# Laravel - CRUD

- Basis
- Read
- Create
- Update
- Delete
- Views optimaliseren
- Extra - Euro-teken en totaal
- Debug (nog uitwerken)
- Evaluatie

# Basis

## Inleiding

Op de financiële beurs kan je aandelen kopen van bijvoorbeeld Ajax, Netflix Faceboot (Meta), Tesla of ABN Amro. Met een aandeel koop je een (heel) klein stukje van het bedrijf. Maakt het bedrijf winst dan deel jij via dat aandeel ook mee in de winst.

Je opdrachtgever heeft gevraag om een CRID te maken zodat hij zijn aandelenbezit kan bijhouden.

De applicatie moet in het Engels. De naam van de applicatie wordt *stock*, dat is Engels voor aandeel.

## Wat moet je weten?

Je kent HTML, PHP, CSS en je hebt ervaring met programmeren in de MVC-structuur. Zoals we dat met Yii gedaan hebben.

## Composer

Composer is de installer voor PHP-tools en frameworks zoals Yii en Laravel.

Je hebt composer al geïnstalleerd.

Maar voor de zekerheid hier nog een keer de link: https://getcomposer.org/doc/00-intro.md#installation-windows

## Maak nieuw Laravel project

We maken een nieuw Laravel project en noemen dat *stock*.

```
composer create-project laravel/laravel stock
```

Open dit project in VCS.

In de terminal start je de development server van Laravel.

```
php artisan serve
```

Ga naar de browser en open localhost:8000.

Je ziet de welkom pagina van Laravel.

# Maak een nieuwe database

Start XAMPP en maak met http://localhost/phpmyadmin een nieuwe database en noem die *stock*.

image-1667033012461.png

In Laravel (in VCS) op je de .env file en je verandert de database *laravel* naam naar *stock*

image-1667033254058.png

# Migrations

In Laravel maak je de database tabel niet in phpmyadmin, maar dat doe je met Laravel. De reden hiervoor is dat als je je code aan iemand anders geeft de database tabellen door Laravel worden aangemaakt.

Als we het model maken dan wordt er vanzelf een migratie file aangemaakt.

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

Ga nu naar database/migrations en open de file xxxxxxx_create_stocks_table.php

De functie up() is aangemaakt, maar bevat twee standaard regels. Bereid de functie uit zodat deze er als volgt komt uit te zien.

```
public function up()
{
    Schema::create('stocks', function (Blueprint $table) {
        $table->increments('id');
        $table->string('stock_name');
        $table->string('ticket');
        $table->decimal('value', 8, 2);
        $table->timestamps();
    });
}
```

Nu geven we Laravel het commando om de database tabel te maken volgens onze beschrijving.

```
php artisan migrate
```

Ga naar phpmyadmin en controleer of de tabel is aangemaakt.

Stel je wil de tabel veranderen dan kun je met het volgende commando kun je de migratie dan opnieuw uitvoeren.

```
php artisan migrate:refresh
```

Laravel maakt meer tabellen aan, maar de stock tabel moet er ook bij staan.

image-1667034023416.png

# Model

Het model voor stock staat in app/Models/Stock.php.

Met regel 9 verwijzen we naar de juiste tabelnaam, *stocks*.

```
class Stock extends Model
{
    protected $fillable = [
        'stock_name',
        'ticket',
        'value'
    ];
    protected $table = 'stocks';
    use HasFactory;
}
```

Regel 3 t/m 8 zorgt ervoor dat de velden door de gebruiker via Laravel zijn aan te passen.

Regel 9 zorgt ervoor dat we de database met handige Laravel functies kunnen raadplegen en aanpassen.

# Waar staat wat in Laravel?

De belangrijkste file locaties in Laravel zijn.

| | |
|---|---|
| **database naam** | .env |
| **migratie bestanden** | database/migrations |
| **routes** | web.php |
| **models** | app/Models/ |

| | |
|---|---|
| **controllers** | app/Http/Controllers/ |
| **views** | resources/views/ |

--

# Read

## Inleiding

We hebben een database aangemaakt.

We hebben via een migration een tabel aangemaakt.

En we hebben een model aangemaakt zodat Laravel 'weet' waar de data staat en hoe de tabel heet.

We gaan nu de Read functie maken. We gaan daarvoor de **controller**, **view** en de **routing** opzetten.

## Controller

Als de gebruiker van de webapplicatie de applicatie gebruikt dan gaat hij altijd eerst naar de controller. De controller gebruikt dan het model om de database te raadplegen en de output wordt view de view aan de gebruiker getoond. Dit is de MVC-structuur waarmee Laravel een applicatie opbouwt.

Met het volgende commando maken we een controller.

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

Dit commando maakt een file app/Http/Conttrollers/StockController.php

image-1667035179108.png

Open deze file en plaats in de index functie de volgende code

```
public function index()
{
    $stocks = Stock::all();

    return view('stocks.index', compact('stocks')); // ->
resources/views/stocks/index.blade.php
}
```

Regel 3 haalt alle records uit de database en regel 5 opent de view (die we nog moeten maken).

Voeg bovenaan in de controller (op regel 6 bijvoorbeeld) de volgende regel code toe.

```
use App\Models\Stock;
```

Hiermee 'kent' de controller het model van stock.

# View

In Laravel staan alle views in resources/view en alle view files hebben de file extensie blade.php

Waarom precies wordt later uitgelegd.

We maken een nieuwe file in resources/view/stocks/stocks en noemen die file index.blade.php. De folder stocks moeten we ook zelf aanmaken!

We plaatsen de volgende code in resources/view/stocks/index.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">
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@4.0.0/dist/css/bootstrap.min.css"
integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm"
crossorigin="anonymous">
    <title>Stocks</title>
</head>
<body>
  <div class="container" style="margin:40px;">
    <table class="table">
      <thead class="thead-light">
          <tr>
            <th>ID</th>
            <th>Stock Name</th>
            <th>Stock Ticket</th>
            <th>Stock Value</th>
            <th>Updated at</th>
          </tr>
      </thead>
```

```
      <tbody>
          @foreach($stocks as $stock)
          <tr>
              <td>{{$stock->id}}</td>
              <td>{{$stock->stock_name}} </td>
              <td>{{$stock->ticket}}</td>
              <td>{{$stock->value}}</td>
              <td>{{$stock->updated_at}}</td>
          </tr>
          @endforeach
      </tbody>
    </table>
  </div>

</body>
</html>
```

Op regel 7 laden we Bootstrap (CSS framework) en gebruiken dat voor de vormgeving. We gaan daar in deze lessen verder niet op in.

De rest is standaard HTML code waarbij de resultaten in een tabel worden afgedrukt.

De *for-each* loop die we gebruiken  (regel 23 en 31) ziet er iets anders uit als we gewend zijn in PHP en in Yii.

Dit is de Laravel-manier en hoort bij *blade* templates. Hierover later meer.

# Routing

De laatste stap is de routing. In Yii ging dit volgens een vast patroon. In Laravel moeten we de routing zelf coderen.

De routing staat in de file routes/web.php

```
Route::get('stocks', [App\Http\Controllers\StockController::class, 'index']);
```

Dit koppelt de url */stock* aan de functie *index()* in de *StockController* class.

# Testen

Om goed te kunnen testen moeten we de database vullen met wat data. Dat kun je doen door met de onderstaande query in phpmyadmin, 5 regels in de database toe te voegen.

```
INSERT INTO `stocks` (`id`, `stock_name`, `ticket`, `value`, `created_at`, `updated_at`)
VALUES
(1, 'Apple Inc.', 'AAPL', '1200.00', NULL, NULL),
(2, 'NVIDIA Corporation', 'NVDA', '830.00', NULL, NULL),
(3, 'Tesla Inc', 'TSLA', '1950.00', NULL, NULL),
(4, 'Netflix Inc', 'NFLX', '300.00', NULL, NULL),
(5, 'Amazon.com, Inc.', 'AMZN', '1640.00', NULL, NULL);
```

Zorg dat de Laravel ontwikkel-omgeving draait (php artisan serve) en ga naar localhost:8000/stocks. Als het goed is dan zie je het volgende overzicht:

image-1667044390059.png

De laatste kolom *Updated at* is (nog) leeg omdat de records via een query zijn aangemaakt. Als we straks via de browser records aanmaken zul je zien dat deze kolom automatisch wordt gevuld.

# Create

## Inleiding

We hebben nu een werkende read; we kunne de inhoud van een tabel op het scherm tonen.

We het model uitbreiden, routes maken en een view maken zodat we een record aan de tabel in de database kunnen toevoegen.

## Create Route

We beginnen met een nieuwe route.

```
Route::get('stocks/create', [App\Http\Controllers\StockController::class, 'create']);
```

## Controller

Vervolgens maken we een functie create in de StockController file.

```
public function create()
{
    return view('stocks.create'); // -> resources/views/stocks/create.blade.php
}
```

## View

We maken even een tijdelijke view.Maak een nieuwe file resources/views/stock/create.blade.php en plaats er even tijdelijk code in, bijvoorbeeld.

```
<h1>Topper!</h1>
```

In de andere view die we al hadden, de file resources/views/stock/index.blade.php plaatsen we de volgende code vlak boven de <tabel>

```
<h1 class="display-3">Stocks</h1>
<div>
  <a href="/stocks/create" class="btn btn-primary mb-3">Add Stock</a>
```

```
</div>
```

# Testen

Op de stocks pagina (localhost:8000/stocks) staat nu een Create button.

Wat doet deze button?

De button gaat naar /stocks/create.

De route *stocks/create* verwijst naar de *create()* functie in de *StockController* en die verwijst naar de view *resources/views/stock/create.blade.php*

Dus in het kort:

> Create Button -> /stocsk/create -> (route) StockController function create() ->(view) create.blade.php

De routing werkt nu, maar er wordt nog geen nieuwe record aangemaakt.

We gaan nu eerst de create view aanpassen zodat er een form wordt getoond waarin we een nieuw record kunnen aanmaken.

# Input form

Plaats in de create.blade.php view file nu de volgende code.

```
<!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">
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@4.0.0/dist/css/bootstrap.min.css"
integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm"
crossorigin="anonymous">
    <title>Stocks</title>
</head>
<body>
  <div class="container" style="margin:40px;">
```

```
  <h1 class="display-3">Stocks</h1>

    <form method="post" action="/stocks/store">
          @csrf
          <div class="form-group">
              <label for="stock_name">Stock Name:*</label>
              <input type="text" class="form-control" name="stock_name"/>
          </div>

          <div class="form-group">
              <label for="ticket">Stock Ticket:*</label>
              <input type="text" class="form-control" name="ticket"/>
          </div>

          <div class="form-group">
              <label for="value">Stock Value:</label>
              <input type="text" class="form-control" name="value"/>
          </div>
          <button type="submit" class="btn btn-primary">Add Stock</button>
      </form>

  </div>

</body>
</html>
```

# Store Route

Het form doet een post naar /stocks/store. We gaan hiervoor een route toevoegen (in de web.php).

```
Route::get('stocks/store', [App\Http\Controllers\StockController::class, 'store']);
```

# Store functie Controller

De store functie in de controller ziet er als volgt uit.

```
public function store(Request $request)
{
```

```
    // Validation for required fields (and using some regex to validate our numeric value)
    $request->validate([
        'stock_name'=>'required',
        'ticket'=>'required',
        'value'=>'required|max:10|regex:/^-?[0-9]+(?:\.[0-9]{1,2})?$/'
    ]);
    // Getting values from the blade template form
    $stock = new Stock([
        'stock_name' => $request->get('stock_name'),
        'ticket' => $request->get('ticket'),
        'value' => $request->get('value')
    ]);
    $stock->save();
    return redirect('/stocks')->with('success', 'Stock saved.');   // ->
resources/views/stocks/index.blade.php
}
```

## Validatie

In regel 4 tot en met 8 worden de velden gevalideerd. Omdat dit in de controller op de server gebeurt is dit een server-side validatie. Je kunt ook via HTML valideren, dat heet cliënt-side validatie en is minder veilig.

## Nieuw record

Regel 10 tot en met 14 maakt een nieuw record aan en vult dat met de gevalideerde waarden uit het form.

Regel 15 slaat de record op in de database.

Regel 16 gaat terug naar de /stocks url. Via de route wordt hierdoor de index() functie aangeroepen en worden alle regels getoond.

Het tweede gedeelte van de de regel zorgt ervoor dat er een boodschap op het scherm verschijnt.

--

# Update

## Inleiding

Voor de update functie gaan we een knopje maken in ons overzicht.

## Index View

We beginnen met het aanpassen van de index view. Plaats deze code als extra kolom in de tabel en vergeet niet ook een extra <th></th> te plaatsen. Het aantal kolommen in de tabel moet in alle regels gelijk zijn!

```
<td><a href="/stocks/edit/id={{$stock->id}}" class="btn btn-primary">Edit</a></td>
```

## Edit Route

We voegen een route toe in de web.php

```
Route::get('stocks/edit/{id}',   [App\Http\Controllers\StockController::class, 'edit']);
```

Zodra we op de knop edit drukken in het overzicht (in de index.blade.php view) dan wordt de edit() function in deStockController aangeroepen. Het $id wordt bovendien meegegeven.

## Controller edit()

In de controller zetten we de volgende code in de edit() function.

```
public function edit($id)
{
    $stock = Stock::find($id);
    return view('stocks.edit',['stock'=>$stock]);  // -> resources/views/stocks/edit.blade.php
}
```

De edit() functie in de controller haalt het record met het id $is op en het record, $stock wordt aan de view edit.blade.php in de stocks folder meegegeven.

# Edit View

We plaatsen de volgende code in de edit.blade.php file in de stocks view. Deze file moeten we aanmaken.

```
<!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">
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@4.0.0/dist/css/bootstrap.min.css"
integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm"
crossorigin="anonymous">
    <title>Stocks</title>
</head>
<body>
  <div class="container" style="margin:40px;">

    <h1 class="display-3">Stocks</h1>

    <form method="post" action="/stocks/update/{{$stock->id}}">
        @csrf
        <div class="form-group">

            <label for="stock_name">Stock Name:*</label>
            <input type="text" class="form-control" name="stock_name" value="{{ $stock-
>stock_name }}" />
        </div>

        <div class="form-group">
            <label for="ticket">Stock Ticket:*</label>
            <input type="text" class="form-control" name="ticket" value="{{ $stock->ticket }}"
/>
        </div>

        <div class="form-group">
            <label for="value">Stock Value:</label>
            <input type="text" class="form-control" name="value" value="{{ $stock->value }}"
```

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

  </div>

</body>
</html>
```

Test de code en als het goed is, verschijnt het edit form als er op de editknop wordt gedrukt.

image-1667129791204.png

Wanneer de data wordt aangepast in dit edit form dan wordt er een post gedaan /stocks/update/{{$stock-id}}. Ben je bijvoorbeeld record met id 1 aan het aanpassen dan wordt het formulier gepost naar /stocks/update/1.

Hiervoor moeten we nu een route aanmaken.

# Update Route

We voegen de volgende route in web.php.

```
Route::post('stocks/update/{id}',   [App\Http\Controllers\StockController::class, 'update']);
```

Nu moeten we de update() functie in de StockController aanpassen zodat het aangepaste record naar de database wordt weggeschreven.

# Controller update()

plaats de volgende code in de update() functie in de StockController.

```
public function update(Request $request, $id)
{
    // Validation for required fields (and using some regex to validate our numeric value)
    $request->validate([
        'stock_name'=>'required',
        'ticket'=>'required',
        'value'=>'required|max:10|regex:/^-?[0-9]+(?:\.[0-9]{1,2})?$/'
    ]);
    $stock = Stock::find($id);
```

```
    // Getting values from the blade template form
    $stock->stock_name =  $request->get('stock_name');
    $stock->ticket = $request->get('ticket');
    $stock->value = $request->get('value');
    $stock->save();

    return redirect('/stocks'); // -> resources/views/stocks/index.blade.php
}
```

In het eerste gedeelte regel 4 t/m 8 worden de velden gevalideerd. Dit is hetzelfde als bij de *create* .

Daarna wordt op regel 9 t/m 14 het juisten record uit de database geladen en worden de nieuwe waarden weggeschreven.

Op regel 16 wordt naar de index view gesprongen. Dit is het standaard overzicht.

# Samengevat

We beginnen dus met een update vanuit de index view.

Dan gaan we via de route stocks/edit/{{$id}} naar de edit() functie in de StockController.

De StockController haalt de data op en laat deze data via de edit view zien.

Vanuit de edit view worden nieuwe waarden via een post naar de route stocks/update/{{id}} gestuurd.

De route stocks/update/{{id}} verwijst naar de update() functie in de StockController.

Deze functie voert de update uit en gaat daarna (terug) naar de index view.

Of in het kort, ziet het rondje er als volgt uit.

> index view -> (route) stocks/edit/{{$id}} -> edit() functie in StockController -> (view) update -> (route) stocks/update/{{$id}} -> update() functie in StockController -> index view

--

# Delete

## Inleiding

We maken een delete-knop naast de edit-knop. Delete is wat ingewikkelder omdat we niet willen dat iemand zomaar records kan deleten. We moeten dus wat security maatregelen nemen.

Het verwijderen zelf is eenvoudig en we hebben geen aparte view nodig.

## View Index

Net als bij de edit, plaatsen we een extra kolom in de index.blade.php file.

```
<td>
  <form action="/stocks/destroy/{{$stock->id}}" method="post">
    @csrf
    <button onclick="return confirm('Are you sure?')" class="btn btn-danger"
type="submit">Delete</button>
  </form>
</td>
```

### Uitleg over post en @csrf

We kiezen ervoor om geen <a></a> voor de delete button te gebruiken (zoals bij de edit button). Dat doen we omdat we een post willen gebruiken en dat kan alleen met een form en een button.

De reden dat we een post willen gebruiken is dat we niet willen dat de gebruiker rechtstreeks via een URL een record kan verwijderen.

De edit functie kun je gewoon via de route /stocks/edit/1 bereiken. Dit is een GET verzoek.

De destroy/delete functie kun je *niet* via de route /stocks/destroy/1 bereiken omdat dit ook een GET verzoek is en een destroy alleen via een POST verzoek kan worden aangeroepen.

Regel 3 is er om ervoor te zorgen dat je niet vanuit een andere webpagina of server records uit de database kan verwijderen. Met @csrf wordt er een soort unieke sleutel meegegeven waardoor de webserver weet dat er vanuit deze applicatie een delete/destroy mag worden uitgevoerd.

De index pagina van de web site ziet er nu als volgt uit.

image-1667129556823.png

Nu moeten we de knop Delete nog afmaken. We beginnen met het toevoegen van een route.

# Route Destroy

Zoals uitgelegd hierboven mag de destroy functie alleen worden aangeroepen als er een post verzoek is gedaan. De route wordt dan.

```
Route::post('stocks/destroy/{id}',  [App\Http\Controllers\StockController::class, 'destroy']);
```

Voeg deze route aan web.php.

De route verwijst naar de controller functie destroy(). Deze functie moeten we dus nog aanmaken.

# Controller Destroy

In de controller zoeken we het juiste record op en verwijderen we deze. Daarna gaan we terug naar de index view.

```
public function destroy($id)
{
    $stock = Stock::find($id);
    $stock->delete(); // Easy right?

    return redirect('/stocks');  // -> resources/views/stocks/index.blade.php
}
```

Test of alles goed werkt!

--

# Views optimaliseren

## Inleiding

We hebben nu een werkende CRUD.

We gaan nog wat aanpassingen maken, maar we gaan eerst onze view een beetje herorganiseren.

We hebben nu drie views:

1. index
2. edit
3. update

Deze drie views lijken op elkaar en hebben allemaal een standaard HTML template. Eigenlijk is alleen de <body></body> anders.

En als we later een menu of banner willen maken dan zal die banner op alle drie de pagina's getoond moeten worden.

We gaan de template nu splitsen in een basis template en in een deel dat voor de views anders is.

## Basis (blade) template

Maak een nieuwe template in de *resources/view* folder en noem deze base.blade.php.

Plaatst de volgende code.

**resources/view/base.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">
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@4.0.0/dist/css/bootstrap.min.css"
integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm"
```

```
crossorigin="anonymous">
    <title>Stocks</title>
</head>
<body>
  <div class="container" style="margin:40px;">
    <h1 class="display-3">Stocks</h1>
    @yield('main')
  </div>
</body>
</html>
```

In de index.blade.php (in de stocks folder!) passen we de template aan.

**resources/view/stocks/index.blade.php**

```
@extends('base')

@section('content')
  <div>
    <a href="/stocks/create" class="btn btn-primary mb-3">Add Stock</a>
  </div>

  <table class="table">
    <thead class="thead-light">
        <tr>
          <th>ID</th>
          <th>Stock Name</th>
          <th>Stock Ticket</th>
          <th>Stock Value</th>
          <th>Updated at</th>
          <th>Update</th>
          <th>Delete</th>
        </tr>
    </thead>
    <tbody>
        @foreach($stocks as $stock)
        <tr>
            <td>{{$stock->id}}</td>
            <td>{{$stock->stock_name}} </td>
            <td>{{$stock->ticket}}</td>
```

```
            <td>{{$stock->value}}</td>
            <td>{{$stock->updated_at}}</td>
            <td><a href="/stocks/edit/{{$stock->id}}" class="btn btn-primary">Edit</a></td>
            <td>
              <form action="/stocks/destroy/{{$stock->id}}" method="post">
                @csrf
                <button onclick="return confirm('Are you sure?')" class="btn btn-danger"
type="submit">Delete</button>
              </form>
          </td>
        </tr>
        @endforeach
    </tbody>
  </table>
@endsection
```

Vanuit de controller wordt nog steeds de *index.blade.php* aangeroepen. Op regel 1 wordt de *base.blade.php* file als het ware included. De *base* view wordt als het ware op regel 1 ingevoegd.

In de base view staat (op regel 13) yield('content'). Dit zorgt ervoor dat alle code die in de index view tussen @section('content') en @endsection staat, wordt ingevoegd op regel 13 van de base view.

Lees dit nog een keer goed door, bekijk de templates en probeer te begrijpen wat er gebeurt.

# Opdracht

Verander de *create* en *update* view ook zodat deze ook gebruik maken van de base view.

Op regel 1 van de *create* en *update* view moet je dus `@extends('base')` plaatsen.

Er zijn nu drie pagina's die allemaal gebruik maken van de base view. Alle pagina's hebben dezelfde <h1></h1> title. Die gaan we aanpassen. De *create*, *update* en *index view* krijgen allemaal een eigen titel.

# Titel aanpassen

In de index.blade.php zetten we vlak voor de `@section('content')` de volgende regel code:

```
@section('title', 'List of Stocks')
```

Dit is eigenlijk hetzelfde als:

```
@section('title')
List of Stocks
@endsection
```

Er is dus nu een nieuwe sectie met allen de string "List of Stocks".

In de base view wordt de titel regel vervangen door:

```
<h1 class="display-3">@yield('title')</h1>
```

Op deze manier wordt de section 'titel' in de <h1> tag geplaatst.

# Opdracht

Als dit werkt dan pas je op dezelfde manier de titel voor de *create* en *update* pagina aan.

| View | Titel |
| --- | --- |
| index | List of Stocks |
| create | New Stock |
| update | Update Stock |

# Test

Controleer goed of alles werkt!

Als het goed is, dan zien de schermen er zo uit:

| | | |
| --- | --- | --- |
| image-1667149687521.png | image-1667149703560.png | image-1667149719155.png |

--

# Extra - Euro-teken en totaal

## Inleiding

In blade views kun je ook PHP-code gebruiken.

In deze les gaan we een de bedragen van alle stocks optellen en onderaan gaan we een totaal telling maken.

Bovendien plaatsen we een euroteken voor alle bedragen.

image-1667153276508.png

## Opdracht, Euro-teken

Het euroteken kun je op verschillende manieren toevoegen aan een balde view. Je kunt het euro tekens ergens vandaag kopiëren of je kunt de HTML-code voor het euroteken gebruiken.

Verander de index view zodat voor alle bedragen een euro teken wordt geplaatst. Net zoals in het voorbeeld hierboven.

## PHP code in Blade views

Om het totaal aan stock values te berekenen moeten we alle waarden bij elkaar optellen.

Dat kan via de controller, maar het kan ook in de blade view met PHP-code.

We beginnen met een variabele op 0 te zetten, dit doen we in de index view voor de loop.

```
@php( $sum=0 )
```

In de loop gaan we dan telkens de $sum ophogen met de waarde van de stock-value.

Dit doen we dus ***in*** de foreach-loop!

```
@php( $sum+=$stock->value )
```

Als de loop is afgelopen dan bevat de variabele $sum dus de waarde van alle stocks.

Plaats een extra regel in de tabel en druk onder de kolom Stock Value de waarde van $sum af.

De waarde van een variabele kan je in een balde template afdrukken met:

```
{{ $sum}}
```

Vergeet het Euroteken niet!

--

# Debug (nog uitwerken)

https://github.com/barryvdh/laravel-debugbar

# Evaluatie

Maak een kort documentje waarin je twee voordelen en twee nadelen van het gebruik van Yii ten opzichte van 'vanilla' PHP (php zonder framework)  beschrijft.

Beschrijf verder welke twee tips jij studenten wilt geven die nog aan de Yii module moeten beginnen.

Je kunt onderstaande template gebruiken.

> Evaluatie-Yii.docx

Noem het documentje evaluatie-jouw-naam.pdf

--