BACHELORPROEF

OPRICHTING VERKEERSCENTRUM GENT

Team Verkeer-1

Simon Backx
Jarno De Meyer
Piet Verheye
Robin Weymans

Academiejaar 2015–2016
Bachelor of Science in de industriële wetenschappen: informatica
Valentin Vaerwyckweg 1, 9000 Gent

INHOUDSTAFEL

INHOUDSTAFEL

ABSTRACT

Nederlands

Het Mobiliteitsbedrijf Stad Gent is sinds 2014 bezig met het opzetten van een regionaal verkeerscentrum. Op termijn is het de bedoeling van het verkeer in de regio semi-automatisch te monitoren met enkel op piekmomenten een bemande dienstverlening. Zo zouden tijdens de week onverwachte incidenten of verhogingen van reistijden automatisch kunnen gesignaleerd worden aan een verantwoordelijke die hierdoor acties kan ondernemen. Verder wil men ook de gevolgen van ingrepen in het mobiliteitsbeleid kunnen analyseren.

Om dit te verwezenlijken is het noodzakelijk van informatie over de verkeerssituatie te verzamelen. Deze verkeersinformatie kunnen we halen bij gespecialiseerde providers zoals bijvoorbeeld TomTom en Coyote. Een server-applicatie vraagt elke 5 minuten informatie op bij deze providers en slaat de gegevens hiervan op in een databank.

Vervolgens wensen we deze informatie op een vlotte en duidelijke manier weer te geven aan een gebruiker. Er is geopteerd voor een tweeledige invulling hiervan: enerzijds stelt een REST API de verwerkte data ter beschikking en anderzijds is er een webapplicatie die deze API gebruikt om een duidelijk totaalbeeld te genereren.

Door deze verzamelde informatie overzichtelijk aan te bieden kan de stad Gent zijn inwoners beter informeren over de verkeerssituatie maar ook de gevolgen van zijn mobiliteitsbeleid opvolgen en bijsturen indien nodig.

English

Mobiliteitsbedrijf Stad Gent has been establishing a regional traffic control centre since 2014. In time, the plan is to monitor the traffic flow on a semi-automatic basis. A system could then automatically notify an operator of sudden incidents or extended travel times which could then be used to respond accordingly. At peak moments a manned service would still be required. As a second requirement, the city wants to be able to analyse the effects of changing traffic policies.

To realise this system, we first need to collect traffic data. This data is available from many specialised providers such as TomTom or Coyote. A server application gathers information from these providers every 5 minutes and subsequently stores it in a database to be retrieved later on.

Afterwards, we want to display this information to a user in a clear and easy-to-use manner. We have opted to use a dual system: first we have a RESTful API to provide access to the processed data and secondly we have a web application which uses this API to generate a clear overview.

By offering this collected data in an easy-to-use manner, the city of Ghent would be able to inform its citizens of the current traffic situation, but also be able to monitor and update changes to its traffic policy.

INLEIDING

Verkeer1 is een project door studenten van Universiteit Gent in het kader van de oprichting van het verkeerscentrum in Gent.

De jongste maatregelen om het verkeer in de binnenstad af te remmen, delen de Gentse binnenstad op in zones. Een automobilist die naar een andere zone wil, moet daardoor altijd gebruikmaken van de binnenring. Een voor de hand liggend gevolg is dat de binnenring sterker belast zal worden.

Gent beschikt nog niet over een verkeerscentrum dat de situatie op de toegangswegen real-time in beeld brengt. Om daarin te voorzien werd de hulp ingeroepen van enkele studenten. Die realiseren op een projectmatige manier een softwareapplicatie die in deze behoefte voorziet. Dit alles gebeurt binnen het kader van een bachelorproef. Om dit project zo realistisch mogelijk te maken is er voorzien in een regelmatig overleg met de opdrachtgever. Tegen deze achtergrond is een warme en hartelijke dank verschuldigd aan iedereen binnen de administratie Verkeer van de stad Gent en iedereen binnen de Universiteit Gent, om dit project zo werkelijkheidsgetrouw mogelijk te maken.

GEBRUIKERSDOCUMENTATIE

Samenvatting

Functionaliteit

Verkeer1 is een applicatie die de congestie van het autoverkeer op de invalswegen naar Gent in beeld brengt. Het is dus geen hulpmiddel die u de snelste weg binnen de agglomeratie gaat adviseren.

Wat nodig

- Een browser met javascript ondersteuning:
  - Internet Explorer (vanaf IE9)
  - Geen Opera Mini
- Actieve internetverbinding
- Schermresolutie 1920x1080 (optimaal)
  - Functionaliteit op mobiel is beperkt tot 'Vandaag'-weergave
- Gebruikersnaam en wachtwoord

Aan de slag

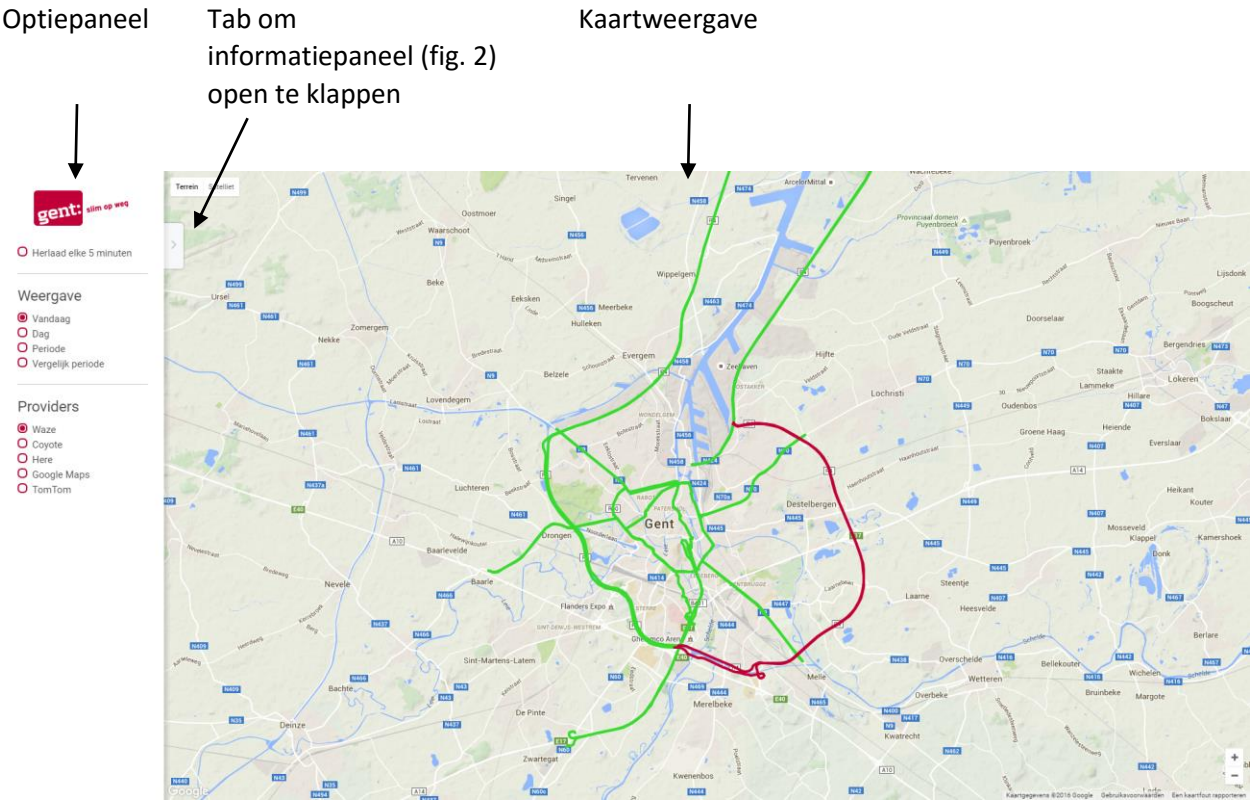
- Surf naar *http://verkeer-1.bp.tiwi.be/*.
- Gebruikersnaam: root
- Wachtwoord: aeSqFPbpUl

Deze gebruikersdocumentatie gaat op een gestructureerde manier in op alle onderdelen van de gebruikersinterface. We hopen dat de functionaliteiten van de applicatie op een intuïtieve manier gerealiseerd zijn, zeker voor personen die vertrouwd zijn met het onderwerp. Daarom kan je ook zelf op verkenning gaan binnen de applicatie. De doelgroep van deze documentatie is de medewerker van het verkeerscentrum die de applicatie zal gebruiken.

Hulp nodig?

Zolang de toepassing in ontwikkeling is kan je terecht bij het ontwikkelteam op het e-mailadres: *robin.weymans@ugent.be*

Na het opstarten van de applicatie verschijnt volgend scherm:



Figuur 1: startpagina webapplicatie

Het opstartscherm (Figuur 1) toont de kaart van de Gentse agglomeratie met alle invalswegen:

- De invalswegen waar de congestie in beeld gebracht worden, zijn ingekleurd. Bij het opstarten van de applicatie toont die kleur de verkeerssituatie op dat moment. De kleuren zijn geïnspireerd op wat we intuïtief aanvoelen.
    - Groen = verkeerssituatie is beter van normaal.
    - Oranje = verkeer is normaal.
    - Rood = verkeerssituatie is minder vlot dan normaal.
- Links zie je het **optiepaneel**. Dat is altijd zichtbaar. Het stelt de eindgebruiker in staat om de verkeerscongestie op een ander tijdstip in beeld te brengen, of om de informatie van een andere dataprovider te visualiseren.
    - Weergave: Wil je de verkeerssituatie op een ander moment dan de huidige situatie, dan kan je kiezen tussen:
        - Vandaag
        - Een moment in het verleden
        - Een periode
        - Een vergelijking tussen twee dagen

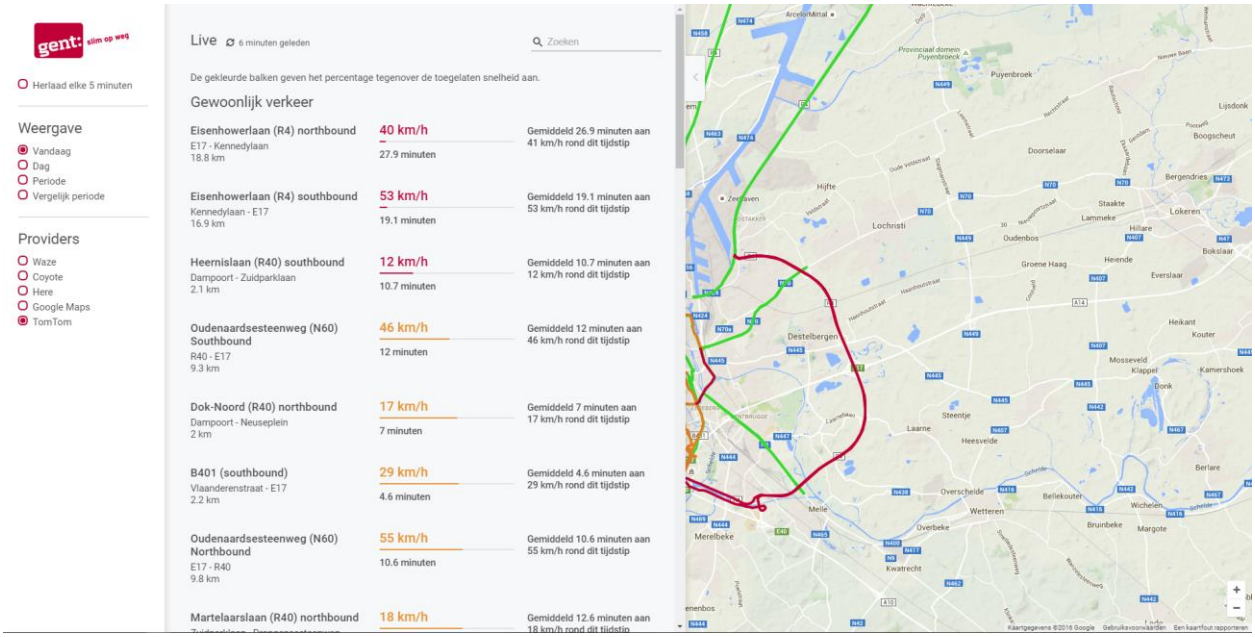
      Deze keuze bepaalt of en welke parameters ingesteld moeten worden, en welke informatie zichtbaar is op de kaart.

    - Providers: je kunt kiezen van welke provider de verkeersinformatie afkomstig is, of alle providers combineren (nog niet in de huidige versie).

      Let wel: niet alle providers kunnen meetwaarden aanleveren voor alle routes.

- Om tekstuele details over de routes weer te geven druk je op de pijl rechtsboven het optiepaneel.

Het informatiepaneel wordt nu uitgeklapt.



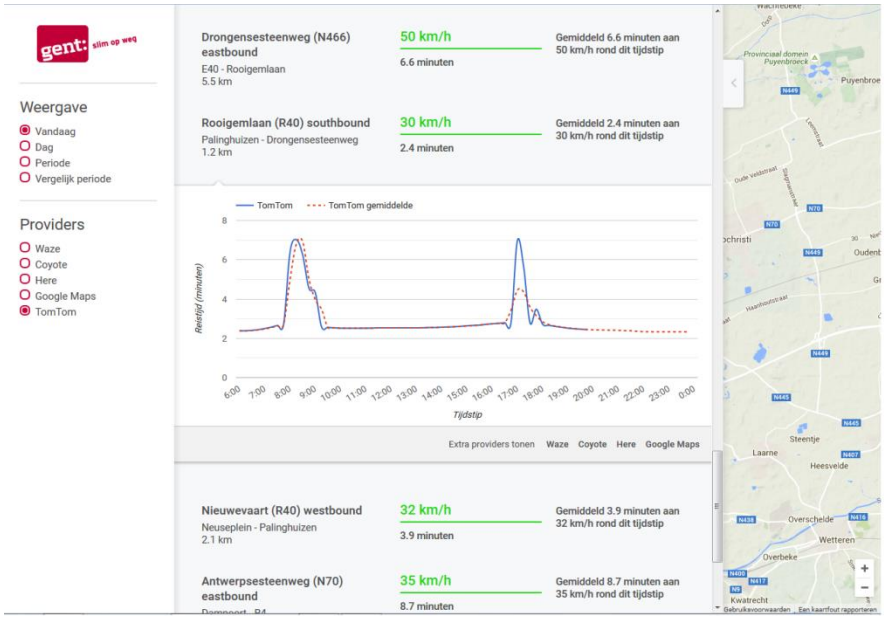
Figuur 2: Beeld bij opengeklapt informatiepaneel

Weergave

Er zijn vier opties, waarvan er op dit moment drie gerealiseerd zin, namelijk 'Vandaag', 'Periode' en 'Vergelijk periode':

Vandaag

Hoe druk is het op dit moment op een route? Deze informatie omvat de snelheid waarmee het verkeer zich verplaatst en de tijd die nodig is om de route af te leggen. Klik op een route voor meer details.



Figuur 3: Extra details van een route

Vervolgens verschijnt er een grafiek die de huidige verkeersdrukte afbeeldt naast de normale verkeersdrukte op deze dag van de week. De blauwe lijn toont de ontwikkeling van de live informatie van vandaag tussen 06:00 ’s morgens tot de laatste waarnemingen (tot ten laatste 23:59:59). De rode stippellijn stelt de normale situatie voor die een weggebruiker mag verwachten op deze dag van de week. Daarmee verschilt dit begrip duidelijk van de ideale reistijd. De ideale reistijd is altijd dezelfde, onafhankelijk van het moment op de dag of van de dag in de week.

Periode en Dag

Hierbij krijg je informatie over de drukte tijdens een bepaalde periode of dag. Deze gebruiker stelt de periode of dag zelf in. In het opengeklapte informatiepaneel kan u een extra grafiek opvragen die de werkelijke situatie in deze periode of op deze dag vergelijk met de normale situatie.

Vergelijk periode

Hiermee kan je twee periodes met elkaar vergelijken. Zo kan je zien op welke routes de situatie na het nieuwe mobiliteitsplan verbeterd is. De routes waar de verschillen in reistijd het grootst zijn verschijnen bovenaan. Op de grafiek zien we een vergelijking van uur tot uur en de gemiddelde reistijd voor dezelfde route in de twee periodes.

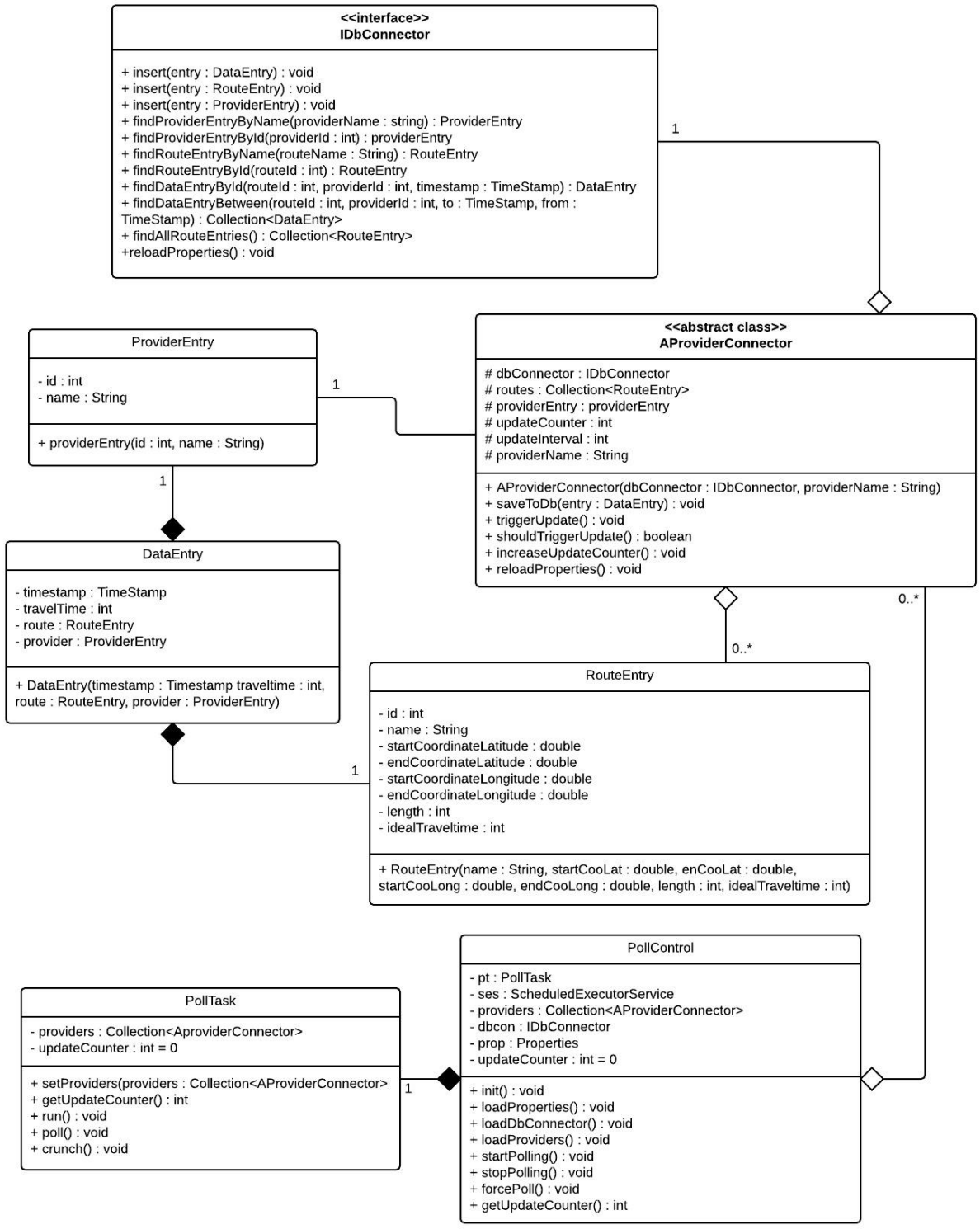
Kaartweergave

Elke route heeft een status zoals 'Trager dan gemiddelde', 'Sneller dan gemiddelde' of 'Trager dan gemiddelde'. Stilstaand verkeer wordt rood weergegeven, traag verkeer oranje en vlot verkeer groen.

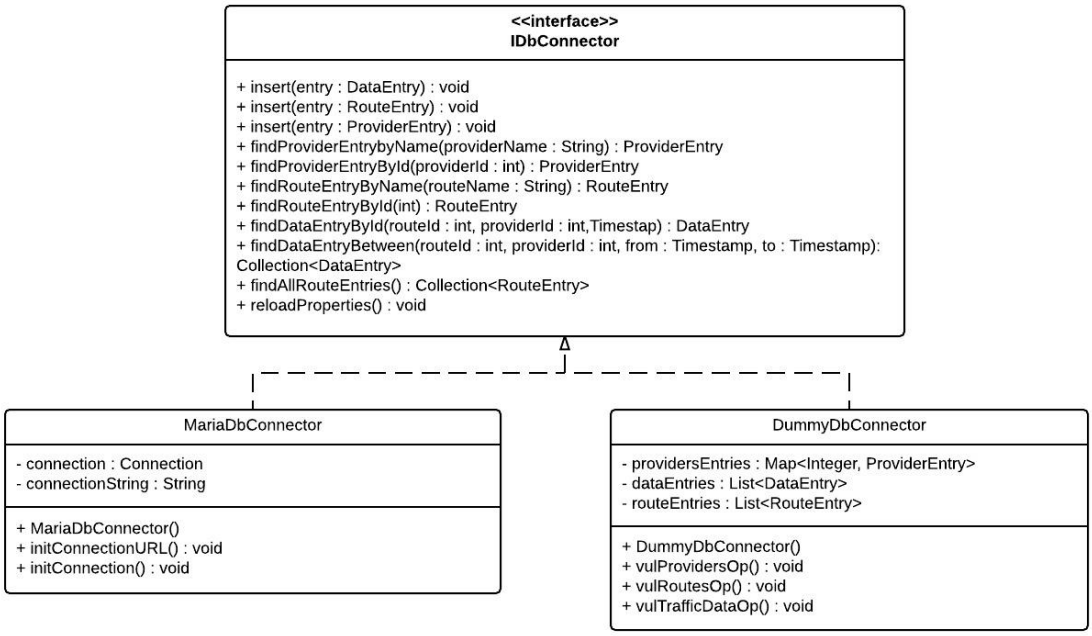
Wanneer de gebruiker met zijn muis een route selecteert, kan men dezelfde informatie bekijken als in het informatiepaneel. Het is hier niet mogelijk om de grafieken te bekijken.

Mogelijke parameteriseerbaarheid

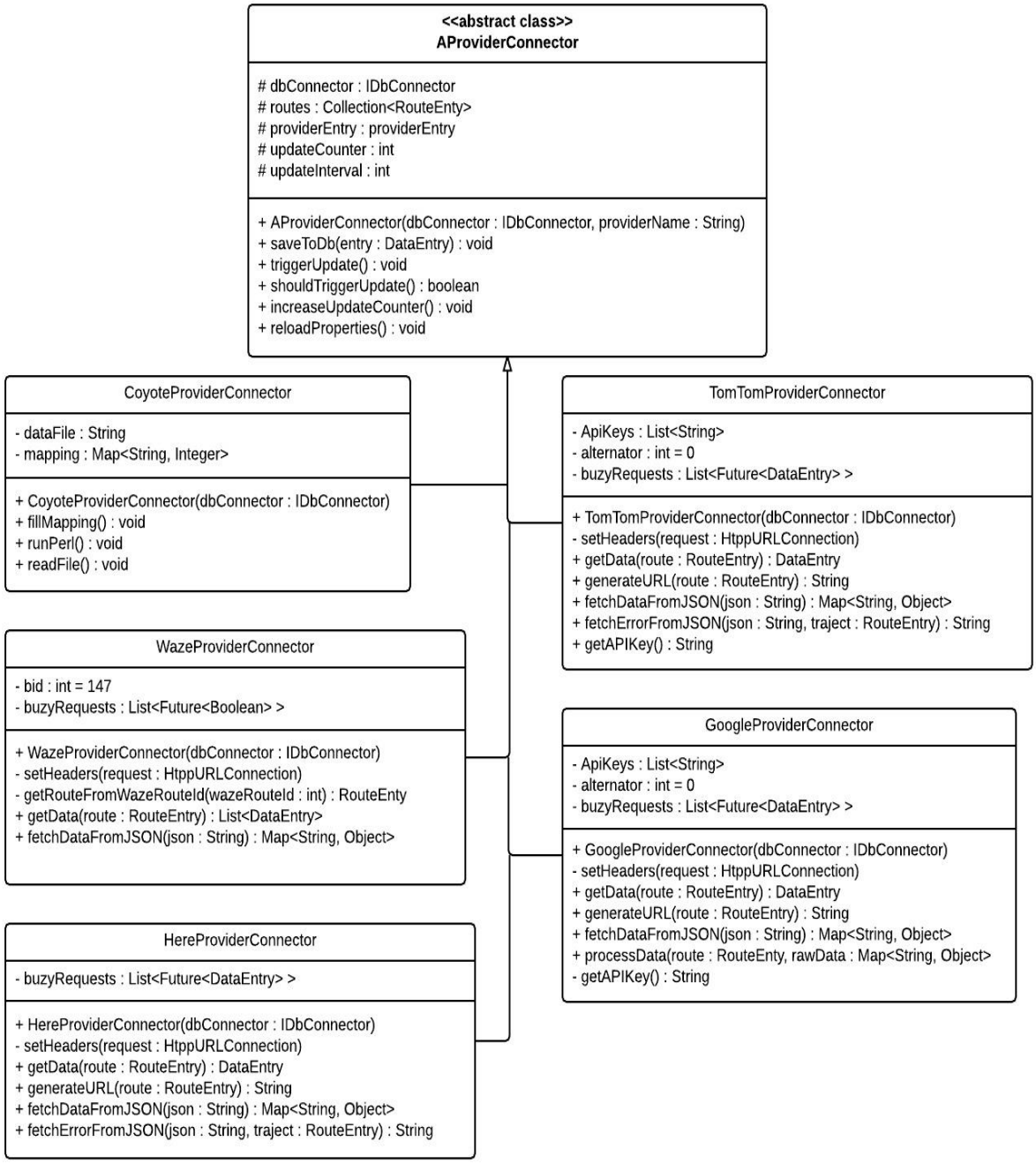
- Volgorde van de providers in het optiepaneel
  kan aangepast worden, maar niet door een eindgebruiker. Neem hiervoor contact op met het ontwikkelingsteam.
- Kleur waarmee de invalswegen ingekleurd zijn
- De grenzen tussen de verschillende statussen van een invalsweg



Figuur 4: klassendiagram verkeer



Figuur 5: klassendiagram IDbConnector



Figuur 6: klassendiagram AProviderConnector

ONTWIKKELAARSDOCUMENTATIE

Samenvatting

Een gebruiker komt in normale omstandigheden enkel in contact met de webapplicatie. Een ontwikkelaar daarentegen zal ook te maken krijgen met de Serverapplicatie, Databank en de API. Hieronder worden de genoemde onderdelen in detail besproken en uitgelegd hoe functionaliteit kan toegevoegd worden. Om te volgen in deze omschrijving neem je best de diagrammen (figuur 4-5-6) bij de hand.

Serverapplicatie

De ruggengraat van de applicatie is een PollThread die elke vijf minuten alle providers aanspreekt en daarbij vraagt om de reistijdinformatie over de routes te verzamelen. Zo'n provider wordt voorgesteld door een afgeleide klasse van AProviderConnector. Wanneer je gebruik zou willen maken van een andere provider kan van deze klasse afgeleid worden. Concrete implementaties hiervan zijn bijvoorbeeld TomTomProviderConnector en WazeProviderConnector.

De opgehaalde reistijdinformatie wordt opgeslagen in een databank. De AProviderConnector-klassen communiceren met deze databank via een IDbConnector-klasse. MariaDbConnector is een klasse die deze interface implementeert. Analoog met de hierboven vermeldde provider-klassen zou je deze klasse kunnen inwisselen door een andere implementatie, zolang de interface gerespecteerd wordt.

Wanneer je de serverapplicatie opstart, dan zal de gewenste IDbConnector-klasse geïnstantieerd worden. De configuratie van dit object kan men terugvinden en aanpassen in het bestand 'config/database.properties'. Een nieuwe implementatie kan van hetzelfde bestand gebruik maken om configuratiegegevens op te slaan.

Meteen hierna worden alle AProviderConnector-klassen geïnstantieerd. Elk van deze klassen krijgt een referentie mee naar hetzelfde IDbConnector-object. Ook hier wordt de configuratie van deze objecten opgeslagen in een configuratiebestand. Dat bestand kan je vinden onder 'config/provider.properties'.

Ook de PollThread zelf heeft een configuratiebestand 'config/app.properties' waarin properties zoals het pollinterval kunnen aangepast worden. Je kan er ook instellen welke provider-klassen er gebruikt moeten worden.

Databank

In de huidige implementatie maakt de applicatie gebruik van een MySQL databank. Zoals hierboven al vermeld kan je, indien je dat wenst, gebruik maken van bijvoorbeeld een Oracle databank.

Er zijn vijf tabellen waarin de informatie over routes, providers en reistijdinformatie in opgeslagen worden. Deze zijn gekozen zodat er zo weinig mogelijk informatie redundant wordt opgeslagen. Momenteel bevat de databank ongeveer 700.000 rijen reistijdinformatie en heeft die een grootte die kleiner is dan 100 Mib.

Providers

Deze tabel bevat de identificatie (‘id’), de naam (‘name)’ en het gewicht (‘weight’) van een provider. Dit gewicht wordt gebruikt om de reistijden van alle providers te aggregeren. Het gewicht geeft aan hoeveel de informatie van een provider doorweegt. In de huidige versie staan deze gewichten allemaal op 1 zodat elke provider even zwaar doorweegt.

Routes

Deze tabel bevat de identificatie (‘id’), naam (‘name’), beschrijving (‘description’), lengte (‘length’), begin- en eindcoördinaat van een route (‘startlat’, ‘startlong’, ‘endlat’ en ‘endlong’) en de maximale snelheid op deze route (‘maxspeed’). De id’s zijn gemapt op de specifieke waarden van de provider Waze. Dit is om praktische redenen.

Waypoints

Deze tabel bevat een identificatie (‘id’), een verwijzende sleutel (‘routeID’) naar de identificatie van een routes, een volgnummer (‘sequence’), een lengte-coördinaat (‘longitude’) en een breedte-coördinaat (‘latitude’). Deze informatie wordt vooral gebruikt om de route te tekenen op de kaartweergave (in de webapplicatie). Een volgnummer is nodig omdat de volgorde van elke waypoint uiteraard belangrijk is.

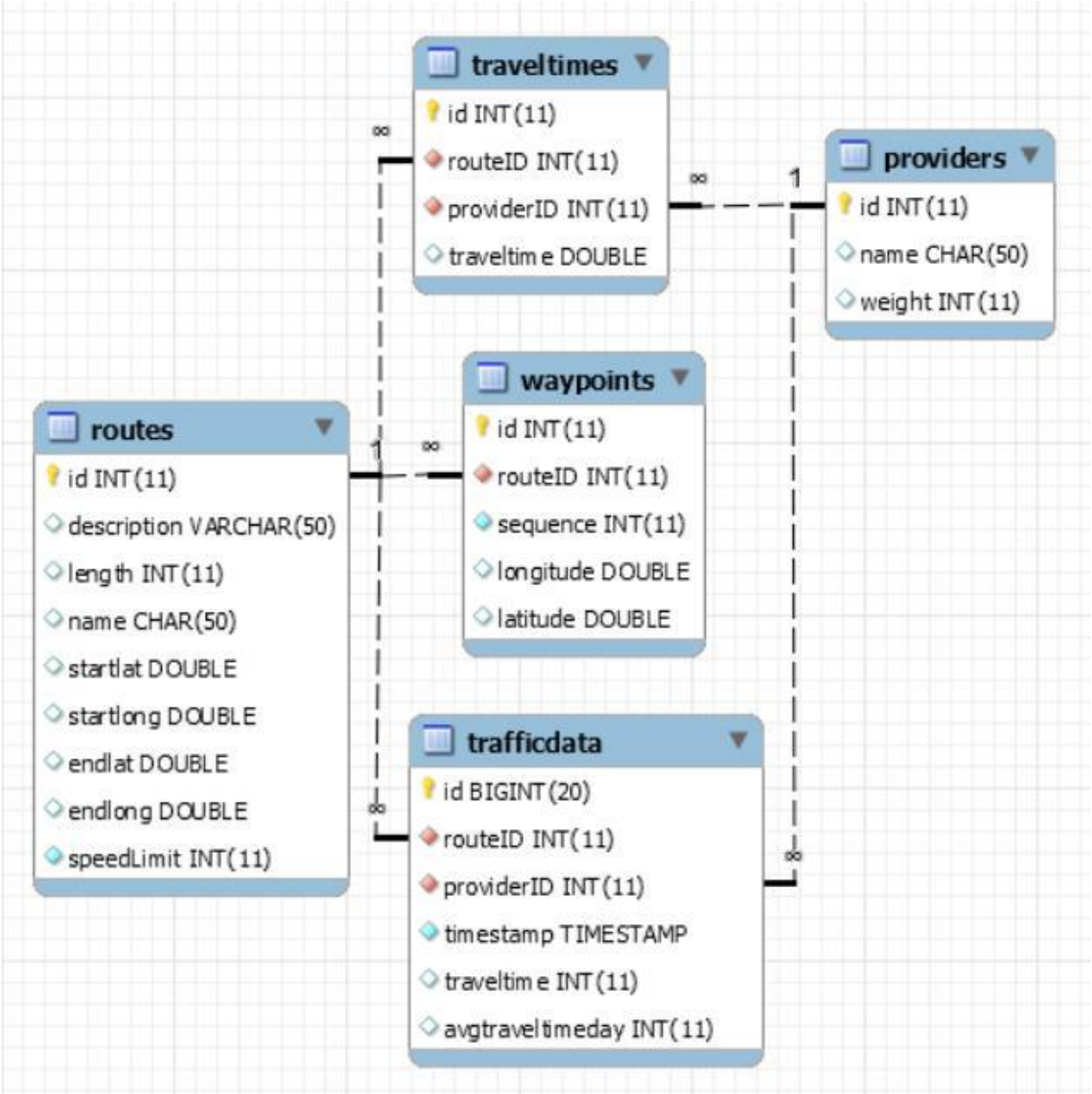
Trafficdata

Deze tabel bevat identificatie (‘id’), een verwijzende sleutel (‘routeID’) naar de identificatie van een route, een verwijzende sleutel (‘providerID’) naar de identificatie van een provider, een timestamp (‘timestamp’) en de reistijdinformatie (‘traveltime’). Deze tabel bevat alle reistijd informatie die door de provider-klassen wordt opgehaald.

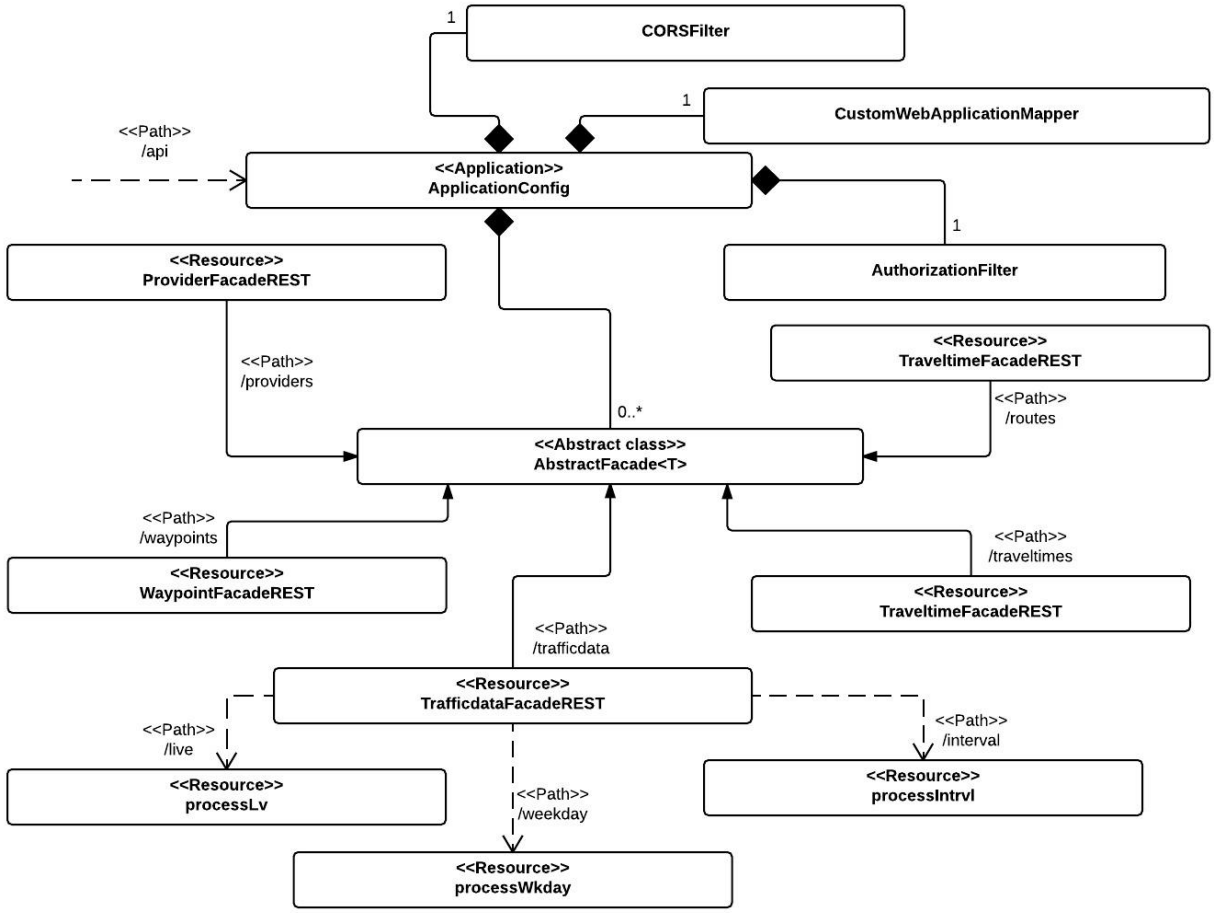
Traveltimes

Deze tabel bevat een identificatie (‘id’), een verwijzende sleutel (‘routeID’) naar de identificatie van een route, een verwijzende sleutel (‘ProviderID’) naar de identificatie van een provider en een ideale reistijd (‘traveltime’). Hierin wordt per route en per provider aangegeven welke reistijd door de webapplicatie als ideaal mag beschouwd worden.

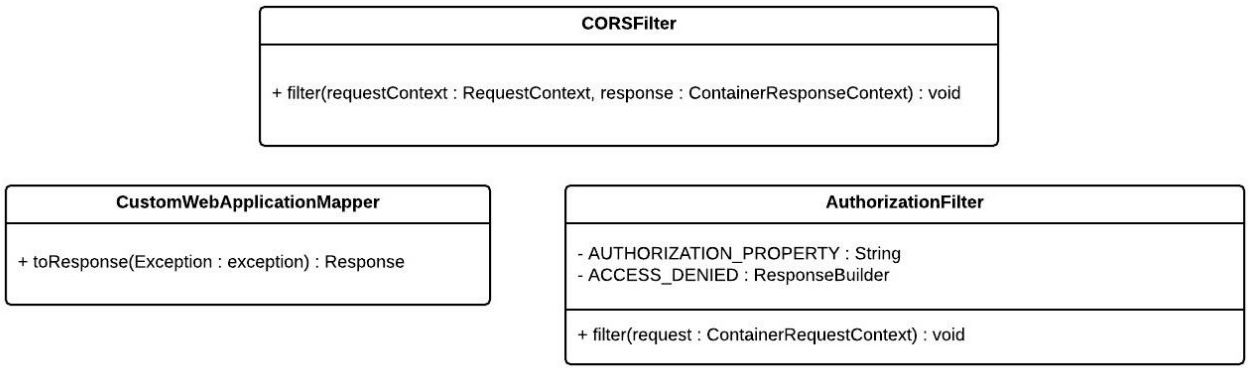
Een visuele weergave van de databank vind je op figuur 7.



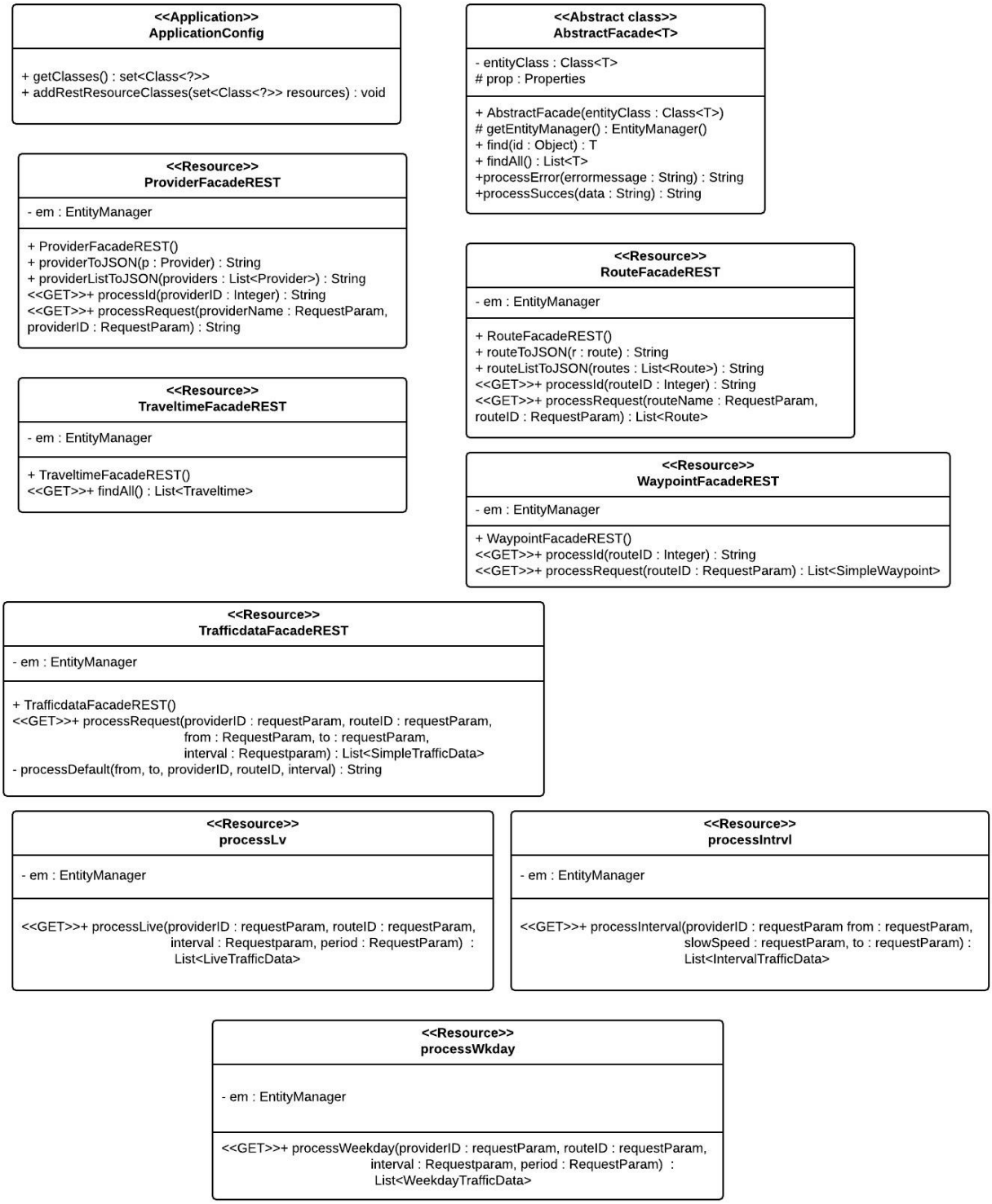
Figuur 7: databankschema



Figuur 8: klassendiagram Rest API



Figuur 9: Details van applicatie-klassen



Figuur 10: klassendiagram javascript gedetailleerd

API

Werking

Om de gegevens op te halen uit de databank en deze te gebruiken in de webapplicatie wordt gebruik gemaakt van een REST API (Representation State Transfer Application Programming Interface). Dit is een interface van de databank, die bereikt wordt door http-berichten naar specifieke URL's te sturen. Deze interface kan ook eventueel door derden worden gebruikt die van de data willen gebruik maken om een eigen applicatie te ontwikkelen. Deze toegankelijkheid is afhankelijk van het authenticatie-systeem die voor de API wordt gebruikt.

Momenteel gebeurt de authenticatie via één enkele API-key die hard gecodeerd in de REST API zit. De API-key moet meegegeven worden in de custom-header 'X-API-KEY' bij elke aanvraag. Indien deze niet aanwezig is of niet correct is wordt een error bericht teruggegeven aan degene die de aanvraag uitvoerde. Bedoeling is dat dit authenticatiesysteem wordt uitgebreid naar een meer geavanceerde variant door gebruik te maken van een login- en registratiescherm die accounts (met bijhorende username en password) linkt aan personen die geautoriseerd zijn om API-keys op te vragen. Een andere mogelijkheid is gebruik maken van Auth0 of OAuth2 die gebruik maken van users en tokens. Op basis hiervan kan je dan autorisatie geven per sessie of een API-key aanbieden.

De REST API is gebaseerd op het principe van annotaties, die gekenmerkt worden door het '@'-teken. Deze annotaties zorgen dat het API-systeem bepaalde objecten, klassen en methodes herkent en er geen web.xml en andere configuratie-bestanden extra nodig zijn om alle klassen als 1 geheel te laten samenwerken. Zo zal bijvoorbeeld @Application-annotatie instaan voor de Applicatie-klasse (zie verder) te identificeren.

De REST API is deels gegenereerd op basis van de databank en bevindt zich in het webapplicatie-project VerkeerREST, hetzelfde project waarin de webapplicatie zelf is geïmplementeerd. De databank wordt voorgesteld door objecten in de package 'domain', hierin gebeurt dus het mappen van de tabellen (@Table) en hun kolommen (@Columns) uit de databank naar objecten in Java.

In de package 'service' bevinden zich de klassen die zorgen dat URL's (voor de http-aanvragen naar de API) afgehandeld worden. Voorlopig worden enkel GET-requests afgehandeld. Data opslaan of updaten in de databank is dus nog niet mogelijk via onze REST API. De centrale klasse in dit systeem is de applicatieklasse (welke overerft van Application) 'ApplicationConfig'. Deze beheert de filters, de exceptionMapper en de facadeREST klassen. Het URL-pad van deze klasse is **/**api. Paden worden geregistreerd door de annotatie @Path.

In de 'util' package bevinden zich deze filters en de exceptionmapper. Filters worden gebruikt om requests en responses te filteren en te manipuleren. De REST API maakt gebruik van 2 filters en 1 exceptionmapper.

De CORSfilter filtert de responses en zorgt dat Cross Origin Resource-Sharing mogelijk wordt door headers aan de responses toe te voegen. CORS is nodig om andere domeinen toegang te verlenen tot de bronnen die de REST API aanbiedt. De AuthorizationFilter filtert aanvragen en controleert de X-API-KEY-header (zie bovenstaand deel over authenticatie).

De CustomWebApplicationExceptionMapper linkt webapplicatie-excepties aan een deftig antwoord met een overeenkomende http-statuscode en -statusbericht. Deze kan omgevormd worden naar een CustomExceptionMapper indien je ook de foutafhandeling van andere excepties en zelfgemaakte excepties hiermee wilt verwerken.

De FacadeREST-klassen erven allemaal over van AbstractFacade. Deze klassen staan in voor de afhandeling van de GET-requests en specifiëren elk hen eigen URL-pad. Zo zal de bijvoorbeeld de klasse ProviderFacadeREST afgebeeld worden op het /providers pad. Iedere methode in de klasse die een GET-request afhandelt voor dit pad zal op zijn beurt een @GET-annotatie, een @Path-annotatie en een @Produces({“application/json”}) hebben. Deze laatste annotatie registreert JSON als het return-type voor het http-antwoord van de methode. De parameters van deze methoden zijn opnieuw geannoteerde Objecten, zoals @Context of @PathParam. Documentatie over alle mogelijke objecttypes is eenvoudig te vinden. De REST API maakt gebruik van @RequestParam. Hierdoor worden URL-paden begrensd op het einde door een ‘?’ teken en volgt daarna de query-string. Er is voor dit soort URL-pad gekozen omdat dit tot een RESTful ontwerp van de API leidt. Een voorbeeld van een URL wordt dan:

http://verkeer-1.bp.tiwi.be/VerkeerREST/API/trafficdata?providerID=1&routeID=1

Bovengenoemde methoden in de FacadeREST-klassen handelen de Query’s af. Via Query.getResultList() kan je de resultaten van een query en de returnwaarde van deze methode ophalen.

Naast objecten als returnwaarde kan je ook Strings teruggeven bij de aanvragen. Dit zorgt ervoor dat je complexere JSON-data en ook foutmeldingen kan teruggeven aan de gebruiker door gebruik te maken van een StringBuilder die eenvoudig te manipuleren is. In de REST API wordt dit vooral gebruikt in de klasse TrafficFacadeREST. Zoals hierboven vermeld, worden foutmeldingen beter opgeworpen en afgehandeld door antwoorden door te geven via de ExceptionMapper in plaats van deze in JSON te verwerken.

De laatste package, ‘simpledomain’, bevat de voorstellingen van de objecten die opgehaald worden uit de databank met behulp van query’s. Dit gebeurt als de objecten niet kunnen weergegeven worden door de objecten die zich reeds in domain bevinden.

Mogelijke requests

Het domein van onze applicatie en de context van onze REST API is verkeer-1.bp.tiwi.be. Alle aanvragen moeten dus voorafgegaan worden door deze URL. Hieronder wordt telkens een opsomming en verklaring van de mogelijke parameters samen met de output van een aanvraag gegeven. De output gebeurt in een JSON-formaat. Variabelen (parameters) worden meegegeven met query strings die zijn voorafgegaan door een '?'. Query strings zijn van de vorm naam_variabele=[waarde_variabele]. Meerdere query strings worden van elkaar gescheiden door een '&'.

Bij alle mogelijke aanvragen is enkel de methode GET voorzien. Indien een aanvraag geslaagd is (code 200) dan is dit ook zichtbaar in de JSON door het 'result'-object "success" mee te geven. Bij foutmeldingen is er een onderscheid gemaakt tussen logische fouten (bv. onbestaand ID) die nog steeds de http-status code 200 hebben, maar waarbij het 'result'-object "error" is, en de standaardfouten (bad request, not found, …). Deze standaardfouten kunnen door alle bronnen veroorzaakt worden, en worden dus door de exceptionMapper verwerkt.

Voorbeelden van de standaardfouten

GET http://verkeer-1.bp.tiwi.be/api/providers?id=1 HTTP/1.1

HTTP/1.1 401 (Unauthorized)
"Unauthorized. Authorization header is required to perform this request."

GET http://verkeer-1.bp.tiwi.be/api/providers?id=1 HTTP/1.1

HTTP/1.1 401 (Unauthorized)
"6qKKfkX7u2lmJqxd8RrpLk7 is an invalid API-key. Use a valid API-key."

GET http://verkeer-1.bp.tiwi.be/api/wrongpath HTTP/1.1

HTTP/1.1 404 (Not Found)
"The requested resource does not exist. Change the url-path before trying again."

GET http://verkeer-1.bp.tiwi.be/api/providers?id=woord HTTP/1.1

HTTP/1.1 400 (Bad Request)
"One of the given query-parameter values is invalid. Change the incorrect query-parameter before trying again."

Providers opvragen

Het opvragen van 1 of alle providers.

Relatief eindpunt: GET /api/providers

Query parameters

indien geen parameters worden meegegeven, geeft de aanvraag alle providers terug.

ID	ID van de op te vragen provider (Optioneel)
Name	Naam van de op te vragen provider (Optioneel)

Voorbeeld van een antwoord

```
GET http://verkeer-1.bp.tiwi.be/api/providers?id=1 HTTP/1.1

HTTP/1.1 200 (ok)
{
	"result": "success",
	"data":
	{
		"id": 1,
		"name": "Waze",
		"weight": 1
	}
}
```

Routes opvragen

Het opvragen van 1 of alle routes.

Relatief eindpunt: GET /api/routes

Query parameters

indien geen parameters worden meegegeven, geeft de aanvraag alle routes terug.

ID	ID van de op te vragen route (Optioneel)
Name	Naam van de op te vragen route (Optioneel)

Voorbeeld van een antwoord

```
GET http://verkeer-1.bp.tiwi.be/api/routes?id=1 HTTP/1.1

HTTP/1.1 200 (ok)
{
	"result": "success",
	"data":
	{
		"description": "Palinghuizen - Neuseplein",
		"endlat": 51.067521106257,
		"endlong": 3.7276444075241,
		"id": 1,
		"length": 2183,
		"name": "Gasmeterlaan (R40) eastbound",
		"startlat": 51.06632352,
		"startlong": 3.69968998,
		"speedLimit": 30
	}
}
```

Waypoints opvragen

Het opvragen van de waypoints voor 1 of alle routes.

Relatief eindpunt: GET /api/waypoints

Query parameters

indien geen parameters worden meegegeven, geeft de request de waypoints voor alle routes terug.

routeID	ID van de op te vragen provider (Optioneel)

Voorbeeld van een antwoord

```
GET http://verkeer-1.bp.tiwi.be/api/waypoints?routeID=1 HTTP/1.1

HTTP/1.1 200 (ok)
{
	"result": "success",
	"data":
	[
		{
			"latitude": 51.06629,
			"longitude": 3.69971,
			"routeID": 1,
			"sequence": 0
		},
		{
			… // hier komt telkens een volgende waypoint
		}
	]
}
```

Standaard trafficdata opvragen

Het opvragen van de geaggregeerde data (wordt vast geaggregeerd over 5 minuten in de databank, dit om dataopslag te beperken) voor een bepaalde route van een bepaalde provider.
De data bestaat uit een timestamp, een reistijd en een gemiddelde reistijd.

Relatief eindpunt: GET /api/trafficdata

Query parameters

providerID	ID van de provider waarvoor je data wil ophalen (Verplicht)
routeID	ID van de route waarvoor je data wil ophalen (Verplicht)
from	Startdatum van de periode waar je data wil voor ophalen (Optioneel) formaat: yyyy-MM-dd HH:mm:ss (default: 1900-01-01 00:00:00)
to	Einddatum van de periode waar je data wil voor ophalen (Optioneel) formaat: yyyy-MM-dd HH:mm:ss (default: huidige tijdstip)
interval	Het interval tussen timestamps in de gevraagde periode(Optioneel) uitgedrukt in minuten (default: 15 minuten)

Voorbeeld van een antwoord

```
GET http://verkeer-1.bp.tiwi.be/api/trafficdata?routeID=1&providerID=1
                &from=2016-04-12 21:00:00&to=2016-04-12 21:15:00 HTTP/1.1

HTTP/1.1 200 (ok)
{
    "result": "success",
    "data":
    {
        "2016-04-12 21:00:00.0":
        {
            "traveltime": 269,
            "average": 267
        },
        "2016-04-12 21:15:00.0":
        {
            "traveltime": 326,
            "average": 288
        }
    }
}
```

Live trafficdata opvragen

Het opvragen van de laatst opgehaalde data voor een bepaalde provider en een bepaalde route. Geeft hierbij de datum van de data (datum van ophalen bij provider, niet van opvragen van gebruiker aan API), de snelheid en de tijd van het traject. Naast deze live data wordt ook een gemiddelde teruggegeven voor de data rond dezelfde tijd als deze live data. Omdat er een query wordt gebruikt die ook weekdagen teruggeeft, wordt ook de huidige dag teruggegeven. (0 = maandag, …, 6 = zondag)

Relatief eindpunt: GET /api/trafficdata/live

Query parameters

providerID	ID van de provider waarvoor je data wil ophalen (Verplicht)
routeID	ID van de route waarvoor je data wil ophalen (Verplicht)
interval	Het aantal minuten voor en na de timestamp van de meest recente live-data waarvoor andere data mag gebruikt worden om de gemiddelde tijd te berekenen (Optioneel) (default: 15 minuten)
period	het aantal dagen voor de timestamp van de meest recente live-data waarvan de data mag gebruikt worden om de gemiddelde tijden te berekenen (Optioneel) (default: 30 dagen)

Voorbeeld van een antwoord

```
GET http://verkeer-1.bp.tiwi.be/api/trafficdata/live?routeID=1&providerID=1 HTTP/1.1

HTTP/1.1 200 (ok)
{
	"result": "success",
	"data":
	{
	"1":
		{
			"live":
			{
				"createdOn": "2016-05-14 15:35:48.0",
				"speed": "27.7",
				"time": "4.733333333333333"
			},
			"avg":
			{
				"speed": "29.5",
				"time": "4.433333333333334"
			}
		}
	}
}
```

Trafficdata voor weekdagen opvragen

Het opvragen van de data voor 1 bepaalde weekdag of alle weekdagen. (Hierbij is waarde 0 = maandag, …, 6 = zondag) Deze functie is aan te raden om grafieken op te stellen.

Relatieve eindpunt: GET /api/trafficdata/weekday

Query parameters

Merk op dat bij het opvragen van weekdagen, providerID en routeID niet verplicht is. Het is dus mogelijk de data te gebruiken van alle providers en/of van alle routes.

providerID	ID van de provider waarvoor je data wil ophalen (Optioneel)
routeID	ID van de route waarvoor je data wil ophalen (Optioneel)
from	Startdatum van de periode waarvoor je data wil ophalen (Optioneel) formaat: yyyy-MM-dd HH:mm:ss (default: 1900-01-01 00:00:00)
to	Einddatum van de periode waarvoor je data wil ophalen (Optioneel) formaat: yyyy-MM-dd HH:mm:ss (default: huidig tijdstip)
interval	Het interval tussen de timestamps in de gevraagde periode (Optioneel) uitgedrukt in minuten (default: 15 minuten)
weekday	Enkel nodig als je uitsluitend data van een bepaalde weekdag (Optioneel) wil ophalen. De waarde is een getal tussen 0 en 6

Voorbeeld van een antwoord

```
GET http://verkeer-1.bp.tiwi.be/api/trafficdata/weekday?routeID=1&providerID=1
                &weekday=1 HTTP/1.1

HTTP/1.1 200 (ok)
{
        "result": "success",
        "data":
        {
                "0": {},
                "1":
                {
                        "00:00":
                        {
                                "traveltime": 230.2381,
                                "average": 230.0476
                        },
                        "00:15":
                        {
                                "traveltime": 230.619,
                                "average": 230.8571
                        },
                        …
                },
                …
        }
}
```

Trafficdata voor een interval opvragen

Het opvragen van de data voor een bepaald interval en voor een bepaalde provider. Hierbij wordt voor iedere route van de provider de gemiddelde reistijd en snelheid teruggegeven. Deze functie is ook aan te raden voor grafieken op te stellen.

Relatieve eindpunt: GET /api/trafficdata/interval

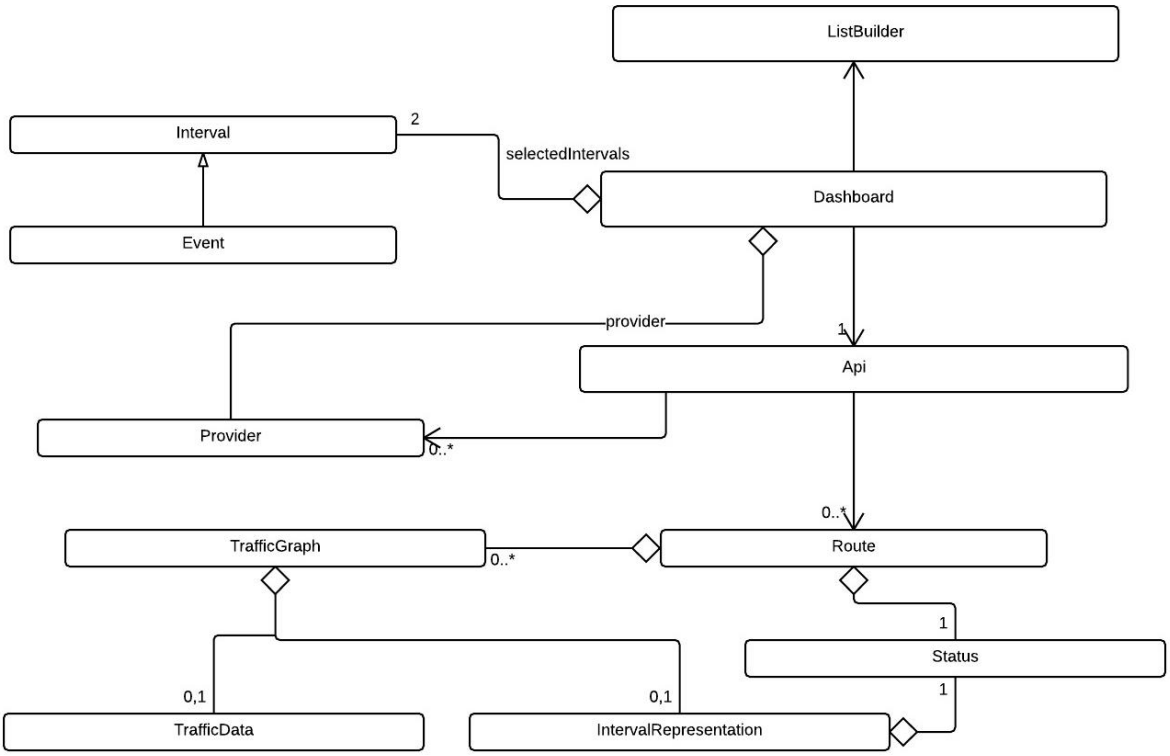
Query parameters

providerID	ID van de provider waarvoor je data wil ophalen (Verplicht)
from	Startdatum van de periode waarvoor je data wil gebruiken (Optioneel) om het gemiddelde te bepalen formaat: yyyy-MM-dd HH:mm:ss (default: 1900-01-01 00:00:00)
to	Einddatum van de periode waarvoor je data wil gebruiken (Optioneel) om het gemiddelde te bepalen formaat: yyyy-MM-dd HH:MM:ss (default: huidig tijdstip)
interval	Het interval tussen de timestamps in de gevraagde periode (Optioneel) uitgedrukt in minuten (default: 15 minuten)

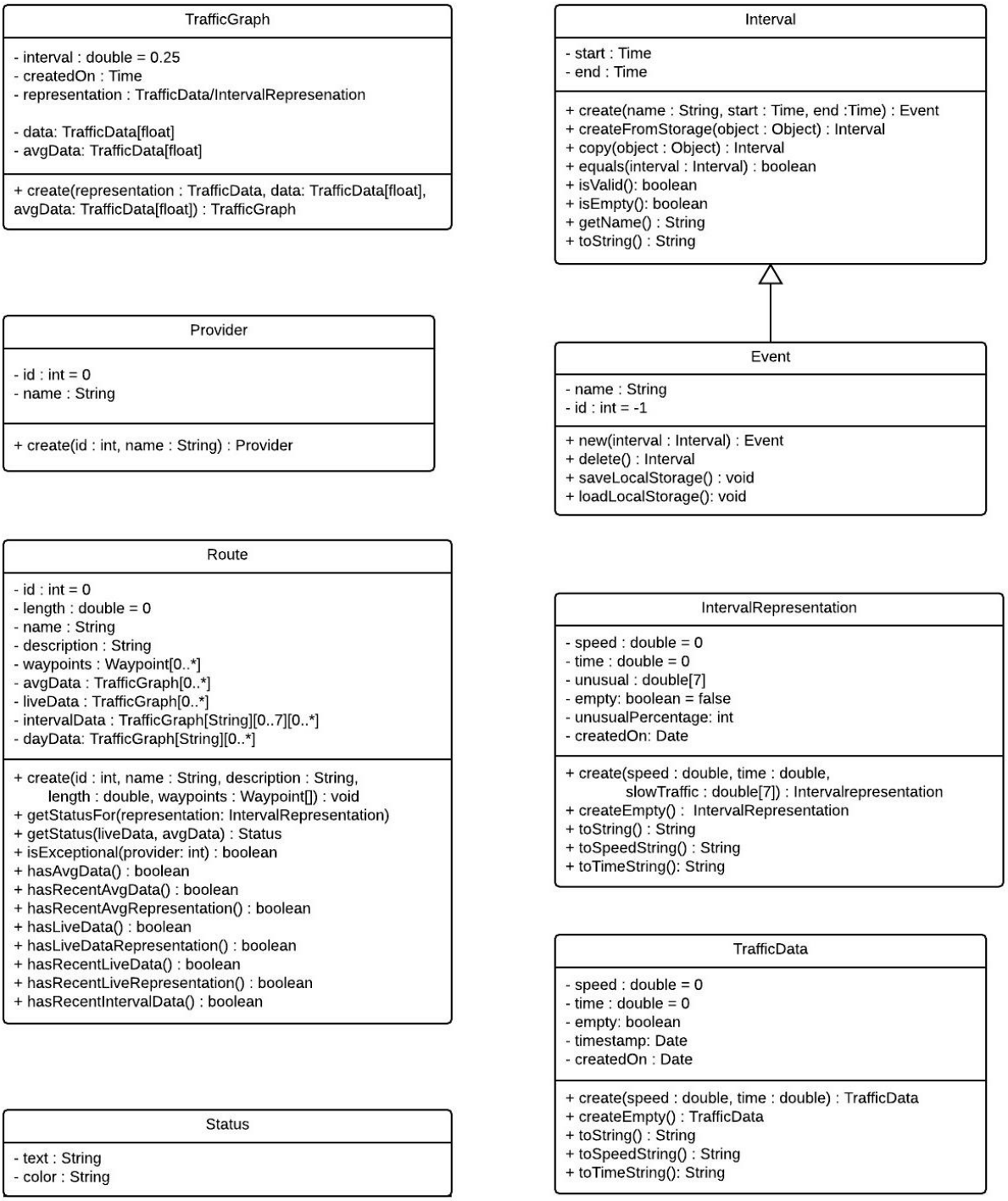
Voorbeeld van een antwoord

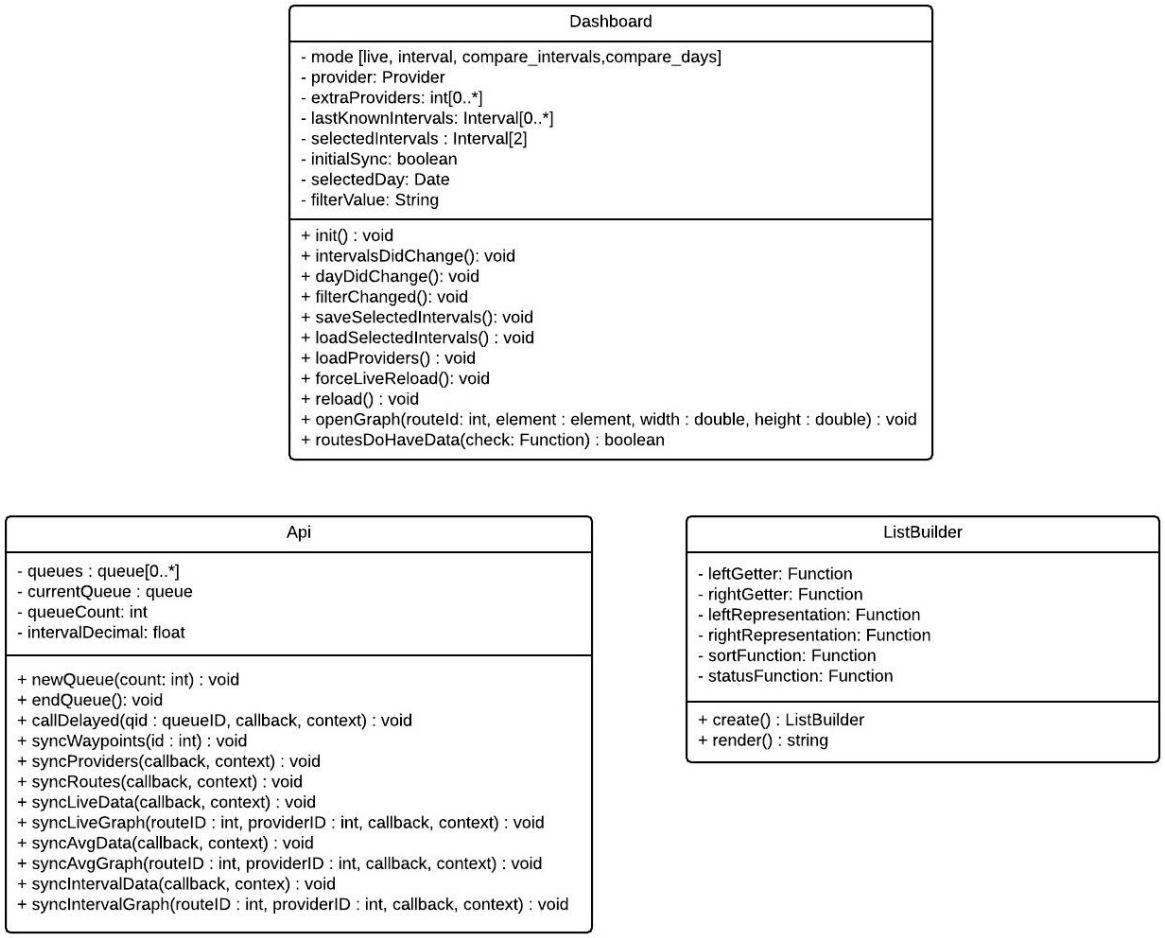
```
GET http://verkeer-1.bp.tiwi.be/api/trafficdata/interval?providerID=1 HTTP/1.1

HTTP/1.1 200 (ok)
{
	"result": "success",
	"data":
	{
		"1":
		{
			"speed": "33",
			"time": "245",
			"slow": "41"
		},
		{
			…
		}
	}
}
```



Figuur 11: klassendiagram javascript


TrafficGraph
- interval : double = 0.25
- createdOn : Time
- representation : TrafficData/IntervalRepresenation
- data: TrafficData[float]
- avgData: TrafficData[float]
+ create(representation : TrafficData, data: TrafficData[float], avgData: TrafficData[float]) : TrafficGraph
Interval
- start : Time
- end : Time
+ create(name : String, start : Time, end :Time) : Event
+ createFromStorage(object : Object) : Interval
+ copy(object : Object) : Interval
+ equals(interval : Interval) : boolean
+ isValid(): boolean
+ isEmpty(): boolean
+ getName() : String
+ toString() : String
Provider
- id : int = 0
- name : String
+ create(id : int, name : String) : Provider
Event
- name : String
- id : int = -1
+ new(interval : Interval) : Event
+ delete() : Interval
+ saveLocalStorage() : void
+ loadLocalStorage(): void
Route
- id : int = 0
- length : double = 0
- name : String
- description : String
- waypoints : Waypoint[0..*]
- avgData : TrafficGraph[0..*]
- liveData : TrafficGraph[0..*]
- intervalData : TrafficGraph[String][0..7][0..*]
- dayData: TrafficGraph[String][0..*]
+ create(id : int, name : String, description : String, length : double, waypoints : Waypoint[]) : void
+ getStatusFor(representation: IntervalRepresentation)
+ getStatus(liveData, avgData) : Status
+ isExceptional(provider: int) : boolean
+ hasAvgData() : boolean
+ hasRecentAvgData() : boolean
+ hasRecentAvgRepresentation() : boolean
+ hasLiveData() : boolean
+ hasLiveDataRepresentation() : boolean
+ hasRecentLiveData() : boolean
+ hasRecentLiveRepresentation() : boolean
+ hasRecentIntervalData() : boolean
IntervalRepresentation
- speed : double = 0
- time : double = 0
- unusual : double[7]
- empty: boolean = false
- unusualPercentage: int
- createdOn: Date
+ create(speed : double, time : double, slowTraffic : double[7]) : Intervalrepresentation
+ createEmpty() : IntervalRepresentation
+ toString() : String
+ toSpeedString() : String
+ toTimeString(): String
Status
- text : String
- color : String
TrafficData
- speed : double = 0
- time : double = 0
- empty: boolean
- timestamp: Date
- createdOn : Date
+ create(speed : double, time : double) : TrafficData
+ createEmpty() : TrafficData
+ toString() : String
+ toSpeedString() : String
+ toTimeString(): String



Figuur 12: klassendiagram javascript gedetailleerd

Webapplicatie

De webapplicatie vinden we in /VerkeerREST/web. Het bestaat uit slechts één statisch html-bestand. Daarin wordt met javascript een dashboard geladen waarin onder andere een kaart van Gent wordt weergegeven.

Het is niet doenbaar om de structuur van alle klassen in detail uit te leggen, maar de belangrijkste methodes vinden we terug in het UML-diagram. Wel leggen we hieronder kort uit hoe het de javascript code werkt. In de javascript-bestanden staat in commentaar altijd duidelijk uitgelegd wat de methodes en de code precies doen.

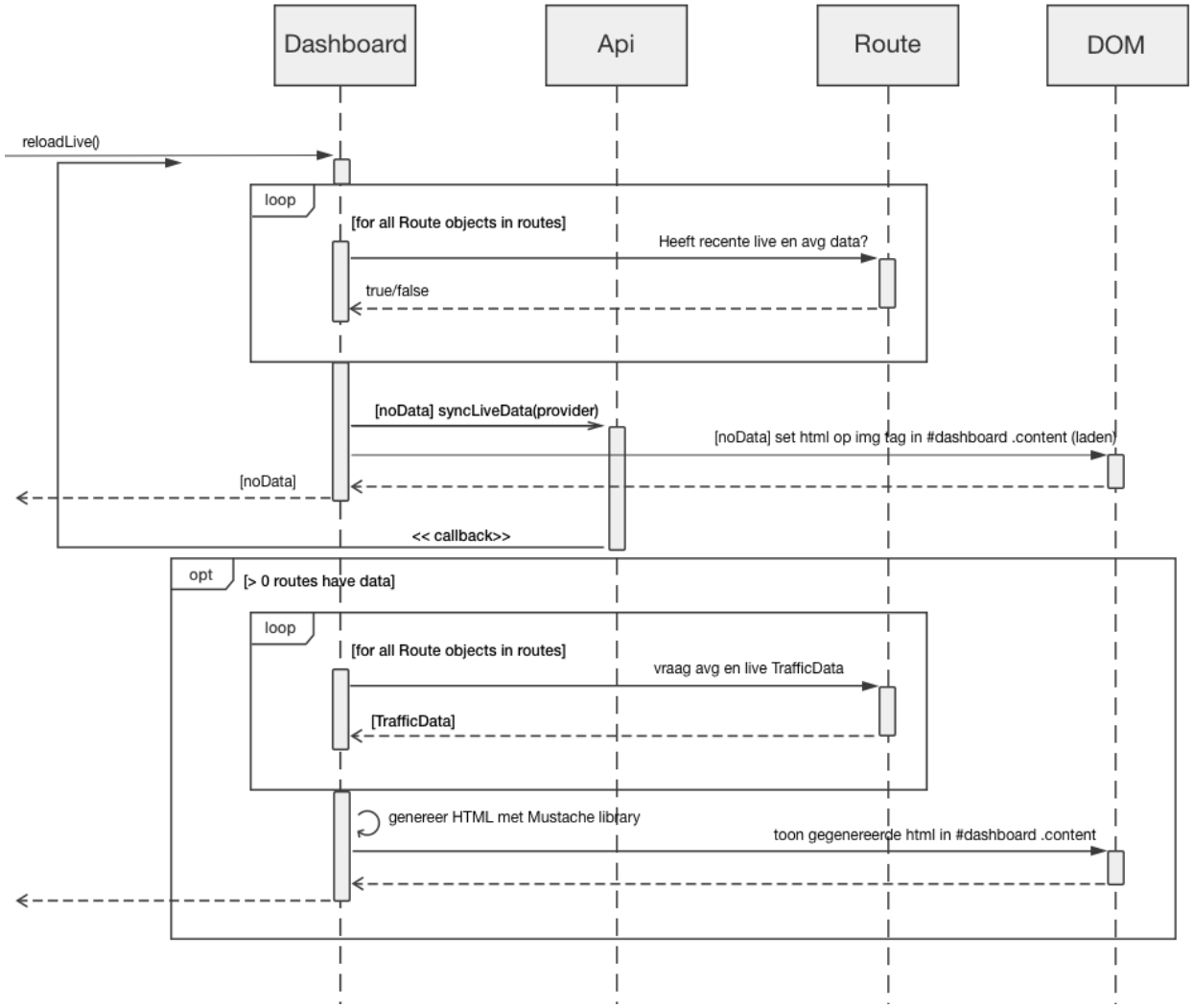
Werking

Als de pagina geladen is, roepen we Dashboard.init() aan. Deze zorgt ervoor dat we onze routes en onze providers als eerste laden en weergeven. Daarna wordt Dashboard.reload() opgeroepen, dit is een methode die kijkt welk dashboard we open hebben (Vandaag, Periode, …) en voert specifieke code uit die de html van het uitklapbare deel opnieuw genereert. Hiervoor wordt de data zoveel mogelijk uit reeds opgehaalde informatie van onze API opgehaald: deze worden opgeslagen in routes - een array van Route-objecten. De reeds opgehaalde informatie die verband houdt met een route, wordt door deze route bijgehouden. Zo maken we geen overbodige API aanvragen.

Indien niet voldoende informatie beschikbaar is, of als deze is te oud, dan gaan we via het Api object nieuwe informatie van de server vragen. Hierbij geven we een callback-methode en context (het object dat de callback zal uitvoeren, en dus het ‘this’-object is in de callback) terug, zodat we op de hoogte worden gebracht als onze asynchrone aanvragen klaar zijn. Hierna kunnen we weer een reload uitvoeren, maar deze keer zal er wel data beschikbaar zijn.

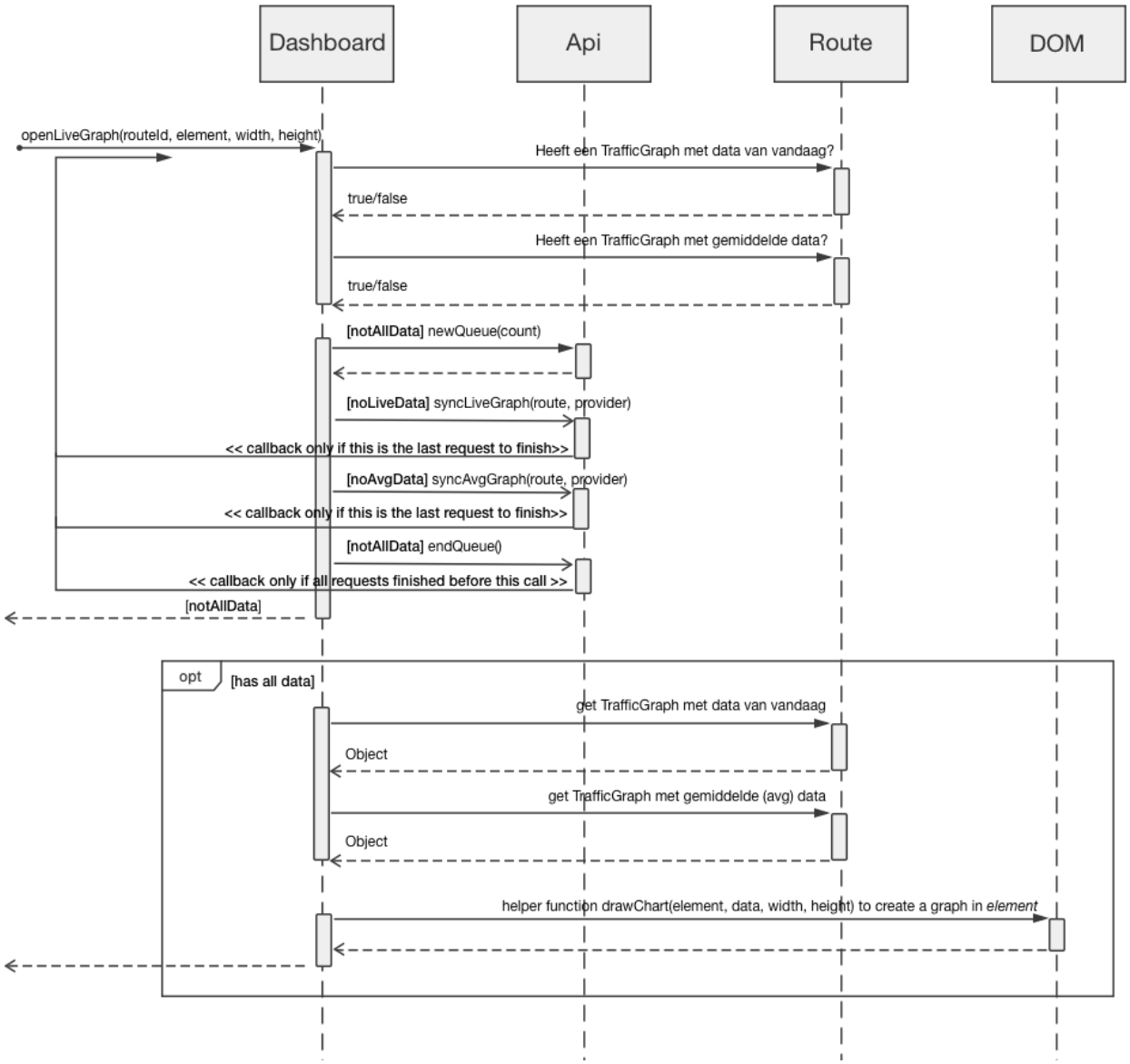
De Api-klasse heeft ook een mogelijkheid om de callback pas uit te voeren als een groep aanvragen klaar is. Hiervoor kunnen we een queue aanmaken (zie Api.js).

Het sequentiediagram van reloadLive() van de Dashboard klasse vind je hieronder. Dit is een van de methodes die door de reload() functie aangeroepen wordt, afhankelijk van de geselecteerde optie in het linker menu (Vandaag, Periode, Periode vergelijken, …). De andere methodes zijn gelijkaardig.



Figuur 13: sequentiediagram

Het sequentiediagram van openLiveGraph() van de Dashboard klasse vind je hieronder. Dit is een van de methodes die door de openGraph() functie aangeroepen wordt, afhankelijk van de geselecteerde optie in het linker menu (Vandaag, Periode, Periode vergelijken, …). De andere methodes zijn gelijkaardig. Als de gebruiker op een route drukt zorgt de globale functie toggleGraph() (in stats.js) dat er een vak openklapt met daarin een draaiend icoontje (bezig met laden). Daarna roept deze de methode Dashboard.openGraph() aan en specifieert hierin in welk element er een grafiek moet komen. Daarna vindt onderstaande sequentiediagram plaats.



Figuur 14: sequentiediagram

In de map /javascript vind je onder andere:

- Prototypes.js:
  Dit bestand bevat alle objectklassen, met uitzondering van de Api en het Dashboard.
- Api.js:
  Dit bestand bevat de Api-klasse (en singleton). Deze is verantwoordelijk voor de communicatie met de server.
- DummyApi.js:
  Dit bestand bevat een klasse die werd gebruikt om het dashboard te testen zonder de Api. Deze genereert volledig willekeurige data en simuleert een vertraging per aanvraag. Deze wordt geactiveerd met de eenvoudige javascript-code 'Api = DummyApi' op het einde van het Api.js bestand.
- Dashboard.js:
  Dit bestand bevat de Dashboard-klasse (en singleton). De reload methode herlaadt het dashboard als er bv. een andere mode wordt gekozen (Vandaag, Periode, …) of als van provider wordt veranderd. Als niet voldoende data beschikbaar is, vraagt dit singleton aan de Api-klasse voor meer informatie en toont het voorlopig een draaiend icoontje.
- Map.js:
  Dit bestand bevat alles om een laag met routes over een kaart van Google (via de javascript API van Google maps) te tekenen. Hier worden ook pop-ups gegenereerd als op een route wordt geklikt.

SASS

Met SASS - meer bepaald SCSS - is het mogelijk om CSS-code te schrijven die eenvoudiger kan onderhouden worden. Dit komt onder andere door gebruik te maken van variabelen (hierdoor hoeft men bijvoorbeeld kleuren slechts 1 keer definiëren) en de mogelijkheid van CSS-code te nesten. Ook kan de SCSS-code verspreid worden over verschillende bestanden, terwijl deze in 1 CSS-bestand wordt gecompileerd. Dit voorkomt veel overbodige http-aanvragen en het is dit bestand dat in de HTML-code ingeladen wordt.

Meer informatie hierover kan men vinden op *http://sass-lang.com*

Libraries

Voor grafieken wordt de Google Charts Api (*https://developers.google.com/chart/*) gebruikt en voor de kaart de Google Maps Api (*https://developers.google.com/maps/*). Beide javascript libraries zijn erg goed gedocumenteerd door Google zelf. Voor templating wordt Mustache gebruikt (https://mustache.github.io/).

INSTALLATIEHANDLEIDING

Onze applicatie bestaat uit 2 delen: een webapplicatie -die voor de web interface en de REST API zorgt - en de polling-applicatie: deze zorgt dat we de data van de verschillende providers elke 5 minuten opvragen en opslaan. Dit stuk legt uit hoe we deze applicaties compileren en op een server kunnen plaatsen.

Onze repository (onze source code via git) is beschikbaar op *https://github.ugent.be/iii-vop2016/verkeer-1* Hiervoor moet eerst toegang gegeven worden. Daarna kan ze worden gedownload naar de eigen computer.

Development toestel

- Netbeans
- Java 7 of hoger
- Zorg dat Perl op je eigen computer staat geïnstalleerd
- Installeer SASS op je eigen computer. Hoe dit moet vind je op http://sass-lang.com/install
- Mac OSX: Installatie met sudo moet vermeden worden, dit kan voor problemen zorgen. Gebruik bij voorkeur rbenv om een nieuwe ruby environment aan te maken waarin je de SASS gem zonder sudo kan installeren.
- Windows gebruikers hebben ook een bash nodig (BourneAgain Shell, dit is een tekstuele console waar Unix-commando's kunnen ingevoerd worden). Een mogelijkheid is 'Cygwin64 Terminal': download de cygwin64 terminal: (*https://www.cygwin.com/)* kies voor de opties ->net->ssh.

Server

- Ubuntu 14.04.4 of een gelijkaardige linux distributie
- MySQL moet zijn geïnstalleerd. Wij gebruiken versie 5.5.47, andere versies zouden normaal gezien ook moeten werken, tenzij deze te veel afwijken van onze gebruikte versie. MariaDB werkt ook.
- Perl v5.18 of nieuwer. Komt standaard met Ubuntu. Kan geïnstalleerd worden of geüpdatete worden met “sudo apt-get install perl”
- De JSON library voor Perl moet geïnstalleerd zijn. Als het commando “perl -MJSON -e 1” geen fouten geeft, is deze al geïnstalleerd. Anders kan dit via het commando “sudo perl -MCPAN -e 'install JSON'”
- Glassfish
- Java
- Het screen commando. Installeren kan via het commando “sudo apt-get install screen”. Meer informatie kan u vinden op *https://www.digitalocean.com/community/tutorials/how-to-install-and-use-screen-on-an-ubuntu-cloud-server*

Voor volgende instructies is het aangeraden om een ssh-key toe te voegen aan de server voor je verder gaat.

Database aanmaken

- Open een terminal (of cygwin64 terminal in Windows) venster in de scripts map van onze repository (navigeer hierheen met het cd commando).
- Voer het commando "./init-database.sh $ipadres $mysqlUsername $mysqlPassword" uit, vul hierbij eerst het IP-adres of domeinnaam van de server in en de MySQL-logingegevens. Dit moet het root account zijn. Bijvoorbeeld "./init-database.sh 146.185.150.100 root wachtwoord". Indien een fout 'permission denied' voorkomt, probeer dan eerst "chmod 500 init-database.sh" uit te voeren en probeer het nog eens opnieuw. **Dit script zal het root wachtwoord veranderen in root.** Dit is voorlopig noodzakelijk omdat dit zo in onze applicatie configuratie opgeslagen zit.
  Voer dit script ook slechts 1 keer uit, aangezien het mogelijk data kan wissen als de database al eens aangemaakt is geweest.

Polling-applicatie compileren en starten op een server

- Open het Verkeer project uit deze repository (Open Netbeans> bestand > open project en selecteer /Verkeer)
- Klik op het hamertje met de bezem. Wacht tot het compileren klaar is. De gecompileerde bestanden en andere scripts vinden we nu in /Verkeer/dist.
- Open een terminal (of cygwin64 terminal in Windows) venster in de scripts map van onze repository (navigeer hierheen met het cd commando).
- Voer het commando "./deploy-polling.sh $ipadres" uit, vul hierbij eerst het IP-adres of domeinnaam van de server in, bijvoorbeeld "./deploy-polling.sh 146.185.150.100". Indien een fout 'permission denied' voorkomt, probeer dan eerst "chmod 500 deploy-polling.sh" uit te voeren en probeer het nog eens opnieuw.
- Als dit is gelukt dan staan de bestanden van uit de beide /dist folders op de server in /root/verkeer
- Het opstarten moet voorlopig nog manueel gebeuren
- Open je terminal venster en voer "ssh root@$ipadres" uit
- Gebruik cd /root/verkeer om in de map te belanden met onze jar.
- Voer "screen java -Xmx256m -jar ./Verkeer.jar" uit om onze jar uit te voeren in een nieuwe 'venster'.
- Nu zie je de output van het programma en kan je een 'poll' forceren met het commando poll.
- Je kan de status zien met het commando 'status'.
- Om dit in de achtergrond te houden is het heel belangrijk om het volgende te doen: druk ctrl + a (hiervan zal je geen visuele feedback zien) en druk daarna op de 'd' toets. Daarna zal je terug in de terminal belanden.
- Gebruik "screen -ls" om te zien of de polling applicatie draait. Normaal gezien staat '(detached)' achter verkeer-1.
- Gebruik "screen -r" om terug in de console van onze applicatie te gaan. Het kan zijn dat om een id wordt gevraagd, die bekom je door "screen -ls".
- **Vergeet ook hier niet om ctrl + a, d te gebruiken om terug te keren.**
- We willen dit process later vereenvoudigen en in ons deploy-polling.sh script plaatsen, maar dit is ons momenteel nog niet gelukt. Onze webapplicatie deployen verloopt wel volledig automatisch.

Webapplicatie starten op Glassfish server

- Open een nieuw terminal venster (of cygwin64) en gebruik het cd commando om naar de map /scripts uit onze repository te navigeren.
- Voer daar “perl sass.pl” uit. Dit script compileert onze scss-code aangezien we onze CSS-code (bewust) niet in onze repository opslaan. Het commando geeft 'klaar' als alles gelukt is. Los anders de aangegeven problemen op.
- Zorg ervoor dat Netbeans geïnstalleerd is op je eigen computer, en open VerkeerREST uit onze repository.
- Klik op het hamer symbool met de bezem erbij. Het project zou nu zonder problemen moeten compileren en in de map /VerkeerREST/dist geplaatst worden: VerkeerREST.war Het is dit bestand dat we straks zullen deployen op Glassfish (automatisch met scripts).
- Open terug hetzelfde terminal venster, en voer “./deploy-rest.sh $ipadres $gebruikersnaam $wachtwoord” uit, vul hierbij eerst het IP-adres of domeinnaam van de server in en de Glassfish-gebruikersnaam met zijn wachtwoord zoals bijvoorbeeld “./deploy-rest.sh 146.185.150.100 admin aeSqFPbpUI” voor onze DigitalOcean server. Indien een fout 'permission denied' voorkomt, voer dan eerst “chmod 500 deploy-rest.sh” uit te voeren en probeer het nog eens opnieuw.

Als de installatie is gelukt, dan kan je naar de server surfen via *http://mijndomein.com/VerkeerREST/* om het verkeerscentrum te bekijken.

TESTPLAN

Het testplan bestaat uit 3 delen: een overzicht van alle testen en hun implementatiestatus, de testen voor de backend, en die voor de front-end. De laatste 2 delen zijn telkens opgedeeld in unit, integration, en usability testen per onderdeel van de applicatie. Er staan dus delen bij waar bepaalde testen niet voor bestaan.

Overzicht

Onderdeel	Type	korte omschrijving	beschikbaar
Backend	Unit	Ophalen data van providers (Coyote, Here…)	ja
Backend	Unit	Opslaan data in databank	ja
Backend	Unit	Functionaliteit van console-input onderhouden	nee
Backend	Integration	Interactie providers en databank	ja
Backend	Integration	Interactie polling-klasse en providers	nee
Backend	Integration	Interactie console-input en polling-klasse	nee
Front-end	Unit	Werking REST-API	nee
Front-end	Unit	Werking Dashboard-onderdelen	ja*
Front-end	Integration	Communicatie tussen REST-API en Dashboard	ja*
Front-end	Integration	Communicatie tussen de onderdelen van het Dashboard	ja*
Front-end	Usability	Gebruikers(groep) het Dashboard laten testen	ja*

*: test gebeurt manueel en niet met specifieke code

Backend

Dit deel gaat over de testen die worden uitgevoerd op de ophaling van de data bij de providers en het wegschrijven hiervan in de databank.

Unit testen

Provider-klassen

Locatie van testen:

project: 'Verkeer'
pad: test/connectors/provider/*.java

Voor de klassen die de data ophalen bij providers (afgeleide klassen van *AProviderConnector* zoals bijvoorbeeld *GoogleProviderConnector*) schrijft men best enkel algemene testen die de gehele werking van de klasse controleren. Dit omdat enkel deze methode publiek naar buiten moet gebracht worden.

Momenteel bestaat er per connector-klasse een test-klasse met 1 methode (triggerTest) die de triggerUpdate-methode gaat aanroepen van een provider. Deze methode kijkt eenvoudigweg of er in een dummy-databank het juiste aantal records zijn toegevoegd door de connector-klasse. De test zal falen wanneer er geen records zijn toegevoegd of als er een exceptie wordt opgeworpen.

Bij het falen van deze test kan de ontwikkelaar dan in zijn klasse gaan kijken waar de fout zit. Aangezien het ontbreken van records normaal gezien betekent dat er ergens een fout in het ophalen van data zit, zal deze test bijna altijd falen door een exceptie die opgeworpen wordt bij communicatie met de externe provider. Deze exceptie kan men dan gebruiken om het probleem doelgericht op te lossen. Zo zal bij de *GoogleProviderConnector* de foutmelding die Google teruggeeft getoond worden. Is er geen exceptie opgeworpen, dan is de kans groot dat er een implementatiefout zit in het overlopen van de routes.

Deze testen worden het best manueel uitgevoerd bij het toevoegen of aanpassen van een provider, maar ook indien het log in productie of op de testserver problemen aantoont.

Databank-klassen

Locatie van testen:

project: 'Verkeer'
pad: test/connectors/database/*.java

Voor de klassen die de data opslaan in een databank (implementerende klassen van *IDbConnector* zoals bijvoorbeeld *MariaDbConnector*) schrijft men best voor elke methode die van de interface komt een test. Deze methodes zijn de enige methodes die mogen gebruikt worden door andere klassen.

Zo worden in MaraDbConnector alle insert- en delete-methodes in 1 test uitgevoerd en verder testen we ook elke zoekmethode (IDbConnector.findXXXbyYY(value)) in een aparte test

De testen hiervoor worden best manueel uitgevoerd bij toevoegen of aanpassen van een databank of als er een foutmelding in het applicatielog voorkomt over de databank connectie.

Polling-klasse

Het zelfstandig testen van deze klasse heeft niet veel zin, want deze stelt een eenvoudige achtergrondthread voor die periodiek de provider-klassen laat werken. Hiervoor worden dan ook geen unit-testen voorzien maar wel integration testen.

Console-klasse

Locatie van testen: testen niet beschikbaar

Voor de console-parser (klasse *ConsoleParser*) schrijft men enkel testen waarbij men nagaat of de verschillende commando's kunnen aangeroepen worden. Deze worden dan bij ontwikkeling of uitbreiding van de parser manueel uitgevoerd om er zeker van te zijn dat de vorige commando's beschikbaar blijven.

Integration testen

Samenwerking provider- en databank-klassen

Locatie van testen:

project: 'Verkeer'
pad: test/connectors/provider/*.java

Om deze samenwerking te testen, kan men de unit-testen van de provider herhalen, maar deze keer met een echte databank in plaats van een dummy-databank. Dit zijn dus letterlijk dezelfde testen als de unit-testen van de providers maar de dummy-databank moet vervangen worden door een echte databank.

Deze testen worden best manueel bij elke wijziging of toevoeging van een provider- of databank-klasse uitgevoerd of als er een foutmelding is die niet opgelost kan worden door de unit testen van beide klassen.

Samenwerking de polling-klasse en provider-klassen

Locatie van testen: testen niet beschikbaar

Om deze samenwerking te testen schrijft men een test die de polling-klasse voor een vaste periode laat lopen en kijkt men of de provider-klassen het correcte aantal records hebben opgehaald.

Deze test wordt best manueel uitgevoerd bij elke wijziging in de polling-klasse of bij grote wijzigingen in de provider- of databank-klassen.

Samenwerking console-klasse en polling-klasse

Locatie van testen: testen niet beschikbaar

Om deze samenwerking te testen, laat men de polling klasse lopen en gaat men de unit testen van de console-klasse herhalen, deze keer lettend op de output of de bewerkingen die moeten uitgevoerd worden. Deze testen worden best uitgevoerd bij wijzigingen in 1 van beide klassen.

Usability testen

De backend behoort niet tot wat de gebruiker te zien krijgt. Hiervoor worden dus geen usability-testen voorzien.

Front-end

Unit testen

REST-API

Locatie van testen: testen niet beschikbaar

Het testen van de REST-API kan vlot gebeuren door dummy data uit een test-databank aan deze API te geven en de output hiervan te controleren. De eenvoudigste manier hiervoor is een testscript aan te maken in een van de vele scripting talen (perl, ruby, bash, powershell…). De belangrijkste voorwaarde die moet voldaan zijn door de scripting taal is dat de taal op een bepaalde manier http-berichten moet kunnen sturen en ontvangen.

Deze testen worden best uitgevoerd na elke wijziging aan de API om de consistentie van de output te waarborgen.

Dashboard

Locatie van testen: testen niet beschikbaar

De code van het dashboard bestaat uit 3 grote delen: de kaart, de tekstuele weergave en de code die communiceert met de API:

Het testen van de kaart en tekstuele weergave gaat door dummy-data door te geven aan deze objecten en dan de weergave te controleren. Programmatorisch is het niet eenvoudig haalbaar om dit te controleren. De controle van de data moet hier manueel gebeuren. Zo kan men de kaartweergave controleren door na te gaan of elke route de juiste tijden, kleuren en afstanden weergeeft en de tekstuele weergave door na te gaan of elke route de juiste tijden, kleuren, grafieken weergeeft.

Deze testen worden ook manueel uitgevoerd na elke wijziging aan de API.

De code die communiceert met de API en informatie ter beschikking stelt kan niet losstaand getest worden om duidelijke redenen.

Integration testen

Communicatie tussen REST-API en dashboard

Locatie van testen: testen niet beschikbaar

De communicatie tussen de REST-API en het dashboard kan men eenvoudig testen door de REST-API dummy-data te laten aanbieden. Het dashboard hoeft deze dan gewoon uit te lezen en weergeven. Deze weergave is rechtstreeks af te leiden uit de dummy-data en dus eenduidig bepaald. Deze testen lijken zeer goed op de unit testen van het dashboard, de data komt nu van de REST-API in plaats van de data rechtstreeks in het dashboard te injecteren via een dummy-klasse in javascript. Deze test is programmatorisch niet vlot uitvoerbaar en wordt beter met behulp van een manuele controle uitgevoerd.

Om te simuleren dat de REST-API onbeschikbaar is, kunnen we deze eenvoudigweg in de testomgeving uitschakelen. Deze situatie zou het dashboard correct moeten kunnen opvangen en weergeven.

Communicatie tussen dashboard onderdelen

Locatie van testen: testen niet beschikbaar

Zoals hierboven vermeld bestaat het dashboard uit 3 delen. De kaart en de tekstuele weergave zijn afhankelijk van de code die communiceert met de API. Om te zien of deze 3 delen correct samenwerken kan men de communicerende code een vooraf bepaalde (dummy-)dataset aanbieden die dan een eenduidige output moet geven bij de 2 andere delen. Opnieuw is dit een test die programmatorisch niet vlot uitvoerbaar is en beter met behulp van een manuele controle wordt uitgevoerd.

Usability testen

Voor de eerste regels code van het dashboard werden geschreven, heeft de klant enkele mock-ups ontvangen welke hij dan kon beoordelen en feedback over geven. Na deze feedback zijn we pas beginnen coderen om een eerste echte implementatie naar wens van de klant te voorzien.

Bij de eerste usability test (als het merendeel van de implementatie af was) hebben we de klant en verschillende mensen uit onze omgeving gevraagd van de applicatie te testen en ons op probleempunten te wijzen. Deze opmerkingen waren vrij miniem (icoontjes wat aanpassen, kleine legende toevoegen bij de kleuren die we gebruiken…) en hebben we dan verwerkt in de applicatie.

In het algemeen kunnen we stellen dat bij grondige wijzigingen aan de interface er aan verschillende mensen (vrienden/familie/medestudenten maar ook medewerkers verkeerscentrum Gent…) feedback over de bruikbaarheid van alle gewijzigde functies gevraagd moet worden. Dit gebeurt best zo vroeg mogelijk in het ontwikkelingsproces van die functies om onnodig aanpassingswerk te vermijden. Eventueel kan men de testers een demoversie aanbieden die de functionaliteit simuleert aan de hand van dummy-objecten.

USE CASES

Hieronder zijn de voornaamste use cases vermeld die aanwezig moeten zijn in onze software

Het systeem wenst nieuwe informatie op te halen bij de providers en deze op te slaan in zijn databank

Het systeem probeert periodiek informatie op te halen bij de geïmplementeerde providers, dit gebeurt als volgt:

1. Een klok triggert de methodes in de provider-klassen om data op te halen.
   (ga naar stap 2)
2. De provider -klassen gaan op hun beurt informatie gaan opvragen bij hun externe providers.
   (ga naar stap 3)
3. De provider antwoordt en
   a. Het antwoord bevat nieuwe informatie.
      (ga naar stap 4.a)
   b. Het antwoord bevat een foutmelding.
      (ga naar stap 4.b)
   c. Het antwoord blijft uit en de provider formuleert een foutmelding.
      (ga naar stap 4.b)
4. De provider klasse verwerkt het antwoord:
   a. De informatie wordt omgezet naar een interne datastructuur en doorgegeven aan de databank-klasse.
      (ga naar stap 5)
   b. De foutmelding wordt naar het logbestand weggeschreven en het systeem wacht terug op de klok.
      (ga naar stap 1)
5. De databank-klasse communiceert met de databank welke de data opslaat. Het systeem wacht terug op de klok.
   (ga naar stap 1)

De gebruiker wenst de huidige situatie in te schatten door deze te bekijken op de kaart of in een lijst

1. De gebruiker opent het dashboard en klikt indien nodig op de knop “vandaag”
   (ga naar stap 2)
2. Het systeem haalt op de achtergrond de informatie uit de databank. Dit gebeurt in de volgende stappen:
   a. Het dashboard spreekt de REST-API aan
      i. De REST-API reageert
         (ga naar stap 2.b)
      ii. De REST-API reageert niet
         (ga naar stap 3.b)
   b. De REST-API spreekt zijn databank aan via een Entity-manager en geeft de informatie terug aan het dashboard
      i. De databank reageert
         (ga naar stap 2.c)
      ii. De databank reageert niet
         (ga naar stap 3.b)
   c. Het dashboard verwerkt deze informatie en
      i. Toont deze aan de gebruiker
         (ga naar stap 3.a)
      ii. Merkt dat er incorrecte of onvolledige data aanwezig is.
         (ga naar stap 3.b)
3. De gebruiker krijgt informatie te zien:
   a. De gevraagde informatie wordt weergegeven op de kaart en in de lijst
      (ga naar EINDE)
   b. Er wordt een foutmelding getoond dat het systeem momenteel problemen vertoont
      (ga naar EINDE)

De gebruiker wenst te zien hoe druk het verkeer in een bepaalde periode of op een bepaalde dag is

1. De gebruiker opent het dashboard en klikt indien nodig op de knop “periode” of “dag”. Er komt een drop-down menu tevoorschijn die leeg is of waarin al een periode staat. Indien het om een dag gaat komt er een kalender tevoorschijn waar men de gewenste dag kan kiezen. *(ga naar stap 3)*
    a. De periode die de gebruiker wil bestaat al, de gebruiker klikt hier op.
    (ga naar stap 3)
    b. De periode die de gebruiker wil bestaat nog niet
    (ga naar stap 2)
2. De gebruiker klikt op de knop “nieuw” of “aanpassen”. In de pop-up voegt de gebruiker start- en eind-tijdstip toe. Indien de gebruiker deze periode een naam wil geven klikt hij op “hergebruik”. Is de gebruiker tevreden met de periode dan klikt hij op “ok”
    (ga naar stap 3)
3. Het systeem haalt op de achtergrond de informatie uit de databank. Dit gebeurt in de volgende stappen:
    a. Het dashboard spreekt de REST-API aan
        i. De REST-API reageert
        (ga naar stap 2.b)
        ii. De REST-API reageert niet
        (ga naar stap 3.b)
    b. De REST-API spreekt zijn databank aan via een Entity-manager en geeft de informatie terug aan het dashboard
        i. De databank reageert
        (ga naar stap 2.c)
        ii. De databank reageert niet
        (ga naar stap 3.b)
    c. Het dashboard verwerkt deze informatie en
        i. Toont deze aan de gebruiker
        (ga naar stap 3.a)
        ii. Merkt dat er incorrecte of onvolledige data aanwezig is.
        (ga naar stap 3.b)
4. De gebruiker krijgt informatie te zien:
    a. De gevraagde informatie wordt weergegeven in de lijst
    (ga naar EINDE)
    b. Er wordt een foutmelding getoond dat het systeem momenteel problemen vertoont
    (ga naar EINDE)

De gebruiker wenst 2 periodes te vergelijken

1. De gebruiker opent het dashboard en klikt indien nodig op de knop “vergelijk periode”. Er komen 2 drop-down menu’s tevoorschijn die leeg zijn of waarin al een periode staat.

a. De periodes die de gebruiker wil bestaat al, de gebruiker klikt hier op.
(ga naar stap 3)

b. De periodes die de gebruiker wil bestaat nog niet
(ga naar stap 2)

2. De gebruiker klikt op de knop “nieuw” of “aanpassen”. In de pop-up voegt de gebruiker start- en eind-tijdstip toe. Indien de gebruiker deze periode een naam wil geven klikt hij op “hergebruik”. Is de gebruiker tevreden met de periode dan klikt hij op “ok”
(ga naar stap 3)

3. Het systeem haalt op de achtergrond de informatie uit de databank. Dit gebeurt in de volgende stappen:

a. Het dashboard spreekt de REST-API aan

i. De REST-API reageert
(ga naar stap 2.b)

ii. De REST-API reageert niet
(ga naar stap 3.b)

b. De REST-API spreekt zijn databank aan via een Entity-manager en geeft de informatie terug aan het dashboard

i. De databank reageert
(ga naar stap 2.c)

ii. De databank reageert niet
(ga naar stap 3.b)

c. Het dashboard verwerkt deze informatie en

i. Toont deze aan de gebruiker
(ga naar stap 3.a)

ii. Merkt dat er incorrecte of onvolledige data aanwezig is.
(ga naar stap 3.b)

4. De gebruiker krijgt informatie te zien:

a. De gevraagde informatie wordt weergegeven in de lijst
(ga naar EINDE)

b. Er wordt een foutmelding getoond dat het systeem momenteel problemen vertoont
(ga naar EINDE)

TAAKVERDELING

Sprint 1

De doelstelling voor sprint 1 was de backend van de verkeersapplicatie realiseren. Hierbij was het de bedoeling van verkeersinformatie op te halen bij de providers en deze in een databank op te slaan. Dit deel bestaat dus uit 2 grote delen die telkens door 2 personen zijn gemaakt.

Jarno en Simon hebben de eerste hoofdtaak, de data ophalen bij providers, voor hun rekening genomen. Dit gedeelte hield in dat verschillende AProviderConnector-klassen (één voor elke provider) de verkeersinformatie van de verschillende trajecten moeten ophalen en deze omvormen naar objecten die kunnen opgeslagen worden in de databank. Het ophalen van de data bij een provider gebeurt via de API die deze provider aanbiedt, waardoor de manier van data ophalen bij alle providers verschillend was. De klassen die zorgen voor de connectie met de providers Waze en Here heeft Simon geïmplementeerd, terwijl Jarno deze voor Google en Coyote heeft gemaakt. Omdat Perl in de applicatie moest verwerkt worden, is de Coyote-connector grotendeels op deze manier gerealiseerd. Enkel het opslaan in de databank is nog Java-code. Uiteindelijk heeft Robin de TomTom connector er nog aan toegevoegd.

Robin en Piet waren verantwoordelijk voor de 2^{e} hoofdtaak, de opgehaalde data opslaan in een databank. De RDBMS van de databank is MySQL maar in sprint 1 was dit nog MariaDB, daarom draagt de DAO (Data Access Object) van de applicatie nog steeds de naam MariaDbConnector. Deze DAO zorgt ervoor dat Jarno en Simon de opgehaalde data via eenvoudige insert-commando's in de databank konden opslaan. De andere taak van de DAO is een interface aanbieden van de databank. Robin had al heel wat ervaring met Linux en heeft de databank aangemaakt en gezorgd voor de nodige configuratie hiervan (het aanmaken van accounts, tabellen…).

De klasse die het project runt en de klasse die instaat voor het ophalen van de data -dit is het pollen, uitgevoerd dus door Pollthread in sprint 1 en 2 - is ook door Robin ontworpen. Dit pollen zorgt dat elke 5 minuten deze provider-connectors worden aangewend om de data op te halen en op te slaan. Een extra taak die erbij kwam was het loggen van de fouten die voorkomen in deze verkeer-applicatie, deze was origineel ontworpen door Piet maar Robin had hier een beter alternatief voor gevonden door bibliotheken te gebruiken.

Sprint 2

In sprint 2 was het de bedoeling om deze opgehaalde data uit sprint 1 te visualiseren en beschikbaar te stellen voor gebruikers. De opdracht werd opnieuw ingedeeld in 2 grote taken. Enerzijds de data aanbieden aan gebruikers en derden via een REST API en anderzijds een dashboard ontwerpen voor de klant op basis van deze data.

Robin en Piet hebben de eerste hoofdtaak bij sprint 2, de data aanbieden via een REST API, verwezenlijkt. Hierbij moest op basis van JAX-RS jersey een REST API tot stand gebracht worden, waarbij de data aangeboden wordt in de vorm van URL's. Robin deed grotendeels het ontwerp van de query's, waarbij hij ook verbeteringen aanbracht in de databank om deze query's efficiënter te maken. Vervolgens heeft hij de meeste scripts ontworpen voor bijvoorbeeld dumps te halen van de productieomgeving. Piet heeft voor de methodes gezorgd die de aanvragen aan de REST API afhandelen, Robin heeft deze dan helpen optimaliseren voor een betere output naar de gebruiker.

Jarno en Simon hebben zich over de 2e hoofdtaak ontfermd, het visualiseren van de data voor de klant via een eenvoudig te gebruiken dashboard. Het basisidee is door Jarno met pen en papier getekend. Simon heeft hier uitgebreide digitale mock-ups van getekend. Onze klant kon deze al bekijken en beoordelen. Daarna heeft Simon deze volledig vertaald naar HTML en CSS met aanvullingen van door Jarno. Het uitschuifbaar paneel van het dashboard is grotendeels ontworpen door Simon maar Jarno heeft ook aan het visuele aspect ervan zijn bedrage geleverd. Dit paneel staat in voor de tekstuele voorstelling van de data door verschillende functies aan te bieden (data over bepaald interval, live data, …) en deze te verwerken in grafieken. Jarno heeft zich dan weer vooral toegespitst op het kaart-gedeelte. Hierbij heeft hij de waypoints op de kaart aangeduid en aangepast en heeft hij het ook mogelijk gemaakt meer informatie te zien over routes via een pop-up systeem. Simon heeft ook enkele noodzakelijke REST API aanvragen ontwikkeld om de dashboard-functies te realiseren. Simon was ook verantwoordelijk voor het grootste deel van de javascript objecten.

Sprint 3

Dit was een korte sprint waardoor de backlogs beperkt zijn gebleven. Het debuggen van de applicatie en enkele verbeteringen en toevoegingen aan het dashboard stonden voorop.

Robin heeft de taak van het debuggen van de backend op zich genomen. Problemen met de slechte performantie door een overmatig gebruik van threads, en problemen met limieten voor het aantal aanvragen bij onze providers, zijn door hem en Simon opgelost. Hiernaast heeft Robin ook gezorgd dat de globale performantie van het dashboard verbeterd werd door bepaalde geaggregeerde data van de databank periodiek te berekenen in plaats van in real time. Dit wil zeggen dat de data nu elke 5 minuten geaggregeerd wordt wat resulteert in iets meer data, maar snellere query's.

Jarno heeft de kaart verder gedebugd. Er zaten bijvoorbeeld nog enkele fouten in de waypoints van de kaart. Coyote zorgde ook voor problemen omdat het meer trajecten opvolgt dan de andere providers. Vervolgens heeft hij ervoor gezorgd dat de dashboard instelling (kaartcoördinaten, opgeslagen intervallen, …) toegevoegd worden aan de URL, waardoor het delen van een specifieke weergave met iemand anders significant vereenvoudigd is. Als laatste heeft Jarno ook nog een knop toegevoegd die het toelaat van de live data elke 5 minuten automatisch te laten updaten.

Simon heeft de dag weergave toegevoegd aan het dashboard, zo kan je data ophalen voor een bepaalde dag en hoef je dit niet in de interval-functie op te vragen. Naast grote aanpassingen in de weergave van het dashboard heeft hij heeft ook de javascript code wat meer object-georiënteerd gemaakt om dubbele code te verwijderen.

Piet heeft enkele verbeteringen aan de REST API aangebracht zoals een eenvoudige beveiliging met API-keys (deze kan vanzelfsprekend uitgebreid worden naar een betere en uitgebreidere methode) en een goede communicatie met de gebruikers van de API indien er fouten zijn opgetreden.

Uiteindelijk heeft elk van ons zich ook beziggehouden met de code overzichtelijker te maken door onder andere gebruik te maken van een opsplitsing in submethodes en meer en duidelijkere annotaties toe te voegen.

STATUSVERSLAG

(Features die wel/niet werden gerealiseerd)

Backend features

In de databank is een tabel 'traveltimes' geïmplementeerd geweest die niet gebruikt wordt. Deze tabel ging de ideale reistijden, berekend tijdens het lopen van de applicatie, moeten opslaan waardoor op een eenvoudige manier de reistijd van een traject vergeleken kan worden met de ideale reistijd. Een extra tabel hiervoor leek ons achteraf overbodig en er is gekozen geweest deze in de tabel van de routes op te slaan. Dit gebeurt niet meer onder de vorm van een ideale reistijd maar door een 'speedlimit' van het traject, die hetzelfde effect heeft (een ideale reistijd is uiteindelijk ongeveer hetzelfde als een traject aan de maximale snelheid afleggen). Naast vergelijken met de ideale snelheid van een traject is het nu ook mogelijk met de gemiddelde reistijd per dag te vergelijken, deze waarde bevindt zich in de trafficdata tabel.

Dit was voordien niet gepland maar het is nu ook mogelijk eigenschappen van de applicatie te wijzigen terwijl de applicatie nog draait. Hiervoor zijn de properties-bestanden (database.properties, app.properties, provider.properties) van de applicatie extern in een 'config' bestand geplaatst.

De applicatie wordt afgesloten van 12u s' nachts tot 6u s' morgens, oorspronkelijk was het idee om deze wel aan te laten s' nachts maar met een groter pollinterval. Door onbetrouwbare data van de providers s' nachts (te weinig weggebruikers) is er toch niet gekozen geweest voor deze optie.

De REST API beveiliging is maar deels verwezenlijkt. Het plan voor de beveiliging was niet concreet vastgelegd maar de uiteindelijke beveiliging is ondermaats qua veiligheid, uitbreiding van dit onderdeel is dus een vereiste.

Front-End features

Het selecteren van meerdere providers in de menubalk van het dashboard om hen data te aggregeren (gemiddelde van hen data nemen) is niet verwezenlijkt geweest. Hier is bewust voor gekozen omdat je anders de data van een betere provider kan verzieken met data van een slechte provider. Er is geopteerd geweest voor een alternatief waarbij je in de grafieken kan selecteren of je meerdere providers wil vergelijken met elkaar.

De dag-functie van het dashboard ging origineel niet aanwezig zijn. Dit omdat deze data voor een bepaalde dag ook kan opgevraagd worden via de interval-functie maar om praktische redenen leek de dag-functie wel een handige tool.

De zoekfunctie voor een route te zoeken op naam is ook een feature die maar tijdens het creatieproces van het dashboard is ontworpen. Uiteindelijk heeft dit een grote bijdrage tot de gebruiksvriendelijkheid van de applicatie.

Pop-ups in de kaart zijn ook maar tijdens het verloop van sprint 2 ter sprake gekomen. Deze pop-ups maken het mogelijk nu ook data over een traject op te vragen door erop te klikken in de. Hierdoor wordt de keuze van hoe de applicatie gebruikt wordt overgelaten aan de gebruiker.

De instellingen van het dashboard, zoals de opgeslagen intervallen en de kaart-coördinaten, worden nu meegegeven in een URL om collaboratie tussen verschillende gebruikers te vereenvoudigen (kopiëren en doorsturen van de URL). Dit is maar tijdens het ontwikkelingsproces van sprint 3 aan de orde gekomen.

Gekende problemen

De pollthread maakte voor iedere route één thread aan, dus tot 30 threads, om de aanvragen uit te voeren. Dit overmatige gebruik van threads zorgde voor een vertraagde werking van de applicatie waardoor nu één thread per provider, in totaal voorlopig 5 threads, wordt gebruikt.

De query's spelen een grote rol in de performantie van de applicatie. Deze moesten dan ook meerdere malen geoptimaliseerd worden om toch een behoorlijke aanvraagtijd te verkrijgen bij query's die veel worden uitgevoerd. Daarnaast leidt alle verwerking van data aan client-side (javascript) tot een vertraagd dashboard waardoor extra query's en interfaces moesten toegevoegd worden.

De productieomgeving was enorm traag en dit is deels opgelost geweest door geheugen te laten swappen bij het uitvoeren van de applicatie. Bij het invoeren van de nieuwe productieomgeving was het geheugenprobleem weg en werkte alles veel rapper.

Glassfish zorgde voor enorm veel problemen. Sommige onverklaarbaar die opgelost werden met Glassfish te herstarten en andere die veroorzaakt waren door de recentste versie, Glassfish 4.1.1. Vervolgens leidde dit voordien vermelde gebrek aan geheugen ook nog eens tot het feit dat Glassfish enorm traag was. De REST API ontwerpen en uittesten was bijgevolg een moeizaam proces die meer tijd in beslag nam dan voorzien.

De Coyote providerconnector, die geïmplementeerd is in Perl, zorgde voor heel wat problemen. Enkele voorbeelden hiervan zijn het parsen van de JSON die opgevraagd werd bij Coyote, de cookie voor de authenticatie bij hen API die niet werd onthouden en de interactie tussen Java en JSON tot stand brengen.

In de databank was per ongeluk bij de waypoints de tabelnaam latitude en longitude omgewisseld. De routes werden daardoor getekend in de hoorn van Afrika, het heeft heel wat tijd gekost voor we dit hadden ontdekt.

TomTom heeft een zeer beperkt aantal aanvragen per dag. Dit probleem hebben we opgelost door 8 API-KEYS toe te voegen aan de applicatie en dus nu 8 accounts tegelijkertijd aanvragen te laten uitvoeren.

Op de kaart van het dashboard kwam boven water dat de waypoints niet correct samenvallen met de routes. Handmatige aanpassing van de waypoints voor een mooie kaartstructuur zorgde dus voor een lastig en tijdrovend proces.

Bij het ontwerp van de tekstuele data en grafieken van het dashboard traden ook heel wat problemen op.

Verbeterpunten

De fouten die door de REST API worden opgeworpen worden deels verwerkt in de REST API zelf, bijvoorbeeld bij gebrek aan een verplichte parameter, en deels door de ExceptionMapper. Er zou minder dubbele code optreden en de code zou in het algemeen meer leesbaar zijn indien alle opgeworpen fouten worden verwerkt in ExceptionMapper's.

De documentatie van de code is soms in het Nederlands en soms in het Engels, eenheid hierin creëren geeft een professioneler beeld van de applicatie.

De TomTom-provider die 8 API-KEYS nodig heeft is geen correcte oplossing voor het probleem. Een alternatief aanvragen aan TomTom, bijvoorbeeld uitbreiding van het aangeboden pakket, is waarschijnlijk een eenvoudigere oplossing.

Uitbreidingsmogelijkheden

Doordat de properties bestanden nu buiten de applicatie staan, wordt het mogelijk om een instellingen optie te implementeren in het dashboard waarbij je bijvoorbeeld een provider kan toevoegen, het pollinterval kan aanpassen en eigenschappen van een provider kan veranderen.

Qua beveiliging van de applicatie, het dashboard eventueel meegerekend, is het mogelijk login en register-scherm te implementeren die gebaseerd is op het systeem met API-keys dat reeds aanwezig is. Een ingelogde gebruiker zou vervolgens dan een aanvraag voor een API-KEY kunnen indienen. Wat wel aanwezig is, is het controleren van iedere API-aanvraag en hierbij kijken of er een API-key header aanwezig is en of deze geldig is. Deze API-KEY is hard gecodeerd in de applicatie.

De REST API is volledig gedocumenteerd in het projectdossier maar deze is niet aanwezig in de applicatie zelf, wat het gebruik ervan vermoeilijkt voor derden die de API ook willen gebruiken. Deze documentatie in een overzichtelijke interface aanbieden is een uitstekende uitbreidingsmogelijkheid.

Naast een login-scherm zou ook een pagina kunnen voorzien worden die de verschillende functionaliteiten aanbiedt van de applicatie (zoals de instellingen aanpassen, een interval toevoegen, de REST API documentatie, …). Dit zorgt dat de klant meer geleid wordt in het gebruik van de applicatie wat gebruikersvriendelijker is. Nadeel is echter dat er ook meer klikwerk wordt veroorzaakt.

Opmerkelijke vertragingen (die wellicht veroorzaakt zijn door een ongeval of dergelijke) worden reeds voorgesteld door een traject rood te kleuren. Deze vertragingen worden berekend door reistijden te vergelijken met de gewoonlijke reistijd van het traject. Om de opvolging en de verwerking van deze gegevens te vereenvoudigen zou een oorzaak meegeven een goede aanvulling van de applicatie zijn. Deze oorzaken kunnen opgehaald worden bij enkele van de providers.