}"
                class="btn btn-primary">Overzicht</a>
        </div>

        <table class="table table-striped">
        <tbody>
            <tr>
                <td>Voornaam:</td>
                <td>{{ $contact->first_name }}</td>
            </tr>
            <!-- hier stond meer -->
        </tbody>
        </table>
    </div>
</div>
```

Downloaden

Je kunt alle voorbeeldcode tot nu downloaden van:

<https://github.com/DrentheCollege/A5PHPVoorbeeld1.git>

Object Relational Mapping

Naast de MVC structuur waarin Laravel opgebouwd is, beschikt Laravel ook over een ORM. In Laravel kan je door middel van het aanroepen van de Model klasse een query uitvoeren. Bijvoorbeeld:

```
Contact::where(company_id, 1)->get();
```

In deze voorbeeld query worden alle contacten opgehaald die behoren bij de company met een ID van 1. Dit zorgt ervoor dat query's schrijven een stuk eenvoudiger wordt. Echter ontkom je er soms niet aan om een database query handmatig uit te schrijven voor complexe datastructuren.

Om het geheel overzichtelijker te maken wordt gebruik gemaakt van meerdere Models, views en controllers en is er over het algemeen voor elke database tabel, een model en een controller. Hierdoor wordt er onderscheid gemaakt tussen de verschillende resources die beschikbaar zijn en blijft de code overzichtelijker en beter te onderhouden voor als er iets niet goed gaat of er een update naar wensen van de klant moet worden door gevoerd.

Verschillende resources kunnen onderling ook relaties hebben. Een gebruiker van de applicatie kan meerdere contacten hebben, terwijl een contact ook meerdere gebruikers kan hebben. Dit wordt ook wel een veel op veel relatie genoemd. Daarnaast kan het bedrijf meerdere contacten hebben terwijl een contact maar bij één bepaalde bedrijf kan horen. Dit wordt een één op veel relatie genoemd.

Ook deze relaties tussen de database tabellen kunnen worden gedefinieerd in Laravel in de Models waardoor deze ook gemakkelijker toegankelijk zijn. In het model voor de gebruikers wordt de relatie gelegd met het model voor de facturen en andersom wordt dit ook gedaan. Om voor een gebruiker alle facturen op te halen kan in de code de volgende aanroep worden gedaan:

```
$contacts = $user->contacts;
```

Dit is praktisch hetzelfde als:

```
$contacts = Contact::where('user_id', $user->id)->get()
```

Bovenstaande code is langer en wordt er een extra query uitgevoerd omdat er geen gebruik wordt gemaakt van 'Eager Loading'. Dit zorgt voor een langere laadtijd van de betreffende pagina

Simpele queries

Met een Laravel Model kun je allerlei simpele queries uitvoeren.

Zoals:

```
$contact = Contact::find(14);
```

Dit levert het contact op waarvan het id = 14.

```
$contacts = Contact::all();
```

Dit levert een lijst met alle contacten op

```
$contacts = Contact::where('id', '>', '4')
```

Deze geeft alle contacten terug met een id >4

A5: MVC EN TESTEN

En als je ze sorteren wilt op voornaam is dit een voorbeeld:

```
$contacts = Contact::where('id', '>', '4')->orderBy('firstname','asc');
```

In de bronnen vind je een link naar de documentatie van Laravel hierover, en er zijn natuurlijk meer bronnen te vinden.

Relationeel veld toevoegen

We gaan een bedrijfsnaam toevoegen aan contact. En omdat we weten dat er meer contacten bij hetzelfde bedrijf kunnen werken, maken we een nieuwe tabel, object etc.

Database en model

We beginnen met de volgende opdracht in het cmd scherm

```
php artisan make:model Company --migration
```

Deze actie leveren twee bestanden op:

app/Company.php

en

database/migrations/2019_12_08_132914_create_companies_table.php (in dit geval, de datum en tijd van creatie zit immers in de naam)

A5: MVC EN TESTEN

Deze zien er (na invullen van het naam veld) als volgt uit:

database\migrations\2019_12_08_132914_create_companies_table.php

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateCompaniesTable extends Migration
{
    public function up()
    {
        Schema::create('companies', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->string('name');
            $table->timestamps();
        });
    }
}
```

app/Company.php

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Company extends Model
{
    protected $fillable = [
        'name'
    ];
}
```

Deze kunnen we in onze database migreren door de volgende opdracht:

php artisan migrate

A5: MVC EN TESTEN

Oefening

Zorg dat het Company Model ook aangepast kan worden. Dus maak een **Controller**, voeg deze toe aan de **routing** en maak de **schermen**. Zoals hierboven gedaan is met Contact.

Toevoegen aan de bestaande tabel

Maar dat is niet genoeg, we willen ook graag dat de tabel gekoppeld wordt aan contacts. Daarvoor hebben we nog een migratie nodig. Die kunnen we genereren met:

```
php artisan make:migration alter_contacts_table
```

Deze actie levert weer een migratie op (in dit geval **database\migrations\2019_12_08_135352_alter_contacts_table.php**) en daarin gaan we de tabel aanpassen. Deze ziet er uit zoals hieronder, om te zorgen dat de code compact blijft, zijn de commentaren hier weggehaald:

```
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class AlterContactsTable extends Migration
{
    public function up()
    {
        Schema::table('contacts', function (Blueprint $table) {
            $table->unsignedBigInteger('company_id');
            $table->foreign('company_id')
                ->references('id')->on('companies');
        });
    }

    public function down()
    {
        Schema::table('contacts', function (Blueprint $table) {
            $table->dropForeign('contacts_company_id_foreign');
            $table->dropColumn('company_id');
        });
    }
}
```

Uitleg

In de methode up() maak je het nieuwe veld aan. unsignedBigInteger() zorgt ervoor dat je een integer veld maakt die geschikt is voor referentie naar een andere tabel, dit veld is in dit voorbeeld verplicht.

De methode down() wordt gemaakt om ervoor te zorgen dat je de migratie ook terug kunt draaien, bijvoorbeeld omdat je naar een vorige versie van je software zou willen. Of als de migratie niet zo gaat als je zou willen.

Toevoegen aan het model

We hebben nu een nieuwe tabel gemaakt en die in de database gekoppeld aan de bestaande contact tabel. Dit betekent onder andere dat je daar alleen maar een regel uit de companies tabel kunt kiezen die ook daadwerkelijk bestaat in de companies tabel. Dit heet **referentiele integriteit**, en is uitvoerig behandeld in module B3.

Maar je wilt er ook gebruik van maken in het Contact model. Je wilt de naam van het bedrijf ook afdrukken op het scherm. Daarvoor moet je in het Contact model de volgende regels (die in vet gedrukt staat) toevoegen.

```
class Contact extends Model
{
    public function company(){
        return $this->belongsTo(Company::class, 'company_id');
}
}
```

Daarna kun je in **resources\views\contacts\index.blade.php** de volgende regels toevoegen (we hebben voor het overzicht een groot aantal regels weggehaald):

```
<!-- weggelaten voor overzicht -->
<div class="row">
<div class="col-sm-12">
    <h1 class="display-3">Contacten</h1>
    <table class="table table-striped">
        <thead>
            <tr>
                <td>ID</td>
                <td>Naam</td>
                <td>Bedrijf</td>
                <td>Email</td>
                <!-- weggelaten voor overzicht -->
            </tr>
        </thead>
        <tbody>
            @foreach($contacts as $contact)
                <tr>
                    <td>{{ $contact->id }}</td>
                    <td>{{ $contact->first_name }} {{ $contact->last_name }}</td>
                    <td>{{ $contact->company->name }}</td>
                    <td>{{ $contact->email }}</td>
                    <!-- weggelaten voor overzicht -->
                </tr>
            @endforeach
        </tbody>
    </table>
<!-- weggelaten voor overzicht -->
```

Uitleg

De vetgedrukte regels in index.blade.php moeten toegevoegd worden, die zorgen ervoor dat je de naam van het bedrijf te zien krijgt.

Toevoegen aan de schermen van contacts

Voor je verder gaat met het aanpassen van de schermen moet je de volgende opdracht uitvoeren op de command-line. Deze zorgt ervoor dat de laravelcollective bibliotheek geïnstalleerd wordt. Deze is handig voor onder andere het maken van dropdown boxjes die gevuld worden vanuit de database. Gemak voor alles.

```
composer require laravelcollective/html
```

Vervolgens moeten een aantal schermen aangepast worden.

resources\views\contacts\create.blade.php

```
<!-- weggelaten voor overzicht -->
    <form method="post" action="{ route('contacts.store') }">
        @csrf
        <div class="form-group">
            <label for="first_name">Voornaam:</label>
            <input type="text" class="form-control" name="first_name"/>
        </div>
    <!-- weggelaten voor overzicht -->

        <div class="form-group">
            <label for="company">Bedrijf</label>
            {!! Form::select('company_id', $companies, null,
                ['placeholder' => 'Kies een bedrijf...',
                 'class' => 'form-control']) !!}
        </div>

    <!-- weggelaten voor overzicht -->

        <button type="submit" class="btn btn-primary">Toevoegen</button>
    </form>
</div>
</div>
</div>
@endsection
```

Uitleg

De vetgedrukte regels in het midden zorgen ervoor dat je een dropdownbox krijgt, die een placeholder heeft en bovendien bewaard wordt in het veld **company_id**.

A5: MVC EN TESTEN

Het volgende scherm dat we gaan aanpassen is het edit-scherm:

`resources\views\contacts\edit.blade.php`

```
<!-- weggelaten voor overzicht -->
<form method="post"
      action="{{ route('contacts.update', $contact->id) }}">
    @method('PATCH')
    @csrf
    <div class="form-group">
        <label for="first_name">Voornaam:</label>
        <input type="text" class="form-control" name="first_name"
              value="{{ $contact->first_name }}" />
    </div>

    <!-- weggelaten voor overzicht -->
    <div class="form-group">
        <label for="company">Bedrijf</label>
        {!! Form::select('company_id', $companies,
                        $contact->company_id, ['class' => 'form-control']) !!}
    </div>

    <!-- weggelaten voor overzicht -->

    <button type="submit" class="btn btn-primary">Aanpassen</button>
</form>
```

Uitleg

De vetgedrukte regels in het midden zorgen ervoor dat je een dropdownbox krijgt, die een placeholder heeft en bovendien bewaard wordt in het veld **company_id**. Wat verder opvalt ten opzichte van het vorige scherm, is dat de placeholder weggehaald is (want immers was het veld al ingevuld) en dat `$contact->company_id` als derde parameter is ingevuld i.p.v. *null*. En dat is omdat bij een editscherm het bedrijf is ingevuld, en bij een toevoeging is deze in eerste instantie natuurlijk niet ingevuld.

Contact model aanpassen

Maar ook het model moet aangepast worden. De schermen verwachten nl een variabele `$companies` waar alle bedrijven in zitten, en die moeten we meegeven vanuit de controller.

Dus in `app\Http\Controllers\ContactController.php` worden nog een paar aanpassingen gedaan:

```
<?php
// hier stond meer...
class ContactController extends Controller
{
    // hier stond meer...
    public function create()
    {
        $companies = Company::pluck('name', 'id');
        return view('contacts.create', compact('companies'));
    }
    public function store(Request $request)
    {
        $request->validate([
            'first_name'=>'required',
            'last_name'=>'required',
            'email'=>'required',
            'company_id'=>'required'
        ]);

        Contact::create($request->all());
        return redirect()->route('contacts.index')
            ->with('success', 'Contact saved!');
    }

    // hier stond meer...
    public function edit(Contact $contact)
    {
        $companies = Company::pluck('name', 'id');

        return view('contacts.edit', compact('contact', 'companies'));
    }
    public function update(Request $request, Contact $contact)
    {
        $request->validate([
            'first_name'=>'required',
            'last_name'=>'required',
            'email'=>'required',
            'company_id'=>'required'
        ]);

        $contact->update($request->all());

        return redirect('/contacts')->with('success', 'Contact
updated!');
    }

    // hier stond meer...
}
```

A5: MVC EN TESTEN

Uitleg

In de store() en update() methode is het veld 'company_id'=>'required' aan de validatie toegevoegd. Immers het bedrijf is een verplicht veld.

En verder wordt er gebruik gemaakt van de pluck() methode van het object (ps: elke controller die overerft van het standaard Laravel controller heeft deze methode). pluck() levert een array met het tweede veld als sleutel en het eerste veld als waarde.

ORM: selecteren van gegevens voor een dropdown

Pluck() kun je ook gebruiken om de inhoud van een ORM-query in een dropdown te zetten, hierboven hebben we dit gedaan:

```
$companies = Company::pluck('name', 'id');
```

Maar je zou ook dit kunnen doen:

```
$companies = Company::orderBy('name', 'desc')->pluck('name', 'id');
```

Dit zorgt ervoor dat je de bedrijven in omgekeerde alfabetische volgorde in de dropdownbox krijgt. Daaruit kunnen we concluderen dat de eerste regel eigenlijk geen goed idee is, en dus kunnen we in dit geval beter dit doen (omdat de meeste mensen het zo verwachten, gesorteerd op alfabet)

```
$companies = Company::orderBy('name', 'asc')->pluck('name', 'id');
```

ORM: Joins

In Laraval kun je ook joins doen om bijvoorbeeld de informatie op een andere manier te presentatie dan bijvoorbeeld in een standaard contacten lijst. Dit staat ook beschreven in de documentatie van Laravel maar hier geven we een paar voorbeelden. **Let op: bij deze voorbeelden is geen rekening gehouden met het feit dat het bedrijf verplicht is in onze app!**

Voorbeeld 1: alle contacten plus bedrijven

Stel je wilt alle contacten zien plus de namen van de bedrijven waar deze werken. In SQL zou je dat dit doen (je krijgt alleen contacten te zien waarbij het bedrijf is ingevuld)

```
select * from contacts
join companies on contacts.company_id = companies.id
```

In Eloquent (de ORM van Laravel) ziet dat in code er als volgt uit:

```
$contacts =
    Contact::join('companies', 'contacts.company_id', '=', 'companies.id')
->get();
```

A5: MVC EN TESTEN

Voorbeeld 2: alle contacten en eventueel de bedrijfsnaam

Alle contacten en als ze bij een bedrijf werken ook de naam en anders geen bedrijfsnaam:

```
select * from contacts  
left join companies on contacts.company_id = companies.id
```

Dat levert deze code op:

```
$contacts =  
Contact::leftJoin('companies', 'contacts.company_id', '=', 'companies.id')  
->get();
```

Voorbeeld 3: wie werkt bij welk bedrijf

Als je alle bedrijven gesorteerd wilt zien op alfabet, en dan vervolgens wie bij dat bedrijf werkt. Deze query zal alleen contacten laten zien waar het bedrijf is ingevuld

```
select * from companies  
join contacts on contacts.company_id = companies.id  
order by companies.name
```

Dat wordt:

```
$companies =  
Company::join('contacts', 'companies.id', '=', 'contacts.company_id')  
->get();
```

Voorbeeld 4: alle eventuele bedrijven met contacten

Als laatste voorbeeld: stel je wilt alle bedrijven gesorteerd op bedrijfsnaam met de contacten, maar ook de contacten zonder een bedrijf.

```
select * from companies  
right join contacts on contacts.company_id = companies.id  
order by companies.name
```

In Eloquent wordt de code als volgt:

```
$companies =  
Company::rightJoin('contacts', 'companies.id', '=', 'contacts.company_id')  
->get();
```

Downloaden

De code die we tot nu hebben gemaakt kun je ook downloaden op:

<https://github.com/DrentheCollege/A5PHPVoorbeeld2.git>

Login systeem

Laravel heeft standaard al een inlogsysteem aan boord, al moet je wel een paar zaken configureren voor je er echt gebruik van kunt maken. We gaan er vanuit dat je je Laravel installatie bijgehouden hebt tot hier, of hem heb samengesteld vanuit github.

Voer deze opdracht uit op de commandprompt, deze zal de originele app.blade overschrijven, maar dat gaan we verderop repareren:

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

En daarna nog

```
npm install  
npm run dev
```

Daarna kun je inloggen (eerst moet je je wel registreren natuurlijk), al doet de rest van de website daar nog niet veel mee. Het enige verschil dat je ziet is dat je bijvoorbeeld logout links zou kunnen krijgen of juist een registratie link.

Reparatie van app.blade

Allereerst moeten de links naar de contact en bedrijf pagina's worden hersteld, dat kun je doen door in **resources\views\layouts\app.blade.php** dit

```
<!-- Left Side Of Navbar -->  
<ul class="navbar-nav mr-auto">  
  
</ul>
```

te vervangen door dit:

```
<!-- Left Side Of Navbar -->  
<ul class="navbar-nav mr-auto">  
  <li class="nav-item">  
    <a class="nav-link" href="{{ route('contacts.index') }}">Contacten</a>  
  </li>  
  <li class="nav-item">  
    <a class="nav-link" href="{{ route('companies.index') }}">Bedrijven</a>  
  </li>  
</ul>
```

En als je de layout van de webpagina weer net zo wilt hebben als eerst, moet je ook dit:

```
<main class="py-4">
```

vervangen door dit:

```
<main class="container">
```

Beschermen van de app d.m.v. het loginsysteem

Nu hebben we een loginsysteem maar het lijkt niets uit te maken of je ingelogd bent of niet. Om het loginsysteem ook daadwerkelijk functioneel te maken, moeten er een paar aanpassingen gedaan worden.

In `routes/web.php` kun je dit

```
Route::get('/', function () {  
    return view('welcome');  
});  
  
Route::resource('contacts', 'ContactController');  
Route::resource('companies', 'CompanyController');  
  
Auth::routes();  
  
Route::get('/home', 'HomeController@index')->name('home');
```

aanpassen zodat er dit komt te staan:

```
Route::get('/', function () {  
    return view('welcome');  
});  
  
Auth::routes();  
  
Route::group(['middleware' => ['auth']], function() {  
    Route::resource('contacts', 'ContactController');  
    Route::resource('companies', 'CompanyController');  
    Route::get('/home', 'HomeController@index')->name('home');  
});
```

Uitleg

Alles wat tussen {} staat op de regel `Route::group` wordt 'beschermd' door een login scherm. Of te wel, alle links naar contacten en bedrijven kun je pas gebruiken als je ingelogd bent.

Dat kun je natuurlijk pas testen als je jezelf geregistreerd hebt, maar dat is eenvoudig in dit geval.

Voor meer informatie:

<https://laravel.com/docs/6.x/authentication>

De handigste manier

Het meest handige manier om een Laravel systeem op te zetten, is om als eerste te bedenken of je een login-systeem nodig hebt. In 9 van de 10 gevallen zal het antwoord **JA** zijn.

Als dat zo is kan de opbouw van het systeem begonnen worden met deze opdracht:

```
laravel new blog --auth
```

Dat maakt het vervolg eenvoudiger.

Downloaden

Je kunt de code die tot nu toe geprogrammeerd is downloaden van

<https://github.com/DrentheCollege/A5PHPVoorbeeld3.git>

Automatisch Testen

Onderdeel van Software Development is het testen. In eerste instantie zal je dat zelf doen, door bijvoorbeeld rond te klikken in je applicatie en gegevens in te voeren. Al heel snel wordt dat ingewikkeld omdat je behalve gewenste gegevens (om te kijken of het werkt) ook ongewenste gegevens invoeren (om te kijken of je applicatie goed daarop reageert). Je moet gaan bijhouden wat je allemaal test en wat bijhorende gegevens zijn. Behalve vervelend is het ook repeterend. Bovendien zal je ook elke keer als je iets bouwt of aanpast eigenlijk alles moeten testen om te controleren of er niet iets anders kapot gaat of zich op een andere manier verkeerd gedraagt.

Hiervoor zijn de automatische tests bedacht. Als je bezig bent geweest met A4 Java dan heb je de unittests gezien maar ook in PHP kan dit. We gaan het hier behandelen omdat Laravel ook hiervoor al alles aan boord heeft.

Meer uitgebreide informatie vind je op:

<https://laravel.com/docs/6.x/testing>

A5: MVC EN TESTEN

Simpele tests

De simpelste tests bestaan uit bijvoorbeeld tests van de modellen. De meest basale tests zouden kunnen zijn dat je test of een model bewaard wordt en daarna weer goed opgehaald wordt. Maar deze zijn vaak in de context van Laravel te triviaal om uit te voeren.

Dus pakken we het testen op bij een simpele methode die we nodig gaan hebben. We gaan een methode maken die kan zoeken op een bepaald contact.

Allereerst maken we een **factory**. Dit is een object dat op verzoek een Model-object teruggeeft waarbij de gewenste velden een gedefinieerde waarde hebben. Dat kunnen we doen met de volgende opdracht:

```
php artisan make:factory ContactFactory --model=Contact
```

En deze ziet er zo uit:

```
<?php

/** @var \Illuminate\Database\Eloquent\Factory $factory */

use App\Contact;
use Faker\Generator as Faker;

$factory->define(Contact::class, function (Faker $faker) {
    return [
        'first_name' => $faker->name,
        'last_name' => $faker->name,
        'city' => 'Stad',
        'country' => 'Land',
        'email' => $faker->email
    ];
});
```

Uitleg

Een factory klasse genereert objecten met gegevens. We vullen een aantal velden, sommige met concrete waarden en andere vanuit een faker object. Die genereert willekeurige gegevens die voldoen aan standaard voorwaarden. Ideaal dus om namen of emailadressen te bedenken. Factories worden over het algemeen gebruikt als je veel testgegevens moet bedenken.

Unittest maken

Vervolgens kun je een test bestand maken die standaard testen uitvoert met:

```
php artisan make:test ContactTest --unit
```

Deze opdracht maakt het volgende bestand: **tests\Unit\ContactTest.php**

Die kan gevuld worden met de volgende code:

```
<?php

namespace Tests\Unit;

use Tests\TestCase;
use App\Contact;
use Illuminate\Foundation\Testing\RefreshDatabase;

class ContactTest extends TestCase
{
    use RefreshDatabase;
    /** @test */
    function test_contactSearch()
    {
        factory(Contact::class, 5)->create();
        $first = factory(Contact::class)->create(['first_name' => 'Name']);
        $second = factory(Contact::class)->create(['last_name' => 'Name']);

        $contacts = Contact::contactSearch("Name");

        //Er moeten 2 contacten in de lijst zitten
        $this->assertEquals($contacts->count(), 2);

        //De eerste is bekend
        $this->assertEquals($contacts->first()->id, $first->id);

        //De tweede zou ook nog getest kunnen worden
        $this->assertEquals($contacts->last()->id, $second->id);
    }
}
```

Uitleg

We kijken even vooruit: deze test zal gedraaid worden tegen een database die in het geheugen zit, en dus na uitvoering van de tests dus weer weg is, en dus geen plek inneemt op de harde schijf.

use RefreshDatabase;

Deze regel zorgt ervoor dat na elke test de database opnieuw opgebouwd wordt, en dus leeg is. Daarna worden alle tabellen gemaakt volgens de migraties die je gemaakt hebt. Je krijgt dus een lege database vanuit de applicatie gezien.

factory(Contact::class, 5)->create();

Deze regel maakt 5 contacten met willekeurige gegevens zoals gedefinieerd in de factory en opgeslagen in de testdatabase.

\$first = factory(Contact::class)->create(['first_name' => 'Name']);

Hiermee wordt een Contact gemaakt in de testdatabase met als voornaam 'Name'.

\$contacts = Contact::contactSearch("Name");

Dit is de regel die je wil testen, in dit geval wil je alle contacten zoeken waarvan voor- of achternaam 'Name' bevat. Die komen terecht als een dataset in \$contacts.

A5: MVC EN TESTEN

```
$this->assertEquals($contacts->count(), 2);  
$this->assertEquals($contacts->first()->id, $first->id);  
$this->assertEquals($contacts->last()->id, $second->id);
```

Deze drie regels beschrijven wat je verwacht in de dataset:

1. Je verwacht 2 contacten, als die er niet in zetten krijg je een melding
2. De eerste is hetzelfde als \$first, je controleert alleen op id
3. De tweede is hetzelfde als \$second.

Uitvoeren van de test

De test kun je uitvoeren door dit te starten:

```
vendor\bin\phpunit
```

Dat gaat natuurlijk fout, want de methode bestaat nog niet. Dus voegen we die toe aan het Contactmodel:

```
class Contact extends Model  
{  
    // de rest is weggelaten  
  
    public static function contactSearch($name) {  
        return Contact::where('first_name', 'LIKE', "%$name%")  
            ->orWhere('last_name', 'LIKE', "%$name%")->get();  
    }  
}
```

Uitleg

Hier kun je zien hoe je zo'n ORM query opbouwt. **get()** zorgt ervoor dat je daadwerkelijk de gegevens terug krijgt.

A5: MVC EN TESTEN

Migratie aanpassen en repareren

Als je de test nu start gaat nog steeds fout, dit door een bug in SQLite. Dat is het systeem dat gebruikt wordt om de testdatabase op te bouwen. Configuratie vind je terug in **phpunit.xml**.

Dus passen we de migratie **AlterContactsTable** een beetje aan zodat SQLite niet fout gaat maar ook de oorspronkelijke migratie niet verandert, (de rest is voor het overzicht even weggelaten). Je ziet dat alleen de nullable() methode is toegevoegd:

```
public function up()
{
    //fixing a sqlite bug ->nullable()
    Schema::table('contacts', function (Blueprint $table) {
        $table->unsignedBigInteger('company_id')->nullable();
        $table->foreign('company_id')
            ->references('id')->on('companies');
    });
}
```

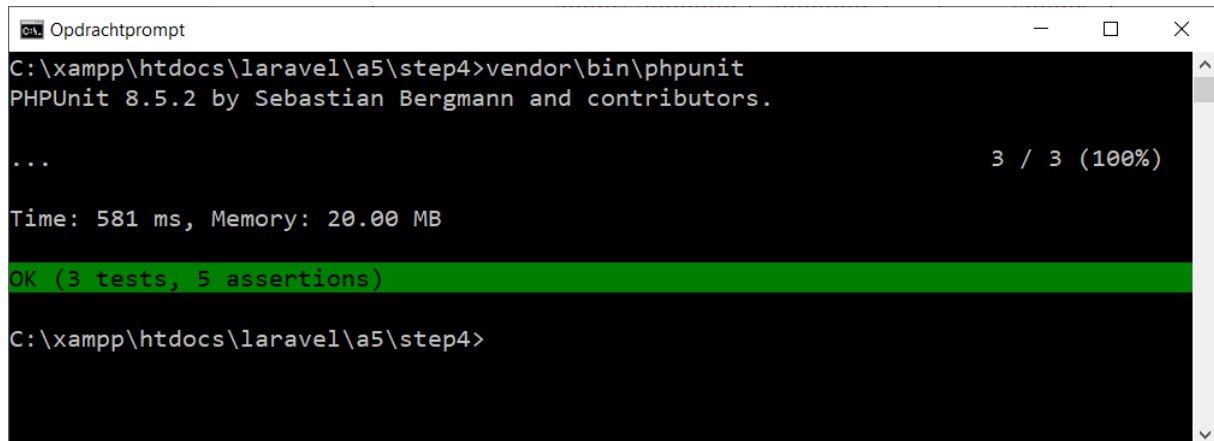
Wijzigingen in de migraties kun je eenvoudig testen met:

```
php artisan migrate:refresh
```

Deze opdracht maakt de database helemaal van het begin aan, zodat duidelijk wordt of alle migraties op elkaar aansluiten.

Testen

Het starten van de test zal dit opleveren. Succes!



Aanpassingen aan het scherm

Als je deze tests gedaan hebt, en die zijn goed verlopen, dan kan je dit gebruiken in je contact index scherm. Handige bijkomstigheid is dat je dus pas iets toevoegt aan het scherm als je zeker weet dat de onderliggende code werkt!

In `resources\views\contacts\index.blade.php` het vet gedrukte toevoegen.

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

@section('content')
<!--hier stond meer -->

<div>
    <a style="margin: 19px;" href="{{ route('contacts.create')}}" class="btn
    btn-primary">Contact toevoegen</a>
</div>
<!--van hier toevoegen -->
<div class="row">
    {!! Form::open(['method'=>'GET', 'url'=>'/contacts/', 'class'=>'navbar-form
    navbar-left', 'role'=>'search']) !!}
    <div class="input-group custom-search-form">
        <input type="text" class="form-control" name="keyword"
        placeholder="Zoek...">
        <span class="input-group-btn">
            <button class="btn btn-default-sm" type="submit">
                <i class="fa fa-search"><span class="glyphicon glyphicon-
        search"></span></i>
            </button>
        </span>
    </div>

    {!! Form::close() !!}
</div>
<!--toevoegen tot hier -->
<div class="row">

<!--hier stond meer -->
```

In `app\Http\Controllers>ContactController.php` de index methode vervangen door de index methode in onderstaand vak:

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Contact;
use App\Company;

class ContactController extends Controller
{
    public function index(Request $request)
    {
        $keyword = $request->keyword;
        if (isset($keyword)){
            $contacts = Contact::contactSearch($keyword);
        } else {
            $contacts = Contact::all();
        }

        return view('contacts.index', compact('contacts'));
    }
}
```

En daarna kun je zoeken op voor en achternaam in de contacten.

Conclusie

Met bovenstaande kennis kun je allerlei functionaliteit testen voordat je het implementeert, maar ook terwijl je je app verder ontwikkelt kun je voortdurend de functionaliteit die je al gebouwd hebt controleren op de correcte werking. Dus je bent steeds tijdens het proces op de hoogte of er iets kapot gaat terwijl je bezig bent, en kunt daar dan op reageren.

A5: MVC EN TESTEN

User Interface Tests

Ook de User-interface is te testen. Dit is handig om aan de hand van de use-case te controleren of je app wel steeds blijft werken zoals gedefinieerd en er geen bugs optreden. Hierna zullen we naar een aantal testbestanden gaan kijken en gaandeweg proberen te begrijpen wat er allemaal kan.

SiteTest

Het eerste testbestand dat we gaan behandelen is SiteTest.php. Die kun je maken door dit uit te voeren:

```
php artisan make:test SiteTest
```

Je krijgt dan een bestand **tests\Feature\SiteTest.php** zoals hieronder:

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithFaker;
use Tests\TestCase;

class SiteTest extends TestCase
{
    public function testExample()
    {
        $response = $this->get('/');

        $response->assertStatus(200);
    }

    public function testBasicSite() {
        $response = $this->get('/');

        $response->assertSeeText('Laravel');
        $response->assertSeeText('Voorbeeld');
    }

    public function testContactsNotLoggedIn() {
        $response = $this->get(route('contacts.index'));

        $response->assertSeeText('Redirecting to');
        $response->assertSeeText('login');
    }

    public function testLogin() {
        $response = $this->get(route('login'));

        $response->assertSeeText('E-Mail');
        $response->assertSeeText('Password');
    }
}
```

Uitleg

testExample()

Dit is een methode die standaard aangemaakt wordt. Wat deze doet is contact maken met de webserver en / ophalen. Vervolgens verwacht (assert) je dat het werkt en dus zal `assertStatus(200)` (het bestand is opgehaald en dus succesvol verlopen)

testBasicSite()

Deze test haalt / op en verwacht dat **Laravel** en **Voorbeeld** op de pagina staat.

testContactsNotLoggedIn()

Deze test haalt de route **contacts.index** op (dat is bij ons /contacts, zie web.php) en verwacht dat je vervolgens een redirect doet naar de login pagina (omdat alleen ingelogde gebruikers naar de contacts pagina mogen)

testLogin()

De laatste test gaat naar de route **login** (deze verwijst naar /login in ons geval) en verwacht dat daar **E-Mail** en **Password** staan.

Op deze manier kun je simpel testen of de webapp werkt zoals verwacht en je bijvoorbeeld naar de login-pagina gestuurd wordt als je niet ingelogd bent.

LoginTest

De volgende test heeft meer de focus op of het login systeem werkt zoals gewenst.

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithFaker;
use Tests\TestCase;
use App\User;

class LoginTest extends TestCase
{
    use RefreshDatabase;

    public function testLoginView()
    {
        $response = $this->get(route('login'));
        $response->assertSuccessful();
        $response->assertViewIs('auth.login');
        $response->assertSeeText('Login');
    }

    public function testCannotSeeLoginViewWhenLoggedIn()
    {
        $user = factory(User::class)->create();

        $response = $this->actingAs($user)->get(route('login'));
        $response->assertRedirect(route('home'));
    }

    public function testUserLoggedIn()
    {
        $user = factory(User::class)->create([
            'password' => bcrypt($password = 'i-love-laravel')]);

        $response = $this->post(route('login'), [
            'email' => $user->email,
            'password' => $password,
        ]);

        $response->assertRedirect(route('home'));
        $this->assertAuthenticatedAs($user);
    }

    public function testWrongUser()
    {
        $user = factory(User::class)->create([
            'password' => bcrypt($password = 'i-love-laravel')]);

        $response = $this->post(route('login'), [
            'email' => $user->email,
            'password' => 'secret',
        ]);

        $response->assertRedirect('/');
        $this->assertGuest(null);
    }
}
```


A5: MVC EN TESTEN

Spreekt vanzelf dat je die maakt met de volgende opdracht:

```
php artisan make:test LoginTest
```

Uitleg

testLoginView()

Deze methode gaat weer naar de login route, en verwacht dat dat succesvol verloopt. Daarna verwachten we dat we de view `auth\login.blade.php` te zien krijgen (dus `auth.login` in Laravel) en tot slot verwachten we dat het woord **Login** op het scherm staat.

testCannotSeeLoginViewWhenLoggedIn()

Hier beginnen we met het maken van een gebruiker (vanuit de factory). Deze gebruiker wordt door het testsysteem behandeld als een gebruiker die geregistreerd is en kan inloggen.

Vervolgens ga je als die ingelogde gebruiker naar de login route, en je verwacht dat je automatisch door gestuurd wordt naar de home route.

testUserLoggedIn()

De volgende test begint met het maken van een user (in dit geval met een wachtwoord dat je in de test bepaald. Vervolgens log je in in het systeem. De verwachting is dat je met een redirect door gaat naar de home route. Daarna verwacht je ook nog dat je ingelogd bent als die gebruiker.

testWrongUser()

Tot slot doe je ook nog een test om te kijken als je verkeerde logingegevens gebruikt, je daadwerkelijk niet ingelogd bent. Dus je maakt een gebruiker met een bekend wachtwoord, en probeert met een ander wachtwoord in te loggen.

We verwachten dat we weer naar de root van de app gaan, en als gast onze app bezoeken.

SearchContactTest

De meest gecompliceerde test die we doen, is testen of onze zoekmethode (die we eerder gemaakt hebben) ook daadwerkelijk werkt in het scherm. Daarvoor moeten we de test maken:

```
php artisan make:test SearchContactTest
```

Terwijl de tests gemaakt worden blijkt dat we ook een `CompanyFactory` moeten maken, mede omdat het bedrijf verplicht is in de schermen. Dus doen we dat ook direct:

```
php artisan make:factory CompanyFactory --model=Company
```

We beginnen met de `CompanyFactory`. Dat is een klasse die we nodig hebben om testdata te genereren voor onze tests. Deze vind je terug in `database\factories\CompanyFactory.php` en ziet er als volgt uit:

```
<?php

/** @var \Illuminate\Database\Eloquent\Factory $factory */

use App\Company;
use Faker\Generator as Faker;

$factory->define(Company::class, function (Faker $faker) {
    return [
        'id' => $faker->unique()->randomDigit,
        'name' => $faker->name
    ];
});
```

Uitleg

Hier wordt gedefinieerd dat de Companies een willekeurig id krijgen en een willekeurige naam. Kanttekening is dat in de naam de naam van fictieve mensen komt te staan, maar dat is natuurlijk altijd beter dan qwerty of hfhfhf.

A5: MVC EN TESTEN

Uitleg van de test

Voor de uitleg is dit bestand opgeknijpt:

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithFaker;
use Tests\TestCase;
use App\User;
use App>Contact;
use App\Company;

class SearchContactTest extends TestCase
{
    use RefreshDatabase;

    public function testContactView()
    {
        $user = factory(User::class)->create([
            'password' => bcrypt($password = 'i-love-laravel')]);

        $response = $this->actingAs($user)->get(route('contacts.index'));
        $response->assertSuccessful();
        $response->assertViewIs('contacts.index');
        $response->assertSeeText('Contacten');
    }
}
```

Uitleg

We gebruiken RefreshDatabase om ervoor te zorgen dat bij elke test de database in het geheugen bij elke methode weer opgebouwd wordt, en dan duidelijk gedefinieerde inhoud heeft.

Met de factory maken we weer een gebruiker die kan inloggen. Als die gebruiker ga je naar de route **contacts.index**. De verwachting is dat we succesvol de pagina kunnen opvragen. We verwachten dat het de view **contacts.index** is (uit het bestand **resources\views\vontacts\index.blade.php** dus) en tot slot verwachten we het woord **Contacten** te zien in het scherm.

A5: MVC EN TESTEN

```
public function testSearchContactViewAll()
{
    $user = factory(User::class)->create([
        'password' => bcrypt($password = 'i-love-laravel')]);

    factory(Contact::class, 5)->create();
    $randomName = Contact::all()->first()->first_name;

    $first = factory(Contact::class)->create(['first_name' => 'hhh']);
    $second = factory(Contact::class)->create(['last_name' => 'kkk']);

    $response = $this->actingAs($user)->get(route('contacts.index'));

    $response->assertSeeText('Contacten');
    $response->assertSeeText('hhh');
    $response->assertSeeText('kkk');
    $response->assertSeeText(Contact::all()->first()->first_name);
}
```

Uitleg

Voor een gedeelte is de test hier hetzelfde als hierboven. We maken weer een user die kan inloggen. Daarna maken we ook vijf contacten die willekeurige informatie bevatten. Voor de test bewaren we de `first_name` van de eerste.

Daarna maken we nog twee contacten met **hhh/kkk** erin. Er is gekozen voor deze namen omdat die niet per ongeluk voorkomen in de gegenereerde namen.

Vervolgens halen we de pagina met route **contacts.index** op. We verwachten dat het woord **Contacten** erin staat (dat is de kop) en ook dat de woorden **hhh** en **kkk** voorkomen in de pagina.

Tot slot verwachten we ook dat de `first_name` die we bewaard hebben in `randomName` ook voorkomt op de pagina.

```
public function testSearchContactView()
{
    $user = factory(User::class)->create([
        'password' => bcrypt($password = 'i-love-laravel')]);

    factory(Company::class, 2)->create();
    $company_id = Company::all()->first()->id;

    factory(Contact::class, 5)->create(["company_id" => $company_id]);
    $randomName = Contact::all()->first()->first_name;

    $first = factory(Contact::class)
        ->create(['first_name' => 'hhh', "company_id" => $company_id]);
    $second = factory(Contact::class)
        ->create(['last_name' => 'hhh', "company_id" => $company_id]);

    $response = $this->actingAs($user)
        ->get(route('contacts.index').'?keyword=hhh');

    $response->assertSeeText('Contacten');
    $response->assertSeeText('hhh');
    $response->assertDontSeeText($randomName);
}
}
```

Uitleg

Ook hier geldt hetzelfde als hierboven, het enige verschil is dat we twee bekende contacten aanmaken waarvan de ene **hhh** als `first_name` heeft en de andere als `last_name`.

Daarna halen we weer de `contact.index` route op maar nu met als zoekwoord **hhh**. Dus we testen of we in de userinterface de zoekfunctie kunnen aanroepen.

We verwachten de woorden **Contacten** en **hhh** in de pagina, maar nu **niet** dat `randomName` er in staat. Daar zoeken we immers niet op.

Downloaden

Je kunt de code downloaden van

<https://github.com/DrentheCollege/A5PHPVoorbeeld4.git>

Een project: het grote plaatje

Hier onder staat een voorbeeld hoe je een project zou kunnen beginnen, opdelen en uitvoeren. De punten zijn geïnspireerd op de kerntaken en werkprocessen van je opleiding, en bevatten dus heel veel oefenende stof voor een examen.

1. Definitiestudie
 - a. Overzichten geleverd door opdrachtgever
 - b. Andere informatie
2. FO schrijven
3. Database structuur maken
 - a. Normaliseren van de overzichten
4. Objecten bedenken (niet schrijven!)
5. Taken in Scrumboard zetten
 - a. Per tabel
 - b. Per Migratie
 - c. Per Object/Controller
 - d. Per View
6. Prioriteren van taken
 - a. Taken die niet van andere taken afhankelijk zijn eerst.
 - b. Taken waarvan andere taken afhankelijk zijn allereerst.
7. Start van project
 - a. TO maken of bijwerken
 - b. Basisproject installeren
 - c. Eventueel delen op Github oid.
8. Eerste sprint

Eindopdracht

De eisen zijn als volgt:

- FO is verplicht. Daarin staan alle mogelijke use-cases beschreven en verder de gebruikelijke hoofdstukken
- TO is verplicht. Daarin willen we in ieder geval een uitleg over hoe Laravel werkt qua MVC, routing en migraties. In het TO willen we ook een databaseschema zien.
- Als je deze module combineert met B4, dan verwachten we ook de formulieren of overzichten die als basis dienden voor de normalisatie, en ook in een apart worddocument uitgewerkt hoe de tussenstappen waren.
- Het wordt aangeraden om dit project met een team van minimaal 2 personen te doen, we verwachten van elk teamlid ook een evenwichtig aantal commits in Git. Dit moeten we kunnen controleren d.m.v. een link naar Github/GitLab etc.
 - We verwachten branches te zien in git, en ook tags voor de sprints.
- Er is een sprintboard dat we kunnen inzien, ook moeten er regelmatig screenshots gemaakt worden.
- Werkende software moet als team gedemonstreerd worden.

Voor de eindopdracht moet je uit de onderstaande projecten één kiezen om te maken.

Electronic Press Kit

Elke band heeft een Presskit, een pakket met informatie voor bedrijven die willen boeken of voor het geval een tijdschrift of website meer wil weten. Dit wordt tegenwoordig steeds meer elektronisch gedaan.

Eisen zijn:

- Een gebruiker moet een inlog hebben
- Een gebruiker kan meer dan één band onder zijn beheer hebben, en er kunnen meer gebruikers zijn die een band beheren.
- Een gebruiker komt dat in een dashboard hebben waar zijn gebruikersgegevens kunnen worden aangepast, en doorgeklikt naar een van de band EPK's die hij beheert.
- Een band EPK bestaat uit een pagina met bovenaan een foto, daaronder korte beschrijving en biografie tekst en daaronder de mogelijkheid voor (op dit moment) drie embedded Youtube video's. Achtergrond- en tekstkleur moeten aanpasbaar zijn.
- Als iemand als gast de website bezoekt, moet je kunnen zoeken op bandnaam, en dan krijgt de gast een tabel met bandnamen en korte beschrijvingen die voldoen aan de zoekcriteria.
- Na doorklikken op de bandnaam, kom je op een pagina die eruitziet zoals hierboven beschreven.

Je eigen project

Aanvullende eisen:

- Je moet in contact gaan met één van de docenten om uit te leggen dat je project voldoet aan de eisen
 - Minimaal vier tabellen exclusief de standaard Laravel tabellen.

A5: MVC EN TESTEN

- Relationeel verband tussen de tabellen.
- Bij voorkeur een document waarin beschreven staat wat het gaat doen (definitiestudie)

Afspraken

De opdracht lever je in in Magister. Zorg altijd de website ingepakt verstuurd wordt (zipfile). Als je niet weet hoe je een zipfile maakt: Klik rechts op de map waar jouw bestanden staan, en kies daarna voor “Kopiëren naar”->Gecomprimeerde (gezipte) map. Er wordt dan een zipfile gemaakt die je kunt opsturen.

Bronnen

Algemeen

- ✓ **Overerving**
[https://en.wikipedia.org/wiki/Inheritance_\(object-oriented_programming\)](https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming))

Laravel tutorials en documentatie

- ✓ **Tutorialspoint Laravel**
<https://www.tutorialspoint.com/laravel/index.htm>
- ✓ **Laravel Blade Templates**
https://www.tutorialspoint.com/laravel/laravel_blade_templates.htm
- ✓ **Laravel documentatie over queries**
<https://laravel.com/docs/6.x/eloquent>
- ✓ **Key too long error in Laravel**
<https://laravel-news.com/laravel-5-4-key-too-long-error>
- ✓ **Tutorial CRUD systeem**
<https://www.techiediaries.com/php-laravel-crud-mysql-tutorial/>
- ✓ **Login Laravel**
<https://www.techiediaries.com/laravel-authentication-tutorial/>
- ✓ **Authenticatie in Laravel**
<https://laravel.com/docs/6.x/authentication>
- ✓ **Documentatie Laravel over testen**
<https://laravel.com/docs/6.x/testing>
- ✓ **Laravel Model Testing**
<https://emcorrales.com/blog/laravel-testing-models>

- ✓ **Testing Laravel authentication Flow**
<https://medium.com/@DCzajkowski/testing-laravel-authentication-flow-573ea0a96318>

- ✓ **Documentatie Faker object**
<https://github.com/fzaninotto/Faker>

Voorbeeldcode

- ✓ **Voorbeeld code deel 1**
<https://github.com/DrentheCollege/A5PHPVoorbeeld1.git>

- ✓ **Voorbeeld code deel 2**
<https://github.com/DrentheCollege/A5PHPVoorbeeld2.git>

- ✓ **Voorbeeld code deel 3**
<https://github.com/DrentheCollege/A5PHPVoorbeeld3.git>