

[UCLL reisgids]

Project dossier Backend

Dit project dossier bevat alle informatie van het UCLL REISGIDS project.

Project omschrijving, requirements analyse, systeem analyse, programma code, documentatie.

Project dossier

[UCLL reisgids]

Inhoud

[AFbeeldingenlijst 3](#_Toc11829591)

[2 Project omschrijving 4](#_Toc11829592)

[3 Requirements analyse 4](#_Toc11829593)

[4 Wireframes 10](#_Toc11829594)

[5 Software development analyse 14](#_Toc11829595)

[5.1 Klasse diagram 14](#_Toc11829596)

[5.2 Sequentie diagram 18](#_Toc11829597)

[6 Programma code 23](#_Toc11829598)

[6.1 Watervalsysteem 23](#_Toc11829599)

[6.2 Laravel 23](#_Toc11829600)

[6.3 Opbouw van de code 24](#_Toc11829601)

[6.4 Controllers 24](#_Toc11829602)

[6.5 Models 25](#_Toc11829603)

[6.6 Interfaces 25](#_Toc11829604)

[6.7 Eloquent 26](#_Toc11829605)

[6.8 Views 27](#_Toc11829606)

[6.9 Migrations 28](#_Toc11829607)

[6.10 Seeders 29](#_Toc11829608)

[6.11 Opmaak 30](#_Toc11829609)

[7 Besluit 30](#_Toc11829610)

# AFbeeldingenlijst

[Figuur 1: use-case diagram 4](#_Toc11829400)

[Figuur 2: Wireframe planning 11](#_Toc11829401)

[Figuur 3: wireframe planning pop-up 11](#_Toc11829402)

[Figuur 4: Wireframe algemene informatie 12](#_Toc11829403)

[Figuur 5: wireframe hotel 12](#_Toc11829404)

[Figuur 6: wireframe vervoersinformatie 13](#_Toc11829405)

[Figuur 7: Klasse diagram planning 14](#_Toc11829406)

[Figuur 8: klasse diagram Algemene informatie 15](#_Toc11829407)

[Figuur 9: klasse diagram hotel 16](#_Toc11829408)

[Figuur 10: klasse diagram vervoer 17](#_Toc11829409)

[Figuur 11: sequentie diagram planning deel 1 18](#_Toc11829410)

[Figuur 12: sequentie diagram planning deel 2 19](#_Toc11829411)

[Figuur 13: sequentie diagram algemene informatie 20](#_Toc11829412)

[Figuur 14: sequentie diagram hotel 21](#_Toc11829413)

[Figuur 15: sequentie diagram vervoer 22](#_Toc11829414)

# Project omschrijving

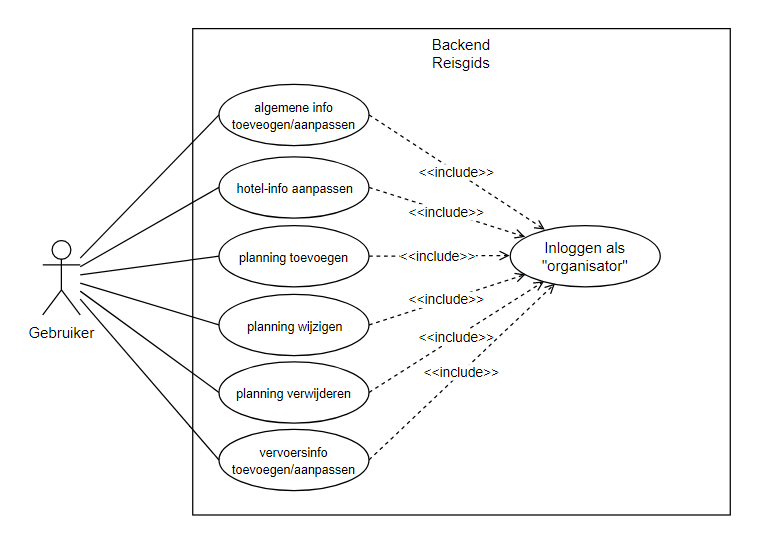
Voor de buitenlandse reis van de studenten technologie wordt er steeds een reisbrochure gemaakt die heel wat waardevolle informatie bevat gaande van contactgegevens, geplogenheden en regels van het gastland, kamer en auto verdelingen, het reisprogramma van dag tot dag …

Het is de bedoeling om deze papieren reisgids te vervangen door een online variant.

Wij hebben een backend voorzien, waarop een reisbegeleider gemakkelijk de benodigde informatie (die in de online reisgids wordt getoond) kan toevoegen of wijzigen.

# Requirements analyse

**Use-case diagram**



Figuur 1: use-case diagram

****

****





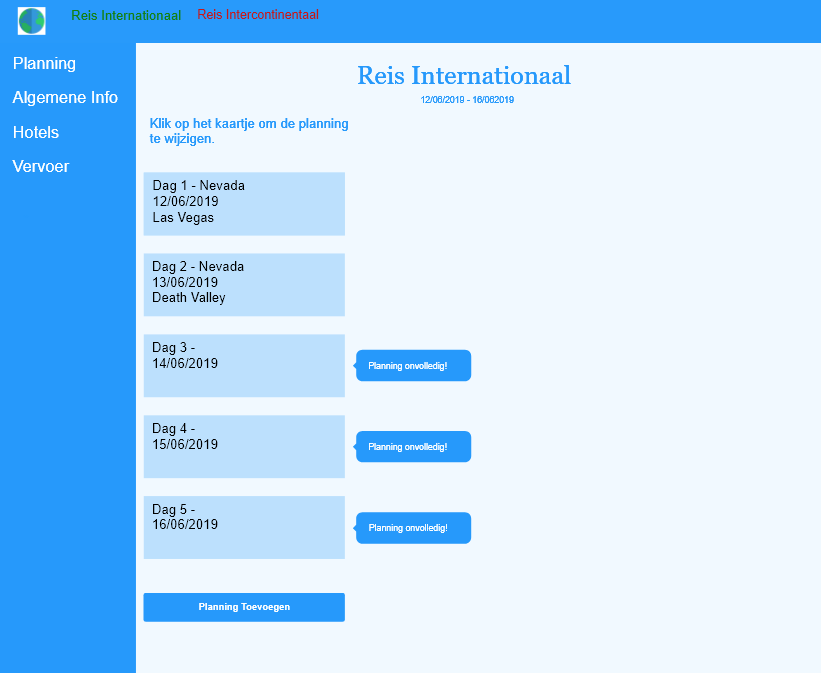




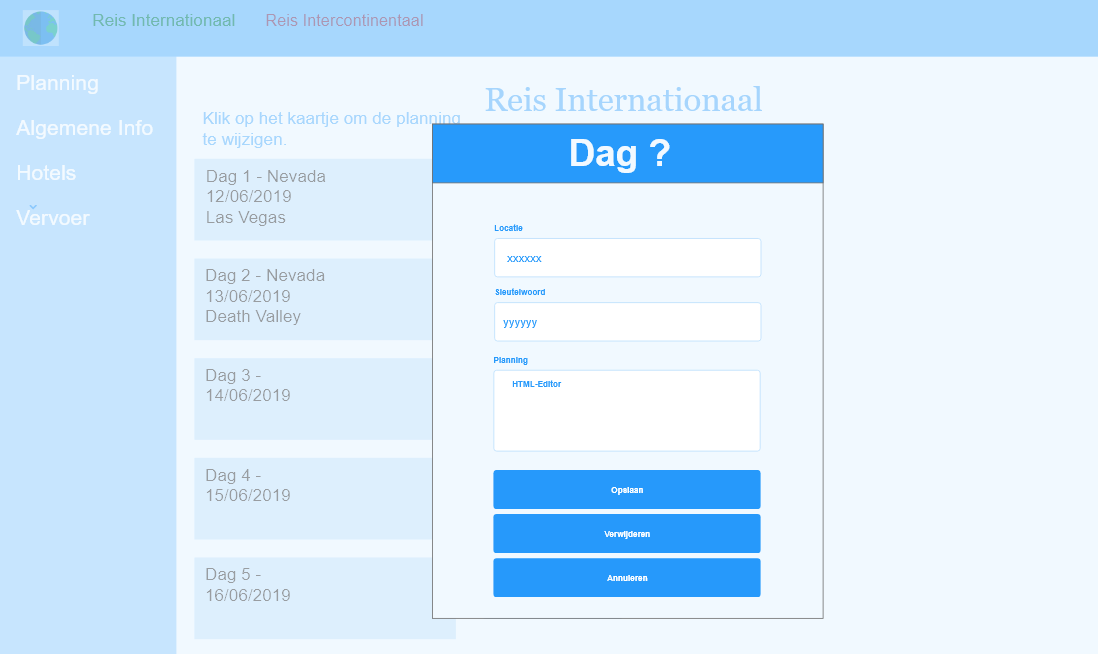
# Wireframes

Om de klant een beeld te geven van de lay-out en werking van onze site, voor we beginnen coderen, hebben we van iedere pagina een wireframe gemaakt. Hieronder is telkens de desktop wireframe weergegeven.

**Planning**

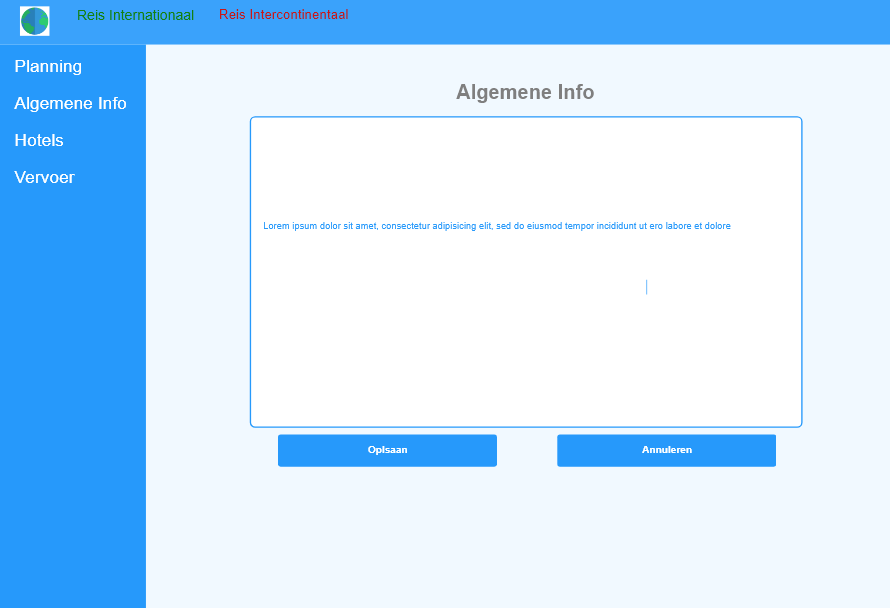


Figuur 2: Wireframe planning



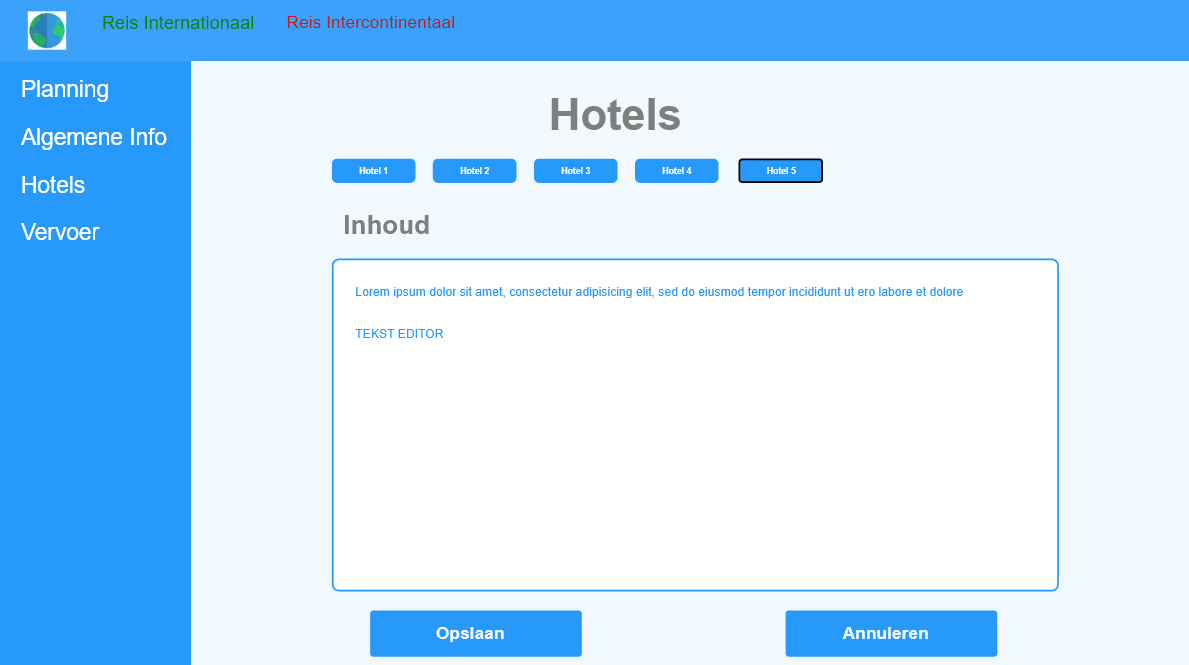
Figuur 3: wireframe planning pop-up

**Algemene informatie**



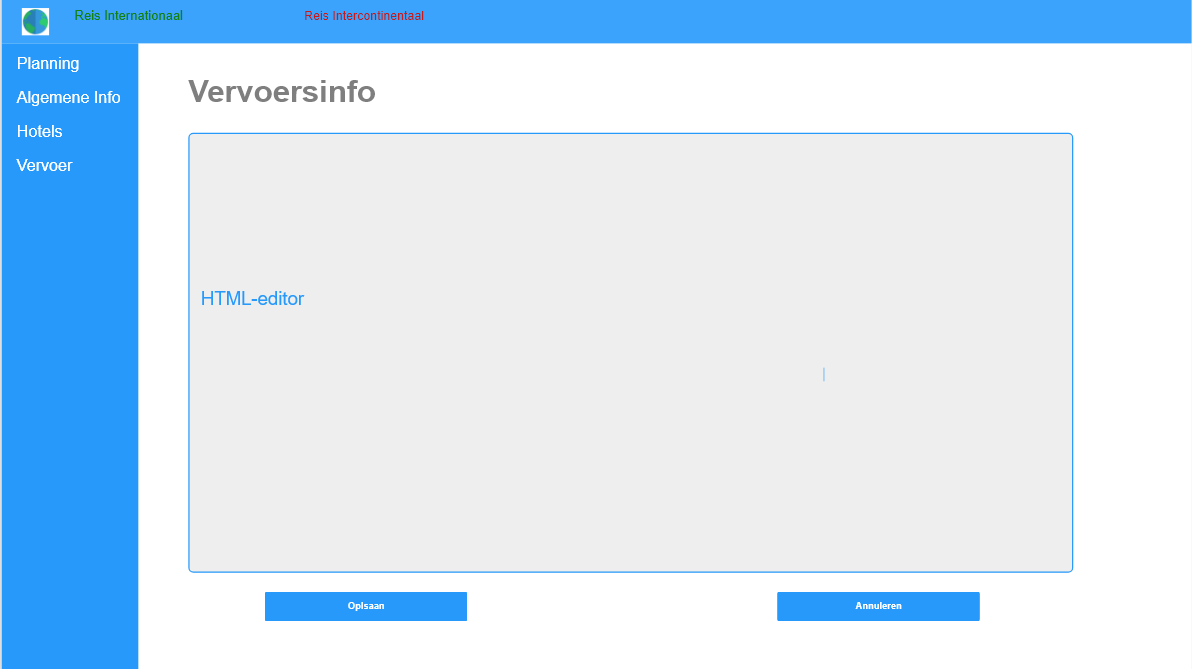
Figuur 4: Wireframe algemene informatie

**Hotel**



Figuur 5: wireframe hotel

**Vervoersinformatie**

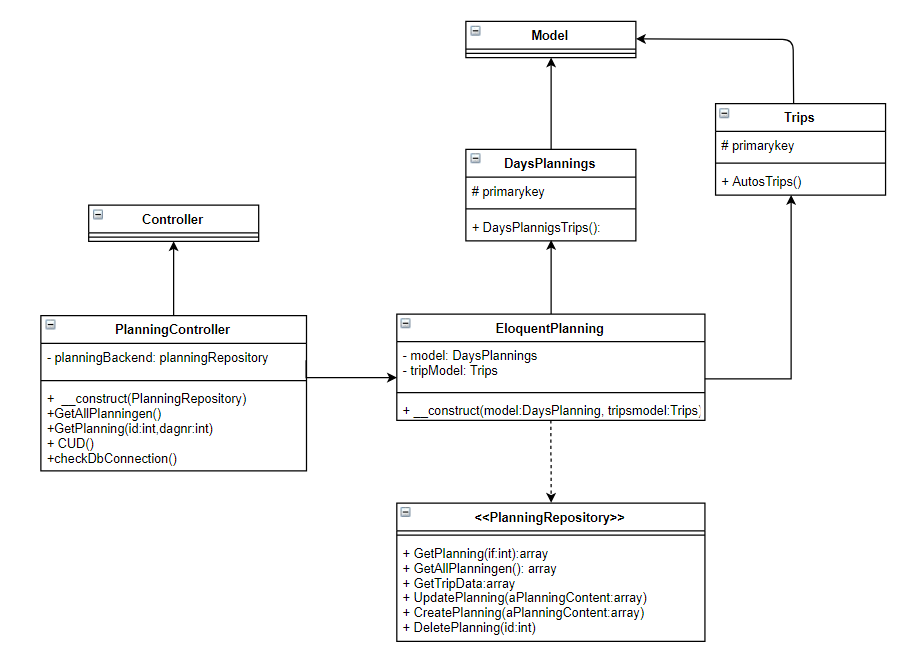


Figuur 6: wireframe vervoersinformatie

# Software development analyse

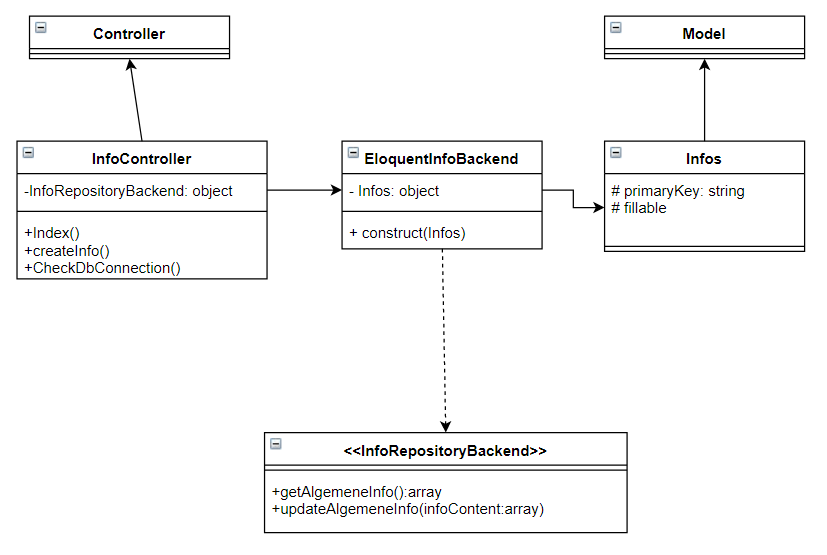
## Klasse diagram

**Planning**



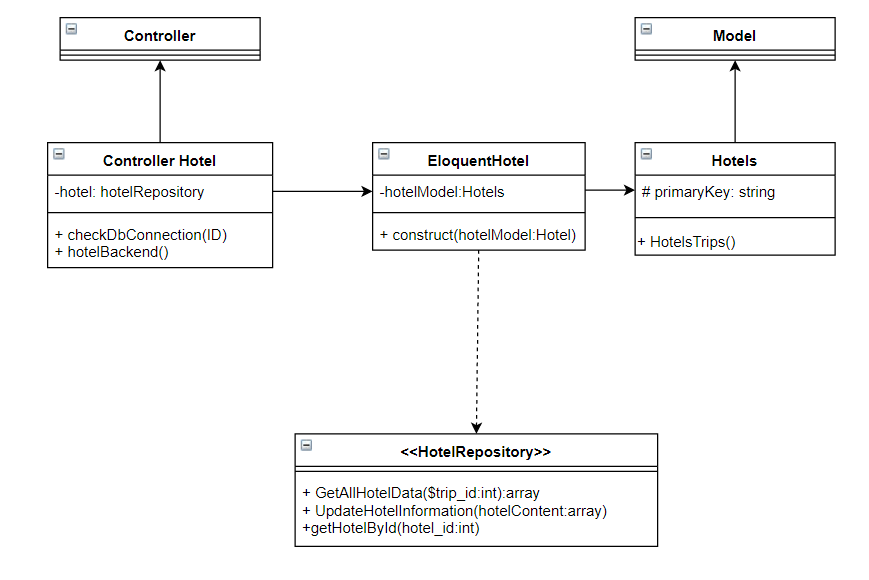
Figuur 7: Klasse diagram planning

**Algemene informatie**



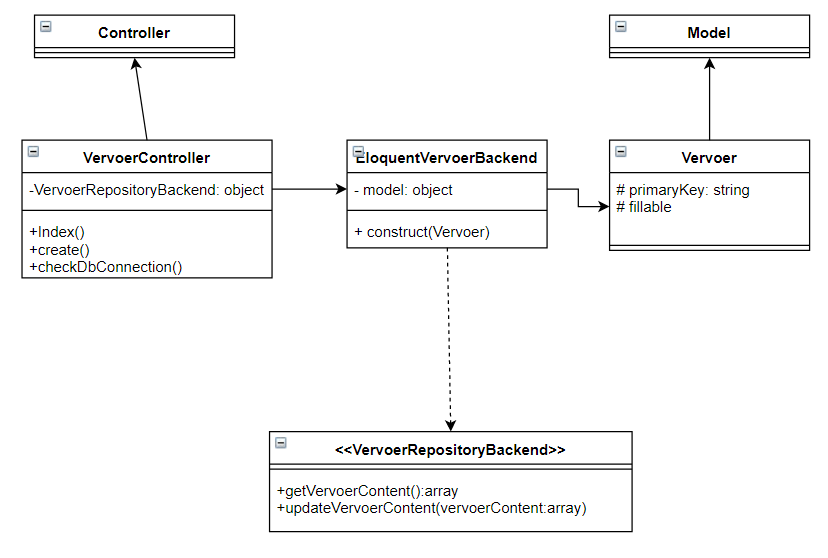
Figuur 8: klasse diagram Algemene informatie

**Hotel**



Figuur 9: klasse diagram hotel

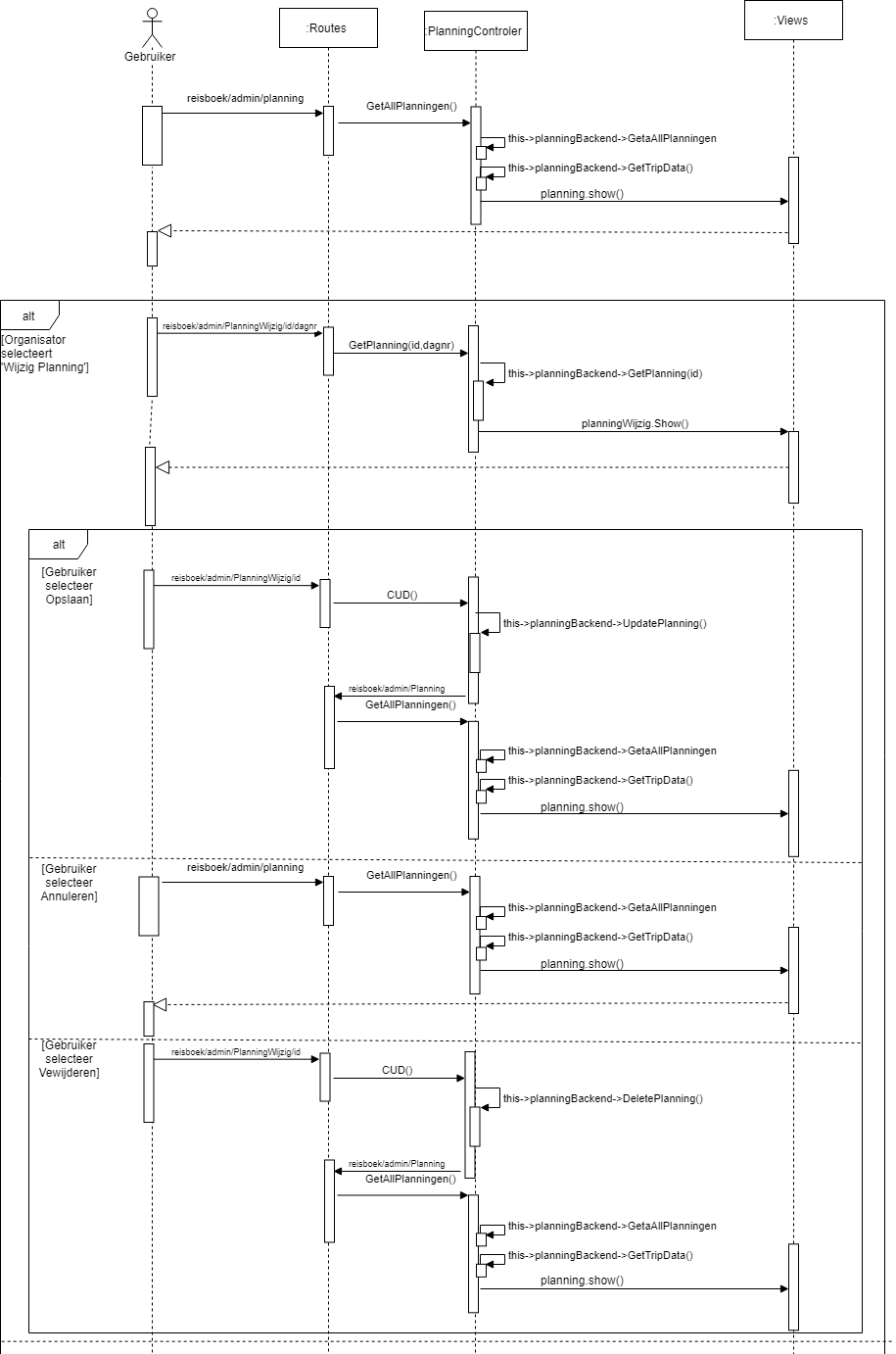
**Vervoersinformatie**



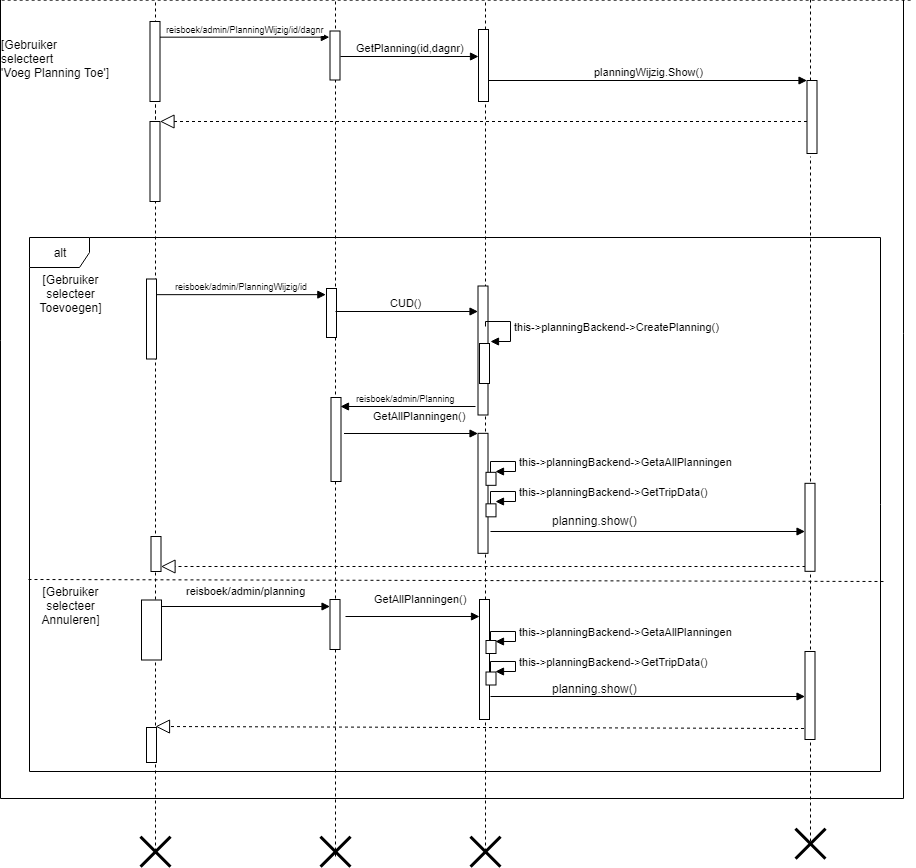
Figuur 10: klasse diagram vervoer

## Sequentie diagram

**Planning**

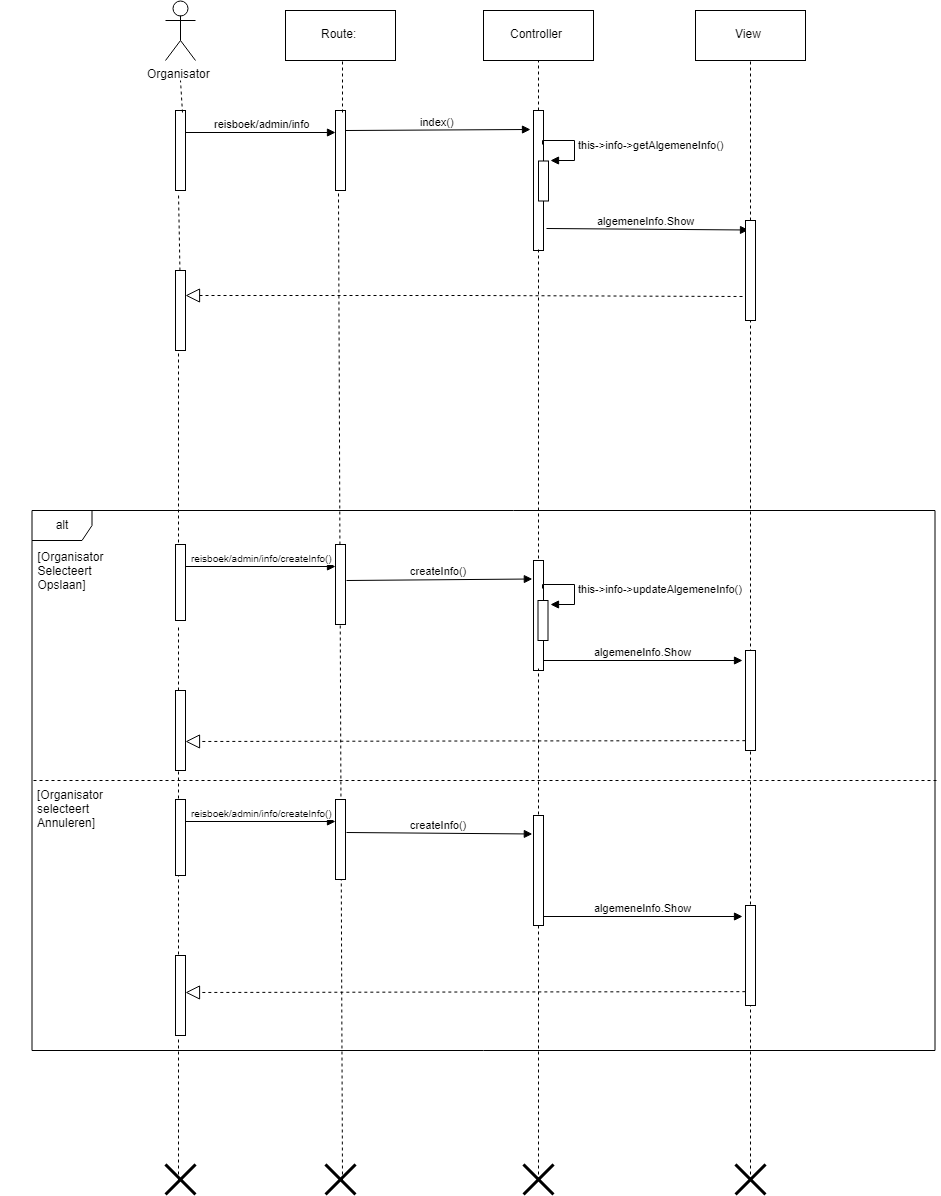


Figuur 11: sequentie diagram planning deel 1



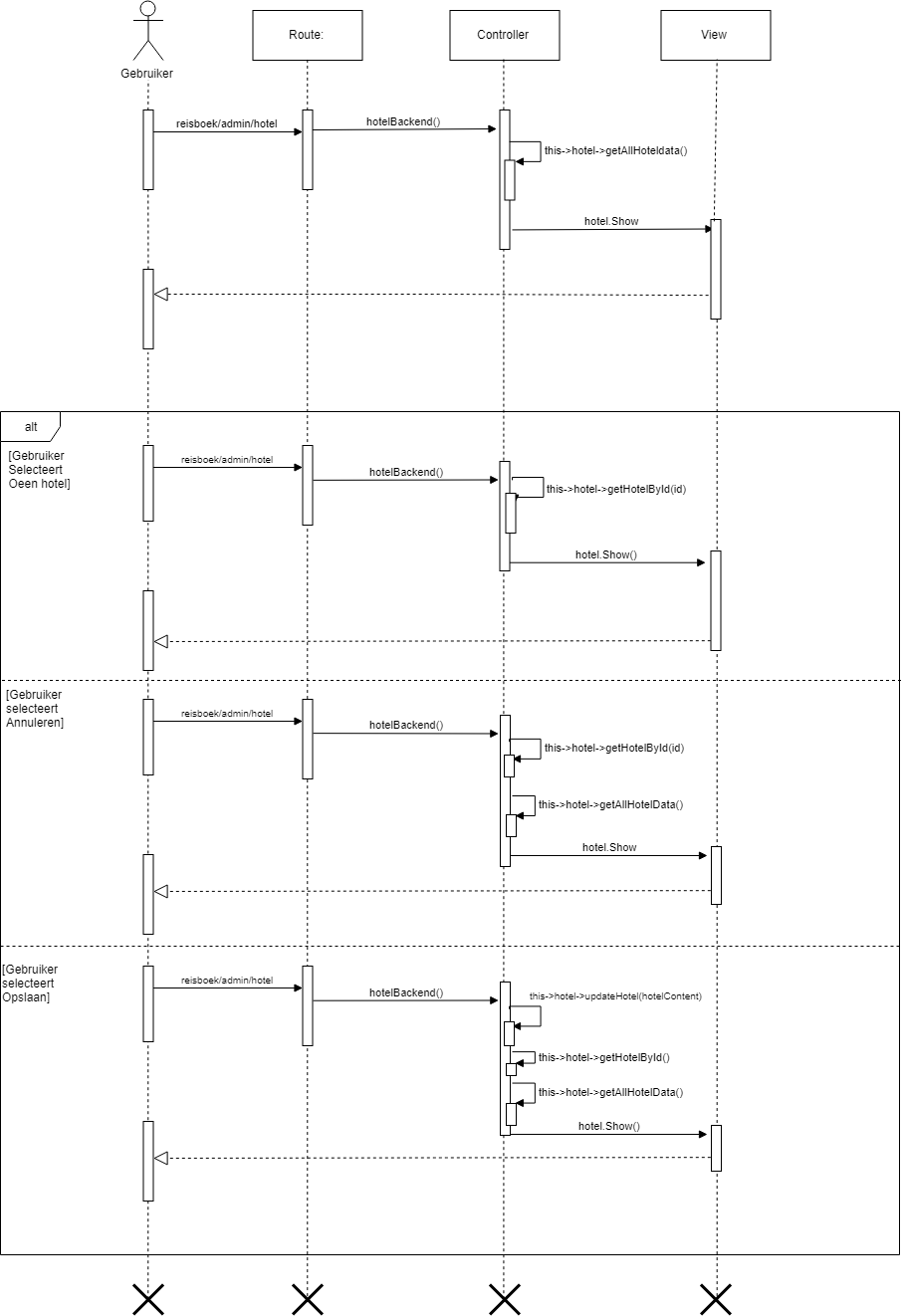
Figuur 12: sequentie diagram planning deel 2

**Algemene informatie**



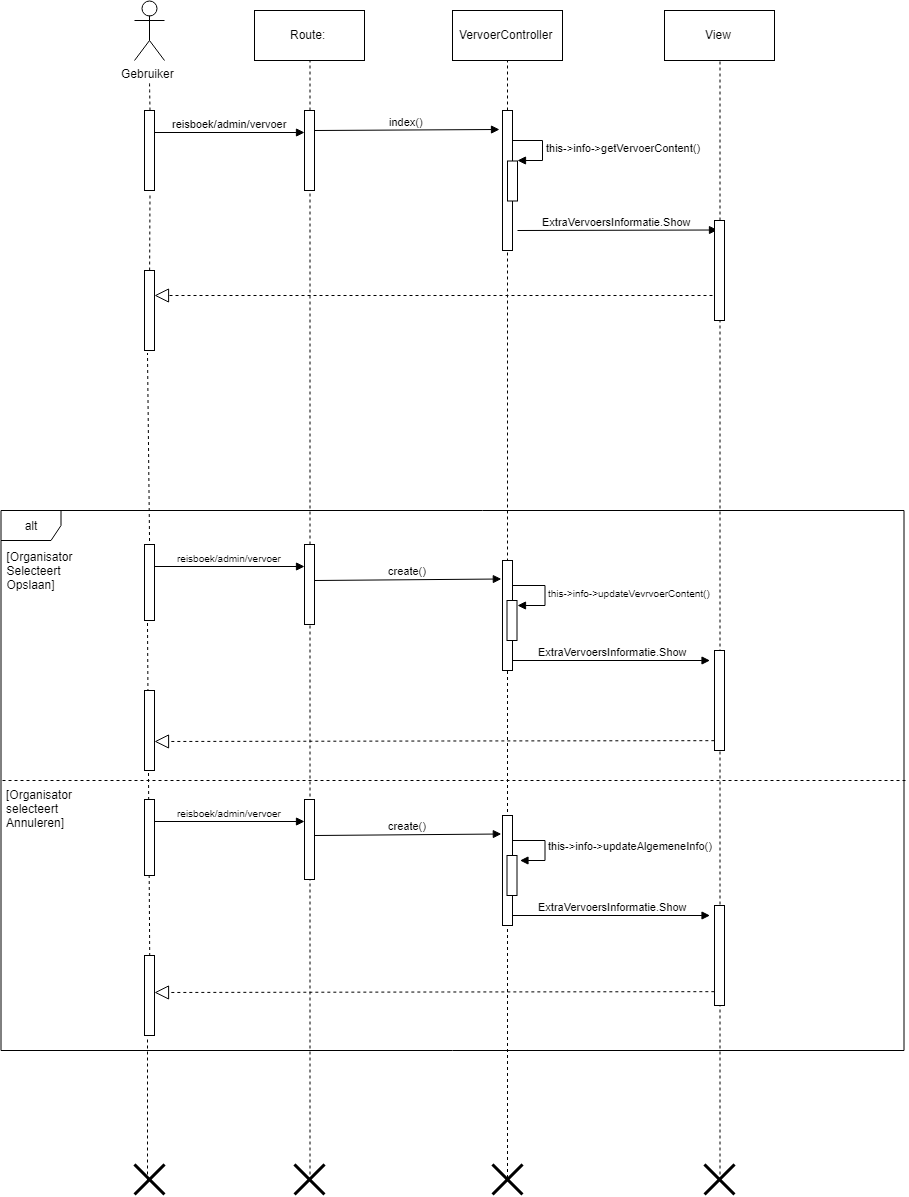
Figuur 13: sequentie diagram algemene informatie

**Hotel**



Figuur 14: sequentie diagram hotel

**Vervoersinformatie**



Figuur 15: sequentie diagram vervoer

# Programma code

## Watervalsysteem

Om ons project op te bouwen gaan we ons houden aan het principe van het watervalsysteem. Dit is een manier om een project op te delen in verschillende stappen. Het watervalsysteem bestaat uit 5 stappen. Requirements, Design, Implementation, Verification en Maitenance. De laatste stap is voor ons niet van toepassing, omdat wij ons project niet moeten gaan onderhouden. Daarom gaan we ons aan de eerste 4 stappen houden. Binnen de eerste stap gaan we samen zitten met de klant en gaat de klant uitleggen wat hij/zij in gedachten heeft voor het project. Hiervan worden dan de use-cases uitgewerkt en eventueel al een paar wireframes zodat de klant een duidelijk beeld heeft bij de use-cases. Als de use-cases zijn goedgekeurd kunnen we verder de wireframes gaan uitwerken. Dit is de tweede stap, design. Wanneer alle wireframes gemaakt zijn gaan we deze ook aan de klant voorleggen. Als deze zijn goedgekeurd kunnen we doorgaan met de volgende stap, implementatie. Binnen deze stap gaan we het meeste doen. Eerst gaan we alle sequentie- en klasse diagrammen opstellen zodat we weten hoe we de code moeten opbouwen. Hierna gaan we de structuur van de database opstellen en nakijken welke velden en relaties we nodig hebben. Wanneer dit af is kunnen we gaan beginnen met het schrijven van de code. Als alle code geschreven en getest is kunnen we doorgaan naar de laatste stap. Dit is de demo voor de klant. Als alle voorgaande stappen juist zijn uitgevoerd, dan zal de klant tevreden zijn met de demo. Als de stappen niet zijn opgevolgd bestaat de kans dat de klant niet tevreden is met het project en dus niet zal willen betalen.

## Laravel

De hele website wordt geschreven in het PHP-framework Laravel. Hierdoor krijgt het hele project een bepaalde structuur waar elk onderdeel zijn plaats heeft binnen de mappen-structuur. Eén van de veelgebruikte mappen is “app”, hierin worden de models, controllers, repositories geplaatst, elk nog eens in hun eigen sub-map. Een andere veelgebruikte map is de “resource” map, hierin komt alles omtrent de html, javascript en css van de webpagina. Verder hebben we ook nog de mappen “database” en “routes”, hierin staat respectievelijk de structuur van de database, hoe de tabellen zijn opgebouwd en eventuele seeders en welke routes er ter beschikking zijn.

Hieronder even een layout van waar de meeste gebruikte mappen gelocaliseerd zijn:

Project

Ͱ app

ǀ Ͱ Http

ǀ ǀ └ Controllers

ǀ Ͱ Models

ǀ Ͱ Repositories

ǀ ǀ Ͱ Contracts

ǀ ǀ └ Eloquent

Ͱ database

ǀ Ͱ migrations

ǀ └ seeds

Ͱ resources

ǀ Ͱ js

ǀ Ͱ sass

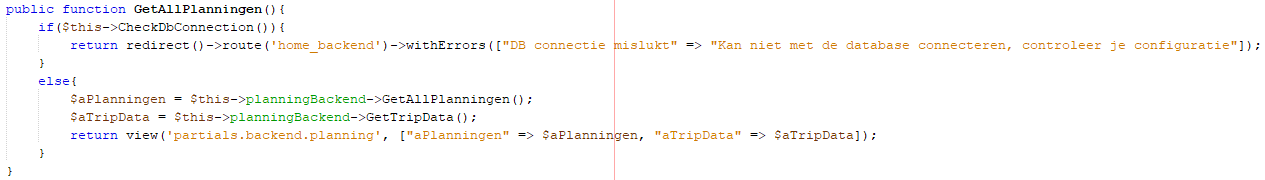
ǀ └ views

└ database

## Opbouw van de code

Voor elke use-case hebben we een sequentie- en een klasse diagram moeten aanmaken.Dit betekent dat we voor elke actie die we uitvoeren en diagram moeten maken. Aan de hand van deze diagrammen kunnen we nu onze code gaan schrijven. We gaan telkens één actie uitkiezen bij de use-cases en deze volledig opbouwen. Om deze te gaan opbouwen gaan we ons baseren op het sequentie- en klasse diagram van deze actie. Zo weten we welke files we moeten aanmaken, welke functies we moeten schrijven en welke views er nodig zijn. Zo kunnen we beginnen schrijven aan de code. Op het moment dat we alle files, functies en views van een bepaalde actie geschreven hebben, kunnen we alles gaan testen. Wanneer alles werkt van deze actie kunnen we overgaan naar de volgende. Zo kunnen ook meerdere personen aan het project werken zonder dat we elkaar hinderen bij het schrijven van de code.

## Controllers

Controllers worden gebruikt om routes af te handelen, meerdere routes kunnen gebruik maken van dezelfde controller maar van andere functies binnen de controller. Natuurlijk zijn we niet verplicht van de controller te gebruiken om de route af te handelen, het kan ook perfect zonder, maar om het overzicht te behouden is het best om de controller te gebruiken.

In het algemeen gaan we in de controller het “CRUD” systeem uitwerken. CRUD staat voor “create” (of maak), “Read” (of lees, selecteer), “Update” (of bijwerken) en “Delete” (of verwijderen). In sommige gevallen moeten we deze niet allemaal uitschrijven.

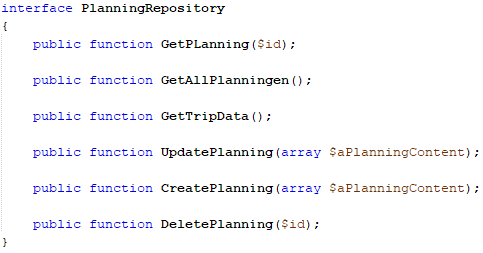
Vanuit de controller gaan we m.b.v. de repository data opvragen of wegschrijven. Hoe dit wordt gedaan maakt niet uit voor de controller, het effectief behandelen van de data gebeurt in de eloquent (zie verder).

## Models

https://scontent.fbru1-1.fna.fbcdn.net/v/t1.15752-9/64732309_2045118499116343_880241302363439104_n.png?_nc_cat=100&_nc_ht=scontent.fbru1-1.fna&oh=f4126d639b3895473e260afb8bace379&oe=5D8AFC5FVoor elke tabel binnen de database gaan we een model aanmaken. Binnen deze models gaan we definiëren welk veld de primaire sleutel is van de tabel. Dit doen we door een protected variabele $primaryKey aan te maken. Zie de volgende figuur.

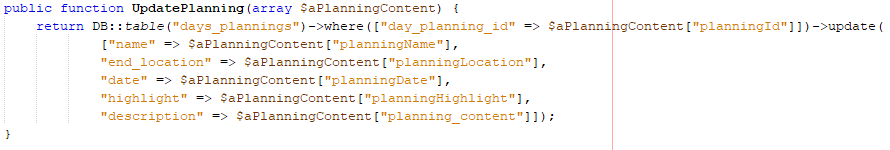
https://scontent.fbru1-1.fna.fbcdn.net/v/t1.15752-9/64659353_454250548453148_2780315047199506432_n.png?_nc_cat=110&_nc_ht=scontent.fbru1-1.fna&oh=f2ea2adfefedc43e2b99da57571e9994&oe=5DC63F20De naam van deze key is standaard de naam van de tabel\_id. Binnen de models gaan we ook de relaties van de tabel leggen. Dit doen we door een public function aan te maken met als naam de naam van de tabel waarmee de relatie gelegd moet worden. Binnen deze functie gaan we definiëren of we te maken hebben met een één of veel relatie. Bij een één relatie moeten we belongsTo gebruiken en bij een veel relatie gebruiken we hasMany. Verder moeten we ook het model van de tweede tabel gaan meegeven en dan alles gaan returnen met het statement return. De code van de relatie tussen DaysPlannings en DaysPlanningsTrips ziet er zo uit zoals hieronder weergeven.

## Interfaces

Interfaces bevatten geen code die we gaan uitvoeren, maar bevatten functies die een andere file gaat moeten gebruiken. In de andere file kunnen we een interface gaan implementeren en zo ook deze functies krijgen. Dit is handig als we met meerdere programmeren, zo kunnen we geen functies over het hoofd gaan zien. Voor elke view gaan we een interface aanmaken, op deze manier blijft de code van alle views mooi gescheiden. Op de onderstaande figuur zien we de interface voor planning met de naam PlanningRepository. We zien duidelijk de 6 functies die we gaan nodig hebben voor alles van planning.

## Eloquent

De files in de map “Eloquent” gaan de interfaces implementeren uit de map “Contracts”. Voor elke interface gaan we dus ook een file maken binnen deze map. Binnen deze files gaan we dus de functies van de interfaces opvullen met code. De functies hebben de bedoeling om data uit de database te gaan halen en deze te gaan filteren zodat de gewenste data overblijft. Op het moment dat deze files, de view en de controller van de juiste pagina af is, kunnen we de data in de view gaan steken. Op dit moment kunnen we dus gaan testen of de pagina is zoals we deze hadden voorgesteld. Op **Fout! Verwijzingsbron niet gevonden.** zie je de functie UpdatePlanning van de functies van EloquentPlanning. Deze functie krijgt een array mee met data in. Daarna word er in de taben days\_plannings op de juiste velden deze data ingevuld waar het day\_planning\_id klopt met degene die in de array is meegestuurd.



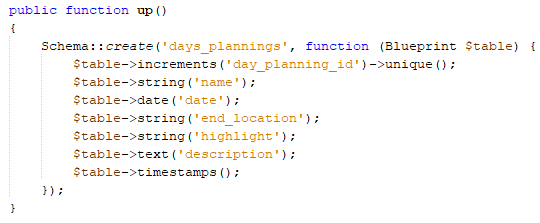
## Views

https://scontent.fbru1-1.fna.fbcdn.net/v/t1.15752-9/64753712_454986238393023_5330445741691240448_n.png?_nc_cat=111&_nc_ht=scontent.fbru1-1.fna&oh=c9e295f9e1c0bed98a783b0dc42063e6&oe=5D929391Voor het maken van de views werken we niet met gewone html-bestanden maar met blade.php bestanden. Blade is een sjabloonengine voorzien van Laravel. Het grote voordeel van Blade is dat dit niet beperkt is tot PHP-code. De 2 belangrijkste voordelen van Blade zijn template-overerving en secties. Binnen Blade kunnen we dankzij de template-overerving een vaste lay-out gaan opstellen. Deze lay-out zal gebruikt worden door de rest van de views als een template, en de andere views zullen hier dus op gaan voortbouwen. Hierdoor moeten we code die in elke view voorkomt maar 1 keer schrijven. Een voorbeeld hiervan is onze navbar. Deze hebben we opgesteld in een aparte lay-out, zodat we deze maar 1 keer moeten opstellen. Om deze te gaan invoegen moeten we code schrijven  
zoals op de volgende figuur.

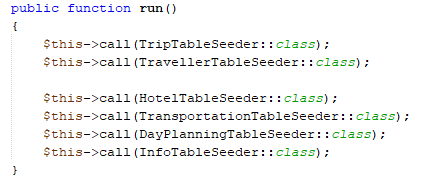
Verder gaan we ook gebruik maken van secties. Deze secties worden gebruikt om een deel van de inhoud weer te geven. Blade werkt gewoon met de standaard html tags. Deze tags worden naast secties en overerving nog aangevuld met andere sleutelwoorden.

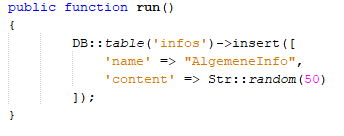
Deze sleutelwoorden worden voorafgegaan door een ‘@’ en zullen een uitbreiding vormen op de secties en de overerving. Het is ook mogelijk om javascript te gaan toevoegen binnen een Blade bestand. Deze code gaan we schrijven tussen de tags <script></script>. Deze code wordt dan automatisch toegepast op de juiste elementen binnen het bestand. Op het moment dat een Blade bestand af is en alle andere pagina’s voor deze pagina bestaan, kunnen we dit gaan testen.

## Migrations

Migrations gaan we gebruiken voor het opbouwen van de structuur van de database. Voor elke tabel gaan we een migration moeten aanmaken. Dit doe je door het commando php artisan make:migration “naam van de tabel” te gaan ingeven in Git Bash in de map van het project. Vervolgens wordt het bestand in de map “migrations” binnen database geplaatst. Binnen deze migrations gaan we elk veld binnen de tabel definiëren. Dit gaan we doen binnen de functie up(). Binnen deze functie gaan we ook meegeven wat de naam van de tabel is en dat dit een tabel is. Hierna gaan we alle velden ingeven met het corresponderende type. De primaire sleutel van de tabel gaan we ook de tag Unique meegeven, zo weet Laravel dat dit de primaire sleutel is. Verder moeten ook binnen de migrations alle relaties woorden gelegd. Dit doe je door de volgende regel mee te geven $table->foreign(‘gemeenschappelijk veld’)->references(‘gemeenschappelijk veld’)->on(‘tweede tabel’);. Deze regel en de functies binnen de models gaan ervoor zorgen dat alle relaties tussen de tabellen van de database worden gelegd. Een volledig uitgewerkte migration kan je zien onder deze alinea. Om de structuur van de database te laten genereren geven we nu het commando php artisan migrate in de Git Bash. Als alle tabellen en relaties juist zijn krijgen we nu de structuur van de database.

## Seeders

Wanneer de structuur van onze database in orde is, hebben we nog altijd geen data om te gaan testen. Hiervoor gaan we gebruik maken van seeders. Seeders gaan willekeurige data in de database steken, zodat we ons project kunnen gaan testen. Om seeders aan te maken voeren we het commando php artisan make:seeder “naam van de seeder” uit in Git Bash. De seeder wordt binnen de map “seeds” geplaatst. Binnen deze map staat ook de file DatabaseSeeder.php. Binnen deze file gaan we elke seeder die we willen gebruiken definiëren, als we dit niet doen zal de seeder niet gaan werken. Zie de volgende figuur.

Wanneer de seeders zijn toegevoegd kunnen we deze gaan vullen met data. Om te beginnen gaan we de ervoor zorgen dat we gebruik kunnen maken van de klasse Str. Dit doen we door de regel use Illuminate\Support\Str; in te geven vanboven in het bestand van de seeder. Als deze klasse is toegevoegd kunnen we willekeurige stukken tekst van een bepaalde lengte gaan invoegen als data. Vervolgens gaan we de functie run() opvullen. Binnen deze functie gaan we de data in de database steken. Dit doen we met de regel DB::table(tabel')->insert([]). Tussen de vierkante haakjes gaan we elk veld binnen de tabel opvullen met data. Dit doen we met de regel ‘veld’ => “waarde”. Als we bijvoorbeeld namen moeten gaan invoegen kunnen we gebruik maken van de functies van de klasse Str. Door Str::random(lengte) in te geven als data, wordt er willekeurige data met een bepaalde lengte gegenereerd. Een volledig uitgewerkte seeder is te zien op **Fout! Verwijzingsbron niet gevonden.**. Als alle seeders opgevuld zijn moeten we deze nog gaan inladen binnen ons project. Dit doen we door de regel php artisan migrate:fresh –seed in te geven in de Git Bash. Deze regel gaat ervoor zorgen dat alle migrations een seeds die in het project zitten overschreven worden door alle migrations en seeds die in de code staan op het moment van de upload. Hierdoor krijgen we nooit dubbele migrations of seeds. Nu is de volledige pagina werkende en moeten we alleen nog de opmaak gaan invoegen.

## Opmaak

Als alle voorgaande stappen werken is alle functionaliteit van de site afgerond. Nu moeten we enkel nog de opmaak gaan aanpassen. Sommige elementen hebben al opmaak omdat we deze hebben voorzien van een Style-tag in de HTML-code. Voor de overige elementen gaan we de opmaak nog moeten aanpassen. Normaal doen we dit via CSS. Maar bij Laravel hebben we hiervoor een alternatief, namelijk Sass. Sass werkt via sccs-files. Dit zijn files waarin we onze CSS-code gaan schrijven. Deze code gaat niet rechtstreeks als CSS worden opgenomen door de site. We gaan hiervoor het commando mix.sass('resources/sass/\_frontend.scss', 'public/css/\_frontend.css'); moeten uitvoeren. Door dit commando komt alle code van \_frontend.scss in het bestand \_frontend.css. Dit bestand staat in de map public. In dit bestand gaan we zelf geen code schrijven, maar we gaan dit wel linken met alle views. Als alles goed verloopt heeft de hele website nu de opmaak die is ingegeven in \_frontend.scss

# Besluit

Uit dit project hebben we geleerd hoe we in teamverband moesten werken. Ook hebben we voor het eerst gewerkt met het framework laravel. Bovendien hebben we ook nog voor het eerst kennis gemaakt met het watervalsysteem. We kunnen uit dit watervalsysteem besluiten dat dit geen goed systeem is. De requirements en vooral de software development analyses namen meer tijd in beslag dan gepland. Hierdoor geraakten we ver achter op schema en moesten we de laatste weken voor de deadline nog veel bezig zijn met programmeren. Ook is dit systeem nadelig voor de klant. De klant krijgt het product namelijk pas op de deadline voor de eerste keer te zien. Indien de klant niet tevreden zou zijn met het geleverde product, moeten sommige delen misschien volledig opnieuw worden gemaakt. Een mogelijke oplossing hiervoor kan zijn dat men het project in verschillende kleinere onderdelen splitst. Deze onderdelen kunnen we dan apart aan de klant laten zien om te zien of deze tevreden is of niet. Indien deze niet tevreden is, zal niet het hele project moeten worden aangepast maar slechts dit onderdeel.