SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Databázové systémy
Zadanie 5

Samuel Švenk

Cvičenie: Štv 8:00/9:50
Cvičiaci: Ing. Jakub Dubec

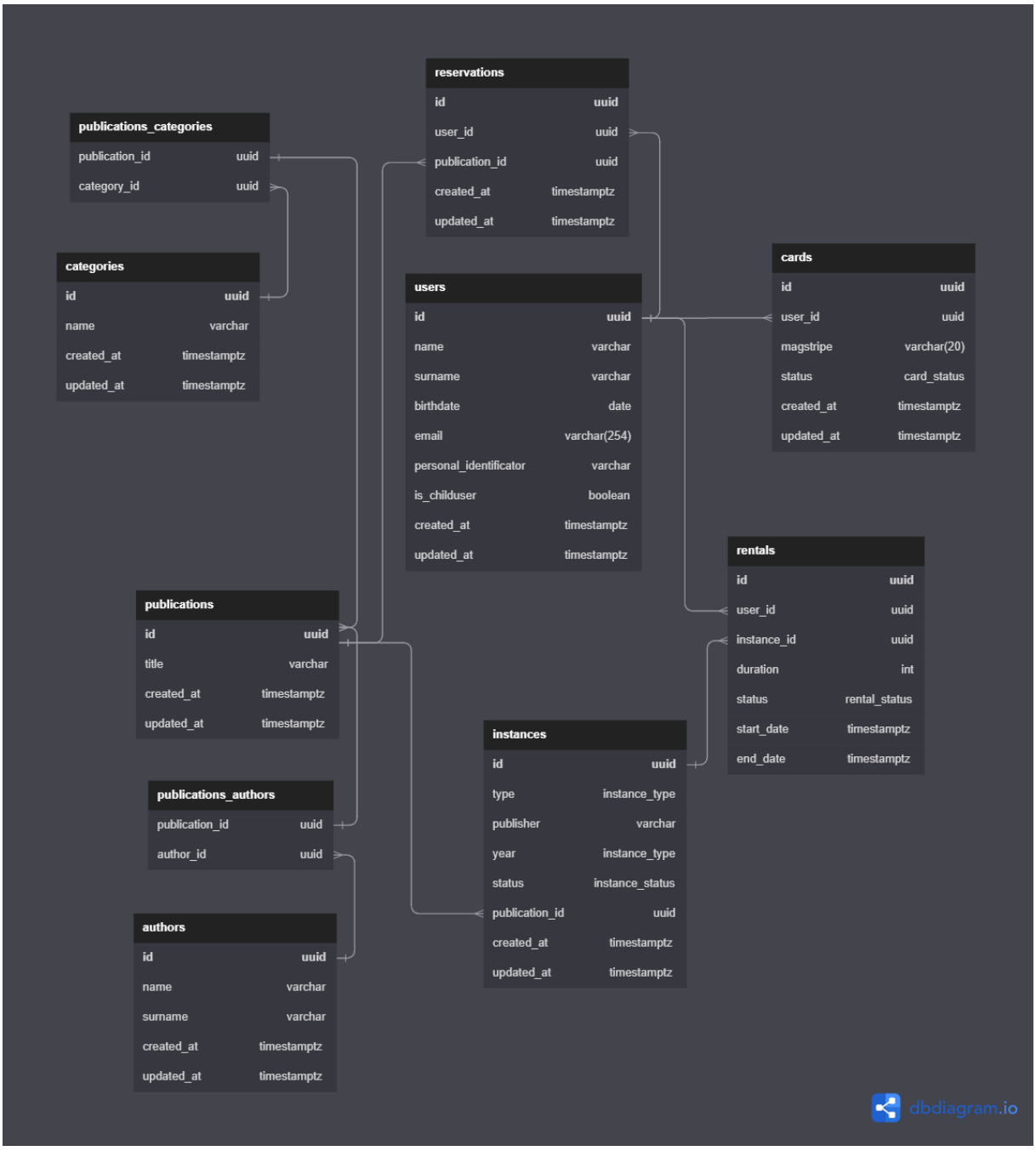
7.5.2023

Obsah

Popis zadania

V tomto zadaní sme mali implementovať návrh svojho návrhu zo Zadania 4, keďže môj návrh bol veľmi nedostačujúci pracoval som hlavne s poskytnutými endpointmi. To znamená som musel prekopať celý návrh napríklad pri zisťovaní o akého usera sa jedná či o detský účet alebo o účet takzvaného mainusera zisťujem tak, že ak má daný user viac ako 18 rokov, v table users atribút is_childuser nastavím na false, ak sa jedná o usera, ktorý nedovŕšil 18, bude táto hodnota true.

Fyzický návrh databázy



Takto som definoval svoje ENUMy:

```
enum card_status{
  active
  inactive
  expired
}

enum instance_type{
  physical
  ebook
  audiobook
}

enum instance_status{
  available
  reserved
}

enum rental_status{
  active
  returned
}
```

Endpoint Users

V tomto endpointe sa metódy POST,GET a PATCH, nachádza sa v users.py.

POST /users/

Takto napríklad vyzerá ošetrenie či je dospelý email unikátny, pri detských emailoch môže byť aj duplicitný.

```
# Je dospely email unikatny?
if not user.is_childuser and db.query(UserModel).filter_by(email=user.email).filter(UserModel.is_childuser==False).first() is not None:
    raise HTTPException(status_code=409, detail="Email Already Taken")
```

Takto či vložený dátum je v správnom formáte

```
try:
    datetime.strptime(user.birth_date, "%Y-%m-%d")
except ValueError:
    raise HTTPException(status_code=400, detail="Bad Request")
```

Takto nastavujem či je user detský alebo dospelý

```
# Je to child user?
user_birth_date = datetime.strptime(user.birth_date, "%Y-%m-%d")
if (datetime.now().year - user_birth_date.year) < 18:
    user.is_childuser = True
```

GET /users/{id}

GET je veľmi jednoduchý, keďže hľadáme usera pomocou jeho id, ak user neexistuje vrátim 404.

```
user = db.query(UserModel).filter(UserModel.id == id).first()
if user is None:
    raise HTTPException(status_code=404, detail="User Not Found")
```

Raw query:

```
SELECT * FROM users
WHERE id = {id}
LIMIT 1;
```

PATCH /users/{id}

Podobne ako pri GET hľadáme usera podľa jeho id, no rozidel je napríklad, že musíme testovať či je daný email už v databáze alebo sme dostali na vstup správne id. Ak sa nevyplnia všetky polia vložíme tam staré.

```
to_patch = db.query(UserModel).filter(UserModel.id == id).first()
if not to_patch:
    raise HTTPException(status_code=404, detail="User Not Found")

if not user.is_childuser and db.query(UserModel).filter_by(email=user.email).filter(UserModel.is_childuser==False).first() is not None:
    raise HTTPException(status_code=409, detail="Email Already Taken")
```

```
to_patch.name = user.name or to_patch.name
to_patch.surname = user.surname or to_patch.surname
to_patch.email = user.email or to_patch.email
to_patch.birth_date = user.birth_date or to_patch.birth_date
to_patch.personal_identificator = user.personal_identificator or to_patch.personal_identificator
to_patch.updated_at = datetime.now()
to_patch.is_childuser = user.is_childuser or to_patch.is_childuser
```

Raw query je rovnaká.

Endpoint Cards

Endpoint Cards funguje veľmi podobne s rozdielom že máme tu aj DELETE na odstránenie karty užívateľa.

POST /cards/

Najdôležitejšie tu je že ak nám nie je poskytnuté unikátne id karty tak si ho vygenerujeme.

```
if not card.id:
    card.id = str(uuid4())
```

GET /cards/{id}

GET karty je takmer rovnaký ako pri user, ak nedostaneme správne id vyhodíme 404.

```
card = db.query(CardModel).filter(CardModel.id == id).first()
if card is None:
    raise HTTPException(status_code=404, detail="Not Found")
```

Raw query:

```
SELECT *
FROM card
WHERE card.id = {id}
LIMIT 1;
```

PATCH /cards/{id}

Pri PATCH môže nastať že dostaneme zlý formát card.status, okrem toho postupujeme ako pri users, tak že za nedoplnené doplníme staré.

```
if card.status not in ["active", "inactive", "expired"]:
    raise HTTPException(status_code=400, detail="Bad Request")

to_patch = db.query(CardModel).filter(CardModel.id == id).first()
to_patch.magstripe = card.magstripe or to_patch.magstripe
to_patch.status = card.status or to_patch.status
to_patch.updated_at = datetime.now()
```

Raw query je rovnaká.

DELETE /cards/{id}

V DELETE postupujem rovnako ako pri GET alebo PATCH, podľa card.id vymažem danú kartu z databázy

```
to_delete = db.query(CardModel).filter(CardModel.id == id).first()
db.delete(to_delete)
db.commit()
return
```

Endpoint Authors

Endpoint Authors je taktiež veľmi jednoduchá ako predošlé. Tiež má POST,GET,PATCH a DELETE.

POST /authors /

Zaujímavé na tomto POSTe je ak existuje autor s rovnakým meno a priezvyskom nechcem ho zapísať.

```
    # Check na kombinaciu mena a priezviska
if db.query(AuthorModel).filter(AuthorModel.name == author.name).filter(AuthorModel.surname == author.surname).first():
    raise HTTPException(status_code=409, detail="Conflict")
```

Raw query:

```
SELECT *
FROM author
WHERE name = author_name AND surname = author_surname
LIMIT 1;
```

Takto checkujem ako pri karte ak nedostaneme author.id vygenerujeme si ho, ak už máme autora s tým istým id máme konflikt.

```
if not author.id:
    author.id = str(uuid4())
else:
    # Check na id
    if db.query(AuthorModel).filter(AuthorModel.id == author.id).first():
        raise HTTPException(status_code=409, detail="Conflict")
```

GET /authors/{id}

GET funguje ako predošlé podľa id dostanem autora, ak zadáme nesprávne id vyhodím 404.

```
author = db.query(AuthorModel).filter(AuthorModel.id == id).first()
if author is None:
    raise HTTPException(status_code=404, detail="Not Found")
```

Raw query:

```
SELECT *
FROM author
WHERE id = {id}
LIMIT 1;
```

PATCH /authors/{id}

PATCH funguje rovnako ako pri karte ak nedostaneme napríklad meno, doplníme staré. Tiež checkujeme či sme zadali správne id autora.

```
to_patch = db.query(AuthorModel).filter(AuthorModel.id == id).first()
to_patch.name = author.name or to_patch.name
to_patch.surname = author.surname or to_patch.surname
to_patch.updated_at = datetime.now()
```

Raw query je rovnaká

DELETE /authors/{id}

DELETE je najjednoduchší keďže tou istou query iba nájdeme autora a vymažeme ho.

```
to_delete = db.query(AuthorModel).filter(AuthorModel.id == id).first()
db.delete(to_delete)
db.commit()
return
```

Raw query je rovnaká

Endpoint Categories

Endpoint kategórie je takmer rovnaký ako Authors. Má POST,GET,PATCH a DELETE

POST /categories/{id}

Rovnaky postup ako pri authors, okrem toho ak máme rovnaké meno kategórie vyhodíme konflikt.

```python
if not category.id:
    category.id = str(uuid4())


if db.query(CategoryModel).filter(CategoryModel.name == category.name).first():
    raise HTTPException(status_code=409, detail="Conflict")
```

Raw query:

```sql
SELECT *
FROM category
WHERE category.name = 'category_name'
LIMIT 1;
```

GET /categories/{id}

Rovnaký check ako pro authors

```python
category = db.query(CategoryModel).filter(CategoryModel.id == id).first()

if category is None:
    raise HTTPException(status_code=404, detail="Not Found")
```

Raw query:

```sql
SELECT *
FROM category
WHERE id = {id}
LIMIT 1;
```

PATCH /categories/{id}

Rovnako ako pri authors

```
if not db.query(CategoryModel).filter(CategoryModel.id == id).first():
    raise HTTPException(status_code=404, detail="Not Found")

# Check ci uz mame categoriu s tymto menom
if db.query(CategoryModel).filter(CategoryModel.name == category.name).first():
    raise HTTPException(status_code=409, detail="Conflict")


to_patch = db.query(CategoryModel).filter(CategoryModel.id == id).first()
```

Raw query rovnaké.

DELETE /categories/{id}

Rovnako vymažem ako pri authors

```
if not db.query(CategoryModel).filter(CategoryModel.id == id).first():
    raise HTTPException(status_code=404, detail="Not Found")

to_delete = db.query(CategoryModel).filter(CategoryModel.id == id).first()
db.delete(to_delete)
db.commit()
return
```

Endpoint Publications

V tomto endpointe som implementoval iba POST a GET.

POST /publications/{id}

Pri poste napĺňam polia authors a catogories.

```python
# Add existing authors to the publication
for author in publication.authors:
    existing_author = db.query(AuthorModel).filter(AuthorModel.name == author["name"], AuthorModel.surname == author["surname"]).first()
    if existing_author is None:
        raise HTTPException(status_code=400, detail="Not Found")
    else:
        new_publication.authors.append(existing_author)

# Add existing categories to the publication
for category in publication.categories:
    existing_category = db.query(CategoryModel).filter(CategoryModel.name == category).first()
    if existing_category is None:
        raise HTTPException(status_code=400, detail="Not Found")
    else:
        new_publication.categories.append(existing_category)
```

Raw queries:

```sql
SELECT *
FROM author
WHERE name = 'author_name' AND surname = 'author_surname'
LIMIT 1;
```

```sql
SELECT *
FROM category
WHERE name = 'category_name'
LIMIT 1;
```

GET /publications/{id}

V GET vypisujem polia autorov a kategórií query je rovnaká ako v predošlom GET.

```python
publication = db.query(PublicationModel).filter(PublicationModel.id == id).first()
if publication is None:
    raise HTTPException(status_code=404, detail="Not Found")
else:
    categories = []
    authors = []
    for category in publication.categories:
        categories.append(category.name)
    for author in publication.authors:
        authors.append({"name": author.name, "surname": author.surname})
```

Endpoint Instances

Pre instances som implementoval POST a GET.

POST /instances/

Pri POSTe checkujem to čo vždy, no musím checkovať aj publikáciu a či status je v mojom ENUMe.

```python
if not instance.id:
    instance.id = str(uuid4())

# Check ci existuje publikacia
if not db.query(PublicationModel).filter(PublicationModel.id == instance.publication_id).first():
    raise HTTPException(status_code=404, detail="Not Found")

# check if correct status is given
if instance.status not in ["available", "reserved"]:
    raise HTTPException(status_code=400, detail="Not Found")
```

Raw query:

```sql
SELECT *
FROM publications
WHERE id = {instance.publication_id}
LIMIT 1;
```

GET /instances/{id}

GET funguje rovnako ako predošlé. Napríklad authors.

```python
instance = db.query(InstanceModel).filter(InstanceModel.id == id).first()
if instance is None:
    raise HTTPException(status_code=404, detail="Not Found")
```

Raw query:

```sql
SELECT *
FROM instance
WHERE id = {id}
LIMIT 1;
```

Endpoint Rentals

Pri Rentals som implementoval iba POST a GET, keďže som s týmto endpointom mal problémy a je podľa mňa najobtiažnejší.

POST /rentals/

Tento post bol najobťažnejší funguje takto: Zistujem či je voľná inštancia knihy dalej, zistím či existuje rezervácia a na koniec si do reservation si vložím orderby, ktorý má default ASC takže dostaneme najstaršiu rezerváciu a dáme ho v skratke tomu userovi ak sa rovná s id čo dostaneme.

```python
#check ci existuje instance
if not db.query(InstanceModel).filter(InstanceModel.publication_id == rental.publication_id).filter(InstanceModel.status == "available").first():
    if db.query(RenservationModel).filter(RenservationModel.publication_id == rental.publication_id).first():
        reservation = db.query(RenservationModel).filter(RenservationModel.publication_id == rental.publication_id).order_by(RenservationModel.created_at).first()
        if reservation.user_id == rental.user_id:
            instance_id = db.query(InstanceModel).filter(InstanceModel.publication_id == rental.publication_id).filter(InstanceModel.status == "available").first().id
```

Raw queries:

```sql
SELECT id
FROM instance_model
WHERE publication_id = {rental.publication_id}
AND status = 'available'
LIMIT 1;
```

```sql
SELECT *
FROM renservation
WHERE publication_id = {rental.publication_id}
ORDER BY created_at
LIMIT 1;
```

```sql
SELECT *
FROM instance
WHERE publication_id = {rental.publication_id}
AND status = 'available'
LIMIT 1;
```

GET /rentals/{id}

Pri poste používam tie isté rovnaké query ako pri authors atď.

```python
if not db.query(RentalModel).filter(RentalModel.id == rental_id).first():
    raise HTTPException(status_code=404, detail="Not Found")

rental = db.query(RentalModel).filter(RentalModel.id == rental_id).first()
```

Endpoint Reservations

Pri tomto endpointe som kvôli nedostatku času implementoval iba POST,GET a DELETE.

POST /reservations/

Pro tomto POSTe checkujem či existuje voľná inštancia ak áno tak nemusím rezervovať dostanem 400.

```
# check ci je volna instance
publication = db.query(PublicationModel).filter(PublicationModel.id == reservation.publication_id).first()
instance = db.query(InstanceModel).filter(InstanceModel.publication_id == publication.id, InstanceModel.status == "available").first()
```

Raw queries:

```
SELECT *
FROM publications
WHERE id = {reservation.publication_id};


SELECT *
FROM instances
WHERE publication_id = {publication.id} AND status = 'available'
LIMIT 1;
```

GET /reservations/{id}

To isté ako predošlé GET.

```
if not db.query(ReservationModel).filter(ReservationModel.id == reservation_id).first():
    raise HTTPException(status_code=404, detail="Not Found")
else:
    reservation = db.query(ReservationModel).filter(ReservationModel.id == reservation_id).first()
    return {
```

DELETE /reservations/{id}

Rovnaký ako predošlé DELETE.

```
if not db.query(ReservationModel).filter(ReservationModel.id == reservation_id).first():
    raise HTTPException(status_code=404, detail="Not Found")
else:
    reservation = db.query(ReservationModel).filter(ReservationModel.id == reservation_id).first()
```