Технології серверного програмного забезпечення

Лабораторна робота 3

Валідація, обробка помилок, ORM

Мета роботи: Покращити проект шляхом валідації вхідних даних, обробки помилок та використання ORM з базою даних

Завдання:

- Навчитись валідації та обробки помилок, типових для розробки бекенду на python
- Визначити моделі ORM для застосунку
- Використати базу даних
- Протестувати роботу застосунку

Практичне завдання:

Крім того, що потрібно додати валідацію, обробку помилок та ORM моделей, також потрібно зробити новий функціонал по варіанту:

1. Валюти
2. Користувацькі категорії витрат
3. Облік доходів

Варіант визначаємо за остачею від ділення свого номеру групи на 3 де остача 1 та 2 відповідає валютам та користувацьким категоріям витрат відповідно, а остача 0 відповідає обліку доходів. Визначення варіанту повинно бути в README.md в репозиторії.

Для валют - потрібно зробити окрему сутність, також для кожного користувача повинна бути валюта по замовчуванню(її можна встановити) а також при створенні витрати, можна вказувати валюту, проте не обов’язково(якщо не вказали то буде використана валюта по замовчуванню).

Для користувацьких категорій витрат - повинні бути загальні категорії витрат, які видно всім користувачам, та користувацькі, які можуть вказати тільки користувачі, які їх визначили.

Облік доходів - потрібно зробити сутність “рахунок” куди можна додавати гроші по мірі їх надходження(для кожного користувача

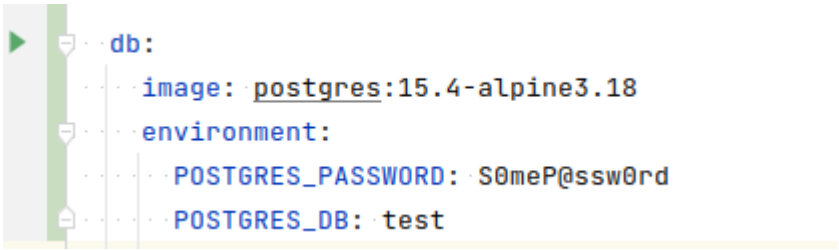
свій) і звідти списуються кошти атоматично при створенні нової витрати. Логіка щодо заходу в мінус лишається на розсуд студентів(можете або дозволити це, або заборонити).

Вимоги до виконання:

- Всі вимоги першої лабораторної актуальні і для другої.
- Реалізація додаткового завдання.

Методичні рекомендації:

1. Створити тег в репозиторії за допомогою команди git tag v2.0.0 -a -m "Lab 2", де v2.0.0 - це версія вашого застосунку
2. Далі пушаємо теги - git push --tags
3. Робити реліз тегу в github не обов'язково, так як це не бібліотека, проте можете додати.
4. Тепер потрібно додати базу даних для подальшої реалізації логіки застосунку. Для локальної розробки додайте конфігурацію для контейнера бази даних в файл docker-compose.yaml:



Враховуйте що значення паролю та бази даних вказані для прикладу, і їх потрібно визначати в змінних середовища

5. Після цього перевірте що контейнер запускається командою docker-compose up db.
6. Тепер встановіть бібліотеки: Flask-Migrate - для роботи з міграціями, flask-sqlalchemy - ORM, та psycopg2-binary - для роботи з самою базою даних.
   pip install psycopg2-binary flask-sqlalchemy Flask-Migrate
7. Додаємо в залежності бібліотеку flask-smorest за допомогою команди pip install flask-smorest
8. Оновлюємо файл requirements.txt за допомогою команди pip freeze > requirements.txt
9. Додаємо в наші ендпоінти обробку помилок

10. Додаємо marshmallow сутності. Приклади сутностей:

```python
from marshmallow import Schema, fields


5 usages  new *
class PlainUserSchema(Schema):
    id = fields.Str(dump_only=True)
    name = fields.Str(required=True)


8 usages  new *
class CategorySchema(Schema):
    id = fields.Str(dump_only=True)
    name = fields.Str(required=True)

```

11. Робимо валідацію в ендпоінтах по даних сутностях.
12. Додаємо ORM моделі наших сутностей. Приклади ORM моделі з flask-sqlalchemy:

```python
5 usages  new *
class UserModel(db.Model):
    __tablename__ = "user"

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(128), unique=True, nullable=False)

    record = db.relationship("RecordModel", back_populates="user", lazy="dynamic")

```

```
7 usages  new *
class RecordModel(db.Model):
    __tablename__ = "record"

    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(
        db.Integer,
        db.ForeignKey("user.id"),
        unique=False,
        nullable=False,
    )
    category_id = db.Column(
        db.Integer,
        db.ForeignKey("category.id"),
        unique=False,
        nullable=False,
    )
    created_at = db.Column(db.TIMESTAMP, server_default=func.now())
    sum = db.Column(db.Float(precision=2), unique=False, nullable=False)

    user = db.relationship("UserModel", back_populates="record")
    category = db.relationship("CategoryModel", back_populates="record")

```

13. Також не забуваємо про конфігурацію, її на даному етапі бажано винести в окремий файл таким чином:

```
import os

PROPAGATE_EXCEPTIONS = True
FLASK_DEBUG = True
SQLALCHEMY_DATABASE_URI = f'postgresql://{os.environ["POSTGRES_USER"]}:{os
SQLALCHEMY_TRACK_MODIFICATIONS = False
API_TITLE = "Finance REST API"
API_VERSION = "v1"
OPENAPI_VERSION = "3.0.3"
OPENAPI_URL_PREFIX = "/"
OPENAPI_SWAGGER_UI_PATH = "/swagger-ui"
OPENAPI_SWAGGER_UI_URL = "https://cdn.jsdelivr.net/npm/swagger-ui-dist/"
```

і зчитати в application factory за допомогою функції from_pyfile:

```
>app.config.from_pyfile('config.py', silent=True)
```

14. Після додавання моделей потрібно зробити міграції за допомогою бібліотеки flask-migrate - документація по міграціям знаходиться тут - https://flask-migrate.readthedocs.io/en/latest/#example
15. Перевіремо що все працює з базою даних
16. Додаємо нові ендпоінти в postman

Критерії оцінювання:

Всього можна отримати 20 балів за лабораторну
Розподіл балів за лабораторну 3:
5 балів - реалізація валідації та обробки помилок
5 балів - робота з ORM та базою
5 балів - коректно зроблене додаткове завдання(якщо не буде визначення варіанту в README.md - бали за цей пункт не будуть нараховані)
3 бали - є postman колекція з відповідними environment та новими ендпоінтами
2 бали - коректна робота з git: повідомлення комітів та адекватна кількість комітів
Також на розсуд перевіряючого може бути доставлено 1-2 бали за якісь цікаві рішення.