Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
Ilkovičova 2, 842 16 Bratislava 4

Predmet / Subject

– Databázové systémy / Database systems –

- Dokumentácia / Documentation -

Zadanie č.5

Ak. Rok / Academic term: 2022/2023, letný semester

Cvičiaci / Instructors:
Ing. Jakub Dubec

Študent / Student:
Adam Grík

Bratislava, 2023.

Obsah

Úvod

Úlohou tohto zadania bolo implementovať API rozhranie nad návrhom databázy zo zadania č. 4. Štruktúra datbázy sa samozrejme mohla zmeniť podľa potreby implementácie. Bolo potrebné implementovať všetky endpointy, ktoré boli uvedené v zadanej API špecifikácií. Následné bolo potrebné danú API aplikáciu otestovať na školskom testeri, kde nato aby bola aplikácia funkčá sme museli vytvoriť pred vykonávaním daných endpointov aj migrácie, ktoré nám zabezpečili aby sa daná štruktúra databázy vytvorila na serveri testra.

Danú API aplikáciu som implementoval v jazyku Python, konkrétne s ORM knižnicami SQLAlchemy a Pydantic.

1 Zmeny oproti návrhu databázy

V zadaní č.5 som neimplementoval všetky tabuľky, ktoré sa nachádzajú v návrhu databázy, implementoval som len tie tabuľky, ktoré boli potrebné na vykonanie endpointov. Okrem neimplementovania niektorých tabuliek, zmeny nastali aj v premenovaní niektorých tabuliek. Konkrétne tabuľka *borrowings* bola premenovaná na *rentals,* a tabuľka *offline_copies* bola premenovaná na *instances.* Takisto neboli implementované všetky stĺpce, ktoré sa nachádzajú v návrhu databázy a sú implementované iba tie stĺpce, ktoré sú uvedené v API špecifikáacií.

2 Setup pripojenia na databázu

```
from dbs_assignment.config import settings

database_host = os.getenv("DATABASE_HOST")
database_port = os.getenv("DATABASE_PORT")
database_name = os.getenv("DATABASE_NAME")
database_user = os.getenv("DATABASE_USER")
database_password = os.getenv("DATABASE_PASSWORD")

engine = create_engine(f"postgresql://{database_user}:{database_password}@{database_host}:{database_port}/{database_name}", echo=True, pool_pre_p

if not database_exists(engine.url):
    create_database(engine.url)

Base = declarative_base()

SessionLocal = sessionmaker(bind=engine)
```

Pripojenie na databázu prebieha tak, že si vytiahnem *enviroment* premenné buď tie na ktoré sa pripája docker container, alebo tie ktoré mám uvedené v súbore *.env.*

3 Setup migrácií

```python
class User(Base):
    __tablename__ = 'users'
    id = Column(UUID(as_uuid=True), primary_key=True, index=True, unique=True, nullable=False, autoincrement=False)
    name = Column(String(255), nullable=False)
    surname = Column(String(255), nullable=False)
    email = Column(EmailType, nullable=False, unique=True)
    birth_date = Column(Date, nullable=False)
    personal_identificator = Column(String, nullable=False)
    reservations = relationship('Reservation', back_populates='user')
    rentals = relationship('Rental', back_populates='user')
    created_at = Column(DateTime, default=datetime.datetime.utcnow())
    updated_at = Column(DateTime, default=datetime.datetime.utcnow(), onupdate=datetime.datetime.utcnow())

class Card(Base):
    __tablename__ = 'cards'
    id = Column(UUID(as_uuid=True), primary_key=True, index=True, unique=True, nullable=False, autoincrement=False)
    user_id = Column(UUID(as_uuid=True), nullable=False)
    magstripe = Column(String(255), nullable=False)
    status = Column(Enum('active', 'inactive', 'expired', name='enum'), nullable=False)
    created_at = Column(DateTime, default=datetime.datetime.utcnow())
    updated_at = Column(DateTime, default=datetime.datetime.utcnow(), onupdate=datetime.datetime.utcnow())
```

Jednotlivé tabuľky, ktoré chcem pri spustení aplikácie zmigrovať vytváram ako *Base* triedy vrámci ORM v knižnici SQLAlchemy.

```python
from dbs_assignment.database import Base,engine
from dbs_assignment.models import *

Base.metadata.create_all(engine)
```

Následne to v súbore *create_db.py* všetko spúšťam vďaka jednému príkazu.

4 Opis použitých SQL dopytov

4.1 Všetky GET metódy

```python
@router.get("/instances/{instance_id}", status_code=status.HTTP_200_OK, description="Instance was found", response_model=Instance)
def get_instance(instance_id: str):
    if not db.query(models.Instance).filter(models.Instance.id == instance_id).first():
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Instance not found")

    instance = db.query(models.Instance).filter(models.Instance.id == instance_id).first()

    return instance
```

V mojej implementácií vyzerajú všetky metódy GET rovnako. A to tak, že sa zavolá jeden SQL dopyt, ktorý nájde na základe ID daný záznam v tabuľke, a vráti konkrétny zaznám na základe Pydantic modelu. V prípade ak sa záznam s daným ID nenachádza v databáze tak aplikácia vráti eror 404.

4.2 Metóda GET *(publications)*

Jediná metóda GET, ktorá je odlišná od všetkých ostatným tabuliek je v tabuľke *publications*.

```python
@router.get("/publications/{publication_id}", status_code=status.HTTP_200_OK, description="Publication was found", response_model=Publication)
def get_publication(publication_id: str):
    if not db.query(models.Publication).filter(models.Publication.id == publication_id).first():
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Publication not found")

    publication_id = db.query(models.Publication.id).filter(models.Publication.id == publication_id).scalar()
    publication_title = db.query(models.Publication.title).filter(models.Publication.id == publication_id).scalar()

    authors = select(models.publication_authors.c.author_id).filter(models.publication_authors.c.publication_id == publication_id)
    result_authors = db.execute(authors)
    results_authors = result_authors.scalars().all()

    categories = select(models.publication_categories.c.category_id).filter(models.publication_categories.c.publication_id == publication_id)
    result_categories = db.execute(categories)
    results_categories = result_categories.scalars().all()

    created_at = db.query(models.Publication.created_at).filter(models.Publication.id == publication_id).scalar()
    updated_at = db.query(models.Publication.updated_at).filter(models.Publication.id == publication_id).scalar()


    data = {"id": publication_id,  "title": publication_title, "authors": [], "categories": [], "created_at": created_at, "updated_at": updated_at}
    for author in results_authors:
        author_name = db.query(models.Author.name).filter(models.Author.id == author).scalar()
        author_surname = db.query(models.Author.surname).filter(models.Author.id == author).scalar()
        data["authors"].append({"name": author_name, "surname": author_surname})
    for category in results_categories:
        category_name = db.query(models.Category.name).filter(models.Category.id == category).scalar()
        data["categories"].append(category_name)

    return data
```

Ako prvé si najprv získam ID a *title* hľadanej publikácie. Následne si získam všetkých autorov a katogóre z asociačných tabuliek *publication_authors* a *publication_categories*, ktoré su viazané s ID danej publikácie. Ako posledné už len v cykloch for prechádzam cez všetkých získaných autorov a kategórie a vkladám všetko do JSON objektu *data*, ktorý potom vraciam ako response v danom endpointe.

4.3 Všetky DELETE metódy

```python
@router.delete("/instances/{instance_id}", status_code=status.HTTP_204_NO_CONTENT)
def delete_instance(instance_id: str):
    if not db.query(models.Instance).filter(models.Instance.id == instance_id).first():
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Instance not found")

    instance_to_delete = db.query(models.Instance).filter(models.Instance.id == instance_id).first()

    db.delete(instance_to_delete)
    db.commit()
```

Tak ako pri metóde GET aj pri metóde DELETE sú všetky pri každom endpointe rovnaké, a to tak že si nájdem na základe ID ktorý záznam chcem vymazať a následne ho vďaka príkazu *db.commit* vymažem.

Rozdiel je opäť len pri tabuľke *publications*, kde keď mažem danú publikáciu, tak vždy pozerám či existujú aj nejaké záznamy v tabuľke *instances*, ktoré sa viažu na publikáciu ktorú chcem vymazať.

4.4 Metóda POST (*authors, cards, categories, instances, reservations a users)*

```python
@router.post("/users", response_model=User, status_code=status.HTTP_201_CREATED)
def add_user(user: User):

    if user.id is None or user.name is None or user.surname is None or user.email is None or user.birth_date is None or user.personal_identificator is None:
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail="Missing required information")
    if user.id == "" or user.name == "" or user.surname == "" or user.email == "" or user.birth_date == "" or user.personal_identificator == "":
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail="Bad request")
    if db.query(models.User).filter(models.User.email == user.email).first():
        raise HTTPException(status_code=status.HTTP_409_CONFLICT, detail="Email already taken")


    new_item = models.User(
        id = user.id,
        name = user.name,
        surname = user.surname,
        email = user.email,
        birth_date = user.birth_date,
        personal_identificator = user.personal_identificator,
        created_at = datetime.datetime.utcnow(),
        updated_at = datetime.datetime.utcnow()
    )

    db.add(new_item)
    db.commit()
    db.rollback()

    return new_item
```

Metóda POST je pri tabuľkách *authors, cards, categories, instances, reservations a user* v podstate vždy totožná. Ako prvé si vždy v podmienkach skontrolujem či sú vyplnené všetky povinné údaje, či sú vyplnené správne, či napr. v tomto prípadne už existuje zadaný mail a pod., podľa toho potom vraciam jednotlivé erory. Následne si vždy vytvorím inštanciu *new_item,* kde vložím všetký novovytvorené údaje a hodnoty. Potom už len nový záznam vložím do danej tabuľky, a vrátim novovytvorený objekt.

4.5 Metóda POST *(publications)*

```python
@router.post("/publications", status_code=status.HTTP_201_CREATED, description="Publication was created", response_model=Publication)
def create_publication(publication: Publication):
    if publication.title is None or publication.authors is None or publication.categories is None:
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail="Missing required information")
    if publication.title == "" or len(publication.authors) == 0 or len(publication.categories) == 0:
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail="Bad request")


    if publication.id is None:
        publication.id = uuid.uuid4()

    authors_list = []
    categories_list = []

    for item in publication.authors:
        if not db.query(models.Author).filter(models.Author.name == item["name"],models.Author.surname == item["surname"]).first():
            raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail="Author not exists")
        author = db.query(models.Author).filter(models.Author.name == item["name"], models.Author.surname == item["surname"]).first()
        authors_list.append(author)

    for item in publication.categories:
        if not db.query(models.Category).filter(models.Category.name == item).first():
            raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail="Category not exists")
        category = db.query(models.Category).filter(models.Category.name == item).first()
        categories_list.append(category)

    new_item = models.Publication(
        id=publication.id,
        title=publication.title,
        authors=authors_list,
        categories = categories_list,
    )

    db.add(new_item)
    db.commit()

    data = {"id": new_item.id, "title": new_item.title, "authors": [], "categories": [], "created_at": new_item.created_at, "updated_at": new_item.updated_at}
```

V metóde POST pri tabuľke *publications*, ako pri všetkých ostaných metódach POST najprv skontrolujem všetky možné erory, ktoré môžu nastať. Ako ďalšie doplním UUID, ak nebolo požívateľom zadané. Následne si v dvoch cykloch *for* doplním všetkých autorov a kateorie, ktoré boli zadané na vstupe, vďaka knižnici SQLAlchemy sa mi ID autorov, kategórií a publikácií automaticky doplnia do asociačných tabuliek *publication_authors* a *publication_categories*. Potom pomocou inštancie *new_item* vložím všetky zadané údaje do modelu *Publication*. A potom ako posledné už len pridám novovytvorený záznam do databázy a vytváram JSON objekt, ktorý následne vraciam ako response.

4.6 Metóda POST (*rentals*)

```python
@router.post("/rentals", status_code=status.HTTP_201_CREATED, response_model=RentalResponse, description="Rental was created")
def create_rental(rental: Rental):

    if rental.user_id is None or rental.publication_id is None or rental.duration is None:
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail="Missing required information")

    if rental.duration > 14:
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail="Maximal duration is 14 days")


    if rental.id is None:
        rental.id = uuid.uuid4()

    publication_instances = select(models.Instance.id).filter(models.Instance.publication_id == rental.publication_id).where(models.Instance.status == "available")
    result_publication_instances = db.execute(publication_instances)
    available_publication_instances = result_publication_instances.scalars().all()

    if len(available_publication_instances) == 0:
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail="Bad request")

    get_reservations = db.query(models.Reservation.user_id).filter(models.Reservation.publication_id == rental.publication_id).order_by(models.Reservation.created_at.asc())
    result_reservations = db.execute(get_reservations)
    reservations_users = result_reservations.scalars().all()
```

Metóda POST pre tabuľku *rentals* je najobsiahlejšia zo všetkých metód. Ako prvé opäť ošetrím všetky možné erory, ktoré môžu nastať a doplním si UUID ak nie je zadané používateľom na vstupe. Potom si pomcou *SELECTU-u* vytiahnem všetky dostupné

inštancie, ktoré sú viazané na ID publikácie ktoré je zadané na vstupe, ak nie je žiadna dostupná inštancia publikácie tak vypíšem eror *Bad request.* Následne si ešte vytiahnem všetky rezervácie, ktoré sú viazané na danú publikáciu.

```
if len(reservations_users) > 0:
    if len(reservations_users) >= len(available_publication_instances):
        pom = 1
        for reservation in reservations_users:
            reservation_to_delete = db.query(models.Reservation).filter(models.Reservation.publication_id == rental.publication_id,
                                                                        models.Reservation.user_id == reservation).first()
            if (reservation == rental.user_id and pom <= len(available_publication_instances) and reservation_to_delete):
                # db.delete(reservation_to_delete)
                # db.commit()
                can_create_rental = True
            pom = pom + 1
    if len(reservations_users) < len(available_publication_instances):
        for reservation in reservations_users:
            reservation_to_delete = db.query(models.Reservation).filter(models.Reservation.publication_id == rental.publication_id,
                                                                        models.Reservation.user_id == reservation).first()
            # if (reservation == rental.user_id and reservation_to_delete):
                # db.delete(reservation_to_delete)
                # db.commit()
        can_create_rental = True
if len(reservations_users) == 0:
    can_create_rental = True
```

V ďalšej časti tejto funkcie vykonávam viacúrovňovú podmienku *if*. Ak existujú rezervácie, ktoré sa viažu na danú publikáciu, tak potom ešte porovnávam či je počet rezervácií väčší ako počet dostupných inštancií, ak je tak postupne prechádzam cez všetky konkrétne rezervácie ktoré sú zoradené podľa dátumu, a ak sa rezervácia toho používateľa ktorý chce vykonať výpožičku nachádza na mieste menšom alebo rovnom ako je počet dostupných inštancií tak metóda POST môže byť vykonaná. A logicky ak je počet rezervácií menší ako počet dostupných inštancií tak výpožička može byť vykonaná. Tak isto aj keď neexistujú žiadne rezervácie, tak výpožička môže byť automaticky vytvorená.

```
first_available_instance_id = None
for first_instance in available_publication_instances:
    first_available_instance_id = first_instance
    break

user = db.query(models.User).filter(models.User.id == rental.user_id).first()

new_item = models.Rental(
    id = rental.id,
    user_id = rental.user_id,
    publication_instance_id = first_available_instance_id,
    duration = rental.duration,
    start_date = date.today(),
    end_date = date.today() + timedelta(days=rental.duration),
    status = "active",
    user = user
)

db.add(new_item)
db.commit()
```

A následne už ako pri každej metóde POST iba vložím zadané hodnoty do inštancie *new_item* a vložím nový záznam do databázy.

4.7 Metóda PATCH *(authors, cards, categories, instances, rentals, users)*

```python
@router.patch("/users/{user_id}", response_model=User, status_code=status.HTTP_200_OK, description="User updated")
def update_user(user_id: str, user: User):
    if not db.query(models.User).filter(models.User.id == user_id).first():
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="User not found")
    if user.id == "" or user.name == "" or user.surname == "" or user.email == "" or user.birth_date == "" or user.personal_identificator == "":
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail="Bad request")

    user_to_update = db.query(models.User).filter(models.User.id == user_id).first()


    if(user_to_update.email != user.email):
        if db.query(models.User).filter(models.User.email == user.email).first():
            raise HTTPException(status_code=status.HTTP_409_CONFLICT, detail="Email already taken")

    if not user.name is None:
        user_to_update.name = user.name
    if not user.surname is None:
        user_to_update.surname = user.surname
    if not user.email is None:
        user_to_update.email = user.email
    if not user.birth_date is None:
        user_to_update.birth_date = user.birth_date
    if not user.personal_identificator is None:
        user_to_update.personal_identificator = user.personal_identificator

    db.commit()

    return user_to_update
```

Metóda PATCH je pri tabuľkách *authors, cards, category, instances, rentals a users* v podstate rovnaká. Vždy si ako prvé zistím či daný záznam ktorý chceme updatovať existuje, ak nie vypíšeme 404 eror. Následne kontrolujeme aj ďalšie iné podmienky ako napr. v tomto prípade, či nám používateľ nezadal prázdne *"stringy"*, prípadne kontrolujem či už daný email existuje, prípadne v iných tabuľkách iné hodnoty či sú správne alebo či už existujú. Ako ďalšie si potom nájdem záznam ktorý chcem updatovať a postupne mu updatujem všetky hodnoty, ale iba tie ktoré boli zadané na vstupe čo zisťujem zakaždým vďaka podmienke napr. "i*f not user.name is None:".*

4.8 Metóda PATCH *(publications)*

```python
@router.patch("/publications/{publication_id}", status_code=status.HTTP_200_OK, description="Publication was updated")
def update_publication(publication_id: str, publication: Publication):
    if not db.query(models.Publication).filter(models.Publication.id == publication_id).first():
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Publication not found")

    publication_to_update = db.query(models.Publication).filter(models.Publication.id == publication_id).first()

    if not publication.title is None:
        if publication.title == "":
            raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail="Bad request")

        publication_to_update.title = publication.title
        publication_to_update.updated_at = datetime.datetime.utcnow()

    authors_list = []
    categories_list = []

    if not publication.authors is None:
        if len(publication.authors) == 0:
            raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail="Bad request")
        for item in publication.authors:
            if not db.query(models.Author).filter(models.Author.name == item["name"],models.Author.surname == item["surname"]).first():
                raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail="Author not exists")
            author = db.query(models.Author).filter(models.Author.name == item["name"], models.Author.surname == item["surname"]).first()
            authors_list.append(author)
        publication_to_update.authors = authors_list
        publication_to_update.updated_at = datetime.datetime.utcnow()
```

Metóda PATCH pri tabuľke *publications* je v podstate skoro rovnaká ako metóda POST. Na začiatku vždy skontrolujem či daná publikácia existuje, a následne si nájdem tú publikáciu ktorú chcem updatovať. Ak bol na vstupe zadaný titul tak ako prvý zmením titul danej publikácie. Ako ďalšie updatujem všetkých autorov zadaných používateľom na vstupe.

```
if not publication.categories is None:
    if len(publication.categories) == 0:
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail="Bad request")
    for item in publication.categories:
        if not db.query(models.Category).filter(models.Category.name == item).first():
            raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail="Category not exists")
        category = db.query(models.Category).filter(models.Category.name == item).first()
        categories_list.append(category)
    publication_to_update.categories = categories_list
    publication_to_update.updated_at = datetime.datetime.utcnow()

db.commit()

data = {"id": publication_to_update.id, "title": publication_to_update.title, "authors": [], "categories": [],
        "created_at": publication_to_update.created_at, "updated_at": publication_to_update.updated_at}
```

Potom robím to isté aj so zadanými kategóriami a updatujem ich v danej publikácií. A nakoniec už si len vyskladávam JSON objekt *data*, kde vkladám údaje o updatovanom objekete publikácie.