

# Dokumentacja Projektu PIPR

## (System do obsługi wypożyczalni pojazdów)

## Wstęp

Realizacja projektu składa się z dwóch części:

- backend w postaci REST API
- frontend w postaci aplikacji webowej

## Backend

REST API w Pythonie zbudowane przy użyciu frameworka FastAPI (<https://fastapi.tiangolo.com/>).

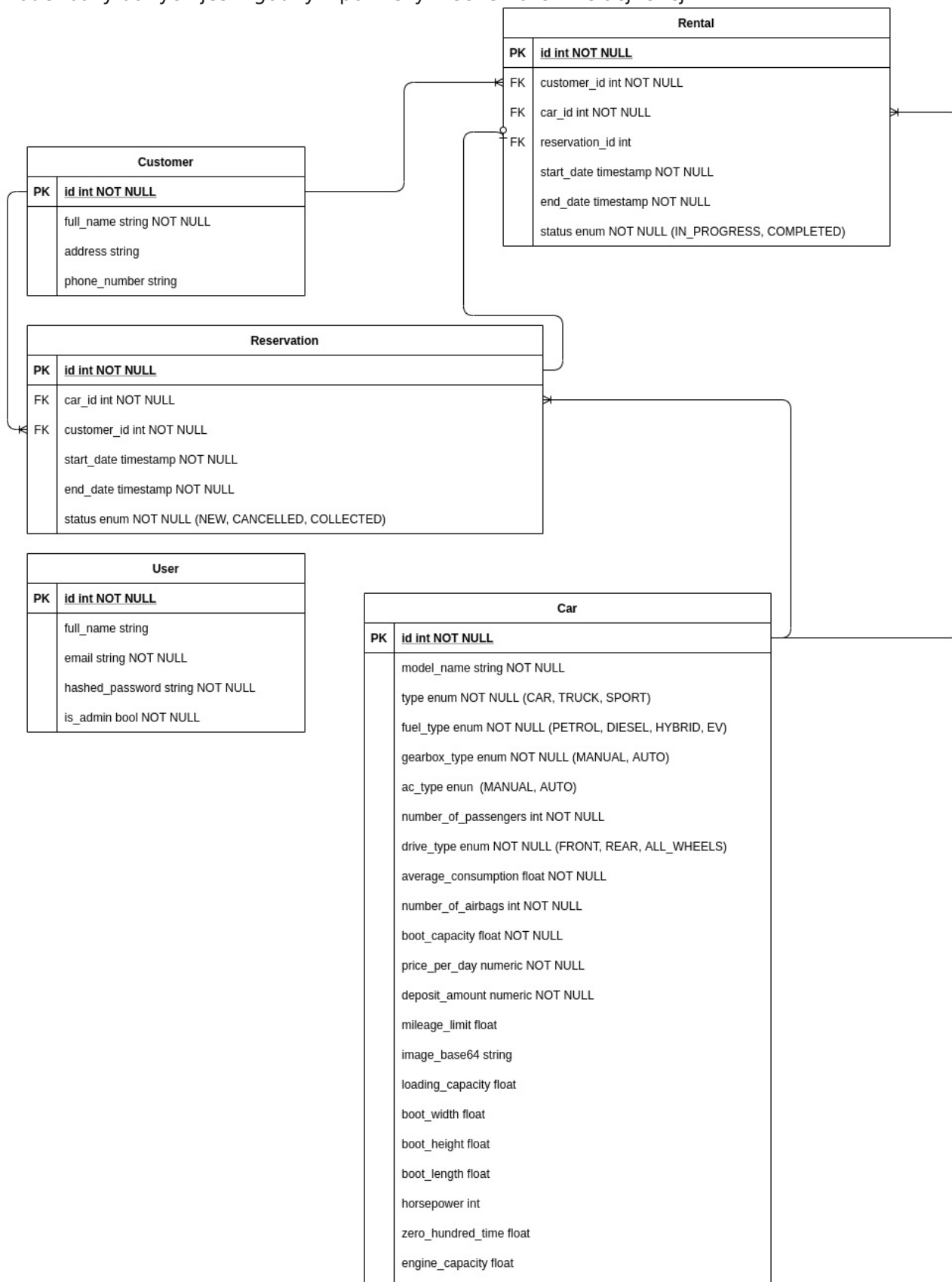
Baza danych użyta do zapisywania danych używanych w aplikacji to PostgreSQL.

API jest zgodne z poniższym arkuszem:

<https://docs.google.com/spreadsheets/d/1ewicTL3VWaDlt85r7Q2gd7PxisiJbTwCgodmpcGsaMI/eusp=sharing>

metoda	ścieżka	request body	response body	opis	uprawnienia
POST	/api/v1/login/access-token	obiekt OAuth2PasswordRequestForm (username: str password: str)	obiekt Token (access_token: str token_type: str)	zwraca token JWT używany do autoryzacji użytkownika	WSZYSCY
GET	/api/v1/users/me	N/D	obiekt User	zwraca użytkownika przypisanego do wysłanego tokenu JWT	WSZYSCY
GET	/api/v1/users/[id]	N/D	obiekt User o podanym id	zwraca użytkownika o danym id	ADMIN
POST	/api/v1/users	obiekt UserCreateDto (email: str password: str)	nowo utworzony obiekt User	tworzy nowy obiekt User	ADMIN
GET	/api/v1/users	N/D	listę obiektów User	zwraca listę obiektów User	ADMIN
GET	/api/v1/cars	N/D	listę obiektów Car	zwraca wszystkie obiekty Cars	WSZYSCY
POST	/api/v1/cars/query	obiekt CarsQueryCriteria	listę obiektów Car na podstawie podanych kryteriów	zwraca listę obiektów Cars na podstawie kryteriów zawartych w obiekcie CarsSearchQuery CarsSearchQuery -> (model, typ, typ paliwa, typ skrzyni biegów, typ klimatyzacji, rodzaj napędu, liczba pasażerów(zakres), cena(zakres), daty dostępności)	WSZYSCY
GET	/api/v1/cars/[id]	N/D	obiekt Car	zwraca obiekt Car o podanym id	WSZYSCY
POST	/api/v1/cars	CreateCarDto	nowo utworzony obiekt Car	tworzy nowy obiekt Car	ADMIN
PUT	/api/v1/cars/[id]	UpdateCarDto	zaktualizowany obiekt Car	przyjmuje CreateCarDto zgodne z obiektem Car (bez ID)	ADMIN
DELETE	/api/v1/cars/[id]	N/D	N/D	aktualizuje obiekt Car na podstawie UpdateCarDto	ADMIN
GET	/api/v1/customers	N/D	listę obiektów Customer	usuwa obiekt Car o danym id	WSZYSCY
GET	/api/v1/customers/[id]	N/D	Customer o podanym id	zwraca listę obiektów Customer	WSZYSCY
POST	/api/v1/customers	CreateCustomerDto	nowo utworzony obiekt Customer	zwraca obiekt Customer o podanym id	WSZYSCY
PUT	/api/v1/customers/[id]	UpdateCustomerDto	zaktualizowany obiekt Customer	tworzy nowy obiekt Customer	WSZYSCY
DELETE	/api/v1/customers/[id]	N/D	N/D	przyjmuje CreateCustomerDto zgodne z obiektem Customer (bez ID)	WSZYSCY
POST	/api/v1/reservations	obiekt CreateReservationDto	nowo utworzony obiekt Reservation	aktualizuje obiekt Customer na podstawie UpdateCustomerDto	WSZYSCY
GET	/api/v1/reservations/[id]	N/D	obiekt Reservation	usuwa obiekt Customer o danym id	WSZYSCY
GET	/api/v1/reservations	N/D	listę obiektów Reservation	tworzy nową rezerwację samochodu	WSZYSCY
PUT	/api/v1/reservations/[id]	UpdateReservationDto	zaktualizowany obiekt Reservation	CreateReservationDto -> (car_id, customer_id, start_date, end_date), status = NEW	WSZYSCY
DELETE	/api/v1/reservations/[id]	N/D	N/D	zwraca obiekt Reservation o danym id	WSZYSCY
POST	/api/v1/rentals	obiekt CreateRentalDto	nowo utworzony obiekt Rental	zwraca listę rezerwacji	WSZYSCY
GET	/api/v1/rentals	N/D	listę obiektów Rental	aktualizuje obiekt Reservation na podstawie UpdateReservationDto	WSZYSCY
GET	/api/v1/rentals/[id]	N/D	obiekt Rental	usuwa obiekt Reservation o danym id	ADMIN
PUT	/api/v1/rentals/[id]	UpdateRentalDto	zaktualizowany obiekt Rental	tworzy nowe wypożyczenie samochodu	WSZYSCY
DELETE	/api/v1/rentals/[id]	N/D	status usuwania (sukces czy błąd)	CreateRentalDto -> (customer_id, car_id, start_date, end_date) status = IN_PROGRESS, może przyjmować reservation_id, wtedy customer_id i car_id muszą być zgodne z rezerwacją. Ustawia wtedy status podanej rezerwacji na COLLECTED	WSZYSCY

Model bazy danych jest zgodny z poniższym schematem relacji encji:



## Frontend

Aplikacja webowa zbudowana przy użyciu biblioteki React.js oraz bibliotek pomocniczych (material-ui, axios).

## Jak uruchomić aplikację/testy

### Backend (testy)

Aby uruchomić testy, najlepiej uruchomić skrypt `scripts/test-docker.sh` który tworzy 2 kontenery Dockera przy pomocy `docker-compose` (backend + baza danych PostgreSQL) na podstawie zmiennych środowiskowych zawartych w pliku `.env`

```
cd backend
sudo chmod +x ./scripts/test-docker.sh
./scripts/test-docker.sh
```

Można też uruchomić testy w standardowy sposób, jednak wymaga to działającej instancji bazy danych PostgreSQL, zainicjalizowania środowiska wirtualnego aplikacji oraz uruchomienia skryptów `prestart.sh` i `tests-start.sh`.

### Backend (aplikacja)

Aby uruchomić aplikację należy najpierw zainstalować jej zależności oraz zainicjalizować środowisko wirtualne. Do zarządzania zależnościami, zamiast `pip`-a użyłem `poetry`, które działaniem przypomina `node package manager` (`npm`).

Narzędzie to należy zainstalować zgodnie z instrukcją na stronie producenta (<https://python-poetry.org/docs/>)

Po zainstalowaniu `poetry`, należy wykonać następujące komendy:

```
cd backend
poetry install
poetry shell
```

Po wykonaniu powyższych komend powinniśmy znajdować się w kontekście środowiska wirtualnego aplikacji i możemy przejść do jej uruchomienia.

Aplikacja wymaga do działania bazy danych PostgreSQL z ustawieniami zgodnymi ze zmiennymi w pliku `.env`. Domyślne wartości to:

```
POSTGRES_SERVER=localhost:5432
POSTGRES_USER=rentally
POSTGRES_PASSWORD=rentally
POSTGRES_DB=rentally
```

W pliku `.env` znajdują się również login i hasło pierwszego użytkownika (z rolami administratora) oraz klucz używany do generowania tokenów JWT.

Ponadto, przed pierwszym uruchomieniem należy przeprowadzić migracje schematu bazy danych oraz utworzyć pierwszego użytkownika. Aby to zrobić należy uruchomić skrypt `prestart.sh`:

```
sudo chmod +x ./prestart.sh
./prestart.sh
```

Aby uruchomić aplikację należy uruchomić skrypt start.sh

```
sudo chmod +x ./start.sh  
./start.sh
```

Domyślnie serwer uruchomi się na porcie 8080, można to zmienić w pliku start.sh lub podać zmienną środowiskową PORT.

## Frontend (aplikacja)

Aby uruchomić aplikację potrzebujemy środowiska node i menedżera pakietów npm (<https://nodejs.org/en/>).

Następnie uruchamiamy poniższe komendy:

```
cd frontend  
npm i  
npm start
```

Domyślnie aplikacja uruchomi się na porcie 3000.

Aby aplikacja mogła skomunikować się z serwerem należy ustawić URL serwera w pliku src/config.js. Domyślnie jest to (zgodne z domyślnymi ustawieniami serwera):

```
const API_URL = "http://localhost:8080/api/v1";
```

## Podział kodu