Міністерство освіти і науки України
Національний технічний університет України “Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №9
З дисципліни «Технології розроблення програмного забезпечення»
Тема: **«РІЗНІ ВИДИ ВЗАЄМОДІЇ ДОДАТКІВ: CLIENT-SERVER, PEER-TO-PEER, SERVICE-ORIENTED ARCHITECTURE »**
Flexible Automatical Tool

Виконав:
Студент групи ІА-22
Сидорін Д.О.

Перевірив:
Мягкий М. Ю.

Київ-2024

Тема:
РІЗНІ ВИДИ ВЗАЄМОДІЇ ДОДАТКІВ: CLIENT-SERVER, PEER-TO-PEER, SERVICE-ORIENTED ARCHITECTURE

Мета:
Ознайомитися з основними принципами та шаблонами взаємодії між додатками, такими як «Client-Server», «Peer-to-Peer» та «Service-Oriented Architecture», дослідити їхні особливості та способи реалізації для створення гнучкого та масштабованого програмного забезпечення.

Варіант

Інструмент автоматизації (стратегія, прототип, абстрактна фабрика, міст, композит, SOA)
Десктопний додаток для автоматизації повсякденних завдань із можливістю створення правил (аналогічно сервісу IFTTT), запису макросів (натискання клавіш, дії миші) та використання планувальника завдань. Додаток забезпечує функції, як-от автоматичне завантаження нових серій серіалів, книг чи інших файлів у визначений час, зміну статусів у месенджерах (наприклад, встановлення статусу "відсутній" у Skype при довгій неактивності), а також виконання завдань за розкладом (наприклад, запуск роздачі торрентів о 5 ранку).

Хід роботи

1. Реалізувати функціонал для роботи в розподіленому оточенні (логіку роботи)

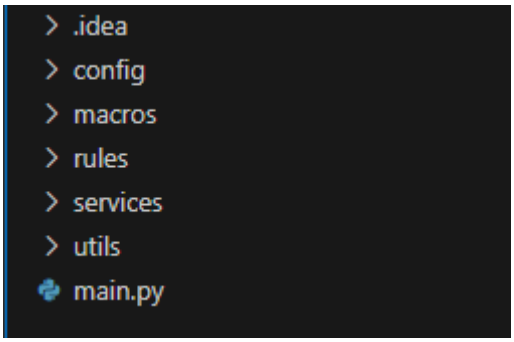


Рис. 1 — Загальна структура проекту

1. **config/**
   Містить конфігураційні файли програми:
   - **config.json**: Загальні налаштування для програми.

- **service_config.json**: Налаштування для кожного сервісу, включаючи параметри для специфічних задач, таких як завантаження, оновлення статусу, планування задач.

2. **services/**
   Визначає окремі сервіси, що виконують різні завдання:
   - **download_service/**: Відповідає за завантаження файлів. Може включати як скачування нових файлів (фільмів, книг), так і роботу з торрентами.
   - **status_service/**: Використовується для оновлення статусу в комунікаторах чи месенджерах, наприклад, зміна статусу на "відсутній" при нульовій активності.
   - **task_service/**: Відповідає за планування задач, наприклад, запуск певних процесів або автоматичне виконання дій о визначений час.
3. **rules/**
   Логіка обробки правил для автоматизації:
   - **rule_manager.py**: Клас для обробки та виконання визначених правил, що можуть бути використані для автоматизації.
   - **rule_parser.py**: Аналізатор правил, що дозволяє системі зрозуміти та інтерпретувати умови, зазначені користувачем (наприклад, IFTTT-стиль).
4. **macros/**
   Модуль для запису та виконання макросів:
   - **macro_recorder.py**: Логіка запису макросів, які можуть містити комбінації клавіш або дії миші.
   - **macro_executor.py**: Логіка виконання записаних макросів за певними командами або умовами.
5. **utils/**
   Допоміжні утиліти, які забезпечують загальну підтримку програмі:
   - **logger.py**: Логування для моніторингу подій та помилок.
   - **notifier.py**: Відповідає за сповіщення користувача про важливі події, наприклад, про завершення завдання або виникнення помилки.
6. **main.py**
   Основний файл, який запускає програму. Тут ініціалізуються сервіси та їх взаємодія між собою.
7. **requirements.txt**
   Список бібліотек та залежностей, необхідних для роботи програми.
8. **README.md**
   Документація, яка описує, як користуватися програмою, як налаштовувати сервіси та запускати їх.

Принципи роботи:

- Кожен сервіс є незалежним і виконує чітко визначену задачу, що забезпечує високу гнучкість і можливість масштабування.
- Логіка обробки правил дозволяє створювати умови для автоматичних дій, таких як завантаження файлів або оновлення статусу.
- Макроси дозволяють автоматизувати дії користувача, зокрема комбінації клавіш і рухи миші.
- Утиліти допомагають в управлінні помилками і сповіщеннях, а також для логування важливих подій.

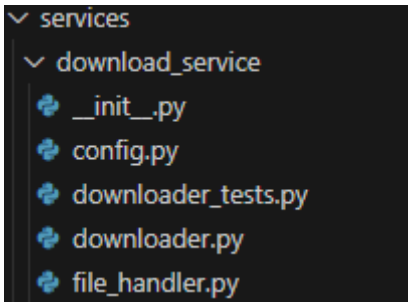


Рис. 2 — Структура сервісу карт пам’яті

 __init__.py:
Ініціалізація пакету download_service. Тут можна вказати, які класи чи функції будуть доступні для імпорту при використанні сервісу. Може містити базові налаштування для сервісу.

 downloader.py:
Основна логіка завантаження файлів. Це може включати взаємодію з торрентами, API для завантаження контенту (наприклад, через бібліотеки як requests або спеціалізовані для торрентів).
Приклад функціоналу:

- Отримання URL файлів для завантаження.
- Початок завантаження.
- Перевірка наявності файлів.
- Завершення завантаження з відправкою повідомлення про успіх або помилку.

 config.py:
Містить конфігураційні параметри для завантаження:

- Списки URL-адрес для завантаження.
- Таймаути для з'єднань.
- Логін/пароль для доступу до обмежених ресурсів.
- Інші параметри, специфічні для конкретного джерела файлів (наприклад, сервери торрентів).

 file_handler.py:
Служить для управління файлами, з якими працює сервіс. Це можуть бути функції для:

- Перевірки наявності файлу перед завантаженням.
- Збереження завантажених файлів у відповідних директоріях.
- Очищення застарілих файлів або створення архівів.

 downloader_tests.py:
Тести для перевірки правильності роботи сервісу завантаження. Може включати:

- Тестування коректності завантаження з різних джерел.
- Перевірка обробки помилок, наприклад, при недоступності URL.
- Тестування збереження файлів та їх цілісності.

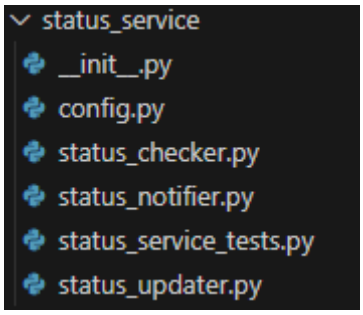


Рис. 3 — Структура сервісу статусу

 __init__.py: Ініціалізація пакету для імпорту функціональності сервісу.
 status_updater.py: Логіка для оновлення статусу в комунікаторах (наприклад, встановлення статусу "занятий" чи "відсутній").
 config.py: Налаштування для сервісу, включаючи API-ключі для месенджерів і параметри тривалості бездіяльності.
 status_checker.py: Перевірка активності користувача та визначення бездіяльності для зміни статусу.
 status_notifier.py: Сповіщення користувача про зміни статусу через відповідні канали.
 status_service_tests.py: Тести для перевірки правильності роботи сервісу оновлення статусу.

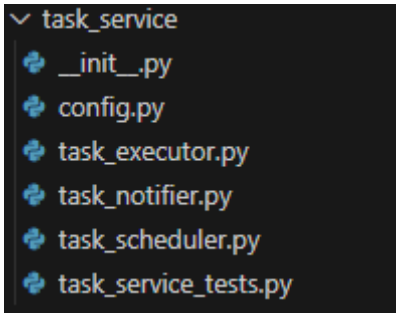


Рис. 4 — Структура сервісу Завдань

 __init__.py: Ініціалізація пакету для імпорту функціональності сервісу.

 task_scheduler.py: Логіка для планування задач, таких як виконання завдань у певний час (наприклад, завантаження файлів або зміна статусу).
 config.py: Налаштування для сервісу, включаючи параметри планувальника, час виконання задач і інтервали.
 task_executor.py: Логіка для виконання задач, визначених у планувальнику.
 task_notifier.py: Сповіщення про виконання задач, успіх чи помилки при їх виконанні.
 task_service_tests.py: Тести для перевірки правильності роботи планувальника задач і виконання завдань.

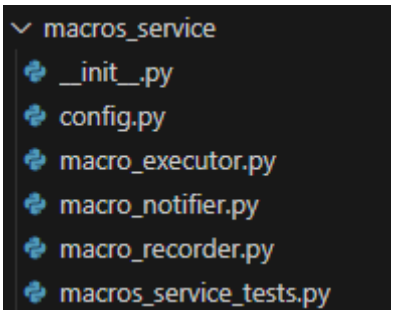


Рис. 5 — Структура сервісу макроссів

 __init__.py: Ініціалізує пакет для імпорту функціональності макросів.
 macro_recorder.py: Записує дії користувача, такі як натискання клавіш і рухи миші.
 macro_executor.py: Виконує раніше записані макроси, відтворюючи дії користувача.
 config.py: Налаштування для сервісу макросів, включаючи параметри збереження та часу виконання.
 macro_notifier.py: Сповіщає користувача про статус виконання макросів.

 `macros_service_tests.py`: Тести для перевірки коректності роботи сервісу макросів.

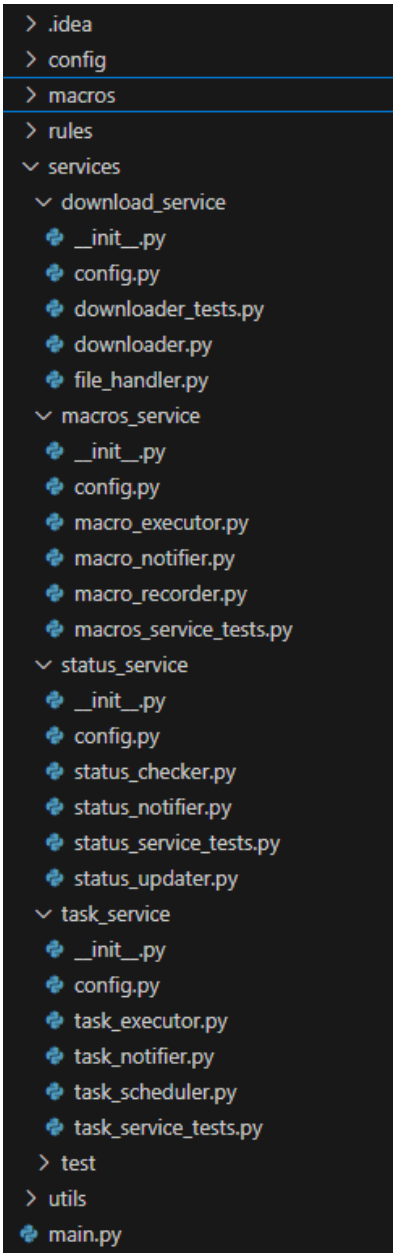


Рис. 6 — Структура головного додатку

- Центральний модуль **MainController** координує всі сервіси за допомогою клієнтських інтерфейсів:
  - **DownloadServiceClient** для завантаження файлів.
  - **StatusServiceClient** для управління статусами.
  - **TaskServiceClient** для планування і виконання задач.
  - **MacroServiceClient** для роботи з макросами.
- Модуль забезпечує єдину точку входу, через яку відбувається маршрутизація запитів до відповідних сервісів.
- Центральний контролер агрегує функціональність для:

- Завантаження файлів (наприклад, автоматичне завантаження нових серій, книг).
- Оновлення статусів в комунікаторах.
- Планування задач (наприклад, виконання задач о певному часу).
- Запис та виконання макросів.

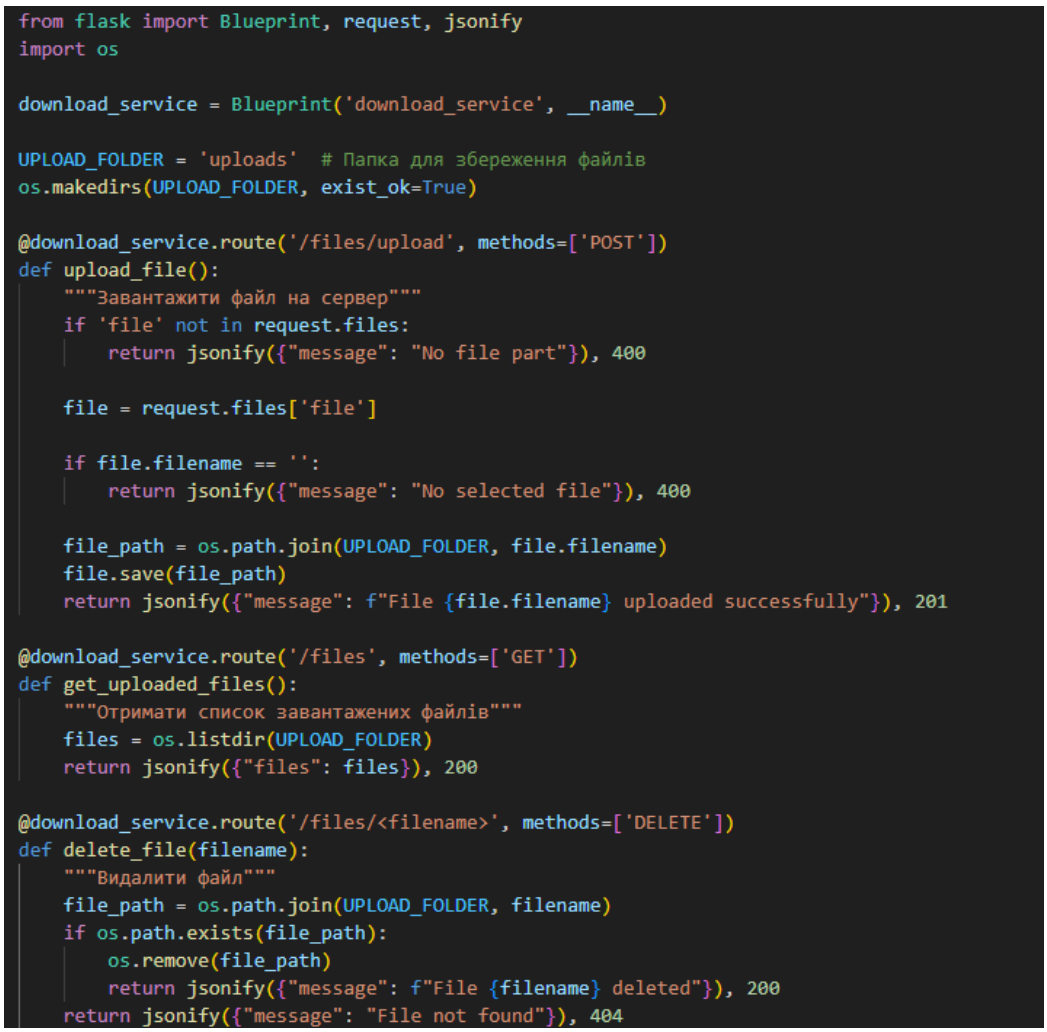
Сервіси:

1. **DownloadService**:
   - Відповідає за завантаження файлів з різних джерел.
   - Взаємодіє з іншими сервісами для автоматизації завантаження нових файлів.
2. **StatusService**:
   - Оновлює статуси в комунікаторах на основі активності користувача.
   - Взаємодіє з планувальником задач для зміни статусу в заданий час.
3. **TaskService**:
   - Планує виконання задач на основі правил і часу.
   - Виконує заплановані задачі (наприклад, завантаження торрентів або зміна статусу).
4. **MacroService**:
   - Записує та виконує макроси для автоматизації дій користувача (натискання клавіш, дії миші).

Переваги такої архітектури:

- **Розподіл відповідальності**: Кожен сервіс має чітко визначену роль і функціональність.
- **Централізоване керування**: Головний модуль координує запити, забезпечуючи централізоване управління всіма функціями.
- **Масштабованість**: Кожен сервіс може масштабуватися незалежно залежно від навантаження.
- **Гнучкість**: Додаються нові сервіси або розширюється функціональність через чітко визначені інтерфейси.
- **Ізольовані тести**: Завдяки чітким сервісам, кожен компонент можна тестувати окремо.

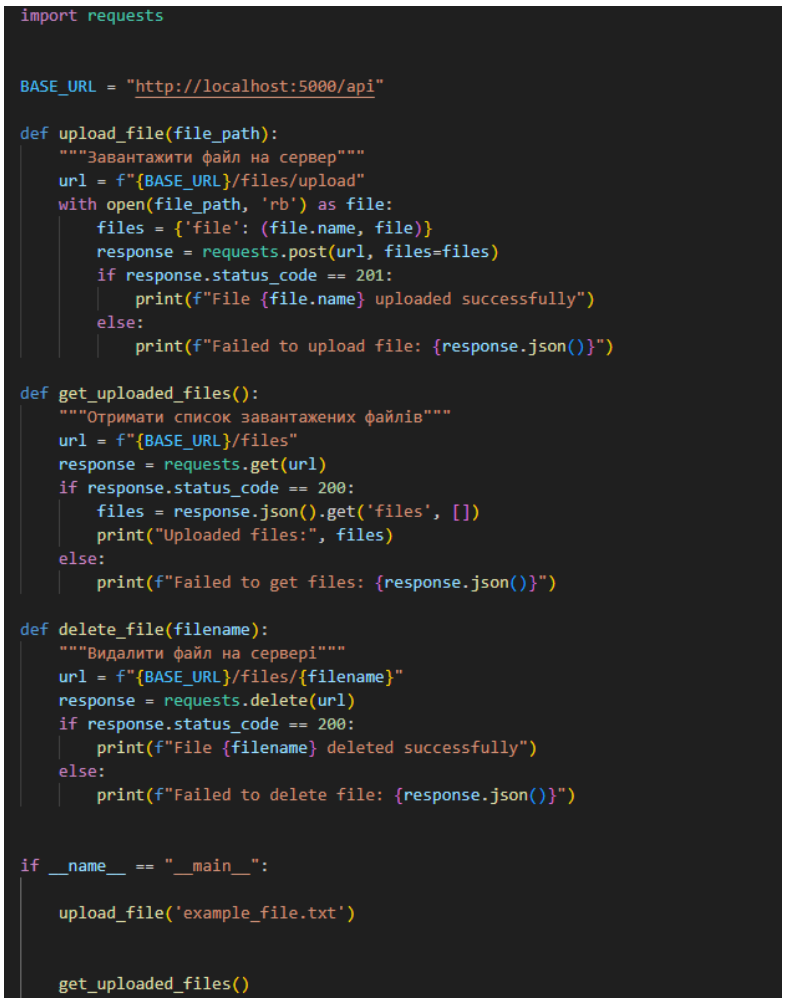
2. Реалізувати взаємодію розподілених частин



```python
from flask import Blueprint, request, jsonify
import os

download_service = Blueprint('download_service', __name__)

UPLOAD_FOLDER = 'uploads'  # Папка для збереження файлів
os.makedirs(UPLOAD_FOLDER, exist_ok=True)

@download_service.route('/files/upload', methods=['POST'])
def upload_file():
    """Завантажити файл на сервер"""
    if 'file' not in request.files:
        return jsonify({"message": "No file part"}), 400

    file = request.files['file']

    if file.filename == '':
        return jsonify({"message": "No selected file"}), 400

    file_path = os.path.join(UPLOAD_FOLDER, file.filename)
    file.save(file_path)
    return jsonify({"message": f"File {file.filename} uploaded successfully"}), 201

@download_service.route('/files', methods=['GET'])
def get_uploaded_files():
    """Отримати список завантажених файлів"""
    files = os.listdir(UPLOAD_FOLDER)
    return jsonify({"files": files}), 200

@download_service.route('/files/<filename>', methods=['DELETE'])
def delete_file(filename):
    """Видалити файл"""
    file_path = os.path.join(UPLOAD_FOLDER, filename)
    if os.path.exists(file_path):
        os.remove(file_path)
        return jsonify({"message": f"File {filename} deleted"}), 200
    return jsonify({"message": "File not found"}), 404
```

Рис. 7 — Реалізація сервіса по завантаженню

Цей рисунок ілюструє код контролера для роботи з завантаженням файлів у RESTful веб-сервісі. Сервіс завантаження файлів реалізований як окремий сервіс для керування файлами, надаючи такі функції: завантаження файлів на сервер, перегляд списку завантажених файлів та видалення файлів. Реалізація методу upload_file дозволяє користувачам завантажувати файли через API. Функція get_uploaded_files відповідає за отримання списку всіх завантажених файлів, а метод delete_file дозволяє видаляти файли з сервера на основі їх імені. Цей сервіс забезпечує просту взаємодію з файлами, дозволяючи керувати ними через HTTP запити.



```python
import requests


BASE_URL = "http://localhost:5000/api"

def upload_file(file_path):
    """Завантажити файл на сервер"""
    url = f"{BASE_URL}/files/upload"
    with open(file_path, 'rb') as file:
        files = {'file': (file.name, file)}
        response = requests.post(url, files=files)
        if response.status_code == 201:
            print(f"File {file.name} uploaded successfully")
        else:
            print(f"Failed to upload file: {response.json()}")

def get_uploaded_files():
    """Отримати список завантажених файлів"""
    url = f"{BASE_URL}/files"
    response = requests.get(url)
    if response.status_code == 200:
        files = response.json().get('files', [])
        print("Uploaded files:", files)
    else:
        print(f"Failed to get files: {response.json()}")

def delete_file(filename):
    """Видалити файл на сервері"""
    url = f"{BASE_URL}/files/{filename}"
    response = requests.delete(url)
    if response.status_code == 200:
        print(f"File {filename} deleted successfully")
    else:
        print(f"Failed to delete file: {response.json()}")


if __name__ == "__main__":

    upload_file('example_file.txt')


    get_uploaded_files()
```

Рис. 8 — Зв’язок з сервісом завантаження файлів в головному додатку

Цей рисунок демонструє клас-клієнт для сервісу завантаження файлів, який реалізує функціональність для роботи з файлами в межах вашого додатку. Клієнт має методи для завантаження файлів на сервер, отримання списку завантажених файлів та видалення файлів з системи. Взаємодія з сервісом здійснюється через локальні API запити, де кожен метод відповідає за певну операцію над файлами. Цей сервіс можна інтегрувати з іншими модулями

системи, такими як модулі для обробки даних або управління іншими об’єктами

```python
from download_service import download_file, get_uploaded_files, delete_file
from status_service import update_status
from task_service import schedule_task
from macros import record_macro, execute_macro

class MainController:
    def __init__(self):

        self.download_service = download_file
        self.status_service = update_status
        self.task_service = schedule_task
        self.macro_service = (record_macro, execute_macro)

    def handle_file_upload(self, file_path):
        """Обробка завантаження файлів"""
        self.download_service(file_path)
        print(f"File {file_path} uploaded successfully.")

    def show_uploaded_files(self):
        """Отримання списку завантажених файлів"""
        files = get_uploaded_files()
        print("Uploaded files:", files)

    def remove_file(self, filename):
        """Видалення файлів"""
        delete_file(filename)
        print(f"File {filename} deleted successfully.")

    def update_user_status(self, status):
        """Оновлення статусу"""
        self.status_service(status)
        print(f"Status updated to {status}.")

    def schedule_new_task(self, task_time):
        """Запланувати нове завдання"""
        self.task_service(task_time)
        print(f"Task scheduled at {task_time}.")

    def record_new_macro(self, macro_actions):
        """Запис макроса"""
        self.macro_service[0](macro_actions)
        print(f"Macro recorded with actions: {macro_actions}.")

    def execute_macro(self, macro_name):
        """Виконання макроса"""
        self.macro_service[1](macro_name)
        print(f"Macro {macro_name} executed.")

if __name__ == "__main__":

    controller = MainController()


    controller.handle_file_upload("example_file.txt")
    controller.show_uploaded_files()
    controller.remove_file("example_file.txt")
    controller.update_user_status("Available")
    controller.schedule_new_task("2025-01-20 09:00:00")
    controller.record_new_macro(["Ctrl+S", "Ctrl+P"])
    controller.execute_macro("auto_save_macro")
```

Рис. 9 — Головний контролер

У цьому тексті представлене описання основного контролера, який координує роботу між різними сервісами у вашій системі. Головний контролер взаємодіє з сервісами для завантаження файлів, оновлення статусів, планування задач та запису макросів. Він реалізує методи для кожного з цих сервісів, делегуючи виконання конкретних операцій безпосередньо відповідним сервісам.

Головний контролер також забезпечує уніфіковану взаємодію з іншими модулями через чітко визначені методи, що спрощує інтеграцію з іншими частинами системи. Це дає змогу централізовано керувати роботою всіх сервісів, спрощуючи обробку даних та виконання завдань.

Висновки:

Ми реалізували систему, де кожен сервіс виконує свою специфічну задачу в межах загальної системи. Це дозволяє забезпечити масштабованість, незалежність і гнучкість в оновленні кожного з сервісів, а також надає можливість для подальшої модифікації без порушення цілісності всього додатку.

Код: https://github.com/Lepseich/trpz/tree/main/lab9/files