Mark Gerarts

21 april 2016

Opleiding Webontwikkelaar 2015-2016

Eindwerk

Inhoud

1. Samenvatting 3
2. Inleiding 4
3. Database 5
4. Backend 7
   4.1 Gebruikte technologiën 7
   4.2 Structuur 8
   4.3 Models & Repositories 9
   4.4 Controllers 13
   4.5 Views 19
   4.6 Authenticatie 21
   4.7 Berichtenservice 22
   4.8 Unit Testing 25
5. Frontend 27
   5.1 Gebruikte technologiën 27
   5.2 Vue.js 27
   5.3 Routing 29
   5.4 Mixins 30
   5.5 Stores 33
   5.6 Dashboard Home 36
   5.7 Contacts 40
   5.8 Account 41
   5.9 Authenticatie 41
   5.10 Datetime picker 42
6. Werkwijze 46
7. Conclusie 47

1. Samenvatting

RemindMe is een webapplicatie waarmee een gebruiker reminders kan instellen die hij later per SMS toegestuurd krijgt. Registratie is niet nodig, maar geeft wel verschillende voordelen, zoals bijvoorbeeld het bijhouden van contactpersonen.

De applicatie is gebouwd rond het framework Laravel. Er worden ook verschillende third-party libraries aangewend. Om SMS-berichten te versturen wordt de Twilio API gebruikt. Betalingen kunnen uitgevoerd worden dankzij Mollie, en authenticatie verloopt via JSON Web Tokens.

Het dashboard van de applicatie is een zogenaamde 'Single Page Application'. Hiervoor wordt Vue.js gebruikt, een JavaScript framework dat is geïnspireerd door AngularJS en React. Dit betekent ook dat de applicatie een API voorziet, om via AJAX HTTP requests te kunnen uitvoeren.

Er is getracht de applicatie zo volledig mogelijk te bouwen. Zo wordt de API voorzien van unit tests, en krijgen de models een extra abstractielaag in de vorm van repositories.

2. Inleiding

RemindMe is een webapplicatie waarmee gebruikers een reminder kunnen maken, die ze later dan per SMS toegestuurd krijgen. Er kan ook 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 kan ook een zogenaamde 'Quick reminder' gemaakt worden. Het nadeel hiervan is dat de prijs hoger ligt.

De applicatie is geschreven in Laravel. Dit is een populair PHP framework dat dankzij de vlotte syntax en vele features zorgt voor snelle ontwikkeling en duidelijke code.

Omdat het zo'n veelgebruikt framework is, zijn er ook vele libraries beschikbaar. Deze kunnen snel en makkelijk geïnstalleerd worden via de package manager van PHP, Composer. De applicatie maakt dan ook gebruik van een aantal libraries. Zo is er de Twilio API om berichten te versturen, jwt-auth om authenticatie te vergemakkelijken, en Mollie om betalingen uit te voeren.

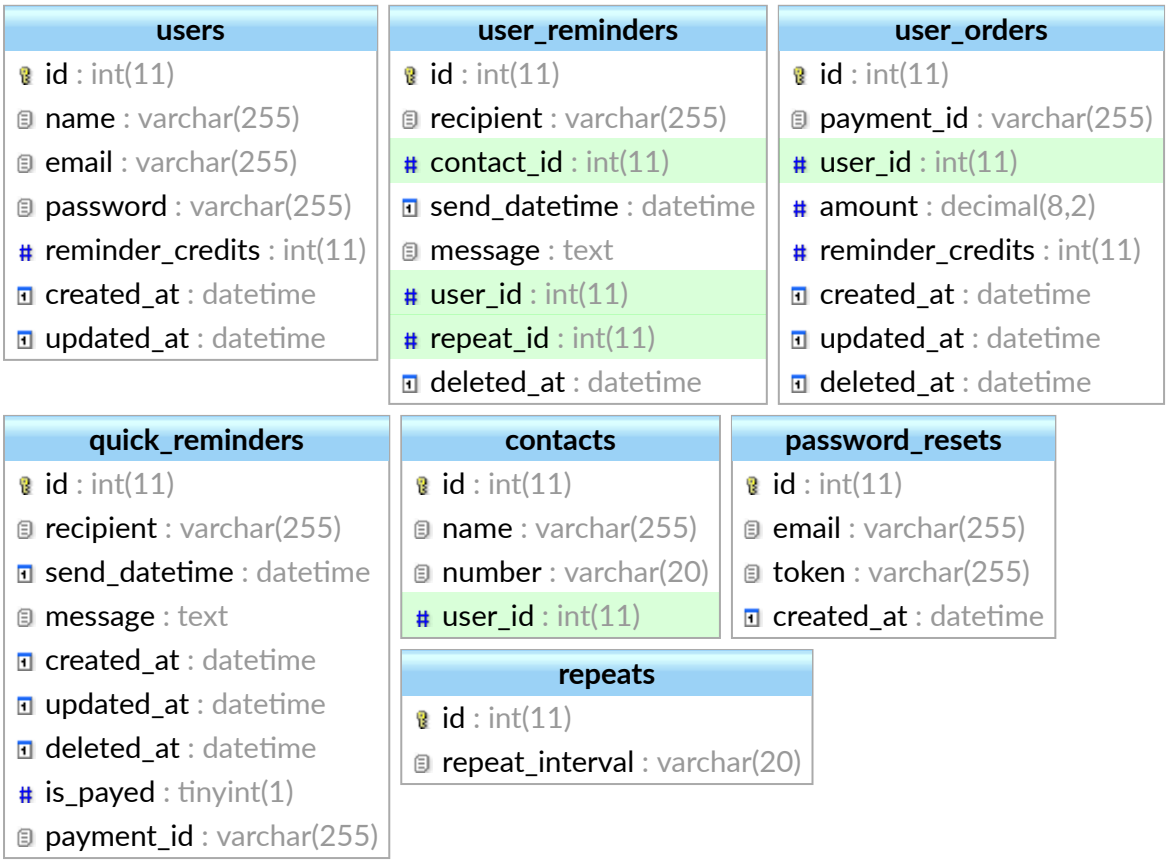
Het dashboard van de applicatie is, net zoals vele andere moderne applicaties, een SPA. Dit wil zeggen dat wanneer de gebruiker naar het dashboard surft, er direct alle nodige code wordt geladen. Alle volgende interacties worden in de achtergrond geregeld via AJAX requests. Dit zorgt voor een vlotte gebruikservaring waarbij de site niet meer herladen hoeft te worden.

Om de SPA te maken wordt er gebruik gemaakt van Vue.js, een JavaScript framework dat veel lijkt op AngularJS en React. Het is een MV* framework dat het onder andere mogelijk maakt om databinding toe te passen op HTML elementen. Het grote voordeel van dit framework is de intuïtieve syntax.

Om de verschillende AJAX request af te kunnen handelen, voorziet de applicatie ook een API. Er is getracht deze zo 'RESTful' mogelijk te maken. Zo worden de gepaste werkwoorden gebruikt in de requests: GET voor een Read, PUT voor een update, enz. Ook worden JSON Web Tokens gebruikt om de API stateless te maken.

3. Database

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



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.

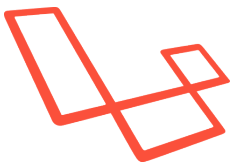
Contacts
De tabel om contacten in op te slaan. Elke rij bevat een verwijzing naar de user waarmee ze geässocieerd is.

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 technologiën



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 kunnen versturen 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 voor een delete.

De requests naar de API 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 geauthorizeerde actie levert een 403 op.

Endpoint	Method	Omschrijving
/contacts/{id?}	GET	Antwoord met een array van alle contactpersonen van de geauthenticeerde 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 geauthenticeerde gebruiker.

4.3 Models & Repositories

Eloquent ORM

Laravel maakt gebruikt 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 data access 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.

```php
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.

```php
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`.

```php
$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 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.

```php
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 allerhande 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.

```php
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 gebruiker geauthenticeerd is, en gebruikt daarom de `jwt.auth` middleware. Deze is gedefiniëerd in `routes.php`. De geauthenticeerde gebruiker wordt via de `toUser()` method in een property gestoken.

Vrijwel alle methods van de controller volgen dezelfde 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 geüpdate, en wordt de actie uitgevoerd waar de gebruiker voor betaald heeft.

In de constructor wordt eerst de Mollie API library geïnstantieerd. 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 het 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 gebruik geredirect naar de opgegeven URL. Aan de hand van het 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 geplaats 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. De applicatie maakt gebruik van dit laatste.

JSON Web Token
Een JSON Web Token is een compact token dat mee verstuurd kan worden in de header van een request. Het wordt serverside aangemaakt en geëncrypteerd aan de hand van een secret key. Hierdoor kan de authenticiteit geverifiëerd worden.

Een token bestaat uit 3 delen: header, payload en signature. Deze worden gescheiden door een punt. Een token heeft dus de volgende vorm:
`xxxx.yyyy.zzzz`

De header is een JSON object dat doorgaans bestaat uit twee properties: het type token, en het soort hashing algoritme. Dit object wordt vervolgens Base64 geëncodeerd.

Het tweede deel, de payload, bevat het grootste deel van de informatie. Ook dit is een JSON object dat Base64 geëncodeerd wordt. Het bevat enkele vaste properties, zoals bijvoorbeel de issuer (`iss`) en de expiration date (`exp`). Naast deze properties kunnen er nog custom properties aangemaakt worden, die bijvoorbeeld meer informatie over de gebruiker bevatten.

Het laatste deel is de signature. Dit is een geëncrypteerde vorm van de header en de payload, aan de hand van een secret key. Deze signature wordt gebruikt om de authenticiteit te verifiëren. Het bevat de header en de payload om ervoor te zorgen dat deze niet gewijzigd kunnen worden.

Een token kan er dan uitendelijk als volgt uitzien:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

JWT Auth
De applicatie maakt gebruikt van de `tymon/jwt-auth` library. Deze open-source library is specifiek voor Laravel geschreven en biedt alle nodige middelen om met JWT te werken. Zo voorziet het bijvoorbeeld een method om van een user een token te maken met `JWTAuth::fromUser()`, of de `make()` method om een eigen payload te maken.

De library voorziet ook een eigen authenticatie middleware. Deze middleware zal het token uit een request halen en proberen een gebruiker te authenticeren. Alle API controllers die authenticatie nodig hebben worden dan ook gegroepeerd onder deze middleware. Dit gebeurt in het `routes.php` bestand.

```
Route::group(['namespace' => 'API', 'prefix' => 'api'], function() {
    Route::group(['middleware' => 'jwt.auth'], function() {
        // Routes die authenticatie nodig hebben.
    });
    // Overige API routes
});
```

4.7 Berichtenservice

Om reminders per SMS te versturen maakt de applicatie gebruik van een Laravel command. Dit is een class die zowel in de applicatie gebruikt kan worden, alsook via de console. Zo kunnen de reminders verstuurd worden via het volgende console command:

```
$ php artisan reminders:check
```

Via een cronjob wordt dit command elke minuut uitgevoerd. De class wordt opgeslagen in `app/Console/Commands` en ziet er als volgt uit:

```
class CheckReminders extends Command
{
    protected $signature = 'reminders:check';

    protected $description = 'Checks if there are ...';

    private $_quickReminderRepository;
    private $_userReminderRepository;
    private $_userRepository;
    private $_contactRepository;

    public function __construct(..) { ... }

    public function handle() { ... }

    private function _sendReminder(..) { ... }

    private function _handleRepeatedReminder(..) { ... }

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

In de constructor worden weer de gebruikte repositories geïnjecteerd. De kern van de logica bevindt zich in de `handle()` method. Hier worden eerst alle reminders opgehaald die een `send_datetime` hebben eerder dan de huidige datetime. Voor elk van deze berichten wordt dan een SMS verstuurd met behulp van de Twilio API. De Twilio library zorgt ervoor dat dit in één regel code kan:

```
Twilio::message($reminder->recipient, $reminder->message);
```

Naast het versturen van het bericht, wordt er ook nog een log hiervan gemaakt. Dit gebeurt allebei in de `_sendReminder()` method.
Voor quick reminders houdt het er hierbij op. Voor user reminders daarentegen moet er ook nog gekeken worden of de reminder herhaalt dient te worden. Als dit het geval is, wordt er een nieuwe reminder aangemaakt met een latere `send_datetime`. Het gebruik van de PHP `DateTime` class zorgt dat dit in duidelijke code kan gebeuren.

```
$newDate = new \DateTime($reminder->send_datetime);
switch($reminder->repeat_id)
{
    case 2: // Daily.
        $newDate->add(new \DateInterval("P1D"));
        break;
    case 3: // Weekly.
        $newDate->add(new \DateInterval("P7D"));
        break;
    case 4: // Monthly.
        $newDate->add(new \DateInterval("P1M"));
        break;
    case 5: // Yearly.
        $newDate->add(new \DateInterval("P1Y"));
        break;
}
```

De `_getRecipient()` method is een helperfunctie die nagaat of de reminder een `contact_id` heeft. Als dit zo is, dan wordt het nummer van de bijhorende contactpersoon opgehaald.

Scheduler

Laravel voorziet een Task Scheduler. Hier maakt de applicatie dan ook gebruik van. Om van de scheduler gebruik te kunnen maken, moet er een cronjob opgesteld worden op de server:

```
* * * * * php /path/to/artisan schedule:run >> /dev/null 2>&1
```

Deze cronjob voert elke minuut het artisan command `schedule:run` uit. Wat dit command doet wordt bepaald in `app/Console/Kernel.php`. Omdat we een apart command hebben geschreven voor de berichtenservice, is deze class heel compact:

```php
namespace App\Console;

use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;

class Kernel extends ConsoleKernel
{
    /**
     * The Artisan commands provided by your application.
     *
     * @var array
     */
    protected $commands = [
        Commands\CheckReminders::class
    ];

    /**
     * Define the application's command schedule.
     *
     * @param  \Illuminate\Console\Scheduling\Schedule  $schedule
     * @return void
     */
    protected function schedule(Schedule $schedule)
    {
        // Checks if there are reminders that need to be sent.
        $schedule->command('reminders:check')->everyMinute();
    }
}
```

4.8 Unit Testing

Alle API controllers worden voorzien van unit tests. De pagina's van de website krijgen ook een aantal kleine integratie testen. Op deze manier wordt bevestigd dat de applicatie werkt zoals verwacht.

Om de testen uit te voeren wordt er gebruik gemaakt van PHPUnit. Dit is standaard opgenomen in Laravel. Buiten de verschillende methods van PHPUnit, biedt Laravel zelf ook nog enkele nuttige functies om een applicatie te testen.

Webpagina's
Om de webpagina's de testen zijn er enkele kleine testen voorzien. Zo wordt voor alle links op de site nagegaan of ze ook naar de jusite pagina verwijzen. De expressieve syntax van Laravel zorgt ervoor dat de tests makkelijk te schrijven en te lezen zijn.

```php
$this->visit('/')
    ->click('Pricing')
    ->seePageIs('/pricing');
```

Met behulp van de `see()` method wordt er ook nagegaan of de verwachte informatie op de pagina staat.

API
Elke route van de API wordt getest. Buiten positief testen, wordt er ook zoveel mogelijk negatief getest. Het aanmaken van een reminder wordt bijvoorbeeld op de volgende manier getest.

```
$reminder = factory(App\Models\User_reminder::class)->make();

$request = [
    "recipient" => $reminder->recipient,
    "send_datetime" => $reminder->send_datetime,
    "message" => $reminder->message,
    "repeat_id" => $reminder->repeat_id
];

$creditsBefore = $this->_user->reminder_credits;

$this->json('POST', 'api/reminders', $request, $this->_headers())
    ->assertResponseOk();

$this->seeInDatabase('user_reminders', [
    "recipient" => $request["recipient"],
    "send_datetime" => $request["send_datetime"],
    "message" => $request["message"],
    "repeat_id" => $request["repeat_id"]
]);

$this->seeInDatabase("users", [
    "id" => $this->_user->id,
    "reminder_credits" => ($creditsBefore - 1)
]);
```

Eerst wordt er een reminder gemaakt met behulp van een `factory`. Dit maakt gebruik van de `Faker` library om voor een model random data te genereren. Aan de hand van deze reminder wordt er een request gestuurd naar de API. Vervolgens wordt er via `seeInDatabase()` nagegaan of de insert ook effectief is uitgevoerd.

Na de positieve test komen er een heel aantal negatieve testen. Hier wordt bijvoorbeeld onvolledige of incorrecte data doorgestuurd.

5. Frontend

5.1 Gebruikte technologiën

Vue.js
Vue is een klein en flexibel JavaScript framework. Het stelt ons in staat om snel interactieve pagina's te maken. Het is vergelijkbaar met andere MV* frameworks zoals Angular of React, en deelt ook veel eigenschappen. Het dashboard van de applicatie is volledig rond dit framework gebouwd.
Alternatieven: AngularJS, React

Bootstrap Grid
Omdat Bootstrap sites vaak erg op elkaar lijken, heb ik ervoor gekozen om de styling van de site volledig zelf te schrijven. Wel gebruik ik het gridsysteem om responsive design te vergemakkelijken.
Op de website van Bootstrap kunnen custom builds compiled worden. De applicatie maakt gebruik van een build die enkel het gridsysteem bevat, met wat kleine aanpassingen.
Alternatieven: Foundation, Kickstart

Font Awesome
Font Awesome is een populair icon font. Het zorgt voor kwalitatieve icons, die gemakkelijk zijn in het gebruik. Font Awesome voorziet zelfs handige CSS classes om bijvoorbeeld een loading icon te spinnen.
Alternatieven: Glyphicons, Fontello

5.2 Vue.js

Vue is een JavaScript framework dat staat voor flexibiliteit en snelheid. Het is simpel om te gebruiken dankzij zijn intuïtieve syntax. Het is een soort 'Angular Lite', met kenmerken van andere frameworks zoals React. Om Vue klein te houden, bevat het enkel de kern functionaliteit. Functionaliteiten die niet overal nodig zijn worden in aparte componenten gehouden. Zo gebruikt de applicatie ook de `vue-router` package, om routing te voorzien, en de `vue-resources` package, om makkelijk HTTP requests te kunnen sturen.

Vue maakt gebruik van de `Vue` class om een viewmodel te maken. Aan dit viewmodel wordt een view gebonden. In deze view kunnen dan Vue directives gebruikt worden. De applicatie is opgebouwd uit zogenaamde 'components'. Dit zijn Vue classes die makkelijk door elkaar gebruikt kunnen worden.
De structuur van zo'n component ziet er als volgt uit:

```
var vm = new Vue({
    // Het template verwijst naar de view waar Vue
    // gebruikt kan worden.
    template: '#template-element',

    // Mixins zijn herbruikbare functies die gebundeld
    // worden in een object.
    mixins: [myMixin],

    // Code die eventueel uitgevoerd moet worden bij het
    // instantiëren van de component.
    ready: function() {
        console.log('Component loaded');
    },

    // Een component kan zelf andere components gebruiken.
    // Deze heten dan child components.
    components: {
        'childcomponent': childComponent
    },

    // Op de data property worden de models geïnstantieerd
    data: function() {
        return {
            model: 'Hello world'
        }
    },

    // Deze property kan gebruikt worden om models te
    // definiëren die op voorhand berekend moeten worden.
    computed: { },

    // De methods van de component.
    methods: { }

});
```

Elke route heeft zijn eigen component. Ook zijn er components voor bijvoorbeeld de datetimepicker. Bij elk van deze components hoort ook een view. Deze worden opgeslagen in de `resources/view/vue` folder.

Vue maakt ook gebruik van mixins. Een mixin is een bundeling van herbruikbare functies. Zo is er een contactsMixin die de verschillende HTTP requests bevat.

Om data te delen over de verschillende components, wordt er gebruikt gemaakt van stores. Dit zijn simpele JavaScript objecten die de data bevat, samen met getters en setters.

De initialisatie van de applicatie gebeurt in `app.vue.js`. Dit zorgt voor volgende folderstructuur:

```
vue/
  |-- components/
      `-- ...
  |-- mixins/
      `-- ...
  |-- stores/
      `-- ...
  |-- app.vue.js
  `-- homepage.vue.js
```

5.3 Routing

Routing zit niet in het standaard Vue.js pakket. Hiervoor moet er gebruikt gemaakt worden van een extra plugin. Er zijn er verschillende beschikbaar, maar de applicatie maakt gebruik van `vue-router`. Dit omdat deze officieel ondersteund wordt.

Het gebruik van de router zorgt ervoor dat de SPA zich gedraagt als een gewone webapplicatie. Zo worden de routes ook opgeslagen in de browsergeschiedenis, om de gebruiksvriendelijkheid te verhogen.

De routing wordt geïnstantieerd in `app.vue.js`. Aan elke route dient er een component gekoppeld te worden. Deze component voorziet dan de view en de logica voor de route. Ook krijgt elke route een `auth` property. Deze wordt gebruikt om bepaalde views enkel bereikbaar te maken voor een geauthenticeerde gebruiker.

Om de router te kunnen gebruiker dient er eerst een basis Vue instantie gemaakt te worden. Deze zal dan dienen als parent voor alle routes.

```
var App = Vue.extend({..})
var router = new VueRouter();

router.map({
    '/': {
        component: Home,
        auth: true
    },
    '/contacts': {
        component: Contacts,
        auth: true
    },
    '/account': {
        component: Account,
        auth: true
    },
    '/thankyou': {
        component: Thankyou,
        auth: true
    },
    '/login': {
        component: Login,
        auth: false
    },
    '/register': {
        component: Register,
        auth: false
    },
    '/logout': {
        component: Logout,
        auth: false
    },
    '*': {
        component: Home
    }
});
```

Alle routes in `router.map` staan voor een pagina in de applicatie. De uitzonderingen zijn `'/logout'` en `'*'`. `'/logout'` wordt gebruikt om te gebruiker lokaal uit te loggen en redirect vervolgens terug naar de homepage. `'*'` is een 'jokerroute'. Het zal ervoor zorgen dat een request naar een onbestaande pagina redirect naar de `Home` component.

5.4 Mixins

Omdat components vaak dezelfde functies gebruiken, worden deze in een apart object gebundeld om herhalende code te vermijden. Dit soort objecten worden door Vue 'mixins' genoemd. Om van zo'n mixin gebruik te kunnen maken, moet deze gedeclareerd worden op de component.

```
var myComponent = Vue.extend({

    mixins: [myMixin, otherMixin],

    ...

});
```

De applicatie maakt gebruik van 4 mixins. De `remindersMixin` en de `contactsMixin` bundelen de http requests voor deze objecten. De `authMixin` bevat alle functies in verband met authenticatie. De `validatorMixin` zorgt ten slotte voor alle validatiefuncties.

```
|-- mixins/
    |-- authMixin.js
    |-- contactsMixin.js
    |-- remindersMixin.js
    `-- validatorMixin.js
```

RemindersMixin
Alle functies die HTTP requests uitvoeren in verband met de reminders, worden in deze mixin geplaatst. Het gaat dan over de gewoonlijke CRUD operaties.

```
var remindersMixin = {
    methods: {
        getUpcomingReminders: function() {..},
        submitReminder: function(..) {..},
        cancelReminder: function(..) {..},
        submitQuickReminder: function(..) {..}
    }
}
```

Alle methods lijken erg op elkaar. Voor het request wordt verstuurd, worden er enkele variabelen geïnitialiseerd om bijvoorbeeld een laadicoontje te kunnen tonen. In het request worden dan een success- en errorfunctie gedefiniëerd. Ten slotte wordt er een `finally` functie gebruikt om de variabelen te resetten. Als voorbeeld de `getUpcomingReminders`.

```
getUpcomingReminders() {
    this.isLoading.getUpcomingReminders = true;
    this.error = false;

    this.$http.get('api/reminders/upcoming').then(function(response) {
        //Success
        if(response.status == 200) {
            // Binds the response object to the data object
            remindersStore.setUpcomingReminders(response.data);
        }
    }, function(error) {
        //Error
        if(error.status == 401) {
            this.$dispatch('not-logged-in');
        }
        this.error = true;
    }).finally(function() {
        this.isLoading.getUpcomingReminders = false;
    });
}
```

Als de applicatie een `401` statuscode terugstuurt, betekent dit dat de gebruiker niet meer is ingelogd. Wanneer dit gebeurt, wordt er een event afgevuurd door middel van `$dispatch`. Dit event bubblet naar de parent component. Hier wordt dit dan afgehandeld.

ContactsMixin

Deze mixin volgt hetzelfde principe als de remindersMixin. Het bevat methods om een contact te updaten, deleten en inserten. De requests zijn op dezelfde manier opgebouwd als eerder besproken.

ValidatorMixin

Alle input wordt clientside gevalideerd voor er een request versturd wordt. Deze validatie speelt zich volledig af in de validatorMixin.

```
var validatorMixin = {
    methods: {
        _maxlength: function(..) {..},
        _minlength: function(..) {..},
        _required: function(..) {..},
        _numeric: function(..) {..},

        _validate: function(..) {..},

        validateContact: function(..) {..},
        validateReminder: function(..) {..},
        validateQuickReminder: function(..) {..},
        validateLogin: function(..) {..},
        validateSignup: function(..) {..},
    }
};
```

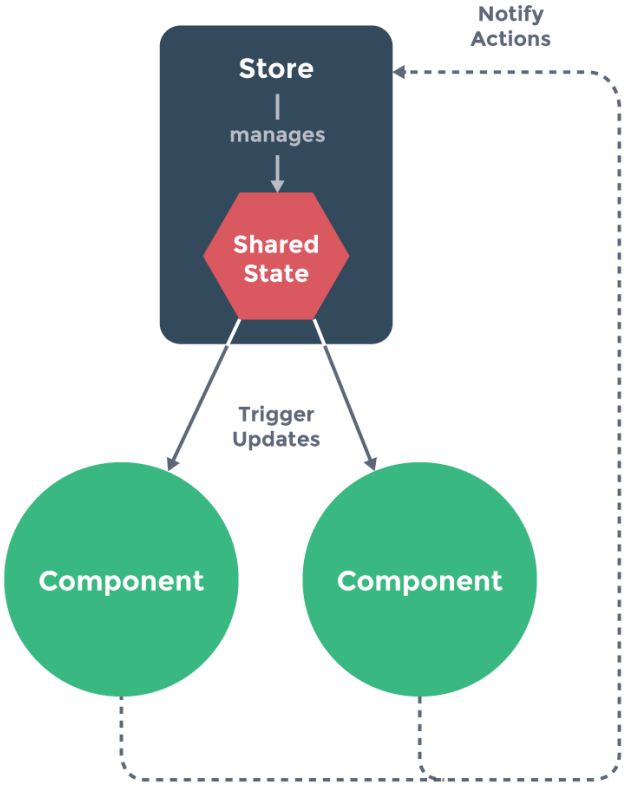
Voor elk formulier dat gevalideerd moet worden, wordt er een aparte method aangemaakt. Vervolgens kan elke input gevalideerd worden aan de hand van een syntax die op Laravel lijkt. De `_validate` method parset de validatiestring en past dan de bijhorende validatiemethod toe.

```
this._validate(contact.number, 'required|max:20|min:6');
```

AuthMixin
Deze mixin is verantwoordelijk voor alle authenticatiemethods. De login en registratie gebeuren via HTTP requests. De logout verwijdert alle data in de `localStorage`. Ook wordt deze mixin gebruikt om details over de gebruiker op te halen.

5.5 Stores

Om data te kunnen delen tussen de verschillende components wordt er gebruik gemaakt van stores. Dit is een vorm van state management. Een store heeft een shared state. Deze bevat de data die de components kunnen gebruiken. Naast deze shared state bevat de store een aantal methods om de data te manipuleren. Hierdoor wordt de state voor alle components ook geüpdate. Dit principe wordt ook gebruikt in React (Flux). Volgende afbeelding, overgenomen van vuejs.org, vat het principe samen.



De applicatie maakt gebruik voor een store voor de contactpersonen, de reminders en de gebruikersgegevens. Een extra voordeel van het gebruik van stores, is dat de data niet telkens opnieuw opgehaald dient te worden van de server.

```
|-- stores/
    |-- authStore.js
    |-- contactsStore.js
    `-- remindersStore.js
```

ContactsStore

Zoals eerder vermeld bestaat een store enerzijds uit de gedeelde state, en anderzijds uit een aantal methods om deze state te manipuleren. Zoals de naam al doet vermoeden, bestaat de state van de contactsStore uit een array van contactpersonen.

```
var contactsStore = {
    state: {
        contacts: []
    },

    isLoaded: false,
    setLoadStatus: function(..) { .. },

    getIndexOf: function(..) { .. },
    setContacts: function(..) { .. },
    removeContact: function(contact) (..) { .. },
    addContact: function(..) { .. },
    updateContact: function(..) { .. },
};
```

De `isLoaded` property dient ervoor om snel na te gaan of de data al geladen is of niet. De andere methods dienen er dan voor om CRUD operaties uit te voeren op de shared state. Zo'n method ziet er dan als volgt uit:

```
removeContact: function(contact) {
    var index = this.getIndexOf(contact);
    // Check if the given contact exists in the shared state.
    if(index > -1) {
        this.state.contacts.splice(index, 1);
    }
}
```

RemindersStore

Deze store bevat alle reminders. De werking ervan is volledig analoog aan de `contactsStore`.

AuthStore

De authStore is een kleine store die enerzijds de gegevens over de gebruiker bijhoudt, en anderzijds de authenticatiestatus. Deze status wordt door de router gebruikt om te bepalen welke routes getoond mogen worden en welke niet.

```
var authStore = {
    state: {
        user: null,
        authenticated: false
    },

    setUser: function(user) {
        this.state.user = user;
    },

    incrementCredits: function (amount) {
        this.state.user.reminder_credits += amount;
    },

    setAuthenticationStatus: function(bool) {
        this.state.authenticated = bool;
    },

    getAuthenticationStatus: function() {
        return this.state.authenticated;
    }
}
```

5.6 Dashboard Home

De homepage van het dashboard is, net zoals elke andere route, een Vue component. Het is ook veruit de grootste component van de applicatie. Ze wordt dan ook best stapsgewijs overlopen. De volgende fragmenten code zijn telkens deel van de `Home` component.

Als eerste wordt aangeduid welke view de component gebruikt. Dit wordt gedaan met de `template` property.

```
template: '#dashboard-home',
```

Als volgende worden de gebruikte mixins aangeduid. Op de homepage kunnen nieuwe reminders gemaakt worden. Daarvoor zijn de `remindersMixin` en de `validatorMixin` nodig. Om een suggestiebox te kunnen geven voor de contactpersonen, is ook de `contactsMixin` nodig.

```
mixins: [contactsMixin, remindersMixin, validatorMixin],
```

Wanneer de component geladen is, wordt er nagekeken of de contactpersonen en reminders al geladen zijn. Dit gebeurt in de `ready` functie. Moesten ze nog niet geladen zijn, dan wordt er een request verstuurd met behulp van de bijhorende mixin.

```
ready: function() {
    // Check if the stores are loaded, otherwise sent a request.
    if(!contactsStore.isLoaded) {
        this.getContacts();
        contactsStore.setLoadStatus(true);
    }
    if(!remindersStore.isLoaded) {
        this.getUpcomingReminders();
        remindersStore.setLoadStatus(true);
    }
},
```

Vervolgens worden de gebruikte components aangegeven. Een reminder wordt weergegeven in een tabel met behulp van de `reminderRow` component. Het formulier om een nieuwe reminder aan te maken, maakt gebruikt van de `dateTimePicker` component.

```
components: {
    'reminder-row': reminderRow,
    'datetimepicker': dateTimePicker
},
```

Gebruikte models worden aangegeven in de `data` property. Hierbij zit onder andere het model voor een nieuwe reminder, alsook een heel aantal variabelen. Ook de gebruikte stores worden in de `data` property gezet.

Sommige data models dienen berekend te worden. Hiervoor dient de `computed` property. Voor deze pagina is er slechts één computed property, namelijk de `filteredContacts`. Deze wordt gebruikt om in de suggestiebox de contacten te tonen die overeenkomen met de query.

```
computed: {
    filteredContacts: function() {
        var self = this;
        function contains(a, b) {
            return a.toLowerCase().indexOf(b.toLowerCase()) != -1;
        }
        return this.sharedState.contacts.filter(function(contact) {
            return contains(contact.name, self.query) ||
                   contains(contact.number, self.query);
        });
    }
},
```

Aan de viewmodel worden ook een aantal methods gebonden. Deze kunnen dan in de view gebruikt worden.

```
methods: {
    // Het selecteren van een contact in de suggestiebox.
    // Hier wordt de geselecteerde contactpersoon aan de
    // viewmodel gebonden.
    selectContact: function(contact) { .. },

    // Het highlighten van een contactpersoon in de
    // suggestiebox, door middel van de pijltjestoetsen.
    highlightContact: function(input) { .. },

    // Valideren van de input aan de hand van de
    // validatorMixin.
    validate: function() { .. },

    // Hulpfunctie om de input te trimmen.
    trim: function() { .. },

    // Het submitten van een reminder, met behulp
    // van de remindersMixin.
    handleReminderSubmit: function() { .. },
}
```

De views worden opgeslagen in `resources/views/vue`. Dankzij Vue kan er databinding toegepast worden. Dit gebeurt rechtstreeks op de HTML elementen. Vue gebruikt de `v-` syntax. Events kunnen ook rechtstreeks op de elementen geplaatst worden. Dit gebeurt met de verkorte `@` syntax.

Als eerste is er het formulier om een nieuwe reminder aan te maken.

```
<form class="flat-form overflow" @submit.prevent="handleReminderSubmit">

    ...

    <input type="submit" value="Submit" v-if="user.reminder_credits>0">
    <input type="submit" value="No reminders!" disabled v-else>
</form>
```

In het formulier zijn alle inputs gebonden aan het bijhorende model. Dit gebeurt met de `v-model` tag. Het formulier wordt verzonden met behulp van de `handleReminderSubmit` method. Hier wordt eerst validatie uitgevoerd alvorens er een request verstuurd wordt.

De verzendknop wordt conditioneel gerenderd. Wanneer de gebruiker geen reminder credits meer heeft, kan het formulier niet verzonden worden. De knop wordt dan disabled, en krijgt een andere tekst. Dit gebeurt met `v-if` en `v-else`.

De meest uitgebreide input van het formulier is deze voor de ontvanger. omdat de gebruiker hiervoor ook een contact kan opgeven, krijgt hij een suggestiebox te zien wanneer hij een contact begint te typen.

```
<input  type="text" id="fi_search"
        v-model="query"
        @focus="showSuggestions = true"
        @blur="showSuggestions = false"
        @keyup.down.prevent="highlightContact('down')"
        @keyup.up.prevent="highlightContact('up')"
        @keyup.enter.prevent="selectContact(highlightedContact)"
        @input="validate"
        autocomplete="off"
        v-if="!isContactSelected"
>
```

Als eerste wordt de databinding aangegeven door `v-model`. Vervolgens worden er een hele reeks events aan de input gekoppeld. `@focus` en `@blur` zorgen ervoor dat de suggestiebox zichtbaar wordt of niet. De `@keyup` events maken het mogelijk om de pijltjestoetsen te gebruik om een contactpersoon uit de lijst te selecteren. het HTML-attribuut `autocomplete` wordt op `false` gezet. Dit om te voorkomen dat de standaard suggestiebox van de browser tevoorschijn komt.

Bij de input hoort dan ook nog de suggestiebox zelf. Dit is een lijst van contactpersonen die absoluut gepositioneerd wordt onder het input veld. Via `v-for` wordt de gefilterde lijst doorlopen. Dit is vergelijkbaar met `ng-repeat` in Angular.

```
<div class="suggestionbox-wrapper"
     v-show="query.length > 1 && showSuggestions">
    <div class="suggestionbox">
        <p  v-for="(index, contact) in filteredContacts"
            :class="{ 'active': contact.id == highlightedContact.id }"
            @mouseenter="highlightContact(index)"
            @mousedown="selectContact(contact)"
        >
            @{{contact.name}} (@{{contact.number}})
        </p>
    </div>
</div>
```

De suggestiebox wordt pas zichtbaar wanneer de query minstens 2 letters bevat. Dit om een nodeloos lange lijst te voorkomen. Elke contactpersoon krijgt zijn eigen `<p>` tag. Deze wordt herhaald met `v-for`. Met behulp van de `:class` syntax kan er conditioneel een CSS class gebonden worden aan een element. in dit geval krijgt de geselecteerde contactpersoon een andere achtergrondkleur.

De naam en het nummer van een contactpersoon worden getoond door middel van dubbele accolades. Omdat de Razer syntax van Laravel ook deze van dubbele accolades, moeten deze voorafgegaan worden door een `@`. Op deze manier worden ze genegeerd door de parser van Laravel.

De gebruiker krijgt de opkomende reminders te zien in een tabel. Omdat we hiervoor een aparte component gebruiken, bespaart dit heel wat code op deze pagina.

```
<tbody v-if="remindersState.upcomingReminders.length != 0">
    <tr v-for="reminder in remindersState.upcomingReminders
                | orderBy 'send_datetime'"
        is="reminder-row"
        :repeats="repeats"
        :reminder.sync="reminder">
    </tr>
</tbody>
<tbody v-else>
    <tr>
        <td colspan="4" v-if="isLoading.getUpcomingReminders">
            <i class="fa fa-spinner fa-pulse">i> Loading...
        </td>
        <td colspan="4" v-else>
            No reminders yet!
        </td>
    </tr>
</tbody>
```

Doorgaans wordt een component gebruikt zoals een HTML tag. Om bijvoorbeeld de `dateTimePicker` component te gebruiken, kan deze op de volgende manier gedeclareerd worden:

```
<datetimepicker></datetimepicker>
```

Omdat de browser standaard alle elementen uit een `<table>` haalt die er niet horen, kan de component niet op deze manier gebruikt worden. In de plaats voorziet Vue de `is` syntax.

De reminders worden weer herhaald door `v-for`. `:reminder.sync` zorgt ervoor dat de component up-to-date blijft met de model, en andersom.

Zijn er geen reminders om te tonen, dan krijgt de gebruiker "No reminders yet!" te zien. Dit gebeurt weer via `v-if`. Op dezelfde manier wordt er ook een laadicoontje getoond als de contacten nog opgehaald moeten worden.

5.7 Contacts

Op de contacts pagina kan de gebruiker zijn contactpersonen beheren. Er is een klein formulier om een nieuwe contactpersoon aan te maken, alsook een tabel om bestaande contacten te beheren.

De opbouw van de component komt overeen met die van de `Home` component. Ook de view is vergelijkbaar. De tabel maakt weer gebruik van een component om contacten in een table row weer te geven. Deze is iets uitgebreider dan die van de reminders omdat de contactpersonen ook geüpdate kunnen worden.

5.8 Account

De accountpagina is de kleinste van alle components. De enige data die gebruikt wordt, is die van de `authStore`. Hierdoor kan er informatie over de gebruiker weergegeven worden.

```
var Account = Vue.extend({
    template: '#dashboard-account',

    data: function() {
        return {
            state: authStore.state
        };
    }
});
```

5.9 Authenticatie

De `authMixin` vormt, in combinatie met de `authStore`, de kern van de authenticatie.

```
var authMixin = {
    methods: {
        authSignIn: function(..) { .. },
        authRegister: function(..) { .. },
        getUserInfo: function() { .. },
        authLogOut: function(..) { .. },
        setToken: function(..) { .. }
    }
};
```

`authSignin()` stuurt een request naar de server met de logingegevens van de gebruiker. Als de gegevens kloppen wordt in de `authStore` de authenticatiestatus op `true` gezet. Deze boolean wordt onder andere door de router gebruikt. Het antwoord van de server bevat een JSON Web Token. Deze wordt via de `setToken` method aan de Vue headers toegevoegd, en wordt vanaf dan met elk volgend request meegestuurd. Ook wordt het token opgeslagen in de `localStorage`.

De `authRegister()` method stuurt een request om een gebruiker te registreren. Lukt deze registratie, dan worden dezelfde handelingen uitgevoerd als bij `authSignin()`.

`authLogout()` dient ervoor om een gebruiker uit te loggen. De authenticatiestatus in de `authStore` wordt terug op `false` gezet, het token wordt uit de Vue headers verwijderd, en de `localStorage` wordt leeggemaakt.

De gebruiker kan zich inloggen op de `'/login'` pagina. Deze pagina hangt af van de `Login` component. Buiten de gewoonlijke properties heeft deze component nog een extra `route` property.

```
route: {
    // Checks if the user is already logged in.
    // If so, redirects to /home.
    activate: function(transition) {
        if(authStore.getAuthenticationStatus()) {
            router.go('/home')
        } else {
            transition.next();
        }
    }
}
```

Deze property wordt door de `vue-router` gebruikt. Wanneer de gebruiker al is ingelogd, zorgt dit ervoor dat de pagina geredirect wordt. De `Register` component gebruikt dit ook.

Als de gebruiker het browservenster sluit, is alle data in de `authStore` verloren, dus ook de authenticatiestatus. Om de gebruiksvriendelijkheid te verhogen moet er een manier gevonden worden om dit toch te onthouden.

Zoals eerder gezegd wordt er gebruik gemaakt van de `localStorage`. Wanneer de login succesvol verloopt, wordt hierin de JWT en het aanmaaktijdstip opgeslagen. Wanneer de applicatie dan opstart, wordt eerst de `localStorage` nagekeken op de aanwezigheid hiervan. Dit gebeurt in de root `App` component.

Als er data wordt gevonden, wordt het aanmaaktijdstip nagekeken. Als dit een tijdstip korter dan de token time to live is (60m) , dan wordt aangenomen dat de gebruiker ingelogd is. Moest het token toch niet meer geldig zijn, of niet authentiek, dan wordt dit opgevangen bij het eerste request dat verstuurd wordt. Dit zal dan een 401 statuscode terugsturen.

Is de `localStorage` leeg, of het aanmaaktijdstip te lang geleden, dan start de applicatie normaal op. De gebruiker zal zich dan eerst moeten authenticeren.

5.10 Datetime picker

Om een reminder te maken wordt er gebruikt gemaakt van een datetimepicker om de verzenddatum in te geven. Dit wordt geregeld door een aparte component. Als op de parent component de datetimepicker als volgt gedefiniëerd wordt:

```
components: {
    'datetimepicker': dateTimePicker
},
```

Dan kan een datetimpicker toegevoegd worden in een formulier op de volgende manier:

```
<datetimepicker>datetimepicker>
```

De structuur van de datetimepicker ziet er als volgt uit:

```
var dateTimePicker = Vue.extend({
    template: '#datetimepicker-template',

    props: ['name', 'result'],

    ready: function() { .. },

    data: function() { .. },

    computed: { .. },

    methods: {
        parseFromString: function(..) { .. },
        generateMonth: function(..) { .. },
        nextMonth: function(..) { .. },
        previousMonth: function(..) { .. },
        selectDay: function(..) { .. },
        format: function(..) { .. },
        increment(..) { .. },
    }
});
```

De `props` property wordt gebruikt om variabelen te kunnen doorgeven vanuit de parent. In dit geval wordt `name` gebruikt om de input een naam te geven, en zal `result` gebruikt worden voor de databinding.

In de `ready` functie wordt de datetimepicker geïnitialiseerd. De huidige maand wordt berekend en weergegeven, en de datum en tijd worden op de huidige datum en tijd gezet.

In de `data` property worden weer de gebruikte models gedefiniëerd. Het gaat bijvoorbeeld om de labels van de dagen en maanden, de geselecteerde datum, enz.

De `computed` property is speciaal voor deze component. Omdat de gebruiker enerzijds de datetimepicker kan gebruiken om een datum en tijd te kiezen, maar ook gewoon een tijdstip kan invullen, moet er een getter en een setter gebruikt worden.

```
result: {
    get: function() {
        return this.selectedValues.year + '-'
                + this.format(this.selectedValues.month) + '-'
                + this.format(this.selectedValues.date) + ' '
                + this.format(this.selectedValues.hour) + ':'
                + this.format(this.selectedValues.minute);
    }, set: function(newValue) {
        this.parseFromString(newValue);
    }
}
```

In de `get` property wordt het resultaat weergegeven aan de hand van de `selectedValues`. Dit zijn de waardes die de gebruiker selecteerd in de datetimepicker. De `set` gebruikt de `parseFromString()` method om handmatig getypte input te verwerken. `parseFromString()` zal ook de `selectedValues` updaten, waardoor de getter de juiste waardes teruggeeft.

de `generateMonth()` method wordt, zoals de naam al doet vermoeden, gebruikt om een maand te genereren. Aan de hand van een gegeven jaar en maand, zal het een array van dagen teruggeven. Deze wordt gebruikt in de view om de tabel op te vullen. Er wordt ook rekening gehouden met schrikkeljaren.

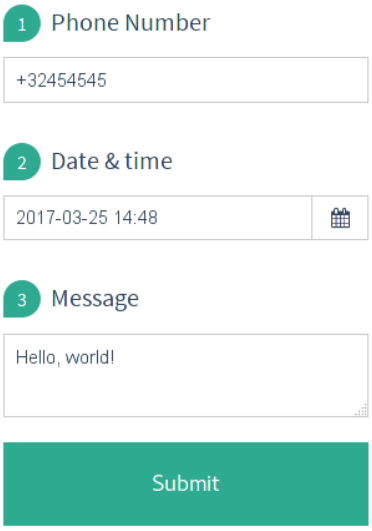
`nextMonth()` en `previousMonth()` zijn de functies die opgeroepen worden om de volgende of vorige maand te berekenen.

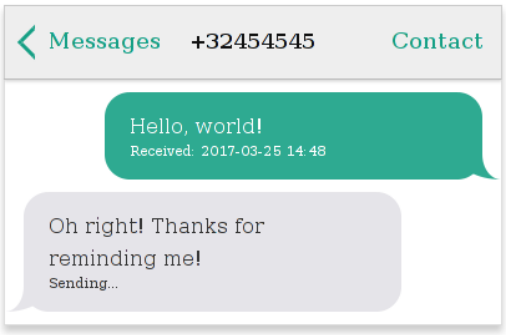
`selectDay()` wordt opgeroepen wanneer de gebruiker een dag selecteert. `increment()` wordt gebruikt om de tijd te kunnen selecteren. Ten slotte is er nog `format()`. Dit is een functie die de datum formateert, zodat deze van het formaat `Y-m-d H:i:s` is.

5.11 Homepage

De homepage van de website wordt ook ondersteunt door Vue. Dit omdat hier ook een formulier verzonden kan worden. Het voorziet onder andere validatie, en zorgt ervoor dat de datetimepicker gebruikt kan worden.

Het telefoonscherm rechts van het formulier wordt ook geüpdate wanneer de gebruiker iets invult. Dit gebeurt door de databinding.


1 Phone Number
+32454545
2 Date & time
2017-03-25 14:48
3 Message
Hello, world!
Submit


Messages
+32454545
Contact
Hello, world!
Received: 2017-03-25 14:48
Oh right! Thanks for reminding me!
Sending...

6. Werkwijze

Alvorens aan de ontwikkeling te beginnen, werd er eerst angedacht over de aanpak. Het project werd opgedeeld in kleine deelproblemen. Deze deelproblemen werde vervolgens gesorteerd via het MoSCoW principe.

Taken die essentiëel waren, kregen de hoogste prioriteit. Dit waren de zogenaamde **Must Have's**. Het ging dan bijvoorbeeld om het versturen van een SMS, of het aanmaken van een reminder. Deze functionaliteiten samen vormden het MVP (Minimum Viable Product).

Vervolgens waren er de **Should Have's**. Deze taken kregen iets minder prioriteit. Het aanmaken van een account was bijvoorbeeld zo'n Should have. Deze taken waren niet essentiëel, maar wel belangrijk voor een afgewerkt product.

De **Could Have's** en **Won't Have's** werden onder dezelfde categorie gebundeld. Deze functionaliteiten waren niet belangrijk voor het eindresultaat, maak konden wel geïmplementeerd worden als er nog tijd over was. Het contactformulier is een voorbeeld van deze categorie. Dit is uiteindelijk wel in de applicatie opgenomen. Een ander voorbeeld was het versturen van mails. Dit is uiteindelijk weggevallen.

De taken werden vervolgens op een *kanban* bord geplaats. Hier werden de taken opgedeeld in 'To Do', 'In Progress' en 'Done'. De taken in de 'To Do'-lijst werden dan gesorteerd op prioriteit.

Voor het project werd ook gebruik gemaakt van source control, meer specifiek *git*. Dit maakte het mogelijk om een rollback uit te voeren wanneer er iets misliep. Ook vergemakkelijkte het de ontwikkeling op meerdere computers, alsook de deployment. Het project is te vinden op https://github.com/mark-gerarts/ReminderApp.

7. Conclusie