

Sprawozdanie z projektu

PAMIW 2021

JAN DOBROWOLSKI, 299242

Wstęp

W ramach realizacji projektu stworzyłem mini portal społecznościowy. System składa się z dwóch aplikacji – backendowej i frontendowej. Obie aplikacje zostały przeze mnie napisane w języku Python, z wykorzystaniem frameworka FastAPI.

Aplikacja frontendowa korzysta z silnika do dynamicznego wypełniania szablonów HTML – Jinja2. Szablony napisałem sam, z wykorzystaniem HTML, CSS i Javascripta.

Aplikacja backendowa łączy się z relacyjną bazą danych – przy pierwszym wykonaniu migracji tworzą się automatycznie pliki SQLite. Dodatkowo wykorzystuje ona klienta pocztowego Sendgrid.

Wymagania ogólne

1. Projekt powinien być przechowywany w repozytorium Github

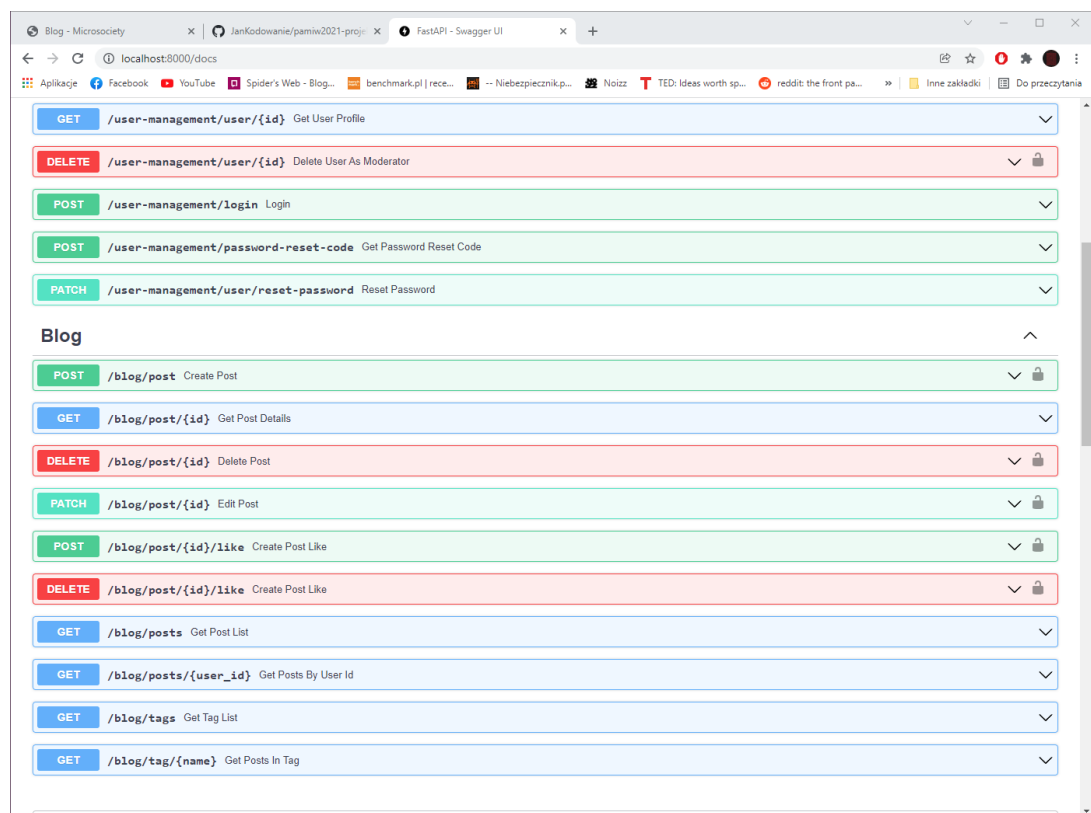
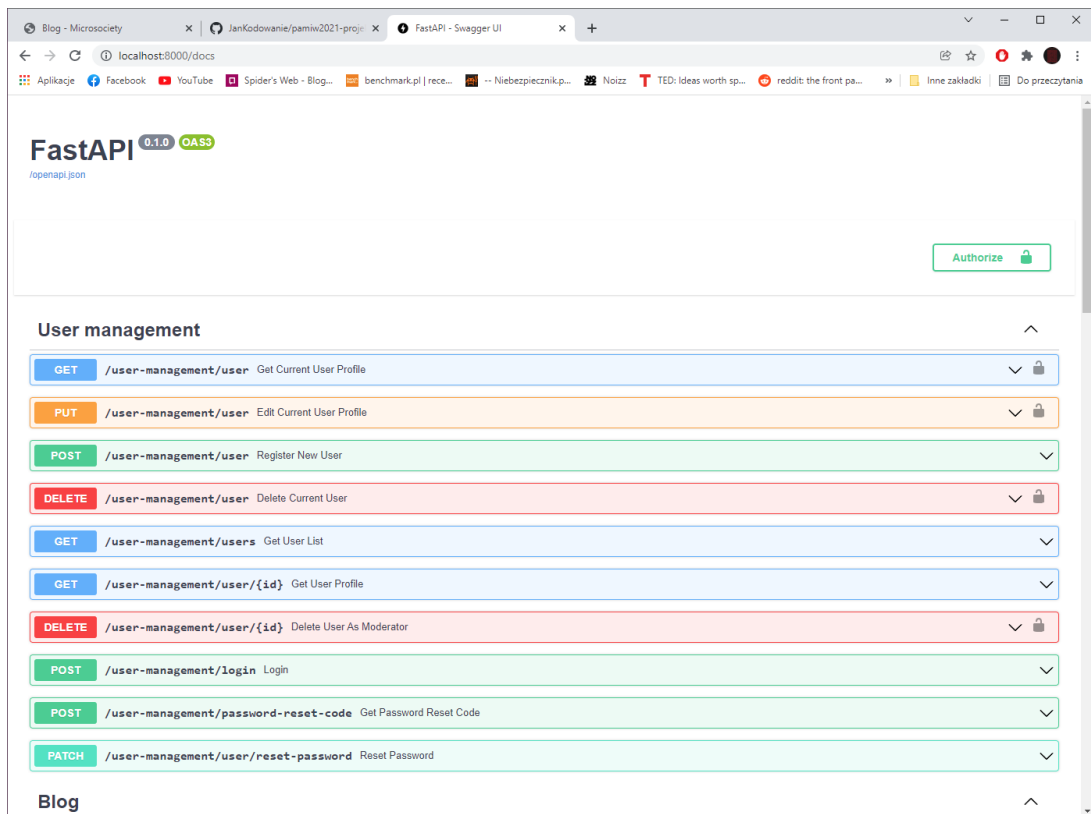
Link do repozytorium: <https://github.com/JanKodowanie/pamiw2021-projekt>

2. Zastosowanie architektury cebulowej

*

3. Powiązanie z interfejsem użytkownika odbywa się za pośrednictwem REST/SOAP

Framework FastAPI automatycznie dokumentuje API za pomocą Swaggera i Redoc. Jak widać na poniższych zrzutach ekranu, komunikacja z API odbywa się za pomocą plików JSON:



Blog - Microsociety x JanKodowanie/pamir2021-proje x FastAPI - Swagger UI x +

localhost:8000/docs/#/user-management/user Edit Current User Profile

PUT /user-management/user Register New User

Parameters Try it out

No parameters

Request body ^{required} application/json

Example Value | Schema

```
{
  "username": "string",
  "email": "user@example.com",
  "password": "string",
  "gender": "male"
}
```

Responses

Code	Description	Links
201	Successful Response	No links
	Media type: application/json Controls Accept header	
	Example Value Schema	
	<pre>{ "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6", "username": "string", "role": "standard", "bio": "string", "date_joined": "2022-01-12T18:01:54.776Z", "gender": "male" }</pre>	
401	Unauthorized	No links

Blog - Microsociety x JanKodowanie/pamir2021-proje x FastAPI - Swagger UI x +

localhost:8000/docs/#/blog/get_post_list_blog_posts_get

DELETE /blog/post/{id}/like Create Post Like

GET /blog/posts Get Post List

Parameters Try it out

No parameters

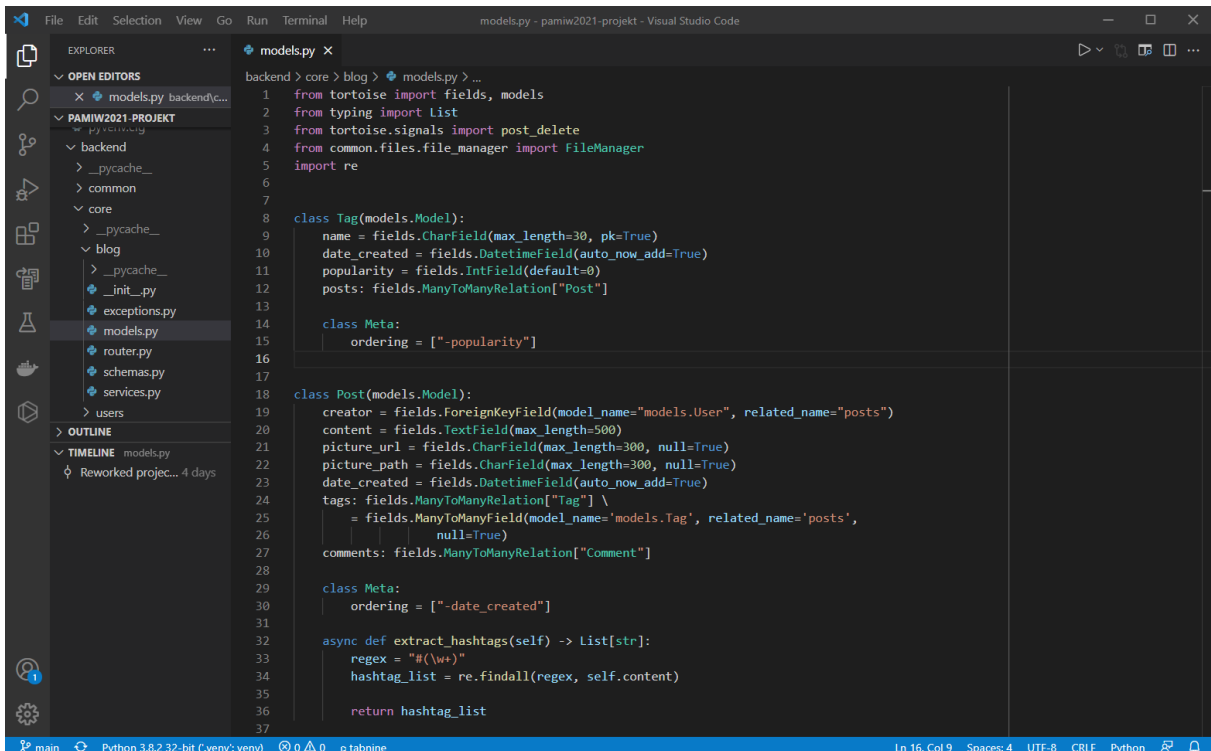
Responses

Code	Description	Links
200	Successful Response	No links
	Media type: application/json Controls Accept header	
	Example Value Schema	
	<pre>{ "id": 0, "creator": { "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6", "username": "string", "role": "standard" }, "content": "string", "picture_url": "string", "date_created": "2022-01-12T18:02:08.388Z", "tags": [{ "name": "string" }], "likes": [{ "creator_id": "3fa85f64-5717-4562-b3fc-2c963f66afa6" }] }</pre>	
401	Unauthorized	No links

Nie wszystkie funkcjonalności z API zostały zaimplementowane na frontendzie.

4. Stosowanie modeli domenowych (Domain-Driven Design). (ew. db-first i mapowanie na obiekty)

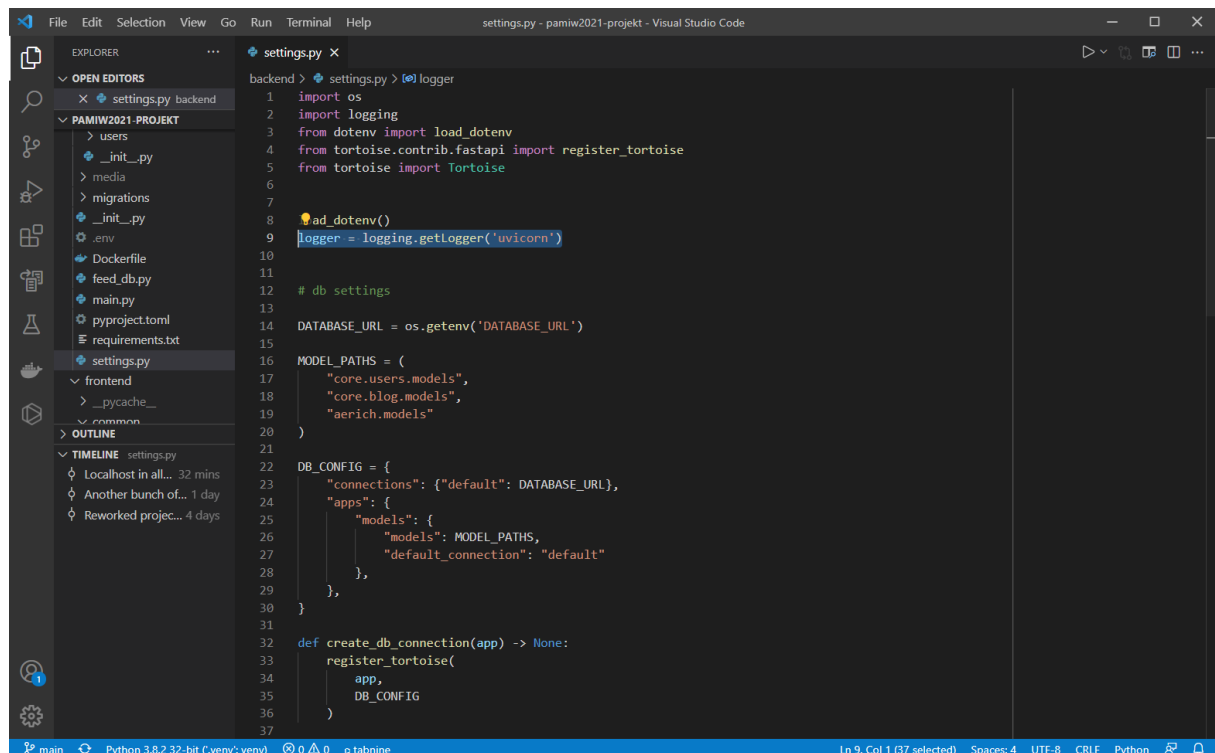
Aplikacja backendowa do połączeń z bazą danych wykorzystuje asynchroniczny ORM – Tortoise. Nie jest to podejście „Database-first”, ponieważ tabele w bazie tworzone są na podstawie kodu i migracji. Jedyną różnicą między Tortoise a Entity Framework jest taka, że obiekty Tortoise definiuje się z wykorzystaniem atrybutów, które opakowują standardowe typy Pythona, np. zamiast str stosuje się CharField – podczas odczytu te złożone typy są konwertowane na standardowe. Przykład:



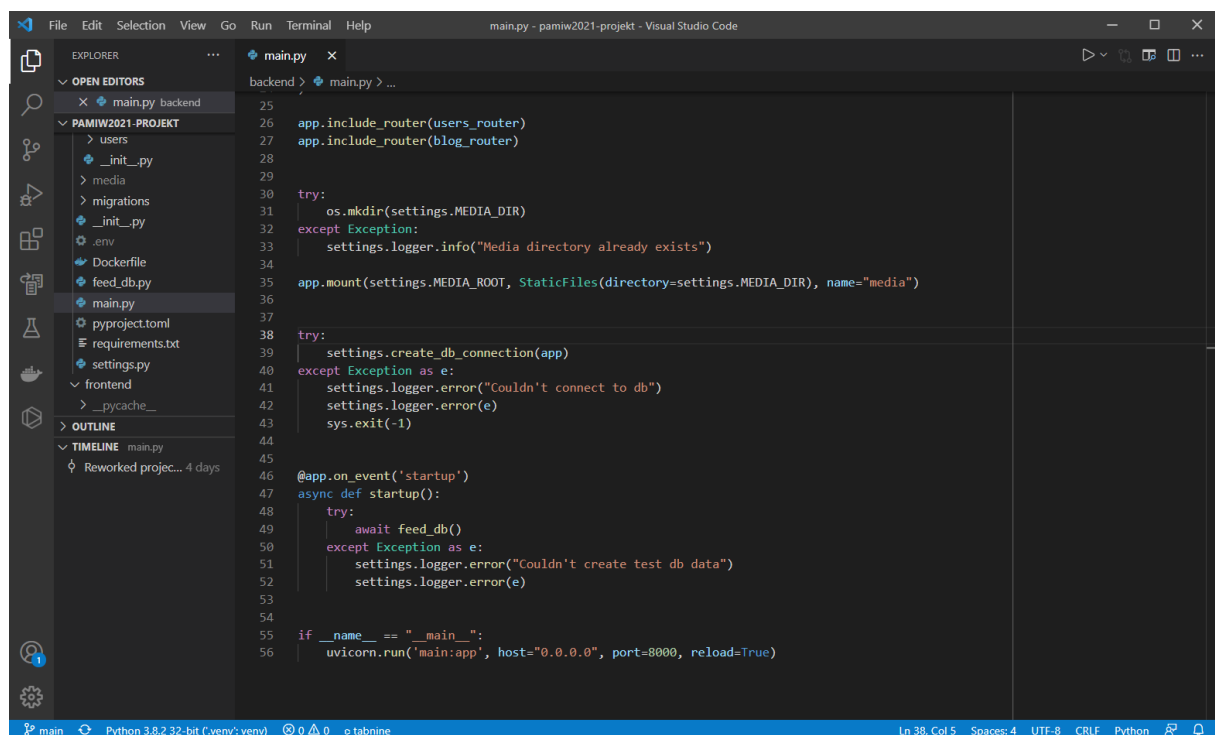
```
1 from tortoise import fields, models
2 from typing import List
3 from tortoise.signals import post_delete
4 from common.files.file_manager import FileManager
5 import re
6
7
8 class Tag(models.Model):
9     name = fields.CharField(max_length=30, pk=True)
10    date_created = fields.DateTimeField(auto_now_add=True)
11    popularity = fields.IntField(default=0)
12    posts = fields.ManyToManyField["Post"]
13
14    class Meta:
15        ordering = ["-popularity"]
16
17
18 class Post(models.Model):
19     creator = fields.ForeignKeyField(model_name="models.User", related_name="posts")
20     content = fields.TextField(max_length=500)
21     picture_url = fields.CharField(max_length=300, null=True)
22     picture_path = fields.CharField(max_length=300, null=True)
23     date_created = fields.DateTimeField(auto_now_add=True)
24     tags = fields.ManyToManyField["Tag"] \
25         = fields.ManyToManyField(model_name='models.Tag', related_name='posts',
26                                null=True)
27     comments = fields.ManyToManyField["Comment"]
28
29     class Meta:
30         ordering = ["-date_created"]
31
32     async def extract_hashtags(self) -> List[str]:
33         regex = "#(\w+)"
34         hashtag_list = re.findall(regex, self.content)
35
36         return hashtag_list
37
```

5. Logowanie błędów aplikacji przy pomocy dodatkowego frameworka

Aplikacje hostowane są za pomocą serwera ASGI Uvicorn, który wykorzystuje wbudowany w Pythona framework logging. Aby zrealizować ten punkt, w aplikacji backendowej skorzystałem z loggera Uvicorn:



```
backgend > settings.py > [0] logger
1 import os
2 import logging
3 from dotenv import load_dotenv
4 from tortoise.contrib.fastapi import register_tortoise
5 from tortoise import Tortoise
6
7
8 load_dotenv()
9 logger = logging.getLogger('uvicorn')
10
11
12 # db settings
13
14 DATABASE_URL = os.getenv('DATABASE_URL')
15
16 MODEL_PATHS = (
17     "core.users.models",
18     "core.blog.models",
19     "aerich.models"
20 )
21
22 DB_CONFIG = {
23     "connections": {"default": DATABASE_URL},
24     "apps": {
25         "models": {
26             "models": MODEL_PATHS,
27             "default_connection": "default"
28         },
29     },
30 }
31
32 def create_db_connection(app) -> None:
33     register_tortoise(
34         app,
35         DB_CONFIG
36     )
37
```



```
backgend > main.py > ...
25
26 app.include_router(users_router)
27 app.include_router(blog_router)
28
29
30 try:
31     os.mkdir(settings.MEDIA_DIR)
32 except Exception:
33     settings.logger.info("Media directory already exists")
34
35 app.mount(settings.MEDIA_ROOT, StaticFiles(directory=settings.MEDIA_DIR), name="media")
36
37
38 try:
39     settings.create_db_connection(app)
40 except Exception as e:
41     settings.logger.error("Couldn't connect to db")
42     settings.logger.error(e)
43     sys.exit(-1)
44
45
46 @app.on_event('startup')
47 async def startup():
48     try:
49         await feed_db()
50     except Exception as e:
51         settings.logger.error("Couldn't create test db data")
52         settings.logger.error(e)
53
54
55 if __name__ == "__main__":
56     uvicorn.run('main:app', host="0.0.0.0", port=8000, reload=True)
```

Za pomocą loggera Uvicorn wyświetlam w konsoli informacje o błędach m.in. przy połączeniu z DB, przy tworzeniu danych testowych, przy tworzeniu folderu do przechowywania zdjęć z postów.

```
MINGW64:d:/Dev/pamiw2021-1 x + - □ x

=> CACHED [4/4] RUN pip install --no-cache-dir --upgrade -r requirements.txt 0.0s
=> exporting to image 0.0s
=> exporting layers 0.0s
=> writing image sha256:f6b68f05fa018b2027ab10ca6cf29a1d410e4b9134b005e8d6c6bad547d46f58 0.0s
=> naming to docker.io/library/pamiw2021-projekt_frontend 0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
Creating pamiw2021-projekt_backend_1 ... done
Creating pamiw2021-projekt_frontend_1 ... done
Attaching to pamiw2021-projekt_backend_1, pamiw2021-projekt_frontend_1
backend_1 | INFO: Media directory already exists
backend_1 | INFO: Started server process [1]
backend_1 | INFO: Waiting for application startup.
frontend_1 | INFO: Started server process [1]
frontend_1 | INFO: Waiting for application startup.
frontend_1 | INFO: Application startup complete.
frontend_1 | INFO: Uvicorn running on http://0.0.0.0:3000 (Press CTRL+C to quit)
backend_1 | INFO: Application startup complete.
backend_1 | INFO: Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
backend_1 | INFO: 172.30.0.3:41276 - "GET /blog/posts HTTP/1.1" 200 OK
frontend_1 | INFO: 172.30.0.1:54986 - "GET / HTTP/1.1" 200 OK
frontend_1 | INFO: 172.30.0.1:54986 - "GET /static/styles.css HTTP/1.1" 200 OK
frontend_1 | INFO: 172.30.0.1:54990 - "GET /static/logout.js HTTP/1.1" 200 OK
frontend_1 | INFO: 172.30.0.1:54996 - "GET /static/blog.js HTTP/1.1" 200 OK
frontend_1 | INFO: 172.30.0.1:54996 - "GET /favicon.ico HTTP/1.1" 404 Not Found
backend_1 | INFO: 172.30.0.1:47716 - "GET /docs HTTP/1.1" 200 OK
backend_1 | INFO: 172.30.0.1:47716 - "GET /openapi.json HTTP/1.1" 200 OK
backend_1 | INFO: 172.30.0.1:47724 - "GET /blog/posts HTTP/1.1" 200 OK
backend_1 | INFO: 172.30.0.1:47728 - "GET /blog/posts HTTP/1.1" 200 OK
```

6. Dodanie mechanizmu uwierzytelnienia i autoryzacji

Stworzony przeze mnie system wykorzystuje uwierzytelnienie za pomocą JWT. Aplikacja backendowa generuje podpisane za pomocą sekretnego klucza tokeny, które przesyłane są wraz z podstawowymi danymi użytkownika w odpowiedzi przy operacji logowania.

Klasy odpowiedzialne za uwierzytelnienie na backendzie:

```
File Edit Selection View Go Run Terminal Help middleware.py - pamiw2021-projekt - Visual Studio Code

EXPLORER
main.py
middleware.py x
main.py backend
middleware.py backe...
PAMIW2021-PROJEKT
backend
_pycache_
common
core
_pycache_
blog
users
_pycache_
auth
_pycache_
_init_.py
exceptions.py
middleware.py
permissions.py
utils
_init_.py
enums.py
exceptions.py
models.py
router.py

OUTLINE
middleware.py
Login, logout an... 2 days
Reworked projec... 4 days

class TokenManager:

    def create_token(self, account: User):
        to_encode = self._create_jwt_data(account)
        data = UserDataSchema(**to_encode)
        encoded_jwt = jwt.encode(to_encode, settings.SECRET_KEY, algorithm=settings.ALGORITHM)
        return encoded_jwt, data

    def decode_token(self, token: str) -> UserDataSchema:
        try:
            payload = jwt.decode(token, settings.SECRET_KEY, algorithms=[settings.ALGORITHM])
            token_data = UserDataSchema(**payload)
        except Exception:
            raise MalformedAccessToken()

        if token_data.exp < datetime.now(timezone.utc):
            raise AccessTokenExpired()

        return token_data

    def _create_jwt_data(self, account: User) -> dict:
        iat = datetime.now(timezone.utc)
        exp = iat + timedelta(minutes=settings.ACCESS_TOKEN_EXPIRE_MINUTES)

        to_encode = {
            'sub': str(account.id),
            'username': account.username,
            'role': account.role.value,
            'iat': iat,
            'exp': exp
        }

        return to_encode
```

```
47         'exp': exp
48     }
49
50     return to_encode
51
52
53 class AuthHandler:
54
55     async def authenticate_user(self, email, password) -> LoginResponse:
56         try:
57             user = await UserService().get_by_email(email)
58         except AccountNotFound:
59             raise InvalidCredentials()
60
61         if not Hash().verify(user.password, password):
62             raise InvalidCredentials()
63
64         access_token, data = TokenManager().create_token(user)
65         token_type = 'Bearer'
66
67         return LoginResponse(access_token=access_token, token_type=token_type, user=user)
68
69
70 @classmethod
71 async def get_user_from_token(cls, token: str = Depends(oauth2_scheme)) -> User:
72     try:
73         data = TokenManager().decode_token(token)
74         user = await UserService().get_by_id(data.sub)
75     except Exception as e:
76         raise HTTPException(401, detail=e.detail)
77
78     return user
```

Odpowiedź z backendu – endpoint /user-management/login

Code	Description	Links
201	Successful Response	No links

Media type

Controls Accept header.

Example Value | Schema

```
{
  "access_token": "string",
  "token_type": "string",
  "user": {
    "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
    "username": "string",
    "role": "standard"
  }
}
```

Aplikacja frontendowa przy operacji logowania wysyła żądanie z danymi logowania do backendu. Jeśli dane są poprawne, otrzymuje ona token i dane użytkownika, które zapisuje w plikach Cookie, odpowiednio „token” i „user_data” (dane użytkownika w formacie JSON).

```
router.py - pamiw2021-projekt - Visual Studio Code

frontend > core > users > router.py > login_user

25 @router.get("/login", response_class=HTMLResponse)
26 async def get_login_page(request: Request):
27     return templates.TemplateResponse("login.html", {"request": request})
28
29
30 @router.post(
31     "/login",
32     response_model=OkResponse,
33     status_code=status.HTTP_201_CREATED
34 )
35 async def login_user(
36     response: Response,
37     email: str = Form(...),
38     password: str = Form(...),
39 ):
40     login_data = LoginRequest(username=email, password=password)
41     async with httpx.AsyncClient() as client:
42         backend_response = await client.post(f'{settings.BACKEND_URL}/user-management/login', data=login_data.dict())
43         response_data = backend_response.json()
44         if backend_response.status_code == status.HTTP_401_UNAUTHORIZED:
45             raise HTTPException(status.HTTP_401_UNAUTHORIZED, detail=response_data['detail'])
46
47     response_data = LoginResponse(**response_data)
48     response.set_cookie(key="token", value=f"({response_data.token_type}) {response_data.access_token}", httponly=True)
49     response.set_cookie(key="user_data", value=response_data.user.json(), httponly=True)
50
51     return OkResponse(detail="Logowanie powiodło się.")
52
53
54 @router.get(
55     "/logout",
56     response_model=OkResponse,
57     status_code=status.HTTP_200_OK
58 )
59 async def logout_user(response: Response):
60     response.delete_cookie('token')
```

Formularz logowania:

Logowanie - Microsociety

localhost:3000/login

Blog Zarejestruj się Zaloguj się

Logowanie

Zaloguj się, by móc pisać posty na naszym blogu!

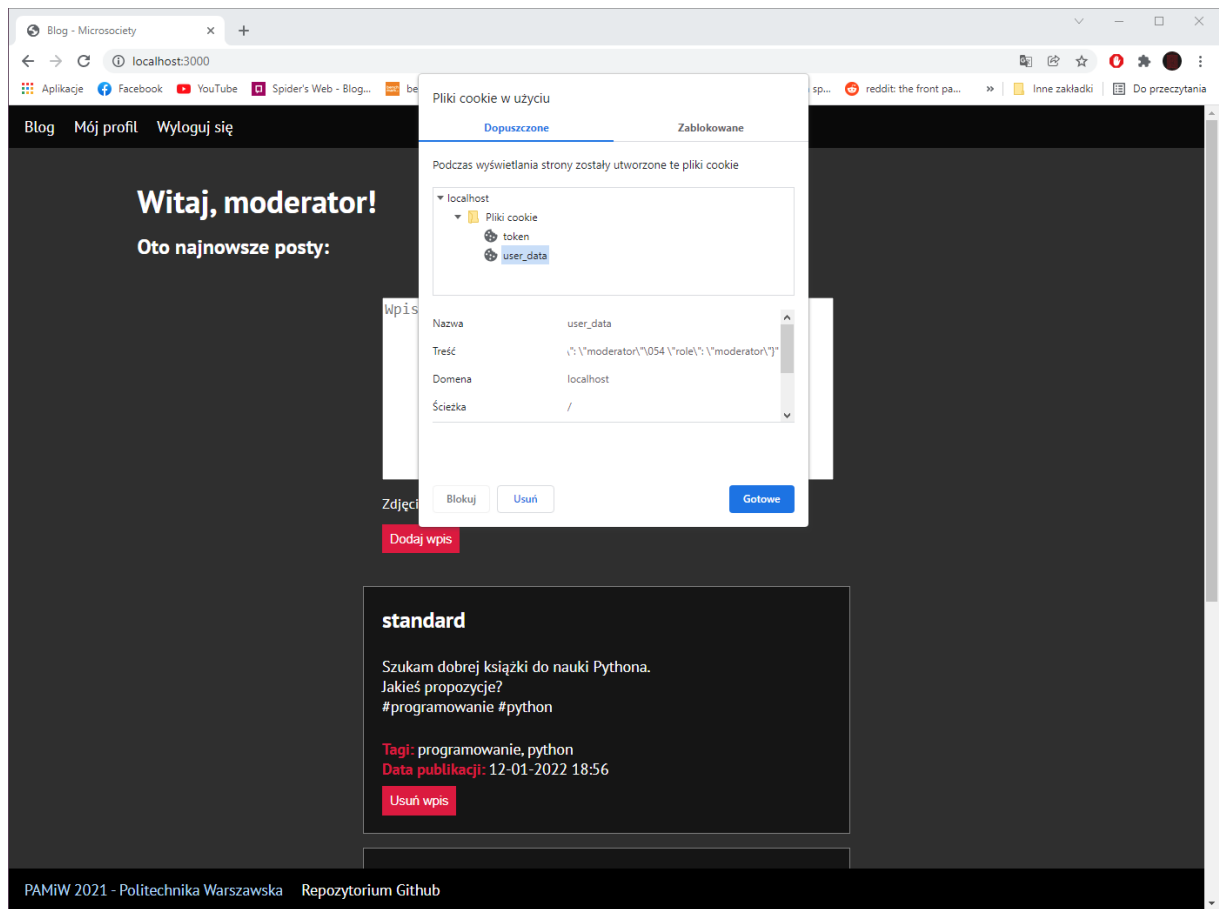
Email

Hasło

[Nie pamiętasz hasła?](#)

PAMIW 2021 - Politechnika Warszawska Repozytorium Github

Pliki Cookie po zalogowaniu:

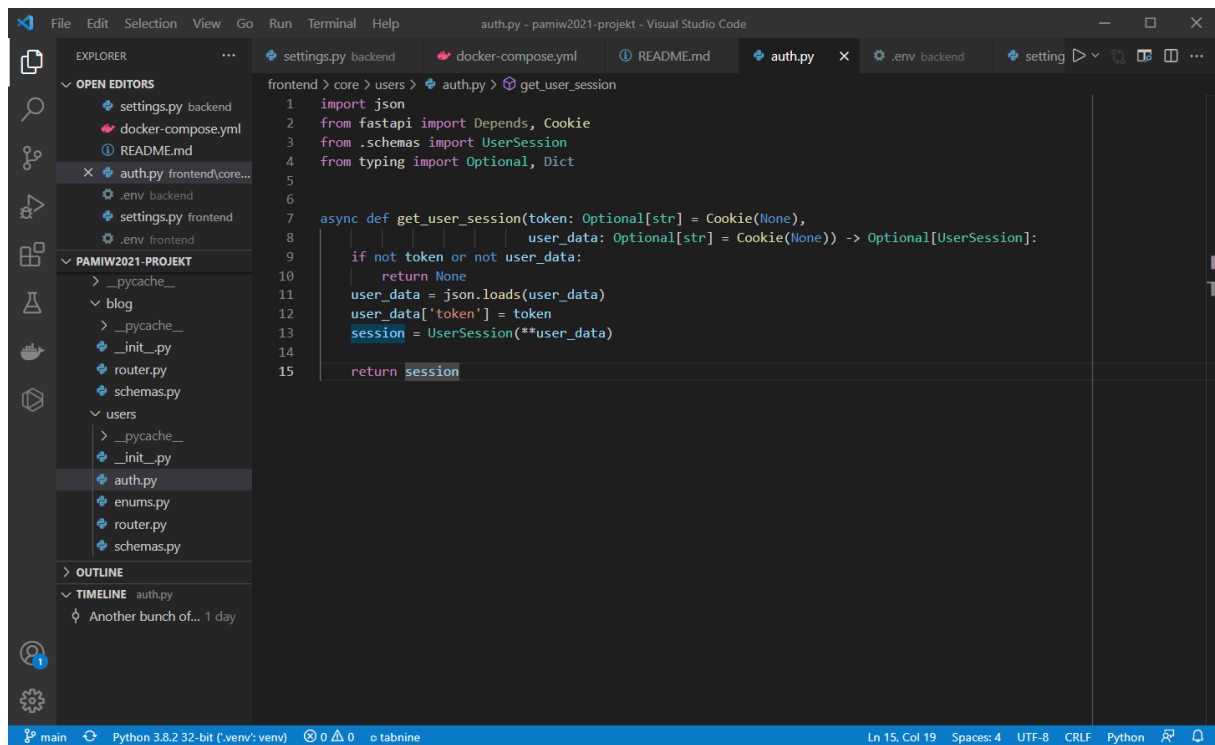


Frontend przy renderowaniu stron HTML sprawdza, czy użytkownik jest zalogowany, jaką ma rolę, czy dane elementy przynależą do niego, i na tej podstawie określa, jakie elementy strony są dostępne. Przykład – szablon dla pokazanego powyżej widoku bloga: jeśli użytkownik nie jest zalogowany, to nie jest tworzony formularz do tworzenia postów, a także m.in. przyciski do usuwania wpisów, które nie należą do niego:

```
12
13 router = APIRouter(
14     tags=['Blog'],
15     responses= {
16         status.HTTP_401_UNAUTHORIZED: NotAuthenticatedResponse().dict(),
17         status.HTTP_403_FORBIDDEN: ForbiddenResponse().dict(),
18         status.HTTP_404_NOT_FOUND: NotFoundResponse().dict()
19     }
20 )
21
22 templates = Jinja2Templates(directory="templates")
23
24 @router.get("/", response_class=HTMLResponse)
25 async def get_blog_page(
26     request: Request,
27     user: Optional[UserSession] = Depends(get_user_session)
28 ):
29     async with httpx.AsyncClient() as client:
30         response = await client.get(f'{settings.BACKEND_URL}/blog/posts')
31         data = response.json()
32         posts = PostListSchema(posts=data)
33
34     return templates.TemplateResponse("blog.html", {"request": request, "user": user, "posts": posts.dict()})
35
36
37
38
39 @router.get("/tag/{name}", response_class=HTMLResponse)
40 async def get_tag_view(
41     name: str,
42     request: Request,
43     user: Optional[UserSession] = Depends(get_user_session)
44 ):
45     async with httpx.AsyncClient() as client:
46         response = await client.get(f'{settings.BACKEND_URL}/blog/tag/{name}')
47         if response.status_code == status.HTTP 200 OK:
```

```
1 {% extends "layout.html" %}
2 {% block title %}Blog{% endblock title %}
3 {% block head %}
4     {{ super() }}
5     <script src="/static/blog.js"></script>
6 {% endblock head %}
7 {% block content %}
8     <div>
9         <h1 class="blog-header1">{% if user %}Witaj, {{ user.username|e }}!{% else %}Witaj!{% endif %}</h1>
10     </div>
11     <div>
12         <h2 class="blog-header3">Oto najnowsze posty:</h2>
13         <div class="posts-view">
14             {% if user %}
15                 <div class="new-post">
16                     <form class="new-post-form" id="new-post-form">
17                         <ul>
18                             <li>
19                                 <textarea placeholder="wpisz treść tutaj..." id="new-post-content" name="content" form="new-post-form" required></textarea>
20                             </li>
21                             <li>
22                                 <label for="picture">Zdjęcie: </label>
23                                 <input type="file" id="picture" name="picture" accept="image/png, image/jpeg, image/jpg">
24                             </li>
25                             <li>
26                                 <button type="submit" class="button1 create-post-button">Dodaj wpis</button>
27                             </li>
28                         </ul>
29                     </form>
30                 </div>
31             {% endif %}
32             {% for post in posts %}
33                 <div class="post">
34                     <div class="post-data">
35                         <div class="post-creator"><a class="link" href="/profile/{{ post.creator.id }}">{{ post.creator.username|e }}</a></div>
36                         <div class="post-text">{{ post.content|e }}</div>
```

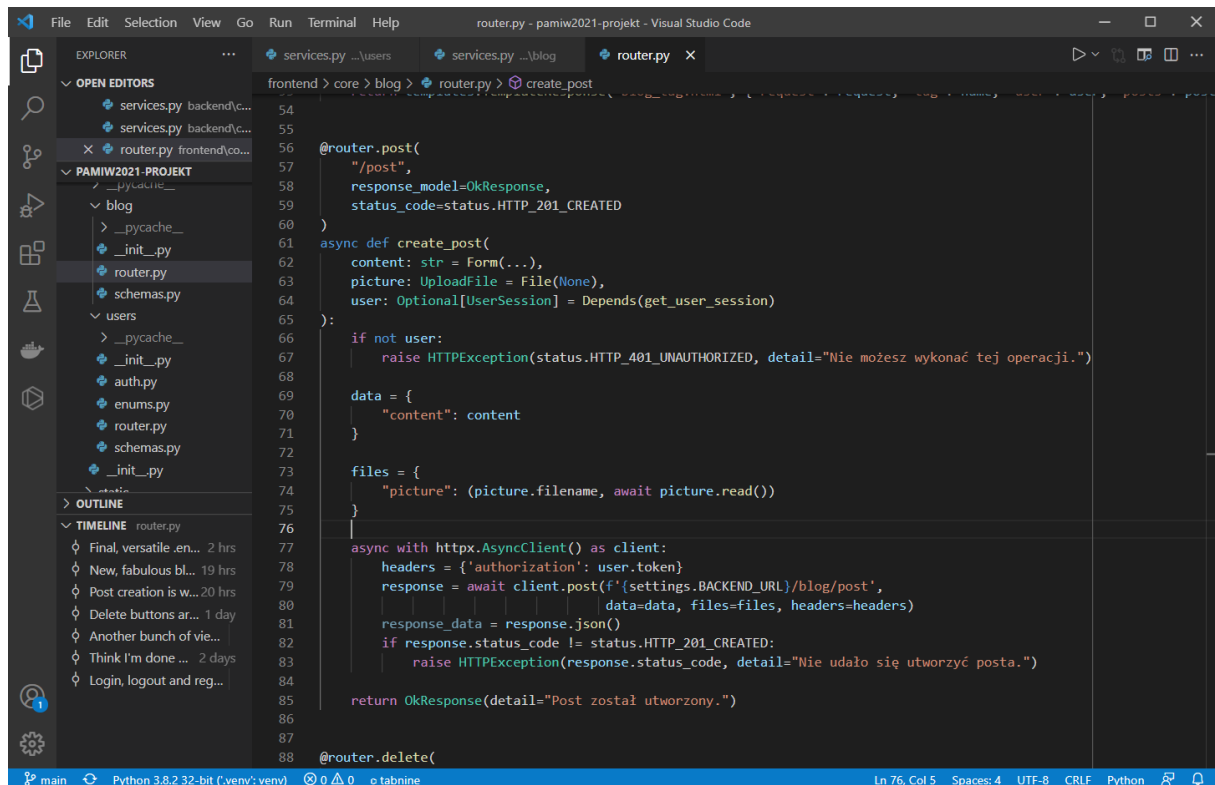
Funkcja odpowiedzialna za pobranie z Cookies tokena i danych użytkownika:



The screenshot shows the Visual Studio Code editor with the file explorer on the left and the code editor in the center. The file explorer shows the project structure for 'PAMIW2021-PROJEKT', with the 'users' directory expanded, showing 'auth.py' selected. The code editor displays the 'get_user_session' function in 'auth.py'. The function is an asynchronous def that takes an optional token and user_data as arguments. It checks if the token is None or if user_data is None, returning None in those cases. Otherwise, it loads the user_data from a JSON string, sets the token, creates a UserSession object, and returns it.

```
1 import json
2 from fastapi import Depends, Cookie
3 from .schemas import UserSession
4 from typing import Optional, Dict
5
6
7 async def get_user_session(token: Optional[str] = Cookie(None),
8                             user_data: Optional[str] = Cookie(None)) -> Optional[UserSession]:
9
10     if not token or not user_data:
11         return None
12     user_data = json.loads(user_data)
13     user_data['token'] = token
14     session = UserSession(**user_data)
15
16     return session
```

W przypadku operacji wymagających autentykacji, frontend wysła token w headerze „Authorization”:



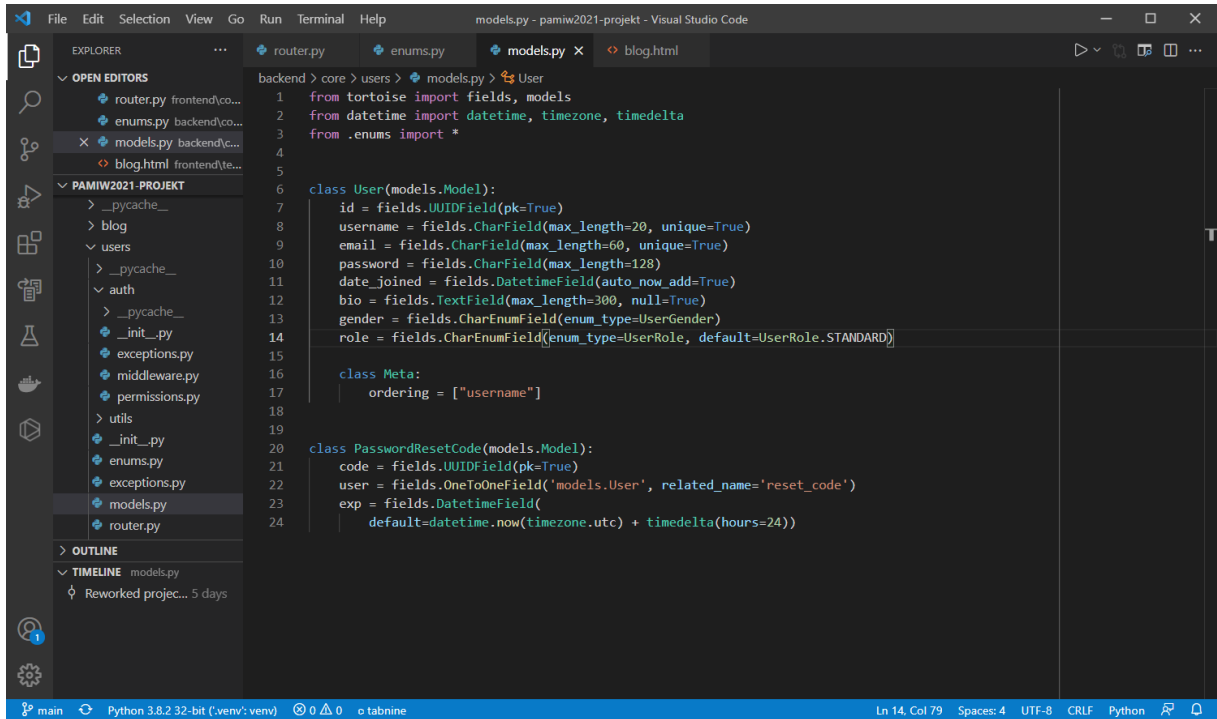
The screenshot shows the Visual Studio Code editor with the file explorer on the left and the code editor in the center. The file explorer shows the project structure for 'PAMIW2021-PROJEKT', with the 'blog' directory expanded, showing 'router.py' selected. The code editor displays the 'create_post' function in 'router.py'. The function is an asynchronous def that takes content, picture, and user as arguments. It checks if the user is None, raising a 401 Unauthorized exception if so. Otherwise, it creates a data dictionary with the content and a files dictionary with the picture. It then uses the httpx.AsyncClient to post the data to the backend, including the user's token in the Authorization header. If the status code is not 201 Created, it raises an exception. Finally, it returns an OkResponse with the detail 'Post został utworzony.'

```
54
55
56 @router.post(
57     "/post",
58     response_model=OkResponse,
59     status_code=status.HTTP_201_CREATED
60 )
61 async def create_post(
62     content: str = Form(...),
63     picture: UploadFile = File(None),
64     user: Optional[UserSession] = Depends(get_user_session)
65 ):
66     if not user:
67         raise HTTPException(status.HTTP_401_UNAUTHORIZED, detail="Nie możesz wykonać tej operacji.")
68
69     data = {
70         "content": content
71     }
72
73     files = {
74         "picture": (picture.filename, await picture.read())
75     }
76
77     async with httpx.AsyncClient() as client:
78         headers = {'authorization': user.token}
79         response = await client.post(f'{settings.BACKEND_URL}/blog/post',
80                                     data=data, files=files, headers=headers)
81         response_data = response.json()
82         if response.status_code != status.HTTP_201_CREATED:
83             raise HTTPException(response.status_code, detail="Nie udało się utworzyć posta.")
84
85     return OkResponse(detail="Post został utworzony.")
86
87
88 @router.delete(
```

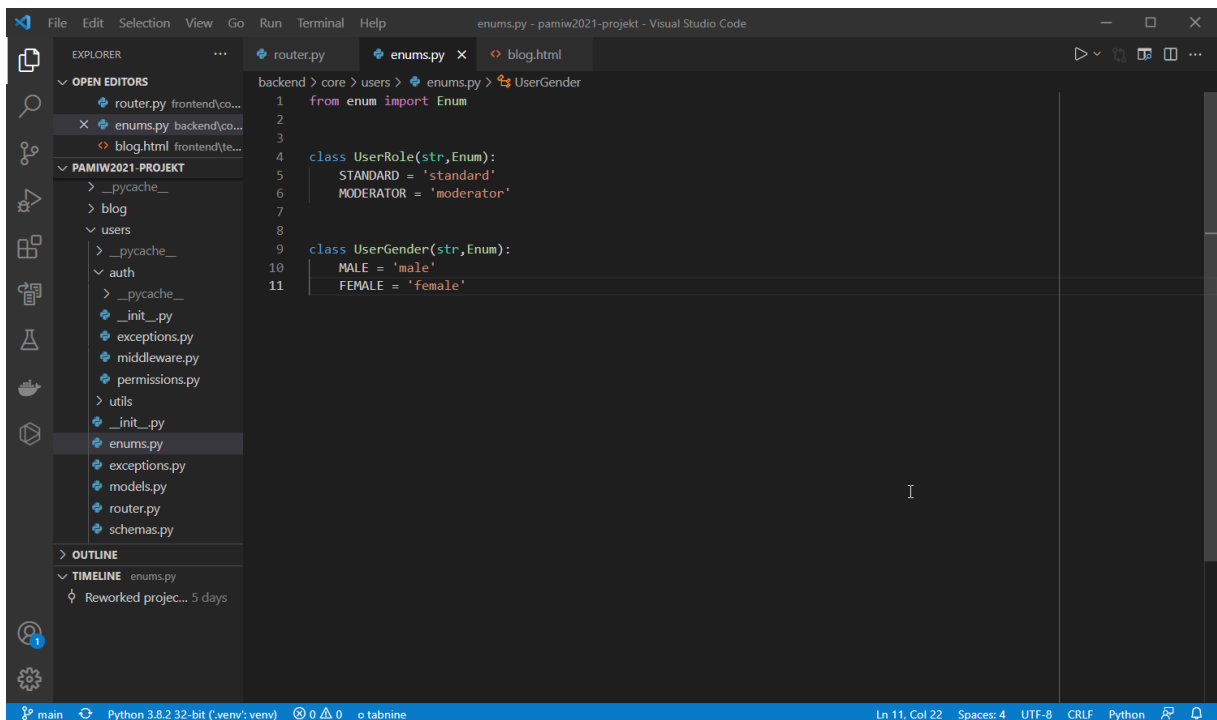
7. Obsługa systemu ról

W mojej aplikacji istnieją dwie role: użytkownik standardowy oraz moderator. Kod aplikacji backendowej:

Model użytkownika:

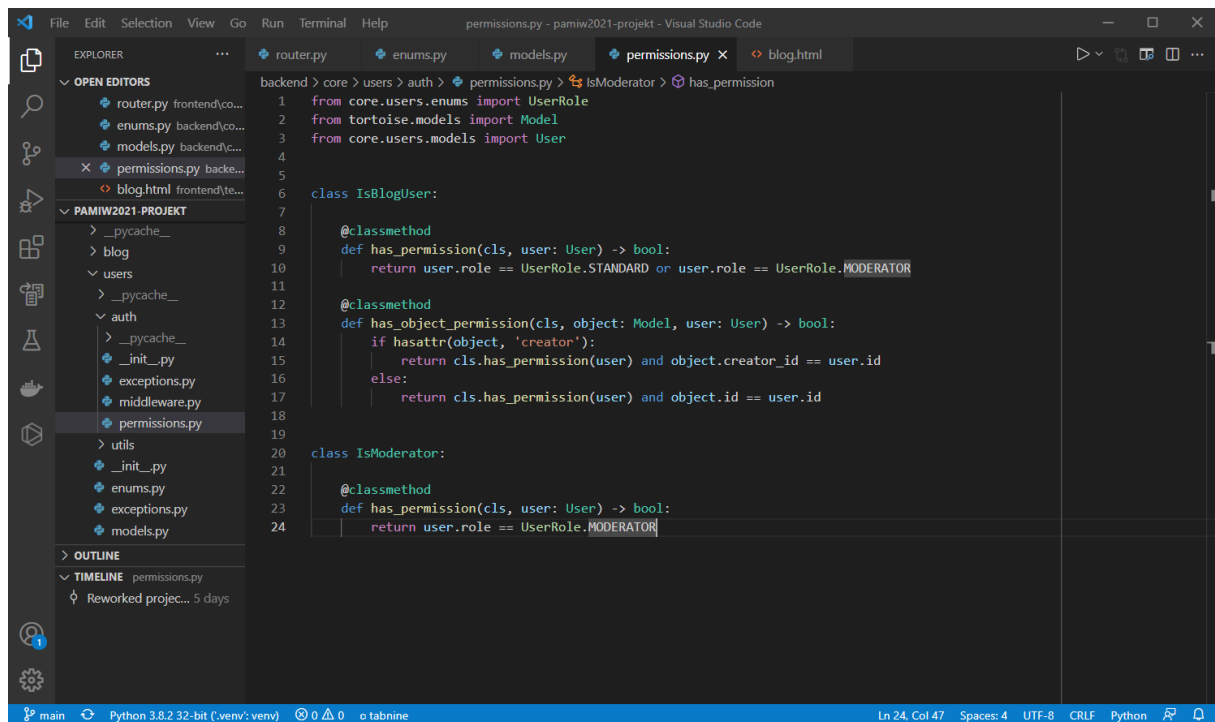


```
1 from tortoise import fields, models
2 from datetime import datetime, timezone, timedelta
3 from .enums import *
4
5
6 class User(models.Model):
7     id = fields.UUIDField(pk=True)
8     username = fields.CharField(max_length=20, unique=True)
9     email = fields.CharField(max_length=60, unique=True)
10    password = fields.CharField(max_length=128)
11    date_joined = fields.DateTimeField(auto_now_add=True)
12    bio = fields.TextField(max_length=300, null=True)
13    gender = fields.CharField(enum_type=UserGender)
14    role = fields.CharField(enum_type=UserRole, default=UserRole.STANDARD)
15
16    class Meta:
17        ordering = ["username"]
18
19
20 class PasswordResetCode(models.Model):
21     code = fields.UUIDField(pk=True)
22     user = fields.OneToOneField('models.User', related_name='reset_code')
23     exp = fields.DateTimeField(
24         default=datetime.now(timezone.utc) + timedelta(hours=24))
```



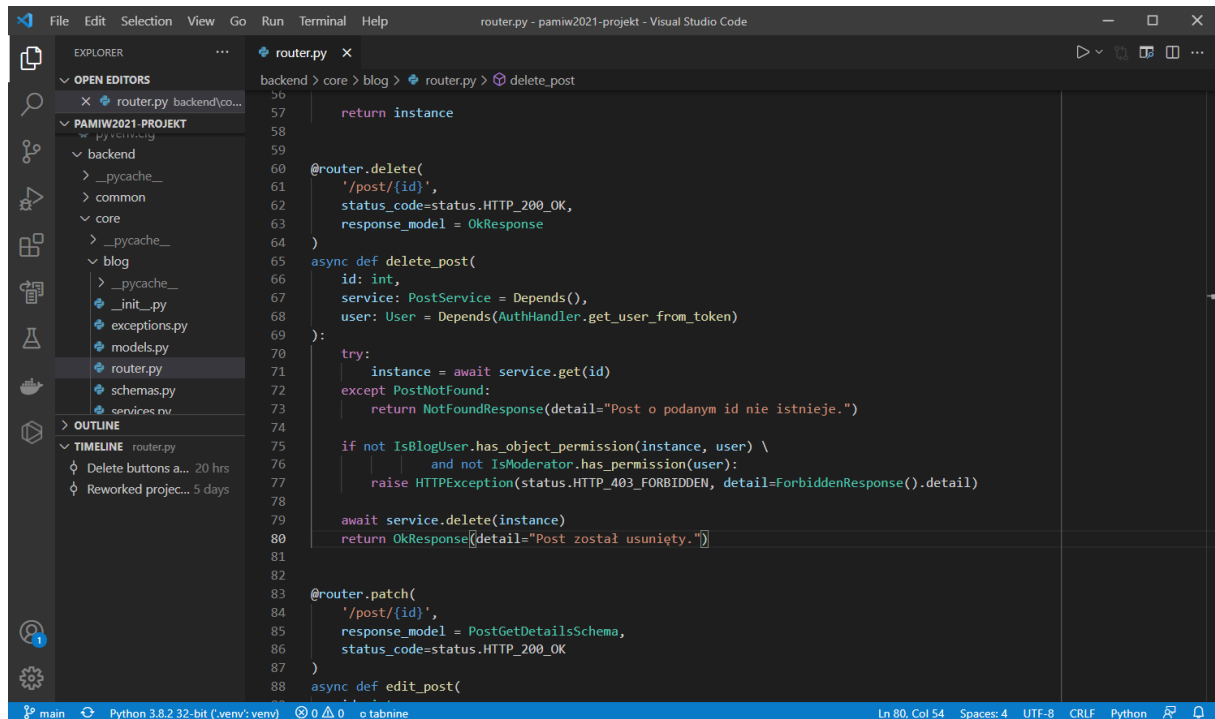
```
1 from enum import Enum
2
3
4 class UserRole(str, Enum):
5     STANDARD = 'standard'
6     MODERATOR = 'moderator'
7
8
9 class UserGender(str, Enum):
10     MALE = 'male'
11     FEMALE = 'female'
```

Klasy uprawnień:



```
1 from core.users.enums import UserRole
2 from tortoise.models import Model
3 from core.users.models import User
4
5
6 class IsBlogUser:
7
8     @classmethod
9     def has_permission(cls, user: User) -> bool:
10         return user.role == UserRole.STANDARD or user.role == UserRole.MODERATOR
11
12     @classmethod
13     def has_object_permission(cls, object: Model, user: User) -> bool:
14         if hasattr(object, 'creator'):
15             return cls.has_permission(user) and object.creator_id == user.id
16         else:
17             return cls.has_permission(user) and object.id == user.id
18
19
20 class IsModerator:
21
22     @classmethod
23     def has_permission(cls, user: User) -> bool:
24         return user.role == UserRole.MODERATOR
```

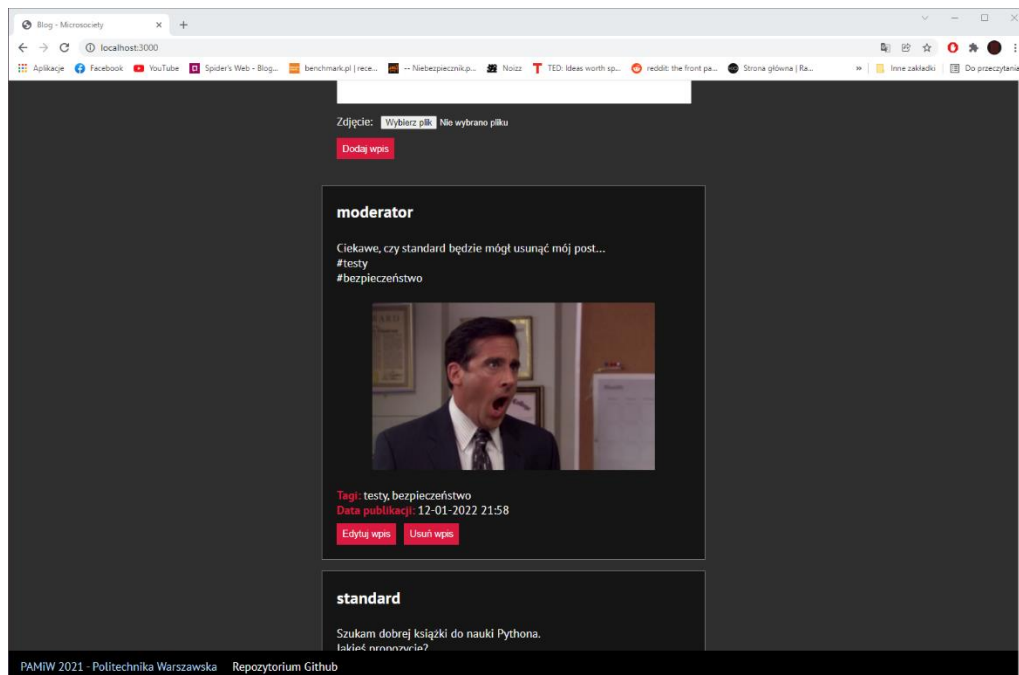
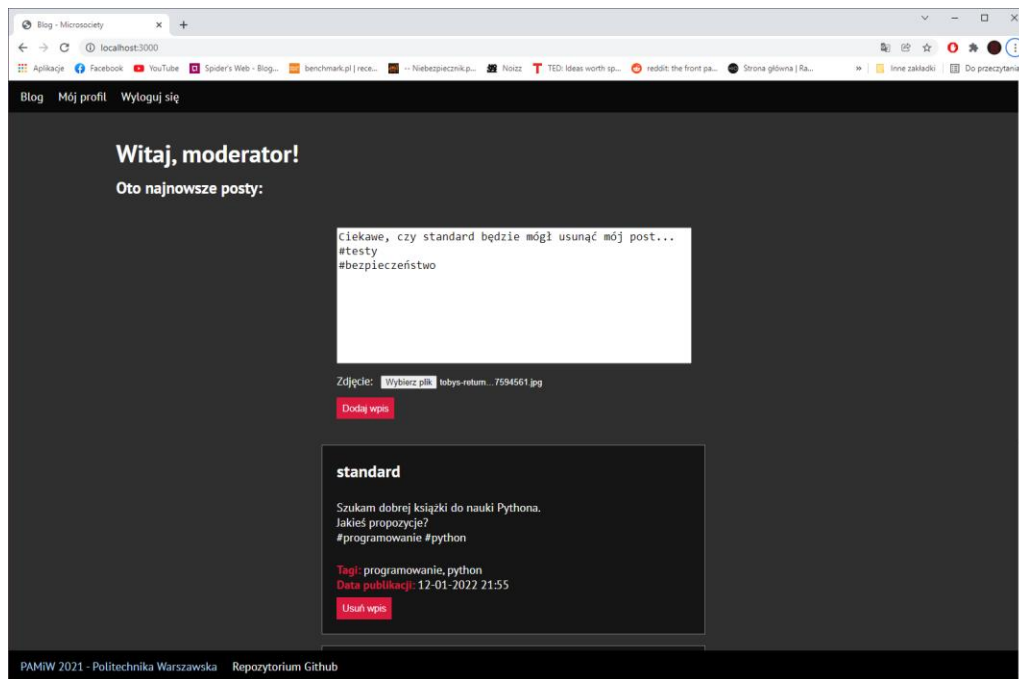
Przykład wykorzystania – kasowanie posta:



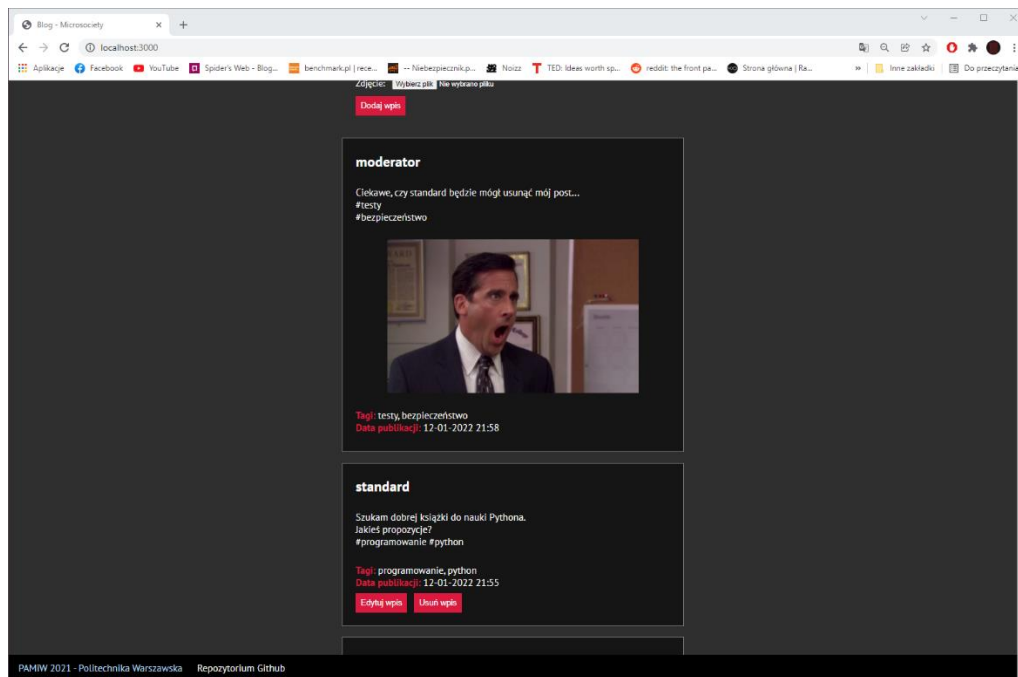
```
56 return instance
57
58
59 @router.delete(
60     '/post/{id}',
61     status_code=status.HTTP_200_OK,
62     response_model = OkResponse
63 )
64
65 async def delete_post(
66     id: int,
67     service: PostService = Depends(),
68     user: User = Depends(AuthHandler.get_user_from_token)
69 ):
70     try:
71         instance = await service.get(id)
72     except PostNotFound:
73         return NotFoundResponse(detail="Post o podanym id nie istnieje.")
74
75     if not IsBlogUser.has_object_permission(instance, user) \
76         and not IsModerator.has_permission(user):
77         raise HTTPException(status.HTTP_403_FORBIDDEN, detail=ForbiddenResponse().detail)
78
79     await service.delete(instance)
80     return OkResponse(detail="Post został usunięty.")
81
82
83 @router.patch(
84     '/post/{id}',
85     response_model = PostGetDetailsSchema,
86     status_code=status.HTTP_200_OK
87 )
88 async def edit_post(
```

Nie ma możliwości zarejestrowania moderatora z poziomu frontendu – jedno takie konto tworzone jest przy uruchomieniu aplikacji backendowej automatycznie.

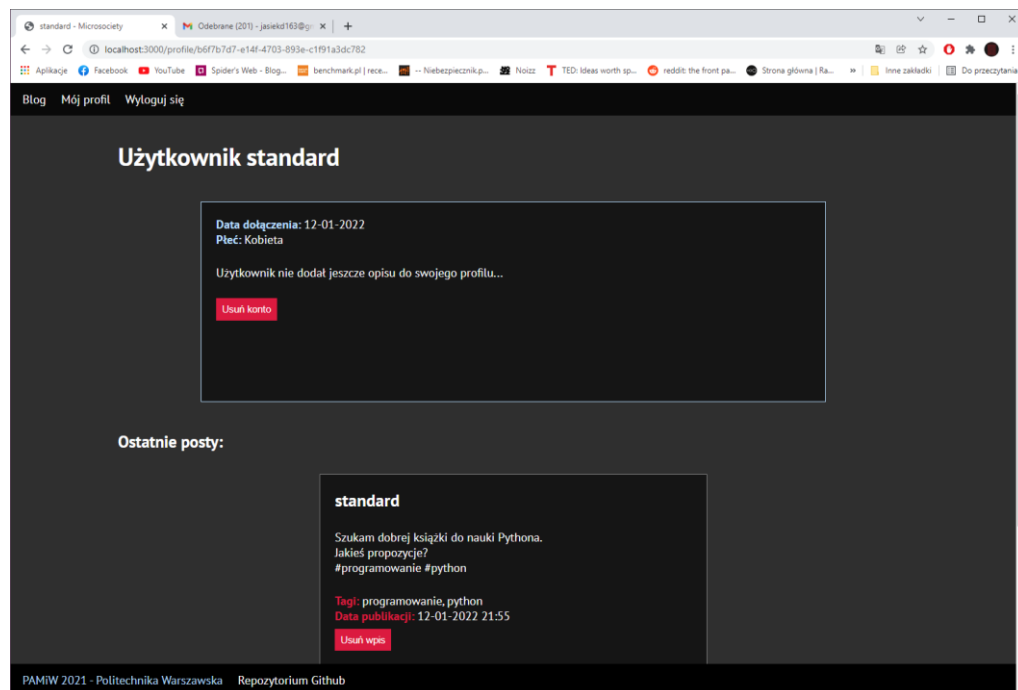
Moderator, w przeciwieństwie do standardowego użytkownika, może skasować dowolny post oraz konto dowolnego użytkownika. Widok bloga dla moderatora:



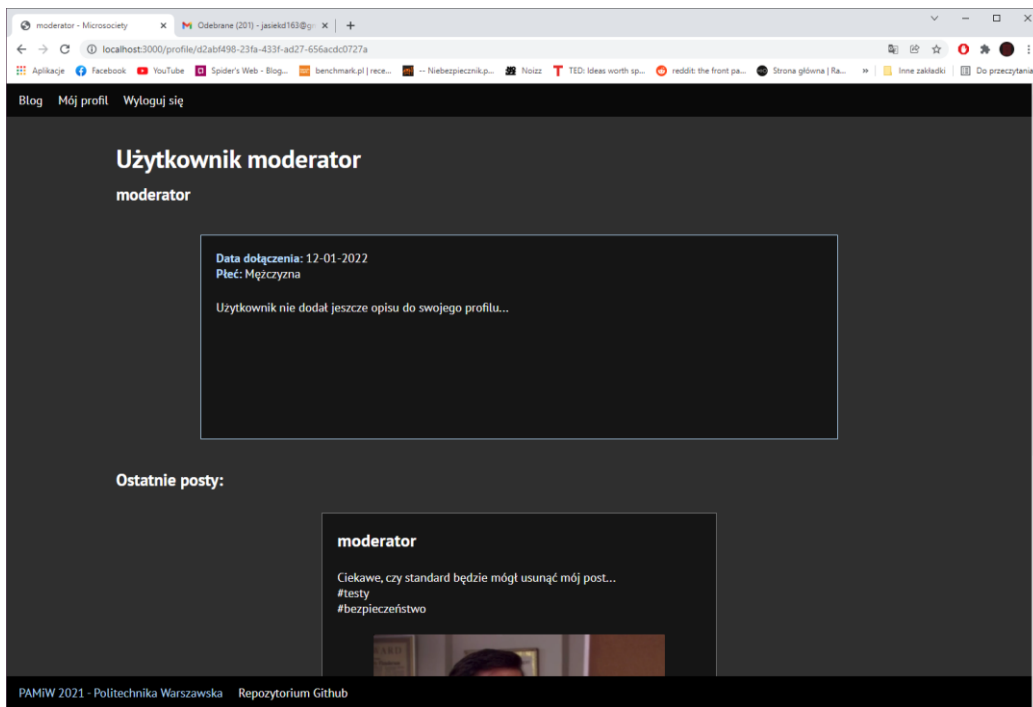
Widok bloga dla użytkownika standard:



Moderator może kasować profil użytkownika standard:



Odwrótne operacje nie są możliwe:



8. Identyfikacja użytkowników przy pomocy JWT

Warunek został spełniony, co zostało udowodnione w dwóch poprzednich punktach.

9. Stworzenie testów jednostkowych jednego repozytorium z mockami bazy

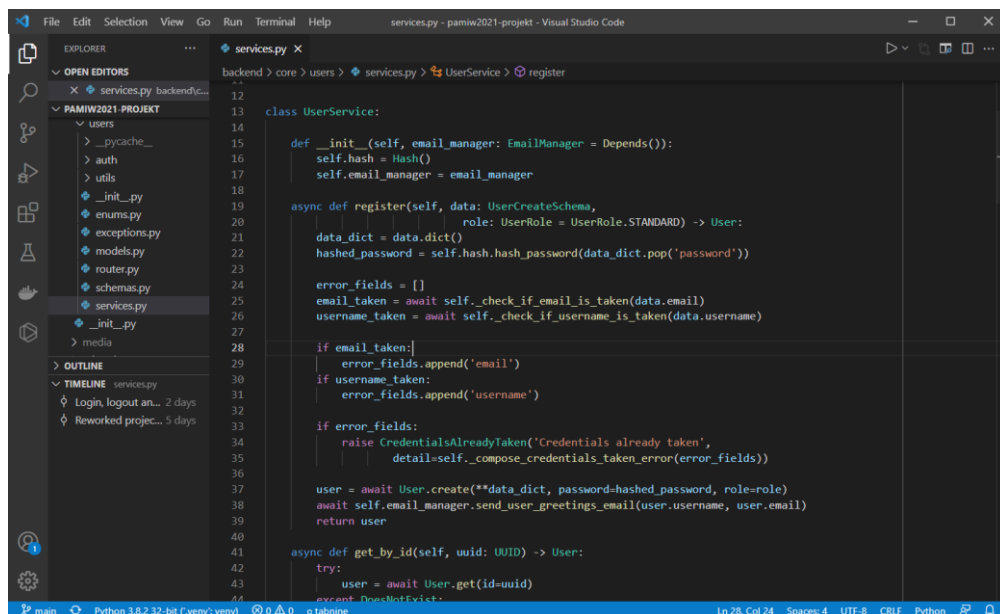
*

10. Stworzenie serwisów agregujących kilka operacji (np: dodanie użytkownika do bazy i wysłanie maila)

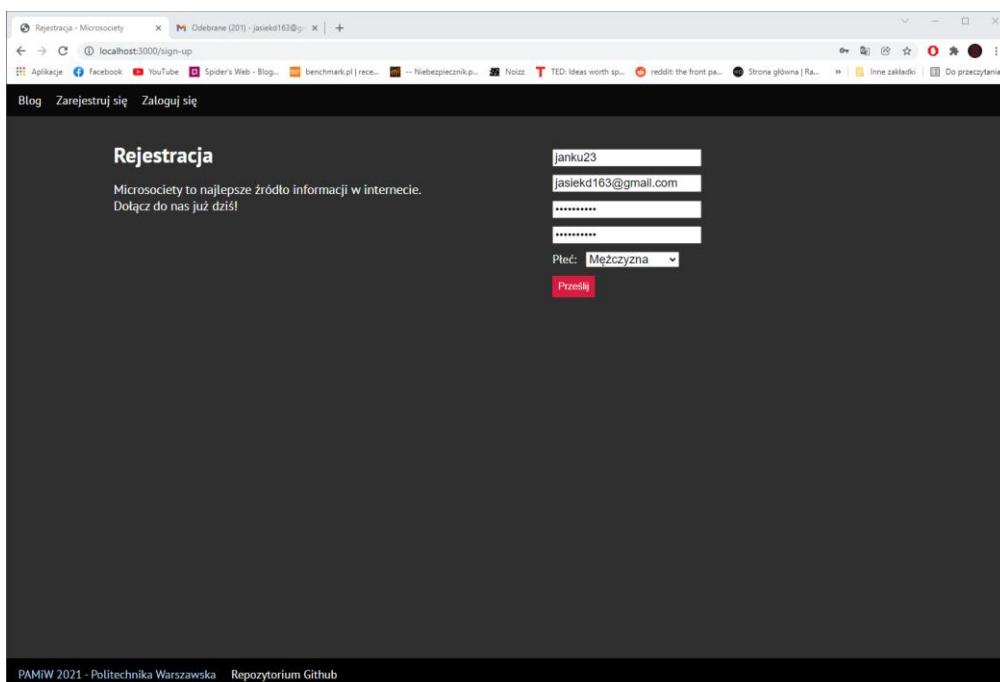
Choć nie zastosowałem w moim projekcie architektury cebulowej i jest ona powiązana z konkretnym ORM, to stworzyłem serwisy, który realizują zasadniczą logikę biznesową i są wstrzykiwane do widoków za pomocą mechanizmu Dependency Injection.

Przykłady serwisów realizujących kilka operacji:

UserService – rejestracja użytkownika i wysłanie maila powitalnego:



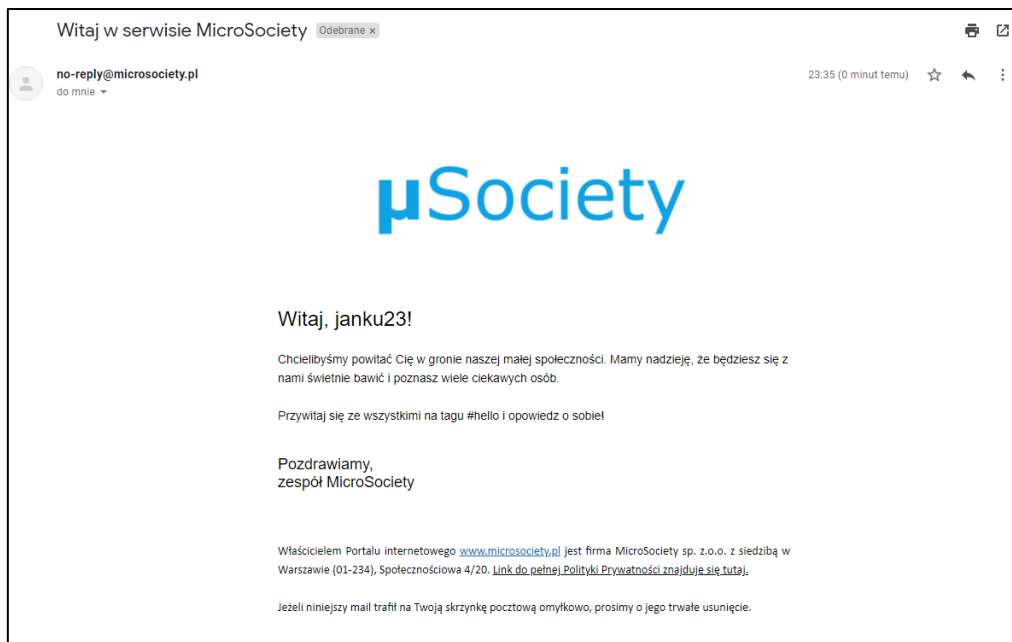
```
12
13
14 class UserService:
15
16     def __init__(self, email_manager: EmailManager = Depends()):
17         self.hash = Hash()
18         self.email_manager = email_manager
19
20     async def register(self, data: UserCreateSchema,
21                       role: UserRole = UserRole.STANDARD) -> User:
22         data_dict = data.dict()
23         hashed_password = self.hash.hash_password(data_dict.pop('password'))
24
25         error_fields = []
26         email_taken = await self._check_if_email_is_taken(data.email)
27         username_taken = await self._check_if_username_is_taken(data.username)
28
29         if email_taken:
30             error_fields.append('email')
31         if username_taken:
32             error_fields.append('username')
33
34         if error_fields:
35             raise CredentialsAlreadyTaken('Credentials already taken',
36                                           detail=self._compose_credentials_taken_error(error_fields))
37
38         user = await User.create(**data_dict, password=hashed_password, role=role)
39         await self.email_manager.send_user_greetings_email(user.username, user.email)
40         return user
41
42     async def get_by_id(self, uuid: UUID) -> User:
43         try:
44             user = await User.get(id=uuid)
45         except DoesNotExist:
```



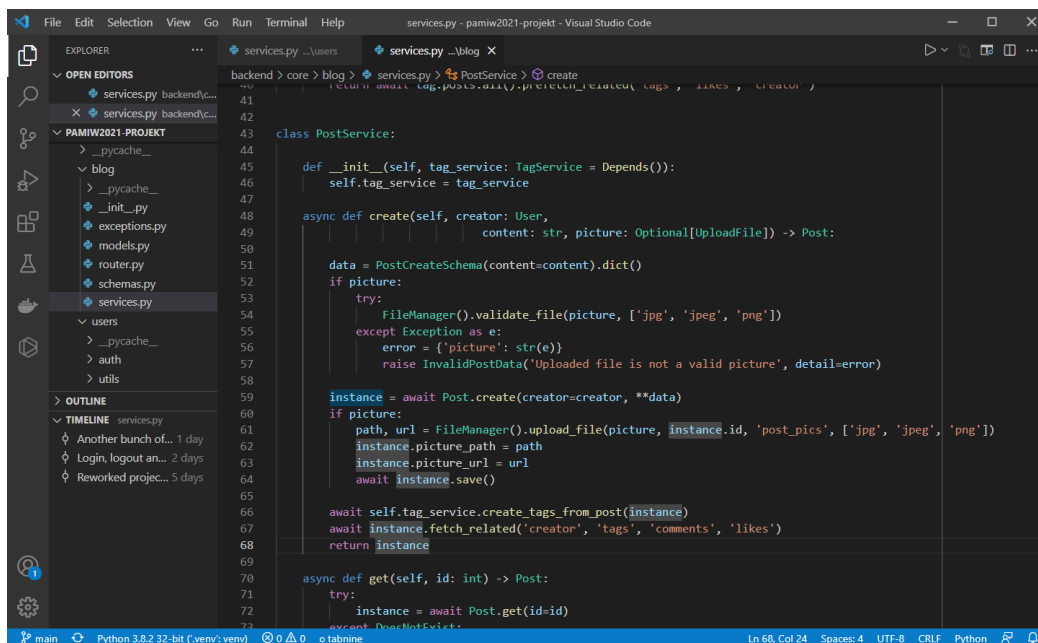
Rejestracja

Microsociety to najlepsze źródło informacji w internecie.
Dołącz do nas już dziś!

Płeć:



PostService – stworzenie posta, zapisanie zdjęcia (jeśli zostało przesłane) na dysku:

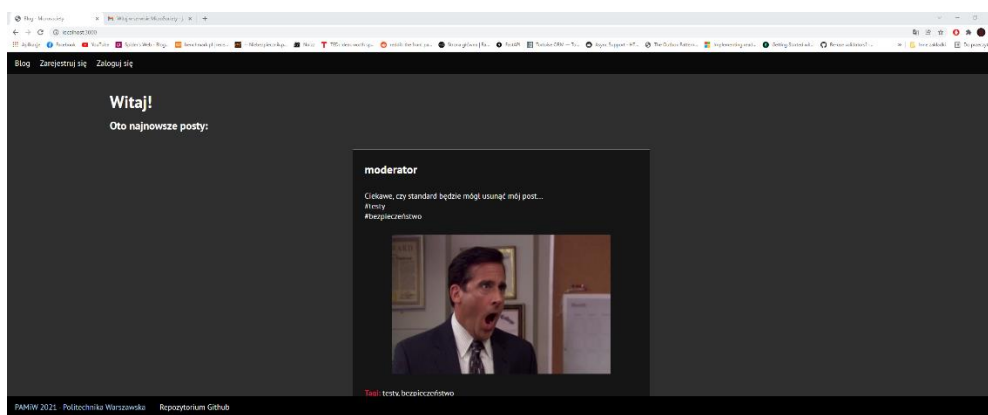


UI

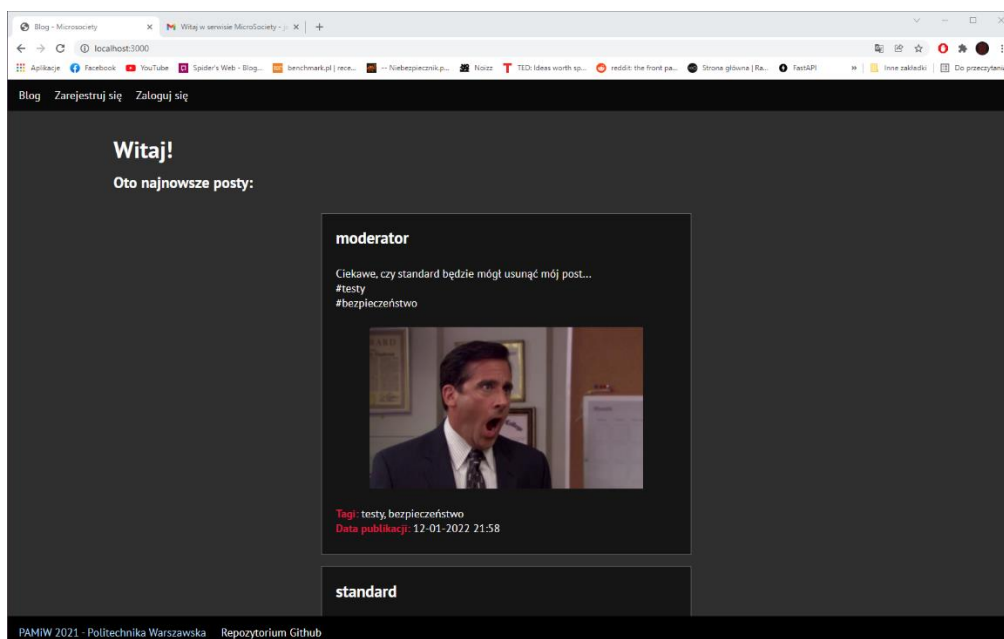
1. Zastosowanie biblioteki Bootstrap (obsługa urządzeń mobilnych)

Chociaż nie zastosowałem na frontendzie biblioteki Bootstrap, to jednak w swoim pliku ze stylami zastosowałem tylko relatywne wymiary elementów (zależne od szerokości lub wysokości ekranu urządzenia/elementu nadrzędnego) wykorzystałem także kontenery Flex. Dzięki temu moja strona wygląda ładnie i czytelnie niezależnie od proporcji ekranu, także na smartfonach:

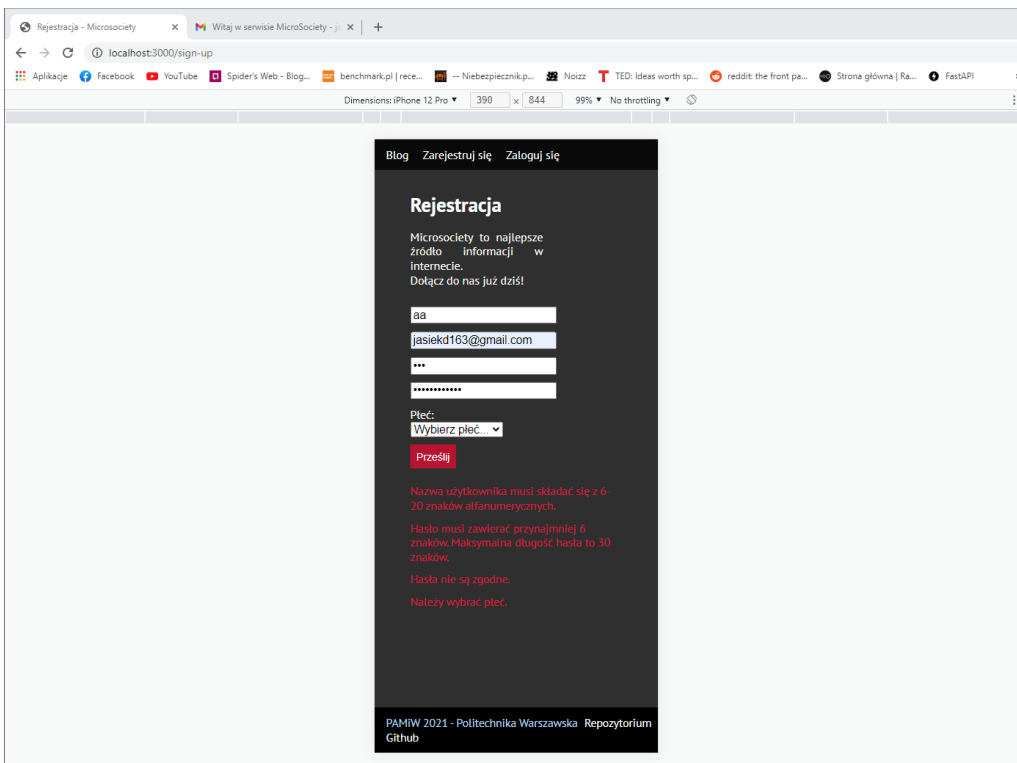
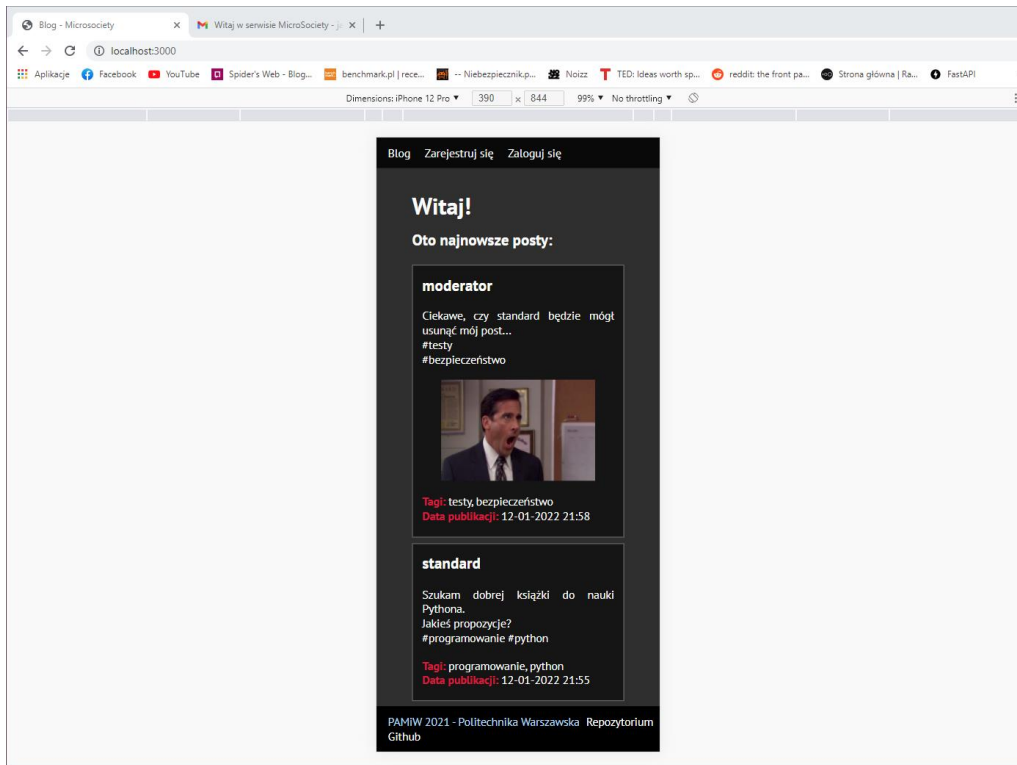
Ekran 21:9



Ekran 16:9

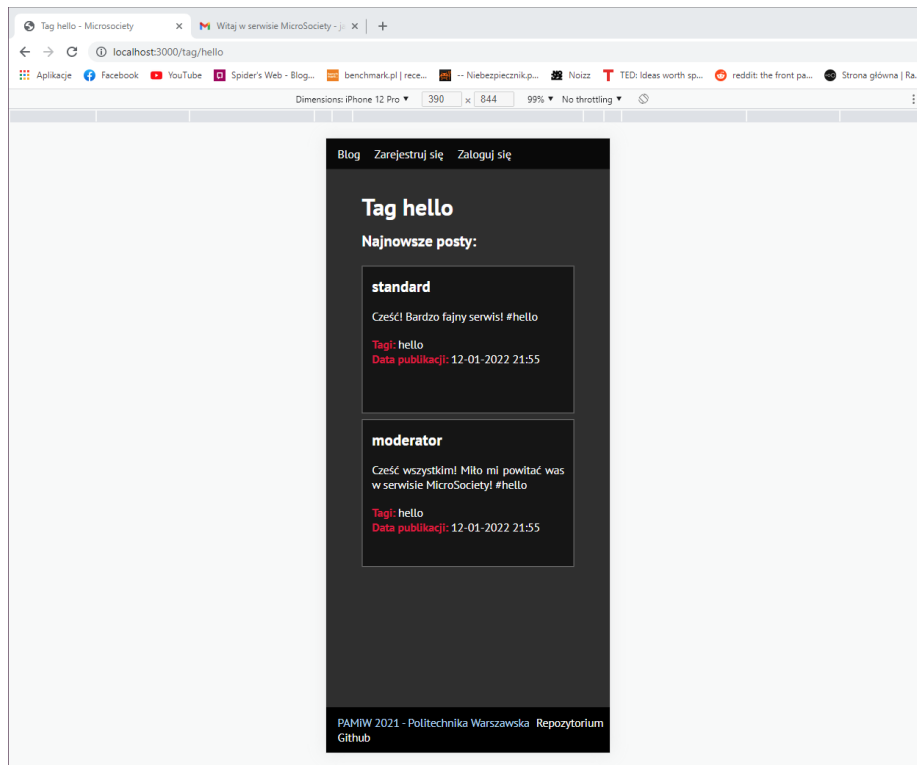


Smartfon iPhone 12 Pro:



2. Wyświetlanie (przeglądanie danych)

Przeglądanie danych zostało przedstawione w poprzednich sekcjach, chociażby w widoku bloga, czy profilu użytkownika. Oprócz tego, np. po kliknięciu na tag można wyświetlić posty, w których użyto danego tagu:



3. Filtrowanie danych (AJAX)

*

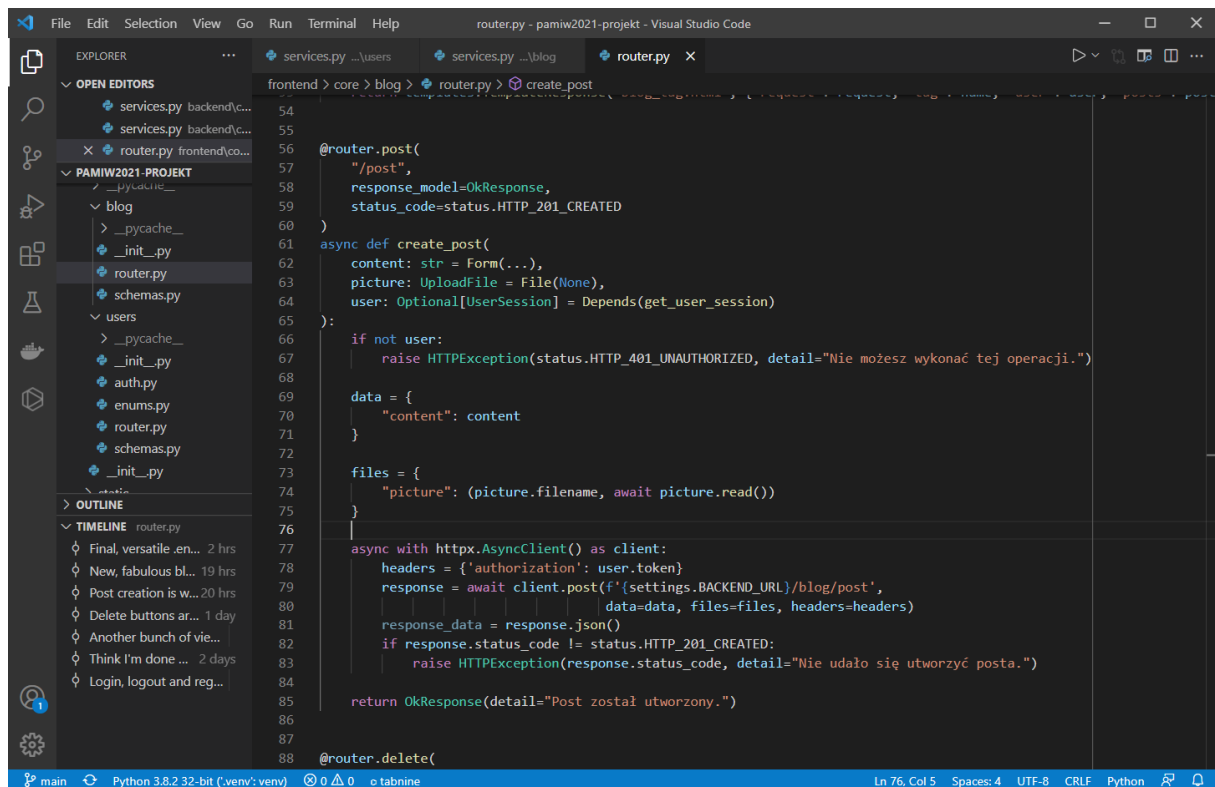
4. Zastosowanie stronicowania

*

5. Widok dodania nowego rekordu

W punkcie 7 sekcji Wymagania Ogólne pokazałem m.in. tworzenie posta ze zdjęciem z poziomu formularza dostępnego dla zalogowanych użytkowników na blogu (strona główna). Oprócz tego, w widoku rejestracji tworzony jest nowy użytkownik.

Tworzenie postów – kontroler na frontendzie:



```
54  
55  
56 @router.post(  
57     "/post",  
58     response_model=OkResponse,  
59     status_code=status.HTTP_201_CREATED  
60 )  
61 async def create_post(  
62     content: str = Form(...),  
63     picture: UploadFile = File(None),  
64     user: Optional[UserSession] = Depends(get_user_session)  
65 ):  
66     if not user:  
67         raise HTTPException(status.HTTP_401_UNAUTHORIZED, detail="Nie możesz wykonać tej operacji.")  
68  
69     data = {  
70         "content": content  
71     }  
72  
73     files = {  
74         "picture": (picture.filename, await picture.read())  
75     }  
76  
77     async with httpx.AsyncClient() as client:  
78         headers = {'authorization': user.token}  
79         response = await client.post(f'{settings.BACKEND_URL}/blog/post',  
80                                     data=data, files=files, headers=headers)  
81         response_data = response.json()  
82         if response.status_code != status.HTTP_201_CREATED:  
83             raise HTTPException(response.status_code, detail="Nie udało się utworzyć posta.")  
84  
85     return OkResponse(detail="Post został utworzony.")  
86  
87  
88 @router.delete(  
89     "/post/{id}",  
90     response_model=OkResponse,  
91     status_code=status.HTTP_204_NO_CONTENT  
92 )  
93 async def delete_post(id: int, user: Optional[UserSession] = Depends(get_user_session)):
```

6. Widok edycji rekordu

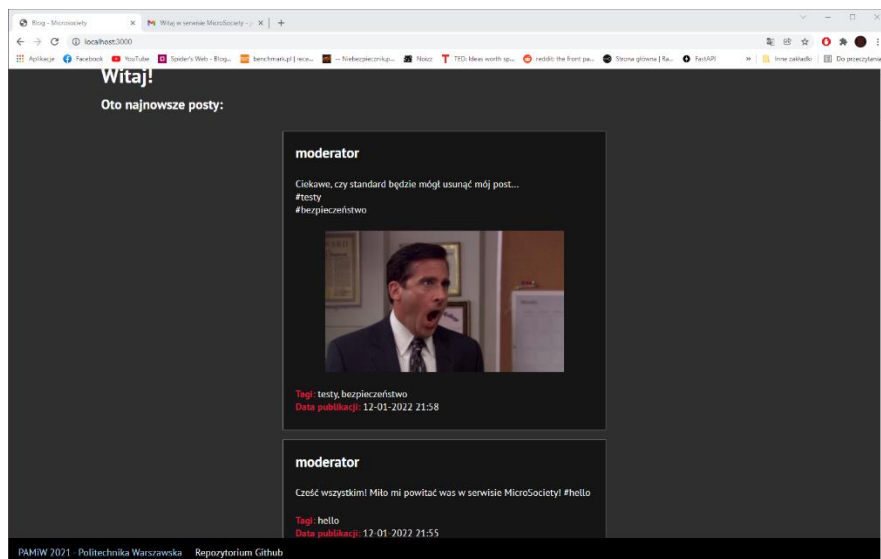
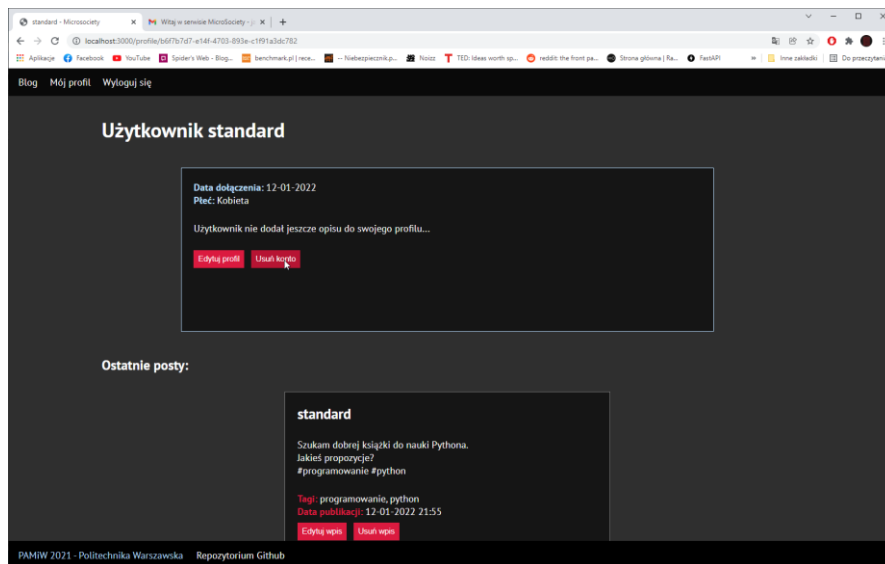
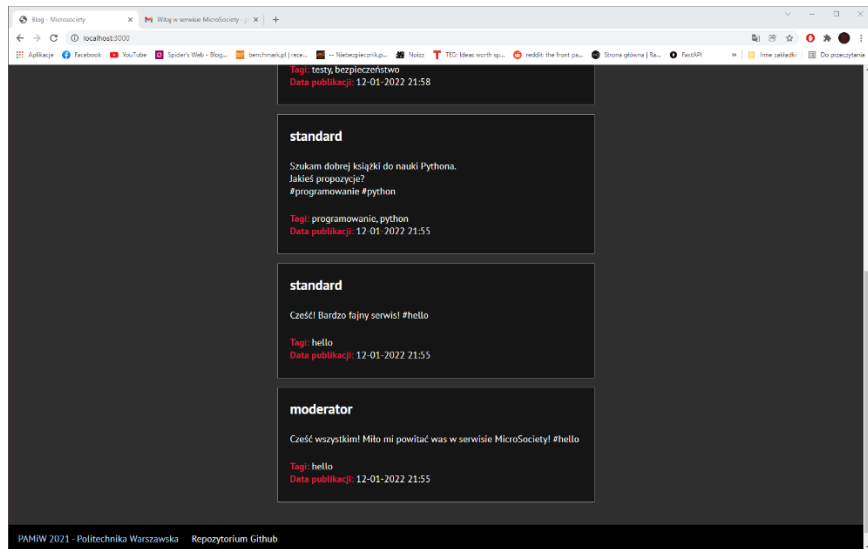
*

Przyciski do edycji widoczne na blogu nie mają podpiętej funkcji. Edycja działa tylko na backendzie.

7. Opcja usunięcia rekordu

Możliwe jest usunięcie konta, a także usuwanie postów na blogu.

Usunięcie konta powoduje usunięcie powiązanych postów z bloga – np. usunięcie konta przez użytkownika standard:



8. Opcja wgrania zdjęcia z możliwością przesłania pliku na serwer

Realizacja tego punktu widoczna jest przy tworzeniu posta (pokazano w punkcie 7, wymagania ogólne), gdzie wraz z treścią posta na serwer wysyłane jest zdjęcie w żądaniu typu Multipart.

9. Opcja instalacji aplikacji jako PWA

Niezrealizowane.

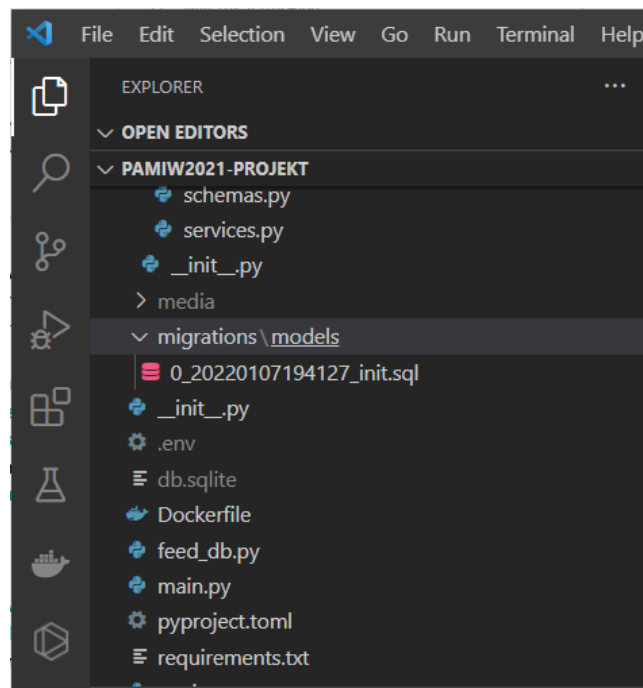
10. Wdrożenie nowoczesnego interfejsu użytkownika z wykorzystaniem szablonu HTML

Niezrealizowane.

Baza danych:

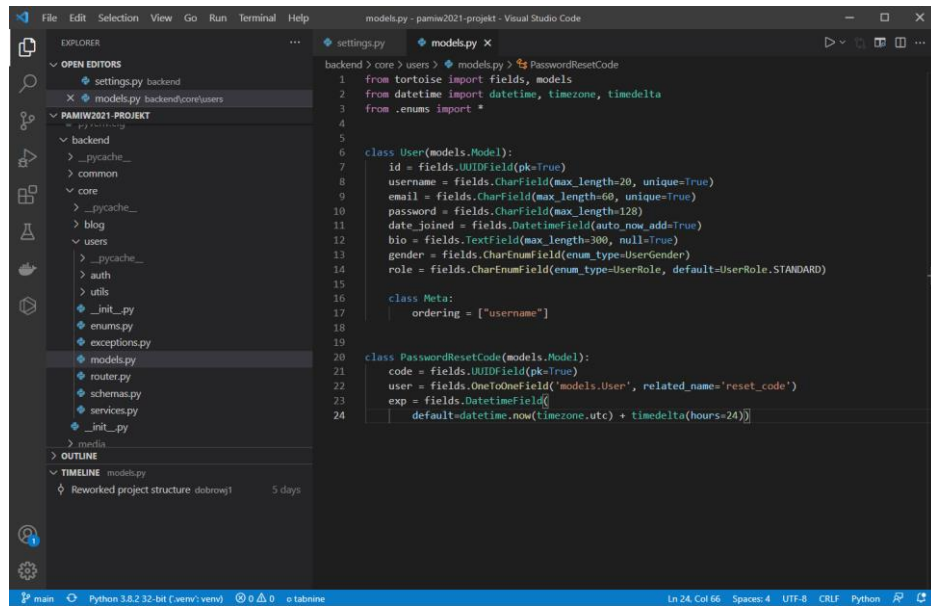
1. Zastosowanie relacyjnej bazy danych

Aplikacja backendowa może korzystać z dowolnej bazy relacyjnej obsługiwanej przez ORM Tortoise – ja korzystałem z plików SQLite.



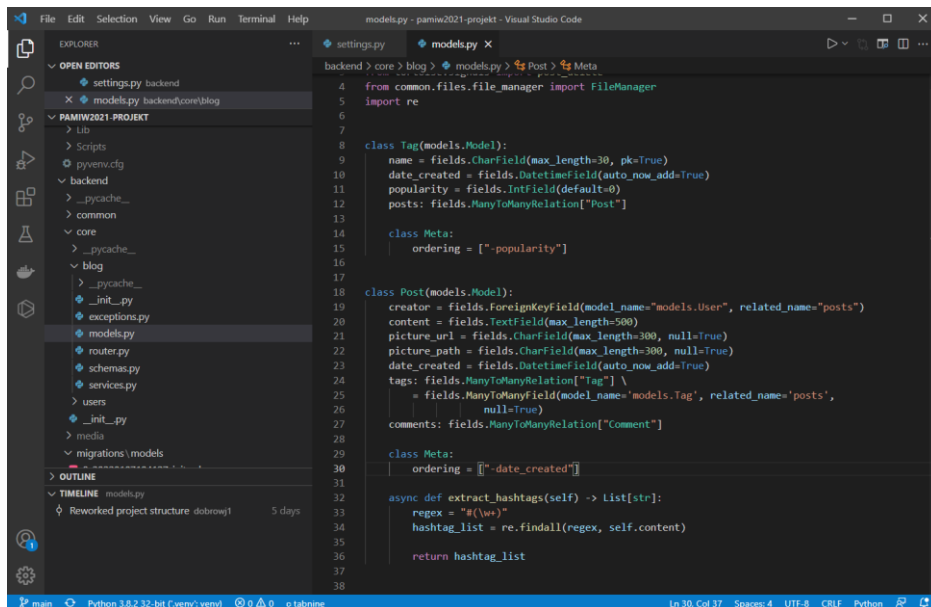
2. Użycie min. 5 tabel

Za pomocą modeli Tortoise zdefiniowałem 6 tabel: User, PasswordResetCode, Post, Comment, Like, Tag. W aplikacji możliwe jest tworzenie rekordów we wszystkich tabelach poza Comment:



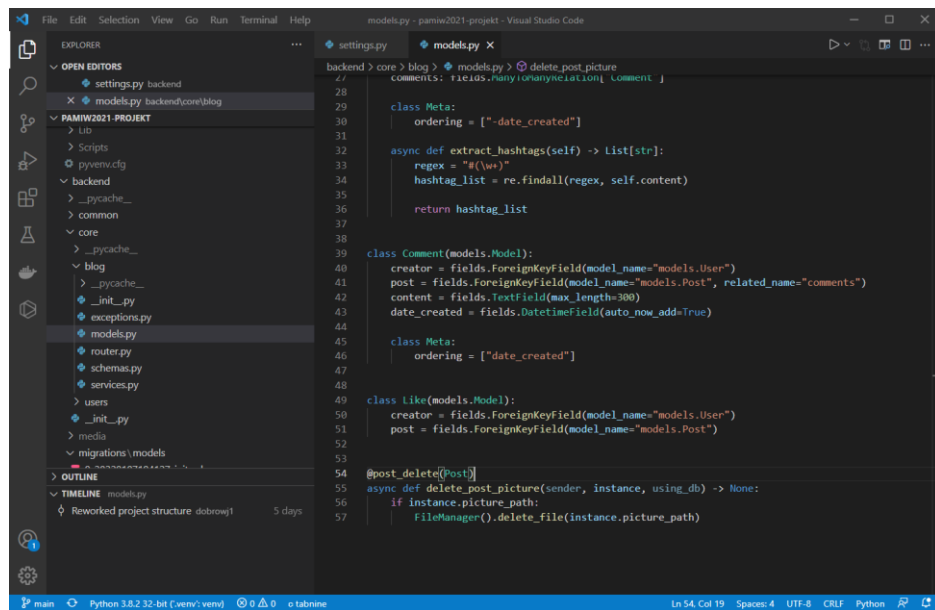
```
models.py - pamiw2021-projekt - Visual Studio Code

backends > core > users > models.py > PasswordResetCode
1 from tortoise import fields, models
2 from datetime import datetime, timezone, timedelta
3 from .enums import *
4
5
6 class User(models.Model):
7     id = fields.UUIDField(pk=True)
8     username = fields.CharField(max_length=20, unique=True)
9     email = fields.CharField(max_length=60, unique=True)
10    password = fields.CharField(max_length=128)
11    date_joined = fields.DateTimeField(auto_now_add=True)
12    bio = fields.TextField(max_length=300, null=True)
13    gender = fields.CharField(enum_type=UserGender)
14    role = fields.CharField(enum_type=UserRole, default=UserRole.STANDARD)
15
16    class Meta:
17        ordering = ["username"]
18
19
20 class PasswordResetCode(models.Model):
21    code = fields.UUIDField(pk=True)
22    user = fields.OneToOneField('models.User', related_name='reset_code')
23    exp = fields.DateTimeField(
24        default=datetime.now(timezone.utc) + timedelta(hours=24))
```



```
models.py - pamiw2021-projekt - Visual Studio Code

backends > core > blog > models.py > Post > Meta
4 from common.files.file_manager import FileManager
5 import re
6
7
8 class Tag(models.Model):
9     name = fields.CharField(max_length=30, pk=True)
10    date_created = fields.DateTimeField(auto_now_add=True)
11    popularity = fields.IntegerField(default=0)
12    posts = fields.ManyToManyField('Post')
13
14    class Meta:
15        ordering = ["-popularity"]
16
17
18 class Post(models.Model):
19    creator = fields.ForeignKeyField(model_name='models.User', related_name='posts')
20    content = fields.TextField(max_length=500)
21    picture_url = fields.CharField(max_length=300, null=True)
22    picture_path = fields.CharField(max_length=300, null=True)
23    date_created = fields.DateTimeField(auto_now_add=True)
24    tags = fields.ManyToManyField('Tag')
25    = fields.ManyToManyField(model_name='models.Tag', related_name='posts',
26        null=True)
27    comments = fields.ManyToManyField('Comment')
28
29    class Meta:
30        ordering = ["-date_created"]
31
32    async def extract_hashtags(self) -> List[str]:
33        regex = "#(\w+)"
34        hashtag_list = re.findall(regex, self.content)
35
36        return hashtag_list
37
38
```



3. Określenie kluczy głównych i obcych

Widoczne na zdjęciach w poprzedniej sekcji.

4. Zastosowanie mechanizmu ORM

Opisane w poprzednich sekcjach.

5. Użycie co najmniej jednej relacji każdego typu

Relacja 1-1: User i PasswordResetCode (użytkownik może mieć jeden aktywny kod do resetu hasła)

Relacja 1-*: User i Post, User i Comment, User i Like, Post i Like

Relacja *-*: Post i Tag