



**RemindMe**

# Inhoud

1. Inleiding .....	3
2. Samenvatting .....	4
3. Database.....	5
4. Backend .....	7
4.1 Gebruikte technologieën .....	7
4.2 Structuur.....	8
4.3 Models & Repositories .....	9
4.4 Controllers.....	13
4.5 Views .....	19
4.6 Authenticatie .....	21
4.7 Berichten service .....	21
5. Frontend .....	22
5.1 Gebruikte technologieën .....	22
5.2 Structuur.....	22
6. Conclusie .....	23

# 1. Inleiding

RemindMe is een webapplicatie waarmee gebruikers een herinnering kunnen maken, die ze later dan per sms toegestuurd krijgen. Er kan een account aangemaakt worden. Dit biedt onder andere de mogelijkheid om contacten te beheren en herinneringen te laten herhalen. Registreren is echter niet nodig, er kon ook een zogenaamde 'Quick reminder' gemaakt worden. Het nadeel hiervan is dat de prijs veel hoger ligt.

De applicatie is geschreven in Laravel. Wanneer een gebruiker zich inlogt, komt hij terecht bij het 'dashboard'. Dit deel van de site maakt gebruik van het JavaScript framework Vue.js. Deze communiceert met de backend door middel van een API.

## 2. Samenvatting

Zie inleiding.

### 3. Database

De applicatie maakt gebruik van een MySQL (MariaDB) database. De structuur ziet er als volgt uit:

<b>users</b> 🔑 id : int(11) 📄 name : varchar(255) 📄 email : varchar(255) 📄 password : varchar(255) # reminder_credits : int(11) 📄 remember_token : varchar(100) 🕒 created_at : datetime 🕒 updated_at : : datetime	<b>user_reminders</b> 🔑 id : int(11) 📄 recipient : varchar(255) # contact_id : int(11) 🕒 send_datetime : datetime 📄 message : text # user_id : int(11) # repeat_id : int(11) 🕒 deleted_at : : datetime	<b>user_orders</b> 🔑 id : int(11) 📄 payment_id : varchar(255) # user_id : int(11) # amount : decimal(8,2) # reminder_credits : int(11) 🕒 created_at : : datetime 🕒 updated_at : : datetime 🕒 deleted_at : : datetime
<b>quick_reminders</b> 🔑 id : int(11) 📄 recipient : varchar(255) 🕒 send_datetime : datetime 📄 message : text 🕒 created_at : : datetime 🕒 updated_at : : datetime 🕒 deleted_at : : datetime # is_paid : : tinyint(1) 📄 payment_id : varchar(255)	<b>contacts</b> 🔑 id : int(11) 📄 name : varchar(255) 📄 number : varchar(20) # user_id : int(11) <b>repeats</b> 🔑 id : int(11) 📄 repeat_interval : varchar(20)	<b>password_resets</b> 🔑 id : int(11) 📄 email : varchar(255) 📄 token : varchar(255) 🕒 created_at : datetime

#### Users:

Deze tabel bevat details over de gebruiker, samen met een aantal timestamps. Het wachtwoord wordt gehashed in de database opgeslagen. Het veld "reminder\_credits" bevat het aantal reminders dat de gebruiker nog heeft.

#### User\_reminders

Bevat reminders die door een geregistreerde gebruiker gemaakt zijn. Zo'n reminder wordt ofwel naar een opgeslagen contact verstuurd, of naar een willekeurig nummer. Daarom heeft deze tabel naast het veld "recipient" ook nog een referentie naar de Contacts tabel.

#### User\_orders

Bevat gegevens over betalingen die door geregistreerde gebruikers worden uitgevoerd. Zo wordt het bedrag opgeslagen, en het aantal reminders dat gekocht werd. "Payment\_id" verwijst naar de betaling die via de Mollie API wordt gemaakt. Dit kan gebruikt worden om de betalingsstatus na te gaan.

### **Quick\_reminders**

Hier worden alle details opgeslagen over reminders die door niet-geregistreerde gebruikers worden gemaakt. Ook hier wordt een referentie naar de betaling van Mollie bijgehouden. Ook is er een boolean voorzien om na te gaan of de betaling al voltooid is of niet. Dit om ervoor te zorgen dat er niet elke minuut een request naar de Mollie API gestuurd dient te worden.

### **Contacts**

De tabel om contacten in op te slaan. Elke rij bevat een verwijzing naar de user waarmee ze geassocieerd is.

### **Password\_resets**

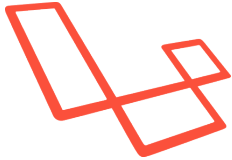
Deze tabel is nodig om gebruikers de mogelijkheid te geven hun wachtwoord te veranderen.

### **Repeats**

Dit is niet meer dan een enum voor de beschikbare herhalingen.

## 4. Backend

### 4.1 Gebruikte technologieën



#### Laravel 5.2

Laravel is het meest gebruikte PHP framework. Het bevat veel tools die het ontwikkelingsproces helpen versnellen. Zo maakt het gebruik van artisan om code te genereren voor veelgebruikte zaken. De Eloquent ORM vergemakkelijkt database operaties, en het Blade templating systeem zorgt voor overzichtelijke views. Bovenop dit alles beschikt Laravel over een expressieve en elegante syntax.

Persoonlijk heb ik voor Laravel gekozen omwille van de gelijkenissen met ASP.NET, een C# framework waar ik al eerder ervaring in had.

**Alternatieven:** CodeIgniter, CakePHP, Symfony



#### Twilio

Om SMS-berichten te versturen kunnen maakt de applicatie gebruik van Twilio. Deze service voorziet een API, waardoor een bericht verstuurd kan worden door middel van een HTTP request. Twilio voorziet ook een library die beschikbaar is via Composer. Dit zorgt ervoor dat de service snel opgezet kan worden.

**Alternatieven:** Plivo, Nexmo



#### Mollie

Mollie is een service om online betalingen te regelen. Het voorziet veel betalingsmogelijkheden, gaande van Bancontact tot Paypal en zelfs bitcoin. Mollie host een API en voorziet hiervoor ook een PHP library, wat maakt dat er met een paar regels code al een betaling kan worden uitgevoerd.

Voor de ontwikkeling van een applicatie kan een test-API gebruikt worden. Deze stelt de developer in staat om betalingen na te bootsen, zonder deze effectief uit te voeren.

**Alternatieven:** Adyen, Ingenico



### JSON Web Token

De applicatie maakt hevig gebruik van een API. Een RESTful API is per definitie stateless. Dit wil zeggen dat session-based authenticatie geen goede optie is. JSON Web Token (JWT) biedt hiervoor een oplossing. Een token bevat geëncrypteerde authenticatiegegevens van de gebruiker, en wordt client-side bijgehouden. Dit token wordt meegegeven in de header van elk request naar de API, om zo de gebruiker te kunnen authenticeren.

De applicatie maakt gebruik van de opensource `tymon/jwt-auth` library, specifiek voor Laravel.

**Alternatieven:** OAuth, basic

## 4.2 Structuur

Het informatieve deel van de applicatie is een 'normale' website. Het gaat over de homepage, de contactpagina en de pricing pagina. Dit zijn vrijwel volledig statische pagina's.

Het dashboard daarentegen is een zogenaamde 'Single Page Application (SPA)'. Bij het navigeren naar het dashboard wordt eenmalig alle nodige code geladen. Een JavaScript framework zorgt ervoor dat alle requests asynchroon geladen worden. Ook de routing wordt afgehandeld door het framework. Hierdoor hoeft de gebruiker nooit rechtstreeks een nieuwe pagina op te vragen of te herladen. Dit zorgt voor een vlotte user experience.

Om dit allemaal te kunnen voorzien maakt de applicatie gebruik van een RESTful API.

De API voorziet een aantal endpoints om met de applicatie te werken. Ze maakt gebruik van de zogenaamde 'HTTP werkwoorden' die overeenstemmen met een CRUD operatie. Zo staat een POST request voor een insert, GET voor een read, PUT voor een update, en een DELETE request zorgt voor een delete.

Deze requests worden gebundeld onder de 'api' route. Om bijvoorbeeld alle contacten op te vragen, dient er een GET request gestuurd te worden naar `../api/contacts`. De API maakt gebruik van JSON als dataformaat. De server stuurt ook logische HTTP statuscodes. Een niet gevonden record geeft bijvoorbeeld een 404 als antwoord, of een niet geautorizeerde actie levert een 403 op.



Endpoint	Method	Omschrijving
/contacts/{id?}	GET	Antwoord met een array van alle contactpersonen van de geauthenteerde gebruiker, of met slechts 1 contactpersoon als er een id wordt gegeven
/contacts	POST	Insert een contactpersoon aan de hand van de POST-gegevens.
/contacts/{id}	DELETE	Delete de contactpersoon met het gegeven ID.
/contacts	PUT	Update de contactpersoon aan de hand van de gegevens.
/reminders/upcoming	GET	Antwoord met een array van alle reminders met een datum later dan vandaag.
/reminders/cancel/{id}	GET	Annuleert de reminder met het gegeven ID.
/reminders	POST	Insert een nieuwe reminder aan de hand van de POST-gegevens.
/quickreminder	POST	Insert een quick reminder aan de hand van de POST-gegevens.
/login	POST	Probeert een gebruiker te authenticeren. Antwoord met een JWT als de authenticatie succesvol is
/register	POST	Registreert een nieuwe gebruiker.
/user	GET	Antwoord met enkele gegevens van de geauthenteerde gebruiker.

## 4.3 Models & Repositories

### Eloquent ORM

Laravel maakt gebruik van het Eloquent ORM. Elk Eloquent model komt overeen met een tabel in de database. Op dit model kunnen dan relaties gedefiniëerd worden. Standaard komt de classnaam overeen met de tabelnaam. Zo zal het model Contact queries uitvoeren op de 'contacts' tabel. Het Contact model ziet er bijvoorbeeld als volgt uit:

```
class Contact extends Model
{
    public $timestamps = false;

    public function user()
    {
        return $this->belongsTo('User');
    }
}
```

De `$timestamps` property wordt gebruikt om aan te duiden of er voor een model de datum moet worden bijgehouden waarop het aangemaakt of gewijzigd wordt.

Relaties met andere tabellen worden aangeduid aan de hand van functies. Zo wordt bijvoorbeeld een OneToMany relatie uitgedrukt door `hasMany()` en `belongsTo()`.

Eloquent zorgt ervoor dat veelgebruikte queries geëncapsuleerd worden op de models. Zo voorziet het enkele logische functies zoals `delete()`, `save()`, `where()`, enz. Meer complexe operaties kunnen nog altijd in rauwe SQL worden uitgevoerd.

### Repository pattern

De applicatie maakt gebruik van het repository pattern. Dit is een extra abstractie van de database layer. Een repository bundelt een aantal operaties op een model in een class. De controller hoeft dan niet langer rechtstreeks de database layer aan te spreken via de models, maar gebruikt in de plaats de repositories.

De repository en het model zijn zo echter wel nog tightly coupled. Om dit tegen te gaan worden interfaces gebruikt. De interface definiëert een 'contract' waar de repository zich aan moet houden. In de controller wordt vervolgens de interface als dependency gebruikt. Nu maakt het niet meer uit hoe de data opgehaald wordt, als de implementatie zich maar aan de interface houdt.

Het gebruik van repositories zorgt niet enkele voor beter leesbare code, maar verbetert ook de mogelijkheid tot unit testing.

De interface voor de contactpersonen ziet er bijvoorbeeld als volgt uit:

```
interface IContactRepository
{
    public function getContactById($contactid);

    public function getContactsByUserId($userid);

    public function insertContact($contact);

    public function deleteContact($contactid);

    public function updateContact($id, $newValues);
}
```

Vervolgens implementeert de repository de bijhorende interface. Zo ziet de `getContactsByUserId()` er als volgt uit.

```
namespace App\Repositories\Contact;

use App\Models\Contact;

class ContactRepository implements IContactRepository
{
    private $_contact;

    public function __construct(Contact $contact)
    {
        $this->_contact = $contact;
    }

    ...

    public function getContactsByUserId($userid)
    {
        return $this->_contact->where('user_id', $userid)->get();
    }

    ...
}
```

Om de repository te gebruiken wordt de interface geïnjecteerd in de constructor.

```
public function __construct(IContactRepository $contact)
{
    $this->_contactRepository = $contact;
}
```

Omdat enkel de interface wordt gebruikt in de controller, weet Laravel niet welke implementatie ze hiervoor moet gebruiken. Daarom moeten de interface en de bijhorende implementatie aan elkaar gekoppeld worden. Dit gebeurt in de `register` method in de `AppServiceProvider`.

```
$this->app->bind('App\Repositories\Contact\IContactRepository',
    'App\Repositories\Contact\ContactRepository');
```

### Structuur

Elk model krijgt op deze manier zijn eigen repository. Omdat dit project relatief klein is, heeft elke repository zijn eigen interface. Een betere oplossing voor grotere projecten is om een basis interface te schrijven met algemene CRUD acties. Vervolgens kunnen specifieke interfaces deze erven en eventueel nog methods toevoegen of aanpassen.

De folderstructuur voor de models ziet er als volgt uit:

```
app/
...
.
|-- Models/
|   |-- Contact.php
|   |-- Quick_reminder.php
|   |-- User_order.php
|   |-- User_reminder.php
|   `-- User.php
`-- Repositories/
    |-- Contact/
    |   |-- ContactRepository.php
    |   `-- IContactRepository.php
    |-- Quick_reminder/
    |   |-- Quick_reminderRepository.php
    |   `-- IQuick_reminderRepository.php
    |-- User_order/
    |   |-- User_orderRepository.php
    |   `-- IUser_orderRepository.php
    |-- User_reminder/
    |   |-- User_reminderRepository.php
    |   `-- IUser_reminderRepository.php
    `-- User/
        |-- UserRepository.php
        `-- IUserRepository.php
```

Alle repositories hebben enkele basis CRUD methods. De enige repository die hiervan afwijkt is de UserRepository. De `tymon/jwt-auth` library gebruikt standaard het zijn eigen `UserInterface`. Daarom moet de UserRepository een extra interface implementeren. Ook moet in `config/jwt.php` een verwijzing toegevoegd worden naar de nieuwe implementatie.

```
namespace App\Repositories\User;

use App\Models\User;
use Tymon\JWTAuth\Providers\User\UserInterface;

class UserRepository implements IUserRepository, UserInterface
{
    ...
}
```

## 4.4 Controllers

De controllers worden opgesplitst in twee delen. Enerzijds zijn er de controllers die verantwoordelijk zijn voor de API. Deze bevatten allerlei methods om de gewenste database operaties uit te voeren. Anderzijds zijn er de controllers die de webpagina's genereren. Deze doen vaak niets anders dan een view returnen.

```
Controllers/  
|-- API/  
    |-- AuthenticateController.php  
    |-- ContactsController.php  
    |-- RemindersController.php  
    `-- UserController.php  
|-- Controller.php  
|-- DashboardController.php  
|-- HomeController.php  
`-- PaymentController.php
```

### ContactsController

De ContactsController is niet meer dan een bundeling van CRUD operaties. De methods krijgen dan ook namen die overeenstemmen met de actie.

```
namespace App\Http\Controllers\API;  
  
use App\Http\Requests;  
use ...  
  
class ContactsController extends Controller  
{  
    private $_contactRepository;  
    private $_userRepository;  
    private $_userReminderRepository;  
  
    private $_user;  
  
    public function __construct(..) { .. }  
  
    public function get(..) { .. }  
  
    public function insert(..) { .. }  
  
    public function delete(..) { .. }  
  
    public function update(..) { .. }  
}
```

In de constructor worden de nodige repositories via dependency injection geïnstantieerd. Deze worden dan als properties bijgehouden om zo gebruikt te kunnen worden in de methods.

```
public function __construct(IContactRepository $contact, ...)  
{  
    $this->_user = JWTAuth::parseToken()->toUser();  
  
    $this->_contactRepository = $contact;  
    $this->_userRepository = $user;  
    $this->_userReminderRepository = $userReminder;  
}
```

De ContactsController kan alleen maar gebruikt worden als de user geauthenticeerd is, en gebruikt daarom de `jwt.auth` middleware. Deze is gedefinieerd in `routes.php`. De geauthenticeerde gebruiker wordt via de `toUser()` method in een property gestoken.

Vrijwel alle methods van de controller volgen de zelfde structuur. Eerst wordt de eventuele input gevalideerd. Vervolgens wordt de logica uitgevoerd die van de method verwacht wordt. Ten slotte wordt er een JSON antwoord teruggestuurd. Moest er ergens in de method iets foutlopen, dan wordt ook een logische HTTP code teruggestuurd. Zo zal foutieve data voor een 422 code zorgen.

Als voorbeeld de `delete()` method:

```
public function delete(Request $request, $id)
{
    // Get the contact.
    $contact = $this->_contactRepository->getContactById($id);

    if(!$contact)
    {
        return response()->json("Not found", 404);
    }
    // Check if the contact really belongs to the user.
    if($contact->user_id != $this->_user->id)
    {
        return response()->json("Not authorized", 403);
    }

    // Remove the contact id from all associated reminders, and
    // set the recipient to the contact's number.
    // This is so the set reminders can still be sent.
    $reminders = $this->_userReminderRepository
        ->getUserRemindersWhere([[ "contact_id", $id ]]);
    foreach($reminders as $reminder)
    {
        $this->_userReminderRepository
            ->updateUserReminder($reminder->id, [
                "recipient" => $contact->number,
                "contact_id" => NULL
            ]);
    }

    // Finally, delete the contact & return the result.
    $result = $this->_contactRepository->deleteContact($id);
    return response()->json($result);
}
```

### RemindersController

De RemindersController is verantwoordelijk voor de operaties op de User\_reminders tabel. Deze is vrijwel analoog aan de ContactsController. Het enige verschil hier is dat er een extra `_createUserReminder()` method is. Deze wordt gebruikt bij `insertReminder()`. Dit is een aparte method om de code overzichtelijker te maken.

```
class RemindersController extends Controller
{
    private $_userRepository;
    private $_userReminderRepository;
    private $_user;

    public function __construct(..) { .. }

    public function getUpcomingReminders() { .. }

    public function insertReminder(..) { .. }

    public function cancelReminder(..) { .. }

    private function _createUserReminder(..) { .. }
}
```

### UserController

De UserController heeft slechts één method, `getUserDetails()`. Deze wordt gebruikt, zoals de naam al zegt, om enkele details van de gebruiker op te vragen. Uiteraard gebruikt deze controller ook de `jwt.auth` middleware.

### AuthenticateController

Deze controller zorgt voor de authenticatie en registratie van de gebruiker. `Authenticate()` probeert aan de hand van de logingegevens een JWT token aan te maken. Als de authenticatie succesvol is, worden er, samen met het token, enkele details over de gebruiker meegegeven. Dit om een request te besparen.

De `register()` method wordt gebruikt om een gebruiker te registreren.

```
class AuthenticateController extends Controller
{
    private $_userRepository;

    public function __construct(..) { .. }

    public function authenticate(..) { .. }

    public function register(..) { .. }
}
```

### DashboardController

Het enige wat deze controller doet, is een view returnen voor het dashboard.

### HomeController

De HomeController zorgt voor alle pagina's van de 'gewone' website: de homepage, de pricing pagina, en de contactpagina. Ook heeft de controller een method om het verzenden van het contactformulier af te handelen.



## PaymentController

De PaymentController is vrijwel de meest uitgebreide controller. De controller zorgt voor zowel het kopen van quick reminders alsook het kopen van credits door een gebruiker. Ook zorgt deze controller voor het aanmaken van de quick reminder wanneer de betaling succesvol is afgehandeld.

```
namespace App\Http\Controllers;

use ...;

class PaymentController extends Controller
{
    private $_mollie;
    private $_userOrderRepository;
    private $_userRepository;
    private $_quickReminderRepository;

    public function __construct(..) { .. }

    public function createUserOrder(..) { .. }

    public function userOrderRedirect(..) { .. }

    public function createQuickReminderOrder(..) { .. }

    public function quickReminderOrderRedirect(..) { .. }
}
```

Een betaling bestaat altijd uit twee delen, de `create` en de `redirect`. In de `create` wordt eerst de input gevalideerd. Vervolgens wordt een order aangemaakt. Deze gegevens worden hierna naar de Mollie API verstuurd. Het order wordt hierna geüpdate met het ID van de betaling. Ten slotte wordt de gebruiker geredirect naar de website van Mollie om te betaling te voltooien.

Wanneer de gebruiker de betaling succesvol afhandelt, wordt hij doorverwezen naar de `redirect` method. Hier wordt nagegaan of de betaling gelukt is. Vervolgens wordt het order geupdate, en wordt de actie uitgevoerd waar de gebruiker voor betaald heeft.

In de constructor wordt eerst de Mollie API library geïntanceerd. Dit gebeurt aan de hand van de API key. Vervolgens worden ook alle nodige repositories geladen.

```
public function __construct(IUserRepository $userRepository, ...)
{
    $this->_mollie = new Mollie_API_Client;
    $key = env('MOLLIE_API_KEY');
    $this->_mollie->setApiKey($key);

    $this->_userOrderRepository = $userOrderRepository;
    $this->_userRepository = $userRepository;
    $this->_quickReminderRepository = $quickReminderRepository;
}
```

Als voorbeeld het aanmaken van een quick reminder order:

```
public function createQuickReminderOrder(Request $request)
{
    $this->validate($request, [
        'recipient' => 'max:255|required',
        'send_datetime' => 'required',
        'message' => 'required|max:255'
    ]);

    // Set up the new reminder.
    $values = [
        "recipient" => $request->recipient,
        "send_datetime" => $request->send_datetime,
        "message" => $request->message,
        "is_payed" => false
    ];

    // Insert the reminder & get the new ID.
    $identity = $this->_quickReminderRepository
        ->insertQuickReminder($values);

    // Create the Mollie payment.
    $payment = $this->_mollie->payments->create(array(
        "amount" => 0.50,
        "description" => "Your quick reminder.",
        "redirectUrl" => url('/thankyou/'.$identity)
    ));

    // Update the reminder with the payment ID.
    $this->_quickReminderRepository
        ->updateQuickReminder($identity, ["payment_id" => $payment->id]);

    // Redirect the user to the Mollie payment page.
    header("Location: " . $payment->getPaymentUrl());
    exit;
}
```

Nadat de request gevalideerd is, wordt een nieuwe reminder aangemaakt in de database. De boolean `is_payed` wordt op `false` gezet. Hierdoor zal de reminder genegeerd worden door de berichtenservice.

Het ID van de nieuwe reminder wordt meegegeven aan de Mollie API. Ook wordt dit ID verwerkt in de `redirectUrl`. Wanneer de betaling verstuurd is, wordt de reminder geüpdate met de `payment_id`. Dit ID wordt gebruikt om later de betaalstatus op te vragen.

Als de betaling is uitgevoerd, wordt de gebruiker geredirect naar de opgegeven URL. Aan de hand van de orderID in deze URL wordt de reminder terug opgevraagd uit de database, en wordt de betalingsstatus nagekeken. Is alles goed verlopen, dan wordt de boolean `is_payed` op `true` gezet, en krijgt de gebruiker te zien dat de betaling gelukt is. De reminder is nu klaar om verstuurd te worden.

Als er iets is misgelopen, of als de betaling nog niet voltooid is, dan krijgt de gebruiker dit ook te zien.

```
$payment = $this->_mollie->payments->get($reminder->payment_id);

// Check if the payment is successful.
if ($payment->isPaid())
{
    // Update the reminder & set the message.
    $this->_quickReminderRepository
        ->updateQuickReminder($reminder->id, ["is_payed" => true]);
    $data["message"] = "We received your payment successfully!";
}
```

## 4.5 Views

### Blade templates

Alle pagina's maken gebruik van het Blade templating systeem van Laravel. Dit is een systeem dat veel gelijkenissen toont met Razor in ASP.NET. Het zorgt ervoor dat er gemakkelijk layouts gedefiniëerd kunnen worden, en dat scripts en styles op een overzichtelijke manier kunnen worden ingeladen.

Inline PHP code ziet er ook een stuk beter uit door het gebruik van accolades in plaats van `<?php?>` tags.

### Structuur

De views worden gesorteerd in folders. Elk deel van de site krijgt zijn eigen folder, alsook de errorpagina's en de gedeelde layouts. Vue.js maakt ook gebruik van een aantal templates, dat hier ook een eigen folder krijgt

```
views/
|-- dashboard/
    |-- home.blade.php
|-- errors/
    |-- 404.blade.php
    |-- 503.blade.php
|-- home/
    |-- contact.blade.php
    |-- index.blade.php
    |-- paymentcomplete.blade.php
    |-- pricing.blade.php
|-- layouts/
    |-- site.blade.php
|-- vue/
    |-- ...
```

## Layouts

De website maakt gebruik van een Blade layout. Dit zorgt ervoor dat gemeenschappelijke delen, zoals header of footer, maar op één plaats aangepast hoeven te worden. De verschillende pagina's 'extenden' dan de layout. Scripts of stylesheets die specifiek zijn voor een pagina kunnen in sections geplaatst worden.

De structuur van de layout ziet er als volgt uit:

```
<!DOCTYPE html>
<html>
<head>
    ...
    @yield('style')
</head>

<body>
    <header>...</header>
    <main>
        @yield('content')
    </main>
    <footer>...</footer>

    @yield('scripts')
</body>
</html>
```

Een pagina ziet er dan als volgt uit: eerst wordt aangeduid welke layout gebruikt wordt, en vervolgens wordt voor elke section specifieke code geschreven. Moest een pagina bijvoorbeeld geen speciaal script nodig hebben, dan kan deze section gewoon weggelaten worden.

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

@section('style')
    <link rel="stylesheet" href="{{ url('css/main.css') }}">
@endsection

@section('content')
    <!-- Page specific HTML -->
@endsection

@section('scripts')
    <script src="myscript.js"></script>
@endsection
```

De views in de Vue folder worden nog verder besproken in het deel frontend.

## 4.6 Authenticatie

De applicatie maakt gebruik van een RESTful API. Zo'n API is per definitie stateless. Concreet wil dit zeggen dat elk request alle nodige informatie bevat om afgehandeld te worden door de server. Dit betekent dat de API technisch gezien geen gebruik mag maken van session based authenticatie. Alternatieven zoals OAuth en JSON Web Tokens (JWT) bieden hiervoor een oplossing.

### JSON Web Token

## 4.7 Berichten service

Scheduler, reminders:check command

# 5. Frontend

## 5.1 Gebruikte technologieën

VueJS, grid, fa, ....

## 5.2 Structuur

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Dolorum unde, pariatur dicta ipsa commodi, sit quasi autem blanditiis nesciunt ullam harum accusantium neque in incidunt modi labore quis nulla iure.

## 6. Conclusie

...