Laravel 2022 - L2

Introductie

In Laravel Level 2 ga je jouw webshop verder uitbreiden.

Zo wil je bijvoorbeeld producten tonen in jouw webshop door ze uit de database op te halen. Daarnaast geven wij jou een introductie van het CSS framework Bootstrap en maken we meerdere pagina's door views en controllers aan te maken en de routes goed in te stellen.

Hieronder is er een lijstje wat er in Laravel level 2 wordt behandeld:

- Meer pagina's (views/routings/controllers toevoegen)
- Introductie met Bootstrap
- Een losse pagina maken waar je al jouw producten laat zien

Bootstrap Installatie

Inleiding

Voor de vormgeving en styling gaan we Bootstrap gebruiken.

Bootstrap is een CSS (en Javascript) framework. Je kunt het zien als een groot aantal voorgedefinieerde styles die je kan gebruiken. Op die manier kun je snel 'even' je site oppimpen.

Wat is het en hoe werkt het?

Kijk maar eens op

> https://getbootstrap.com/docs/5.0/examples/cheatsheet/

Daar zie je allemaal user interface-elementen die met Bootstrap zijn vormgegeven. Je herkent vast wel wat van Yii want daar hebben we ook gebruik gemaakt van Bootstrap.

Alle uitgebreide informatie over Bootstrap vind je op:

> https://getbootstrap.com/docs/5.2/getting-started/introduction/

Installatie

Installatie kan op veel manieren.

De eenvoudigste manier is om de volgende regel in de <head> van je pagina te plaatsen:

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-Zenh87qX5JnK2Jl0vWa8Ck2rdkQ2Bzep5IDxbcnCeuOxjzrPF/et3URy9Bv1WTRi"
crossorigin="anonymous">
```

Hiermee laadt je de (hele grote) CSS style sheet.

Installatie via download

Een iets beter alternatief is om de stylesheet in je project te plaatsen. Dan zijn we niet afhankelijk van een andere server.

Download daarvoor de "Compiled CSS and JS" files.

Pak de zip file uit en plaats de folder die in de zip file staat in jouw Laravel project in de public folder.

Je krijgt dus dit:

De folder in de zip file heet zoiets als "bootstrap-5.2.2-dist". Deze folder kopieer je en hernoem je daarna zodat de naam van de folder "bootstrap" wordt.

In je header kun je nu de volgende regel plaatsen. Dit zorgt ervoor dat de gedownloade versie van Bootstrap wordt geladen.

```
<link href="{{asset('bootstrap/css/bootstrap.min.css')}}" rel="stylesheet">
```

Opdracht Menu

Installeer Bootstrap in je laravel project via de download-methode.

Plaats vervolgens deze HTML code in je <body> van de product-read.blade.php file. Plaats de code vóór de @foreach.

Dus regel 2 t/m10 uit de volgende code worden op de juiste plaats in prodcut-read.blade.php geplaatst.

```
....
<body>
  <nav class="navbar navbar-expand-md navbar-dark fixed-top bg-dark">
     <div class="collapse navbar-collapse">
        <ul class="navbar-nav mr-auto">
           <li class="nav-item"><a class="nav-link" href="/">Home</a></li>
           <li class="nav-item"><a class="nav-link right" href="/mandje">Mijn Mandje</a></li>
        </ul>
     </div>
  </nav>


  @foreach ($products as $product)
  ....
```

Laadt je productpagina opnieuw en je zult een menu-bar zien, die er zo uit ziet:

image_1666726096016.png

Inleveren

Screenshot van je product pagina mét Bootstrap menu

--

Mandje maken (draft)

Inleiding

We gaan ons boodschappenmandje maken.

De koppeling maken we later. We maken eerst net zoals we onze product controller en view hebben gemaakt in Laravel L1 een *mandje*.

Stappen

Weet je nog welke stappen we moeten uitvoeren?

Stap 1 Migration

```
php artisan make:migration create_basket_table --create=basket
```

Dit commando maakt een file. Deze staat in `database/migrations/`

Open de net gemaakte file en plaats deze code.

```
public function up()
{
    Schema::create('basket', function (Blueprint $table) {
        $table->increments('id');
        $table->integer('product_id');
        $table->integer('number');
        $table->timestamps();
    });
}
```

We hebben in het mandje (=basket) een productnummer staan. Deze verwijst naar de product-tabel en is de *Foreign Key* (FK).

We doen even alsof we maar één gebruiker hebben. In werkelijkheid is dat anders en heeft een mandje ook een username (zeg maar een koppeling naar de geregistreerde gebruiker). Maar we kunnen niet alles tegelijk maken dus we beginnen eenvoudig en doen even net alsof er maar één gebruiker is.

Met het commando

```
php artisan migrate
```

Maken we de tabel aan. Controller dit!

Stap 2 Model

Nu maken we een Model aan.

```
php artisan make:model Basket
```

Nadat je een model hebt aangemaakt kun je dit vinden in de map `app/Models`.

In het model hoeven we alleen maar de tabelnaam te specificeren.

```
class Product extends Model


{
    protected $table = 'basket';
    use HasFactory;
}
```

Stap 3 Controller

We maken de controller aan met:

```
php artisan make:controller BasketController
```

En in de controller zetten we de volgende method/function

```
public function read() {
    $items = \App\Models\Basket::all();
    return view('basket-read', ['items' => $items]);
}
```

Stap 4 View

We maken in de views folder een nieuw bestand basket-read.blade.php

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
</head>
<body>
    <?php dd($items); ?>
</body>
</html>
```

Stap 5 Route

Opdracht

Plaats via phpmyadmin minimaal twee artikelen in je mandje. Zorg ervoor dat de foreign key verwijst naar artikelen in de product tabel. Zet bij amount 1 en 2 neer.

Maak de view af en laat de inhoud van je mandje dat je net in de database hebt gezet zien.

Inleveren

1. Een schermafdruk van je browser waarmee je de inhoud van je (winkel)mandje laat zien. Zorg ook dat de URL zichtbaar is.

--

De kunst van blade files

Nu je een prachtige navigatiebar hebt gemaakt, zou je het dus ook op alle andere pagina's de code van de hele navigatie moeten plaatsen toch? Nou nee... Daarvoor hebben we Blades!

Wat is een Blade?

Blade is een template engine dat standaard is toegevoegd in Laravel. Je kan gemakkelijk php code in jouw .blade.php file stoppen. Ook kan je bladefiles hergebruiken in andere bladefiles. Dat betekent dus ook dat je makkelijk jouw navigatiebar kan hergebruiken!

Hoe hergebruik je jouw navigatie met Blade?

Voordat we beginnen met coderen, moet je eerst begrijpen welke syntax er is in Blade.

Zie hier: Blade Templates - Laravel - The PHP Framework For Web Artisans

Om jouw navigatie makkelijk opnieuw te laten gebruiken, gaan we drie blade variabeles gebruiken:
`@section` - door deze variabele om een stukje code te plaatsen, geef je dat gedeelte een sectienaam
`@yield` - plaatst de `@section` op de plek waar `@yield` staat. Dat kan je doen door de sectionnaam in `@yield` te stoppen
`@include` - voegt een andere `.blade.php` file toe in de huidige `.blade.php` file.
`@extends` - breidt de `.blade.php` uit met een andere `.blade.php` file

Hoe gaat dit visueel te werk?

Omdat dit een beetje ingewikkeld is om in tekst uit te leggen, is hieronder een visuele overzicht hoe een herbruikbare navigatie eruit moet zien:

image-1663542895344.jpg
Image not found or type unknown

Wat doet elke file?

`iets.blade.php`

Daar plaats je gewoon jouw content neer die je op de pagina wil tonen. Daar verwijs je ook met je routes en controller naartoe!
Deze content zit om een `@section`. De reden hiervoor is zodat je in een andere file deze sectie erin kan stoppen. Zoals je ziet ga je in de reusable file (reusable.blade.php) dezelfde sectie `@yield` 'en.

Maar om dat te kunnen doen, moet iets.blade.php file eerst aan de reusable.blade.php file worden verlengd. Dat doe je dus met `@extends('naam van de file')`. In dit geval moet er dus '*reuse*' staan.

reuse.blade.php

De reuse file wordt de pagina loader. Iedere keer als een wordt ingeladen, is het de bedoeling dat deze file geladen wordt door de geëxtende file (`iets.blade.php` in dit geval);. Zoals je daar ziet zet je daar jouw standaard HTML structuur in.

- Je ziet `@yield('content')`. Dat is dus de section die hij pakt van de iets.blade.php.
- Je ziet `@include('navigation')`. Hij zoekt naar een file die `navigation.blade.php` heet, en voegt de code toe in `reuse.blade.php` file.

navigation.blade.php

Het spreekt voor zich. Jouw gemaakte navigatie moet je dus hierin zetten! Je kan ook jouw navigatie in de reuse.blade.php stoppen, maar om alles overzichtelijk te maken kan je het beste in een losse file doen.

Opdracht: maak jouw navigatie herbruikbaar!

Stappenplan:

- Open de views folder en maak een file die **reuse.blade.php** (officieel moet het **app.blade.php** heten).
  - Voeg jouw standaard HTML structuur toe.
  - Voeg `@include('navigation')` en `@yield('content')` toe in jouw `<body>`
- Maak de navigatiefile en noem het naar **navigation.blade.php**
  - Plaats jouw naviagtiebar in de file
- Ga naar jouw file waar jouw content geplaatst wordt (kies bijvoorbeeld de file waar je de navigatie van de vorige opdracht hebt gemaakt).
  - Maak de file leeg en voeg `@extends('reuse')` (of officieel `@extends('app')`) toe.
  - Voeg eronder @section en @endsection. En schrijf wat content tussenin.

Wat lever je in?

1. Jouw `.blade.php` file met content erin
2. Jouw `reuse.blade.php` of app.blade.php file
3. Jouw `navigation.blade.php` file

> Gefeliciteerd! Je hebt jouw content en navigatiebar dynamisch gemaakt met Blade!

Producten toevoegen in jouw database

Nu heb je producten hard-coded toegevoegd. Natuurlijk wil je dat deze producten vanuit de database komen. Daarom gaan we aan de slag om de productdetails die je nu op de productpagina hebt gemaakt, in de database te gaan stoppen.

Product migration en model maken

Om jouw producten in de database te hebben, moet je eerst een tabel maken en die linken aan jouw Laravel project (met model)

Wat zijn migration en models ook alweer?

Migrations: is een 'blauwprint' van jouw tabel. Je geeft aan welke kolommen de tabel heeft en welke datatypes. Vervolgens migreer je het blauwprint in de database. Zo wordt er dan automatisch een tabel gemaakt!

Models: is de 'M' van het MVC model. Models zijn verantwoordelijk voor de link tussen het project en de database (dus niet de migrations! Die regelt alleen de blauwprinten). Zo kan je bijvoorbeeld dan in de controller een model aanroepen om data op te halen uit de database.

Opdracht: een migration en model maken

Laten we beginnen door eerst een migration aan te maken via de console. Hier een stappenlijst:

1. Maak een migration aan via de terminal met de naam `create_products_table`
2. Open de migrationfile en bedenk welke kolommen nodig zijn voor jouw tabel. Kijk wat je op jouw productpagina hebt. Hier alvast een aantal:
   - id
   - title
   - price
3. Voeg deze kolommen toe in jouw migration file

4. Voer de migration uit via de terminal. Mocht je al de tabel hebben, dan moet je het refreshen
5. Check in jouw PHPMyAdmin of jouw tabel is toegevoegd
6. Maak een model via de terminal. Noem de file `Product`
7. Open de model file en voeg de kolomnamen in de model toe van de tabel
8. Voeg in PHPMyAdmin jouw producten toe in de tabel.

> Weet je niet wat de commando's zijn voor het aanmaken van een migration of model? Check dan even Laravel Level 1: Migration en Models!

Wat lever je in?

- Een screenshot van jouw migration file
- Een screenshot van jouw models file
- Een screenshot van jouw gemigreerde tabel in PHPMyAdmin

Producten laten zien vanuit de database en de kracht van Eloquent

Nu je de database gemaakt hebt en producten erin hebt gezet, moet je nog de producten ophalen vanuit de database en tonen op het scherm.

Hoe haal je jouw producten op uit de database?

Zoals in eerdere blokken aangegeven, kan je communiceren met de database via Models. Om aan te geven welk model (dus welk tabel) data wil ophalen, moet je die aangeven in de Controller.

Data opvragen doe je in de Controller

Weet je nog wat de controller doet? Voor recap:
Controllers zorgt voor de logica en verbindt de **Model** met de **View**. In de uitleg van het **MVC model** was de **Controller** de ober. Die moet ervoor zorgen dat de bestellingen van de klant (de view) goed wordt doorgegeven aan de chef (model).

Maar hoe geef je dan aan dat je jouw producten wil via de `Product` model? Daarvoor hebben we **Eloquent**.

Wat is Eloquent?

Eloquent zorgt ervoor dat je **CRUD** acties aansturen met jouw **Models**. In dit geval voor jouw producten, kan je dus **Eloquent** gebruiken bij de **Product** model. Zo kan je nu bijvoorbeeld jouw producten ophalen (C**R**UD van **Read)**

Hoe haal je data op uit een tabel?

Om data op te halen uit een tabel moet je dus aansturen wat de Model moet doen. Alle aansturingen en logica doe je dus in jouw controller. Hieronder is er een voorbeeld hoe je auto's ophaalt:

Carcontroller.php
Stel dat dit een controller is die je hebt gemaakt:

```
namespace App\Http\Controllers;


class CarController extends Controller
  public function showCars(){


    return view('listcars');
  }
}
```

In deze functie wil je alle auto's ophalen, en de data in de view meenemen (daarom heet de functie ook showCars). Daarvoor moet je dus eerst aangeven welk Model je wil aansturen. Voor dit voorbeeld heb ik een `Car` model gemaakt. Om aan te geven welk Model je wil aansturen geef je eerst aan in de controller welk model je wil gebruiken met **use**:

`use App\Models\Car;`

Dit stukje code zet je tussen de namespace en class in:

```
namespace App\Http\Controllers;
use App\Models\Car;


class CarController extends Controller
  public function showCars(){


    return view('listcars');
  }
}
```

Vervolgens geef je in de functie aan met Eloquent wat je wil aansturen met de Car Model. In dit geval wil je dus ophalen met:

`$allCars = Cars::all();`

Dit stukje code zet je in de functie, voordat je de view laat zien zoals hieronder is weergegeven:

```
namespace App\Http\Controllers;
use App\Models\Car;


class CarController extends Controller
  public function showCars(){
```

```
  $allCars = Car::all();
    return view('listcars');
  }
}
```

Opgehaalde data meenemen in een view

Nu heb je alle auto's opgehaald met een model via Eloquent. Die heb je in de variabele `$allCars` gezet. Om de auto's op de view te laten zien, moet je de variabele meenemen waar je jouw view returnt. Dat is in dit geval `return view('listcars');`

De functie `view()` kan je twee parameters geven:

- Paramater 1: jouw view die je wil tonen.
- Parameter 2: data die je in de view wil meenemen (zoals in dit geval de auto's)

Voor parameter 2, kan je op verschillende manieren data (in dit geval de auto's) meenemen. Zo kan je bijvoorbeeld data meenemen hoe de variabelenaam (in dit geval $allCars) heet: `compact('allCars')`

Wat doet `compact()`?

Met compact() kan je een bestaande variabele met data in de controller meenemen naar de view en daarin je dezelfde variabele kan gebruiken.

Dit voeg je dus toe in de `return view()`:

```
namespace App\Http\Controllers;
use App\Models\Car;

class CarController extends Controller
  public function showCars(){

  $allCars = Car::all();
    return view('listcars', compact('allCars'));
  }
}
```

Je kan dus nu in jouw view `$allCars` gebruiken om de data op te halen en te gebruiken (door bijvoorbeeld te tonen)!

Meegenomen data in de view tonen

Met behulp van de compact() functie, kan je dus de variabele $allCars ook gebruiken in jouw view! Zoals je al eerder weet, kan je met Blade PHP code schrijven.

Stel: dit is de view van `listcars`:

```
@extends('reuse')
@section('content')
<div class="container">
    <h1> Mijn Auto's </h1>
    <div class="row">
    <div class="col-4">
        <h3>Porsche 911</h3>
        <p>Supervette auto!</p>
      </div>
    <div class="col-4">
        <h3>Volkswagen Polo</h3>
        <p>Goede prijs-kwaliteit auto!</p>
      </div>
    </div>
</div>
@endsection
```

Zoals je ziet zijn er twee auto's hard-coded gemaakt. Dit gaan we dus dynamisch maken door de data te laten zien uit de database met behulp van Blade Templates:

Deze code:

```
<div class="col-4">
  <h3>Porsche 911</h3>
  <p>Supervette auto!</p>
</div>
<div class="col-4">
  <h3>Volkswagen Polo</h3>
  <p>Goede prijs-kwaliteit auto!</p>
</div>
```

Zetten we nu om naar:

```
@foreach($allCars as $car)
<div class="col-4">
  <h3>{{$car->title}}</h3>
```

```
  <p>{{$car->description}}</p>
</div>
@endforeach()
```

Wat staat er in de code?

- `@foreach` / `@endforeach`: zorgt ervoor dat je jouw array (in dit geval `$allCars`) geloopt wordt. Elke auto kan je aanroepen met `$car`. De variabele kan je zelf noemen.
  - In de loop, loop je ook een `<div class="col-4">`. Dus er passen maximaal 3 kolommen met autogegevens per rij (er passen 3x col-4 per rij).
    - In de kolommen heb je de data die je toont van `$car`. Je kan een specifieke data tonen met behulp van de kolomnaam uit de Model zoals `$car->title` en `$car->description`

Opdracht: data ophalen met Eloquent en op een view laten zien (10p)

Om data van een model (in dit geval jouw **Product** model) op te halen moet je eerst jouw model in jouw controller toevoegen:

Stappenplan:

1. Open jouw `ProductController.php` file.
2. Om jouw model toe te voegen, voeg je deze code toe: `use App\Models\Modelnaam;`. Jouw *Modelnaam* is dus de naam van jouw Model en dat is `Product`
3. Zoek in jouw `ProductController` naar de functie die jouw `view` returnt naar de productpagina.
   1. Maak in de functie code om gegevens op te halen via Product Model met Eloquent en stop die in een variabele.
   2. Maak gebruik met compact() om data mee te nemen in de view
4. Ga in de aangewezen view een toon daar jouw gegevens uit de database met behulp van Blade Templates

Wat lever je in?

- Jouw ProductController.php file
- Jouw Productpagina view