Національний університет біоресурсів і
природокористування України

Інтеграція Python-додатків із зовнішніми сервісами за допомогою API

Лектор: к.ф.-м.н., доцент Кириченко Віктор Вікторович

Огляд лекції

- Поняття API: визначення, типи та принципи взаємодії
- Інструменти Python для роботи з API
- Аутентифікація та авторизація
- Обмеження та ліміти запитів
- Приклади інтеграції з популярними API
- Практична робота: написання Python-скрипту

Мета лекції: навчитись інтегрувати Python-додатки з зовнішніми сервісами за допомогою API

Що таке API?

Визначення

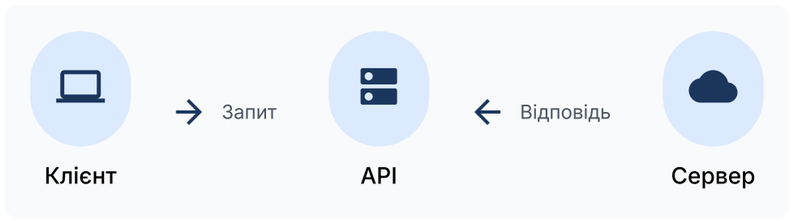
API (Application Programming Interface) — набір визначень та протоколів для створення та інтеграції програмного забезпечення, що дозволяє додаткам обмінюватися даними та функціональністю.

Клієнт-серверна модель

- Клієнт надсилає запит
- Сервер обробляє запит
- Сервер повертає відповідь

Основні принципи

- Абстракція реалізації
- Стандартизація взаємодії
- Інкапсуляція функціональності



Типи API

HTTP REST API

- Використовує HTTP методи
- Станless архітектура
- Формати: JSON, XML
- Простий та гнучкий

SOAP API

- Протокол з жорсткими правилами
- Використовує XML формат
- Підтримує ACID транзакції
- Високий рівень безпеки

GraphQL

- Запит на конкретні дані
- Єдина точка входу (endpoint)
- Сильна типізація
- Уникнення over-fetching

Характеристика	REST	SOAP	GraphQL
Складність	Низька	Висока	Середня
Продуктивність	Середня	Низька	Висока
Гнучкість	Висока	Низька	Дуже висока

Важливо: Вибір типу API залежить від конкретних вимог проєкту, рівня безпеки та необхідної гнучкості.

Принципи взаємодії з API

Клієнт-серверна архітектура

Розділення відповідальності між клієнтом та сервером

- Клієнт: інтерфейс користувача
- Сервер: дані та бізнес-логіка

Безстанність (Stateless)

Кожен запит містить усю необхідну інформацію

- Сервер не зберігає стан клієнта
- Покращує масштабованість

Кешування

Відповіді можуть бути кешовані клієнтом

- Зменшує навантаження на сервер
- Прискорює роботу клієнта

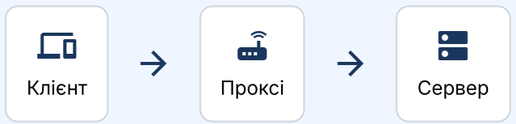
Єдиний інтерфейс

Стандартизований спосіб взаємодії

- Ідентифікація ресурсів (URI)
- Маніпуляція ресурсами через представлення

Багатошаровість (Layered System)

Архітектура може складатися з кількох шарів



Найкращі практики

- Використовувати HTTP методи коректно
- Використовувати версіонування API
- Повертяти відповідні HTTP статуси
- Надавати чітку документацію

Інструменти Python для роботи з API

HTTP Requests

Найпопулярніша бібліотека для HTTP-запитів

- Простий та інтуїтивний API
- Синхронні запити
- Вбудована сесія та cookies

```
import requests
# Простий GET-запит
response =
requests.get('https://api.example.com/data')
```

HTTPX

Сучасний HTTP-клієнт для Python

- Підтримка HTTP/2
- Синхронні та асинхронні запити
- Сумісний з Requests API

```
import httpx
# Асинхронний запит
async with httpx.AsyncClient() as
client:
  response = await client.get(url)
```

AIOHTTP

Асинхронний HTTP клієнт/сервер

- Висока продуктивність
- Побудований на asyncio
- Підтримка WebSockets

```
import aiohttp
# Асинхронний GET-запит
async with aiohttp.ClientSession()
as session:
  async with session.get(url) as
response:
```

Порівняння інструментів

Requests	HTTPX	AIOHTTP
✓ Простота	✓ Гнучкість	✓ Продуктивність
− Асинхронність	✓ HTTP/2	✓ WebSocket

Рекомендація

Для початківців: Requests. Для асинхронних завдань: HTTPX або AIOHTTP. Для високонавантажених систем: AIOHTTP.

Робота з JSON та XML у Python

{} JSON

JavaScript Object Notation — текстовий формат обміну даними

```python
import json

# Перетворення Python-об'єкта в JSON
data = {
  "name": "John",
  "age": 30
}
json_str = json.dumps(data)

# Перетворення JSON в Python-об'єкт
parsed_data = json.loads(json_str)
```

- Легко читається людьми
- Підтримує базові типи даних
- Вбудована підтримка в Python

XML

eXtensible Markup Language — розширювана мова розмітки

```python
import xml.etree.ElementTree as ET

# Парсинг XML з рядка
xml_str = """<person><name>John</name><age>30</age></person>"""
root = ET.fromstring(xml_str)

# Отримання даних
name = root.find('name').text
age = root.find('age').text
```

- Підтримує складні структури даних
- Можливість визначення схеми (DTD, XSD)
- Кілька бібліотек для роботи в Python

Додаткові бібліотеки

Для JSON:

- **orjson** — швидкий парсер JSON
- **ujson** — оптимізований парсер

Для XML:

- **lxml** — потужний парсер XML
- **xmltodict** — конвертація XML в dict

Поради щодо вибору формату

- Використовуйте JSON для простих даних та веб-сервісів
- Використовуйте XML для складних структур та корпоративних систем
- Завжди перевіряйте дані після парсингу

Методи автентифікації API

API ключі

Унікальний ідентифікатор для доступу до API

```
# Використання API ключа у запиті
import requests

headers = {
  'Authorization': 'Bearer YOUR_API_KEY'
}
response = requests.get(
  'https://api.example.com/data',
  headers=headers
)
```

- Простий у реалізації
- Передається у заголовках або параметрах
- Не містить інформації про користувача

Токени доступу

Тимчасові облікові дані для автентифікації

```
# Використання токена доступу
import requests

access_token = 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...'
headers = {
  'Authorization': 'Bearer ' + access_token
}
response = requests.get(url, headers=headers)
```

- Має обмежений термін дії
- Може містити додаткові дані
- Підтримує оновлення (refresh)

Найкращі практики безпеки

- Зберігайте ключі та токени у змінних середовища
- Використовуйте HTTPS для передачі
- Обмежуйте права доступу токенів
- Регулярно оновлюйте ключі безпеки
- Не додавайте ключі до системи контролю версій
- Використовуйте бібліотеки для роботи з токенами

Структура JWT токена

JWT (JSON Web Token) складається з трьох частин: заголовок (Header), корисне навантаження (Payload) та підпис (Signature).

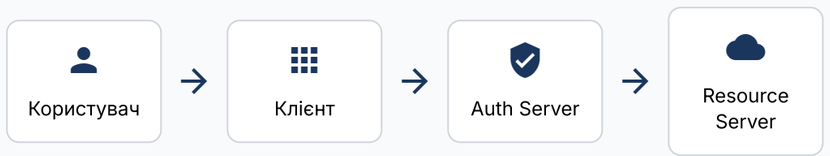
```
header.payload.signature
```

OAuth 2.0

Що таке OAuth 2.0?

Протокол авторизації, який дозволяє додаткам отримувати обмежений доступ до облікових записів користувачів на сервісі HTTP, не передаючи паролі.

Процес авторизації (Authorization Flow)



1. Користувач надає згоду на доступ
2. Клієнт отримує authorization code
3. Клієнт обмінює code на access token
4. Клієнт використовує token для доступу до ресурсів

Типи надання доступу (Grant Types)

- **Authorization Code** — для веб-додатків
- **Implicit** — для SPA та мобільних додатків
- **Resource Owner Password** — для довірених додатків
- **Client Credentials** — для сервер-сервер взаємодії

Популярні бібліотеки для OAuth 2.0 в Python

requests-oauthlib | authlib | python-social-auth

Реалізація в Python

```python
import requests
from requests_oauthlib import OAuth2Session

# Налаштування OAuth2
client_id = 'your_client_id'
client_secret = 'your_client_secret'
authorization_base_url = 'https://example.com/oauth/authorize'
token_url = 'https://example.com/oauth/token'

# Створення сесії OAuth2
oauth = OAuth2Session(client_id)
```

Обмеження та ліміти запитів

Що таке rate limiting?

Механізм контролю кількості запитів, які клієнт може надіслати до API за певний проміжок часу.

Типи обмежень

- **Requests per time window** — напр. 1000 запитів на годину
- **Concurrent requests** — одночасні запити
- **Burst limits** — короткочасні сплески запитів
- **Quotas** — загальна кількість запитів

Навіщо потрібні ліміти?

- Запобігання перевантаженню сервера
- Захист від DDoS-атак
- Справедливий розподіл ресурсів
- Монетизація API (пла тарифікація)

Обробка лімітів у Python

```python
import requests
import time
from requests.adapters import HTTPAdapter
from urllib3.util.retry import Retry

# Налаштування повторних спроб
retry_strategy = Retry(
  total=3,
  backoff_factor=1,
  status_forcelist=[429, 500, 502, 503, 504]
)

# Створення сесії з повторними спробами
adapter = HTTPAdapter(max_retries=retry_strategy)
session = requests.Session()
session.mount("https://", adapter)
session.mount("http://", adapter)
```

Стратегії обробки

- Exponential backoff
- Черга запитів

Індикатори лімітів

- HTTP 429 (Too Many Requests)
- Заголовки (X-RateLimit-*)

Корисні бібліотеки

ratelimit · limits · aiolimiter

Обробка помилок при роботі з API

Поширені HTTP статуси помилок

4xx: Помилки клієнта

- **400** Bad Request
- **401** Unauthorized
- **403** Forbidden
- **404** Not Found
- **429** Too Many Requests

5xx: Помилки сервера

- **500** Internal Server Error
- **502** Bad Gateway
- **503** Service Unavailable
- **504** Gateway Timeout

Успішні відповіді

- **200** OK
- **201** Created
- **204** No Content

Обробка винятків у Python

```python
import requests
from requests.exceptions import RequestException

try:
  response = requests.get(
    'https://api.example.com/data'
  )
  response.raise_for_status()
  data = response.json()
except requests.exceptions.HTTPError as errh:
  print(f"HTTP Error: {errh}")
except requests.exceptions.ConnectionError as errc:
  print(f"Connection Error: {errc}")
except requests.exceptions.Timeout as errt:
  print(f"Timeout Error: {errt}")
except requests.exceptions.RequestException as err:
  print(f"Request Error: {err}")
```

Найкращі практики

- Використовуйте **raise_for_status()** для перевірки HTTP статусів
- Обробляйте різні типи винятків окремо
- Встановлюйте таймаути для запитів
- Логуйте помилки для подальшого аналізу
- Реалізуйте механізм повторних спроб
- Перевіряйте наявність необхідних полів у відповіді

Стратегії обробки помилок

Retry with exponential backoff

Збільшення інтервалу між повторними спробами

Circuit breaker pattern

Вимкнення запитів при частих помилках

Graceful degradation

Надання альтернативного функціоналу

Fallback to cache

Використання кешованих даних при помилці

Приклад інтеграції з GitHub API

GitHub API

RESTful API для доступу до даних GitHub: репозиторіїв, користувачів, issues, pull requests тощо.

Основні endpoints

- **/user** — інформація про поточного користувача
- **/users/{username}** — інформація про користувача
- **/repos** — репозиторії користувача
- **/repos/{owner}/{repo}** — інформація про репозиторій
- **/search/repositories** — пошук репозиторіїв

Автентифікація

- **Personal Access Token** — для приватних даних
- **OAuth Apps** — для сторонніх додатків
- **GitHub Apps** — для інтеграцій з організаціями

```
headers = {
  'Authorization': 'token YOUR_TOKEN',
  'Accept': 'application/vnd.github.v3+json'
}
```

Приклад інтеграції з GitHub API

Приклад коду на Python

```python
import requests
import os

# Налаштування автентифікації
GITHUB_TOKEN = os.environ.get('GITHUB_TOKEN')

headers = {
  'Authorization': f'token {GITHUB_TOKEN}',
  'Accept': 'application/vnd.github.v3+json'
}

# Отримання інформації про користувача
def get_user_info(username):
  url = f'https://api.github.com/users/{username}'
  response = requests.get(url, headers=headers)
  if response.status_code == 200:
    return response.json()
  return None

# Отримання репозиторіїв користувача
def get_user_repos(username):
  url = f'https://api.github.com/users/{username}/repos'
  response = requests.get(url, headers=headers)
  if response.status_code == 200:
    return response.json()
  return None
```

Приклад використання

```python
# Використання функцій
user_info = get_user_info('octocat')
if user_info:
  print(f"Користувач: {user_info['name']}")
  print(f"Публічні репозиторії: {user_info['public_repos']}")

repos = get_user_repos('octocat')
if repos:
  print(f"Перший репозиторій: {repos[0]['name']}")
```

Приклад інтеграції з OpenWeather API

OpenWeather API

API для отримання погодних даних: поточна погода, прогнози, історичні дані, карти погоди тощо.

Основні endpoints

- /weather — поточна погода
- /forecast — 5-денний прогноз
- /onecall — детальний прогноз
- /history — історичні дані
- /air_pollution — якість повітря

Автентифікація

- API Key — унікальний ключ доступу
- Безкоштовний тариф — 60 запитів/хвилину
- Платні тарифи — більше запитів та даних

```
# Запит з API ключем
url = 'https://api.openweathermap.org/data/2.5/weather'
params = {
  'q': 'Kyiv',
  'appid': 'YOUR_API_KEY',
  'units': 'metric'
}
```

Приклад інтеграції з OpenWeather API

Приклад коду на Python

```python
import requests
import os

# Налаштування API
API_KEY = os.environ.get('OPENWEATHER_API_KEY')
BASE_URL = 'https://api.openweathermap.org/data/2.5'

# Отримання поточної погоди
def get_current_weather(city):
  url = f"{BASE_URL}/weather"
  params = {
    'q': city,
    'appid': API_KEY,
    'units': 'metric'
  }
  response = requests.get(url, params=params)
  if response.status_code == 200:
    return response.json()
  return None

# Отримання прогнозу погоди
def get_weather_forecast(city):
  url = f"{BASE_URL}/forecast"
  params = {
    'q': city,
    'appid': API_KEY,
    'units': 'metric'
  }
  response = requests.get(url, params=params)
  if response.status_code == 200:
    return response.json()
  return None
```

Приклад використання

```python
# Використання функцій
weather = get_current_weather('Kyiv')
if weather:
  temp = weather['main']['temp']
  desc = weather['weather'][0]
['description']
  print(f"Температура: {temp}°C")
  print(f"Опис: {desc}")

forecast =
get_weather_forecast('Kyiv')
if forecast:
  print(f"Прогноз на
{len(forecast['list'])} періодів")
```

Приклад інтеграції з Google Maps API

Google Maps API

Набір API для роботи з картами, геокодуванням, маршрутами, місцями та іншими географічними даними.

Основні сервіси

- **Geocoding API** — перетворення адрес у координати
- **Places API** — пошук місць за назвою
- **Directions API** — побудова маршрутів
- **Distance Matrix API** — відстані між точками
- **Static Maps API** — статичні зображення карт

Автентифікація

- **API Key** — унікальний ключ доступу
- **Безкоштовний тариф** — $200 кредит на місяць
- **Обмеження запитів** — залежить від сервісу

```
# Запит з API ключем
url = 'https://maps.googleapis.com/maps/api/geocode/json'
params = {
  'address': 'Київ, Хрещатик',
  'key': 'YOUR_API_KEY'
}
```

Приклад інтеграції з Google Maps API

Приклад коду на Python

```python
import requests
import os

# Налаштування API
API_KEY = os.environ.get('GOOGLE_MAPS_API_KEY')
BASE_URL = 'https://maps.googleapis.com/maps/api'

# Геокодування адреси
def geocode_address(address):
  url = f"{BASE_URL}/geocode/json"
  params = {
    'address': address,
    'key': API_KEY
  }
  response = requests.get(url, params=params)
  if response.status_code == 200:
    data = response.json()
    if data['status'] == 'OK':
      return data['results'][0]['geometry']['location']
  return None

# Побудова маршруту
def get_directions(origin, destination):
  url = f"{BASE_URL}/directions/json"
  params = {
    'origin': origin,
    'destination': destination,
    'key': API_KEY
  }
  response = requests.get(url, params=params)
  if response.status_code == 200:
    return response.json()
  return None
```

Приклад використання

```python
# Використання функцій
location = geocode_address('Київ, Хрещатик, 22')
if location:
  lat = location['lat']
  lng = location['lng']
  print(f"Координати: {lat}, {lng}")

directions = get_directions('Київ', 'Львів')
if directions and directions['status'] == 'OK':
  route = directions['routes'][0]['legs'][0]
  distance = route['distance']['text']
  duration = route['duration']['text']
  print(f"Відстань: {distance}")
  print(f"Час: {duration}")
```

Практична робота: створення Python-скрипту для API

Покрокова інструкція

1 Налаштування середовища

Встановлення необхідних бібліотек: requests, python-dotenv

```
pip install requests python-dotenv
```

2 Отримання API ключа

Реєстрація на OpenWeatherMap та отримання безкоштовного API ключа

3 Створення файлу .env

Збереження API ключа у файлі .env для безпеки

```
OPENWEATHER_API_KEY=your_api_key_here
```

4 Створення класу для роботи з API

Реалізація методів для отримання погодних даних

```
class WeatherAPI:
  def __init__(self, api_key):
    self.api_key = api_key
    self.base_url = "https://api.openweathermap.org/data/2.5"
```

5 Реалізація методів запитів

Створення методів для отримання поточної погоди та прогнозу

```
def get_current_weather(self, city):
  url = f"{self.base_url}/weather"
  params = {
    'q': city,
    'appid': self.api_key,
    'units': 'metric'
  }
  response = requests.get(url, params=params)
  return response.json()
```

6 Обробка даних

Створення функцій для аналізу отриманих погодних даних

```
def analyze_weather(weather_data):
  temp = weather_data['main']['temp']
  humidity = weather_data['main']['humidity']
  desc = weather_data['weather'][0]['description']
  return {'temperature': temp, 'humidity': humidity, 'description': desc}
```

7 Створення основного скрипту

Написання основного скрипту для демонстрації роботи

```
if __name__ == "__main__":
  api = WeatherAPI(api_key)
  weather = api.get_current_weather("Kyiv")
  analysis = analyze_weather(weather)
  print(f"Температура: {analysis['temperature']}°C")
```

Додаткові можливості для розширення

- Збереження даних у файл
- Візуалізація даних
- Обробка помилок
- Логування запитів

Висновки

API — ключовий інструмент інтеграції

API забезпечують стандартизований обмін даними між програмними системами, що дозволяє розширювати функціональність додатків.

Різноманітність типів API

REST, SOAP та GraphQL пропонують різні підходи до взаємодії з API, кожен з яких має свої переваги та сфери застосування.

Python пропонує потужні інструменти

Бібліотеки requests, httpx та aiohttp надають зручні засоби для роботи з API, що значно спрощує процес інтеграції.

Безпека — критичний аспект

API ключі, токени та OAuth 2.0 забезпечують безпечний доступ до зовнішніх сервісів, захищаючи дані користувачів.

Обмеження та помилки потребують обробки

Розуміння лімітів запитів та правильна обробка помилок є необхідними для створення надійних додатків.

Практичний досвід є важливим

Робота з реальними API, такими як GitHub, OpenWeather та Google Maps, дозволяє отримати практичні навички інтеграції.

Головний висновок

Інтеграція Python-додатків із зовнішніми сервісами за допомогою API є важливою навичкою для сучасного розробника, що дозволяє створювати потужні та функціональні програмні рішення.