DeLijnRater

Installation and running

Requirements

This application needs both Docker and Docker Compose to be able to run. The requirements of each container are installed automatically. ### Running the project #### Automatic The project can be ran using the provided scripts. Run the scripts in the order below:

1. install.sh
2. startup.sh
3. fill_db.sh (optional)

The first script builds the images and containers. The second script starts the application which will be visible on http://localhost:80/. The last script fills the databases with all the stops and dummy data. This will take ~40 seconds to complete

Manual

To build the project manually, run the next command in the project root. This will build and setup the docker containers.

```
docker-compose -f docker-compose.yml build
```

When the project is built successfully, run the project using the next command, also called from the project root.

```
docker-compose -f docker-compose.yml up
```

This command can be run with an optional -d tag to run in the background. The application will be visible on http://localhost:80/

The project can be stopped using the following function

```
docker-compose -f docker-compose.yml stop
```

Additional startup commands

The Flask applications provide some extra commands to manipulate some features of the project. Run the commands below to find out which commands are available

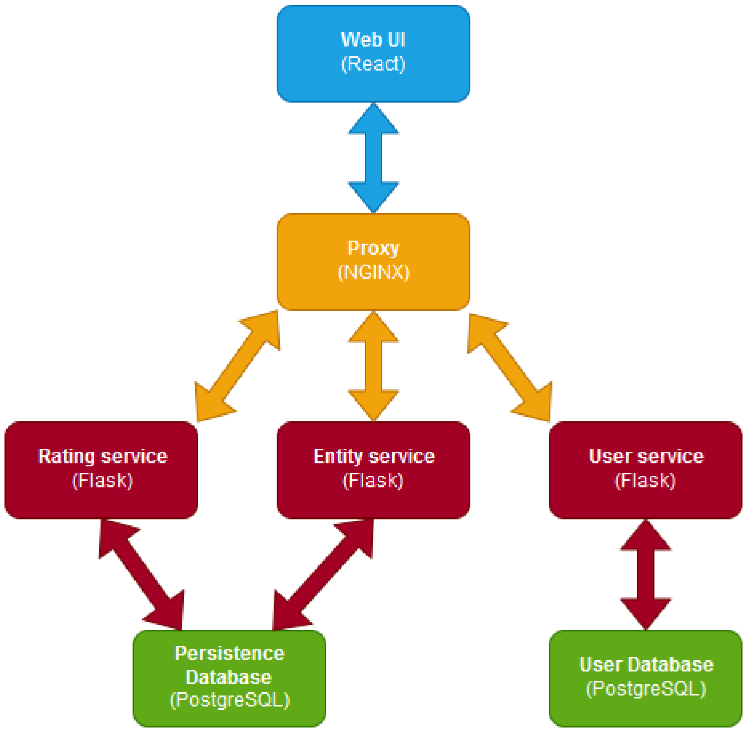
```
docker-compose -f docker-compose.yml exec entities python start.py --help
```

```
docker-compose -f docker-compose.yml exec ratings python start.py --help
```

```
docker-compose -f docker-compose.yml exec users python start.py --help
```

General project structure

This project consists of 7 independent microservices. The microservices communicate via HTTP requests. A visualization of the project structure can be seen below.



The web ui service provides an interface for the user to access the application. The user services communicates with the Nginx service that behaves as a reversed proxy. The reversed proxy redirects the request to the correct Flask applications. There are three Flask applications in total. One handles the API calls that manipulates entities. The entities are the bus stops and the vehicles. Another Flask app is used to handle the rating related functionality. The last Flask app is the user service which is used for logging in and registering. The API calls are described further in this document. The last two services are databases. One is used for persistence data, this are the entities and the ratings, and the other one is the users database.

Flask services API structures

The API's can be accessed in two ways. Each section below shows the two possible URL's to access the Restful resources. Each Flask service uses the same error format described below

```
{
    "message": "message" ('Not Found' for example),
    "error": "additional error message",
    "status": 404 (for example)
}
```

Entities service

The entities service is visible via http://localhost:5001/. The service is also reachable using the following url: http://localhost/api/entities/. The second URL is provided by the reversed proxy.

GET /regions/values

This resource returns the possible values for the regions.

```
Example URL: http://localhost:5001/regions/values
Example URL: http://localhost/api/entities/regions/values

Possible result
{
    "Antwerpen": 1,
    "Oost Vlaanderen": 2,
    "Vlaams Brabant": 3,
    "Limburg": 4,
    "West Vlaanderen": 5
}
```

GET /stops

This resource returns all possible stops stored in the persistence database. Returns with status code 200 if it was successful.

```
Example URL: http://localhost:5001/stops
Example URL: http://localhost/api/entities/stops

Possible result
[
    {
        "id": 1,
        "region": "Antwerpen",
        "name": "Quellinstraat",
        "stop_number": 100254,
        "village": "Hoboken"
    },
    ...
]
```

GET /stops/id/{stop_id}

Returns the stop with the given id. This id is not equal to the stop number. Returns with status code 200 if it was successful.

Parameters

- stop_id: integer

Possible errors

- 500: the parameter stop_id was not convertable to an integer.
- 404: there is no stop with the provided stop_id

```
Example URL: http://localhost:5001/stops/id/1
Example URL: http://localhost/api/entities/stops/id/1

Possible result
{
    "id": 1,
    "region": "Antwerpen",
    "name": "A. Chantrainestraat",
    "stop_number": 101000,
    "village": "Wilrijk"
}
```

GET /stops/village/{village}

Returns the stops that are in situated in the given village. Returns with status code 200 if it was successful.

Parameters

- village: string

```
Example URL: http://localhost:5001/stops/village/Oppuurs
Example URL: http://localhost/api/entities/stops/village/Oppuurs

Possible result
[
    {
        "id": 141,
        "region": "Antwerpen",
        "name": "Kattestraat",
        "stop_number": 101226,
        "village": "Oppuurs"
    },
    {
        "id": 337,
        "region": "Antwerpen",
        "name": "Lippeloseweg",
        "stop_number": 101546,
        "village": "Oppuurs"
    },
...
]
```

GET /stops/region/{region}

Returns the stops that are in situated in the given region. Returns with status code 200 if it was successful.

Parameters

- region: string or integer. The possible values and their meaning are provided by an http call to the '/regions/values' resource.

Possible errors

- 500: the given region could not be converted to a Region type
- 500: the given region does not exist

```
Example URL: http://localhost:5001/stops/region/LIMBURG
Example URL: http://localhost/api/entities/stops/region/LIMBURG

Possible result
[
    {
        "id": 22014,
        "region": "Limburg",
        "name": "Ossenberg",
        "stop_number": 400004,
        "village": "Hulsen"
    },
    {
        "id": 22015,
        "region": "Limburg",
        "name": "Het Oud Blok",
        "stop_number": 400010,
        "village": "Hulsen"
    },
...
]
```

GET /stops/line/{region}/{line_number}

Returns all the stops that are part of a line of 'De Lijn'. This resource makes a http call to the api of De Lijn to get the required stop numbers. This resource returns not only those numbers, but also the other data that is stored. Returns with status code 200 if it was successful.

```
Example URL: http://localhost:5001/stops/line/1/32
Example URL: http://localhost/api/entities/stops/line/1/32

Possible result
[
    {
        "id": 4387,
        "region": "Antwerpen",
        "name": "Kattenbroek",
        "stop_number": 107285,
        "village": "Edegem"
    },
    {
        "id": 1980,
        "region": "Antwerpen",
        "name": "Mgr. Cardijnlaan",
        "stop_number": 104275,
        "village": "Edegem"
    },
...
]
```

GET /vehicles/values

This resource returns the possible values for the vehicle types. Returns with status code 200 if it was successful.

```
Example URL: http://localhost:5001/vehicles/values
Example URL: http://localhost/api/entities/vehicles/values

Possible result
{
    "Bus": 0,
    "Tram": 1
}
```

GET /vehicles

Returns all vehicles that are stored in the database. Returns with status code 200 if it was successful.

```
Example URL: http://localhost:5001/vehicles
Example URL: http://localhost/api/entities/vehicles

Possible result
[
    {
        "id": 5,
        "number": 32,
        "description": "Bus in Antwerpen",
        "type": "Bus",
        "created_by": 3,
        "name": "Antwerpen"
    },
    {
        "id": 0,
        "number": 32,
        "description": "Bus bus bus",
        "type": "Bus",
        "created_by": 0,
        "name": "Bus zonder naam"
    },
...
]
```

POST /vehicles

Adds a new vehicle to the database. This post method requires form data or json as parameters. If this function succeeds, it returns the created vehicle with status code 201.

Data to post

- id: integer, required: unique ID of the vehicle
- number: integer, required: the number of the vehicle
- type: VehicleType (see /vehicles/values), required: the type of the vehicle
- name: string: the name of the vehicle
- description: string: the description of the vehicle
- created_by: integer, required: the ID of the user that created the vehicle

Possible errors

- 500: unable to create vehicle with specified arguments

```
Example URL: http://localhost:5001/vehicles
Example URL: http://localhost/api/entities/vehicles

Example data:
{
  "id": 3,
  "number": 21,
  "type": "Bus",
  "name": "Bus tussen A en B",
  "description": "drives between A and B and sometimes via C. Always ends at 
Bist",
  "created_by": 2
}

Result:
{
  "id": 3,
  "number": 21,
  "type": "Bus",
  "name": "Bus tussen A en B",
  "description": "drives between A and B and sometimes via C. Always ends at
Bist",
  "created_by": 2
}
```

GET /vehicles/id/{vehicle_id}

Returns the vehicle with the specified id. If this function succeeds, it returns status code 200.

Parameters

- vehicle_id: integer

Possible errors

- 404: vehicle with specified id does not exist
- 500: provided vehicle_id could not be converted to an integer

```
Example URL: http://localhost:5001/vehicles/id/1
Example URL: http://localhost/api/entities/vehicles/id/1

Possible result
{
    "id": 1,
    "number": 8,
    "description": "Trim tram trom",
    "type": "Tram",
    "created_by": 0,
    "name": "Tram zonder naam"
}
```

GET /vehicles/creator/{creator_id}

Returns all the vehicles that were created by the user with the given id. Returns 200 if it succeeds.

Parameters

- creator_id: integer

Possible errors

- 500: given creator id could not be converted to an integer.

```
Example URL: http://localhost:5001/vehicles/creator/1
Example URL: http://localhost/api/entities/vehicles/creator/1

Possible result
[
    {
        "id": 2,
        "number": 21,
        "description": "bus",
        "type": "Bus",
        "created_by": 1,
        "name": "Bus met naam zeker"
    },
...
]
```

Rating service

The rating service is visible via http://localhost:5002/. The service is also reachable using the following url: http://localhost/api/ratings/. The second URL is provided by the reversed proxy.

POST /stops/rating

Create a new rating for a stop. If a user already gave a rating for this stop, the old rating is overwritten. If the rating was created successfully, the Flask resource returns the created rating with status code 201. This function needs json or form data as parameters.

Parameters

- entity_id: integer, required: the ID of the stop to rate.
- created_by: integer, required: the ID of the user that created the rating.
- rating: float, required: the score the user gave the stop. The allowed values are in the interval [0, 10]

Possible errors

- 500: the given parameters were wrong.

```
Example URL: http://localhost:5003/stops/rating
Example URL: http://localhost/api/ratings/stops/rating

Possible parameters
{
    "entity_id": 1,
    "created_by": 2,
    "rating": 7.5
}
```

```
Result:
{
    "entity_id": 1,
    "created_by": 2,
    "rating": 7.5
}
```

GET /stops/rating/{stop_id}

Returns all the ratings of a given stop. If this function succeeds, it returns with status code 200.

Parameters

- stop_id: integer: id of the stop to give the ratings for

Possible errors

- 500: the given stop_id could not be converted to an integer.

```
Example URL: http://localhost:5003/stops/rating/1
Example URL: http://localhost/api/ratings/stops/rating/1

Possible result
[
    {
        "id": 1,
        "stop_id": 1,
        "created_by": 3,
        "rating": 6.0
    },
    {
        "id": 3,
        "stop_id": 1,
        "created_by": 1,
        "rating": 8.5
    },
    {
        "id": 4,
        "stop_id": 1,
        "created_by": 4,
        "rating": 2.0
    }
]
```

GET /stops/rating/user/{user_id}

Returns all the ratings a certain user gave. If this function succeeds, it returns with status code 200

Parameters

- user_id: integer: the id of the user to get the ratings of.

Possible errors

- 500: user_id could not be converted to integer

```
Example URL: http://localhost:5003/stops/rating/user/1
Example URL: http://localhost/api/ratings/stops/rating/user/1

Possible result
[
    {
        "id": 3,
        "stop_id": 1,
        "created_by": 1,
        "rating": 8.5
    },
...
]
```

GET /stops/average/{stop_id}

Returns the average rating of a stop. If there are no ratings, the resource returns the string 'No ratings yet'. If this function succeeds, it returns with status code 200.

Parameters

- stop_id: integer: id of the stop to get the average rating of.

Possible errors

- 500: stop_id could not be converted to integer

```
Example URL: http://localhost:5003/stops/average/1
Example URL: http://localhost/api/ratings/stops/average/1

Possible result
5.5
```

POST /vehicles/rating

Create a new rating for a vehicle. If a user already gave a rating for this vehicle, the old rating is overwritten. If the rating was created successfully, the Flask resource returns the created rating with status code 201. This function needs json or form data as parameters.

Parameters

- entity_id: integer, required: the ID of the vehicle to rate.

- created_by: integer, required: the ID of the user that created the rating.
- rating: float, required: the score the user gave the stop. The allowed values are in the interval [0, 10]

Possible errors

- 500: the given parameters were wrong.

```
Example URL: http://localhost:5003/vehicles/rating
Example URL: http://localhost/api/ratings/vehicles/rating

Possible parameters
{
    "entity_id": 1,
    "created_by": 2,
    "rating": 7.5
}

Result:
{
    "entity_id": 1,
    "created_by": 2,
    "rating": 7.5
}
```

GET /vehicles/rating/{vehicle_id}

Returns all the ratings of a given vehicle. If this function succeeds, it returns with status code 200.

Parameters

- vehicle_id: integer: id of the vehicle to give the ratings for

Possible errors

- 500: the given vehicle_id could not be converted to an integer.

```
Example URL: http://localhost:5003/stops/rating/1
Example URL: http://localhost/api/ratings/stops/rating/1

Possible result
[
    {
        "id": 1,
        "stop_id": 1,
        "created_by": 3,
        "rating": 6.0
    },
    {
        "id": 3,
        "stop_id": 1,
        "created_by": 1,
        "rating": 8.5
    },
...
]
```

GET /stops/vehicle/user/{user_id}

Returns all the ratings a certain user gave. If this function succeeds, it returns with status code 200

Parameters

- user_id: integer: the id of the user to get the ratings of.

Possible errors

- 500: user_id could not be converted to integer

```
Example URL: http://localhost:5003/vehicles/rating/user/1
Example URL: http://localhost/api/ratings/vehicles/rating/user/1

Possible result
[
    {
        "id": 1,
        "vehicle_id": 0,
        "created_by": 3,
        "rating": 7.5
    },
...
]
```

GET /stops/average/{vehicle_id}

Returns the average rating of a vehicle. If there are no ratings, the resource returns the string 'No ratings yet'. If this function succeeds, it returns with status code 200.

Parameters

- vehicle_id: integer: id of the stop to get the average rating of.

Possible errors

- 500: stop_id could not be converted to integer

```
Example URL: http://localhost:5002/vehicles/average/1
Example URL: http://localhost/api/ratings/vehicles/average/1

Possible result
"No ratings yet"
or
8.5
```

User service

The users service is visible via http://localhost:5002/. The service is also reachable using the following url: http://localhost/api/users/. The second URL is provided by the reversed proxy.

POST /login

Controls the credentials of a user. This post request needs json encoded or form data. If this function succeeds it returns with status code 200.

Parameters

- username: string: username of the user
- password: string: password of the user

Possible errors

- 403: user with given username does not exist.
- 403: password for given username is incorrect
- 500: an unexpected error occurred

```
Example URL: http://localhost:5002/login
Example URL: http://localhost/api/users/login

Possible parameters:
{
    "username": "test",
    "password": "testpass"
}
```

POST /register

Registers a new user. This post request needs json encoded or form data. This function returns 201 if it succeeded.

Parameters

- username: string: username of the user

- password: string: password of the user
- email: string: email address of the user

Possible errors

- 500: wrong form data was given

GET /get/{user_id}

Gets the user with the user given user ID. If this function succeeds it returns status code 200.

Parameters

- user_id: integer: id of the user

Possible errors

- 404: user with given user_id does not exist
- 500: could not convert ID to integer

```
Example URL: http://localhost:5002/get/
Example URL: http://localhost/api/users/get/1
```

Other services

Persistence database

This service is a postgres database that contains both the entities (the stops and vehicles) and the ratings. The entities and ratings are in separate tables.

Users database

This service is a postgres database that contains the users.

NGINX reversed proxy

This service uses NGINX as a reversed proxy. This made it easier to route all requests from the web interface to all the API's.

Web UI

This services is a website built with the React framework. It displays all the functionality to the end users in a use friendly way.

Design choices

Flask applications

I chose to subdivide the project into 3 Flask applications. One for users, one for ratings and one for entities. Doing so ensured that there is no monolithic Flask app that contains all resources. This introduced some more overhead, but demonstrated the microservice principle more.

Due to the lack of time, I didn't hash the user passwords before storing them in the database. If the application were to be used in real life, this is something urgent that still needs to happen.

Databases

The application has two database services: one for entities and one for users. This made it impossible to do JOIN operations on the entities tables and user tables, and thus introduced some more overhead and manual checking for inconstancies among the tables.

The benefit of this approach is that when for example the entities database gets corrupted, the user database is still safe.

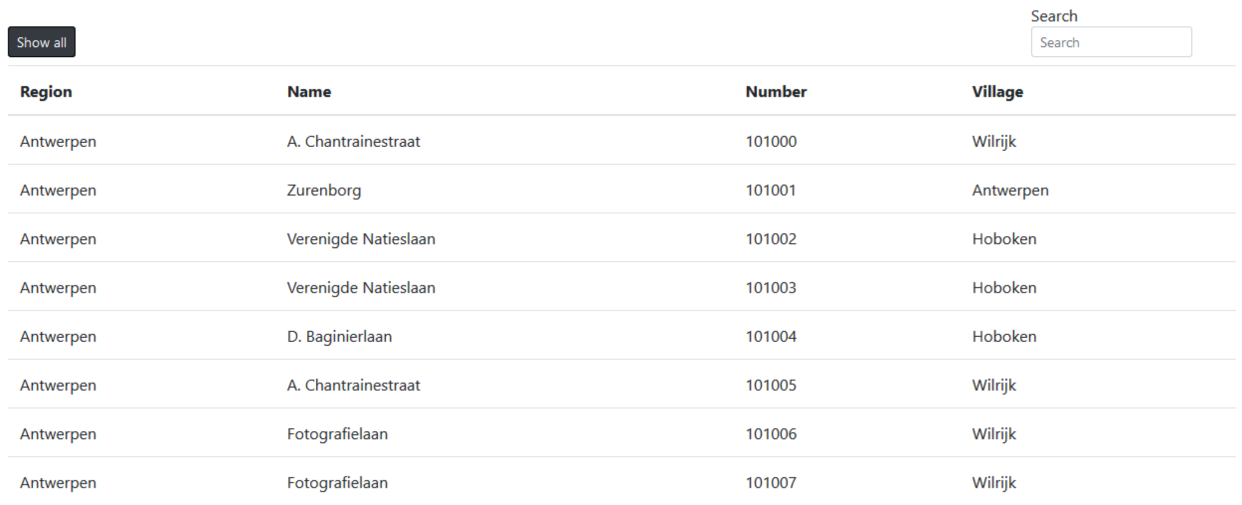
The use of NGINX

Although NGINX as reversed proxy isn't really necessary, it does give a large benefit. I chose to add an NGINX container that handles all the routing because this made it way easier to do the appropriate REST calls in a more convenient way.

React front-end

This application uses React as a front-end. React is a very powerful tool to build websites quite fast. That's why it was interesting to use this framework.

The front-end also uses some extra tools. I chose to use MDB React because this library has a lot built-in easy-to-use React components to for example create a nice looking table. The tables (like the one below) are created using this library.



Region	Name	Number	Village
Antwerpen	A. Chantrainestraat	101000	Wilrijk
Antwerpen	Zurenborg	101001	Antwerpen
Antwerpen	Verenigde Natieslaan	101002	Hoboken
Antwerpen	Verenigde Natieslaan	101003	Hoboken
Antwerpen	D. Baginierlaan	101004	Hoboken
Antwerpen	A. Chantrainestraat	101005	Wilrijk
Antwerpen	Fotografielaan	101006	Wilrijk
Antwerpen	Fotografielaan	101007	Wilrijk

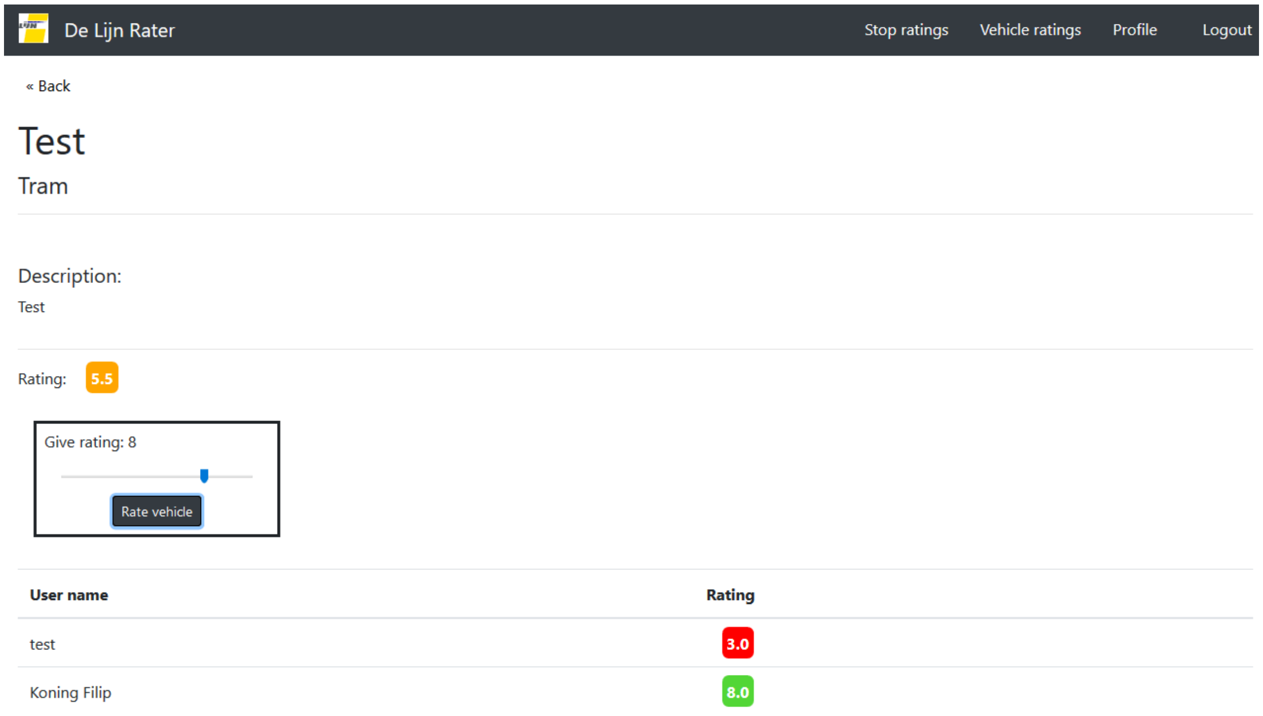
It also features pagination, which was necessary to display all the possible stops at once without the front-end to freeze or even crash.

React is mainly designed to create one-page applications. This application, in my opinion, isn't suitable for a one-pager. That's why I add the React Router library to handle the front-end routing.

Login functionality

Altough it wasn't required, I did create login functionality. With the React provider-consumer pattern, this wan't that hard to achieve. A working login made it possible to render extra components to display the functionality of the application in a cleaner way.

Using this functionality, I created a profile page where the registered user can remove or create vehicles and an extra component where the user can rate stops or vehicles.



Test

Tram

Description:

Test

Rating: 5.5

Give rating: 8

User name	Rating
test	3.0
Koning Filip	8.0

Dependencies

The whole project depends on docker and docker-compose.

Flask applications

The Flask applications all have the same dependencies.

- Flask
- Flask-SQLAlchemy
- Psycopg2
- Flask-Cors
- Flask-Restful
- Jinja2
- Click
- Python3

Databases

- PostgreSQL

Reversed proxy

- NGINX

Front end:

- React
- Axios
- React Router
- MDBootstrap React (https://mdbootstrap.com/)