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

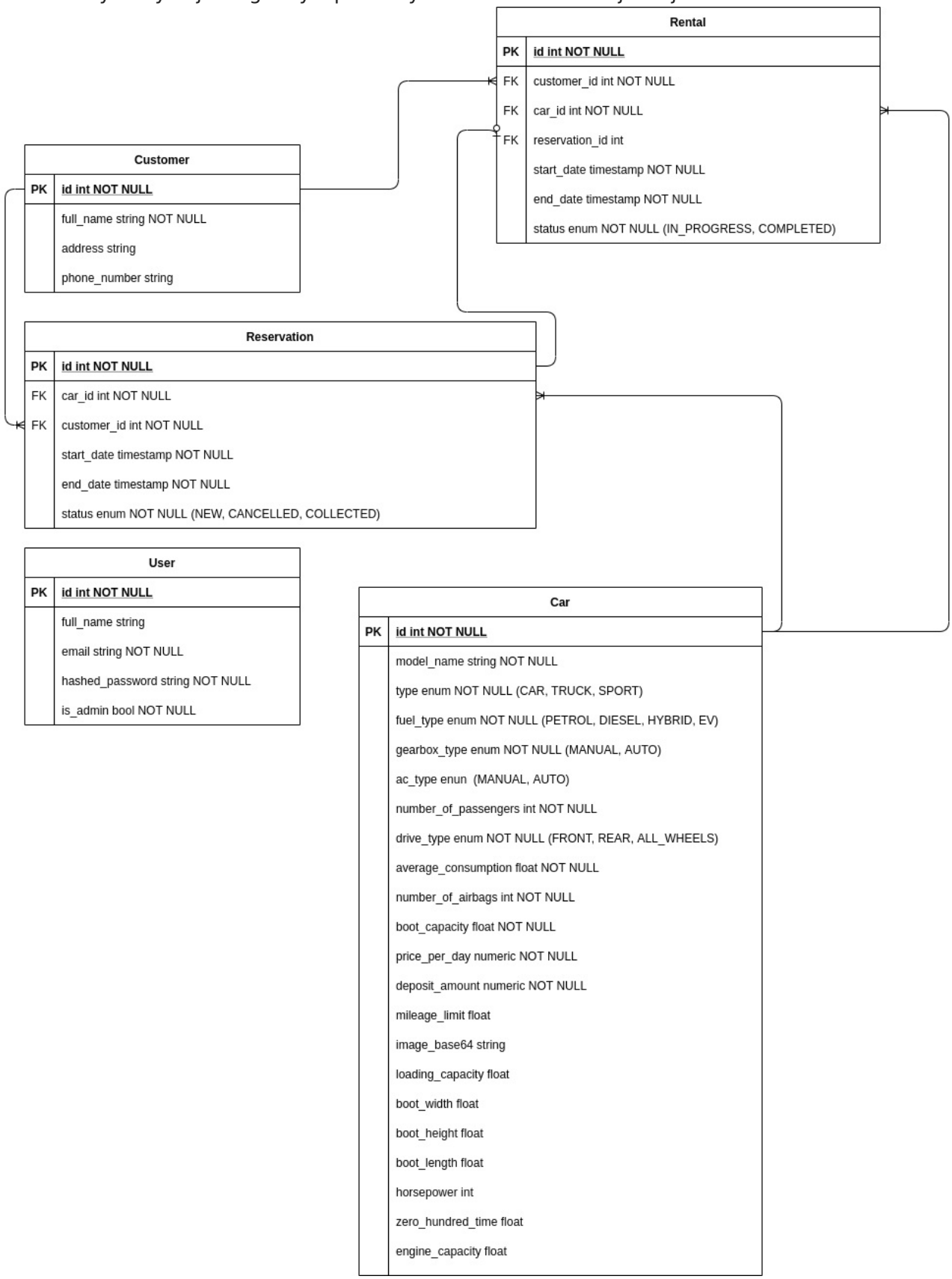
REST API w Pythonie zbudowane przy użyciu frameworka FastAPI [(https://fastapi.tiangolo.com/)](https://fastapi.tiangolo.com/).

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

API jest zgodne z poniższym arkuszem:
[(https://docs.google.com/spreadsheets/d/1ewicTL3VWaDlt85r7Q2gd7PxisjfbTwCgodmpcGsaMI/e
usp=sharing)](https://docs.google.com/spreadsheets/d/1ewicTL3VWaDlt85r7Q2gd7PxisjfbTwCgodmpcGsaMI/edit?usp=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 User -> (id, full_name, email, is_admin)	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 przyjmuje UserCreateDto	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 pasazerow(zakres), cena(zakres), daty dostepnosci)	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 przyjmuje CreateCarDto zgodne z obiektem Car (bez ID)	ADMIN
PUT	/api/v1/cars/[id]	UpdateCarDto	zaktualizowany obiekt Car	aktualizuje obiekt Car na podstawie UpdateCarDto	ADMIN
DELETE	/api/v1/cars/[id]	N/D	N/D	usuwa obiekt Car o danym id	ADMIN
GET	/api/v1/customers	N/D	listę obiektów Customer	zwraca listę obiektów Customer	WSZYSCY
GET	/api/v1/customers[id]	N/D	Customer o podanym id	zwraca obiekt Customer o podanym id	WSZYSCY
POST	/api/v1/customers	CreateCustomerDto	nowo utworzony obiekt Customer	tworzy nowy obiekt Customer przyjmuje CreateCustomerDto zgodne z obiektem Customer (bez ID)	WSZYSCY
PUT	/api/v1/customers/[id]	UpdateCustomerDto	zaktualizowany obiekt Customer	aktualizuje obiekt Customer na podstawie UpdateCustomerDto	WSZYSCY
DELETE	/api/v1/customers/[id]	N/D	N/D	usuwa obiekt Customer o danym id	WSZYSCY
POST	/api/v1/reservations	obiekt CreateReservationDto	nowo utworzony obiekt Reservation	tworzy nową rezerwację samochodu CreateReservationDto -> (car_id, customer_id, start_date, end_date), status = NEW	WSZYSCY
GET	/api/v1/reservations[id]	N/D	obiekt Reservation	zwraca obiekt Reservation o danym id	WSZYSCY
GET	/api/v1/reservations	N/D	listę obiektów Reservation	zwraca listę rezerwacji	WSZYSCY
PUT	/api/v1/reservations/[id]	UpdateReservationDto	zaktualizowany obiekt Reservation	aktualizuje obiekt Reservation na podstawie UpdateReservationDto	WSZYSCY
DELETE	/api/v1/reservations/[id]	N/D	N/D	usuwa obiekt Reservation o danym id	ADMIN
POST	/api/v1/rentals	obiekt CreateRentalDto	nowo utworzony obiekt Rental	tworzy nowe wypożyczenie samochodu 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
GET	/api/v1/rentals	N/D	listę obiektów Rental	zwraca listę wynajmów	WSZYSCY
GET	/api/v1/rentals[id]	N/D	obiekt Rental	zwraca obiekt Rental o podanym id	WSZYSCY
PUT	/api/v1/rentals/[id]	UpdateRentalDto	zaktualizowany obiekt Rental	aktualizuje obiekt Rental na podstawie UpdateRentalDto	WSZYSCY
DELETE	/api/v1/rentals/[id]	N/D	status usuwania (sukces czy błąd)	usuwa obiekt Rental o danym id	ADMIN

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/)](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";
```

Użyte technologie

Backend

- FastAPI - framework do tworzenia RESTowych API
- Pydantic - biblioteka do walidacji danych
- SQLAlchemy - biblioteka do mapowania obiektowo-relacyjnego
- Alembic - narzędzie do generowania i wykonywania migracji SQL
- PostgreSQL - silnik baz danych SQL
- Pytest - biblioteka do wykonywania testów jednostkowych
- Uvicorn - serwer ASGI (Asynchronous Server Gateway Interface) służący do uruchamiania kodu FastAPI
- Tenacity - biblioteka do "powtarzania" czynności okresowo i ponawiania po błędzie
- Passlib - biblioteka do hashowania i weryfikacji m.in. haseł
- psycopg2 - biblioteka do komunikacji z bazą PostgreSQL
- python-jose - biblioteka do obsługi m.in. tokenów JWT
- python-dotenv - biblioteka do ładowania zmiennych środowiskowych z plików .env
- pytz - biblioteka do obsługi stref czasowych
- fastapi-utils - biblioteka zawierająca usprawnienia do frameworka FastAPI, używana do wykonywania zadań cyklicznie

Frontend

- React.js - biblioteka/framework do budowania dynamicznych aplikacji webowych
- Material-ui - biblioteka zawierająca ostylowane komponenty Reacta, służąca do

budowania responsywnych interfejsów użytkownika
- Axios - biblioteka do wykonywania żądań HTTP
- Moment.js - biblioteka do zaawansowanej obsługi i formatowania dat
- React Router - biblioteka do obsługi routingu w aplikacjach napisanych w React.js
- react-json-view - biblioteka dodająca komponent służący do wyświetlania prostego edytora JSON, służąca w projekcie do wyświetlania błędów zwracanych przez API
- history - biblioteka umożliwiająca korzystanie z historii odwiedzionych URLi w aplikacji, umożliwia np. cofanie się

Podział kodu

Backend

- alembic (migracje SQL)
- app (główny folder aplikacji):
    - api (definicje kontrolerów RESTowych)
    - core (konfiguracja zmiennych środowiskowych i JWT)
    - db (konfiguracja połączenia z bazą danych)
    - exceptions (definicje wyjątków)
    - models (definicje modeli bazodanowych)
    - schemas (definicje obiektów - Pydantic)
    - services (definicje serwisów implementujących logikę biznesową)
    - tests (definicje testów):
        - api (testy endpointów)
        - services (testy serwisów)
        - utils (metody pomocnicze, używane w testach)
    - utils (definicje metod pomocnicznych)
    - validators (definicje szeroko pojętych walidatorów - np. walidator dostępności samochodu w podanych datach)
- scripts (skrypty pomocnicze)

Frontend

- public (definicja index.html, stałych zasobów, loga, ikon itd.)
- src: (główny folder aplikacji)
    - components (definicje reużywalnych komponentów)
    - context (definicja kontekstów Reacta, aktualnie jedynie AuthContext)
    - layouts (definicje dwóch układów kompozycyjnych, Main i DashboardLayout oraz sekcji nawigacyjnej)
    - service (definicje serwisów odpowiadających za pobieranie i wysyłanie danych do backendu przez HTTP)
    - theme (definicja motywu material-ui aplikacji)
    - utils (definicje metod pomocnicznych)
    - views (definicje widoków, każdy widok odpowiada za jedną ścieżkę np. /app/cars => CarsListView.js)

Ważniejsze przykłady kodu

Backend

- **CarsSearchQuery (app/schemas/cars_search_query.py)**

```
import abc
from datetime import datetime
from typing import List, Optional

from pydantic import BaseModel
from sqlalchemy.sql.elements import BinaryExpression

from app.models import Car
from app.models.car import AcType, CarType, DriveType, FuelType,
GearboxType


class RangeCriterion(BaseModel):
    __metaclass__ = abc.ABCMeta

    @abc.abstractmethod
    def to_condition(self) -> BinaryExpression:
        raise NotImplementedError()


class NumberOfPassengersRange(RangeCriterion):
    start: int
    end: int

    def to_condition(self) -> BinaryExpression:
        return Car.number_of_passengers.between(self.start, self.end)


class PricePerDayRange(RangeCriterion):
    start: float
    end: float

    def to_condition(self) -> BinaryExpression:
        return Car.price_per_day.between(self.start, self.end)


class AvailabilityDatesRange(BaseModel):
    start: datetime
    end: datetime


class CarsSearchQuery(BaseModel):
    model_name: Optional[str] = None
    type: Optional[CarType] = None
    fuel_type: Optional[FuelType] = None
```

```
    gearbox_type: Optional[GearboxType] = None
    ac_type: Optional[AcType] = None
    drive_type: Optional[DriveType] = None
    number_of_passengers: Optional[NumberOfPassengersRange] = None
    price_per_day: Optional[PricePerDayRange] = None
    availability_dates: Optional[AvailabilityDatesRange] = None

    def to_conditions(self) -> List[BinaryExpression]:
        """
        Returns list of SQLAlchemy filter conditions based on query
object values
        """
        conditions = []
        for field_name in CarsSearchQuery.__fields__.keys():
            value = getattr(self, field_name)
            if value is not None:
                if isinstance(value, RangeCriterion):
                    conditions.append(value.to_condition())
                elif isinstance(
                    value, str
                ):  # use ilike on str fields instead of exact match
                    conditions.append(getattr(Car, field_name).ilike(f"%
{value}%"))
                elif isinstance(value, AvailabilityDatesRange):  # skip
                    pass
                else:
                    conditions.append(getattr(Car, field_name) == value)
        return conditions
```

Powyższa klasa jest używana do budowania kryteriów wyszukiwania samochodów. Na podstawie typu kryterium, metoda *to_conditions()* dodaje odpowiedni blok, który następnie jest przekazywany do metody filter z biblioteki *sqlalchemy*, co w rezultacie produkuje kwerendę SQL filtrującą odpowiednie wiersze z tabeli cars.

Kryterium może być obiektem abstrakcyjnej klasy *RangeCriterion*, co powoduje dodanie do finalnej kwerendy operatora **BETWEEN** porównującego, czy podana wartość mieści się w danym zakresie.

Ponadto, dla kryteriów typu str, zamiast standardowego porównania ==, metoda *to_conditions()* używa funkcji sql **ILIKE**, która przyrównuje dwa łańcuchy tekstowe, pomijając wielkość liter i działa jak pythonowy operator *in*.

- **BaseService (app/services/base.py)**

```
from typing import Any, Generic, List, Optional, Type, TypeVar

from fastapi.encoders import jsonable_encoder
from pydantic import BaseModel
from sqlalchemy.orm import Session

from app.db.base_class import Base
```

```
ModelType = TypeVar("ModelType", bound=Base)
CreateSchemaType = TypeVar("CreateSchemaType", bound=BaseModel)
UpdateSchemaType = TypeVar("UpdateSchemaType", bound=BaseModel)


class BaseService(Generic[ModelType, CreateSchemaType,
UpdateSchemaType]):
    def __init__(self, model: Type[ModelType]):
        """
        Base service object with default methods to Create, Read, Update,
Delete (CRUD).

        **Parameters**

        * `model`: A SQLAlchemy model class
        * `schema`: A Pydantic model (schema) class
        """
        self.model = model

    def get(self, db: Session, _id: Any) -> Optional[ModelType]:
        return db.query(self.model).get(_id)

    def get_all(self, db: Session) -> List[ModelType]:
        return db.query(self.model).all()

    def create(self, db: Session, *, obj_in: CreateSchemaType) ->
ModelType:
        obj_in_data = jsonable_encoder(obj_in)
        db_obj = self.model(**obj_in_data)  # type: ignore
        db.add(db_obj)
        db.commit()
        db.refresh(db_obj)
        return db_obj

    def update(
        self, db: Session, *, db_obj: ModelType, obj_in: UpdateSchemaType
    ) -> ModelType:
        obj_data = jsonable_encoder(db_obj)
        update_data = obj_in.dict(exclude_unset=True)
        for field in obj_data:
            if field in update_data:
                setattr(db_obj, field, update_data[field])
        db.add(db_obj)
        db.commit()
        db.refresh(db_obj)
        return db_obj

    def remove(self, db: Session, *, _id: int) -> ModelType:
        obj = db.query(self.model).get(_id)
        db.delete(obj)
        db.commit()
        return obj
```

BaseService to bazowy serwis obsługujący podstawowe metody typu CRUD (Create, Read, Update, Delete). Jako argumenty przyjmuje ona typy modeli, na których będzie pracować. Są to kolejno:

- ModelType (typ modelu bazodanowego)
- CreateSchemaType (typ schema tworzącego obiekt - CreateDto)
- UpdateSchemaType (typ schema aktualizującego obiekt - UpdateDto)

Klasa ta jest wykorzystywana w każdy serwisie, a w szczególności w CustomersService, który to serwis nie wymaga żadnych funkcjonalności, poza oferowanymi przez klasę BaseService, przez co znacząco zmniejszyłem duplikację kodu:

```
from app.models.customer import Customer
from app.schemas.customer import CustomerCreateDto, CustomerUpdateDto
from app.services.base import BaseService


class CustomerService(BaseService[Customer, CustomerCreateDto,
CustomerUpdateDto]):
    pass


customer = CustomerService(Customer)
```

- **Interval (app/utils/interval.py)**

```python
from datetime import datetime


class Interval:
    def __init__(self, start: datetime, end: datetime):
        self.start = start
        self.end = end

    @property
    def start(self):
        return self._start

    @start.setter
    def start(self, start):
        self._start = start

    @property
    def end(self):
        return self._end

    @end.setter
    def end(self, end):
        self._end = end

    def is_intersecting(self, other) -> bool:
        if (
            self.start.date() == other.end.date()
            or self.end.date() == other.start.date()
        ):
            return True
        return (self.start <= other.start <= self.end) or (
            other.start <= self.start <= other.end
        )
```

Klasa Interval reprezentuje odcinek czasu pomiędzy start a end (włącznie). Posiada ona metodę *is_intersecting()* która przyjmuje drugi obiekt klasy Interval i zwraca *True* jeżeli interwały się przecinają. Jest ona używana przy sprawdzaniu dat dostępności samochodów oraz przy sprawdzaniu kolizji wypożyczeń i rezerwacji.

- **Availability Validator (app/validators/availability.py)**

```python
from datetime import datetime

from sqlalchemy.orm import Session

from app import services
from app.utils.interval import Interval


def is_car_available_in_dates(
    db: Session,
    car_id: int,
```

```
    start_date: datetime,
    end_date: datetime,
    rental_id: int = None,
    reservation_id: int = None,
) -> bool:
    available = True
    timeframe = Interval(start_date, end_date)

    if is_colliding_with_other_rentals(db, car_id, timeframe, rental_id):
        available = False

    if is_colliding_with_other_reservations(db, car_id, timeframe,
reservation_id):
        available = False

    return available


def is_colliding_with_other_rentals(
    db: Session, car_id: int, timeframe: Interval, rental_id: int = None
):
    available = True

    rentals_for_this_car = get_rentals_for_this_car(db, car_id,
rental_id)
    for other_rental in rentals_for_this_car:
        other_rental_timeframe = Interval(
            other_rental.start_date, other_rental.end_date
        )
        if timeframe.is_intersecting(other_rental_timeframe):
            available = False
            break

    return not available


def is_colliding_with_other_reservations(
    db: Session, car_id: int, timeframe: Interval, reservation_id: int =
None
):
    available = True

    reservations_for_this_car = get_reservations_for_this_car(
        db, car_id, reservation_id
    )
    for other_reservation in reservations_for_this_car:
        other_reservation_timeframe = Interval(
            other_reservation.start_date, other_reservation.end_date
        )
        if timeframe.is_intersecting(other_reservation_timeframe):
            available = False
            break
```

```
    return not available


def get_rentals_for_this_car(db: Session, car_id: int, rental_id: int =
None):
    rentals_for_this_car = services.rental.get_active_by_car_id(db,
car_id)
    if rental_id:
        rentals_for_this_car = [
            rental for rental in rentals_for_this_car if rental.id !=
rental_id
        ]
    return rentals_for_this_car


def get_reservations_for_this_car(db: Session, car_id: int,
reservation_id: int = None):
    reservations_for_this_car =
services.reservation.get_active_by_car_id(db, car_id)
    if reservation_id:
        reservations_for_this_car = [
            reservation
            for reservation in reservations_for_this_car
            if reservation.id != reservation_id
        ]
    return reservations_for_this_car
```

Powyższy plik zawiera metody pomocnicze służące do walidacji dostępności samochodu w podanych datach. Są one używane przez serwisy samochodów, rezerwacji i wypożyczeń.

Metody *get_rentals_for_this_car* i *get_reservations_for_this_car* przyjmują opcjonalnie identyfikatory obiektów, dla których sprawdzana jest dostępność.
Jeżeli identyfikatory te są przekazane do wywołania metod, to rekordy o podanych identyfikatorach są odfiltrowywane przed zwróceniem listy obiektów.

Jest to używane przy walidacji aktualizacji obiektów, np. aktualizując rezerwację *xyz*, sprawdzamy kolizje ze wszystkimi obiektami rezerwacji poza samą rezerwacją *xyz*.