

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. СІКОРСЬКОГО»

Кафедра інформаційних систем та технологій

МЕТОДИЧНІ ВКАЗІВКИ
до виконання лабораторних робіт
з курсу «**Основи WEB технологій**»

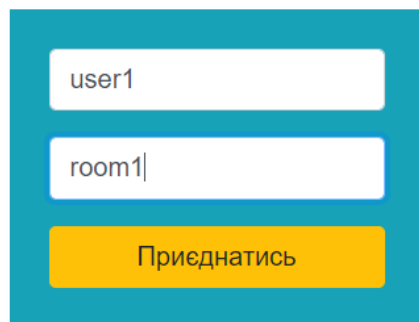
КИЇВ-2024

Лабораторна робота №1 з курсу «Основи WEB технологій»

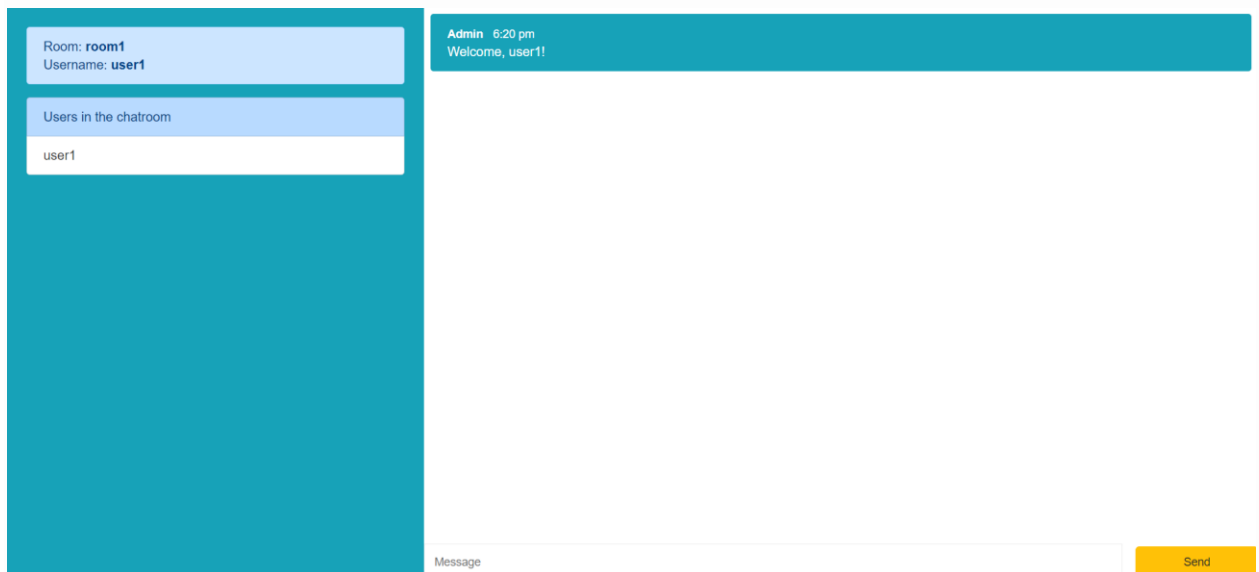
Тема: Протокол WebSocket. Використання Socket.io для розробки чат-додатків

Завдання1. Розробити додаток для обміну повідомленнями між учасниками в режимі реального часу (chat) за допомогою бібліотеки SocketIO.

Приклад роботи chat-додатку



A login form with a teal background. It contains two white input fields: the first contains 'user1' and the second contains 'room1'. Below the fields is a yellow button with the text 'Приєднатись' (Join).



A screenshot of a chat application interface. On the left is a teal sidebar with three sections: 'Room: room1 Username: user1', 'Users in the chatroom' (listing 'user1'), and a large empty teal area. On the right is a white chat area with a teal header bar displaying 'Admin 6:20 pm Welcome, user1!'. At the bottom is a white message input field with a yellow 'Send' button.

Room: room1
Username: user1

Users in the chatroom
user1

Admin 6:20 pm
Welcome, user1!

user1 6:21 pm
Hello!

Message

Send

user2

room1|

Приєднатись

Room: room1
Username: user2

Users in the chatroom
user1
user2

Admin 6:23 pm
Welcome, user2!

Message

Send

Room: room1
Username: user2

Users in the chatroom

user1

user2

Admin 6:23 pm
Welcome, user2!

user2 6:23 pm
Hi!

Message

Send

Room: room1
Username: user1

Users in the chatroom

user1

user2

Admin 6:20 pm
Welcome, user1!

user1 6:21 pm
Hello!

Admin 6:23 pm
user2 has joined!

user2 6:23 pm
Hi!

Message

Send

Room: room1
Username: user1

Users in the chatroom

user1

Admin 6:20 pm
Welcome, user1!

user1 6:21 pm
Hello!

Admin 6:23 pm
user2 has joined!

user2 6:23 pm
Hi!

Admin 6:24 pm
user2 has left!

Message

Send

Додаткова інформація.

Створення серверу за допомогою сокетів

```
var app = require('express')();
var http = require('http').Server(app);

app.get('/', function(req, res){
  res.send('<h1>Hello world</h1>');
});

http.listen(3000, function(){
  console.log('listening on *:3000');
});
```

Використання сокетів

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
  res.sendFile(__dirname + '/index.html');
});

io.on('connection', function(socket){
  console.log('a user connected');
});

http.listen(3000, function(){
  console.log('listening on *:3000');
});
```

Надсилання повідомлень

```
<script src="/socket.io/socket.io.js"></script>
<script src="https://code.jquery.com/jquery-1.11.1.js"></script>
<script>
  $(function () {
    var socket = io();
    $('form').submit(function(){
      socket.emit('chat message', $('#m').val());
      $('#m').val('');
      return false;
    });
  });
</script>
```

Привітальне повідомлення для нових користувачів чату

```
io.on('connection', function(socket){
  socket.on('chat message', function(msg){
    io.emit('chat message', msg);
  });
});
```

Оформлення звіту та порядок захисту Лабораторна робота виконується на комп'ютері. Звіт оформлюється на аркушах А4 , в якому стисло відображається зміст, хід роботи та отримані результати. . Робота захищається на парі (on-line), звіт надсилається на пошту.

Лабораторна робота №2

з курсу «Основи WEB технологій»

Тема: Створення системи авторизації сайту за допомогою JWT токенів.

Завдання.

1. При реєстрації та при оновленні користувача пароль зберігати в зашифрованому вигляді;
2. Створити запит `/users/login` на вхід
3. Застосувати авторизацію: в запиті відправити `authToken` в заголовок `Authorization`. Сервер отримує токен, верифікує його і авторизує користувача (або ні в іншому випадку).

Забезпечити використання ролей (user та admin) з відповідними діями.

Приклад виконання роботи.

localhost:5000/auth/login

POSTlocalhost:5000/auth/login

Send

ParamsAuthorizationHeaders (9)BodyScriptsTestsSettings

noneform-datax-www-form-urlencodedrawbinaryGraphQLJSON

```
1 {
2   "username": "user",
3   "password": "user"
4 }
5 }
```

BodyCookiesHeaders (7)Test Results

200 OKTime: 220 msSize: 441 BSave as example

PrettyRawPreviewVisualizeJSON

```
1 {
2   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY2ODJlZTg2ZDhiNGZlMGQzNDNkODJhOStInSwiaWF0IjoxNzIxMDU4NDQ1LjE3eHA10jE3MjExNDQ4MDV9.eyJwIjoiZmhmIjQr9BB_0XfAK-drXKgH4i5eymUJ4RLA"
3 }
```

localhost:5000/auth/users

GETlocalhost:5000/auth/users

Send

ParamsAuthorizationHeaders (10)BodyScriptsTestsSettings

9 hidden

| Key | Value | Description |
|---------------|---|-------------|
| Authorization | Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZC... | |
| Key | Value | Description |

BodyCookiesHeaders (7)Test Results

200 OKTime: 75 msSize: 850 BSave as example

PrettyRawPreviewVisualizeJSON

```
4 {
5   "USER"
6 },
7   "_id": "6682be86d8b4fe0d343d82a9",
8   "username": "user",
9   "password": "$2a$07$4f0LcAJog3qYPN13Q7hs6ubcwn1UvATSmjZyWh99H1ycLw7qLT8i",
10  "__v": 0
11 },
12 {
13   "roles": [
14     "ADMIN"
15 ],
16   "_id": "6682be99d8b4fe0d343d82ad",
17   "username": "user1",
```

The image displays two screenshots of the MongoDB Compass web interface. The top screenshot shows the 'auth_roles.users' collection. The left sidebar indicates the database 'auth_roles' and the collection 'users'. The main panel shows two documents in the collection. The first document has a role of 'USER' and a specific password. The second document has a role of 'ADMIN' and a different password. The bottom screenshot shows the 'auth_roles.roles' collection. The left sidebar indicates the database 'auth_roles' and the collection 'roles'. The main panel shows two documents in the collection, each representing a role: 'USER' and 'ADMIN', both with a value of 'true'.

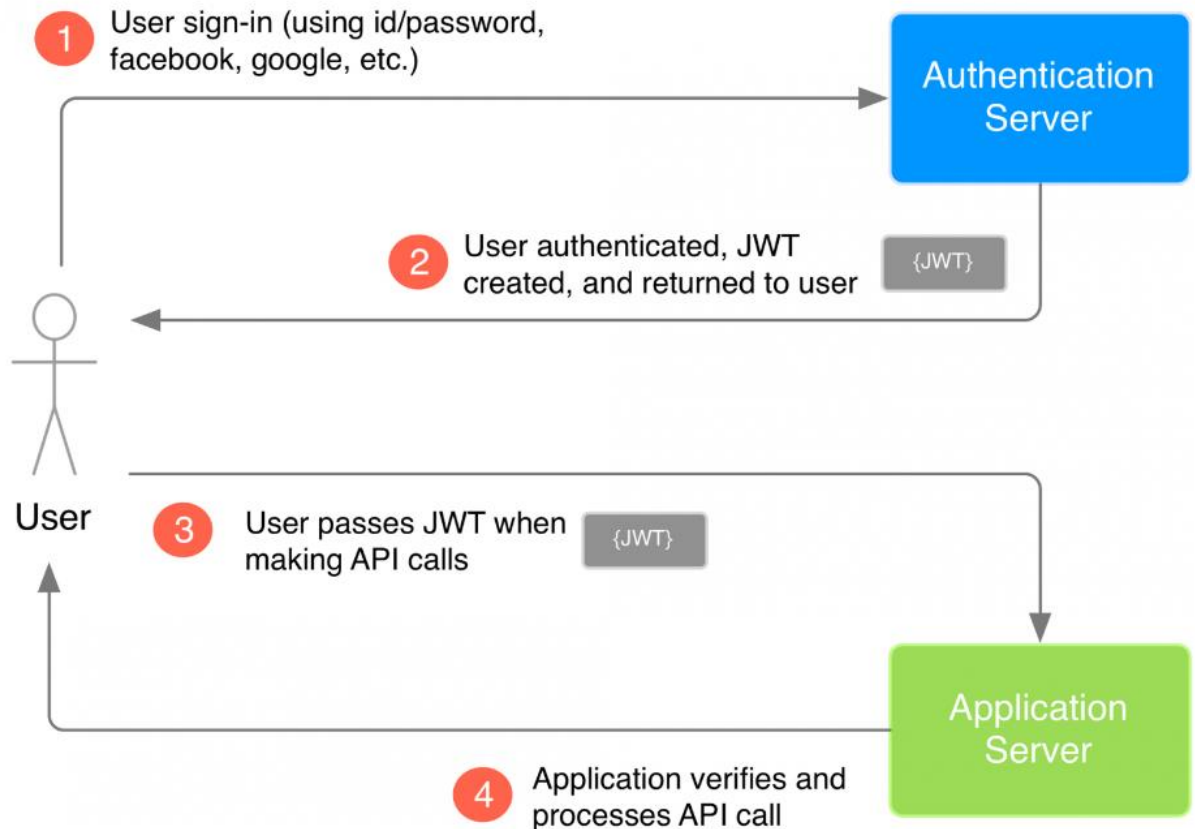
Додаткова інформація.

JSON Web Token (JWT)- це JSON об'єкт, який визначений у відкритому стандарті RFC 7519 . Він вважається одним із безпечних способів передачі інформації між двома учасниками. Для його створення необхідно визначити заголовок (header) із загальною інформацією по токenu, корисні дані (payload), такі як ID користувача, його роль і т.д. та підписи (signature).

До речі, правильно JWT вимовляється як/dʒɔt/

Простими словами, JWT - це лише рядок у наступному форматі header.payload.signature.

Припустимо, що ми хочемо зареєструватись на сайті. У нашому випадку є три учасники - користувач user, сервер програми application serverта сервер аутентифікації authentication server. Сервер автентифікації забезпечуватиме користувача токеном, за допомогою якого він пізніше зможе взаємодіяти з додатком.



Додаток використовує JWT для перевірки аутентифікації користувача таким чином:

Спершу користувач заходить на сервер аутентифікації за допомогою аутентифікаційного ключа (це може бути пара логін/пароль або Facebook ключ, або Google ключ, або ключ від іншого обліку).

Потім сервер аутентифікації створює JWT та відправляє його користувачеві.

Коли користувач робить запит до програми API, він додає до нього отриманий раніше JWT.

Коли користувач робить API запит, програма може перевірити за переданим із запитом JWT чи є користувач тим, за кого себе видає. У цій схемі сервер програми налаштований так, що зможе перевірити, чи є вхідний JWT саме тим, що був створений сервером аутентифікації (процес перевірки буде пояснений пізніше детальніше).

Структура JWT

JWT складається з трьох частин: заголовок header, корисні дані payload та підпис signature. Давайте пройдемося по кожній із них.

Крок 1. Створюємо HEADER

Хедер JWT містить інформацію про те, як повинен обчислюватися JWT підпис. Хедер це теж JSON об'єкт, який виглядає наступним чином:

```
header = { "alg": "HS256", "typ": "JWT" }
```

Поле typ не говорить нам нічого нового тільки те, що це JSON Web Token . Цікавішим тут буде поле alg, яке визначає алгоритм хешування. Він використовуватиметься під час створення підпису. HS256- Не що інше, як HMAC-SHA256, для його обчислення потрібен лише один секретний ключ (детальніше про це в кроці 3). Ще може використовуватися інший алгоритм RS256— на відміну від попереднього, він є асиметричним та створює два ключі: публічний та приватний. За допомогою приватного ключа створюється підпис, а за допомогою публічного лише перевіряється справжність підпису, тому нам не потрібно турбуватися про його безпеку.

Крок 2. Створюємо PAYLOAD

Payload - це корисні дані, які зберігаються всередині JWT . Ці дані також називають JWT-claims (заявки). У прикладі, який ми розглядаємо, сервер автентифікації створює JWT з інформацією про id користувача — userId .

```
payload = { "userId": "b08f86af-35da-48f2-8fab-cef3904660bd" }
```

Ми поклали тільки одну заявку (claim) у payload . Ви можете покласти стільки заявок , скільки захочете. Існує список стандартних заявок для JWT payload — деякі з них:

```
iss (issuer) - визначає додаток, з якого відправляється токен.  
sub (subject) - Визначає тему токена.  
exp (expiration time) - час життя токена.
```

Ці поля можуть бути корисними при створенні JWT , але вони не є обов'язковими. Якщо хочете знати весь список доступних полів для JWT , можете заглянути у Wiki . Але варто пам'ятати, що чим більше передається інформації, тим більший вийде сам JWT . Зазвичай із цим не буває проблем, але все-таки це може негативно позначитися на продуктивності та викликати затримки у взаємодії із сервером.

Крок 3. Створюємо SIGNATURE

Підпис обчислюється з використанням наступного псевдо-коду:

```
const SECRET_KEY = 'cAtwalkkEy'  
const unsignedToken = base64urlEncode(header) + '.' +  
base64urlEncode(payload)  
const signature = HMAC-SHA256(unsignedToken, SECRET_KEY)
```

Алгоритм base64url кодує хедер та payload, створені на 1 та 2 кроці. Алгоритм з'єднує закодовані рядки через точку. Потім отриманий рядок хешується алгоритмом, заданим у хедері на основі нашого секретного ключа.

```
// header eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9  
//payload  
eyJ1c2VySWQiOiJiMDhmODZhZi0zNWRhLTQ4ZjItOGZhYi1jZWYzOTA0NjYwYmQifQ// signature -xN_h82PHVTCMA9vdoHrcZxH-x5mb11y1537t3rGzcM
```

Крок 4. Тепер об'єднаємо всі три JWT компоненти разом

Тепер, коли ми маємо всі три складові, ми можемо створити наш JWT . Це досить просто, ми поєднаємо всі отримані елементи в рядок через точку.

```
const token = encodeBase64Url(header) + '.' +  
encodeBase64Url(payload) + '.' + encodeBase64Url(signature)  
// JWT Token  
//  
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VySWQiOiJiMDhmODZhZi0zNWRhLTQ4ZjItOGZhYi1jZWYzOTA0NjYwYmQifQ.-xN_h82PHVTCMA9vdoHrcZxH-x5mb11y1537t3rGzcM
```

Ви можете спробувати створити свій власний JWT на сайті jwt.io.

Повернемося до нашого прикладу. Тепер сервер аутентифікації може надсилати користувачу JWT .

Як JWT захищає наші дані?

Дуже важливо розуміти, що використання JWT НЕ приховує та не маскує дані автоматично. Причина, чому JWT використовуються - це перевірка, що надіслані дані були дійсно відправлені авторизованим джерелом. Як було продемонстровано вище, дані всередині JWT закодовані та підписані, зверніть увагу, це не одне й те, що зашифровано. Мета кодування даних – перетворення структури. Підписані дані дозволяють одержувачу даних перевірити автентифікацію джерела даних. Таким чином кодування та підпис даних не захищає їх. З іншого боку, головною метою шифрування є

захист даних від неавторизованого доступу. Для більш детального пояснення відмінності між кодуванням та шифруванням, а також про те, як працює хешування, дивіться цю статтю . Оскільки JWT лише закодована і підписана, і оскільки JWT не зашифрована, JWT не гарантує жодної безпеки для чутливих (sensitive) даних.

Крок 5. Перевірка JWT

У нашому простому прикладі з 3 учасників ми використовуємо JWT , який підписаний за допомогою HS256алгоритму і тільки сервер автентифікації та сервер програми знають секретний ключ. Сервер програми отримує секретний ключ від сервера автентифікації під час встановлення автентифікаційних процесів. Оскільки програма знає секретний ключ, коли користувач робить API-запит з прикладеним до нього токеном, програма може виконати той же алгоритм підписування до JWT , що в кроці 3 . Додаток може потім перевірити цей підпис, порівнюючи його зі своїм власним, обчисленим хешуванням. Якщо підписи збігаються, отже JWT валідний, тобто. прийшов від перевіреного джерела. Якщо підписи не збігаються, то щось пішло не так — можливо, це є ознакою потенційної атаки. Таким чином, перевіряючи JWT , додаток додає довірчий шар (a layer of trust) між собою та користувачем.

Оформлення звіту та порядок захисту Лабораторна робота виконується на комп'ютері. Звіт оформлюється на аркушах А4 , в якому стисло відображається зміст, хід роботи та отримані результати. Робота захищається на парі (on-line), звіт надсилається на пошту.

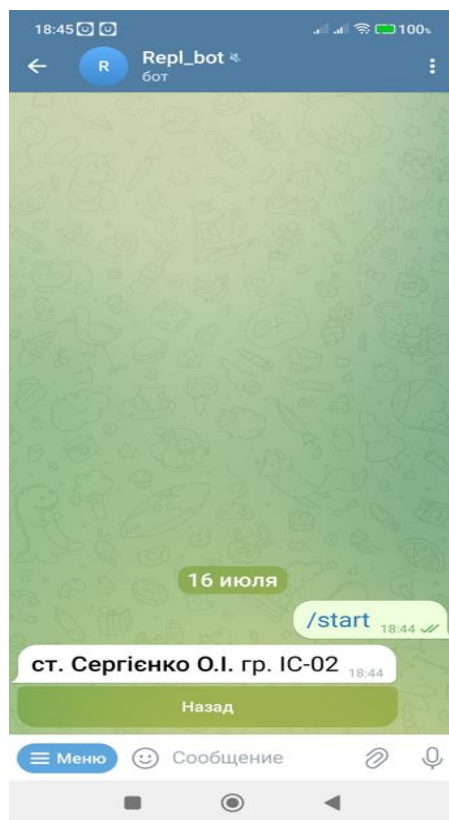
Лабораторна робота №3 з курсу «Основи WEB технологій» Тема: Створення telegram-боту з меню та запитом до ChatGPT. Завдання1.

Створити telegram-бот з меню та задеплоїти його на сервісі <https://pythonanywhere.com/>.

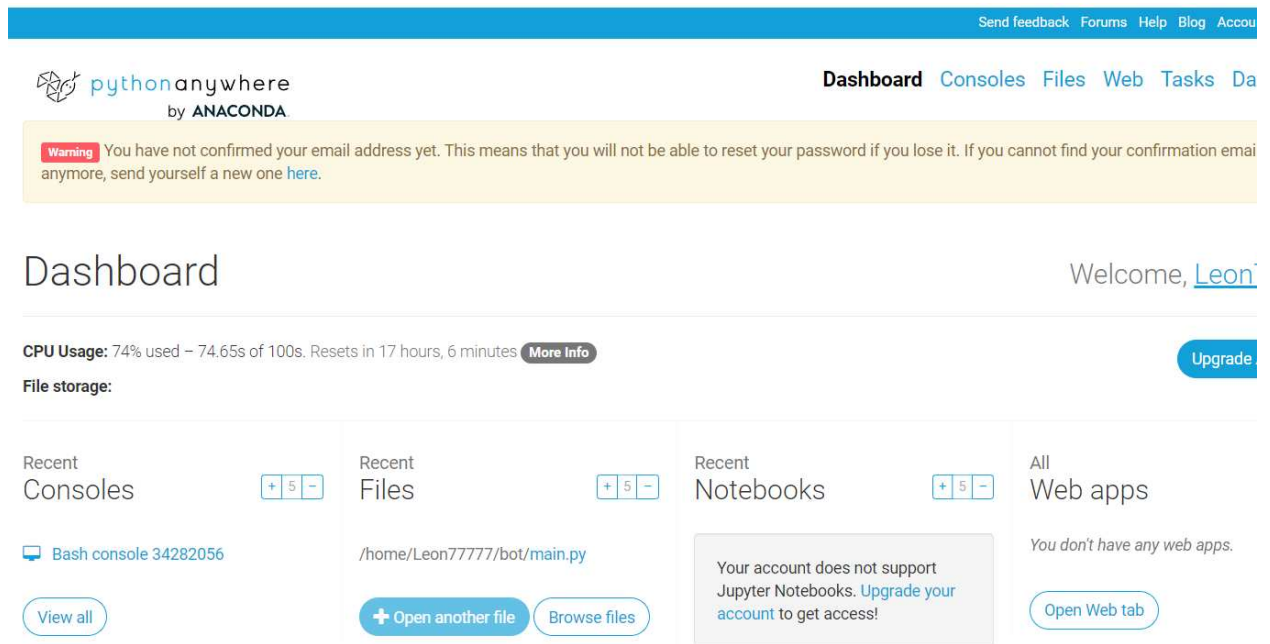
Структура меню:

- Студент (прізвище, група)
- IT-технології (....)
- Контакти (тел., e-mail)
- Prompt ChatGPT

Приклад виконання завдання







Додаткова інформація

Створення безкоштовного ChatGPT телеграм бота на Python

Підготовка

Отже, для початку нам потрібно зробити кілька перевірок на необхідні інструменти:

Python - рекомендована версія 3.10 і вище

Щоб перевірити, чи Python встановлено в командному рядку, введіть:

```
python -V
```

Якщо командний рядок вивів версію Python - все в порядку, можемо продовжувати, в іншому випадку встановіть Python з офіційного сайту і при установці обов'язково переконайтеся, що поставлена галочка на "Add to PATH", продовжуємо

Бібліотека для взаємодії з Telegram API

Щоб її встановити, необхідно використовувати команду:

```
pip install python-telegram-bot
```

Бібліотека взаємодії з ChatGPT

Ми не будемо використовувати офіційну бібліотеку openai, замість неї буде використано revChatGPT посилання на гітхаб автора- клік , щоб її встановити необхідно використовувати команду:

```
pip install --upgrade revChatGPT
```

Токен телеграм-бота

Отримати його можна створивши новий або використовуючи вже існуючий бот у BotFather <https://t.me/BotFather>

Токен ChatGPT

Якщо ви зареєстровані в Openai і можете взаємодіяти з ChatGPT на офіційному сайті, то щоб його отримати треба перейти за посиланням <https://chat.openai.com/api/auth/session> і скопіювати значення "accessToken"

Використання бібліотеки revChatGPT

Отже для початку давайте розберемося як працює ця бібліотека і потім інтегруємо її в бота, зробимо скрипт з постійним циклом для подібної чату всередині консолі:

```
from revChatGPT.V1 import Chatbot #імпортуємо бібліотеку
access_token="тут замість тексту вставте ваш accessToken"
#задаємо змінну токена
```

```
chatbot = Chatbot(config={"access_token":access_token})
#ініціалізуємо чатбота
```

```
while True:
```

```
    message = input("Ви:") #створюємо введення тексту
    output = chatbot.ask(message) #задаємо запит ChatGPT з
набраним текстом
```

```
    print(f"ChatGPT:{output}") #виводимо відповідь ChatGPT
```

Ви можете спробувати використовувати цей скрипт для перевірки токена та інших цілей, у нашому випадку ознайомлення з роботою revChatGPT

Створення Telegram бота

Тепер приступимо до створення коду безпосередньо самого бота

1.Імпорт бібліотек telegram і ChatGPT:

```
from telegram.ext import Updater, CommandHandler,
MessageHandler, filters
from revChatGPT.V1 import Chatbot
```

2. Ініціалізуємо бота та ChatGPT

```
# тут треба вставити свої токени
TOKENTG = "TELEGRAM_TOKEN"
TOKENGPT = "CHATGPT_TOKEN"
updater = Updater(TOKENTG)
chatbot = Chatbot(config={"access_token":TOKENGPT})
```

3. Створюємо функцію відповіді на повідомлення користувача

```
def chatgpt_reply(update, context):
    context.bot.send_chat_action(chat_id=update.effective_chat.id,
    action=ChatAction.TYPING)

    text = update.message.text

    reply = chatbot.ask(text)

    update.message.reply_text(reply)
```

4. Створюємо обробник повідомлень

```
echo_handler = MessageHandler(Filters.text & (~Filters.command),
chatgpt_reply)

updater.dispatcher.add_handler(echo_handler)
```

5. Запуск робота

```
updater.start_polling()
updater.idle()
```

Готово! наш код готовий до роботи, тепер його треба запустити:

```
python bot.py
```

*замість bot.py повинен бути файл де ви зберегли код

Оформлення звіту та порядок захисту Лабораторна робота виконується на комп'ютері. Звіт оформлюється на аркушах А4, в якому стисло відображається зміст, хід роботи та отримані результати. Робота захищається на парі (on-line), звіт надсилається на пошту.

Лабораторна робота №4

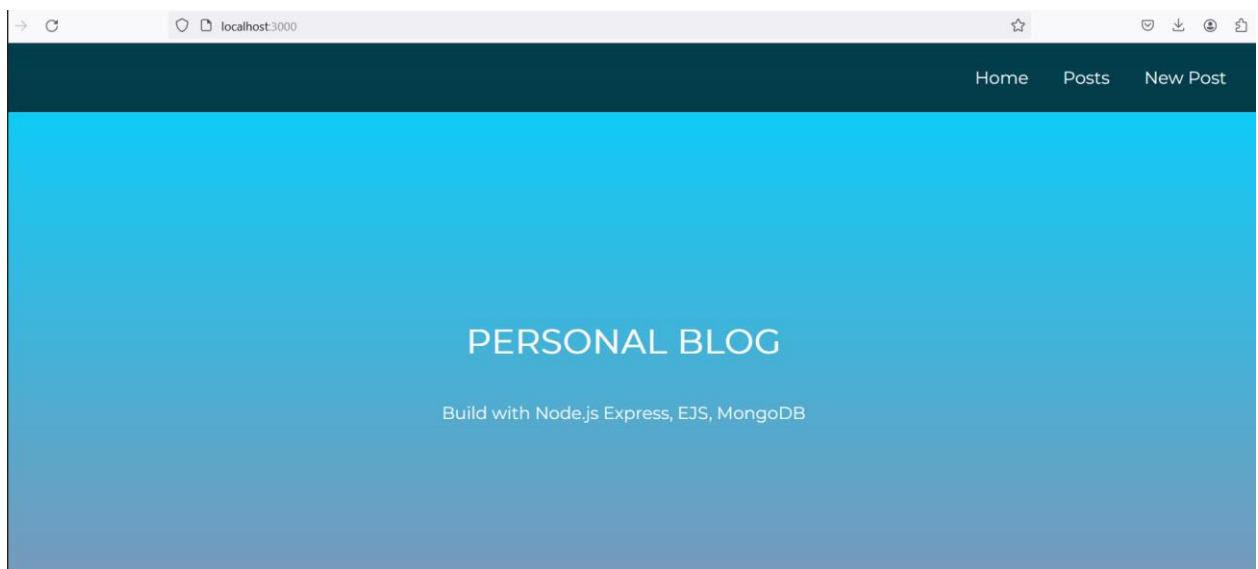
з курсу «Основи WEB технологій»

Тема: NodeJS. Робота з БД. Додаток що реалізує CRUD операції в БД.

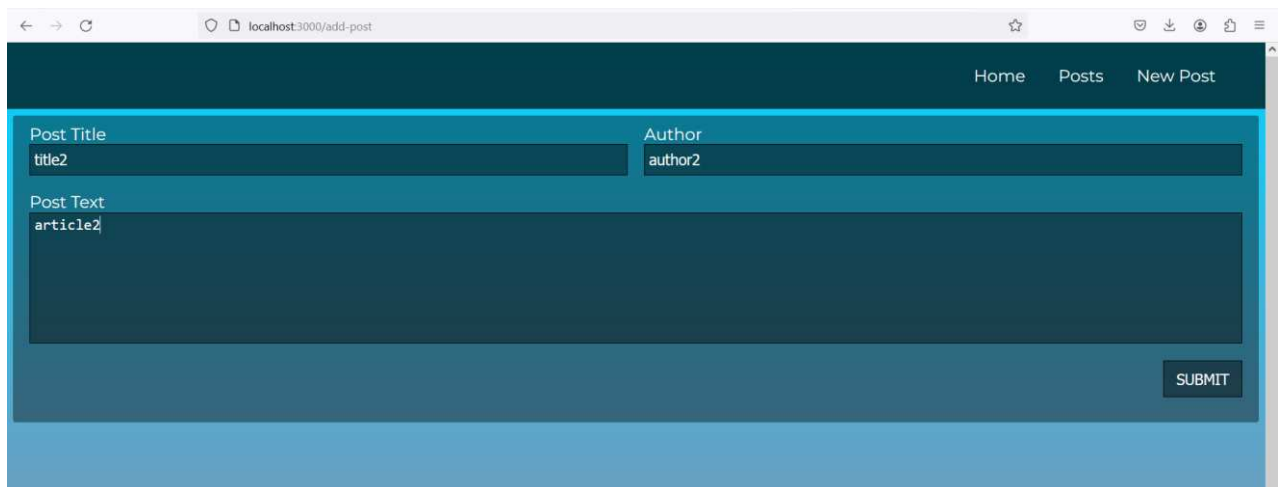
Завдання.

- Створити додаток, що реалізує CRUD операції з БД – додавання, читання, редагування та видалення записів БД (Додаток 1. Таблиця 1).
- Забезпечити роутінг запитів та виведення результатів запитів на WEB-сторінку.
- Додати нові роути для виведення інформації у вигляді json-файлу (API).

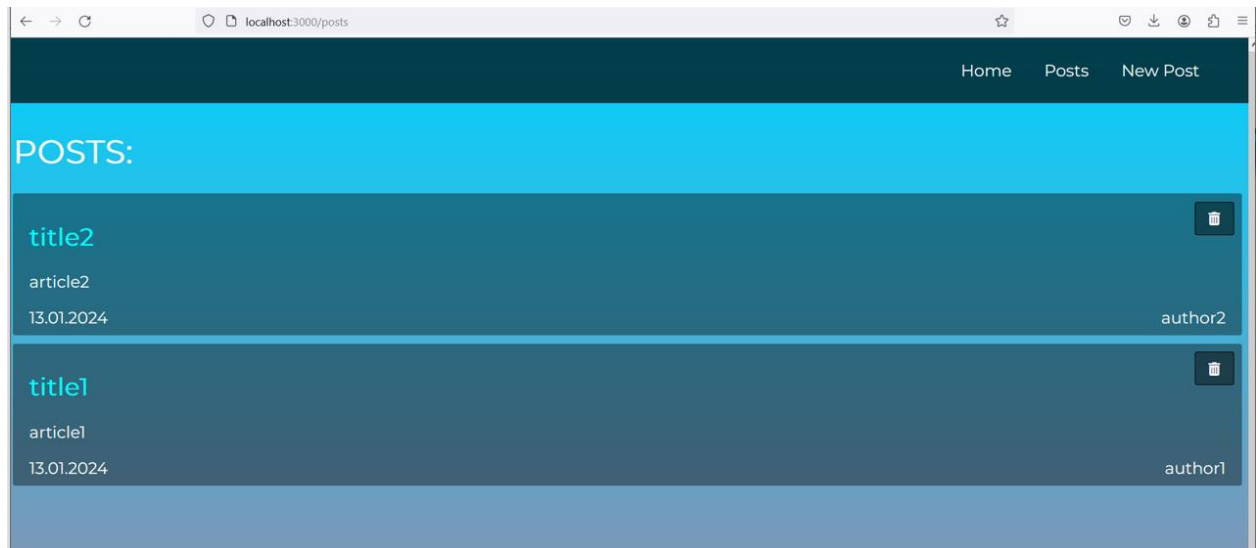
Приклад результатів виконання роботи



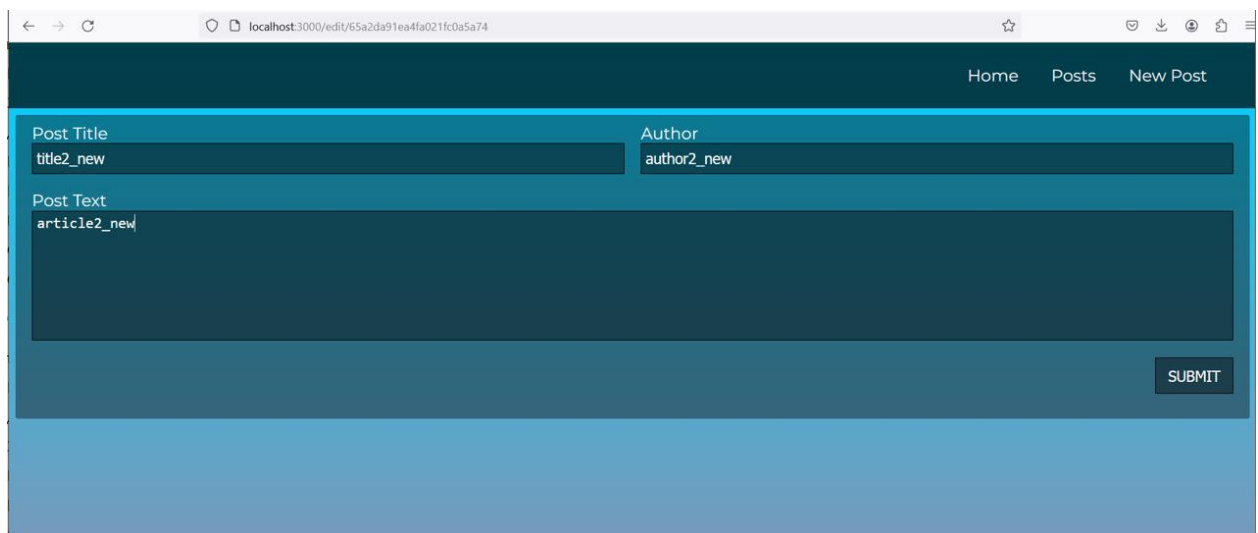
Головна сторінка



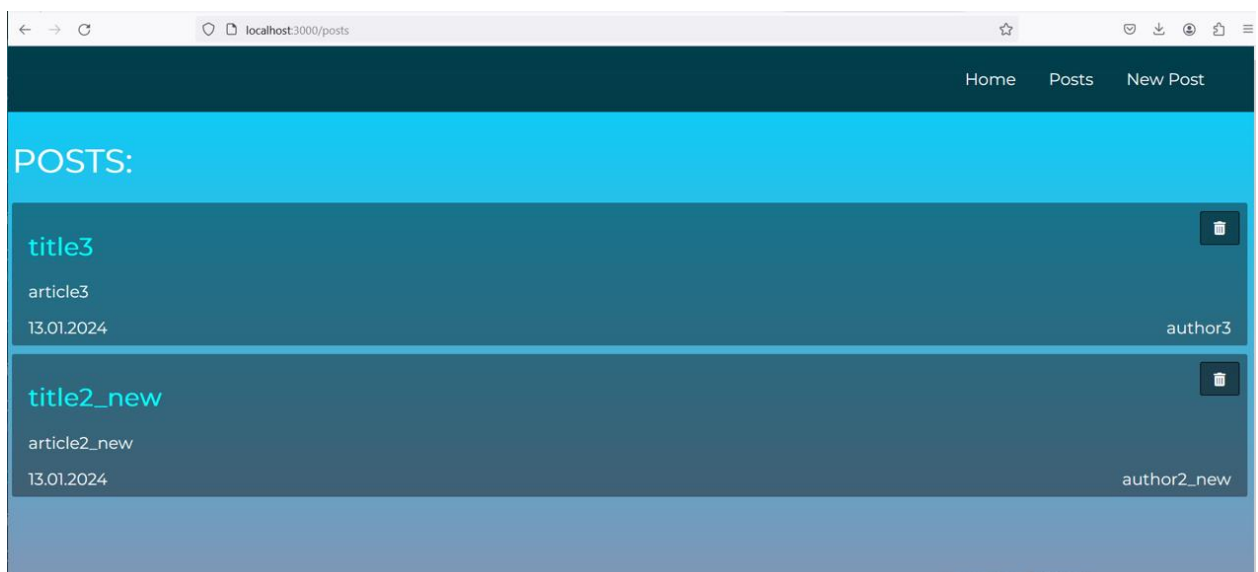
Сторінка додавання замітки

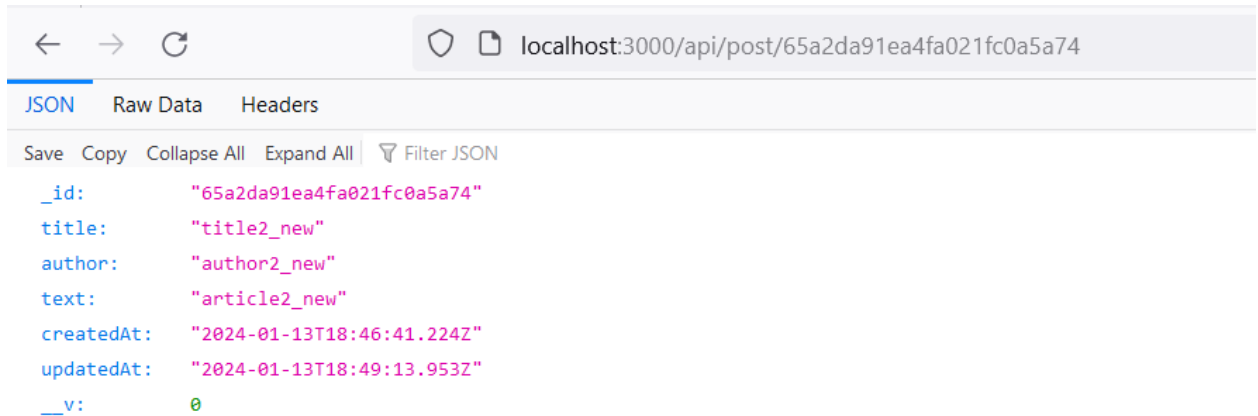


Сторінка перегляду заміток

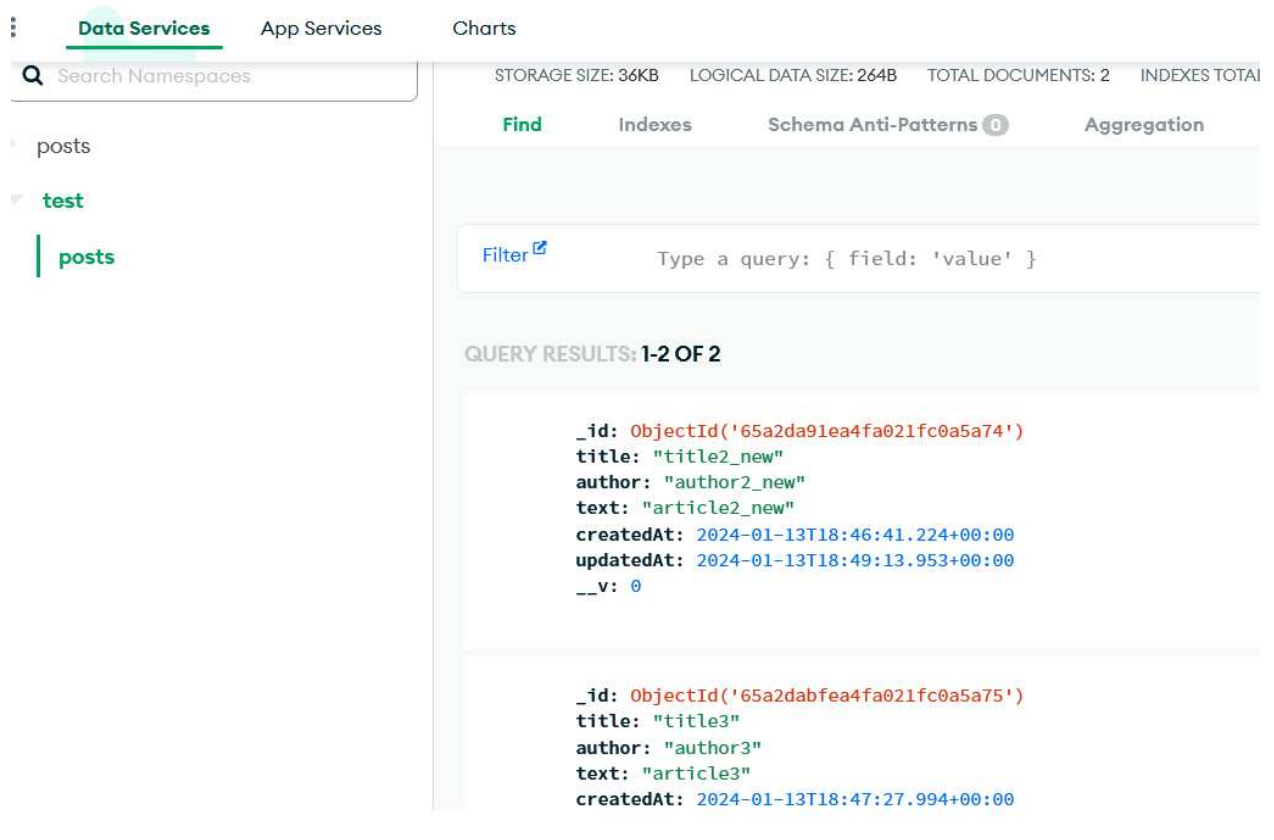


Сторінка редагування заміток





Додавання нового роута та виведення інформації з БД у вигляді JSON-файлу



Інформація в БД після виконання CRUD-операцій

Додаткова інформація



Створення серверу та підключення БД.

```
const app = express();
app.set("view engine", "ejs");
const PORT = 3000;
const db =
  "mongodb+srv://xxxxxxxx:xxxxx@cluster0.fdvk7x7.mongodb.net/?retryWrites=true&w=majority";
mongoose
  .connect(db, { useNewUrlParser: true, useUnifiedTopology: true })
  .then((res) => console.log("Connected to DB"))
  .catch((error) => console.log(error));

app.listen(PORT, (error) => {
  error ? console.log(error) : console.log(`listening port ${PORT}`);
});
```

Створення схеми БД

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const postSchema = new Schema({
  text: {
    type: String,
    required: true,
  },
  title: {
    type: String,
    required: true,
  },
  author: {
    type: String,
    required: true,
  }
}, { timestamps: true });

const Post = mongoose.model('Post', postSchema);
module.exports = Post;
```

Створення роутингу

```
const router = express.Router();

router.get('/posts/:id', getPost);
router.delete('/posts/:id', deletePost);
router.get('/edit/:id', getEditPost);
```

```
router.put('/edit/:id', editPost);
router.get('/posts', getPosts);
router.get('/add-post', getAddPost);
router.post('/add-post', addPost);
```

```
module.exports = router;
```

CRUD- операції з БД

```
const getPost = (req, res) => {
  const title = 'Post';
  Post
    .findById(req.params.id)
    .then(post => res.render(createPath('post'), { post, title }))
    .catch((error) => handleError(res, error));
}

const deletePost = (req, res) => {
  Post
    .findByIdAndDelete(req.params.id)
    .then((result) => {
      res.sendStatus(200);
    })
    .catch((error) => handleError(res, error));
}

const getEditPost = (req, res) => {
  const title = 'Edit post';
  Post
    .findById(req.params.id)
    .then(post => res.render(createPath('edit-post'), { post, title }))
    .catch((error) => handleError(res, error));
}

const editPost = (req, res) => {
  const { title, author, text } = req.body;
  const { id } = req.params;
  Post
    .findByIdAndUpdate(id, { title, author, text })
    .then((result) => res.redirect(`/posts/${id}`))
    .catch((error) => handleError(res, error));
}

const getPosts = (req, res) => {
  const title = 'Posts';
  Post
    .find()
    .sort({ createdAt: -1 })
    .then(posts => res.render(createPath('posts'), { posts, title }))
    .catch((error) => handleError(res, error));
}

const getAddPost = (req, res) => {
  const title = 'Add Post';
  res.render(createPath('add-post'), { title });
}
```

```
const addPost = (req, res) => {  
  const { title, author, text } = req.body;  
  const post = new Post({ title, author, text });  
  post  
    .save()  
    .then((result) => res.redirect('/posts'))  
    .catch((error) => handleError(res, error));  
}
```

Більш детальну інформацію по роботі з NodeJS можна знайти за посиланням [NodeJS](#)

Оформлення звіту та порядок захисту Лабораторна робота виконується на комп'ютері. Звіт оформлюється на аркушах А4 , в якому стисло відображається зміст, хід роботи та отримані результати. Робота захищається на парі (on-line), звіт надсилається на пошту.

Лабораторна робота №5

Тема: GraphQL. Створення Schema GraphQL та Resolvers. Створення Query та Mutation.

Завдання.

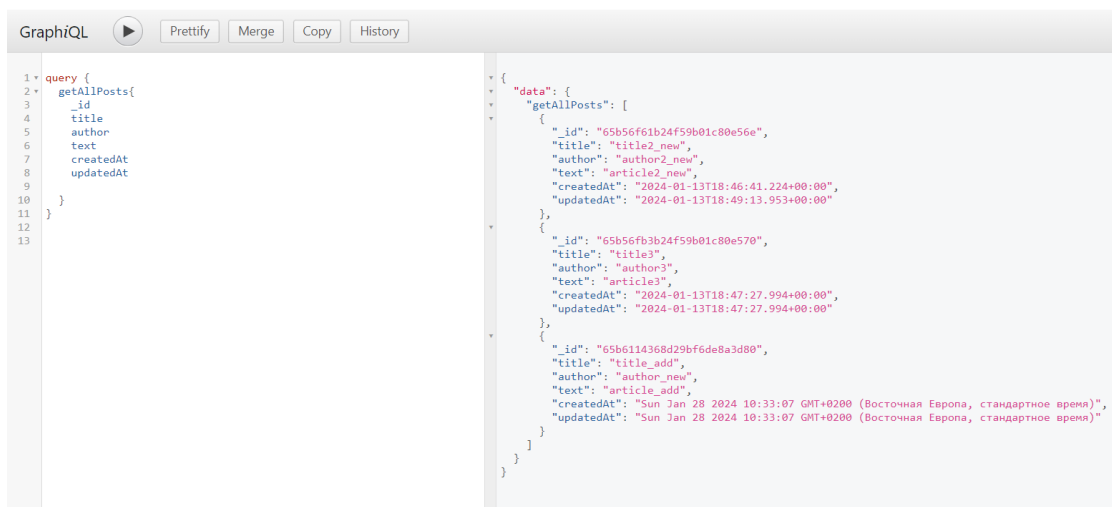
- На свій БД (розробленої в лаб. роб. #4) за допомогою Schema Definition Language (SDL) створити схему GraphQL.
- Додати Resolvers для виконання операцій GraphQL.
- Створити та виконати Query та Mutation для виконання операцій додавання, редагування та видалення інформації (CRUD) в БД.
- Виконати дослідження роботи створених query та mutation за допомогою Postman.

Приклад результатів виконання роботи



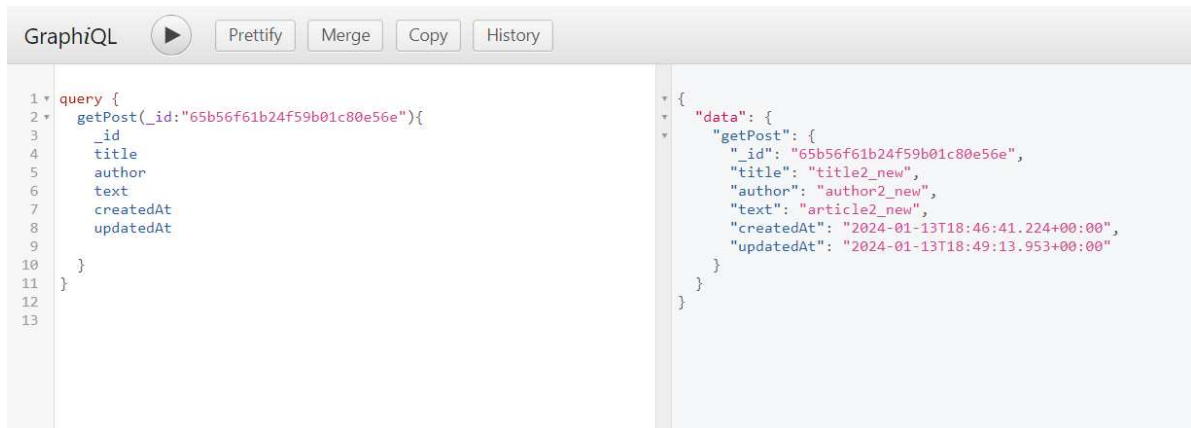
The screenshot shows the GraphQL Playground interface. On the left, the query editor contains a mutation: `mutation { createPost(title:"title_add", author:"author_new", text:"article_add") { _id title author text createdAt updatedAt } }`. On the right, the JSON response is displayed: `{ "data": { "createPost": { "_id": "65b6114368d29bf6de8a3d80", "title": "title_add", "author": "author_new", "text": "article_add", "createdAt": "Sun Jan 28 2024 10:33:07 GMT+0200 (Восточная Европа, стандартное время)", "updatedAt": "Sun Jan 28 2024 10:33:07 GMT+0200 (Восточная Европа, стандартное время)" } } }`.

Mutation для додавання документа



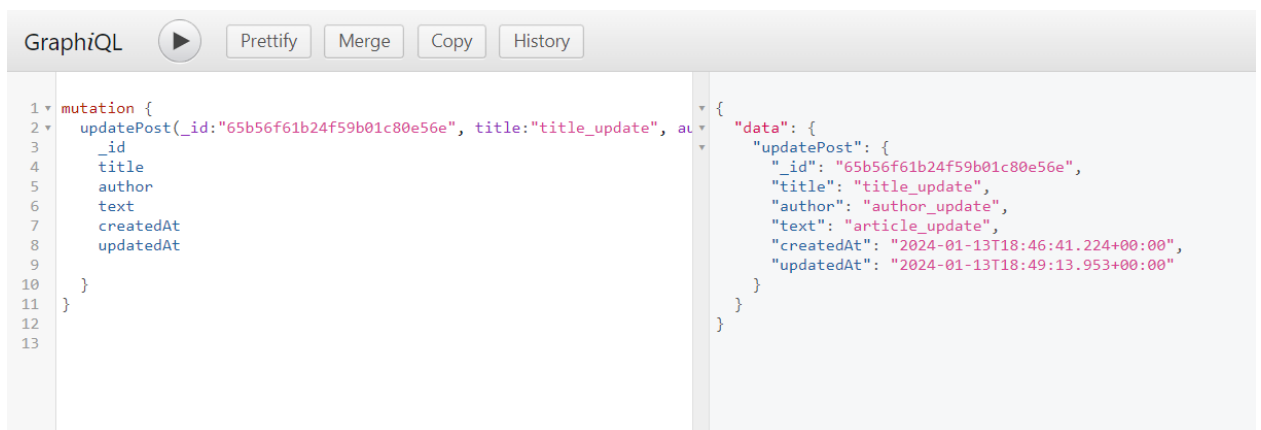
The screenshot shows the GraphQL Playground interface. On the left, the query editor contains a query: `query { getAllPosts { _id title author text createdAt updatedAt } }`. On the right, the JSON response is displayed: `{ "data": { "getAllPosts": [{ "_id": "65b56f61b24f59b01c80e56e", "title": "title2_new", "author": "author2_new", "text": "article2_new", "createdAt": "2024-01-13T18:46:41.224+00:00", "updatedAt": "2024-01-13T18:49:13.953+00:00" }, { "_id": "65b56fb3b24f59b01c80e570", "title": "title3", "author": "author3", "text": "article3", "createdAt": "2024-01-13T18:47:27.994+00:00", "updatedAt": "2024-01-13T18:47:27.994+00:00" }, { "_id": "65b6114368d29bf6de8a3d80", "title": "title_add", "author": "author_new", "text": "article_add", "createdAt": "Sun Jan 28 2024 10:33:07 GMT+0200 (Восточная Европа, стандартное время)", "updatedAt": "Sun Jan 28 2024 10:33:07 GMT+0200 (Восточная Европа, стандартное время)" }] } }`.

Query для виведення всіх документів з БД



The screenshot shows the GraphQL IDE interface. On the left, a query is defined: `query { getPost(_id: "65b56f61b24f59b01c80e56e"){ _id title author text createdAt updatedAt } }`. On the right, the JSON response is displayed: `{ "data": { "getPost": { "_id": "65b56f61b24f59b01c80e56e", "title": "title_new", "author": "author2_new", "text": "article2_new", "createdAt": "2024-01-13T18:46:41.224+00:00", "updatedAt": "2024-01-13T18:49:13.953+00:00" } } }`. The IDE includes buttons for 'Prettify', 'Merge', 'Copy', and 'History'.

Query для виведення завданого документа з БД



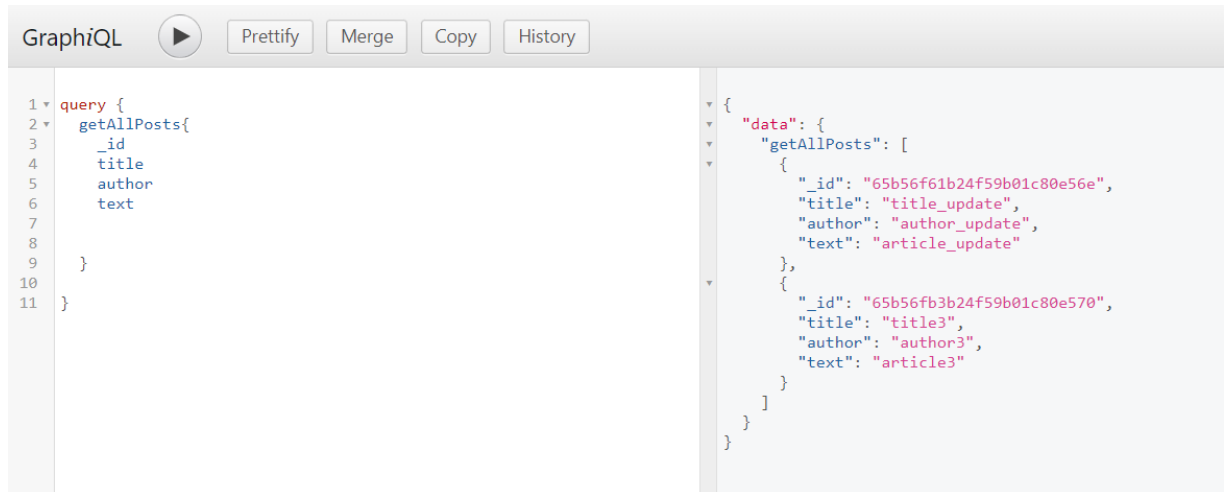
The screenshot shows the GraphQL IDE interface. On the left, a mutation is defined: `mutation { updatePost(_id: "65b56f61b24f59b01c80e56e", title: "title_update", author: "author_update", text: "article_update", createdAt: "2024-01-13T18:46:41.224+00:00", updatedAt: "2024-01-13T18:49:13.953+00:00") { _id title author text createdAt updatedAt } }`. On the right, the JSON response is displayed: `{ "data": { "updatePost": { "_id": "65b56f61b24f59b01c80e56e", "title": "title_update", "author": "author_update", "text": "article_update", "createdAt": "2024-01-13T18:46:41.224+00:00", "updatedAt": "2024-01-13T18:49:13.953+00:00" } } }`. The IDE includes buttons for 'Prettify', 'Merge', 'Copy', and 'History'.

Mutation для оновлення документа в БД



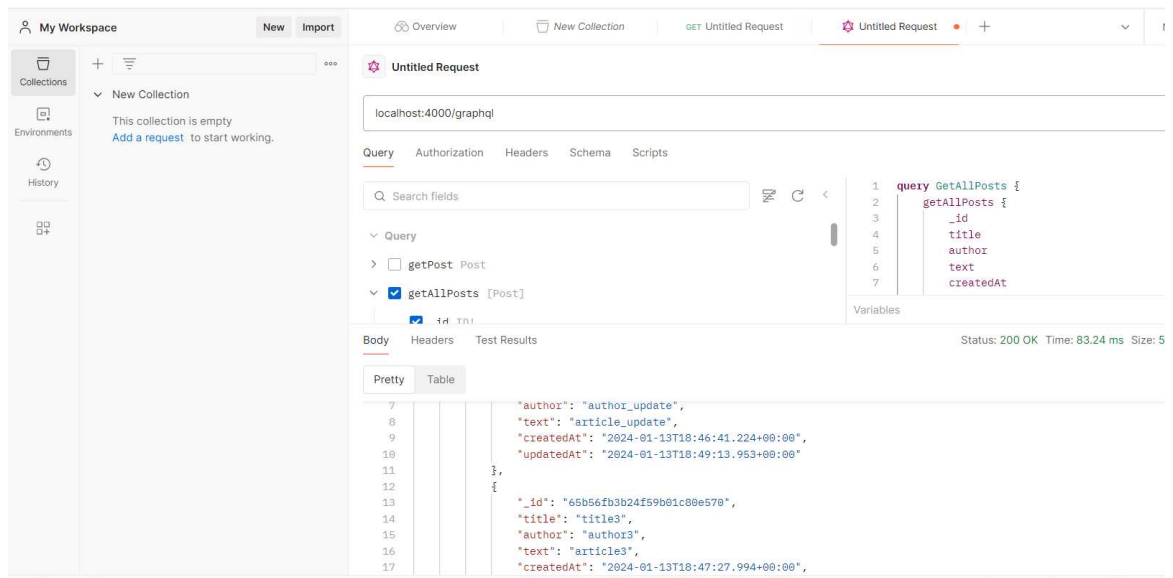
The screenshot shows the GraphQL IDE interface. On the left, a mutation is defined: `mutation { deletePost(_id: "65b6114368d29bf6de8a3d80") }`. On the right, the JSON response is displayed: `{ "data": { "deletePost": true } }`. The IDE includes buttons for 'Prettify', 'Merge', 'Copy', and 'History'.

Mutation для видалення документа з БД



Query для виведення всіх документів з БД після додавання, редагування та видалення документів

Дослідження роботи GraphQL за допомогою Postman



The screenshot shows the Postman interface for an 'Untitled Request'. The URL is 'localhost:4000/graphql'. The 'Query' tab is active, showing a search bar and a list of fields. The 'Mutation' section is expanded, and the 'createPost' mutation is selected. The input fields are: 'title' (String!) with value 'title_postman', 'author' (String!) with value 'author_postman', and 'text' (String!) with value 'article_postman'. The 'Variables' tab is empty. The 'Body' tab is active, showing the JSON response in 'Pretty' format. The response is a 200 OK status with a time of 45.95 ms. The JSON body is:

```
{
  "data": {
    "createPost": {
      "_id": "65b75651b856bda86b4b1de0",
      "title": "title_postman",
      "author": "author_postman",
      "text": "article_postman",
      "createdAt": "Mon Jan 29 2024 09:40:01 GMT+0200 (Восточная Европа, стандартное время)",
      "updatedAt": "Mon Jan 29 2024 09:40:01 GMT+0200 (Восточная Европа, стандартное время)"
    }
  }
}
```

Mutation createPost (Postman)

The screenshot shows the Postman interface for an 'Untitled Request'. The URL is 'localhost:4000/graphql'. The 'Query' tab is active, showing a search bar and a list of fields. The 'Mutation' section is expanded, and the 'updatePost' mutation is selected. The input fields are: '_id' (ID!) with value '65b75651b856bda86b4b1de', 'title' (String!) with value 'title_post_update', and 'author' (String!) with value 'author_post_update'. The 'Variables' tab is empty. The 'Body' tab is active, showing the JSON response in 'Pretty' format. The response is a 200 OK status with a time of 19.36 ms and a size of 51. The JSON body is:

```
{
  "data": {
    "updatePost": {
      "_id": "65b75651b856bda86b4b1de0",
      "title": "title_post_update",
      "author": "author_post_update",
      "text": "text_post_update",
      "createdAt": "Mon Jan 29 2024 09:40:01 GMT+0200 (Восточная Европа, стандартное время)",
      "updatedAt": "Mon Jan 29 2024 09:40:01 GMT+0200 (Восточная Европа, стандартное время)"
    }
  }
}
```

Mutation updatePost (Postman)

Додаткова інформація



GraphQL був розроблений Facebook у 2012 році для запуску мобільних додатків. З моменту появи специфікації GraphQL з відкритим вихідним кодом у 2015 році вона набула великої популярності і зараз використовується багатьма командами розробників, включаючи таких гігантів, як GitHub, Twitter або Airbnb.

Що таке GraphQL

GraphQL - це мова запитів для API, а також середовище для виконання цих запитів з існуючими даними. GraphQL надає повний та зрозумілий опис даних у твоєму API. Він дає клієнтам (браузерам) можливість запитати у сервера тільки саме те, що потрібно, і повертає лише запитані дані, і нічого більше.

Запит даних GraphQL vs REST

У типовій реалізації REST клієнт збирає дані шляхом доступу до кількох кінцевих точок; тобто. спочатку потрібно викликати кінцеву точку, щоб отримати вихідні дані користувача, а потім виконати окремий виклик, щоб отримати всі його властивості. GraphQL обробляє це інакше. Специфікація того, що може бути запитано, лежить на стороні клієнта, і при запиті конкретних даних, сервер GraphQL буде відповідати тільки запитаними даними.

Мова визначення схеми (SDL)

Схема GraphQL є ядром проекту GraphQL. Вкладений у сервер GraphQL схема визначає всі функції, доступні для клієнта GraphQL. Найбільш базовий елемент кожної схеми - це тип, який дозволяє встановлювати відносини між різними елементами схеми, визначати дозволені операції GraphQL на сервері та багато іншого. Щоб спростити розуміння операції, яку може виконувати сервер GraphQL визначає універсальний синтаксис схеми, відомий як мова визначення схеми (SDL). Основними елементами схеми GraphQL є типи об'єктів. Вони є об'єктом, який ми можемо отримати з нашого сервера GraphQL за допомогою доступних полів.

Наприклад:

Schema GraphQL

```
type Post {  
  _id: ID!  
  title: String!  
  author: String!  
  text: String!  
  createdAt: String  
  updatedAt: String  
}
```

Query - це базова операція fetchGraphQL для запиту даних із сервера GraphQL.

```
type Query {  
  getPost(_id: ID!): Post  
  getAllPosts: [Post]  
}
```

Mutation - це одна з основних операцій GraphQL, що дозволяє маніпулювати даними (створювати, змінювати чи видаляти):

```
type Mutation {  
  createPost(title: String!, author: String!, text:String!,  
    createdAt: String, updatedAt: String): Post  
  updatePost(_id: ID!, title: String!, author: String!,  
    text:String): Post  
  deletePost(_id: ID!): Boolean  
}
```

Resolvers

```
const resolvers = {  
  Query: {  
    // Implement your query resolvers here  
    getAllPosts: async () => {  
      try {  
        console.log("inside get allpost");  
        const result = await Post.find({});  
        if (!result.length) {  
          throw new Error("No Posts Added!");  
        }  
        return result;  
      } catch (error) {  
        throw error;  
      }  
    }  
  }  
}
```

```
    },
    getPost: async (parent, args) => {
      try {
        const result = await Post.findById(args._id);
        if (!result) {
          throw new Error("Post does not exists!");
        }
        return result;
      } catch (error) {
        throw error;
      }
    },
  },
  Mutation: {
    // Implement your mutation resolvers here
    createPost: async (parent, args) => {
      try {
        var now = new Date();
        const result = await Post.create({
          title: args.title,
          text: args.text,
          author: args.author,
          createdAt: now,
          updatedAt: now,
        });
        return result;
      } catch (error) {
        throw error;
      }
    },
    updatePost: async (parent, args) => {
      try {
        const result = await Post.findByIdAndUpdate(
          args._id,
          {
            title: args.title,
            author: args.author,
            text: args.text,
          },
          { new: true }
        );
        if (!result) {
          throw new Error("Post does not exists!");
        }
        return result;
      }
    }
  }
}
```

```
    } catch (error) {  
      throw error;  
    }  
  },  
  deletePost: async (_, args) => {  
    try {  
      const result = await Post.findByIdAndDelete(args._id);  
      if (!result) {  
        return false;  
      }  
      return true;  
    } catch (error) {  
      throw error;  
    }  
  },  
},  
};
```

Для новачків GraphQL є вбудований інтерактивний підручник. Натисніть [на це посилання](#) , і ти будеш перенаправлений на інтерактивний підручник GraphQL.

Якщо ти хочеш дізнатися більше про GraphQL, офіційний сайт [GraphQL](#) - те, що тобі потрібно.

Оформлення звіту та порядок захисту Лабораторна робота виконується на комп'ютері. Звіт оформлюється на аркушах А4 , в якому стисло відображається зміст, хід роботи та отримані результати. Робота захищається на парі (on-line), звіт надсилається на пошту.

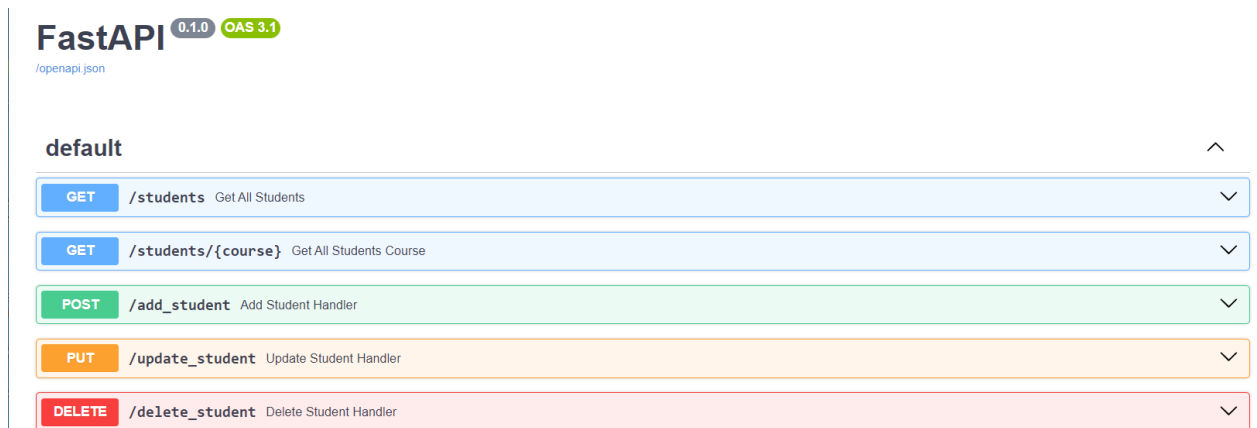
Лабораторна робота №6

Тема: Створення власного API на Python (FastAPI).

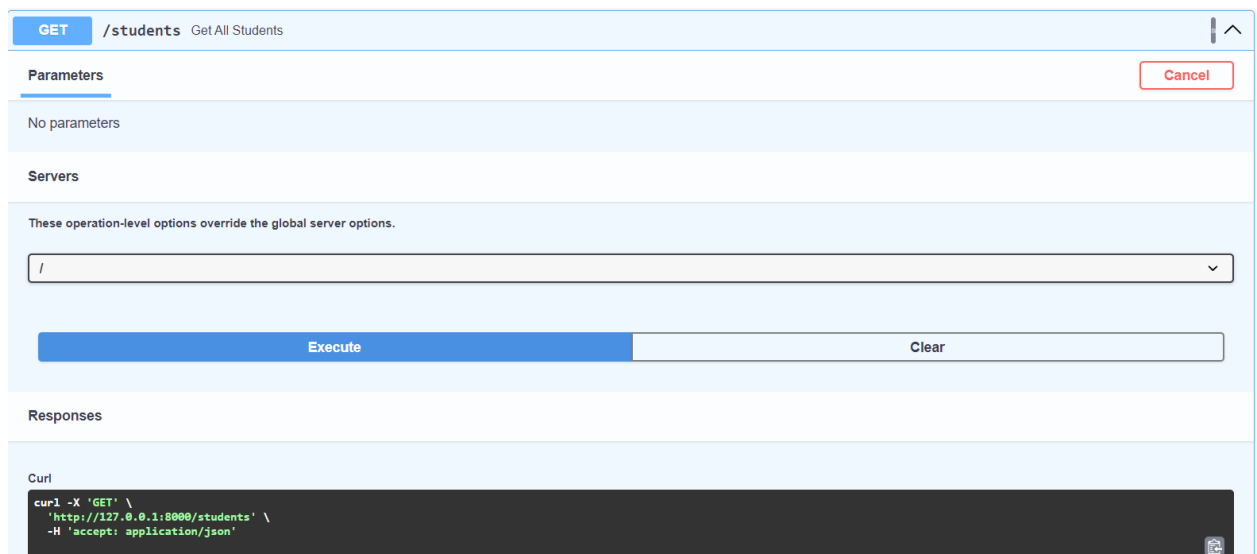
Завдання 1.

Створити web-додаток з використанням технології FastAPI, який буде працювати з інформацією відповідно вашому варіанту (Додаток 1. Таблиця 1)

Приклад виконання завдання



Виведення всіх



Curl

```
curl -X 'GET' \
  http://127.0.0.1:8000/students/ \
  -H 'accept: application/json'
```

Request URL

http://127.0.0.1:8000/students

Server response

Code Details

200

Response body

```
{
  "student_id": 1,
  "first_name": "Богдан",
  "last_name": "Рудь",
  "date_of_birth": "1998-05-15",
  "email": "bogdan.rud@example.com",
  "phone_number": "+380 (44) 456-7890",
  "address": "м. Київ, вул. Шевченко, 6. 10, кв. 5",
  "enrollment_year": 2017,
  "major": "Інформатика",
  "course": 3,
  "special_notes": "без особистих прикмет"
},
{
  "student_id": 2,
  "first_name": "Олена",
  "last_name": "Петренко",
  "date_of_birth": "1999-08-20",
  "email": "olena.petr@example.com",
  "phone_number": "+380 (44) 567-8901",
  "address": "м. Львів, вул. Стуса, 6. 5, кв. 8",
  "enrollment_year": 2018,
  "major": "Економіка",
  "course": 2,
  "special_notes": "без особистих прикмет"
}
```

Download

Виведення 2 курс

GET /students/{course} Get All Students Course

Parameters

Cancel

| Name | Description |
|--|--------------------------------|
| course * required integer (path) | <input type="text" value="2"/> |

Servers

These operation-level options override the global server options.

/

Execute Clear

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/students/2' \
  -H 'accept: application/json'
```

Request URL

http://127.0.0.1:8000/students/2

Server response

Code Details

200

Response body

```
[
  {
    "student_id": 2,
    "first_name": "Олена",
    "last_name": "Петренко",
    "date_of_birth": "1999-08-20",
    "email": "olena.petr@example.com",
    "phone_number": "+380 (44) 567-8901",
    "address": "м. Львів, вул. Стуса, 6. 5, кв. 8",
    "enrollment_year": 2018,
    "major": "Економіка",
    "course": 2,
    "special_notes": "Без особистих прикмет"
  }
]
```

Download

Додавання

POST /add_student Add Student Handler

Parameters

No parameters

Request body *required*

application/json

```
{
  "student_id": 3,
  "phone_number": "+380 (44) 567-8901",
  "first_name": "New",
  "last_name": "New",
  "date_of_birth": "2024-08-12",
  "email": "user@example.com",
  "address": "stringstri",
  "enrollment_year": 2002,
  "major": "Інформатика",
  "course": 1,
  "special_notes": "string"
}
```

Responses

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8000/add_student' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "student_id": 3,
    "phone_number": "+380 (44) 567-8901",
    "first_name": "New",
    "last_name": "New",
    "date_of_birth": "2024-08-12",
    "email": "user@example.com",
    "address": "stringstri",
    "enrollment_year": 2002,
    "major": "Інформатика",
    "course": 1,
    "special_notes": "string"
  }'
```

Request URL

http://127.0.0.1:8000/add_student

Server response

Code Details

200

Response body

```
{
  "message": "Студента успішно додано!"
}
```

Download

Редагування

PUT

/update_student Update Student Handler

Parameters

Cancel

Reset

No parameters

Request body required

application/json

```
{  "filter_student": {    "student_id": 3  },  "new_data": {    "course": 3,    "major": "Право"  }}
```

Responses

Curl

```
curl -X 'PUT' \  'http://127.0.0.1:8000/update_student' \  -H 'accept: application/json' \  -H 'Content-Type: application/json' \  -d '{  "filter_student": {    "student_id": 3  },  "new_data": {    "course": 3,    "major": "Право"  }  }'
```

Request URL

http://127.0.0.1:8000/update_student

Server response

Code

Details

200

Response body

```
{  "message": "Інформація про студента успішно оновлена!"}
```

Download

Видалення

DELETE

/delete_student Delete Student Handler

Parameters

Cancel

Reset

No parameters

Request body required

application/json

```
{  "key": "student_id",  "value": 3}
```

Servers



Додаткова інформація

Що таке FastAPI

FastAPI представляє швидкий високопродуктивний фреймворк для створення веб-застосунків мовою Python.

Офіційний сайт проекту: <https://fastapi.tiangolo.com/>. Вихідний код фреймворку доступний на github за адресою: <https://github.com/tiangolo/fastapi>

На даний момент підтримується Python версії 3.6 та вище.

Необхідні інструменти та встановлення

Для роботи з FastAPI природно знадобиться інтерпретатор Python. Як його встановити, можна прочитати тут: [Установка та перша програма на Python](#)

Для встановлення пакетів FastAPI вимагатиме пакетний менеджер **pip**. Менеджер **pip** дозволяє завантажувати пакети та керувати ними. Зазвичай при установці **python** також встановлюється менеджер **pip**.

Для встановлення пакетів FastAPI відкриємо термінал та введемо команду

```
pip install fastapi
```

```
Microsoft Windows [Version 10.0.22000.856]
(c) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

C:\Users\eugen>pip install fastapi
Collecting fastapi
  Downloading fastapi-0.81.0-py3-none-any.whl (54 kB)
    _____ 54.9/54.9 KB 480.3 kB/s eta 0:00:00
Collecting pydantic<=1.7, >=1.7.1, <=1.7.2, <=1.7.3, <=1.8, <=1.8.1, <2.0.0, >=
1.6.2
  Downloading pydantic-1.9.2-cp310-cp310-win_amd64.whl (2.0 MB)
    _____ 2.0/2.0 MB 3.4 MB/s eta 0:00:00
Collecting starlette==0.19.1
  Downloading starlette-0.19.1-py3-none-any.whl (63 kB)
    _____ 63.3/63.3 KB ? eta 0:00:00
```

акож для роботи з FastAPI нам знадобиться ASGI веб-сервер (веб-сервер з підтримкою протоколу Asynchronous Server Gateway Interface). Як таке в Python можна використовувати [Univer](#) або [Hypercorn](#). В даному випадку будемо використовувати Univer. Також встановимо його пакети за допомогою менеджера pip за допомогою наступної команди:

```
pip install "uvicorn[standard]"
```

Створення першої програми

Визначимо на диску папку, де розміщуватимуться файли з вихідним кодом програми. Наприклад, у моєму випадку це папка **C: fastapi**. Створимо в цій папці новий файл, який назвемо **main.py** і який матиме наступний код:

```
from fastapi import FastAPI

from fastapi.responses import HTMLResponse

app = FastAPI()

@app.get("/")

def read_root():

    html_content = "<h2>Hello KPI!</h2>"

    return HTMLResponse(content=html_content)
```

Для обробки запитів до програми спочатку необхідно створити об'єкт програми за допомогою конструктора **FastAPI** з пакета `fastapi`

```
app = FastAPI()
```

Потім визначаємо функцію, яка оброблятиме запити. До цієї функції застосовується спеціальний декоратор як метод **app.get()** :

```
@app.get("/")
```

У цей метод передається шаблон маршруту, яким функція оброблятиме запити. В даному випадку це рядок `"/"`, який означає, що функція оброблятиме запити по шляху `"/"`, тобто запити до кореня веб-додатку.

Після декоратора `app.get` йде власне визначення функції, яка обробляє запит:

```
def read_root():  
    html_content = "<h2>Hello KPI!</h2>"  
    return HTMLResponse(content=html_content)
```

Це звичайна функція python. Вона називається `read_root` (довільне ім'я). Для надсилання відповіді вона використовує клас **HTMLResponse** з пакета `fastapi.responses`. Клас `HTMLResponse` дозволяє відправити у відповідь деякий вміст у вигляді коду HTML.

Для встановлення вмісту, що відправляється в конструкторі `HTMLResponse` застосовується параметр `content`, якому в даному випадку передається рядок `"<h2>Hello KPI!</h2>"` зі значенням `"Hello KPI!"`. Тобто коли клієнт звернеться до веб-додатку шляхом `"/"`, йому буде відправлено html-код `"<h2>Hello KPI!</h2>"`.

Запуск програми

Тепер запусимо програму. Для цього перейдемо в терміналі до папки, де має файл **main.py** і потім виконаємо команду

```
uvicorn main:app --reload
```

В даному випадку ми запускаємо сервер **uvicorn** та передаємо йому ряд параметрів:

- `Main` вказує на назву модуля, яке за умовчанням збігається з назвою файлу - `main`
- `app` вказує на об'єкт програми, створений у рядку `app = FastAPI()`
- `--reload` дозволяє відстежувати зміни у файлах вихідного коду та автоматично перезапускати проект

Клас FastAPI та обробка запиту

У центрі програми FastAPI знаходиться однойменний клас **FastAPI** з пакета `fastapi`. Цей клас фактично і представляє програму FastAPI. Цей клас успадковується від класу `starlette.applications.Starlette` [Starlette](#) представляє інший легковажний ASGI-фреймворк для створення асинхронних веб-сервісів на Python. Власне fastAPI працює поверх Scarlette, використовуючи та доповнюючи його функціональність. Це стосується не тільки самого класу FastAPI, а й інших класів фреймворку - багато з них використовують функціонал Scarlette.

Конструктор класу FastAPI має близько трьох десятків різних параметрів, які дозволяють налаштувати роботу програми. Але в загальному випадку для створення об'єкта класу, що функціонує, можна не передавати в конструктор жодних аргументів, тоді параметри отримують значення за замовчуванням:

```
from fastapi import FastAPI
```

```
app = FastAPI()
```

Методи FastAPI

Однією з переваг FastAPI є те, що фреймворк дозволяє швидко та легко побудувати веб-сервіс у стилі REST. Архітектура REST передбачає застосування наступних методів або типів запитів HTTP для взаємодії з сервером, де кожен тип запиту відповідає за певну дію:

- **GET** (отримання даних)
- **POST** (додавання даних)
- **PUT** (зміна даних)
- **DELETE** (видалення даних)

Крім цих типів запитів HTTP підтримує ще ряд, зокрема:

- **OPTIONS**
- **HEAD**
- **PATCH**
- **TRACE**

У класі FastAPI для кожного з цих типів запитів визначено однойменні методи:

- **get()**
- **post()**
- **put()**
- **delete()**
- **options()**
- **head()**
- **patch()**
- **trace()**

Наприклад, якщо нам потрібно обробити HTTP-запит типу GET, то застосовується метод `get()`.

Всі ці методи мають безліч параметрів, але вони як обов'язковий параметр приймають шлях, запит яким повинен оброблятися.

Причому ці методи сам запит не обробляють - вони застосовують як декоратор до функцій, які безпосередньо обробляють запит. Наприклад:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
```

```
def root():  
  
    return {"message": "Hello KPI"}
```

В даному випадку метод `app.get()` застосовується як декоратор до функції `root()` (символ `@` вказує на визначення декоратора). Цей декоратор визначає шлях, запити яким оброблятиме функція `root()`. В даному випадку шлях представляє рядок `/`, тобто функція оброблятиме запити до кореня веб-додатку (наприклад, за адресою `http://127.0.0.1:8000/`).

Функція повертає деякі результати. Зазвичай це словник (об'єкт `dict`). Тут словник містить один елемент `"message"`. При надсиланні ці дані автоматично серіалізуються у формат JSON – популярний формат для взаємодії між клієнтом та сервером. А у відповіді для заголовка `content-type` встановлюється значення `application/json`. Взагалі функція може повертати різні дані – словники (`dict`), списки (`list`), одиночні значення типу рядків, чисел тощо, які потім серіалізуються у `json`.

Відповідно, якщо ми запустимо програму і звернемося за адресою `http://127.0.0.1:8000/`, наприклад, у браузері, то ми отримаємо відповідь сервера у форматі `json`.

Подібним чином можна визначати інші функції, які будуть обробляти запити по інших шляхах. Наприклад:

```
from fastapi import FastAPI  
  
app = FastAPI()  
  
@app.get("/")  
  
def root():  
  
    return {"message": "Hello KPI"}  
  
@app.get("/about")  
  
def about():  
  
    return {"message": "Про сайт"}
```


Тут додано функцію `about ()` , яка обробляє запити на шляху `"/about"`.

Оформлення звіту та порядок захисту Лабораторна робота виконується на комп'ютері. Звіт оформлюється на аркушах А4 , в якому стисло відображається зміст, хід роботи та отримані результати. Робота захищається на парі (on-line), звіт надсилається на пошту.

Додаток 1.

Таблиця 1.

Варіанти завдань

Варіант 1. Спроекувати базу даних про студентів для їх розподілу по місцях практики: прізвище, рік народження, стать, група, факультет, середній бал, місце роботи, місто.

Варіант 2. Спроекувати базу даних про автомобілі: номер, рік випуску, марка, колір, стан, прізвище власника, адреса.

Варіант 3. Спроекувати базу даних про квартири, призначених для продажу: район, поверх, площа, кількість кімнат, відомості про власника, ціна.

Варіант 4. Спроекувати базу даних про книги, куплених бібліотекою: назва, автор, рік видання, адреса автора, адреса видавництва, ціна, книго-торговельна фірма.

Варіант 5. Спроекувати базу даних про співробітників, що мають комп'ютер: прізвище, номер кімнати, назва відділу, дані про комп'ютер.

Варіант 6. Спроекувати базу даних про замовлення, що отримані співробітниками фірми: прізвище, сума замовлення, найменування товару, назва фірми-клієнта, прізвище замовника.

Варіант 7. Спроекувати базу даних про оцінки, отриманих студентами на іспитах: прізвище, група, предмет, номер квитка, оцінка, викладач.

Варіант 8. Спроекувати базу даних про викладачів кафедри: прізвище, посада, ступінь, номер аудиторії де читаються курси.

Варіант 9. Спроекувати базу даних про авторів web-сайту і їх статтях: ім'я, адреса, обліковий запис, пароль, тема, заголовок, текст статті, ілюстрації.

Варіант 10. Спроекувати базу даних про список розсилки і передплатників: тема і зміст листа, дата відправки, імена та адреси передплатників, їх облікові записи і паролі.

Варіант 11. Спроекувати базу даних технологічних карт: деталь, вид обробки, тривалість обробки.

Варіант 12. Спроекувати базу даних про документи, переданих виконавцям: виконавець, документ, дата передачі, дата повернення.

Варіант 13. Спроекувати базу даних про викрадені автомобілі: номер, марка, стан (викрадений / знайдений), прізвище власника.

Варіант 14. Спроекувати базу даних оголошень про квартири: вид оголошення (здам / продам / зніму / куплю), адреса, кількість кімнат, дата, ціна.

Варіант 15. Спроекувати базу даних о книгах, виданих в бібліотеці: назва, читач, дата видачі, дата повернення.

Варіант 16. Спроекувати базу даних про договори: назва фірми-клієнта, вид договору, термін дії.

Варіант 17. Спроекувати базу даних про видачу матеріальних цінностей співробітникам: прізвище, номер кімнати, виріб, дата видачі.

Варіант 18. Спроекувати базу даних про галерею зображень: автор, назва файлу, дата розміщення.

Варіант 19. Спроекувати базу даних про турпутівки: країна, курорт, вартість, дата виїзду.

Варіант 20. Спроекувати базу даних про маршрути транспорту: вид транспорту, номер, маршрут (початкова та кінцева зупинки), час у дорозі.