Reduceren van technical debt binnen een online travel platform

Scriptie
9 april 2019
Naam: K de Hoop
Studentnummer: 405643
Opleiding: Kunst & Techniek (CMGT)
Onderwijsinstelling: Saxion Hogeschool Enschede
Begeleider: Hester van der Ent
Tweede lezer: ...

Voorwoord

Voor u ligt de scriptie ‘Reduceren van technical debt binnen een online travel platform’. Het onderzoek voor deze scriptie is uitgevoerd bij het bedrijf Hi,hi Guide.

Deze scriptie is geschreven in het kader van mijn afstuderen aan de opleiding Kunst & Techniek (nu genaamd Creative Media & Game Technologies) aan de Saxion Hogeschool Enschede. Ik ben vanaf december 2018 tot april 2019 bezig geweest met het onderzoek en het schrijven van de scriptie.

Vorig jaar zijn Steven van Eck en ik samen het bedrijf Developmunt gestart, waarmee wij web- en mobiele applicaties ontwikkelen. Voor ons was het afstuderen een uitgelezen kans om een nieuwe manier van het bouwen en koppelen van apps te onderzoeken. Het was natuurlijk ideaal dat wij Hi,hi Guide verder konden helpen met dit onderzoek en het ontwikkelen van de app. Wij houden van de internationale oriëntatie van dit bedrijf, dus we staan helemaal achter hun concept.

Deze scriptie is allereerst bedoeld voor Hi,hi Guide. Ik hoop dat ze door mijn onderzoek een beter inzicht hebben gekregen in de technische keuzes die er gemaakt moeten worden voor het verbeteren van de codebase en voor het ontwikkelen de app.

Dit onderzoek kan uiteraard ook gebruikt worden door andere developers en bedrijven die eindeloos vernieuwingen aanbrengen aan een verouderde opzet. Hierdoor verwatert de oorspronkelijke functie van de code. Feitelijk wordt een basissysteem steeds meer opgelapt met vernieuwingen. Daardoor wordt het steeds complexer om dit systeem te onderhouden. Als het niveau van de oorspronkelijke structuur niet meegroeit met de vernieuwingen, loopt de *technical debt* steeds verder op.

Ik wil graag Hubert Nijmeijer en Tom Groeneveld bedanken voor de leuke en leerzame tijd bij Hi,hi Guide. Ook wil ik Steven van Eck bedanken voor de plezierige samenwerking. Verder wil ik ook de mensen bij de CeeSpot bedanken, omdat zij altijd graag even wilden helpen of meekijken bij een probleem. Tot slot wil ik Hester van der Ent en Kasper Kamperman bedanken voor de begeleiding.

Kaj de Hoop
Enschede, 31 maart 2019

Samenvatting

De travel startup Hi,hi Guide wil graag een mobiele app ontwikkelen voor hun lokale gidsen. Voordat het mogelijk wordt om een app te ontwikkelen, moet de technical debt in het huidige platform verminderd worden.

Het doel van dit onderzoek is om erachter te komen welke stappen moeten worden ondernomen om de technical debt binnen het platform van Hi,hi Guide te verminderen, zodat er in alle vrijheid zo efficiënt mogelijk verder gebouwd kan worden aan het platform. De volgende hoofdvraag wordt hierbij gesteld:

Welke stappen moeten worden ondernomen om de technical debt bij Hi,hi Guide te verminderen?

Om de hoofdvraag te beantwoorden is naar vijf verschillende aspecten gekeken die bijdragen aan het verminderen van de technical debt. Bij elk aspect is een theoretisch onderzoek gedaan, en vervolgens is deze theorie toegepast in de praktijk.

De belangrijkste conclusies van het onderzoek zijn:

1. Er moet gebruikt worden gemaakt van migrations en seeders bij aanpassingen aan de database.
2. Er moet gezorgd worden dat de Laravel code volgens de MVC architectuur wordt opgebouwd.
3. Er moet een GraphQL API geïmplementeerd worden door gebruik te maken van Lighthouse. Vanuit de React Native app moet gebruik worden gemaakt van Apollo Client om met deze API te communiceren.
4. Er moet een server op een VPS gezet worden die code vanaf de git repository kan pullen en automatisch een aantal commands kan runnen om de boel klaar te zetten. Dit is meteen een goede opstap richting een volledige Continuous Integration workflow.
5. De API moet beveiligd worden door OAuth2 te implementeren met de API middleware van Laravel Passport.

Bij Hi,hi Guide zijn deze conclusies inmiddels in de praktijk gebracht om de technical debt significant te verminderen. Hierdoor is het mogelijk geworden om een vernieuwd platform te lanceren en een mobiele app te bouwen en koppelen aan het platform. Het vernieuwde platform staat live op www.hihiguide.com. De resultaten van dit onderzoek kunnen door andere bedrijven worden gebruikt die ook verzanden in technical debt.

De belangrijkste aanbevelingen die verder nog volgen uit dit onderzoek zijn:

1. Indeling van de database verbeteren
2. Tests automatiseren

Dit verslag begint met de praktijkanalyse. Uit een kort vooronderzoek volgt de probleemstelling. Een roadmap geeft een overzicht over het doorlopen ontwerpproces, en daarna worden de deelvragen opgesteld.

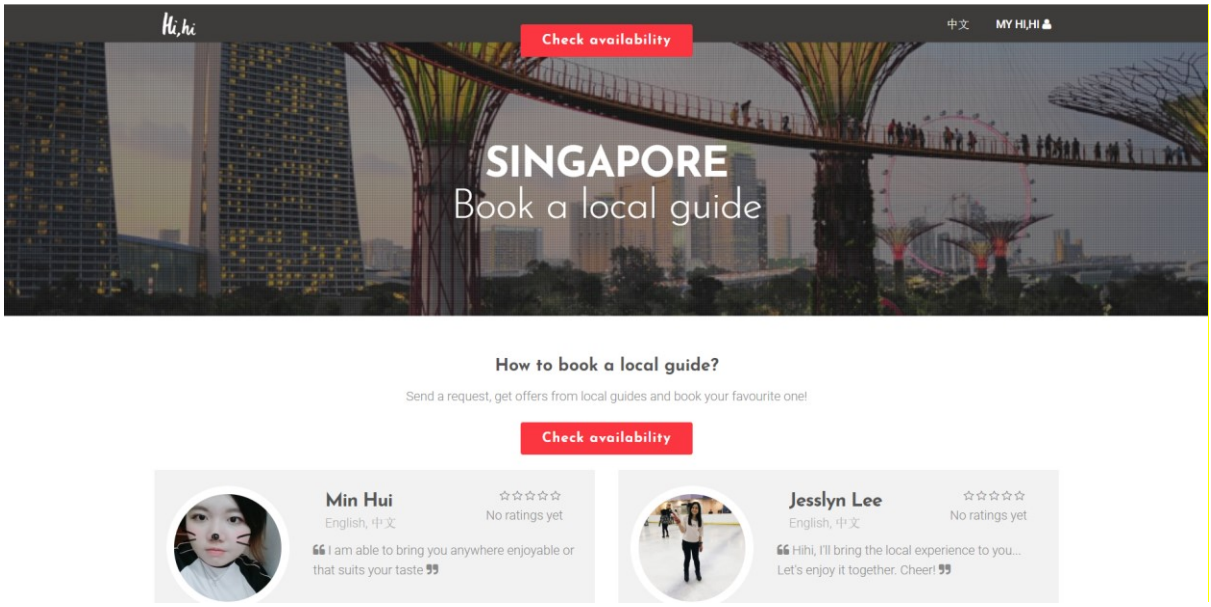
Inhoudsopgave

Inleiding

Praktijkanalyse

Hi,hi Guide van Hubert en Tom is in 2018 verkozen tot Emerce Travel Startup van het jaar. Het bedrijf is in 2017 gestart. Hi,hi Guide koppelt lokale inwoners van steden over de hele wereld aan toeristen die deze steden bezoeken. Ze doen dit door middel van een online platform waarop locals zich als gids kunnen aanmelden en waarop toeristen een verzoek in kunnen sturen om een gids te boeken. Als er een verzoek wordt ingestuurd, krijgen gidsen in de betreffende stad een mail en kunnen ze dit verzoek accepteren. Vervolgens kan een toerist een keus maken uit de gidsen die het verzoek geaccepteerd hebben en deze gids boeken.

De focus van Hi,hi Guide ligt op het persoonlijke aspect van de tour. De klant ervaart een unieke belevenis. Het zijn geen vastgelegde tours en vaste routes, maar juist echte locals die met je op pad gaan. Hi,hi Guide koppelt de persoonlijke interesse van de local met de toerist. Voor de toerist is zelf te bepalen wat hij graag wil zien en doen, de gids denkt hierin mee. Zo ontstaat er een tour die helemaal persoonlijk is toegespitst op de reiziger. Dit persoonlijke aspect is onderscheidend ten opzichte van andere tour-aanbieders, want daar wordt gewoon een standaard tour afgenomen. Ter illustratie is in figuur 1 een screenshot van het huidige platform van Hi,hi Guide te zien.



Figuur 1. Screenshot van het huidige online platform van Hi,hi Guide

Aanleiding

Hi, hi Guide blijkt een avontuurlijke markt te hebben aangeboord. Om verder te groeien willen ze het aantal boekingen voor dit belevingstoeristme via het platform en de engagement met hun gidsen vergroten. Op dit moment wordt er onderling gecommuniceerd met emails. De volgende stap is een mobiele app voor de gidsen en toeristen, waar vanuit gebruikers vraag naar is. Een mobiele app zorgt voor een directere verbinding tussen Hi,hi Guide en hun gidsen en klanten. Voordat deze mobiele app gebouwd kan worden, moet eerst het bestaande platfom op technisch gebied verbeterd worden.

Doelstelling

De technical debt binnen het platform van Hi,hi Guide moet in kaart worden gebracht en worden verminderd. De uitdaging is om innovaties in te brengen die de technical debt omlaag brengen. Zo wordt het makkelijker om bugs in het boekingsproces op te lossen, het platform sneller te maken, en het platform gereed te maken voor een koppeling met een mobiele app. Daarnaast kan er zo in een later stadium een publiekelijk toegankelijke API voor partners worden gerealiseerd. Het is mogelijk om de technical debt binnen een halfjaar al significant te verminderen. Voor het verbeteren van de codebase is geen extra financiële investering nodig, alleen tijd.

Relevantie

Het verminderen van de technical debt heeft de volgende gevolgen voor Hi,hi Guide:

Kortere iteraties in het ontwikkelproces

Er kunnen sneller nieuwe features worden ontwikkeld die per direct live kunnen worden getest. Zo kunnen veel sneller iteraties in het ontwikkelproces worden gedaan en dit leidt sneller tot een beter product.

Lagere ontwikkelingskosten

Het wordt goedkoper om bugs op te lossen en om nieuwe features toe te voegen aan het platform, omdat er minder ontwikkelingstijd nodig is.

Koppeling met nieuwe mobile app te realiseren

Het platform wordt gereed gemaakt voor de koppeling van de nieuwe mobiele app. Er wordt begonnen met de gidsen. Met deze app reageren gidsen sneller op boekingen, waardoor toeristen eerder een keus voor hun gids kunnen maken. Deze koppeling is ook geschikt om de hierop volgende app voor toeristen te ontwikkelen.

Afbakening

Dit onderzoek beperkt zich eerst alleen tot de ontwikkeling van Hi,hi Guide als bedrijf. Ze hebben een interessante markt te pakken. Als zij vrijuit willen kunnen uitbouwen met hun marketing en sales, moet eerst hun technical debt worden opgelost. Met de innovaties worden aangebracht wordt ervoor gezorgd dat hun technical debt vermindert en ook in de toekomst tot het uiterste beperkt blijft. Dat maakt het bedrijf ook interessant voor investeerders.

De focus van het onderzoek ligt op de backend van het platform. De frontend wordt gebruikt als leidraad voor het bouwen van de backend.

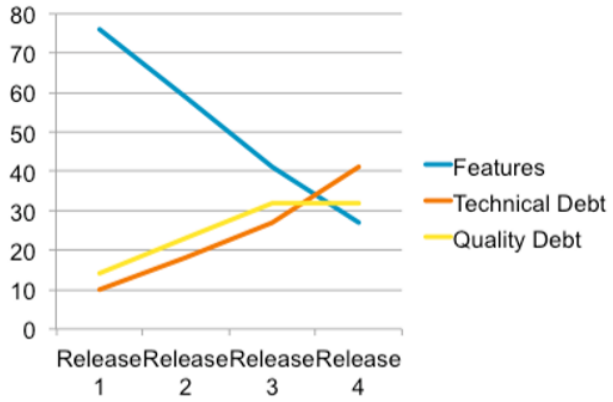
Vooronderzoek

Tijdens het moderniseren van software wordt er vaak niet goed gekeken naar de huidige technische staat van het product. Developers willen vaak al snel alles opnieuw bouwen en bedrijven willen juist geen tijd en geld besteden aan het technisch verbeteren van een applicatie. Technical debt is een term waardoor hierover op de juiste manier te discussieren valt.

Technical debt verwijst naar het verschijnsel dat er vaak tijdens softwareprojecten op de korte termijn voor oplossingen wordt gekozen die op de lange termijn extra kosten met zich meebrengen. Dit kan worden vergeleken met een financiele schuld: in sommige gevallen kan het nuttig zijn om code snel te schrijven en zo snelle stappen te maken, maar deze ‘schuld’ moet wel weer ingelost

worden. Als technical debt niet wordt aangepakt blijft de code maar complexer en rommeliger worden, waardoor het softwareproject steeds verder vastloopt (Bakker, 2013).

Het belangrijkste effect van technical debt is het vertragen van de ontwikkeling van software. Naar mate een project vordert moet er steeds meer tijd worden gestoken in het oplossen van technische problemen. Hierdoor is er minder tijd beschikbaar om te werken aan features die er voor de eindgebruiker echt toe doen (Jensen, 2018). In figuur 2 is dit geïllustreerd met een lijndiagram.



Figuur 2. Voorbeeld van groeiende technical debt bij 4 releases van een product (Power, 2013)

Vier belangrijke redenen voor het ontstaan van een technical debt zijn:

- Ontwerp van de software is vanaf het begin onvoldoende uitgedacht
- Te strakke tijdsplanning
- Geen gebruik gemaakt van vaste conventies en goede manieren van software design en ontwikkeling
- Gebruik van te oude technologie

Dit leidt tot een aantal veel voorkomende soorten technical debt, zoals slecht geformatteerde broncode, te weinig tests, niet-modulaire code, te complexe code (te lange route door code om één bepaalde operatie uit te voeren) en een gebrek aan documentatie (Osetskyi, 2018).

Probleemstelling

Het huidige platform van Hi,hi Guide heeft een grote technical debt. Deze technical debt manifesteert zich op de volgende manieren:

- Database is handmatig opgebouwd
- MVC niet volledig doorgevoerd
- Request lifecycle volgt geen vast patroon
- Er is geen gebruik gemaakt van relationships in het data model
- Live zetten van nieuwe code duurt enkele dagen
- Beveiliging is niet goed uitgewerkt

Vanwege deze problemen is de code lastig te onderhouden en gaat er steeds meer tijd zitten in het toevoegen van nieuwe features en het oplossen van bugs. Dit is een belangrijk probleem omdat Hi,hi Guide een startup is die groeit en het zich dus niet kan veroorloven om op technisch gebied steeds meer vast te lopen. Door deze technical debt aan te pakken wordt de code overzichtelijker en wordt

het goedkoper om bugs op te lossen en nieuwe features toe te voegen aan het platform. Hiermee worden de iteraties in het ontwerpproces dus korter en zitten er minder (beveiligings)fouten in het eindproduct. Dit moet zorgen voor een snellere groei en beter ontworpen en getest platform.

De technical debt kan worden aangepakt door gericht de volgende punten te verbeteren:

- Opbouwen van de database automatiseren
- De codebase duidelijk opsplitsen in Model, View en Controller lagen
- Request lifecycle een vast patroon laten volgen
- Relationships tussen de verschillende data modellen leggen, hier gebruik van maken in de bestaande code en ook in de nieuwe API
- Een testserver opzetten waar binnen enkele seconden updates naartoe gepusht kunnen worden vanuit de code repository.
- De beveiliging van data onder de loep nemen en op een centrale manier regelen

Roadmap ontwerpproces

Figuur 3. Roadmap ontwerpproces

In de roadmap in figuur 3 zijn in grote lijnen de stappen te zien die zijn ondernomen tijdens het ontwerpproces van de backend. Het ontwerpproces had vijf focusgebieden die voortkomen uit de verbeterpunten die genoemd zijn in de probleemstelling, namelijk database, business logic & data model, GraphQL API, server en beveiliging. Deze focusgebieden hebben geleid tot de deelvragen.

Onderzoeksvragen

Hoofdvraag

Welke stappen moeten worden ondernomen om de technical debt bij Hi,hi Guide te verminderen?

Deelvraag 1

Hoe kan een database automatisch worden opgebouwd volgens een vastgelegde structuur?

Deelvraag 2

Wat zorgt ervoor dat een Laravel backend goed in elkaar zit?

Deelvraag 3

Hoe kan de nieuwe React Native app aan de bestaande Laravel backend worden gekoppeld op een manier die toekomstbestendig en schaalbaar is?

Deelvraag 4

Hoe kan nieuwe code binnen enkele seconden live getest worden?

Deelvraag 5

Welke manieren zijn er om persoonlijke data te beschermen?

Leeswijzer

Hieronder volgen per deelvraag de methode en resultaten, met daaronder de manier waarop deze resultaten in de praktijk zijn toegepast. Per deelvraag volgt een conclusie. Deze conclusies vormen samen de conclusie op de hoofdvraag in het hoofdstuk Conclusies. Hierna worden er een aantal aanbevelingen gedaan en het verslag wordt afgesloten met de discussie.

1 Deelvraag 1

Hoe kan een database automatisch worden opgebouwd volgens een vastgelegde structuur?

1.1 Methodologie

Er is door middel van deskresearch onderzoek gedaan naar manieren om een database op te bouwen en aan te passen. Ook is er gezocht naar een manier om deze database vervolgens klaar te maken voor gebruik in de applicatie. Hierbij is gebruik gemaakt van blogs van developers (vooral via Medium) en developer fora, waarvan StackOverflow de meest gebruikte onder developers is (Feldman, 2017).

1.2 Resultaten

Wanneer er een nieuwe versie van een applicatie wordt gedeployed, werkt deze meteen en is hij niet afhankelijk van de vorige versie van de applicatie. Bij een database is dit niet het geval, omdat er in de database data op een bepaalde manier is opgeslagen. Als bijvoorbeeld de manier van opslaan wordt veranderd in een nieuwe release, zou de bestaande data niet meer op de juiste manier opgeslagen zijn, waardoor er van alles kapot gaat. Er zijn twee manier om dit probleem op te lossen.

State-based deployment door schema vergelijking

Er wordt ontwikkeld op een aparte database, naast de huidige database. Wanneer er gedeployed gaat worden, worden de nieuwe database en de oude database met elkaar vergeleken met een programma. Dit programma probeert dan de data uit de oude database in de nieuwe database te passen. Het is heel afhankelijk van het gebruikte programma of dit lukt en de kans is groot dat deze manier van werken bijwerkingen geeft die meer kapot maken dan repareren. Dit wordt ook problematisch wanneer een database regelmatig geupdate moet worden.

Migration-based development

Migrations zijn stappen die nodig zijn om een database op te bouwen zoals hij is. Elke keer als er een aanpassing aan de database wordt gedaan, wordt dit in een script gezet. Dit script is een migration. Zo wordt elke aanpassing volgens een vaste gestructureerde manier gedaan en kan een database altijd opnieuw op dezelfde manier worden opgebouwd. Dit heeft een aantal voordelen:

- Solid deployment proces, omdat dit op elk platform en binnen elke omgeving gelijk is
- State van de database kan precies worden bepaald door migrations te runnen tot het gewenste punt
- Documentatie over versie van database kan in migrations worden gedaan
- Moedigt aan om kleine minder riskante veranderingen aan de database te doen
- Werkt goed binnen continuous integration omdat de database gemakkelijk met tests kan worden gebouwd
- Er is overzicht in de volledige structuur van de database, er kunnen dus geen dingen aan de database gebeuren die niet expliciet geprogrammeerd zijn

Het grootste struikelblok bij migration-based development is dat er veel losse scripts ontstaan. Deze scripts moeten goed georganiseerd worden en het runnen moet automatisch gebeuren (Nitsche, 2017). Laravel heeft gelukkig zelf al ingebouwde functies om migrations te maken en runnen. Deze migrations maken gebruik van de schema builder tool en er zijn functies om migrations tot een bepaald punt te runnen of terug te draaien (Otwell, z.d.-d).

Tijdens het ontwikkelen is het belangrijk dat de database kan werken met wat initiële data en/of mock data (Phoenix, z.d.). Seeden is een geautomatiseerd proces dat tijdens de eerste setup van een applicatie wordt gedraaid. Data kan bestaan uit mock/dummy data (zoals nep users om mee te testen), of data die nodig is voor het functioneren van de applicatie, zoals een administratoraccount. Laravel heeft hier al ingebouwde functionaliteiten voor (Wikipedia, 2018). Om de database te seeden met data die echt lijkt, kan Faker gebruikt worden. Dit is een Laravel package die realistische mock data kan genereren (Korop, 2015).

1.3 Praktijk

De bestaande database van Hi,hi Guide had nog geen migrations. Om deze wel met migrations op te laten bouwen is de live database reverse engineered met MySQL Workbench. Deze functie analyseert de bestaande live database en heeft een EER (database) diagram gegenereerd. Dit diagram geeft de database schematisch weer. Vervolgens is een Python script gebruikt om dit diagram om te zetten in werkende Laravel migrations, zodat hier weer verder mee gewerkt kon worden. Voor de nieuwe features moesten aanpassingen worden gedaan aan de database. Nu de database gebouwd is volgens migrations, is dit vrij eenvoudig. Voor elke aanpassing is een nieuw migration script gemaakt die verder werkt vanaf de laatste migration. In deze scripts wordt gebruik gemaakt van de schema builder van Laravel. Dit zorgt voor vrij overzichtelijke scripts. Een voorbeeld van zo'n migration script is te zien in figuur 4. In de up() functie staat de aanpassing die gedaan wordt. De down() functie heeft de omgekeerde functionaliteit van de up() functie. Deze wordt namelijk gebruikt als er een migration moet worden teruggedraaid.

```php
<?php

use ...

class AddCommentColumnToGuideBookings extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('guide_bookings', function(Blueprint $table) {
            $table->string( column: 'comment', length: 255)->nullable();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::table('guide_bookings', function(Blueprint $table) {
            $table->dropColumn( columns: 'comment');
        });
    }
}
```

Figuur 4. Migration waarmee de comment kolom aan de guide_bookings tabel wordt toegevoegd

Om de initiële data en mock data te genereren zijn er seeders gemaakt. Deze zijn van elkaar gesplitst om te kunnen zorgen dat een nieuwe database wel met initiële data geseed kan worden, maar hier geen mock data in hoeft te worden geseed. Dit is nuttig wanneer er een database moet worden opgezet die wel functioneel is, maar verder leeg moet zijn. De initiële data bestaat uit data voor de tabellen countries, cities, languages, activities, questions, about_tour_items. Dit zijn de tabellen waar bepaalde vooraf vastgelegde data minimaal in moet staan voor een juiste werking van het platform. Bij het testen op de develop omgeving is het nuttig om ook mock gebruikersdata ter beschikking te hebben, zodat het lijkt alsof het platform gewoon werkt. Hiervoor is gebruik gemaakt van Faker om realistische gebruikersdata te genereren. De nieuwe migrations en seeders zijn getest

voordat ze live zijn gebruikt. Dit is gebeurd door een kopie te maken van de live website en database en hierop de migrations en seeders te runnen. Op deze manier zijn er nog een aantal foutjes verholpen, zonder dat de live site hier last van ondervond. Voor een voorbeeld van zo'n test, zie Bijlage 1 - Testen van migrations.

1.4 Conclusie

Er moet gebruik worden gemaakt van migrations bij aanpassingen aan de database. Migrations zorgen ervoor dat de structuur van een database vastgelegd kan worden in een script, zodat deze vervolgens zo vaak als nodig op precies dezelfde manier kan worden opgebouwd. Met seeders kan ervoor worden gezorgd dat een database gevuld wordt met de minimaal benodigde data om de applicatie te laten functioneren. Ook kan hiermee mock data worden toegevoegd, zodat er gemakkelijk met realistische data getest kan worden.

2 Deelvraag 2

Wat zorgt ervoor dat een Laravel backend goed in elkaar zit?

2.1 Methodologie

Er is door middel van deskresearch onderzoek gedaan naar wat ervoor zorgt dat Laravel code goed in elkaar zit. Laravel is gebaseerd op een MVC architectuur. Er is onderzocht hoe het platform van Hi,hi Guide beter kan voldoen aan de MVC architectuur. Hierbij is gebruik gemaakt van blogs van developers (vooral via Medium) en developer fora, waarvan StackOverflow de meest gebruikte onder developers is (Feldman, 2017).

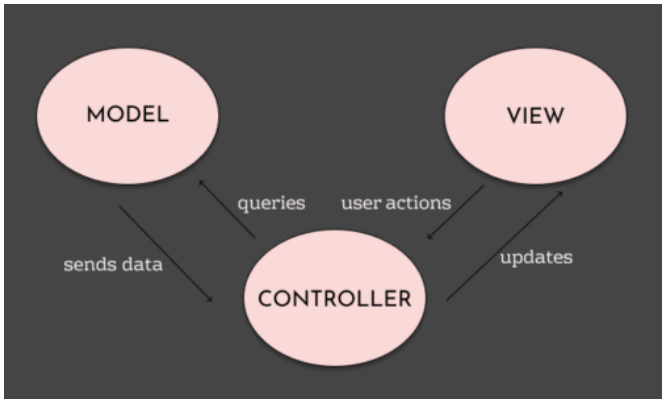
2.2 Resultaten

MVC staat voor Model – View – Controller. MVC gaat over de manier waarop de data flow door de applicatie werkt. De applicatie wordt opgedeeld in drie delen, het model, de view en de controller (figuur 5).

Model - Het model is een abstractie van de data en het management van de data binnen de applicatie. Het beheert het fundamentele gedrag van data. Het model kan de staat van de informatie veranderen en kan bijvoorbeeld gebruik maken van een database, maar ook van andere data-opslagsystemen.

View - De view zorgt voor de gebruikersinterface van de applicatie. Het rendert data van het model op een manier die geschikt is voor de interface.

Controller - De controller ontvangt input van de gebruiker en zorgt ervoor dat de juiste acties worden ondernomen en verstuurt deze naar het model en de view.



Figuur 5. Overzicht van de verbanden tussen model, view en controller

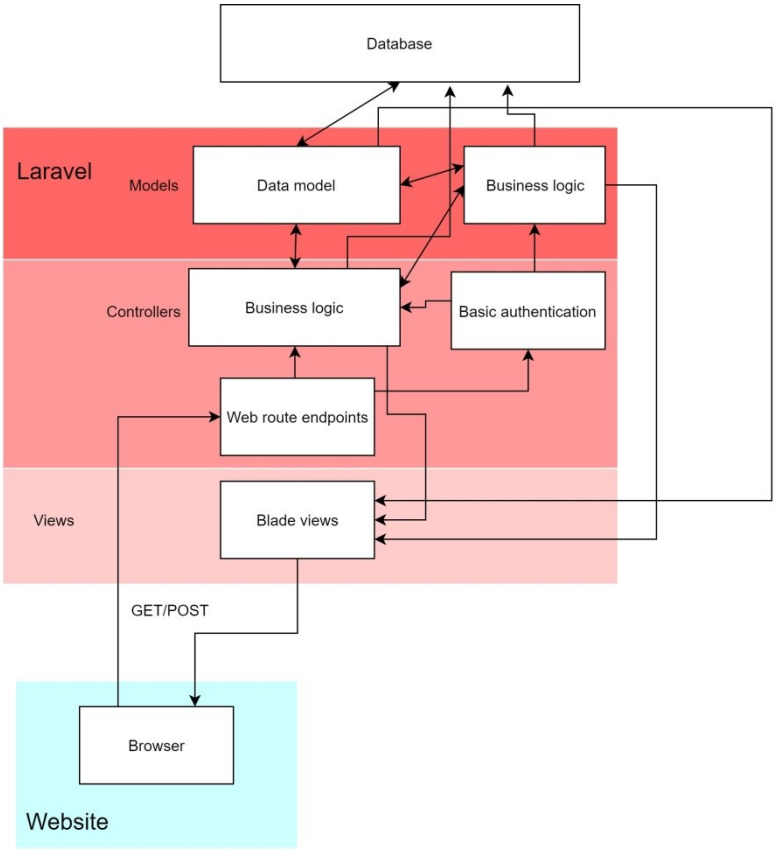
MVC geeft structuur aan complexe applicaties. Volgens Ighodaro (2018) zorgt dit voor de volgende voordelen ten opzichte van andere architecturen:

- Rollen van developers zijn duidelijk gesplitst
- Gemakkelijk vinden van bestanden door logische bestandsstructuur
- Weinig afhankelijkheden in de code door het opsplitsen van verantwoordelijkheden
- Volledige controle over de url's
- Gemakkelijk om code te schrijven volgens het SOLID principe code van Robert C. Martin (Oloruntoba, 2015)

2.3 Praktijk

Er zijn veel verbeteringen aan de codebase aangebracht om de bestaande Laravel code richting een goede MVC architectuur te bewegen. Om een beeld te geven van de oude situatie is deze hieronder beschreven. Vervolgens is de nieuwe situatie beschreven om de verschillen duidelijk te maken.

2.3.1 Oude situatie



Figuur 6. Abstract overzicht van alle onderdelen van de backend en hun onderlinge connecties in de oude situatie

In de oude situatie is de structuur van de Laravel code vrij chaotisch. Deze structuur is schematisch weergegeven in figuur 6. Het valt op dat de model- en controllerlagen niet duidelijk van elkaar gescheiden zijn; er zit business logic in de models en in de controllers. Dit zorgt voor willekeurige verbindingen tussen stukken code in de models en controllers. Ook is het onduidelijk waar code staat wanneer er een aanpassing moet worden gedaan aan een stuk business logic. Deze onduidelijke scheiding zorgt er verder voor dat niet alle requests vanaf de endpoints door de authenticatie hoeven te gaan, waardoor er gemakkelijk lekken in de beveiliging ontstaan. Een voorbeeld hiervan was het feit dat een willekeurige bezoeker van de website een boekingscode in de url in kon vullen en dan op de betaal-pagina van deze boeking kon komen. Hij kon dan vervolgens de boeking van een willekeurig andere gebruiker betalen omdat op dit deel code de authenticatie was vergeten. Dit is een vrij onschuldig lek (iemand zal niet snel de boeking van iemand anders betalen), maar het is niet ondenkbaar dat er meer van dit soort lekken in de code zaten, omdat de onderlinge connecties onduidelijk waren.

Wat verder opvalt, is dat de business logic soms directe bewerkingen op de database doet zonder gebruik te maken van het data model en onderlinge relaties hierin. Dit maakt de code onoverzichtelijk en onnodig complex.

In de views waren ook een aantal rare constructies te vinden omdat er geen gebruik werd gemaakt van de juiste volgorde in de request lifecycle (het pad van een request door de applicatie). Er werden bijvoorbeeld php tags (*<?php ?>*) gebruikt in views, om hier ter plekke nog business logic te doen. Een goed voorbeeld hiervan is de code die de blokjes met gidsen bij een stad rendert (figuur 7).

```
<div class="row">
    <div class="col-md-12">
        <?php
        $nr_guides = count($guides);
        for($i=0;$i<$nr_guides;$i+=2) {
        ?>
        <div class="row mt-10">
            @if (isset($guides[$i]))
                @include('pages.partial.cityguide', ['guide' => $guides[$i]])
            @endif

            @if (isset($guides[$i+1]))
                @include('pages.partial.cityguide', ['guide' => $guides[$i+1]])
            @endif
        </div>
        <?php
        }
        ?>
        <?php
        $nr_guides = count($oldGuides);
        for($i=0;$i<$nr_guides;$i+=2) {
        ?>
        <div class="row mt-10">
            @if (isset($oldGuides[$i]))
                @include('pages.partial.oldcityguide', ['guide' => $oldGuides[$i]])
            @endif

            @if (isset($oldGuides[$i+1]))
                @include('pages.partial.oldcityguide', ['guide' => $oldGuides[$i+1]])
            @endif
        </div>
        <?php
        }
        ?>
    </div>
</div>
```

Figuur 7. Code waarmee gidsblokken worden gerenderd in city page in oude situatie

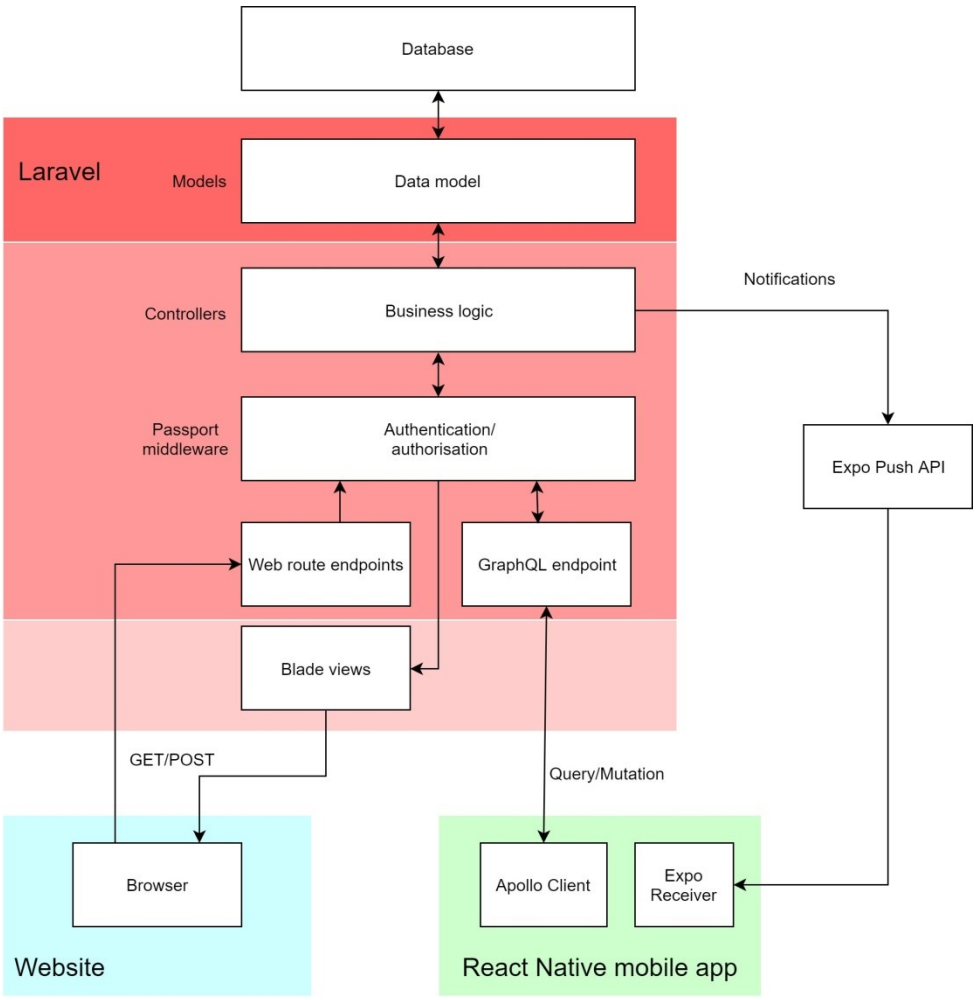
De code in figuur 7 kan worden samengevat in het kortere en duidelijkere stukje code in figuur 8. Door logica in de controller te houden en gebruik te maken van de juiste Blade directives kan een stuk code van 36 regels worden teruggebracht tot 13 regels. De kortere code is ook veel gemakkelijker te onderhouden, omdat er niet handmatig op elke regel twee blokken naast elkaar worden gezet. Stel in de toekomst moeten er niet twee maar drie blokken naast elkaar staan, dan moet bij de bovenstaande code een nieuw if-statement worden toegevoegd die weer een blok rendert, maar dan blok nummer $i+2. Ook moeten dan de beide for-loops worden aangepast om dit weer te geven. In de verbeterde code hoeft hiervoor helemaal niets te veranderen; deze code werkt in elke situatie hetzelfde. Deze code zit namelijk op conceptueel gebied goed in elkaar: het rendert meerdere blokken, en meer doet het niet. De oude code daarentegen is specifiek gebouwd voor deze situatie en gaat dus kapot zodra er iets verandert.

```
<div class="row">
    <div class="col-md-12">
        <div class="row">
            @foreach ($guides as $guide)
                    @include('layouts.master.content.citypage.content.guideblocks.content.guideprofileshort', ['guide' => $guide])
            @endforeach

            @foreach ($oldGuides as $oldGuide)
                @include('layouts.master.content.citypage.content.guideblocks.content.oldguideprofileshort', ['guide' => $oldGuide])
            @endforeach
        </div>
    </div>
</div>
```

Figuur 8. Code waarmee gidsblokken worden gerenderd in city page in nieuwe situatie

2.3.2 Nieuwe situatie



Figuur 9. Abstract overzicht van alle onderdelen van de backend en hun onderlinge connecties in de nieuwe situatie

In de nieuwe situatie is de code geordend in lagen voor models, views en controllers, volgens het MVC design pattern. Dit is schematisch weergegeven in figuur 9. Tussen de endpoints en de business logic zit een authenticatie/autorisatie layer, waardoor elke request gecheckt wordt en hier dus niet per ongeluk routes onbeveiligd kunnen worden gelaten. Een request vervolgt nu altijd de volgende route door de applicatie:

Endpoint (web of GraphQL) -> authenticatie/autorisatie -> controller -> model -> database -> model -> controller -> view/GraphQL API response

Er is een GraphQL endpoint (met bijbehorend schema) toegevoegd, zodat er een koppeling met de nieuwe React Native app voor de gidsen mogelijk is. Voor de push notificaties wordt gebruik gemaakt van de Expo Push API die wordt aangeroepen vanuit een notificatie controller op het moment dat er een notificatie naar een telefoon moet worden verstuurd.

2.4 Conclusie

In de Laravel code moet de MVC architectuur op de juiste manier worden toegepast. Dit zorgt ervoor dat de applicatie een goede structuur heeft. Een belangrijk onderdeel van het toepassen van MVC is het uitdrukkelijk gescheiden houden van model, view en controller. Wanneer deze goed gescheiden zijn, is het pad van een request door de applicatie een stuk korter en duidelijker en is er een kleinere kans dat er fouten ontstaan in de code. Dit zorgt voor een stabielere applicatie die sneller verder ontwikkeld kan worden.

3 Deelvraag 3

Hoe kan de nieuwe React Native app aan de bestaande Laravel backend worden gekoppeld op een manier die toekomstbestendig en schaalbaar is?

3.1 Methodologie

Om een React Native app aan een Laravel backend te koppelen is er een API nodig. Een API is een set regels die een programmeur moet volgen om te kunnen communiceren met een web server (Bettilyon, 2018). Eerst is er door middel van deskresearch gekeken wat voor soort API architecturen er zijn. Er is gebruik gemaakt van blogs van developers (vooral via Medium) en developer fora, waarvan StackOverflow de meest gebruikte onder developers is (Feldman, 2017).

Voor het bouwen van de API is er een vergelijkend onderzoek gedaan met twee implementaties van GraphQL in Laravel: laravel-graphql en Lighthouse. Dit zijn de twee meest gebruikte en meest volwassen implementaties op dit moment. Laravel-graphql heeft 1789 stars op GitHub, Lighthouse heeft er 828 (GitHub, 2019a, -b). In beide implementaties is een GraphQL schema gebouwd waarmee een stad met bijbehorende gidsen kunnen worden opgehaald uit de backend van Hi,hi Guide.

Bij het vergelijken is gekeken naar twee variablen:

Complexiteit van de code

Het aantal stappen om de code werkend te krijgen moet zo laag mogelijk zijn. Dit zorgt ervoor dat de implementatie sneller te gebruiken is en (over het algemeen) simpeler is om te begrijpen. Ook moet de code zelf overzichtelijk zijn; er moet dus niet teveel code zijn die niet direct invloed heeft op de taak die bereikt moet worden met een functie. Dit hangt samen met het single-responsibility principle van SOLID code van Robert C. Martin (Oloruntoba, 2015).

Integratie met Laravel functionaliteit

Met de integratie van Laravel functionaliteit wordt de mate van gebruik maken van native Laravel functionaliteit bedoeld. Hieronder valt het gebruik maken van Laravel models en relationships om verschillende GraphQL types aan elkaar te koppelen.

3.2 Resultaten

Er zijn drie verschillende API architecturen die veel gebruikt worden. SOAP wordt gebruikt voor hele specialistische doeleinden (Wodehouse, z.d.). REST is een stateless API. Dit houdt in dat er niet een state is die wordt bijgehouden in een sessie op de back-end (Sandoval, 2017). Het is op dit moment de meest gangbare architectuur, maar heeft een aantal tekortkomingen, waaronder vooral inflexibiliteit (Castelo, z.d.). GraphQL is de nieuwste van de drie architecturen. Er is nog niet veel documentatie over te vinden, maar het is wel ontzettend flexibel en schaalbaar (Pandya, 2016). Dit maakt het erg geschikt voor startups.

Volgens Sandoval (2016) zijn de voornaamste voordelen van een GraphQL API:

- Een enkele endpoint voor alle requests, deze hoeft dus maar eenmalig ingesteld te worden in de front-end.
- De client krijgt precies de data terug waar hij om vraagt, dus niet een heel object waarvan er maar een paar key/values nodig zijn. Doordat de structuur van de query zelf te bepalen in

door de client, hoeft er ook maar een enkele query te worden gedaan om de relevante data op te halen.
- Er hoeven minder aanpassingen gedaan te worden aan de server kant, als er veranderingen clientside zijn. De client bepaalt welke data hoe wordt opgehaald. Dit maakt het systeem flexibel.

Een GraphQL is gemakkelijk schaalbaar. GraphQL is efficiënt om data op te halen, omdat data wordt opgehaald door middel van een query naar één enkel endpoint. Er worden dus niet meerdere functies aangeroepen om verschillende stukjes data op te halen, zoals het geval is bij REST. Data kan ook uit meerdere resources worden opgehaald met maar één call. GraphQL beschrijft alles in types en fields. Dit zorgt ervoor dat data altijd als het juiste type wordt verstuurd. Als een request niet klopt, kan GraphQL duidelijke errors geven. Ook wordt door de stricte typing zekerheid gegeven dat de data in een voorspelbaar formaat naar de frontend wordt gestuurd. De populariteit van GraphQL groeit nog steeds, waardoor er een groot ecosysteem omheen ontstaat. Er zijn al verschillende goede tools beschikbaar en er zijn steeds meer developers bezig met GraphQL. Er hoeft bijna niets aan een bestaande applicatie te worden aangepast als er nieuwe data beschikbaar moet komen via de API. Er hoeven alleen maar nieuwe fields aan het schema te worden toegevoegd, maar er hoeven geen nieuwe functies te worden geschreven (Manjunath, 2017; Miskiewicz, 2018; Vyshnevskyi, 2019).

React Native kan via simpele AJAX calls requests versturen naar een API, maar omdat er gebruik wordt gemaakt van GraphQL zijn er meer mogelijkheden om requests te versturen en beheren. Voor React Native zijn er drie belangrijke clients beschikbaar om dit te doen. Deze clients zijn Apollo Client, Relay en Urql (Expo, z.d.).

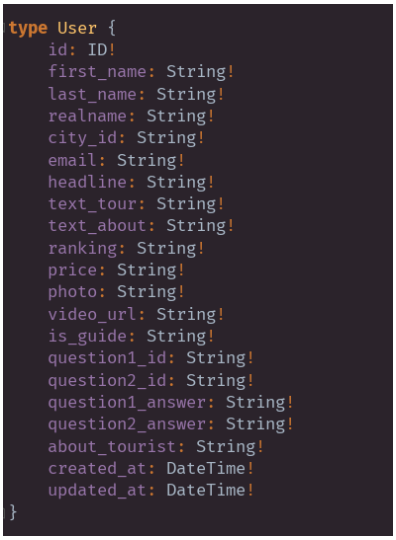
3.3 Praktijk

Er is gekozen om met een GraphQL architectuur een API te maken. Door GraphQL te verkiezen boven SOAP en REST is ervoor gezorgd dat de API snel gebouwd kon worden en zeer gemakkelijk te schalen is voor toekomstige toepassingen. Doordat de verantwoordelijkheid van de structuur van de response bij GraphQL aan de frontend kant ligt, was er weinig overleg nodig tussen frontend en backend om deze op elkaar aan te laten sluiten.

Om erachter te komen welke implementatie van GraphQL in Laravel het beste is, is er een test gedaan. Hieruit zijn een aantal resultaten gekomen.

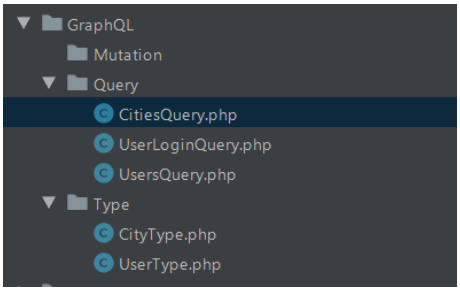
- Het opzetten van de API ging sneller. Met laravel-graphql was er ongeveer een dag nodig om dezelfde functionaliteit op te zetten die in Lighthouse binnen een uur kon worden opgezet.

Figuur 10. Schema in Laravel-graphql

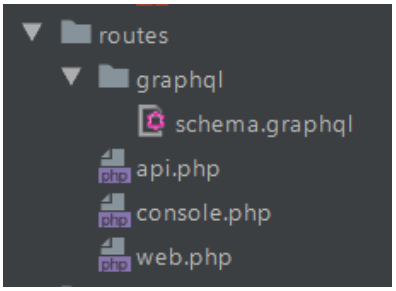


Figuur 11. *Schema in Lighthouse*

- Lighthouse maakt gebruik van de SDL taal van GraphQL zelf. Het schema is dus heel kort en direct, omdat de code is toegespitst op gebruik in een GraphQL schema (figuur 11). Bij laravel-graphql wordt het GraphQL schema eigenlijk vertaald naar PHP. Dit zorgt voor omslachtige en onduidelijkere code (figuur 10).

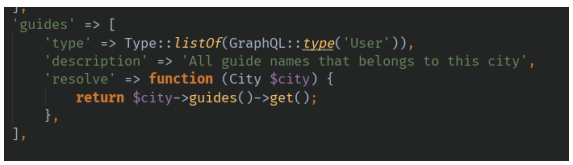


Figuur 12. Mapstructuur in laravel-graphql

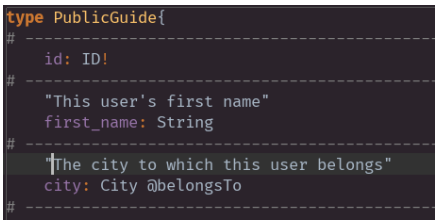


Figuur 13. Mapstructuur in Lighthouse

- Voor laravel-grapql moest het schema worden opgeknipt in losse stukken (figuur 12). Lighthouse heeft maar één bestand nodig waarin het GraphQL schema gedefinieerd wordt, waardoor alles op één plaats overzichtelijk bij elkaar staat (figuur 13).



Figuur 14. Relationship guide->city in laravel-graphql



Figuur 15. Relationship guide->city in Lighthouse

- Lighthouse maakt gebruik van de relationships tussen models. Dit zorgt voor hele directe verwijzingen tussen de types in het schema (figuur 15). In laravel-graphql moet er een resolver functie worden geschreven die eigenlijk hetzelfde doet als een al bestaande relationship (figuur 14).

- De cache van de Laravel configuratie hoeft niet steeds schoongemaakt te worden na elke wijziging. Dit moest wel bij laravel-graphql, omdat het GraphQL schema in deze bestanden werd aangepast. Deze configuratiebestanden zijn niet bedoeld om vaak aangepast te worden.

Uit deze test is gebleken dat Lighthouse de beste optie is. Voor de test, zie Bijlage 2 – Onderzoek GraphQL implementaties in Laravel.

Bij het ontwerpen van de API is al rekening gehouden met de app die in de toekomst voor de toeristen wordt ontwikkeld. Dit is gedaan door in de query's al een onderscheid te maken tussen de data die door een gids en de data die door een toerist kan worden opgevraagd. Deze GraphQL API is in de toekomst ook gemakkelijk uit te breiden, zodat partners hieraan kunnen koppelen.

Aan de frontend kant is gekozen voor Apollo Client, voornamelijk omdat deze de beste documentatie en de meeste mogelijkheden heeft. Voor de voordelen en nadelen van Apollo, Relay en Urql, zie Bijlage 3 - React Native implementaties van GraphQL.

3.4 Conclusie

Uit bovenstaande informatie is geconcludeerd dat GraphQL voor Hi,hi Guide het meest geschikt is. Als startup is flexibiliteit heel belangrijk. Deze flexibiliteit in het API schema zorgt ervoor dat de API schaalbaar is en in de toekomst gemakkelijk en goedkoop kan worden uitgebreid om nieuwe functionaliteit te ondersteunen. De beste manier om de API in Laravel te implementeren is door gebruik te maken van Lighthouse. Deze implementatie werkt het best en zorgt vooral ook voor korte overzichtelijke code door gebruik te maken van relationships in Laravel. Dit zorgt direct al voor minder technical debt. Vanuit React Native kan het best gebruik worden gemaakt van Apollo Client om met de API te communiceren, omdat deze de beste documentatie en meeste mogelijkheden heeft.

4 Deelvraag 4

Hoe kan nieuwe code binnen enkele seconden live getest worden?

4.1 Methodologie

Er is door middel van deskresearch onderzoek gedaan naar verschillende workflows om code live te zetten. Hierbij is gebruik gemaakt van blogs van developers (vooral via Medium) en developer fora, waarvan StackOverflow de meest gebruikte onder developers is (Feldman, 2017). Er is gezocht naar een workflow waarbij de code zo snel mogelijk live gezet kan worden.

4.2 Resultaten

Om code tussen developers onderling en de server te synchroniseren wordt gebruik gemaakt van git. Er bestaan meerdere manieren om git workflows logisch in te delen. Er zijn meerdere git workflows, waaronder Git Flow, GitHub Flow, GitLab Flow en One Flow. Deze hebben elk voor- en nadelen. Voor alle voor- en nadelen, zie Bijlage 4 - Git workflows. De Git Flow is de meest gebruikelijke manier om branches in te delen (Porto, 2018). Deze workflow wordt op dit moment ook in zekere mate gebruikt bij Hi,hi Guide. Een steeds populairdere workflow is Continuous Integration (CI). CI is een manier van developen waarbij teamleden hun werk regelmatig integreren; gewoonlijk doet iedereen dit minstens eenmaal per dag. Elke integration wordt gecheckt met een ingebouwde automatische test, zodat fouten zo snel mogelijk ontdekt worden. Volgens Fowler (2006) heeft dit een aantal voordelen, namelijk:

- Verminderd risico tijdens de uiteindelijke deploy, dit voorkomt onzekerheid op het stressvolle moment dat de code live gezet gaat worden voor gebruikers
- Er is altijd duidelijk hoe goed het product werkt
- Er hoeft minder veel code doorzocht te worden als er een bug gevonden wordt
- Er wordt voorkomen dat bugs stapelen, dus dat bugs zorgen voor andere bugs
- Zorgt voor eerdere toegang tot nieuwe features door gebruikers

Gebruik maken van CI leidt bij projecten tot een hogere kwaliteit en performance dan projecten die geen CI toepassen (Hilton, Tunnell, Huang, Marinov, & Dig, 2016).

Een CI workflow moet volgens Fowler ook aan een aantal eisen voldoen:

- Er moet 1 repository onderhouden worden
- De build moet geautomatiseerd worden
- De build moet zichzelf testen
- Iedereen moet elke dag naar de repository committen
- Elke commit moet de main branch opnieuw builden
- Kapotte builds moeten meteen gefixt worden
- De build moet snel draaien
- Tests moeten gebeuren in een clone van de production omgeving
- Het moet gemakkelijk zijn om voor iedereen de laatste versie te krijgen
- Iedereen kan zien wat er gebeurt
- Deployment moet geautomatiseerd worden

4.3 Praktijk

Tot nu toe moest er elke keer contact worden opgenomen met een developer buiten het bedrijf, als er nieuwe code op een live server getest moest worden. Het duurde daardoor vaak enkele dagen voordat er een nieuwe versie getest kon worden. Dit is opgelost door een testserver in te richten binnen de live VPS, die gekoppeld zit aan de git branch waar het nieuwe platform op ontwikkeld wordt. Hierdoor kan met één commando de nieuwste code live worden gezet. Nu kan iedereen binnen het bedrijf gelijk testen**.** Om de commando's die ook vaker moeten worden gerund (zoals migraten en nieuwe packages installeren of updaten met composer) te automatiseren, is een script geschreven in bash (zie Bijlage 5 - Bash script voor automatische installatie). Voor het live zetten van het volledige nieuwe Hi,hi Guide platform is een stappenplan gemaakt. Door deze stappen in de meest efficiënte volgorde uit te voeren, hoefde het platform niet lang offline te zijn. Er is ook gebruik gemaakt van een production branch op de git repository, zodat deze naar de live site kon worden gezet en met een symlink direct omgezet kon worden (zie Bijlage 6 – Stappenplan live zetten nieuw platform).

4.4 Conclusie

Er kan een server op een VPS gezet worden die code vanaf de git repository kan pullen. De server kan door middel van een bash script automatisch een aantal commands runnen om de boel klaar te zetten. De server, het bash script en het stappenplan waarmee het platform live is gezet kunnen als basis dienen van een volledig geautomatiseerde CI workflow.

Met het opzetten van de live testserver is voldaan aan een aantal van de vereisten voor een CI workflow. Er is namelijk één repository waar iedereen elke dag naartoe pusht, het is gemakkelijk voor iedereen om de laatste versie te krijgen en iedereen kan zien wat er gebeurt in de repository. Doordat de code nu direct live getest kan worden, is de workflow van Hi,hi Guide al dicht bij een CI workflow gebracht. Het bash script en het stappenplan waarmee het platform live is gezet zijn een eerste stap naar een snelle geautomatiseerde build. Op dit moment gebeurt het testen op de live omgeving nog handmatig. Door tests te schrijven die automatisch worden uitgevoerd bij een push naar de repository, kan een volledige CI workflow worden gerealiseerd.

5 Deelvraag 5

Welke manieren zijn er om persoonlijke data te beschermen?

5.1 Methodologie

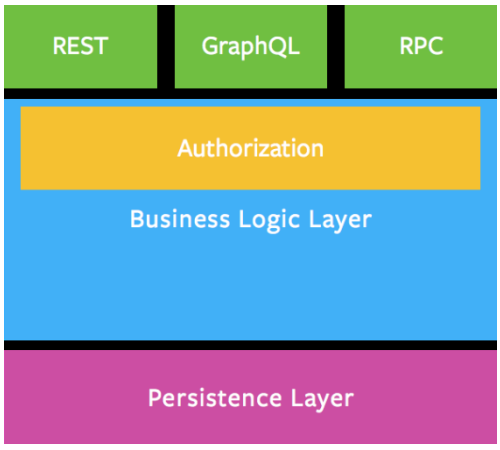
Er is door middel van deskresearch onderzoek gedaan naar manieren om API te beveiligen. Er is gekeken naar beveiliging van een standaard REST API en beveiliging van een GraphQL API. Ook is onderzocht hoe dit het best in Laravel gedaan kan worden. Hierbij is gebruik gemaakt van blogs van developers (vooral via Medium) en developer fora, waarvan StackOverflow de meest gebruikte onder developers is (Feldman, 2017).

5.2 Resultaten

Om gebruikersdata te beschermen moet er een goede beveiliging op de endpoints van de API zitten. Beveiliging van een backend bestaat uit twee delen die vaak met elkaar verward worden: authenticatie en autorisatie. Authenticatie is het vaststellen van de identiteit van de client. Dit gebeurt meestal door middel van een gebruikersnaam en wachtwoord. Autorisatie is het verlenen van toegang tot data aan de hand van opgestelde regels (Kerrek, 2011).

Om een REST API te beveiligen zijn er verschillende methoden beschikbaar, zoals Basic Auth, HMAC, OAuth en OAuth2 (Levin, 2016; Walls, Donald, & Clarkson, 2014). De meest gebruikte methode is tegenwoordig OAuth2 (Raible, 2017). Dit is een methode waarbij een externe provider een gebruiker kan autoriseren om toegang te krijgen tot bepaalde data binnen een applicatie. Wanneer de provider een gebruiker toegang verleent (omdat de gebruiker bijvoorbeeld het juiste emailadres en wachtwoord kan overhandigen), krijgt deze gebruiker een access token van de provider. Deze token kan vervolgens worden gebruikt om elk request te signeren, zodat de server weet wie deze gebruiker is en dat hij te vertrouwen is (Walls, Donald, & Clarkson, 2014) OAuth is hiermee een versimpelde versie van OAuth, die toch net zo veilig is (Menier, 2014).

In GraphQL moeten de authenticatie en autorisatie net als bij een REST API worden afgehandeld binnen de business logic layer van de applicatie, en dus niet binnen het GraphQL schema zelf. Wanneer de autorisatie in GraphQL zelf wordt afgehandeld, bestaat er de mogelijkheid dat er wordt vergeten bepaalde fields met data te beschermen, waardoor de veiligheid van de data in het geding komt.



Figuur 16. De plaats van de autorisatielaag binnen een applicatie (Facebook Inc., z.d.-a).

In figuur 16 is duidelijk te zien dat de autorisatie niet de verantwoordelijkheid is van GraphQL, maar binnen de business logic (HTTP) laag van de back-end valt. In Laravel valt dit dus binnen de

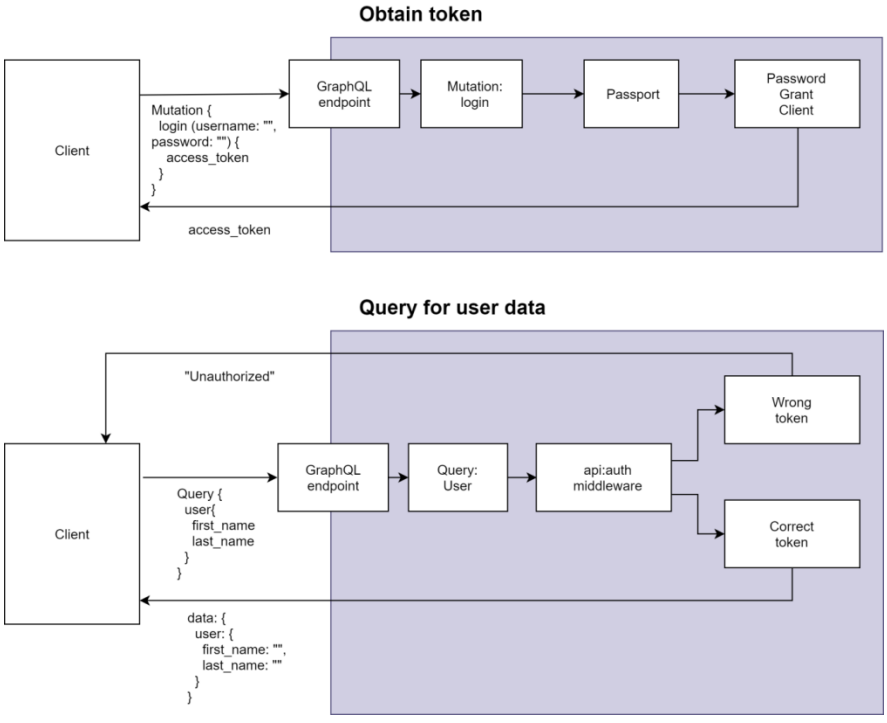
verantwoordelijkheid van de middleware en controllers. Door de autorisatie naar de business logic laag te delegeren kunnen er geen fouten in de graph gemaakt worden die zouden leiden tot datalekken (Facebook Inc., z.d.-a, b).

GraphQL kan ook gebruik maken van OAuth2, dit staat namelijk los van het GraphQL schema (Allsopp, 2016). De meest gebruikte methode om OAuth2 in Laravel te implementeren is door gebruik te maken van het package Laravel Passport. Passport is speciaal gebouwd voor API autorisatie, is veelvuldig getest en is de standaard in veel applicaties. Passport wordt niet alleen binnen Laravel gebruikt, maar is voor veel verschillende frameworks en programmeertalen de standaard (Otwell, z.d.-b, c).

5.3 Praktijk

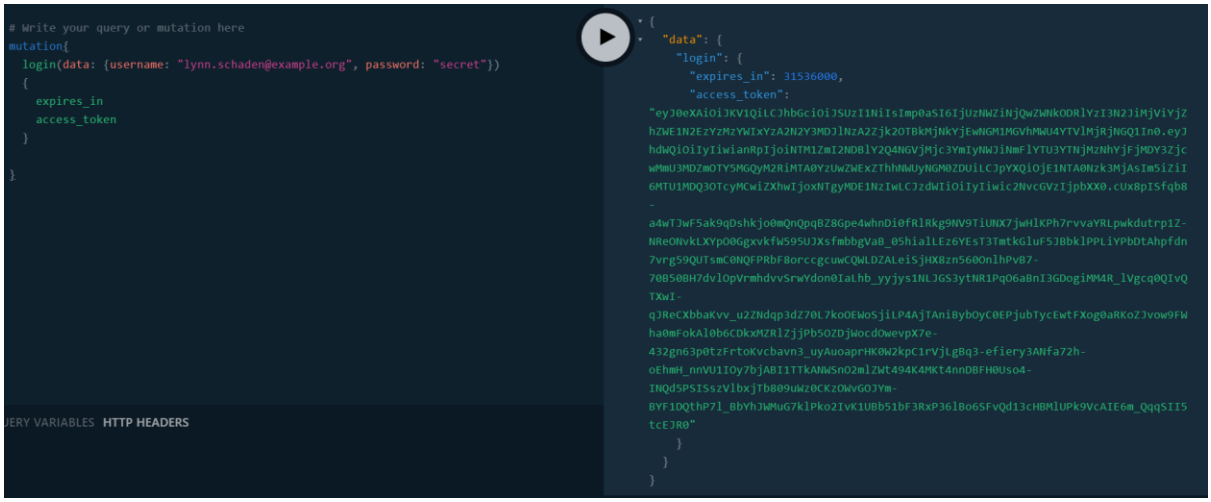
De GraphQL API is beveiligd door gebruik te maken van API authenticatie middleware van Laravel Passport. Deze middleware zorgt ervoor dat alleen een ingelogde gebruiker bij zijn data kan. De middleware maakt gebruikt van OAuth2 om gebruikers te authentiseren en autoriseren. Deze middleware maakt deel uit van de business logic laag van Laravel.

De gebruiker van de API kan met een mutation op het GraphQL endpoint een access token verkrijgen van Passport. Deze token wordt vervolgens meegestuurd met elke request om deze te autoriseren. Zo verloopt de gehele login flow via GraphQL en is deze ook stateless. Er is namelijk geen session nodig om de ingelogde gebruiker te bewaren, omdat elke request apart geverifiëerd wordt met de access token. Deze login flow is geïllustreerd in figuur 17.



Figuur 17. GraphQL login flow

Door de Login mutation te doen (met een email en password) kan een access token van Passport worden verkregen (figuur 18).

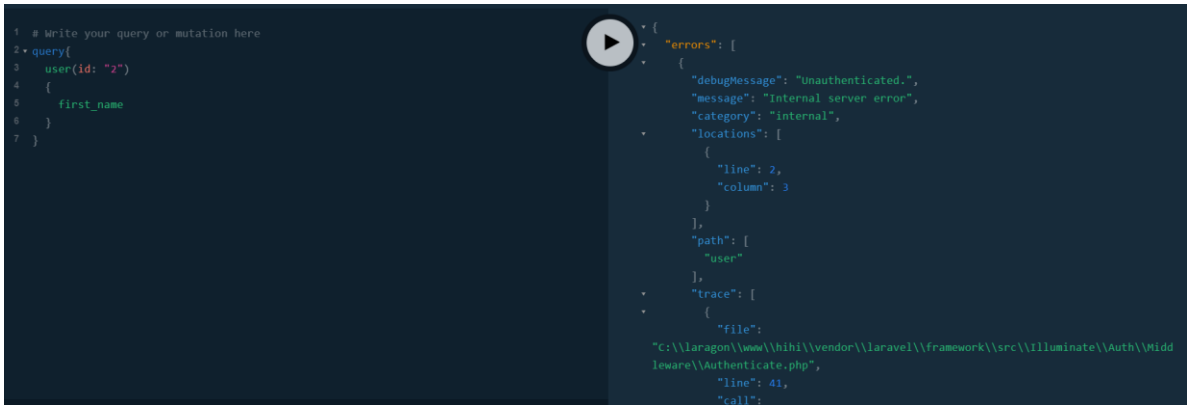


Figuur 18. Login mutation om access token te bemachtigen

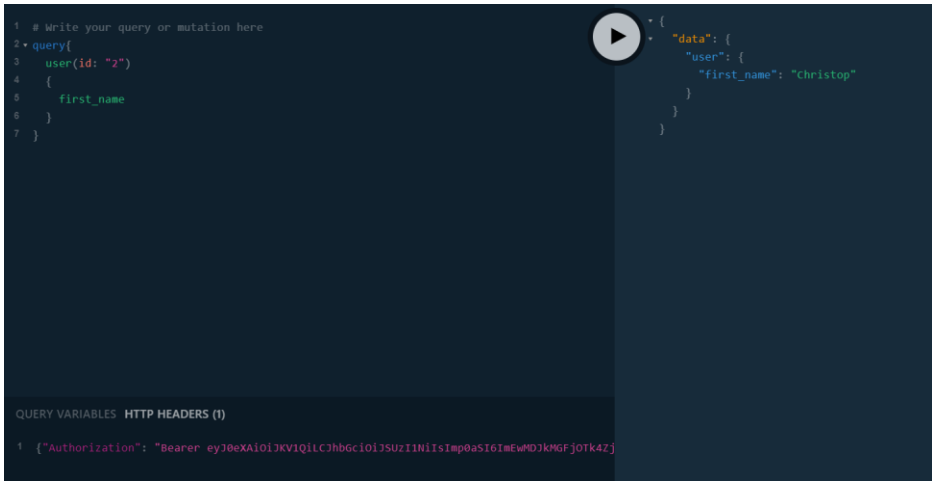
Deze token kan vervolgens worden gebruikt om requests te doen naar de API routes. De API middleware is in het GraphQL schema op de fields gezet door de @middleware directive toe te voegen (figuur 19).

Figuur 19. Middleware op de fields in het GraphQL schema

De fields die zijn beschermd met de middleware zijn nu enkel nog toegankelijk wanneer vanuit de frontend een request wordt gedaan met de Authorization header die de juiste token bevat. In figuur 20 is te zien dat de GraphQL endpoint “Unauthenticated” terug stuurt als de token niet of verkeerd wordt meegestuurd. In figuur 21 is te zien dat de gebruiker toegang heeft tot de data wanneer de juiste token met de Authorization header wordt meegestuurd bij een request.



Figuur 20. “Unauthenticated” wanneer gebruiker van de API niet is ingelogd



Figuur 21. Gebruiker kan bij zijn data door gebruik te maken van de verkregen token

5.4 Conclusie

De API kan beveiligd worden door OAuth2 toe te passen met de API middleware van Laravel Passport. De middleware van Passport zit in de business logic laag van de applicatie. Deze laag staat buiten GraphQL, waardoor er geen datalekken kunnen ontstaan door een eventuele fout in het GraphQL schema. Door het gebruik van OAuth2 wordt de persoonlijke data van gebruikers op een simpele manier toch zeer goed beschermd. Deze bescherming is belangrijk omdat Hi,hi Guide werkt met de data van duizenden gebruikers en deze absoluut niet uit mag lekken.

6 Conclusies

Welke stappen moeten worden ondernomen om de technical debt bij Hi,hi Guide te verminderen?

Samengevat komen uit de deelvragen de volgende stappen naar voren:

1. Er moet gebruikt worden gemaakt van migrations en seeders bij aanpassingen aan de database.
2. Er moet gezorgd worden dat de Laravel code volgens de MVC architectuur wordt opgebouwd.
3. Er moet een GraphQL API geïmplementeerd worden door gebruik te maken van Lighthouse. Vanuit de React Native app moet gebruik worden gemaakt van Apollo Client om met deze API te communiceren.
4. Er moet een server op een VPS gezet worden die code vanaf de git repository kan pullen en automatisch een aantal commands kan runnen om de boel klaar te zetten. Dit is meteen een goede opstap richting een volledige Continuous Integration workflow.
5. De API moet beveiligd worden door OAuth2 te implementeren met de API middleware van Laravel Passport.

Door deze stappen te ondernemen kan de technical debt bij Hi,hi Guide significant verminderd worden. Dit zorgt voor een stabiele, veilige applicatie die sneller ontwikkeld kan worden dan nu. Met het toevoegen van een GraphQL API wordt het mogelijk om de mobile app te koppelen aan het bestaande platform op een manier die schaalbaar en toekomstbestendig is.

Deze stappen zijn inmiddels in de praktijk toegepast in het vernieuwde platform van Hi,hi Guide.

7 Aanbevelingen

De technical debt binnen het platform van Hi,hi Guide kan op een aantal punten nog verder worden verminderd.

7.1 Indeling van de database verbeteren

De structuur en benamingen binnen de database kunnen nog verfijnd worden. De tabel voor bookings heet bijvoorbeeld ‘booking’, terwijl hier meerdere bookings in staan. Laravel begrijpt dan ook niet dat de tabel ‘booking’ bij het model ‘Booking’ hoort, omdat de conventie is dat de bijbehorende tabel dan ‘bookings’ zou heten. Hierdoor moet er in Laravel weer code worden toegevoegd om deze tabel aan dit model te koppelen. Deze extra code kan worden voorkomen met een betere naamgeving.

De term ‘guide_bookings’ dekt eigenlijk niet de lading. Dit zijn gewoon offertes van gidsen. Deze kunnen dus ook gewoon ‘offers’ genoemd worden.

Een gebruiker heeft nu een first_name, last_name (nog ongebruikt) en een real_name (volledige naam). Dit kan beter worden ingedeeld in first_name en last_name. Dan is het in de code gelijk duidelijk welke naam wordt gepakt. Ook is het dan gemakkelijker om bijvoorbeeld de achternaam af te schermen en alleen de voornaam te tonen.

Om de database verder te verbeteren is begonnen aan een plan om het boekingensysteem volledig te veranderen, door elke booking op te splitsen in requests, bookings en trips (zie voor meer informatie Bijlage 8 - Concept nieuw systeem voor boekingen beheren in backend)

7.2 Tests automatiseren

Op dit moment moet er steeds met de hand worden getest op het moment dat de site live wordt gezet. Hi,hi Guide wil er naartoe dat er continu nieuwe code live kan worden gezet, maar dit kan niet zomaar zonder dat er tests voor de code worden geschreven. Het mag nooit zo zijn dat er fouten in de live site zitten die niet worden opgemerkt door Hi,hi Guide. Door goede tests te schrijven kan dit worden afgevangen, zodat de developer op de hoogte wordt gesteld van fouten in de code op het moment dat hij de code probeert live te zetten. Geautomatiseerd testen geeft stabiliteit aan het systeem en voorkomt vele kostbare uren handmatig testen. Dit testen is een eerste opstap naar een systeem dat volledig gebruik maakt van Continuous Integration om snel nieuwe code te kunnen deployen naar gebruikers. Dit zou ervoor zorgen dat code niet alleen binnen enkele seconden live draait, maar ook dat dit op een veilige en stabiele manier gebeurt.

8 Discussie

8.1 Resultaten en verwachtingen

Door het werken aan het platform van Hi,hi Guide is gebleken hoe belangrijk het is om de technical debt van software in te lossen voordat er wordt begonnen aan het bouwen van nieuwe features. Het is een taak die niet direct zichtbaar resultaat levert aan de voorkant van de applicatie, maar wel van wezenlijk belang is om snel te kunnen blijven groeien. Features bouwen terwijl er nog een technical debt openstaat is te vergelijken met het verbouwen van een huis met achterstallig onderhoud. Er is flink aan het huis verbouwd, maar de woonkamer waar je altijd zit is nog steeds aan het tochten en de hypotheek is alleen maar hoger geworden.

Bij Hi,hi Guide zat de grootste technical debt in de opbouw van de Laravel applicatie. Dit is ook heel begrijpelijk. Er moest zo snel mogelijk een platform komen waarop toeristen boekingen kunnen doen om een cash flow op gang te krijgen. Snel een platfom bouwen is op dat moment ook de beste keus geweest, maar het is belangrijk dat er gerealiseerd wordt dat dit wel een schuld op technisch gebied heeft opgeleverd. Zo'n schuld moet eerst ingelost worden, voordat een platform zich verder kan ontwikkelen.

Door het verminderen van de technical debt is het gelukt om al een aantal nieuwe features te realiseren voor Hi,hi Guide. Het bestaande platform is volledig herbouwd, zowel aan de voorkant als achterkant. Dit nieuwe platform is op 29 maart live gegaan en wordt nu gebruikt door gidsen en toeristen wereldwijd.

De belangrijkste nieuwe features zijn:

- Meer interactie vanuit gids door reactie-mogelijkheid
- Video uploader voor de gidsen
- Gidsprofiel en toeristprofiel ondergebracht onder één type profiel
- Persoonlijker gidsprofiel door middel van vragen en antwoorden
- Mobiele app die gidsen door middel van notificaties op de hoogte stelt van nieuwe verzoeken

Zie Bijlage 7 - Gerealiseerde features voor uitgebreidere informatie en afbeeldingen.

8.2 Aanbevelingen voor vervolgonderzoek

Hi,hi Guide heeft ervoor gekozen om eerst een community te bouwen met gidsen die aan de formule van hun belevingstoerisme voldoen en dit leuk vinden. Er is een netwerk opgebouwd met 2000 gidsen over de hele wereld. De volgende stap is om toeristen die een boeking hebben gedaan betrokken te houden bij deze community, zodat ook zij ambassadeurs worden van Hi,hi Guide.

Het is belangrijk dat een toerist direct na de ontmoeting met de gids zijn of haar ervaring gemakkelijk kan delen met anderen.

8.3 Validiteit en betrouwbaarheid

Dit onderzoek heeft zich beperkt tot alleen het bedrijf Hi,hi Guide, hoewel de problematiek rondom technical debt veel breder is en bij veel meer bedrijven voorkomt. De deelvragen zijn wel gekozen met een wat bredere kijk dan alleen Hi,hi Guide. Deze kunnen dus vrij gemakkelijk gegeneraliseerd worden, vooral voor bedrijven die ook gebruik maken van Laravel.

De interne validiteit van dit onderzoek zou vergroot kunnen worden door specifiekere testen met duidelijk meetbare resultaten te doen. Er is echter in dit onderzoek voor gekozen om dit niet te

doen, omdat heel specifieke resultaten voor een snelgroeiend bedrijf als Hi,hi Guide veel minder relevant zijn dan het snel in de praktijk brengen van het onderzoek.

Er moet nog wel worden afgewacht of de praktijk laat zien dat de engagement met de gidsen inderdaad hoger wordt na het verbeteren van het platform en het uitbrengen van de mobiele app.

Bij Hi,hi Guide heeft het verminderen van de technical debt in ieder geval geleid tot het verbeteren van het gehele platform en het geschikt maken voor een koppeling met de nieuwe mobiele app. Ook is het platform nu veel gemakkelijker uit te breiden met nieuwe features, waardoor het bedrijf snel kan groeien.

Deze manier van werken kan natuurlijk ook worden toegepast bij andere technische bedrijven om hun technical debt te verminderen. Daardoor kunnen ook daar veel technologische stappen gezet worden.

Afkortingen

AJAX	Asynchronous Javascript And XML
API	Application Programming Interface
GraphQL	Graph Query Language
MVC	Model – View – Controller
REST	Representational State Transfer
SDL	Schema Definition Language
SOAP	Simple Object Access Protocol

Begrippenlijst

Authenticatie	Het vaststellen van de identiteit van de client
Autorisatie	Het verlenen van toegang tot data, aan de hand van opgestelde regels
Application Programming Interface	Een set aan definities waarmee softwareprogramma's onderling kunnen communiceren
Backend	De achterkant van de app (in dit geval de Laravel server). Dit is wat de meeste functionaliteit van de app regelt, waar de data wordt opgeslagen en waar de authenticatie plaatsvindt.
Blade	De server-side templating engine van Laravel
Business Logic	Vertaling van de business rules naar code
Business Rules	Regels die beschrijven hoe het bedrijf opereert
Endpoint	Een URL waarop een HTTP method gedaan kan worden om toegang te krijgen tot functionaliteit van een API
Frontend	De voorkant van de app (in dit geval de React Native client). Dit is wat de gebruiker ziet en waar hij mee interacteert.
Graph Query Language	Een query language en specificatie voor API's, oorspronkelijk ontwikkeld door Facebook voor intern gebruik als alternatief voor REST
Laravel	PHP framework dat op de backend draait en is gebaseerd op het MVC principe
Middleware	Laag software die draait voordat een request wordt doorgestuurd naar de rest van de backend
Mobile app	App die werkt op iOS en Android
Model - View – Controller	Een manier van programmeren waarbij de applicatie in lagen is opgebouwd die elk een eigen verantwoordelijkheid hebben. Het model is een abstractie van de database, de view is wat de gebruiker ziet en waar hij mee kan interacteren, de controller zorgt voor de logica.
PHP	Veelgebruikte backend programmeertaal
React	Verwijst hier naar React Native (is heel vergelijkbaar met ReactJS). Dit is een JavaScript Framework waarmee op een zeer modulaire manier applicaties kunnen worden ontwikkeld.
Schema Definition Language	De taal waarin GraphQL schema's geschreven worden

Literatuurlijst

Allsopp, C. (2016, 23 januari). GraphQL and Authentication. Geraadpleegd op 8 april 2019, van https://medium.com/the-graphqlhub/graphql-and-authentication-b73aed34bbeb

Bakker, F. (2013, 26 september). Technical debt vaak vergeten bij modernisatie. Geraadpleegd op 26 maart 2019, van https://www.computable.nl/artikel/opinie/development/4878049/1509029/technical-debt-vaak-vergeten-bij-modernisatie.html

Bettilyon, T. E. (2018, 11 januari). What Is an API and Why Should I Use One? [Blogpost]. Geraadpleegd op 5 april 2019, van https://medium.com/@TebbaVonMathenstien/what-is-an-api-and-why-should-i-use-one-863c3365726b

Castelo, A. (z.d.). Laravel API Tutorial: How to Build and Test a RESTful API. Geraadpleegd op 24 februari 2019, van https://www.toptal.com/laravel/restful-laravel-api-tutorial

Expo. (z.d.). Using GraphQL. Geraadpleegd op 25 februari 2019, van https://docs.expo.io/versions/latest/guides/using-graphql/

Facebook Inc.. (z.d.-a). Thinking in Graphs. Geraadpleegd op 25 februari 2019, van https://graphql.org/learn/thinking-in-graphs/

Facebook Inc.. (z.d.-b). Autorisatie. Geraadpleegd op 25 februari 2019, van https://graphql.org/learn/authorization/

Feldman, B. (2017, 24 maart). The Hidden Power of Stack Overflow: How a Website You've Never Heard of Is Holding the Web Together. Geraadpleegd op 5 april 2019, van http://nymag.com/intelligencer/2017/03/the-hidden-power-of-stack-overflow.html

FormidableLabs. (2019, 18 februari). FormidableLabs/urql. Geraadpleegd op 25 februari 2019, van https://github.com/FormidableLabs/urql

Fowler, M. (2006, 1 mei). Continuous Integration. Geraadpleegd op 1 april 2019, van https://martinfowler.com/articles/continuousIntegration.html

GitHub. (2019a). folkloreinc/laravel-graphql. Geraadpleegd op 5 april 2019, van https://github.com/folkloreinc/laravel-graphql

GitHub. (2019b). nuwave/lighthouse. Geraadpleegd op 5 april 2019, van https://github.com/nuwave/lighthouse

Hilton, M., Tunnell, T., Huang, K., Marinov, D., & Dig, D. (2016). Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. Geraadpleegd van http://cope.eecs.oregonstate.edu/papers/OpenSourceCIUsage.pdf

Ighodaro, N. (2018, 24 mei). How Laravel implements MVC and how to use it effectively. Geraadpleegd op 1 april 2019, van https://blog.pusher.com/laravel-mvc-use/

Jensen, O. (2018, 18 mei). 5 Ways Technical Debt Impacts Your Business. Geraadpleegd op 8 april 2019, van https://www.clearlaunch.com/technical-debt-impact/

Kerrek, S. B. (2011, 2 juli). Authenticatie versus Autorisatie. Geraadpleegd op 25 februari 2019, van https://stackoverflow.com/questions/6556522/authentication-versus-authorization

Korop, P. (2015, 30 juli). Generating fake Seeds data with Faker package. Geraadpleegd op 1 april 2019, van https://laraveldaily.com/generating-fake-seeds-data-with-faker-package/

Levin, G. (2016, 28 november). RESTful API Authentication Basics. Geraadpleegd op 3 maart 2019, van https://blog.restcase.com/restful-api-authentication-basics/

Manjunath, M. (2017, 10 oktober). Build a React App With a Laravel RESTful Back End. Geraadpleegd op 25 februari 2019, van https://code.tutsplus.com/tutorials/build-a-react-app-with-laravel-restful-backend-part-1-laravel-5-api--cms-29442

Menier, T. (2014, 6 januari). OAuth 2.0: Benefits and use cases - why? Geraadpleegd op 8 april 2019, van https://stackoverflow.com/questions/7561631/oauth-2-0-benefits-and-use-cases-why

Miskiewicz, A. (2018, 15 juni). Using GraphQL + Apollo at Expo. Geraadpleegd op 24 februari 2019, van https://blog.apollographql.com/using-graphql-apollo-at-expo-4c1f21f0f115?gi=4fbdf2dc918f

Nitsche, S. (2017, 28 november). One does not simply update a database. Geraadpleegd op 1 april 2019, van https://dev.to/pesse/one-does-not-simply-update-a-database--migration-based-database-development-527d

Oloruntoba, S. (2015, 18 maart). S.O.L.I.D: The First 5 Principles of Object Oriented Design. Geraadpleegd op 1 april 2019, van https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

Osetskyi, V. (2018, 13 juli). What Technical Debt Is and How to Calculate It. Geraadpleegd op 26 maart 2019, van https://dzone.com/articles/what-technical-debt-it-and-how-to-calculate-it

Otwell, T. (z.d.-a). Laravel Homestead. Geraadpleegd op 25 februari 2019, van https://laravel.com/docs/5.7/homestead

Otwell, T. (z.d.-b). Request Lifecycle. Geraadpleegd op 25 februari 2019, van https://laravel.com/docs/5.7/lifecycle

Otwell, T. (z.d.-c). API Authenticatie (Passport). Geraadpleegd op 25 februari 2019, van https://laravel.com/docs/5.7/passport

Otwell, T. (z.d.-d). Database: Migrations. Geraadpleegd op 1 april 2019, van https://laravel.com/docs/5.8/migrations

Pandya, D. (2016, 18 augustus). GraphQL Concepts Visualized. Geraadpleegd op 25 februari 2019, van https://blog.apollographql.com/the-concepts-of-graphql-bc68bd819be3?gi=cb23d0051308

Phoenix. (z.d.). Seeding Data. Geraadpleegd op 1 april 2019, van https://phoenixframework.org/blog/seeding-data

Porto, P. (2018, 27 februari). 4 branching workflows for Git. Geraadpleegd op 1 april 2019, van https://medium.com/@patrickporto/4-branching-workflows-for-git-30d0aaee7bf

Power, K. (2013). Understanding the impact of technical debt on the capacity and velocity of teams and organizations: Viewing team and organization capacity as a portfolio of real options.

Geraadpleegd van https://www.researchgate.net/publication/271437496_Understanding_the_impact_of_technical_debt_on_the_capacity_and_velocity_of_teams_and_organizations_Viewing_team_and_organization_capacity_as_a_portfolio_of_real_options/download

Prisma. (2018, 20 september). prisma/graphql-request. Geraadpleegd op 25 februari 2019, van https://github.com/prisma/graphql-request

Raible, M. (2017, 21 juni). What the Heck is OAuth? Geraadpleegd op 8 april 2019, van https://developer.okta.com/blog/2017/06/21/what-the-heck-is-oauth

Rozen, M. (2018, 4 augustus). Apollo Vs Relay: Picking a GraphQL Client. Geraadpleegd op 25 februari 2019, van https://maxrozen.com/2018/08/04/apollo-vs-relay-picking-a-graphql-client

Sandoval, K. (2016, 7 oktober). 5 Potential Benefits of Integrating GraphQL. Geraadpleegd op 3 maart 2019, van https://nordicapis.com/5-potential-benefits-integrating-graphql/

Sandoval, K. (2017, 11 mei). Defining Stateful vs Stateless Web Services. Geraadpleegd op 24 februari 2019, van https://nordicapis.com/defining-stateful-vs-stateless-web-services/

Vyshnevskyi, V. (2019, 16 maart). Building scalable products with GraphQL: business aspects and benefits. Geraadpleegd op 1 april 2019, van https://apiko.com/blog/building-scalable-apps-with-graphql/

Walls, C., Donald, K., & Clarkson, R. (2014, 23 april). Spring Social Reference. Geraadpleegd op 3 maart 2019, van https://docs.spring.io/spring-social/docs/1.1.0.RELEASE/reference/htmlsingle/

Wikipedia. (2019, 15 maart). Robert C. Martin. Geraadpleegd op 1 april 2019, van https://en.wikipedia.org/wiki/Robert_C._Martin

Wikipedia. (2018, 8 maart). Database seeding. Geraadpleegd op 1 april 2019, van https://en.wikipedia.org/wiki/Database_seeding

Wodehouse, C. (z.d.). SOAP vs. REST: A Look at Two Different API Styles. Geraadpleegd op 24 februari 2019, van https://www.upwork.com/hiring/development/soap-vs-rest-comparing-two-apis/

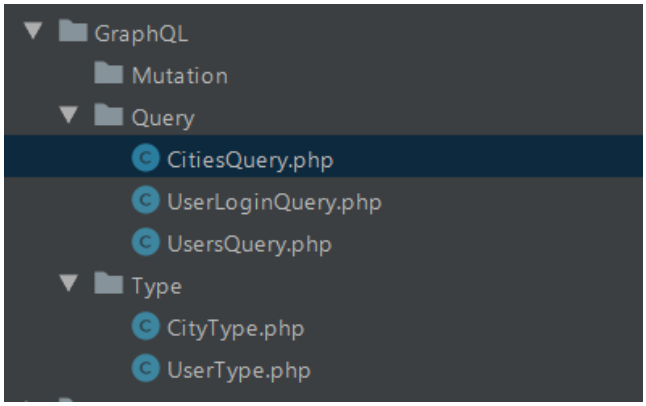
Bijlage 1 – Testen van migrations

Nieuw project gemaakt in Laragon
Nieuwe database gemaakt
Huidige live database geexporteerd van live site
Huidige live database geimporteerd in nieuwe database
php artisan config:cache
php artisan reset:migrations
php artisan migrate
Migrate werkt
php artisan db:seed
Error:
SQLSTATE[23000]: Integrity constraint violation: 1062 Duplicate entry '1' for key 'PRIMARY' (SQL: insert into `country` (`id`,
`name`, `slug`, `created_at`, `updated_at`) values (1, The Netherlands, the-netherlands, ,))
Countries, cities, languages en activities staan al in de live db, hoeven we dus niet te seeden
php artisan db:seed --class=QuestionsTableSeeder
php artisan db:seed --class=AboutTourTableSeeder
php artisan passport:install
Password grant client secret uit oauth_clients tabel gekopieerd naar .env file
php artisan optimize

Bijlage 2 - Onderzoek GraphQL implementaties in Laravel

Laravel-graphql

Om te beginnen maakt laravel-graphql een GraphQL mapje aan met daarin Mutations, Queries en Types.



Figuur 1. Mapstructuur Laravel-graphql

In de type wordt aangegeven welke fields op te vragen zijn en welke types dit zijn.

```php
class CityType extends GraphQLType
{
    protected $attributes = [
        'name' => 'City',
        'description' => 'A city'
    ];

    public function fields()
    {
        return [
            'id' => [
                'type' => Type::nonNull(Type::int()),
                'description' => 'Id of city'
            ],
            'name' => [
                'type' => Type::string(),
                'description' => 'Name of city'
            ],
            'name_cn' => [
                'type' => Type::string(),
                'description' => 'Chinese name of city'
            ],
            'image' => [
                'type' => Type::string(),
                'description' => 'Url to image of the city'
            ],
            'guides' => [
                'type' => Type::listOf(GraphQL::type('User')),
                'description' => 'All guide names that belongs to this city',
                'resolve' => function (City $city) {
                    return $city->guides()->get();
                },
            ],
        ];
    }
}
```

Figuur 2. City type met daarin een aantal fields

In de query wordt aangegeven wat de query moet returnen en wat voor arguments er mee kunnen worden gegeven.

```php
class CitiesQuery extends Query
{
    protected $attributes = [
        'name' => 'cities'
    ];

    public function type()
    {
        return Type::listOf(GraphQL::type('City'));
    }

    public function args()
    {
        return [
            'id' => ['name' => 'id', 'type' => Type::int()],
            'name' => ['name' => 'name', 'type' => Type::string()],
        ];
    }

    public function resolve($root, $args)
    {
        if (isset($args['id'])) {
            return City::where('id' , $args['id'])->get();
        } else if(isset($args['name'])) {
            return City::where('name', $args['name'])->get();
        } else {
            return City::all();
        }
    }
}
```

Figuur 3. Cities query die een id en name returned

Vervolgens worden de queries en types aangegeven in de graphql.php config.

```php
'schemas' => [
    'default' => [
        'query' => [
            //'users' => \App\GraphQL\Query\UsersQuery::class,
            'login' => \App\GraphQL\Query\UserLoginQuery::class,
            'cities' => \App\GraphQL\Query\CitiesQuery::class,
        ],
        'mutation' => [

        ]
    ],
    'protected' => [
        'query' => [
            'users' => \App\GraphQL\Query\UsersQuery::class,
        ],
        'mutation' => [

        ]
    ]
],
```

```php
'types' => [
    'User' => \App\GraphQL\Type\UserType::class,
    'City' => \App\GraphQL\Type\CityType::class,
],

/*
```

Figuur 4. De verschillende schema's in de config bestanden

Op deze manier is het gelukt om via de /graphiql route (een GraphQL test interface) queries te kunnen doen die ook de juiste data returnen.

Met de volgende query kan dan een stad worden opgevraagd:

Figuur 5. Query naar de cities in GraphiQL

En door guides ook toe te voegen aan de query kunnen alle gidsen in een stad worden opgevraagd:

Figuur 6. Query naar de cities, inclusief gidsen in GraphiQL

Door het ‘name’ argument toe te voegen, returnt de query nu 1 specifieke stad:

```
{
  cities (name:"Amsterdam"){
    id
    name
    image
    guides{
      id
      first_name
    }
  }
}
```

```
{
  "data": {
    "cities": [
      {
        "id": 1,
        "name": "Amsterdam",
        "image": "'/images/cities/Amsterdam_C.jpg'",
        "guides": [
          {
            "id": 58,
            "first_name": "henk"
          },
          {
            "id": 60,
            "first_name": "test"
          }
        ]
      }
    ]
  }
}
```

Figuur 7. Query naar de cities met name argument in GraphiQL

Laravel-graphql werkt dus om de gidsen vanuit een stad op te kunnen halen door middel van GraphQL.

Er zijn wel meerdere problemen aan het licht gekomen:

- Er is veel boilerplate code nodig om de boel op te zetten. Er zijn meerdere stukken code die een stuk korter en duidelijker hadden kunnen zijn. Het City type is bijvoorbeeld een class met attributes en een functie ‘fields’, deze functie returnt een array met fields die elk een type en description hebben (zie figuur 2). Dit zou veel korter geschreven kunnen worden, op dezelfde manier zoals een query naar GraphQL wordt geschreven in plaats van een class met functie en arrays. Waarschijnlijk komt dit doordat de eigen GraphQL SDL taal hier vertaald wordt naar PHP.
- Authenticatie wordt lastig om te maken, dit moet waarschijnlijk per field of door een apart schema te maken voor een ingelogde user
- Relationships tussen modellen in een query moeten een beetje custom en hacky in elkaar worden gezet. Er kan namelijk geen directe verwijzing naar een Eloquent relationship worden gemaakt, waardoor er een resolver met een anonieme functie moet worden aangeroepen, die eigenlijk hetzelfde doet als een relationship.

```
'guides' => [
    'type' => Type::listOf(GraphQL::type('User')),
    'description' => 'All guide names that belongs to this city',
    'resolve' => function (City $city) {
        return $city->guides()->get();
    },
],
```

Figuur 8. Custom resolver voor de guides (zelfde functionaliteit als een relationship)

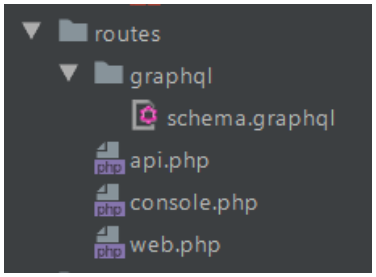
- Queries en types moeten steeds aan de Laravel configs voor GraphQL worden toegevoegd, vervolgens moet er een artisan config:clear worden gedaan om de configs te herladen. Dit moet na elke aanpassing. Dit komt doordat de queries en types in een config bestand worden toegevoegd en deze niet echt bedoeld zijn om regelmatig aan te passen.

Uit deze tests is gebleken dat GraphQL ontzettend handig werkt, want de queries zijn heel flexibel. Doordat je werkt met 1 route die verschillende queries kan afhandelen, hoeft er niet in de backend

voor elke frontend functionaliteit een nieuwe route met bijbehorende functies gemaakt te worden. Dit scheelt veel programmeerwerk in de backend, en er is minder communicatie nodig tussen frontend en backend developers, omdat de frontend bepaalt wat voor data hij krijgt en hoe. De backend bepaalt dan alleen welke data toegankelijk is voor de frontend. De implementatie laravel-graphql is echter niet heel handig vanwege bovengenoemde punten. Na deze test zijn dezelfde queries gebouwd in Lighthouse. Dit is een andere implementatie van GraphQL voor Laravel.

Lighthouse

Voor Lighthouse is veel minder configuratie nodig dan voor laravel-graphql. Er is eigenlijk maar 1 bestand onder de routes, schema.graphql, waar alles in gedefinieerd staat.



Figuur 9. Mapstructuur Lighthouse

In dit bestand staan de Queries, Mutations en Types. Deze staan hier allemaal als een type in. De queries verwijzen naar de types die onderaan het bestand gedefinieerd zijn, en naar een model in Laravel. In het begin van de query kunnen mogelijke arguments gedefinieerd worden, waarmee bijvoorbeeld een specifieke user of stad via de id of name opgevraagd kan worden. Dit bestand is geschreven in SDL (het heeft ook de .graphql extensie). SDL is een eigen taal, speciaal ontwikkeld voor GraphQL. Dit zorgt ervoor dat het schema zo kort en duidelijk mogelijk kan worden uitgedrukt en dus niet hoeft worden omgeschreven naar PHP classes.

```
type Query {
    users: [User!]! @paginate(type: "paginator" model: "App\\User")
    user(id: ID @eq): User @find(model: "App\\User")

    cities: [City!]! @paginate(type: "paginator" model: "App\\City")
    city(id: ID @eq, name: String @eq): City @find(model: "App\\City")

    countries: [Country!]! @paginate(type: "paginator" model: "App\\Country")
    country(id: ID @eq, name: String @eq): Country @find(model: "App\\Country")
}
```

Figuur 10. Queries in het schema

Lighthouse maakt heel goed gebruik van de Eloquent relationships die in Laravel aanwezig zijn. Dit heeft twee grote voordelen:

- Het maakt het GraphQL schema veel korter en duidelijker. Er hoeft geen functionaliteit in het schema te worden gezet zoals bij laravel-graphql wel moest in de resolve functie. In Lighthouse hoeft er in het type alleen maar een verwijzing naar een ander type te staan, met een relationship erbij. Deze relationship wordt dan automatisch uit het bijbehorende model gepakt.
- Het forceert de developer om de bestaande relationships in de Laravel models op een goede manier op te stellen. Dit voorkomt ook onnodige functionaliteit op andere plaatsen in de code.

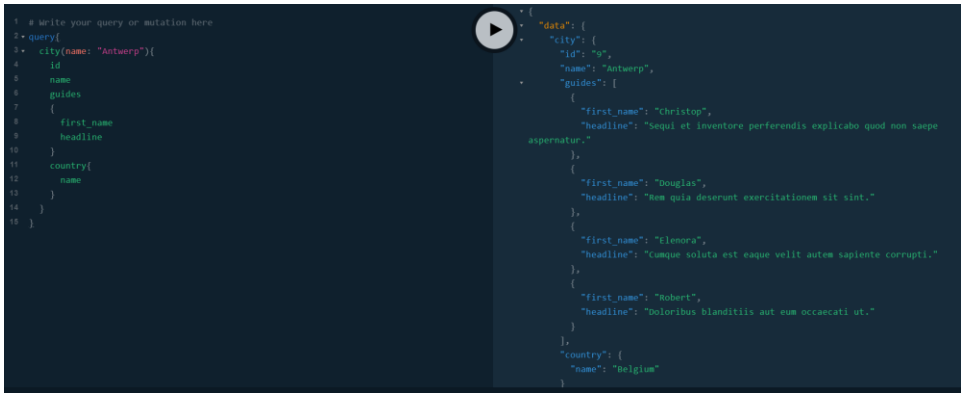
Figuur 11. Schema in Lighthouse

In de guides() functie in het city model stond een hasMany relationship naar de User, zonder de where clausule. Hierdoor returnde deze functie eigenlijk alle users die bij een bepaalde stad horen, niet alleen de gidsen. Waarschijnlijk wordt op een andere plaats in de code een functie gebruikt die vanuit de city->guides gaat filteren welke users daadwerkelijk een guide zijn. Doordat GraphQL forceert om deze relationships goed te definieren, werkt deze guides functie nu precies zoals hij hoort (hij returnt nu alleen de guides die bij de stad horen) en is er op andere plaatsen in de code geen extra filter meer nodig.

Figuur 12. Relationships in Laravel models

Met Lighthouse is het gelukt om binnen een uur al een aantal werkende GraphQL queries op te zetten en daarnaast de bestaande model code te verbeteren. Dit is een groot voordeel ten opzichte van laravel-graphql, want hier is ongeveer een hele dag voor nodig geweest. Dit had vooral te maken met het moeten aanpassen van meerdere losse bestanden en het moeten schrijven van functionaliteit in het schema van GraphQL zelf. Doordat Lighthouse gebruik maakt van de Laravel Eloquent relationships is dit allemaal niet nodig.

Het is ook met Lighthouse gelukt om een query te kunnen doen waarmee de gidsen van een bepaalde stad worden opgevraagd. Het resultaat is te zien in figuur 13.



Figuur 13. Query naar de cities met name argument in graphql-playground

Bijlage 3 - React Native implementaties van GraphQL

Om onze React Native frontend te koppelen aan onze GraphQL endpoint, is het nuttig om een client te hebben die GraphQL implementeert in de frontend. Het is natuurlijk mogelijk om met simpele AJAX calls queries te doen naar de endpoint, maar waarschijnlijk wordt dit op de lange termijn steeds complexer, naar mate er meer queries gedaan moeten worden.

De React Native app wordt gebouwd met behulp van Expo.

Er zijn 3 React GraphQL clients die voor Hi,hi Guide mogelijk van toepassing zijn: Apollo, Relay en Urql (Expo, z.d.). Graphql-request zou waarschijnlijk ook wel werken, maar waarschijnlijk veroorzaakt dit in de toekomst problemen, omdat deze geen goede integratie heeft met frontend frameworks en geen cache (Prisma, 2018).

Apollo

Voordelen

- Open source
- Veel gebruikt
- Makkelijk op te zetten
- Ingebouwde state management, waardoor Redux niet nodig is als de app complexer gaat worden
- Goede documentatie
- Vooral voor React gebouwd
- Werkt met elke implementatie van GraphQL

Nadelen

- Vrij grote bundle
- Doordat de ontwikkelingen snel gaan verandert er nog wel eens wat

Relay

Voordelen

- Specifiek voor React gebouwd
- Heel efficient
- Kleinere front-end bundle

Nadelen

- Errors moeten zelf via een type worden afgehandeld
- ID’s moeten uniek zijn voor alle types
- Soms onduidelijke documentatie
- Relay-compiler moet na elke aanpassing aan het GraphQL schema opnieuw runnen in de front-end

(Rozen, 2018)

Urql

Voordelen

- Flexibel
- Customisable

Nadelen

- Weinig documentatie
- Heel basic

(FormidableLabs, 2019)

Bijlage 4 – Git workflows

Om de code tussen developers onderling en de server te synchroniseren wordt gebruik gemaakt van git. Er bestaan meerdere manieren om git workflows logisch in te delen. Deze hebben elk voordelen en nadelen:

Git Flow

Deze flow heeft de volgende branch structuur:

Master branch – production code

Develop branch – pre-production code

Supporting branches

Feature branch – nieuwe features

Hotfix branch – snelle fixes op de master

Release branch – voorbereiding van een nieuwe production release

Voordelen

- Schone staat van branches op elk moment
- Makkelijk te begrijpen
- Heeft uitbreidingen en support op de meeste git tools
- Ideaal wanneer er meerdere versies in production gebruikt worden

Nadelen

- Git history wordt onleesbaar
- Continuous integration kan lastiger worden door splitsing van master en develop

GitHub Flow

1. Alles in master is deployable
2. Nieuwe code krijgt altijd een branch vanaf master
3. Commits worden lokaal gedaan en regelmatig gepusht naar de server
4. Er wordt een pull request gedaan voor een merge
5. Iemand reviewt de code voordat deze in master wordt gemerged
6. Als de merge met master is gedaan wordt de code meteen gedeployed

Voordelen

- Werkt goed met continuous integration
- Simpeler alternatief dan Git Flow
- Ideaal als er maar 1 versie in production moet worden gehouden

Nadelen

- Production code kan snel instabiel worden
- Releases kunnen lastig gaan in verschillende environments

GitLab Flow

1. Er worden geen directe commits op master gedaan

2. Alle commits worden getest
3. Code review wordt gedaan voordat de merge met master wordt gedaan
4. Deployment is automatisch
5. User gebruikt tags
6. Releases zijn gebaseerd op tags
7. Gepushte commits worden nooit gerebased
8. Iedereen start vanaf master
9. Bugs in master worden eerst gefixt
10. Commit messages laten zien wat de bedoeling van de commit is

Voordelen

- Schonere git history
- Ideaal wanneer er 1 versie in production is

Nadelen

- Complexer dan GitHub Flow
- Complexer dan Git Flow wanneer er meerdere versies in production zijn

One Flow

Het verschil tussen Git Flow en One Flow is dat er geen develop branch is

Voordelen

- Git history is schoner en leesbaarder
- Flexibel
- Ideaal wanneer er 1 versie in production moet zijn

Nadelen

- Niet handig in combinatie met continuous integration

(Porto, 2018)

Bijlage 5 - Bash script voor automatische installatie

Bij het testen van het live zetten van de site bleken er veel commands te zijn die elke keer weer opnieuw gedaan moeten worden in de terminal. Deze zijn in een bash script gezet zodat ze automatisch draaien en dit dus geen tijd meer kost. Het script ziet er als volgt uit:

composer install

installeert alle dependencies

php artisan config:cache

ververst de cache van de configuratiebestanden van Laravel

php artisan reset:migrations

reset de migration history

php artisan migrate

draait de migrations

php artisan db:seed --class=QuestionsTableSeeder

seed de database met questions

php artisan db:seed --class=AboutTourTableSeeder

seed de database met de about-tour items

php artisan passport:install

installeert passport clients in de database

php artisan storage:link

koppelt de Laravel file storage map met een symlink aan de public map

php artisan optimize

ververst alle caches van Laravel, waaronder routes

Bijlage 6 – Stappenplan live zetten nieuw platform

Backups maken

Database backup maken

Kopie van hihinew map op server maken

hihinew map downloaden van server

Production klaarzetten

Laatste pushes naar testing brache

Production brache vanaf testing branche maken

Production map op server maken

Git pull in production map

Database gegevens in .env zetten. (.env van live en .env van testing even handmatig mergen)

composer install

php artisan config:cache

php artisan storage:link

Huidige foto's naar juiste mapjes verplaatsen
public/uploads/300x300 -> public/storage/profilepics/300x300
(Mocht storage:link niet werken, verwijder dan public/storage en probeer opnieuw)

Live site suspenden

php artisan reset:migrations

php artisan migrate

php artisan db:seed --class=QuestionsTableSeeder

php artisan db:seed --class=AboutTourTableSeeder

php artisan passport:install

php artisan optimize

Password grant client id en secret uit oauth_clients tabel kopiëren naar .env file

Symlink in Plesk switchen

PHP settings hihiguide.com goed zetten (16M)

Gzip aanzetten op hihiguide.com

Checken of cities deleted active=0 is

Live site unsuspenden

Facebook login/register testen

Rest van site testen

- Tourist profiel aanmaken
- Guide profiel aanmaken
- Mails (bij het aanmaken)
- Boekingsproces doorlopen (ook betaling testen)
- Foto upload (ook telefoon)
- Video upload (ook telefoon)
- Linkjes in mail
- Debug = false
- Checken of GraphQL uit staat
- Checken of tracking goed wordt opgeslagen
- Alle statussen aanpassen

Alle statussen expired zetten:

```
UPDATE booking SET status = 'expired';
```

Juiste statussen open zetten:

```
UPDATE booking SET status = 'open' WHERE date > '2019-03-29';
```

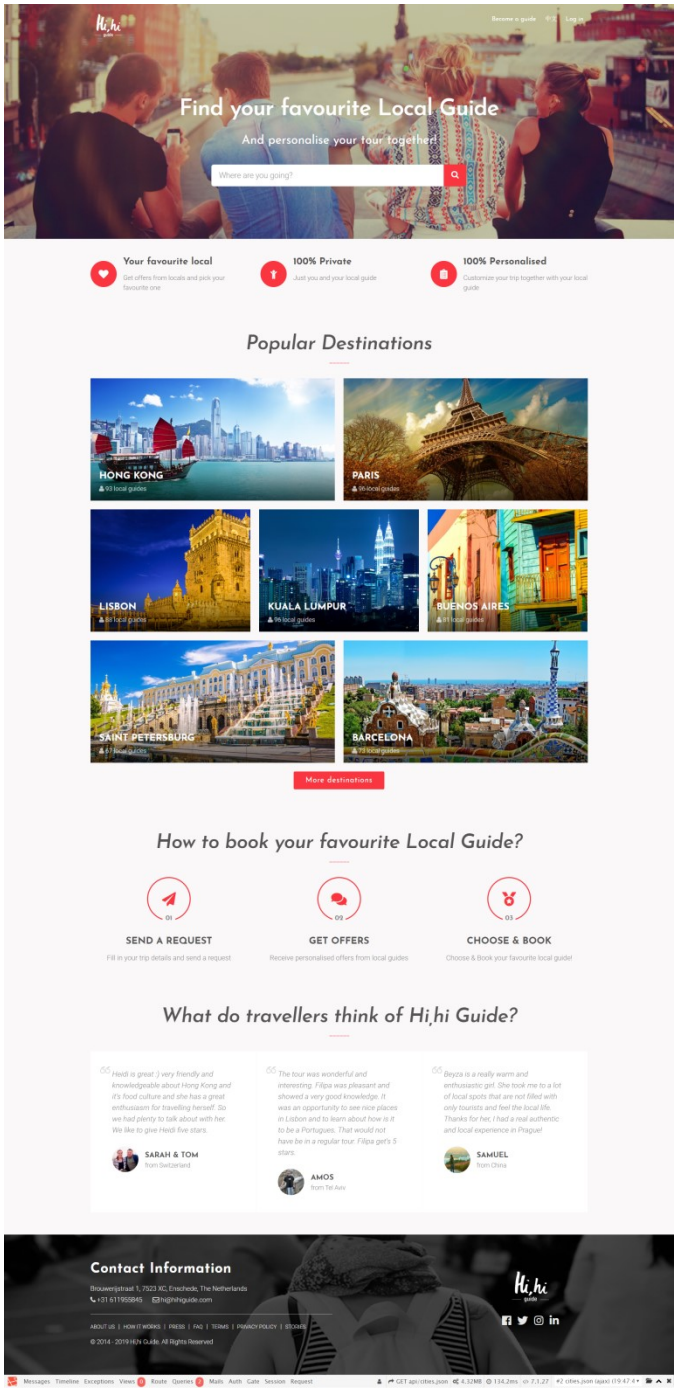
Juiste statussen completed zetten:

```
UPDATE booking SET status = 'completed' WHERE completed = 1;
```

Leaveareview status handmatig zetten adhv review en completed columns

Booking price handmatig zetten

Bijlage 7 - Gerealiseerde features

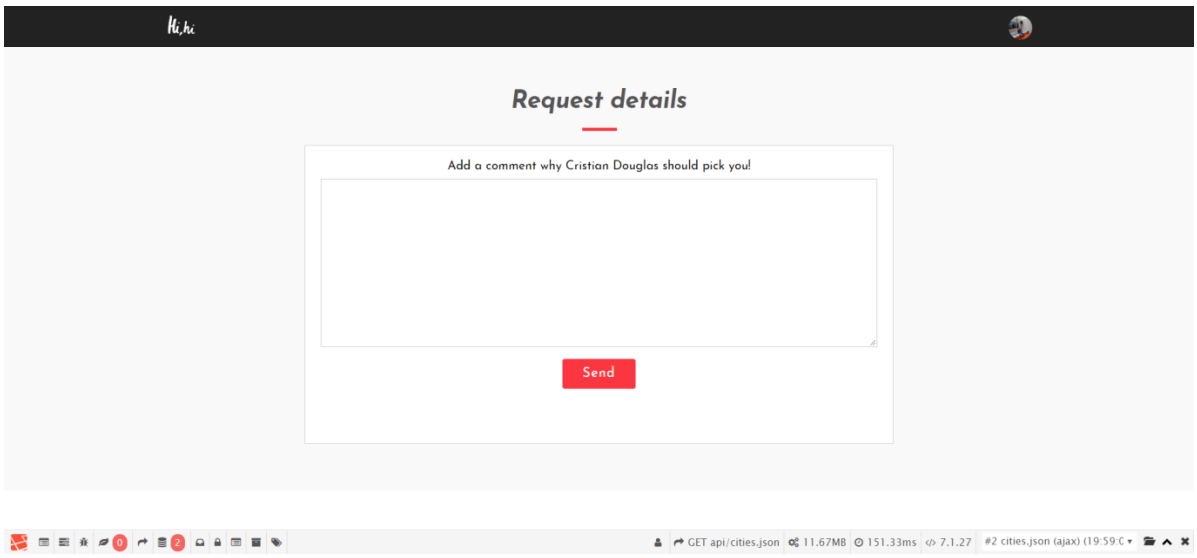


Figuur 1. Het vernieuwde Hi,hi Guide

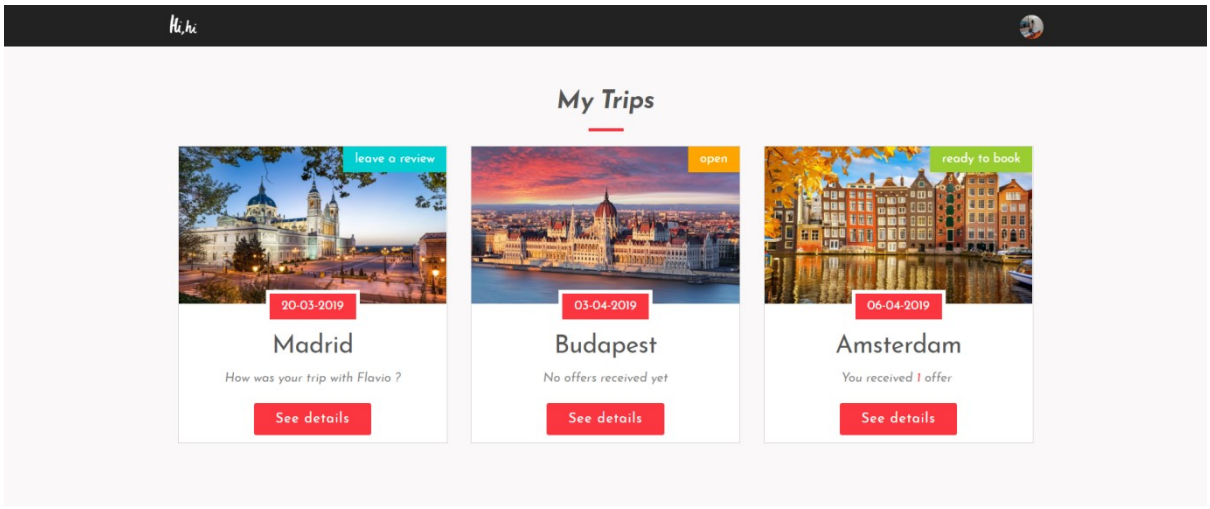
Na het verminderen van de technical debt is er een vernieuwd Hi,hi Guide platform gelanceerd (figuur 1). Dit is online te bezoeken op https://www.hihiguide.com. Het platform heeft dankzij Steven van Eck een volledige visuele make-over gekregen. Het ziet er nu een stuk professioneler uit. De portal waarin boekingen worden beheert door gidsen en toeristen is verbouwd, zodat deze veel logischer is voor de gebruikers. De site is sneller gemaakt, door middel van technieken als lazy loading, compressie van de afbeeldingen en Gzip. Ook zijn er een aantal nieuwe features toegevoegd, waarvan de belangrijkste hieronder genoemd zijn.

Meer interactie vanuit gids door reactie-mogelijkheid

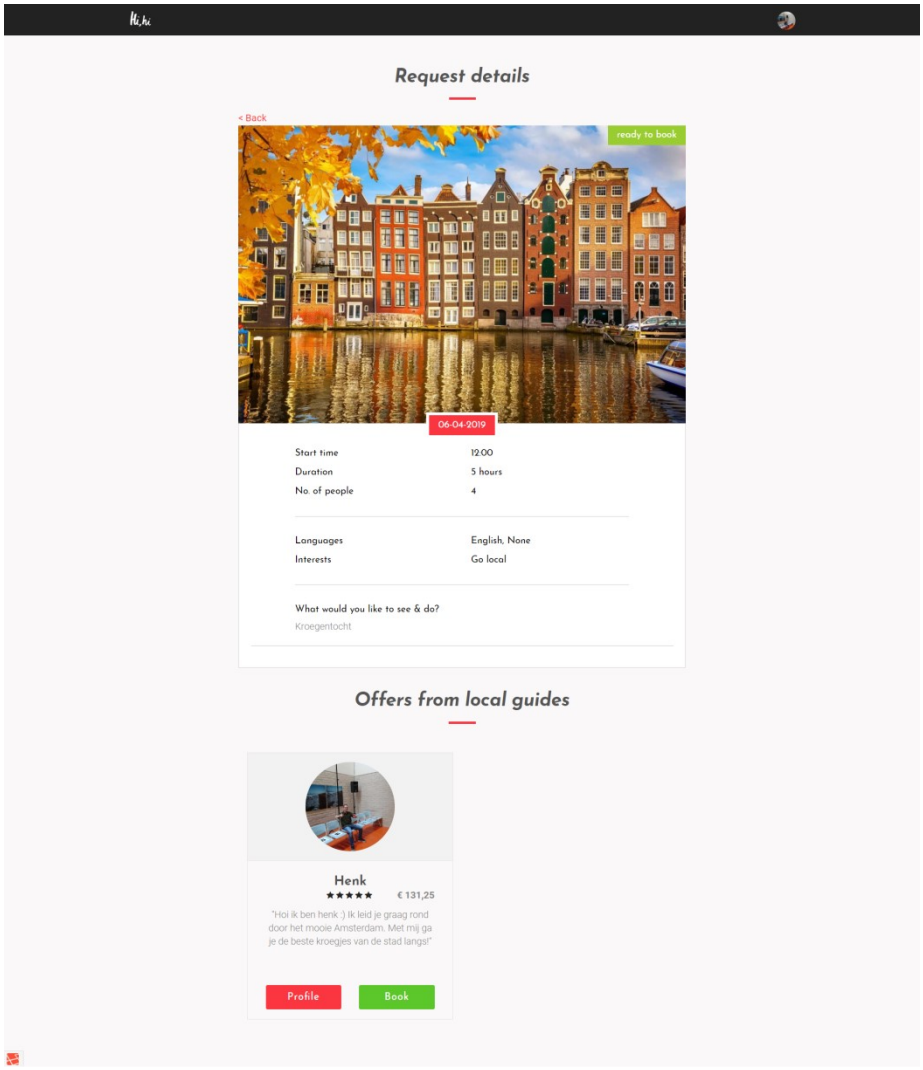
Vanuit een brainstorm met Hi,hi Guide kwam het idee om een gids een comment te laten plaatsen wanneer hij een verzoek accepteert, om de interactie tussen toerist en gids te vergroten. Steven (frontend developer) heeft onder de gidsen een enquête gehouden over de user experience. Hieruit kwam ook naar voren dat gidsen graag een bericht als reactie willen kunnen sturen op een verzoek. Deze functionaliteit is dus toegevoegd aan de frontend en backend, zodat gidsen nu bij het accepteren van een verzoek een inputfield krijgen waarin ze een reactie kunnen typen (figuur 2). Deze verschijnt dan vervolgens bij de toerist in het portal op de plaats waar hij een gids kan selecteren voor zijn tour (figuur 3 en 4).



Figuur 2. Comment field waarin een gids kan reageren op een verzoek



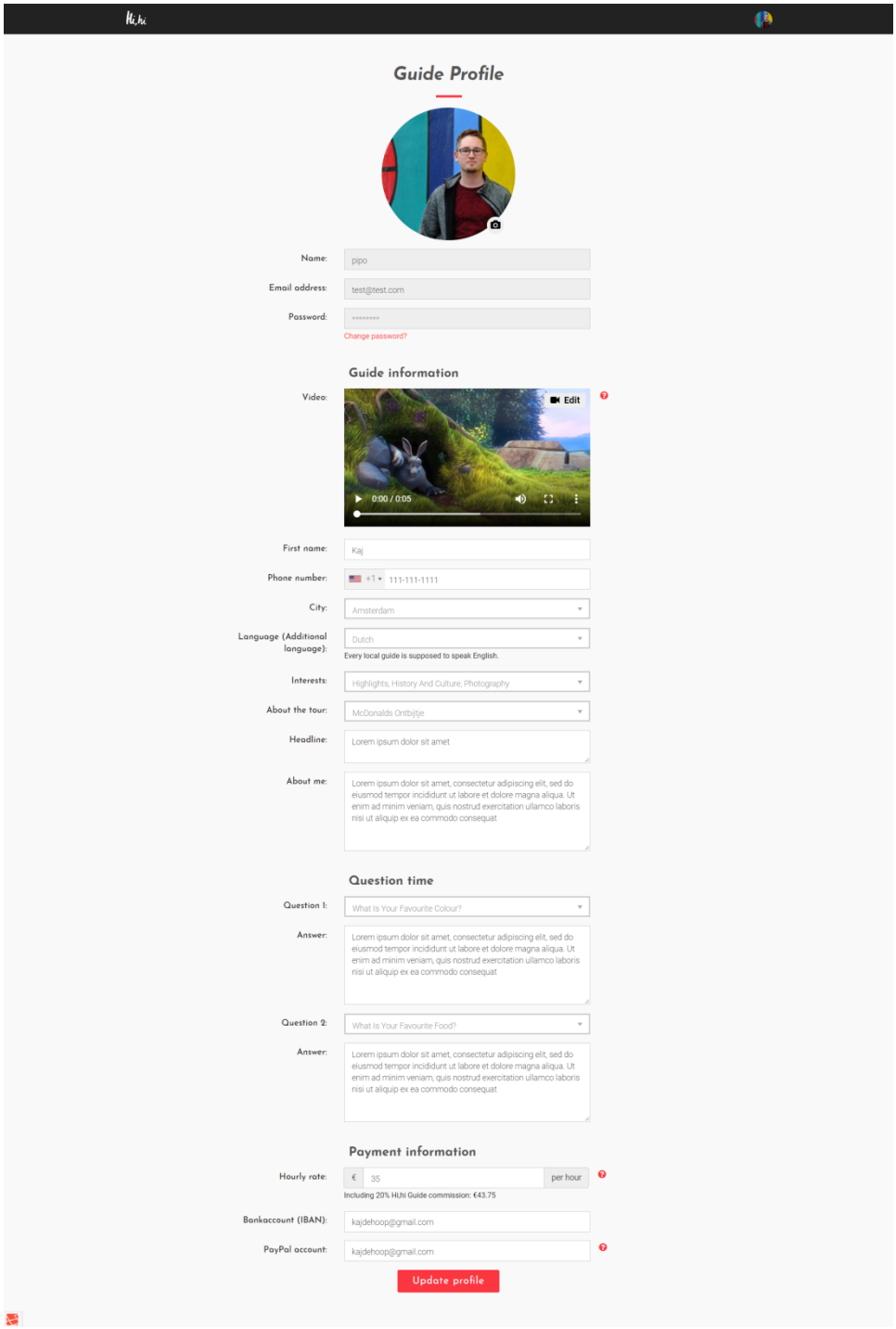
Figuur 3. Het portal waarin een toerist kan zien welke verzoeken hij heeft verstuurd en of hier al op gereageerd is door gidsen



Figuur 4. De toerist ziet de comment van de gids bij de offers staan

Video uploader voor gidsen

Om het persoonlijke aspect van een gidsprofiel te versterken is er de mogelijkheid toegevoegd om video's te uploaden en bekijken (figuur 5). Een gids kan nu in zijn profiel een video van zichzelf uploaden, waarin hij iets vertelt over zichzelf of zijn stad. Deze video verschijnt vervolgens op zijn profiel, zodat toeristen een persoonlijker beeld krijgen van de gids. Voor het opslaan van de video's is gebruik gemaakt van Laravel Storage. Dit is een file storage die gelinkt is aan de publiek toegankelijke website, maar beschermd kan worden tegen benaderen van buitenaf.



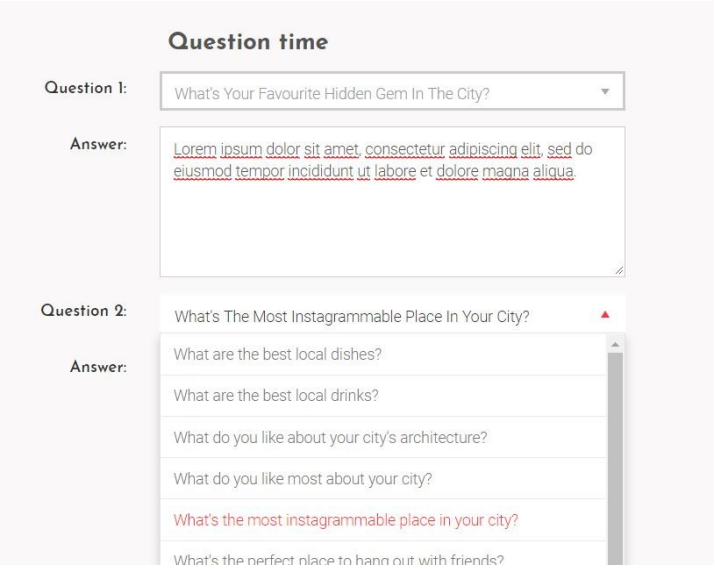
Figuur 5. In het profiel in de portal kan een gids nu een persoonlijke video toevoegen

Gidsprofiel en toeristprofiel ondergebracht onder één type profiel

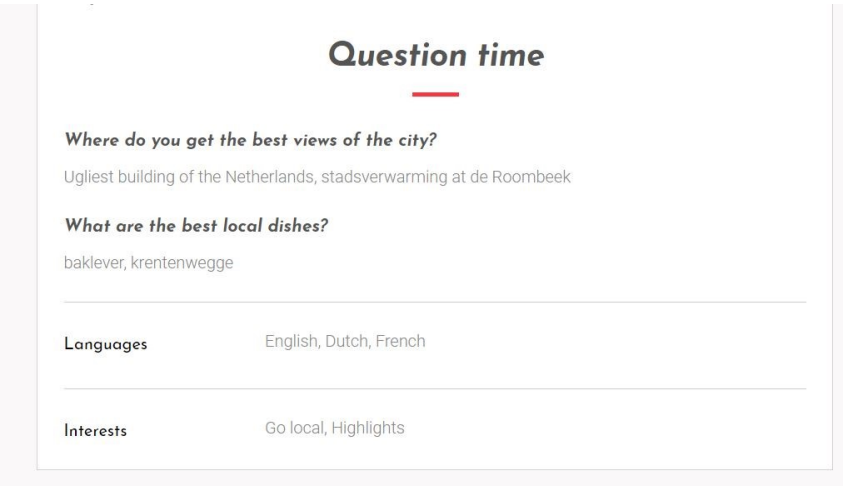
Het gebruikersprofiel van gidsen is gecombineerd met het gebruikersprofiel van toeristen. Elke gebruiker is nu in de basis een toerist (dit is het basisprofiel), en elke toerist kan upgraden naar een gidsprofiel. Dit maakt het mogelijk dat gidsen ook als toerist op pad gaan als ze in een andere stad zijn en met hun eigen profiel een boeking kunnen doen. Waarschijnlijk versterkt dit de Hi,hi community, omdat een gids gemakkelijk een andere keer als toerist op reis kan gaan, en andersom. Het is nog steeds mogelijk om direct vanuit de homepage een gidsprofiel aan te maken.

Persoonlijkere gidsprofielen door middel van vragen en antwoorden

Om de profielen van gidsen interessanter te maken is het tekstvak waarin ze iets over henzelf konden vertellen weggehaald en zijn hier twee menu's met vragen voor in de plaats gekomen. De gids kan een keuze maken uit een aantal vragen. Deze zijn gemaakt om hem op een persoonlijke manier wat over zijn stad te laten vertellen. Een gids kan twee vragen kiezen, waardoor elke gids weer andere vragen beantwoord en elk profiel zo unieker is (figuur 6 en 7).



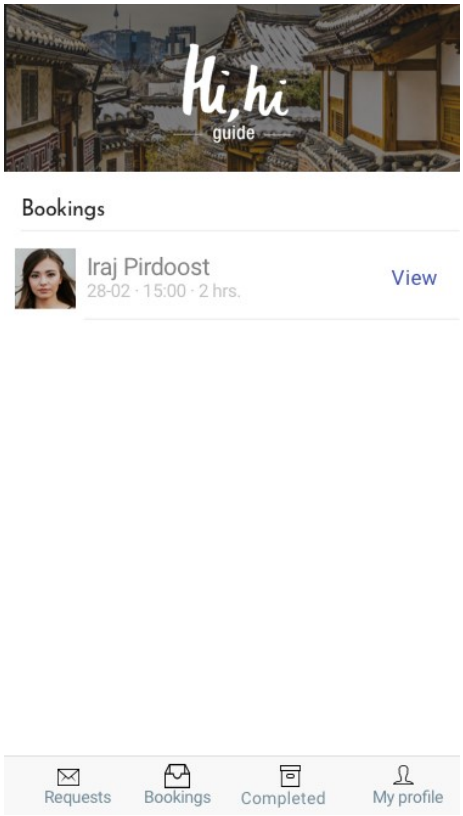
Figuur 6. De vragen en antwoorden die zijn in te vullen in het portal



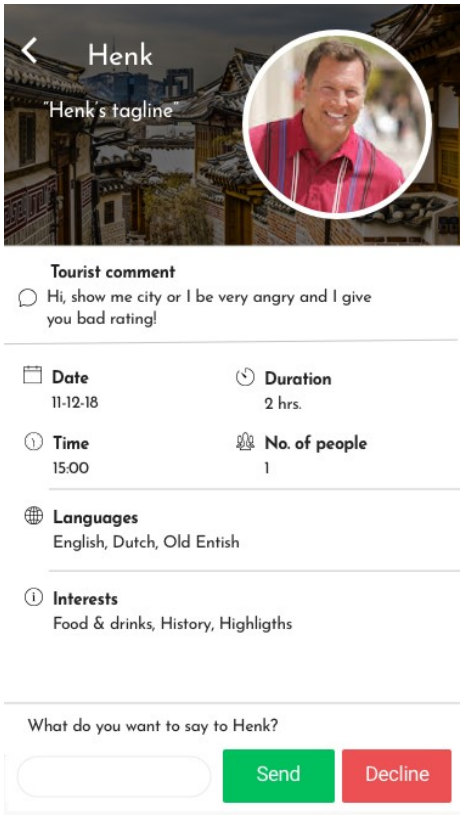
Figuur 7. De vragen en antwoorden zoals te zien op een gidsprofiel

Mobiele app

Voor gidsen is een mobiele app ontwikkeld. Deze app is gebouwd in React Native. Met de app kunnen gidsen hun verzoeken en boekingen beheren (figuur 8 en 9). Ook worden ze via notificaties op de hoogte gehouden van verzoeken en boekingen.



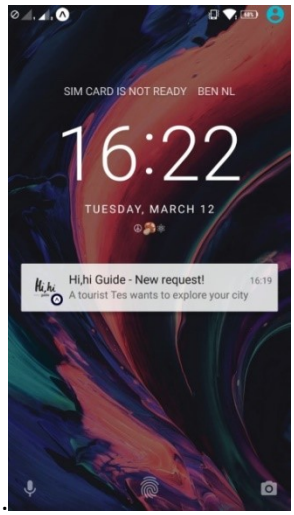
Figuur 8. Boekingen in de app



Figuur 9. Verzoek dat geaccepteerd kan worden in de app

De mobiele app haalt door middel van Apollo Client data op uit de GraphQL API die gerealiseerd is in de backend. Alles werkt dus vanaf hetzelfde centrale systeem.

Wanneer er een nieuw verzoek van een toerist binnenkomt of een gids geboekt wordt, verschijnt er direct een notificatie op zijn telefoon (figuur 10). Wanneer hij deze aanklikt kan hij meteen het verzoek of de boeking bekijken. In het geval van een verzoek kan deze ook vanuit de app direct geaccepteerd worden. Het versturen van notificaties gebeurt vanuit Laravel via de Expo Push API. De notificaties worden vervolgens in React Native afgevangen.



Figuur 10. Een binnenkomende notificatie van de app

Bijlage 8 - Concept nieuw systeem voor boekingen beheren in backend

Om de boekingen in de backend beter in te delen, kunnen deze het best gesplitst worden in requests, bookings en trips. Een request ontstaat op het moment dat een toerist een verzoek indient. Deze bevat informatie over de request en de status van de request en is gekoppeld aan een trip. Een trip bevat informatie over de afspraak zelf, dus bijvoorbeeld de duratie, tijd, datum, gewente talen en activiteiten en het aantal mensen. Wanneer op een request een gids wordt geboekt, verandert deze is een booking. Op dat moment wordt de trip gekoppeld aan deze booking. De booking bevat zelf dus alleen maar informatie over de boeking en niet over de activiteit die gekoppeld is. Door dit los van elkaar te houden wordt voorkomen dat er dingen dubbel worden opgeslagen. Hieronder zijn de aanpassingen aan de database en het data model uitgedacht. De benodigde aanpassingen aan de Laravel code moeten nog worden uitgewerkt.

Database

Table: Bookings

Columns:

- Id
- Trip_id -> trips
- Client_id -> users
- Guide_id -> users
- Status
- Price_paid
- Booking_code
- Created_at
- Updated_at

Table: Trips

Columns:

- Id
- Duration
- Nr_people
- Start_time
- Date
- City_id -> cities
- Language_id -> languages
- Activity_id -> activities
- Created_at
- Updated_at

Table: Requests

Columns:

- Id

- Trip_id -> trips
- Client_id -> users
- Guide_id -> users
- Status
- Guide_comment
- Created_at
- Updated_at

Laravel models

Booking

Relations:

- hasOne Trip
- belongsTo User (client)
- belongsTo User (guide)

Request

Relations:

- hasOne Trip
- belongsTo User (client)
- belongsTo User (guide)

Trip

Relations:

- belongsTo City
- belongsTo Language
- belongsTo Activity

User

Relations:

- hasMany Booking (outgoing, client_id = user_id)
- hasMany Booking (incoming, guide_id = user_id)
- hasMany Trip
- hasMany Request (outgoing, client_id = user_id)
- hasMany Request (incoming, guide_id = user_id)

Aanpassingen aan de Laravel code

Blade verwijzingen

Ipv $booking->duration wordt het bijvoorbeeld $booking->trip->duration

Ipv $booking->city wordt het bijvoorbeeld $request->trip->city

Boekingsstatus

De status van booking en guide_bookings moeten op een andere manier worden gebruikt.