НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
В.о. завідувача кафедри
_____ Михайло НОВОТАРСЬКИЙ
(підпис)

“__” _____2025 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

на тему: Веб-застосунок для централізованої публікації контенту в соціальних мережах

Виконав : студент __4__ курсу, групи _ІМ-11
(шифр групи)

_____Волков Ілля Андрійович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____асистент, Гайдай А.Р._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) асистент, Нікольський С.С._____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент ___доцент кафедри ІСТ, к.т.н. Шимкович В. М._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2025 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Михайло НОВОТАРСЬКИЙ
(підпис)
“__” _____2025 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Волкова Іллі Андрійовича

1. Тема проєкту *Веб-застосунок для централізованої публікації контенту в соціальних мережах*
керівник проєкту Гайдай Анатолій Русланович, асистент,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 23 травня 2025 року №1705-с

2. Термін здачі студентом закінченого проєкту *03 червня 2025 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Огляд існуючих рішень централізованої публікації контенту.

Розділ 2. Вибір архітектури та технологій розробки веб-застосунку.

Розділ 3. Деталі розробки веб-застосунку.

Розділ 4. Тестування та аналіз розробленого проекту.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) принципова схема (блок-схема алгоритму), структурна схема системи, функціональна схема (діаграма класів).

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Нікольський С.С.		

7. Дата видачі завдання «2» січня 2025 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*27.01.2025-02.02.2025*	
2.	*Вивчення та аналіз завдання*	*03.02.2025-15.02.2025*	
3.	*Розробка архітектури та загальної структури системи*	*20.02.2025-15.03.2025*	
4.	*Розробка структур окремих підсистем*	*15.03.2025-20.03.2025*	
5.	*Програмна реалізація системи*	*20.03.2025-01.06.2025*	
6.	*Оформлення пояснювальної записки*	*14.04.2025-31.05.2025*	
7.	*Захист програмного продукту*	*03.06.2025*	
8.	*Передзахист*	*05.06.2025*	
9.	*Захист*	*17.06.2025*	

Студент-дипломник _____ Ілля ВОЛКОВ
(підпис)

Керівник проєкту _____ Анатолій ГАЙДАЙ
(підпис)

АНОТАЦІЯ

У даній дипломній роботі проведено огляд та аналіз існуючих систем для централізованої публікації контенту в соціальних мережах. На основі проведеного аналізу та потреб користувачів було сформульовано вимоги та обрано пріоритетні функції для розробки власного веб-застосунку, що автоматизує процес розповсюдження контенту.

В результаті було розроблено веб-застосунок для централізованої публікації відео- та текстового контенту у популярних соціальних мережах, включаючи YouTube, TikTok, Facebook та Twitter. Програмний продукт реалізовано з використанням мов JavaScript та TypeScript; серверна частина побудована на Node.js та Express.js, клієнтська – на React, для зберігання даних використовується СУБД MongoDB.

Ключові слова: централізована публікація контенту, соціальні мережі, веб-застосунок, API, JavaScript, Node.js, React, TypeScript, MongoDB.

ANNOTATION

This thesis involved a review and analysis of existing systems for centralized content publishing on social networks. Based on this analysis and user needs, requirements were formulated and priority functions were selected for the development of a custom web application to automate the content distribution process.

As a result, a web application was developed for centralized publishing of video and text content to popular social networks, including YouTube, TikTok, Facebook, and Twitter. The software product was implemented using JavaScript and TypeScript languages; the server-side was built on Node.js and Express.js, the client-side on React, MongoDB DBMS is used for data storage.

Keywords: centralized content publishing, social networks, web application, API, JavaScript, Node.js, React, TypeScript, MongoDB.

справки	Формат	*Значення*	*Найменування*	Кіл. Листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Веб-застосунок для	3		
			централізованої публікації			
			контенту в соціальних мережах			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Веб-застосунок для	71		
			централізованої публікації			
			контенту в соціальних мережах			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Веб-застосунок для	1		
			централізованої публікації			
			контенту в соціальних мережах			
			Принципова схема			
			(блок-схема алгоритму)			
	А4	***ІАЛЦ.4672008.005 Д2***	Веб-застосунок для	1		
			централізованої публікації			
			контенту в соціальних мережах			
			Функціональна схема			
			(діаграма класів)			
	А4	***ІАЛЦ.467200.006 Д3***	Веб-застосунок для	1		
			централізованої публікації			
			контенту в соціальних мережах			
			Структурна схема			
			системи			
	А4	***ІАЛЦ.467200.007 Д4***	Веб-застосунок для	60		
			централізованої публікації			
			контенту в соціальних мережах			
			Текст програмного коду			

					ІАЛЦ.467200.001 ОА
Зм	***Лист***	***№ докум.***	***Підп***	***Дата***	
Розроб		Волков І.А.			*Веб-застосунок для централізованої публікації контенту в соціальних мережах* Опис альбому
Перев		Гайдай А.Р.			

Літ.	Аркуш	Аркушів
	1	1

НТУУ "КПІ" ФІОТ ІМ-11

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Веб-застосунок для централізованої публікації контенту в соціальних мережах*»

Київ – 2025

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку веб-застосунку для централізованої публікації контенту (відео та текстових постів) у популярних соціальних мережах, а також на подальшу підтримку та можливе вдосконалення розробленого застосунку.

Областю застосування цієї системи є автоматизація та спрощення процесу публікації контенту для користувачів, які активно працюють з кількома соціальними мережами.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є розробка веб-застосунку, що надасть користувачам єдиний інтерфейс для ефективного завантаження, налаштування та планування публікації контенту до популярних соціальних мереж через їх офіційні API.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелами розробки даного дипломного проєкту є офіційна технічна документація використаних технологій та програмних інтерфейсів, науково-технічна література, а також тематичні публікації та статті в мережі Інтернет.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблений веб-застосунок має виконувати такі вимоги:

- Простий та інтуїтивно-зрозумілий користувацький інтерфейс.
- Забезпечити реєстрацію та безпечну автентифікацію користувачів.
- Надати можливість підключення акаунтів соціальних мереж.
- Надати можливість користувачам завантажувати файли та створювати текстові пости.

5.2. Вимоги до програмного забезпечення

- ОС Windows, Mac чи Linux.
- Visual Studio Code.
- Node.js .
- Веб-браузер.

5.3. Вимоги до апаратної частини

- ROM не менше ніж 5 ГБ.
- RAM не менше ніж 4 ГБ.
- Підключення до мережі Інтернет.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	02.02.2025
Вивчення та аналіз завдання	15.02.2025
Розробка архітектури та загальної структури системи	15.03.2025
Розробка структур окремих частин системи	30.03.2025
Програмна реалізація системи	31.05.2025
Виправлення помилок	02.06.2025
Оформлення пояснювальної записки	03.06.2025

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Веб-застосунок для централізованої публікації контенту в соціальних мережах*»

Київ – 2025

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

API	(Application Programming Interface) Прикладний програмний інтерфейс
CSS	(Cascading Style Sheets) Каскадні таблиці стилів
HTTP	(HyperText Transfer Protocol) Протокол передачі гіпертексту
HTTPS	(HyperText Transfer Protocol Secure) Захищений протокол передачі гіпертексту
HTML	(HyperText Markup Language) Мова розмітки гіпертексту
JSON	(JavaScript Object Notation) Нотація об'єктів JavaScript
JWT	(JSON Web Tokens) Веб-токени JSON
REST	(Representational State Transfer) Передача репрезентативного стану
SQL	(Structured Query Language) Мова структурованих запитів
NoSQL	(Not only SQL) Не тільки SQL
OAuth	(Open Authorization) Відкрита авторизація
URL	(Uniform Resource Locator) Уніфікований покажчик ресурсу
СУБД	Система Управління Базами Даних

ВСТУП

У сучасному інформаційному суспільстві соціальні мережі стали невід'ємною платформою для комунікації, поширення новин, просування брендів та творчості. Для авторів, бізнесу та медійних особистостей ефективна присутність на кількох платформах одночасно стає необхідністю, проте ручне управління публікаціями є часозатратним та знижує загальну продуктивність. Зростання обсягів контенту та вимог до оперативності його поширення зумовлює актуальність розробки автоматизованих інструментів для централізованого управління публікаціями.

Метою даної дипломної роботи є розробка та реалізація веб-застосунку для централізованого завантаження та публікації відео- та текстового контенту на популярні соціальні мережі (YouTube, TikTok, Facebook, Twitter) через єдиний інтерфейс з використанням їх офіційних API.

У процесі виконання роботи буде проведено аналіз існуючих програмних рішень для управління контентом у соціальних мережах та сформульовано вимоги до розроблюваної системи. Також буде здійснено огляд та обґрунтування вибору сучасних веб-технологій для реалізації серверної та клієнтської частин застосунку та розглянуто принципи взаємодії з API соціальних мереж. Наступні етапи передбачають проєктування архітектури, програмну реалізацію веб-застосунку та його тестування для перевірки функціональності й відповідності поставленим завданням.

Розробка такого веб-застосунку є актуальним завданням, оскільки він може стати корисним інструментом для широкого кола користувачів – від індивідуальних авторів контенту та блогерів до SMM-спеціалістів та представників малого бізнесу, допомагаючи їм економити час та підвищувати ефективність своєї присутності в соціальних мережах.

РОЗДІЛ 1. ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ЦЕНТРАЛІЗОВАНОЇ ПУБЛІКАЦІЇ КОНТЕНТУ

1.1 Актуальність та доцільність розроблення

В умовах, коли цифрові технології та соціальні мережі активно розвиваються, питання розповсюдження контенту стає все більш актуальним. Соціальні мережі стали важливим інструментом комунікації як для бізнесу, так і для індивідуальних авторів, блогерів, які просувають особисті бренди, творчість або інформаційні продукти. При цьому контент-стратегії дедалі частіше передбачають публікацію контенту одразу на кількох популярних платформах, щоб охопити більше аудиторії. Однак у більшості випадків реалізувати це доводиться вручну, що вимагає повторного виконання однакових дій: відкривати кожен сайт окремо, заповнювати ті самі поля, завантажувати одні й ті самі файли, враховуючи особливості кожної платформи. Це займає багато часу, створює ризик помилок і знижує продуктивність.

З огляду на експоненційне зростання обсягів створюваного цифрового контенту, ефективні інструменти для його менеджменту та розповсюдження набувають критичного значення. Актуальність проблеми зумовлює потребу в автоматизованих інструментах, які дозволяють централізовано керувати процесом розміщення матеріалів на різних платформах в єдиному інтерфейсі. Такі веб-застосунки дозволяють оптимізувати рутинні операції, знизити ймовірність помилок під час роботи та підвищити ефективність цифрової присутності користувача або організації.

Для таких програмних рішень ключовим є використання прикладних програмних інтерфейсів (API), які надають більшість популярних соціальних

мереж. API забезпечують структурований доступ до функціональності платформи, що включає можливість створювати, редагувати та видаляти публікації, отримувати аналітичні дані, керувати медіафайлами та іншими ресурсами без потреби прямої взаємодії користувача з сайтом окремої платформи. Крім того, широке застосування сучасних веб-технологій, зокрема REST-архітектури та авторизаційних протоколів OAuth 2.0, створює сприятливі умови для реалізації стабільного, масштабованого та безпечного інструменту для централізованої публікації контенту.

1.2 Огляд та приклади існуючих рішень

На ринку існує немало готових програмних продуктів, що реалізують централізовану публікацію контенту в соціальних мережах. Їх функціональність варіюється від базового планування постів до комплексного управління соціальними мережами. Було розглянуто найбільш відомі приклади таких веб-застосунків, зокрема їх функціональні можливості та цільове призначення. Аналіз цих рішень дозволить виявити загальні тенденції розвитку галузі, а також існуючі переваги та недоліки, що є важливим етапом для формування вимог до створення веб-застосунку.

1.2.1 Hootsuite

Hootsuite – один із найстаріших та найвідоміших сервісів для управління соціальними мережами, який позиціонується як комплексне рішення для широкого спектру користувачів, від малого бізнесу до великих підприємств. Він підтримує роботу з низкою популярних платформ, включаючи Facebook, Instagram, Twitter, LinkedIn, YouTube та інші. Ключовими можливостями системи є не тільки відкладене публікування контенту, але й глибокий

моніторинг згадок (social listening) та розширена аналітика взаємодій. Крім того, Hootsuite вирізняється потужними інструментами для спільної роботи в команді. Користувачі можуть керувати потоками даних за допомогою персоналізованої панелі, яка гнучко налаштовує робочий простір (рис. 1.1). Hootsuite також надає можливості API для інтеграції зі сторонніми системами. Слід зазначити, що при всій своїй функціональній повноті, Hootsuite може мати відносно високу вартість, що робить його менш доступним для індивідуальних користувачів або дуже малого бізнесу [1].

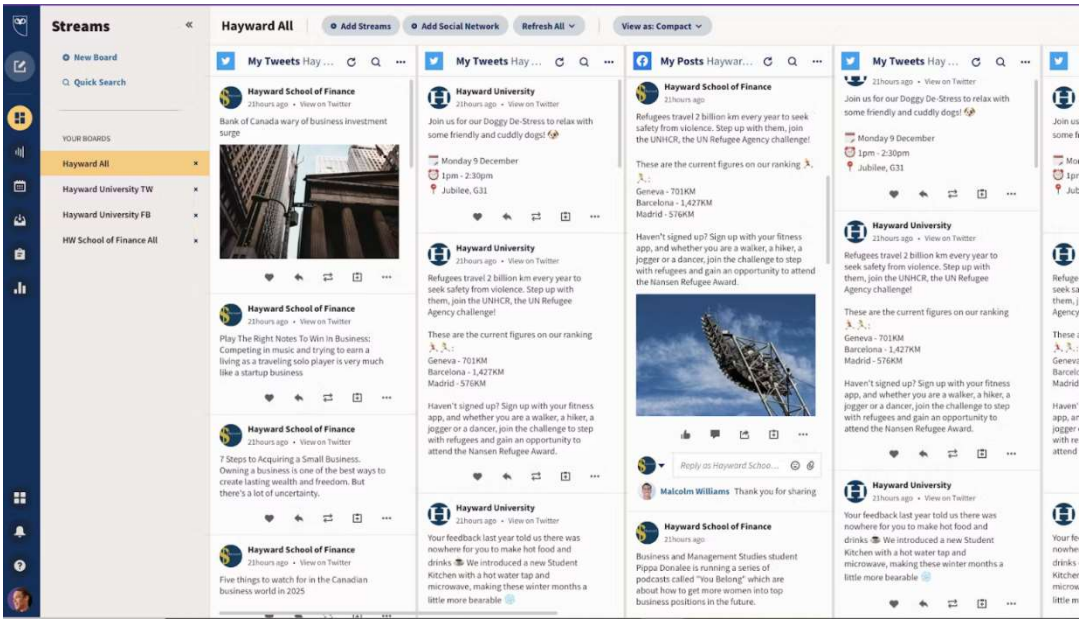


Рисунок 1.1 – Інтерфейс панелі управління публікаціями Hootsuite

1.2.2 Buffer

Buffer – це платформа, яка, на відміну від Hootsuite, робить акцент на простоті використання та ефективному плануванні публікацій. Вона ідеально підходить для індивідуальних маркетологів та невеликих команд, які цінують інтуїтивно зрозумілий інтерфейс. Buffer дозволяє створювати та управляти чергами публікацій для різних соціальних мереж (Facebook, Twitter, Instagram, LinkedIn, Pinterest), відстежувати показники взаємодії (лайки,

поширення, коментарі), а також ефективно працювати в командному режимі завдяки вбудованим інструментам співпраці. А також автоматичне визначення оптимального часу для постингу та легке перепланування контенту. Особливістю Buffer є його фокус на візуальному плануванні та гнучкому налаштуванні графіків публікацій (рис. 1.2). Хоча Buffer може надавати менш розширену аналітику порівняно з Hootsuite, він виграє у зручності використання та швидкості освоєння [1].

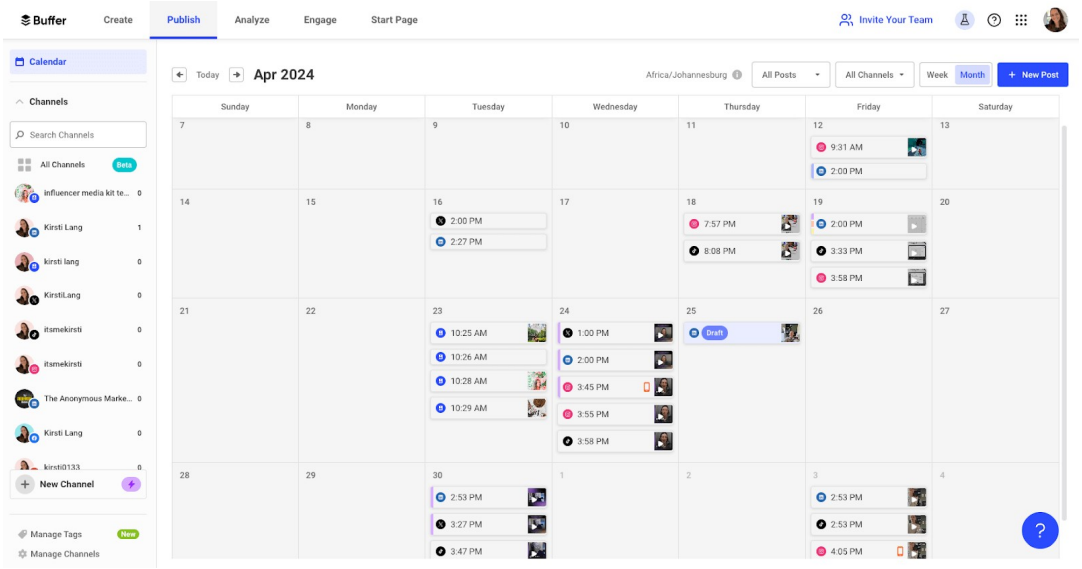


Рисунок 1.2 – Інтерфейс панелі управління публікаціями Buffer

1.2.3 SocialPilot

SocialPilot є ще одним представником програмних рішень для централізованого управління соціальними мережами, який позиціонується як інструмент, що спрощує та оптимізує виконання типових завдань з планування та публікації контенту. Ця платформа орієнтована переважно на малий та середній бізнес, маркетингові агентства, а також індивідуальних користувачів. І пропонує обширний набір функцій, необхідних для підтримки активної присутності в соціальних мережах. SocialPilot підтримує інтеграцію з більшістю популярних соціальних мереж, подібно до інших систем цього класу. Серед

ключових функціональних можливостей слід виділити систему планування та масового завантаження контенту, наявність візуального календарного перегляду, який надає зручний спосіб управління розкладом публікацій, а також функцію адаптивного планування з урахуванням часових поясів, що дозволяє оптимізувати час появи контенту для різних аудиторій (рис. 1.3). Важливою перевагою SocialPilot є його відносно доступна цінова політика порівняно з іншими аналогами, що робить його популярним вибором серед користувачів з обмеженим бюджетом. SocialPilot може не мати деяких просунутих функцій, присутніх у дорожчих комплексних рішеннях, що може обмежувати його застосування для складніших стратегій чи великих підприємств [2].

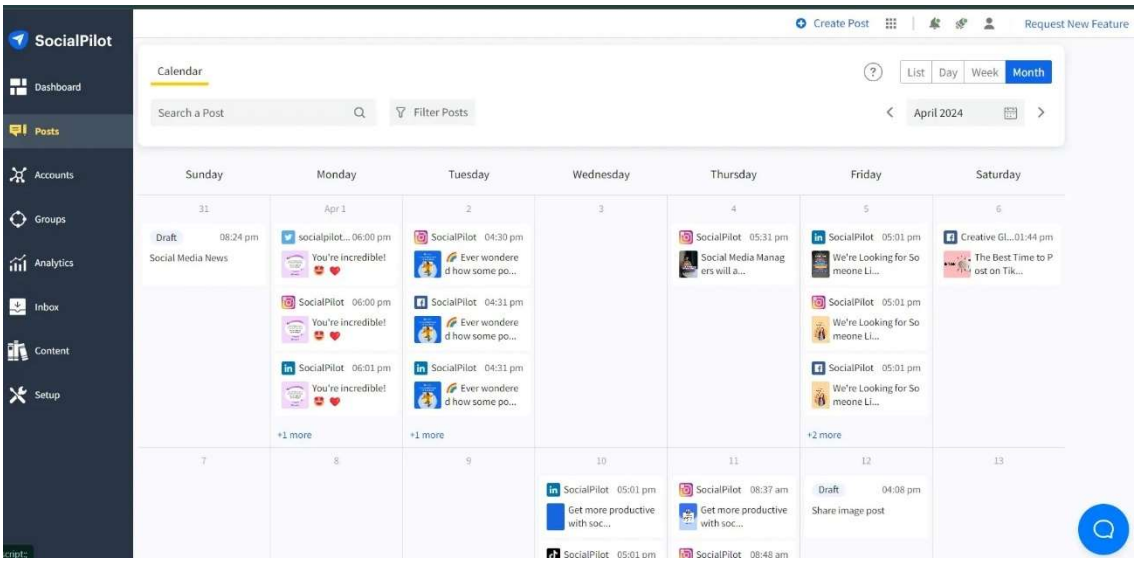


Рисунок 1.3 – Інтерфейс панелі управління публікаціями SocialPilot

1.3 Аналіз функціональності та вимог до системи

1.3.1 Порівняльна характеристика розглянутих рішень

На основі попереднього аналізу існуючих платформ було здійснено порівняння їхніх ключових переваг і недоліків у таблиці 1.1.

Таблиця 1.1 – Порівняльна характеристика систем централізованої публікації контенту

Платформа	Переваги	Недоліки
Hootsuite	1. Підтримка багатьох соціальних мереж 2. Розширена аналітика та моніторинг 3. Підтримка командної роботи 4. Інтеграція зі сторонніми додатками	1. Висока вартість 2. Інтерфейс може бути складним для нових користувачів 3. Надлишкова кількість специфічних функцій
Buffer	1. Інтуїтивно зрозумілий інтерфейс 2. Ефективне планування та управління чергами публікацій 3. Наявність безкоштовного тарифного плану	1. Базовий рівень аналітики без деталізації 2. Ліміт на кількість публікацій 3. Обмежений набір просунутих функцій
SocialPilot	1. Підтримка нішевих мереж 2. Функція масового завантаження контенту 3. Доступна цінова політика	1. Застарілий інтерфейс із навантаженим меню 2. Відсутність просунутої аналітики 3. Обмежені функції у нижчих тарифних планах

Проведений порівняльний аналіз існуючих систем для централізованої публікації контенту демонструє, що кожна з розглянутих платформ має свої

сильні сторони, але водночас і певні обмеження. Hootsuite вирізняється широким функціоналом та підтримкою багатьох соціальних мереж, проте його висока вартість та складність можуть бути бар'єром для індивідуальних користувачів. Buffer приваблює простотою інтерфейсу та наявністю безкоштовного плану, але поступається в аналітичних можливостях та має обмеження на кількість публікацій. SocialPilot пропонує компроміс між функціональністю та ціною, підтримуючи масове завантаження, однак може мати застарілий інтерфейс та обмеження в аналітиці й функціоналі на нижчих тарифах. Отже, незважаючи на наявність різноманітних інструментів, існуючі рішення не завжди повною мірою задовольняють специфічні потреби окремих категорій користувачів, зокрема тих, хто шукає доступний, простий у використанні та функціональний інструмент для роботи з певними типами контенту та платформами.

1.3.2 Вимоги до реалізації веб-застосунку

З урахуванням аналізу існуючих рішень та потреб користувачів, у межах даного дипломного проєкту реалізується веб-застосунок для централізованої публікації контенту в соціальних мережах. Вимоги до системи формувалися з урахуванням прагнення забезпечити базову функціональність при мінімальній складності використання, повній безкоштовності та сумісності з API основних соціальних мереж.

Функціональні вимоги:

1. Інтеграція з популярними соціальними мережами. Застосунок має підтримувати публікацію контенту в YouTube, TikTok, Facebook та Twitter, з можливістю подальшого розширення підтримки на інші соціальні мережі.

2. Підтримка базових форматів контенту. Система має забезпечувати можливість публікації відео-матеріалів та звичайних постів, що можуть містити текст і зображення.
3. Механізм планування. Користувач повинен мати можливість вказати дату і час публікації контенту.
4. Мультикористувацька підтримка. Система повинна забезпечувати реєстрацію, вхід та індивідуальні налаштування для кожного користувача.
5. Безпечна авторизація. Вхід у аккаунти соціальних мереж повинен здійснюватися через протокол авторизації OAuth 2.0 без збереження облікових даних користувача на стороні застосунку.
6. Відображення статусу публікацій. Користувач має бачити, чи була публікація успішною, та мати доступ до посилання на опублікований матеріал.
7. Простий та зрозумілий інтерфейс. Застосунок повинен мати зручну структуру, придатну для користувачів без спеціальних технічних знань.

Нефункціональні вимоги:

1. Безкоштовне використання. Система не повинна вимагати оплати чи підписки за базову функціональність.
2. Можливість масштабування. Архітектура має дозволяти підключення нових платформ або форматів публікацій.
3. Стабільність і надійність. Повинна бути передбачена обробка помилок при зверненні до API платформ та відображення відповідних повідомлень.

Визначені функціональні та нефункціональні вимоги є основою для подальшого проєктування архітектури та безпосередньої програмної реалізації веб-застосунку.

ВИСНОВОК ДО РОЗДІЛУ 1

У першому розділі було проведено аналіз предметної області, що включає вивчення актуальності та проблем, пов’язаних з централізованою публікацією контенту в соціальних мережах в умовах їх інтенсивного розвитку та зростання обсягів цифрового контенту. Було розглянуто основні принципи та переваги використання автоматизованих інструментів для оптимізації цього процесу.

Таким чином, тематика централізованої публікації контенту є вкрай актуальною на сьогодні, про що свідчить наявність значної кількості відомих програмних систем та розробок, що активно використовуються як в Україні, так і за кордоном. Проведений аналіз цих рішень дозволив виявити ключові функціональні особливості та обмеження, що існують на ринку.

Зокрема, було встановлено, що існуючі комерційні платформи, хоча і є потужними, часто мають високу вартість, надлишкову складність або неповну підтримку специфічних потреб, важливих для індивідуальних користувачів та малого бізнесу, орієнтованих на публікацію відео- та текстового контенту в соціальних мережах. Це обґрунтовує доцільність розроблення альтернативного, більш доступного та цілеспрямованого рішення.

На основі проведеного аналізу предметної області та існуючих аналогів було сформульовано функціональні та нефункціональні вимоги до розроблюваного веб-застосунку. Ці вимоги передбачають реалізацію ключового функціоналу для обраних платформ та форматів контенту, забезпечення безпеки та простоти використання, що є основою для подальшого проєктування та програмної реалізації системи в рамках даного дипломного проєкту.

РОЗДІЛ 2. ОГЛЯД ТЕХНОЛОГІЙ ДЛЯ ПРОЕКТУВАННЯ ВЕБ-ЗАСТОСУНКУ

2.1 Загальні принципи побудови веб-застосунків

Сучасні веб-застосунки є складними програмними системами, призначеними для надання інтерактивних сервісів користувачам через мережу Інтернет. Їх розроблення базується на низці фундаментальних принципів та архітектурних підходів, що забезпечують їх функціональність, надійність та масштабованість. Розуміння цих принципів є ключовим для проєктування ефективних веб-рішень, зокрема таких, як система централізованої публікації контенту.

2.1.1 Клієнт-серверна архітектура

В основі більшості сучасних веб-застосунків лежить клієнт-серверна архітектура. Ця модель розподіляє завдання між двома основними типами взаємодіючих компонентів: клієнтом та сервером.

Клієнт (Client) – це програмне забезпечення, яке зазвичай працює на пристрої кінцевого користувача (наприклад, веб-браузер на комп’ютері або мобільному пристрої). Його основні функції включають відображення користувацького інтерфейсу, обробку дій користувача, формування запитів до сервера та відображення отриманих від сервера даних. У контексті веб-застосунків клієнт відповідає за представлення інформації та забезпечення взаємодії з користувачем. Сучасні підходи до розробки фронтенду часто передбачають створення односторінкових застосунків (Single Page Application). При використанні такого підходу весь необхідний код (HTML, CSS, JavaScript)

завантажується один раз при першому відкритті сторінки, а подальша навігація та оновлення даних відбуваються динамічно, без повного перезавантаження сторінки з сервера. Це забезпечує більш швидкий та плавний користувацький досвід, подібний до роботи з настільними програмами. Для створення складних інтерактивних інтерфейсів, зокрема SPA (Single Page Application), часто використовуються сучасні JavaScript-бібліотеки та фреймворки, такі як React, Angular або Vue.js [3].

Сервер (Server) – це потужний комп'ютер або система комп'ютерів, що зберігає дані, обробляє бізнес-логіку застосунку та відповідає на запити від клієнтів. Сервер виконує такі завдання, як автентифікація та авторизація користувачів, доступ до баз даних, виконання складних обчислень, взаємодія з іншими системами (наприклад, API сторонніх сервісів) та генерація відповідей для клієнтів. У розроблюваному веб-застосунку серверна частина відповідатиме за обробку запитів на публікацію контенту, взаємодію з API соціальних мереж та управління даними користувачів.

Взаємодія між клієнтом та сервером відбувається зазвичай за допомогою стандартних мережевих протоколів, найчастіше HTTP/HTTPS. Клієнт надсилає запит (request) до сервера, а сервер, обробивши цей запит, повертає відповідь (response). Така архітектура дозволяє розділити логіку представлення (на клієнті) від логіки обробки даних (на сервері), що сприяє гнучкості розробки, масштабованості та безпеці системи [3].

2.1.2 Основні компоненти веб-застосунку

Типовий веб-застосунок, реалізований за клієнт-серверною архітектурою, складається з трьох ключових компонентів, які тісно взаємодіють між собою: фронтенд, бекенд та база даних.

Фронтенд (Frontend), або клієнтська частина, – це все, з чим безпосередньо взаємодіє користувач у своєму веб-браузері. Він відповідає за візуальне представлення даних, інтерактивність та користувацький досвід. Основними технологіями для розробки фронтенду є HTML для структурування контенту, CSS для його стилізації та JavaScript для реалізації динамічної поведінки та взаємодії з користувачем. Для створення складних інтерактивних інтерфейсів часто використовуються сучасні JavaScript-бібліотеки та фреймворки, такі як React, Angular або Vue.js.

Бекенд (Backend), або серверна частина, – це компонент веб-застосунку, що працює на сервері та недоступний для прямої взаємодії користувачем. Він відповідає за обробку запитів від фронтенду, реалізацію бізнес-логіки, управління даними, автентифікацію та авторизацію, а також взаємодію з іншими сервісами, такими як API соціальних мереж. Для розробки бекенду можуть використовуватися різні мови програмування (наприклад, JavaScript, Python, PHP, Java) та фреймворки.

База даних – це сховище, де зберігається вся необхідна для роботи застосунку інформація. Це можуть бути дані користувачів, налаштування, токени доступу тощо. Бекенд взаємодіє з базою даних для читання, запису, оновлення та видалення даних. Існують різні типи баз даних, зокрема реляційні (SQL), такі як MySQL, PostgreSQL, та нереляційні (NoSQL), наприклад, MongoDB або Cassandra, вибір яких залежить від специфіки даних та вимог до продуктивності.

2.2 Огляд технологій для реалізації веб-застосунку

Вибір відповідного технологічного стеку є критично важливим етапом у розробці будь-якого веб-застосунку, оскільки він безпосередньо впливає на продуктивність, масштабованість, швидкість розробки та подальшу підтримку

системи. Для проєктування системи централізованої публікації контенту було обрано набір сучасних та перевірених технологій, кожна з яких має свої переваги для вирішення поставлених завдань.

2.2.1 Технології серверної частини

Для реалізації серверної частини (бекенду) веб-застосунку було обрано платформу Node.js у поєднанні з фреймворком Express.js.

Node.js визначається як асинхронне, подійно-кероване середовище виконання JavaScript, призначене для створення масштабованих мережевих застосунків. Воно побудоване на JavaScript-рушії V8 від Google Chrome і використовує неблокуючу модель вводу-виводу, що дозволяє ефективно обробляти велику кількість одночасних з'єднань з мінімальним навантаженням на систему. Для даного проєкту Node.js було обрано завдяки його здатності оптимально керувати асинхронними операціями, що є критично важливим для застосунку, який інтенсивно взаємодіє з API соціальних мереж та базою даних. Важливою перевагою є також наявність вбудованого менеджера пакетів npm (Node Package Manager), який надає доступ до великої екосистеми модулів з відкритим кодом, що значно прискорило розробку специфічного функціоналу, такого як робота з API, автентифікація та обробка файлів [4].

Express.js, як мінімалістичний та гнучкий веб-фреймворк для Node.js, дозволяє ефективно створювати REST API ендпоінти. Його перевагами для даного проєкту є простота конфігурації маршрутизації, легкість реалізації проміжного програмного забезпечення (middleware) для обробки запитів (наприклад, для автентифікації та валідації), що дозволяє швидко розгорнути надійну основу для серверної логіки.

2.2.2 Технології клієнтської частини

Для розробки клієнтської частини (фронтенду) веб-застосунку було обрано бібліотеку React у поєднанні з мовою програмування TypeScript та CSS-фреймворком Tailwind CSS.

React – це популярна JavaScript-бібліотека з відкритим вихідним кодом для створення користувацьких інтерфейсів. Її основною перевагою є компонентно-орієнтований підхід, який дозволяє розбивати складний інтерфейс на незалежні, перевикористовувані компоненти зі своїм станом та логікою. Для даного проєкту React було обрано завдяки його здатності ефективно управляти станом застосунку та забезпечувати динамічне оновлення інтерфейсу за допомогою віртуального DOM (Document Object Model), що сприяє високій продуктивності та плавності взаємодії користувача, особливо при роботі з багатоетапними формами налаштування та публікації контенту. Велика спільнота розробників та наявність численних бібліотек розширень також є важливими факторами, що спрощують та прискорюють розробку [5].

TypeScript – це мова програмування з відкритим вихідним кодом, розроблена компанією Microsoft, яка є строго типізованою надмножиною JavaScript та компілюється у звичайний JavaScript [6]. Включення TypeScript до технологічного стеку фронтенду веб-застосунку було зумовлене прагненням підвищити надійність кодової бази та покращити досвід розробки. Статична типізація дозволяє виявляти багато потенційних помилок ще на етапі написання коду та компіляції, а не під час виконання, що особливо важливо для складних застосунків. Також TypeScript полегшує рефакторинг, покращує читабельність коду та, завдяки статичній типізації, суттєво покращує якість автодоповнення та навігації по коду в інтегрованих середовищах розробки, що позитивно вплинуло на розробку користувацьких форм та логіки взаємодії з серверною частиною застосунку.

Tailwind CSS – це сучасний CSS-фреймворк, який надає великий набір готових, низькорівневих класів-утиліт, призначених для безпосереднього застосування в HTML-розмітці. Кожен такий клас відповідає за конкретну властивість стилю (наприклад, колір, відступ, розмір шрифту). Комбінуючи ці класи, розробник може швидко конструювати складні та унікальні дизайни без необхідності писати значні обсяги власного CSS-коду [7]. Tailwind CSS було обрано для даного веб-застосунку через його гнучкість та можливість швидкого прототипування і розробки користувацьких інтерфейсів.

2.2.3 Характеристика системи управління базами даних

Для зберігання даних веб-застосунку було обрано документо-орієнтовану систему управління базами даних MongoDB у поєднанні з бібліотекою Mongoose, що функціонує як інструмент об’єктно-даних моделей (Object Data Modeling) для Node.js.

MongoDB – це популярна NoSQL база даних, яка зберігає дані у гнучких, JSON-подібних документах. Такий підхід забезпечує значну гнучкість схеми, що є перевагою для проєктів, де структура даних може змінюватися або містити різнорідні елементи. Для даного веб-застосунку MongoDB було обрано через її здатність ефективно зберігати дані користувачів, їхні налаштування, інформацію про підключені соціальні мережі (включно з токенами доступу). Можливість роботи з вкладеними документами та масивами спрощує моделювання складних об’єктів, а також забезпечує високу продуктивність при читанні та записі даних, що часто зустрічаються в сучасних веб-додатках [8].

Незважаючи на широку популярність реляційних баз даних, таких як PostgreSQL або MySQL, для даного проєкту обрана була саме NoSQL документо-орієнтована база даних. Ключовим фактором такого вибору стала потреба у гнучкій схемі даних. Оскільки застосунок інтегрується з різними

соціальними мережами, кожна з яких може мати свою специфічну структуру даних для токенів доступу, налаштувань публікацій та метаданих контенту, гнучкість MongoDB у зберіганні JSON-подібних документів без жорстко визначеної наперед структури є суттєвою перевагою. Це спрощує розробку та подальше розширення функціоналу, пов'язаного з підтримкою нових платформ.

Mongoose – це бібліотека для Node.js, яка надає структурований підхід до моделювання даних застосунку при роботі з MongoDB. Вона дозволяє визначати схеми для колекцій документів, включаючи типи даних, валідацію, значення за замовчуванням, віртуальні властивості, проміжне програмне забезпечення (middleware) для хуків життєвого циклу документа та інше [9]. Використання Mongoose у даному проєкті значно спростило взаємодію з MongoDB, дозволивши працювати з даними на більш високому рівні абстракції, використовуючи об’єктно-орієнтований підхід.

2.2.4 Ключові допоміжні бібліотеки та інструменти

Для здійснення HTTP-запитів як на клієнтській, так і на серверній стороні застосунку було обрано бібліотеку Axios. Axios – це популярний HTTP-клієнт, що базується на Promise і надає простий та зручний інтерфейс для взаємодії з REST API. Ключовими перевагами Axios є можливість автоматичного перетворення JSON-даних, підтримка перехоплення запитів та відповідей для централізованої обробки помилок або додавання заголовків (наприклад, токенів автентифікації) [10].

Для обробки завантажуваних файлів (зокрема, відео та зображень) на серверній частині було використано проміжне програмне забезпечення (middleware) Multer. Multer спеціалізується на обробці даних типу multipart/form-data, який є стандартним для передачі файлів через HTTP-форми. Вибір Multer обґрунтований його простотою інтеграції з Express.js, гнучкістю

налаштувань (наприклад, для визначення директорії збереження файлів, обмежень на розмір та тип файлів), а також ефективністю при роботі з потоками даних, що дозволяє обробляти великі файли без надмірного споживання пам’яті сервера [11]. Це є важливим для функціоналу завантаження медіаконтенту в розроблюваному веб-застосунку.

2.3 Механізми автентифікації та авторизації

Забезпечення безпеки даних та контрольованого доступу до функціоналу є фундаментальною вимогою для сучасних веб-застосунків. Ключовими процесами, що реалізують ці аспекти безпеки, є автентифікація та авторизація. Автентифікація – це процедура встановлення та перевірки достовірності особи користувача або сутності, яка намагається отримати доступ до системи, зазвичай шляхом верифікації наданих користувачем облікових даних, що підтверджує заявлену ідентичність [12].

Авторизація, у свою чергу, є процесом надання або обмеження прав доступу вже автентифікованій сутності до конкретних ресурсів, операцій чи функцій системи на основі визначених політик безпеки та призначених привілеїв [12]. У контексті розроблюваного веб-застосунку для централізованої публікації контенту, ці механізми застосовуються як для визначення прав доступу користувачів до самої системи та її функціональних можливостей, так і для делегування застосунку повноважень діяти від імені користувача при взаємодії з API зовнішніх соціальних мереж. Для реалізації цих завдань використовуються сучасні підходи, зокрема технологія JSON Web Tokens (JWT) та протокол OAuth 2.0, які будуть розглянуті детальніше.

2.3.1 Використання JWT для автентифікації

Для реалізації механізму автентифікації зареєстрованих користувачів у розроблюваному веб-застосунку було обрано технологію JSON Web Tokens (JWT). JWT – це відкритий стандарт (RFC 7519), який визначає компактний та самодостатній спосіб безпечної передачі інформації між сторонами у вигляді JSON-об’єкта [13]. Ця інформація може бути перевірена та довірена, оскільки вона має цифровий підпис.

Структура JWT зазвичай складається з трьох частин, розділених крапками:

1. Заголовок (Header): Містить інформацію про тип токена та алгоритм підпису, що використовується (наприклад, HMAC SHA256 або RSA).

2. Корисне навантаження (Payload): Містить інформацію про сутність (зазвичай, користувача) та додаткові метадані. Існують стандартні твердження (наприклад, iss – видавець, exp – час закінчення терміну дії, sub – тема/ідентифікатор користувача), а також можуть бути додані власні.

3. Підпис (Signature): Використовується для перевірки того, що відправник токена є тим, за кого себе видає, і що повідомлення не було змінено в дорозі. Підпис створюється шляхом підписання закодованих заголовка та корисного навантаження за допомогою секретного ключа (для HMAC) або приватного ключа (для RSA) [13].

Використання JWT для автентифікації у веб-застосунках має низку суттєвих переваг. По-перше, це забезпечення відсутності стану на сервері, оскільки вся необхідна для автентифікації інформація міститься в самому токені, що спрощує масштабування системи. По-друге, JWT є компактними, що дозволяє легко передавати їх в HTTP-заголовках або тілі запиту. По-третє, механізм цифрового підпису гарантує безпеку, підтверджуючи цілісність та автентичність токена, а передача токенів через HTTPS захищає їх від

перехоплення. На відміну від традиційної сесійної автентифікації, JWT є самодостатніми токенами. Традиційна сесійна автентифікація зазвичай потребує зберігання стану сесії користувача на сервері, наприклад, у пам'яті сервера або в окремій базі даних. Гнучкість JWT дозволяє використовувати їх для автентифікації в різноманітних типах застосунків, включаючи веб, мобільні та API.

2.3.2 Протокол OAuth 2.0

Для забезпечення безпечного делегованого доступу веб-застосунку до ресурсів користувача, що зберігаються на сторонніх сервісах (зокрема, акаунтів у соціальних мережах), у проєкті використовується протокол авторизації OAuth 2.0. Це відкритий стандарт, який дозволяє користувачам надавати стороннім застосункам обмежений доступ до своїх ресурсів без передачі їм своїх облікових даних (логіна та пароля) [14].

Основна ідея OAuth 2.0 полягає у введенні додаткового рівня абстракції між клієнтом та сервером ресурсів. Протокол визначає декілька ключових ролей: власника ресурсу (користувача), клієнта (веб-застосунку), сервера авторизації та сервера ресурсів. Існує кілька типів потоків (grant types) авторизації, призначених для різних сценаріїв. У даному веб-застосунку для отримання доступу до API соціальних мереж переважно реалізовано потік «Authorization Code Grant» (надання коду авторизації), який вважається найбільш безпечним для серверних веб-застосунків.

Процес авторизації за допомогою потоку «Authorization Code Grant» у загальному вигляді виглядає наступним чином. Спочатку клієнтський застосунок перенаправляє користувача на сервер авторизації відповідної соціальної мережі. Користувач автентифікується на цьому сервері та надає згоду на доступ застосунку до запитуваних ресурсів, що визначаються через

вказані області дозволів (scopes) у запиті. У разі успішної згоди, сервер авторизації перенаправляє користувача назад до клієнтського застосунку, передаючи йому тимчасовий код авторизації. Для підвищення безпеки та запобігання атакам типу CSRF (Cross-Site Request Forgery) під час цього процесу використовується state-параметр, який генерується клієнтом, передається на сервер авторизації та повертається назад для перевірки відповідності. Отримавши код авторизації, серверна частина веб-застосунку обмінює його на сервері авторизації на токен доступу (access token) та, опціонально, на токен оновлення (refresh token). Токен доступу є короткоживучим і використовується для здійснення запитів до захищених ресурсів API від імені користувача. Токен оновлення є довгоживучим і може використовуватися для отримання нового токена доступу після закінчення терміну дії поточного, без необхідності повторної автентифікації користувача. Отримані токени безпечно зберігаються на серверній стороні застосунку, асоційовані з обліковим записом користувача.

2.4 Інтеграція з соціальними мережами у веб-застосунках

Функціонування сучасних веб-застосунків часто передбачає інтеграцію та обмін даними з іншими програмними системами, зовнішніми сервісами або внутрішніми компонентами. Для забезпечення такої структурованої взаємодії використовуються прикладні програмні інтерфейси (API).

2.4.1 Поняття прикладного програмного інтерфейсу

Прикладний програмний інтерфейс (Application Programming Interface, API) – це програмний інтерфейс застосунку, що надає розробникам набір визначених функцій та протоколів для взаємодії між різними програмними

компонентами або системами. Він виступає як контракт, що описує, які операції доступні, які дані необхідно надати для їх виконання та в якому форматі очікується відповідь, при цьому абстрагуючи від розробника деталі внутрішньої реалізації сервісу [15].

У контексті веб-сервісів та сучасних програмних систем загалом, API виконують низку ключових функцій:

1. Забезпечення взаємодії та інтеграції: API дозволяють різним застосункам та сервісам обмінюватися даними та спільно використовувати функціонал. Наприклад, веб-застосунки можуть інтегруватися зі сторонніми сервісами, такими як соціальні мережі, платіжні системи або картографічні сервіси, значно розширюючи свої можливості.

2. Абстракція та повторне використання: Надаючи чітко визначений інтерфейс, API дозволяють розробникам використовувати готовий функціонал, реалізований іншими системами (наприклад, операційною системою, браузером або стороннім веб-сервісом), без необхідності розробляти його з нуля. Це прискорює розробку та зменшує кількість коду.

3. Контроль доступу та безпека: API часто використовуються для управління доступом до програмних та апаратних ресурсів, забезпечуючи механізми авторизації та контролю дозволів перед виконанням запитуваних операцій.

4. Модульність та розмежування відповідальності: У веб-застосунках API допомагають чітко розділити логіку клієнтської частини (фронтенду) від серверної (бекенду), а також сприяють побудові модульних та мікросервісних архітектур.

Для розроблюваного веб-застосунку централізованої публікації контенту, прикладні програмні інтерфейси соціальних мереж (YouTube, TikTok, Facebook, Twitter) є фундаментальною основою. Саме через ці API здійснюється автентифікація користувачів у відповідних мережах та безпосередня публікація

контенту, що робить якість, стабільність та документацію цих інтерфейсів критично важливими для успішного функціонування системи.

2.4.2 Огляд YouTube Data API v3

YouTube Data API v3 є ключовим інструментом, що надається Google для програмної взаємодії з платформою YouTube. Цей прикладний програмний інтерфейс дозволяє розробникам інтегрувати функціональність YouTube у власні застосунки, керувати відеоконтентом, каналами, списками відтворення та отримувати аналітичні дані. API побудований на основі протоколу HTTP та зазвичай використовує JSON для обміну даними, дотримуючись REST-подібних принципів взаємодії [16].

Для виконання операцій від імені користувача, таких як завантаження відео або управління каналом, YouTube Data API вимагає авторизації через протокол OAuth 2.0. Цей механізм забезпечує безпечне делегування прав доступу, коли користувач надає застосунку дозвіл на виконання певних дій через визначені області доступу (scopes), не передаючи при цьому свої облікові дані. Після успішної авторизації застосунок отримує токен доступу, який використовується для автентифікації подальших запитів до API.

Серед основних можливостей, які надає YouTube Data API v3 і які є релевантними для систем централізованої публікації контенту, слід виділити функціонал для управління відео. API дозволяє програмно завантажувати відеофайли на платформу, вказуючи при цьому різноманітні метадані, такі як назва, опис, теги, категорія та налаштування приватності. Важливою функцією є можливість планування публікацій: API підтримує встановлення точної дати та часу, коли завантажене відео має автоматично стати загальнодоступним.

Окрім завантаження та публікації, YouTube Data API надає засоби для отримання інформації про канали користувача, списки його відео, їхній статус

та інші пов’язані дані. Для розробників на платформі Node.js взаємодію з API може спрощувати використання офіційної клієнтської бібліотеки googleapis, яка інкапсулює частину логіки запитів та автентифікації. Слід також враховувати, що використання YouTube Data API регулюється системою квот, яка обмежує кількість запитів, що може зробити застосунок за певний проміжок часу, з метою запобігання надмірному навантаженню на сервіс.

2.4.3 Огляд TikTok API

Платформа TikTok for Developers надає розробникам інструментарій для інтеграції застосунків із соціальною мережею TikTok, що включає API для взаємодії з контентом та функціоналом платформи. Для використання цих можливостей необхідно зареєструвати застосунок на порталі розробників та отримати відповідні облікові дані для доступу до API. Взаємодія з API TikTok здійснюється за протоколом HTTP, а авторизація застосунків для виконання дій від імені користувачів базується на стандарті OAuth 2.0. Це передбачає отримання згоди користувача на доступ до певних областей даних та функцій (scopes), після чого застосунку надається токен доступу [17].

API TikTok пропонує програмні засоби для завантаження відео на платформу. Залежно від типу інтеграції та рівня доступу застосунку, API надає різні механізми публікації. Одним з підходів є пряма публікація (Direct Post), що дозволяє розміщувати відео безпосередньо в профілі користувача з програмним керуванням усіма параметрами, такий тип доступу вимагає додаткової верифікації застосунку. Інший підхід передбачає завантаження контенту до проміжного стану, наприклад, у «Вхідні» (Inbox) або чернетки користувача в мобільному додатку TikTok, звідки користувач вже власноруч завершує процес публікації.

Для цілей розробки та тестування інтеграцій, TikTok for Developers надає середовище Sandbox (пісочниця). Це середовище дозволяє розробникам безпечно експериментувати з API в ізольованих умовах без впливу на реальні дані та без необхідності проходження верифікації, яка необхідна для роботи в повноцінному production-середовищі. Важливо враховувати, що функціонал, доступний у Sandbox, іноді має певні обмеження порівняно з можливостями повністю верифікованого застосунку.

2.4.4 Огляд Facebook Graph API

Для програмної взаємодії з платформою Facebook та її функціоналом, зокрема для публікації контенту, розробники використовують Facebook Graph API. Graph API є основним способом отримання даних з Facebook та їх розміщення на платформі. Він представляє об’єкти Facebook (такі як користувачі, сторінки, пости, фото, відео) та зв’язки між ними у вигляді графової структури, доступ до якої здійснюється через HTTP-запити [18]. Як і більшість сучасних веб-API, Graph API дотримується REST-подібних принципів та використовує JSON для обміну даними.

Авторизація застосунків для роботи з Graph API від імені користувача базується на протоколі OAuth 2.0. Перед тим, як застосунок зможе виконувати будь-які дії або отримувати доступ до даних, користувач повинен надати йому відповідні дозволи (permissions).

Facebook Graph API надає широкі можливості для публікації різноманітного контенту, особливо на Сторінках (Facebook Pages), якими керує користувач, оскільки публікація на особисті профілі через API неможлива з міркувань безпеки. Через API можна публікувати текстові пости, пости із зображеннями або посиланнями, а також завантажувати та публікувати відео.

2.4.5 Огляд X API

Для програмної взаємодії з платформою Twitter (нині X) розробникам надається X API. Цей інтерфейс дозволяє застосункам отримувати доступ до даних Twitter, аналізувати їх, а також, публікувати контент від імені користувачів. Авторизація застосунків для роботи з X API від імені користувача, як і в інших сучасних веб-API, здійснюється за допомогою протоколу OAuth 2.0 [19].

Сучасний Twitter API функціонує на основі різних рівнів доступу, які визначають доступний функціонал та ліміти на кількість запитів. Існує безкоштовний рівень (Free tier), який надає обмежений доступ до API. Зокрема, для операцій запису, таких як публікація твітів, безкоштовний рівень має дуже суттєве обмеження на кількість запитів на місяць, що робить його придатним переважно для тестування або застосунків з дуже низькою активністю публікацій. Для більш інтенсивного використання та доступу до розширених можливостей і вищих лімітів існують платні рівні, такі як Basic та Pro [19].

При роботі з X API необхідно уважно вивчати актуальну документацію щодо доступних ендпоінтів, лімітів запитів для обраного рівня доступу та політик використання платформи, оскільки вони можуть зазнавати змін.

ВИСНОВОК ДО РОЗДІЛУ 2

У другому розділі було проведено огляд та обґрунтування вибору технологій, що використовуються для проєктування та реалізації веб-застосунку централізованої публікації контенту. Розглянуто загальні принципи побудови сучасних веб-застосунків, зокрема клієнт-серверну архітектуру та її основні компоненти.

Для серверної частини було обрано платформу Node.js з фреймворком Express.js завдяки їхній ефективності в обробці асинхронних операцій та гнучкості у створенні REST API. Клієнтська частина реалізується з використанням бібліотеки React та мови TypeScript для забезпечення інтерактивного та надійного користувацького інтерфейсу, а стилізація здійснюється за допомогою CSS-фреймворку Tailwind CSS. Для зберігання даних застосунку було обрано документо-орієнтовану СУБД MongoDB з ODM Mongoose, що забезпечує гнучкість схеми та ефективну роботу з даними. Для HTTP-взаємодії та обробки файлових завантажень обґрунтовано використання бібліотек Axios та Multer відповідно.

Також було детально розглянуто ключові механізми автентифікації та авторизації: технологію JWT для автентифікації користувачів у системі та протокол OAuth 2.0 для безпечного делегованого доступу до API соціальних мереж. Проаналізовано основні можливості та принципи роботи API платформ YouTube, TikTok, Facebook та Twitter, які є фундаментальними для реалізації основного функціоналу розроблюваного застосунку.

Вибір зазначеного комплексу технологій та підходів є обґрунтованим з огляду на поставлені функціональні та нефункціональні вимоги до системи, дозволяючи створити сучасний, масштабований та безпечний веб-застосунок.

РОЗДІЛ 3. ДЕТАЛІ РОЗРОБКИ ВЕБ-ЗАСТОСУНКУ

3.1 Архітектура системи

3.1.1 Загальна архітектурна схема веб-застосунку

Розроблений веб-застосунок для централізованої публікації контенту спроєктовано за клієнт-серверною архітектурною моделлю. Ця модель передбачає логічний поділ системи на три основні взаємодіючі компоненти: лієнтську частину (фронтенд), серверну частину (бекенд) та базу даних, які разом забезпечують взаємодію з користувачем та зовнішніми сервісами.

Основний потік взаємодії в системі виглядає наступним чином:

1. Користувач взаємодіє з клієнтською частиною через веб-браузер.
2. Клієнтська частина надсилає запити (наприклад, на створення поста, підключення акаунта) до серверної частини через REST API, використовуючи захищене з'єднання (HTTPS) та JWT для автентифікації.
3. Серверний частина обробляє запит: валідує дані, взаємодіє з базою даних для читання/запису інформації, та, за необхідності, звертається до API відповідної соціальної мережі.
4. Зовнішні API соціальних мереж обробляють запити від серверної частини та повертають результат (наприклад, підтвердження публікації або помилку).
5. Серверна частина аналізує відповідь від зовнішнього API, формує фінальну відповідь та надсилає її назад клієнтській частині.
6. Клієнтська частина отримує відповідь від сервера та відображає результат операції користувачеві.

Клієнт-серверна архітектура з чітким розмежуванням відповідальності між фронтендом, бекендом та базою даних є критично важливою для даного проєкту. Вона дозволяє незалежно розробляти, тестувати та модифікувати окремі компоненти системи, що особливо актуально при роботі з різними та постійно змінюваними API соціальних мереж. Такий підхід не лише спрощує розробку, але й закладає фундамент для майбутнього масштабування – наприклад, при додаванні підтримки нових соціальних платформ або розширенні функціоналу, зміни в одному компоненті мінімально впливатимуть на інші.

3.1.2 Архітектурні особливості клієнтської частини

Клієнтський застосунок є точкою входу для користувача в систему та відповідає за представлення графічного інтерфейсу, обробку користувацьких дій та ініціювання запитів до серверної частини. Архітектурно він реалізований як односторінковий застосунок, що забезпечує високу швидкість відгуку та плавне оновлення інтерфейсу без повного перезавантаження веб-сторінки.

Ключовими архітектурними рішеннями на клієнтській стороні є використання React Context API для ефективного управління станом React застосунку та організація навігації, що забезпечує інтуїтивну взаємодію користувача з різними функціональними блоками. Основними функціональними обов’язками клієнта є надання інтерфейсів для автентифікації користувачів, створення та налаштування контенту для публікації, управління підключеними акаунтами соціальних мереж та відображення актуального статусу публікацій.

3.1.3 Архітектурні особливості серверної частини

Серверний застосунок (бекенд) є ядром системи, що містить основну бізнес-логіку та слугує посередником між клієнтською частиною, базою даних та зовнішніми API соціальних мереж. Його головне завдання – надання клієнту стандартизованого та безпечного REST API для обміну даними та виконання операцій.

Архітектурно, сервер інкапсулює всю логіку роботи з API соціальних мереж, включаючи процеси авторизації за протоколом OAuth 2.0 та формування запитів на публікацію. Важливим проектним рішенням стало делегування функції планування публікацій безпосередньо механізмам, що надаються API самих соціальних мереж. Це дозволило суттєво спростити внутрішню логіку сервера та уникнути необхідності реалізації власного складного планувальника завдань. Сервер також відповідає за безпечну обробку файлових завантажень від користувача, які потім передаються до API відповідних платформ для публікації.

3.1.4 Архітектура бази даних

База даних, реалізована на основі MongoDB, слугує для зберігання ключової інформації, необхідної для функціонування веб-застосунку. Основними сутностями, що моделюються, є користувачі системи та їхні токени доступу до соціальних мереж.

Колекція User зберігає облікові дані користувачів (ідентифікатор Google, електронна пошта, ім’я, посилання на зображення профілю), токени автентифікації через Google, роль користувача та службові позначки часу.

Колекція Token відповідає за зберігання токенів доступу до API соціальних мереж (YouTube, TikTok, Facebook, Twitter). Кожен документ у цій

колекції пов’язаний з конкретним користувачем через поле userId та містить accessToken, refreshToken, час дії токена, назву платформи та інші специфічні для платформи дані. Для забезпечення унікальності підключення використовується складений індекс (один токен на користувача для кожної платформи). Архітектура передбачає механізм автоматичного оновлення токенів доступу.

Інформація про конкретні пости та їхній статус після запиту на публікацію не зберігається в базі даних на постійній основі. Медіафайли передаються напряму до API соцмереж, а планування покладається на їхні власні можливості. Такий підхід до архітектури бази даних дозволяє спростити її структуру, зосередившись на управлінні даними користувачів та їхніх сесій доступу.

3.2 Реалізація серверної частини

3.2.1 Організація структури модулів серверної частини

Для побудови серверної частини, розробленої на платформі Node.js з використанням фреймворку Express.js, було обрано структурований підхід, що базується на принципах розділення відповідальностей, близьких до архітектурного шаблону MVC (Model-View-Controller), але в контексті REST API, «View» фактично замінюється логікою формування HTTP-відповіді. Така організація дозволяє чітко розмежувати логіку маршрутизації, обробки запитів, виконання бізнес-операцій та взаємодії з базою даних. Це сприяє кращій читабельності коду, спрощує його тестування та подальшу підтримку, що є важливим для системи з потенціалом до розширення.

Головний файл серверного застосунку server.js слугує точкою входу для серверної логіки (рис. 3.1). Цей файл відповідає за ініціалізацію ключових

аспектів роботи сервера, включаючи: завантаження конфігурацій з оточення (змінних середовища), встановлення з'єднання з базою даних MongoDB, підключення файлів маршрутизації для обробки API-запитів та запуск HTTP-сервера на прослуховування визначеного порту.

```
app.use('/auth', authRoutes);
app.use('/api/youtube', youtubeRoutes);
app.use('/api/tiktok', tiktokRoutes);
app.use('/api/users', userRoutes);
app.use('/api/facebook', facebookRoutes);
app.use('/api/twitter', twitterRoutes);

async function startServer() {
  try {
    await connectToDatabase();
    app.listen(port, host, () => {
      console.log(`Backend server running on http://${host}:${port}`);
    });
  } catch (error) {
    console.error('Failed to start server:', error);
    process.exit(1);
  }
}

startServer();
```

Рисунок 3.1 – Ключові аспекти запуску сервера у файлі server.js

Структура директорій серверного проєкту організована наступним чином для логічного групування функціональності:

- routes/: містить файли, що визначають маршрути (ендпоінти) API. Кожен файл відповідає за певну функціональну групу (наприклад, authRoutes.js для процесів автентифікації та авторизації в системі та з соціальними мережами, userRoutes.js для управління даними користувачів) або за операції, специфічні для конкретної соціальної платформи (наприклад, youtubeRoutes.js). Маршрути визначаються за допомогою express.Router() і пов'язуються з функціями-контролерами.
- controllers/: містить контролери, відповідальні за обробку HTTP-запитів, що надходять на визначені маршрути. Основні функції контролерів: отримання та валідація даних із запиту, виклик відповідних сервісних модулів для виконання бізнес-логіки, обробка

результатів, повернутих сервісами, та формування HTTP-відповіді клієнту (включаючи коректні статуси та тіло відповіді).

- services/: у цій директорії розміщені сервісні модулі, які інкапсулюють специфічну бізнес-логіку, зокрема пов'язану з OAuth-авторизацією та управлінням токенами для різних соціальних мереж. Сервіси абстрагують логіку взаємодії з зовнішніми API та базою даних (наприклад, збереження та оновлення токенів доступу), надаючи контролерам більш високорівневий інтерфейс.
- models/: містить визначення Mongoose-схем для колекцій бази даних MongoDB. А саме моделі User (для даних користувачів системи) та Token (для токенів доступу до соціальних мереж).
- lib/: ця директорія призначена для допоміжних модулів та утиліт загального призначення. Зокрема, тут розташований модуль database.js, відповідальний за встановлення та управління з'єднанням з MongoDB, та модуль jwt.js, що надає функції для генерації та верифікації JSON Web Tokens, а також реалізує проміжне програмне забезпечення для автентифікації запитів.

3.2.2 Проєктування та реалізація бази даних

Для реалізації бази даних використовується MongoDB у поєднанні з бібліотекою Mongoose для Node.js. Такий вибір забезпечує гнучкість схеми даних та зручний інтерфейс для моделювання та взаємодії з базою. Архітектура бази даних розроблена з акцентом на мінімалізм та ефективність, зосереджуючись на управлінні користувацькими даними та токенами доступу. Зберігання контенту та логіка планування публікацій делеговані безпосередньо API соціальних мереж, що дозволяє спростити структуру власної бази даних.

Основними сутностями, що моделюються в базі даних, є «Користувач» (User) та «Токен» (Token).

Модель даних «Користувач» (User) призначена для зберігання інформації про зареєстрованих користувачів системи, які проходять автентифікацію через сервіс Google OAuth. Схема Mongoose для цієї моделі включає такі поля:

- googleId (Тип: String): Ідентифікатор користувача, отриманий від Google; поле проіндексовано для оптимізації пошуку.
- email (Тип: String): Електронна адреса користувача.
- name (Тип: String): Повне ім’я користувача.
- profilePicture (Тип: String): URL-адреса зображення профілю користувача.
- role (Тип: String): Роль користувача в системі (‘user’ або ‘admin’).
- accessToken (Тип: String): Токен доступу Google OAuth, отриманий під час автентифікації.
- refreshToken (Тип: String): Токен оновлення Google OAuth.
- expiryTime (Тип: Number): Час дії accessToken від Google.
- lastLogin, createdAt, updatedAt (Date): Службові позначки часу.

Модель даних «Токен» (Token) використовується для зберігання OAuth-токенів доступу, які надають веб-застосунку дозвіл діяти від імені користувача при взаємодії з API різних соціальних мереж. Схема Mongoose для цієї моделі включає такі поля:

- userId (Тип: ObjectId): Посилання на ідентифікатор користувача з колекції User, що встановлює зв’язок між токеном та його власником.
- platform (Тип: String): Назва соціальної мережі (наприклад, ‘youtube’, ‘tiktok’, ‘facebook’, ‘twitter’), для якої призначений токен.
- accessToken (Тип: String): Токен доступу до API платформи.
- refreshToken (Тип: String): Токен оновлення, що використовується для отримання нового accessToken.

- expiryTime (Тип: Number): Час, коли accessToken стає недійсним.
- platformUserId (Тип: String): Ідентифікатор користувача на відповідній соціальній платформі.
- createdAt, updatedAt (Тип: Date): Поля для відстеження часу створення та оновлення запису токена.

Взаємодія з даними та реалізація основних операцій в системі здійснюється через моделі Mongoose у відповідних сервісних модулях.

При першій автентифікації користувача через Google система виконує пошук існуючого користувача за googleId. Якщо користувач не знайдений, створюється новий запис у колекції User з даними, отриманими від Google. В іншому випадку може оновлюватися існуюча інформація (наприклад, токени або час останнього входу). Ця логіка інкапсульована в модулі userAuthService.js (рис. 3.2).

```
async handleGoogleCallback(code) {
  try {
    const { tokens } = await this.userAuthClient.getToken(code);
    this.userAuthClient.setCredentials(tokens);

    const oauth2 = google.oauth2({ version: 'v2', auth: this.userAuthClient });
    const { data } = await oauth2.userinfo.get();

    let user = await User.findOne({ googleId: data.id });

    if (user) {
      user.name = data.name;
      user.firstName = data.given_name;
      user.lastName = data.family_name;
      user.profilePicture = data.picture;
      user.accessToken = tokens.access_token;
      user.lastLogin = new Date();

      if (tokens.refresh_token) {
        user.refreshToken = tokens.refresh_token;
      }

      if (tokens.expiry_date) {
        user.expiryTime = tokens.expiry_date;
      }
    } else {
      user = new User({···
      });
    }

    await user.save();
```

Рисунок 3.2 – Фрагмент коду для обробки даних при Google-автентифікації

При підключенні нового акаунту соціальної мережі або при оновленні токенів використовується підхід «знайти або створити/оновити» («upsert»). Система шукає запис у колекції Token для пари userId та platform. Існуючий запис оновлюється, новий – створюється.

Перед виконанням запиту до API соціальної мережі, система отримує відповідний токен з бази даних за userId та platform. Для цього реалізовано механізм перевірки терміну дії accessToken та його автоматичного оновлення за допомогою refreshToken, якщо токен застарів і платформа підтримує такий механізм (рис. 3.3). Це забезпечує безперервність доступу до API від імені користувача.

```
async ensureValidToken(userId) {
  await this.setOAuthCredentials(userId);

  const currentTokens = await this.getStoredTokens(userId);
  if (!currentTokens.accessToken || !currentTokens.refreshToken) {
    throw new Error('User not authenticated with YouTube or refresh token missing.');
  }

  const isTokenExpired = !currentTokens.expiryTime || Date.now() >= currentTokens.expiryTime;

  if (isTokenExpired) {
    try {
      const { credentials } = await this.oauth2Client.refreshAccessToken();

      await this.storeTokens(userId, {
        accessToken: credentials.access_token,
        refreshToken: credentials.refresh_token || currentTokens.refreshToken,
        expiryTime: credentials.expiry_date
      });

      this.oauth2Client.setCredentials(credentials);

    } catch (refreshError) {
      await this.clearTokens(userId);
      this.oauth2Client.setCredentials({});
      throw new Error('Failed to refresh YouTube token. Please re-authenticate.');
    }
  }

  if (!this.oauth2Client.credentials.access_token) {
    throw new Error('Missing access token after check/refresh.');
  }

  return this.oauth2Client;
}
```

Рисунок 3.3 – Фрагмент коду для валідації та оновлення токена для YouTube

3.2.3 Реалізація API ендпоінтів та логіки обробки запитів

Серверна частина веб-застосунку надає клієнту стандартизований REST API для всіх взаємодій. Ендпоінти API згруповані за функціональним призначенням у відповідних файлах маршрутизації (authRoutes.js, userRoutes.js, youtubeRoutes.js тощо) та підключаються до основного застосунку Express з узгодженими префіксами. Кожен маршрут пов'язаний з відповідним методом контролера, який відповідає за первинну обробку запиту та виклик необхідних сервісів (рис. 3.4).

```
const express = require('express');
const multer = require('multer');
const youtubeController = require('../controllers/youtubeController');

const router = express.Router();

const storage = multer.memoryStorage();
const upload = multer({ storage: storage });

router.get('/status', youtubeController.getStatus);
router.get('/channel', youtubeController.getChannel);
router.post('/upload', upload.single('videoFile'), youtubeController.uploadVideo);

module.exports = router;
```

Рисунок 3.4 – Файл маршрутизації для Youtube

Первинна автентифікація користувачів у системі реалізована через маршрути, визначені у файлі userRoutes.js та доступні за префіксом /api/users. Система використовує Google OAuth 2.0 як основний метод автентифікації.

GET /api/users/login/google – ініціює процес OAuth 2.0 автентифікації з Google. Ендпоінт не вимагає попередньої авторизації та повертає клієнту URL-адресу для перенаправлення користувача на сторінку авторизації Google. Відповідь надається у форматі JSON з полем url, що містить згенеровану адресу.

GET /api/users/login/google/callback – обробляє зворотний виклик від Google після завершення процесу авторизації користувачем. Ендпоінт отримує параметри code (код авторизації) та state через query string запиту. Контролер

делегує обробку userAuthService, який виконує обмін коду на токени доступу, пошук або створення користувача в системі та генерацію JWT токену для подальших сесій.

Для кожної підтримуваної соціальної мережі передбачено набір ендпоінтів для підключення акаунту до профілю користувача в системі та для його відключення (наприклад, /api/auth/youtube або /api/auth/tiktok). Кожна соціальна мережа має відповідний callback-ендпоінт для обробки результату авторизації. Ці ендпоінти отримують параметри code та state через query string та делегують обробку відповідним сервісам для завершення OAuth-процесу. Вихід з підключених акаунтів здійснюється через POST-запити на відповідні ендпоінти logout. Вони видаляють збережені токени доступу для відповідної платформи з бази даних, фактично «відключаючи» акаунт від веб-застосунку.

Для безпосередньої взаємодії з контентом на підключених соціальних платформах використовуються платформо-специфічні набори ендпоінтів, реалізовані у файлах youtubeRoutes.js, tiktokRoutes.js, facebookRoutes.js та twitterRoutes.js. Операції на цих маршрутах, що передбачають дії від імені користувача (наприклад, отримання інформації про профіль, публікація контенту), захищені JWT-автентифікацією.

Кожна платформа надає ендпоінти для перевірки стану підключення та отримання базової інформації:

- GET /api/platform/status – перевірка активності підключення до платформи.
- GET /api/platform/userinfo – отримання інформації про користувацький профіль.
- GET /api/youtube/channel – отримання даних про канал користувача (для YouTube).
- GET /api/facebook/pages – отримання списку керованих користувачем Facebook сторінок.

Основний функціонал публікації контенту реалізований через POST-ендпоінти, які приймають запити у форматі multipart/form-data для підтримки завантаження медіафайлів.

- Для відеоплатформ YouTube та TikTok використовуються відповідно маршрути POST /api/youtube/upload та POST /api/tiktok/upload.
- Facebook підтримує публікацію відео (POST /api/facebook/upload), текстових постів (POST /api/facebook/post) та зображень (POST /api/facebook/photo).
- Для Twitter реалізовано окремий ендпоінт POST /api/twitter/upload для публікації контенту, що може включати текст та медіа.

Усі POST-ендпоінти для публікації приймають файл контенту разом із набором метаданих (назва, опис, параметри видимості, дата публікації тощо), специфічних для вимог API кожної соціальної мережі. Успішна відповідь повертається у форматі JSON та містить інформацію про результат операції. Різноманітність ендпоінтів та їх структури обумовлена специфікою API кожної платформи.

Вся логіка взаємодії зі сторонніми API, включаючи формування запитів, обробку відповідей та помилок, інкапсульована на серверній стороні в відповідних сервісних модулях, що спрощує розробку клієнтської частини.

3.2.4 Реалізація взаємодії сервісних модулів з API соціальних мереж

Для забезпечення можливості взаємодії розробленого веб-застосунку з API соціальних мереж, було виконано необхідні підготовчі етапи. Для кожної з підтримуваних платформ – YouTube, TikTok, Facebook та Twitter – було здійснено реєстрацію додатка на порталі для розробників (наприклад, Google Cloud Console, TikTok for Developers, Facebook for Developers, X Developer Portal). У процесі реєстрації та налаштування цих додатків було визначено

необхідні параметри OAuth 2.0 авторизації, такі як дозволені URI перенаправлення (callback URLs) та запитувані області доступу (scopes), що відповідають функціоналу застосунку (наприклад, публікація контенту, доступ до інформації профілю). В результаті було отримано унікальні ідентифікатори клієнта та секретні ключі, які були безпечно інтегровані в конфігурацію серверної частини застосунку (через змінні середовища) і використовуються для автентифікації запитів до OAuth-серверів та API відповідних соціальних мереж.

Після того, як API ендпоінти серверної частини отримали та первинно обробили запит від клієнта, відповідні сервісні модулі (наприклад, youtubeAuthService.js, tiktokService.js тощо) беруть на себе відповідальність за безпосередню комунікацію з прикладними програмними інтерфейсами соціальних мереж. Цей процес включає автентифікацію запитів за допомогою OAuth 2.0 токенів, коректне формування HTTP-запитів згідно з вимогами кожної платформи та обробку отриманих відповідей.

Автентифікація запитів до зовнішніх API є першим кроком при будь-якій взаємодії. Перед виконанням запиту до API соціальної мережі, сервісний модуль отримує з бази даних актуальний accessToken для поточного користувача та цільової платформ. Функція ensureValidToken перевіряє час дії (expiryTime) поточного токена і якщо він застарів або відсутній, ініціюється запит на оновлення за допомогою refreshToken (рис. 3.5). Отримана нова пара токенів зберігається в базі даних, замінюючи попередні. Валідний accessToken завжди включається до заголовку Authorization усіх подальших HTTP-запитів до API відповідної соціальної платформи.

```
async ensureValidToken(userId) {
  await this.setOAuthCredentials(userId);

  const currentTokens = await this.getStoredTokens(userId);
  if (!currentTokens.accessToken || !currentTokens.refreshToken) {
    throw new Error('User not authenticated with YouTube or refresh token missing.');
  }

  const isTokenExpired = !currentTokens.expiryTime || Date.now() >= currentTokens.expiryTime;

  if (isTokenExpired) {
    try {
      const { credentials } = await this.oauth2Client.refreshAccessToken();

      await this.storeTokens(userId, {
        accessToken: credentials.access_token,
        refreshToken: credentials.refresh_token || currentTokens.refreshToken,
        expiryTime: credentials.expiry_date
      });

      this.oauth2Client.setCredentials(credentials);

    } catch (refreshError) {
      await this.clearTokens(userId);
      this.oauth2Client.setCredentials({});
      throw new Error('Failed to refresh YouTube token. Please re-authenticate.');
    }
  }
```

Рисунок 3.5 – Механізм перевірки та оновлення токена для YouTube

Підходи до формування запитів до зовнішніх API варіюються залежно від платформи. Для взаємодії з YouTube API використовується офіційна клієнтська бібліотека googleapis, яка абстрагує деталі формування HTTP-запитів та спрощує роботу з різними сервісами Google. Це дозволяє абстрагуватися від низькорівневих деталей HTTP-протоколу та використовувати зручні методи, надані бібліотекою, для формування запитів та обробки відповідей. Для інших платформ – TikTok, Facebook та Twitter – взаємодія реалізована через прямі HTTP-запити до відповідних ендпоінтів API цих платформ, сформовані та відправлені за допомогою бібліотеки Axios. Вона надає гнучкий та простий інтерфейс для здійснення HTTP-запитів, управління заголовками, тілом запиту та обробки відповідей, включаючи помилки.

При формуванні запитів для публікації контенту, сервісні модулі трансформують дані, отримані від контролера, у формат, що вимагається конкретним API. Наприклад, для публікації відео на YouTube через бібліотеку

googleapis, у метод youtube.videos.insert передається як сам відеофайл (з буфера Multer у вигляді медіаресурсу), так і об’єкт з метаданими, де вказуються назва, опис для публікації, та інші налаштування (рис. 3.6).

```
const bufferStream = new stream.PassThrough();
bufferStream.end(req.file.buffer);

const youtube = google.youtube({ version: 'v3', auth: authenticatedClient });

const youtubeResponse = await youtube.videos.insert({
    part: 'snippet,status',
    requestBody: metadata,
    media: {
      mimeType: req.file.mimetype,
      body: bufferStream,
    },
  });
```

Рисунок 3.6 – Формування запиту для публікації відео на YouTube

Для платформ, де використовуються прямі HTTP-запити, наприклад, при завантаженні відео на TikTok або Facebook, формується multipart/form-data запит, що містить відеофайл та відповідні поля метаданих. Текстові пости на Facebook чи Twitter створюються шляхом відправки JSON або даних у тілі POST-запиту (рис. 3.7). Особливу увагу приділено специфіці кожної платформи: наприклад, при публікації на Facebook необхідно вказувати ID цільової Сторінки, а завантаження медіа на Twitter X API v2 є багатоетапним процесом (INIT, APPEND, FINALIZE), який реалізовано у відповідному сервісі.

```
uploadVideo = async (req, res) => {
  const userId = req.user.userId;
  const { title, description, selectedPageId } = req.body;

  if (!this.validateRequiredFields(req, res, ['file', 'selectedPageId'])) {
    return;
  }

  try {
    const pages = await this.service.getPages(userId);
    const page = pages.find(p => p.id === selectedPageId);
    if (!page || !page.access_token) {
      return res.status(403).json({ error: 'Not authenticated or missing page access token' });
    }
    const pageToken = page.access_token;

    const form = new FormData();
    form.append('source', req.file.buffer, { filename: req.file.originalname, contentType: req.file.mimetype });
    if (title) form.append('title', title);
    if (description) form.append('description', description);
    form.append('access_token', pageToken);

    const fbRes = await axios.post(
      FACEBOOK_API.VIDEO(selectedPageId),
      form,
      { headers: form.getHeaders() }
    );

    const videoId = fbRes.data.id;
    const videoUrl = FACEBOOK_API.VIDEO_URL(selectedPageId, videoId);
    res.json({ videoId, videoUrl });
  } catch (error) {···
  }
};
```

Рисунок 3.7 – Формування запиту для публікації відео на Facebook

Обробка відповідей та помилок від зовнішніх API є невід'ємною частиною взаємодії. Сервісні модулі аналізують відповіді, отримані від API соціальних мереж. У випадку успішного виконання операції, з тіла відповіді витягується корисна інформація, така як ідентифікатор опублікованого поста або оновлені дані профілю, яка потім передається контролеру. При виникненні помилок, система класифікує їх на основі коду стану HTTP та тіла відповіді. Усі значущі помилки логуються на сервері для подальшого аналізу. Клієнтській частині веб-застосунку повертаються узагальнені повідомлення про помилки з відповідними кодами стану HTTP, щоб забезпечити зрозумілий зворотний зв'язок користувачеві.

3.2.5 Механізми безпеки серверної частини

Забезпечення безпеки серверної частини є критично важливим аспектом розробки веб-застосунку, що обробляє дані користувачів та взаємодіє з зовнішніми сервісами. У даному проєкті реалізовано низку механізмів, спрямованих на захист API, надійну автентифікацію користувачів та безпечне управління конфігураційними даними.

Основним механізмом автентифікації користувачів для доступу до захищених ресурсів API веб-застосунку є використання JWT (JSON Web Tokens), для роботи з якими застосовується бібліотека jsonwebtoken. Після успішної автентифікації користувача, система генерує два типи токенів: короткоживучий accessToken (термін дії – 7 днів), що містить у своєму корисному навантаженні (payload) ідентифікатор користувача, його електронну адресу та роль, та довгоживучий refreshToken (термін дії – 30 днів), призначений для безпечного оновлення accessToken. Обидва токени підписуються за допомогою секретного ключа, що зберігається у змінних середовища. Для перевірки автентичності запитів реалізовано проміжне програмне забезпечення authenticateToken в модулі lib/jwt.js (рис. 3.8). Цей middleware вилучає accessToken з HTTP-заголовка Authorization, верифікує його підпис та термін дії за допомогою секретного ключа. У випадку успішної верифікації, дані з payload токена додаються до об’єкта запиту. При невалідному, відсутньому або простроченому токені, middleware повертає клієнту відповідь з кодом стану HTTP 401 (Unauthorized).

```
exports.verifyToken = (token) => {
  try {
    return jwt.verify(token, JWT_SECRET);
  } catch (error) {
    return null;
  }
};

exports.authenticateToken = (req, res, next) => {
  const authHeader = req.headers['authorization'];

  if (!authHeader?.startsWith('Bearer ')) {
    return res.status(401).json({ error: 'Unauthorized: Missing or invalid token format' });
  }

  const token = authHeader.split(' ')[1];
  const decoded = this.verifyToken(token);
  if (!decoded) {
    return res.status(401).json({ error: 'Unauthorized: Invalid or expired token' });
  }

  req.user = decoded;
  next();
};
```

Рисунок 3.8 – Middleware для перевірки JWT accessToken

Безпека конфігураційних даних забезпечується шляхом зберігання всіх чутливих параметрів, таких як секретний ключ для JWT, облікові дані клієнтських OAuth-додатків та URI для підключення до бази даних, у змінних середовища на сервері. Для їх завантаження в застосунок на етапі ініціалізації сервера використовується бібліотека dotenv, що автоматично зчитує файл .env та робить змінні доступними через process.env у інших модулях.

Для контролю міждоменних запитів та забезпечення безпечної взаємодії між фронтенд- та бекенд-частинами налаштовано механізм CORS (Cross-Origin Resource Sharing). Відповідне middleware, сконфігуроване в головному файлі сервера, дозволяє приймати запити лише з визначеного URL фронтенд-застосунку та підтримує передачу облікових даних, що важливо для коректної роботи з JWT.

3.2.6 Обробка завантаження медіафайлів

Однією з ключових функцій розробленого веб-застосунку є можливість завантаження користувачами медіафайлів (відео, фото) для їх подальшої публікації в соціальних мережах. Для обробки таких файлових завантажень на серверній стороні використовується проміжне програмне забезпечення Multer, яке спеціалізується на обробці запитів у форматі multipart/form-data.

У проєкті Multer ініціалізується з опцією storage: multer.memoryStorage(). Це означає, що завантажені файли зберігаються не на диску сервера, а безпосередньо в оперативній пам'яті у вигляді буфера. Такий підхід має переваги у швидкості обробки, оскільки виключаються операції читання/запису на диск, та спрощує архітектуру, усуваючи необхідність управління тимчасовими файлами на сервері. Дані файлу можуть бути передані напряму до API соціальних мереж.

Для контролю за використанням ресурсів сервера та відповідності вимогам API соціальних мереж, при ініціалізації Multer встановлюються обмеження на максимальний розмір файлу (рис. 3.9). Ці ліміти були визначені на основі офіційної документації та обмежень, що накладаються кожною конкретною платформою для завантаження медіаконтенту.

```
const multer = require('multer');
const router = express.Router();

const storage = multer.memoryStorage();
const upload = multer({ storage, limits: { fileSize: 512 * 1024 * 1024 } }); // 512 MB
```

Рисунок 3.9 – Ініціалізація Multer в файлі маршрутів

Сконфігурований екземпляр Multer інтегрується як middleware у відповідні маршрути Express.js, що призначені для завантаження контенту. Для цього використовується метод .single(fieldName).

Після того, як Multer обробив запит, контролер отримує доступ до інформації про завантажений файл через об’єкт req.file. У контролерах обов’язково виконується перевірка наявності req.file та його буфера перед подальшою обробкою.

З об’єкта req.file використовуються такі ключові властивості:

- buffer: Містить дані файлу у вигляді об’єкта Buffer, що є основним для передачі до API соціальних мереж.
- Originalname: Оригінальна назва файлу, завантаженого користувачем.
- Mimetype: MIME-тип файлу (наприклад, ‘video/mp4’, ‘image/jpeg’).
- Size: Розмір файлу в байтах.

Отриманий буфер файлу (req.file.buffer) разом з іншими метаданими передається з контролера у відповідний сервісний модуль для подальшого формування запиту до API цільової соціальної мережі. Спосіб подальшої обробки цього буфера та його включення до запиту до зовнішнього API залежить від специфіки кожної платформи та реалізований у відповідних сервісних модулях. Це може включати, наприклад, конвертацію буфера у потік (stream), додавання до об’єкта FormData або використання спеціалізованих методів клієнтських бібліотек.

Обмеження memoryStorage() полягає в тому, що при одночасному завантаженні багатьох великих файлів може виникнути значне споживання оперативної пам’яті сервера. Однак, встановлені ліміти на розмір файлів на рівні Multer допомагають мінімізувати цей ризик. У випадку, якщо файл перевищує встановлений ліміт, Multer генерує помилку, яка обробляється стандартними механізмами Express.js.

3.3 Реалізація клієнтської частини

Клієнтська частина веб-застосунку розроблена як односторінковий застосунок з використанням бібліотеки React та мови TypeScript. Для стилізації інтерфейсу застосовується CSS-фреймворк Tailwind CSS, а інструментарій Vite слугує для організації процесу розробки та збірки проєкту.

3.3.1 Організація структури модулів клієнтської частини

Структура програмних модулів у каталозі src/ організована для забезпечення модульності та легкості підтримки коду:

- main.tsx: точка входу React-застосунку, відповідальна за рендеринг кореневого компонента та підключення глобальних стилів.
- App.tsx: головний компонент, що керує загальною структурою інтерфейсу та основними етапами взаємодії користувача.
- types.ts: містить централізовані визначення TypeScript типів та інтерфейсів.
- index.css: визначає глобальні CSS-стилі та інтеграцію з Tailwind CSS.
- components/: містить перевикористовувані UI-компоненти, з можливою внутрішньою організацією за функціональністю (наприклад, піддиректорії post/ для компонентів створення постів та settings/ для компонентів налаштувань платформ). Тут розміщені такі елементи, як Header.tsx, PlatformSelection.tsx, VideoUploadForm.tsx тощо.
- context/: реалізації React Context API для глобального управління станом, зокрема UserAuthContext.tsx для автентифікації та VideoContext.tsx для стану завантаження відео.

- hooks/: React хуки для інкапсуляції перевикористовуваної бізнес-логіки, включаючи хуки для автентифікації з платформами та для операцій з контентом.
- services/: модулі для взаємодії з серверним API, що інкапсулюють логіку HTTP-запитів.
- data/: статичні дані та конфігураційна інформація, наприклад, platforms.ts з описом підтримуваних соціальних мереж.

3.3.2 Розробка ключових сторінок та компонентів інтерфейсу

Інтерфейс клієнтської частини спроєктований для забезпечення покрокового та інтуїтивно зрозумілого процесу створення та публікації контенту. Цей процес координується головним компонентом App.tsx і реалізується через набір спеціалізованих React-компонентів, що представляють окремі етапи та UI-елементи.

Початковим є екран вибору типу контенту, представлений компонентом ContentTypeSelection.tsx, де користувач визначає, чи буде він публікувати відео або текстовий пост. Цей вибір визначає подальший потік взаємодії.

У випадку обрання відео-контенту, користувач послідовно проходить три основні етапи:

1. Перший етап передбачає завантаження відеофайлу через компонент VideoUploadForm.tsx – спеціальну форму, яка підтримує функцію drag-and-drop та надає можливість попереднього перегляду.
2. На другому етапі, за допомогою компонента PlatformSelection.tsx, користувач обирає одну або декілька соціальних платформ для публікації. Тут же користувач може підключити аккаунт потрібної соціальної мережі.

3. Завершальним етапом є конфігурація параметрів відео (назви, опису, тегів, видимості, дати планування) для кожної обраної платформи, що реалізовано у компоненті VideoSettings.tsx.

Для роботи з текстовими постами використовується комплексний компонент PostFlow.tsx. Він має власний багатоетапний процес, що включає вибір платформ (Facebook або Twitter) та заповнення форм контенту, специфічних для кожної соціальної мережі.

Стилізація всіх компонентів та сторінок здійснюється за допомогою Tailwind CSS, що забезпечує швидку розробку, гнучкість та адаптивність дизайну. Іконки з бібліотеки lucide-react використовуються для покращення візуальної навігації та зрозумілості елементів керування.

3.3.3 Керування станом та організація навігації

Для забезпечення якісного користувацького досвіду в веб-застосунку, особливу увагу було приділено ефективному управлінню станом та реалізації інтуїтивної навігації. Ці аспекти втілені з використанням вбудованих можливостей React, таких як Context API для глобального стану, хук useState для локального стану компонентів, та логіки умовного рендерингу для навігації між екранами.

React Context API використовується для даних, які є спільними для різних частин застосунку. Визначено два основні глобальні контексти:

1. UserAuthContext.tsx: призначений для централізованого управління станом автентифікації користувача. Контекст інкапсулює об’єкт користувача, індикатори стану автентифікації (isAuthenticated) та процесу завантаження даних. Він надає абстрактні методи для ініціації процесу входу в та виходу в систему, а також для перевірки статусу автентифікації при завантаженні застосунку (рис. 3.10) .

```
const UserAuthContext = createContext<UserAuthContextType>({
  user: null,
  isAuthenticated: false,
  isLoading: true,
  login: () => {},
  logout: () => {},
});

export const UserAuthProvider = ({ children }: { children: ReactNode }) => {
  const [user, setUser] = useState<User | null>(null);
  const [isLoading, setIsLoading] = useState(true);
```

Рисунок 3.10 – Ініціалізація контексту та визначення стану

2. VideoContext.tsx: призначений для управління станом, пов'язаним із процесом завантаження відео. Він зберігає обраний користувачем відеофайл (videoFile), статус процесу завантаження (isUploading) та його прогрес (uploadProgress). Контекст надає відповідні функції-сетери для маніпулювання цими даними (рис. 3.11).

```
interface VideoContextType {
  videoFile: File | null;
  setVideoFile: (file: File | null) => void;
  isUploading: boolean;
  setIsUploading: (isUploading: boolean) => void;
  uploadProgress: number;
  setUploadProgress: (progress: number) => void;
}
```

Рисунок 3.11 – Глобальний контекст для стану завантаження відео

Для стану, що є специфічним для окремих UI-компонентів (наприклад, значення полів вводу у формах), використовується стандартний React-хук useState. Це дозволяє зберігати компоненти незалежними та контролювати їхню поведінку локально, наприклад, у формах VideoUploadForm.tsx або FacebookPostForm.tsx для управління даними, що вводяться користувачем.

Навігація між основними етапами та функціональними блоками застосунку реалізована на рівні головного компонента App.tsx за допомогою умовного рендерингу, керованого його станом (змінні contentType та

currentStep). Перехід між станами ініціюється викликом відповідних функцій-обробників з дочірніх компонентів, що призводить до рендерингу відповідних UI-елементів (рис. 3.12).

```
<main className="flex-1 py-6 mb-8">
  <div className="mx-auto max-w-2xl px-4">
    {contentType === null && <ContentTypeSelection onSelect={type => type === 'video' ? startVideo() : startPost()} />}
    {contentType === 'post' && <PostFlow />}

    {contentType === 'video' && (
      <div className="space-y-6">
        {currentStep === 1 && (
          <VideoUploadForm onSubmit={handleVideoSubmit} />
        )}
        {currentStep === 2 && (
          <PlatformSelection
            platforms={platforms}
            selectedPlatforms={selectedPlatforms}
            onSelect={handlePlatformSelect}
            onDeselect={handlePlatformDeselect}
            onNext={handlePlatformsSubmit}
            isActive={true}
            onBack={currentStep > 1 ? goBack : undefined}
          />
        )}
        {currentStep === 3 && (
          <VideoSettings selectedPlatforms={selectedPlatforms} isActive={true} onBack={goBack} />
        )}
      </div>
```

Рисунок 3.12 – Логіка керування етапами створення контенту

Компонент PostFlow.tsx, що відповідає за створення текстових постів, також має власну внутрішню логіку навігації між етапами вибору платформ (‘select’) та безпосереднього створення контенту (‘compose’), а також між вкладками для різних соціальних мереж, використовуючи для цього локальний стан (activePlatform) та компоненти табуляції (рис. 3.13).

```
const PostFlow: React.FC = () => {
  const postPlatforms = platformsList.filter(p => p.id === 'facebook' || p.id === 'twitter');
  const [step, setStep] = useState<'select'|'compose'>('select');
  const [selectedPlatforms, setSelectedPlatforms] = useState<SelectedPlatform[]>([]);
  const { publishSingle, results } = usePostPublisher();
  const [activePlatform, setActivePlatform] = useState<string | null>(null);
  const handleSelect = (platformId: string, isAuthenticated?: boolean) => {
```

Рисунок 3.13 – Ініціалізація змінних стану у компоненті PostFlow

3.3.4 Взаємодія з серверним API на клієнті

Ефективна взаємодія клієнтської частини з серверним API є основою для отримання даних, виконання операцій та забезпечення автентифікованого доступу. React-застосунок здійснює запити до бекенду, обробляє відповіді, працює з JWT для автентифікації сесій, а також реалізує клієнтську частину OAuth 2.0 потоку для підключення акаунтів соціальних мереж.

Для HTTP-запитів до серверного API у проєкті використовується бібліотека Axios. Вона застосовується як у контексті UserAuthContext.tsx та кастомних хуках (наприклад, usePlatformAuth.ts) для автентифікаційних процесів, так і у всіх сервісних модулях, що інкапсулюють взаємодію з API для конкретних соціальних платформ. Сервісні функції та хуки формують HTTP-заголовки, включаючи Authorization з JWT-токеном для захищених ендпоінтів, та обробляють відповіді сервера, інформуючи користувача про результати через react-hot-toast.

Процес первинної автентифікації користувача через Google ініціюється клієнтською функцією login() (рис. 3.14). Ця функція безпосередньо відкриває у новому спливаючому вікні URL-адресу серверного ендпоінту за допомогою window.open().

```
const login = () => {
  const width = 600;
  const height = 700;
  const left = window.innerWidth / 2 - width / 2;
  const top = window.innerHeight / 2 - height / 2;

  window.open(
    `${import.meta.env.VITE_API_BASE_URL}/api/users/login/google`,
    'Google Login',
    `width=${width},height=${height},left=${left},top=${top}`
  );
};
```

Рисунок 3.14 – Функція для початку процесу автентифікації через Google

Після завершення авторизації на стороні Google, бекенд обробляє callback і передає результат (системні accessToken, refreshToken та дані користувача) в основне вікно застосунку через window.postMessage(). Клієнтська частина прослуховує ці повідомлення, зберігає отримані системні токени в localStorage для підтримки сесії та оновлює стан автентифікації в UserAuthContext (рис. 3.15). При кожному завантаженні застосунку, accessToken витягується з localStorage і додається до заголовків запитів для доступу до захищених ресурсів.

```
useEffect(() => {
  const handleAuthMessage = (event: MessageEvent) => {
    if (!event.data || !event.data.type) return;

    if (event.data.type === 'USER_AUTH_SUCCESS') {
      localStorage.setItem('accessToken', event.data.accessToken);
      localStorage.setItem('refreshToken', event.data.refreshToken);
      setUser(event.data.user);
      toast.success('Signed in successfully!');
    } else if (event.data.type === 'USER_AUTH_ERROR') {
      toast.error(`Login failed: ${event.data.error || 'Unknown error'}`);
    }
  };

  window.addEventListener('message', handleAuthMessage);
  return () => window.removeEventListener('message', handleAuthMessage);
}, []);
```

Рисунок 3.15 – Реалізація обробки повідомлень від OAuth popup-вікна

Аналогічний механізм відкриття URL-адреси серверного ендпоінту у спливаючому вікні через window.open() використовується для підключення акаунтів соціальних мереж. Кастомні хуки (наприклад, useTikTokAuth()) ініціюють цей процес, а результат підключення (успіх/помилка) отримують через window.postMessage(), оновлюючи відповідний стан на клієнті та автоматично закриваючи спливаюче вікно.

ВИСНОВОК ДО РОЗДІЛУ 3

У третьому розділі було здійснено проєктування та реалізацію веб-застосунку для централізованої публікації контенту в соціальних мережах. Було розроблено загальну архітектуру системи, що включає клієнтську та серверну частини, а також спроєктовано архітектуру бази даних, які разом забезпечують ефективне функціонування та взаємодію компонентів застосунку.

У рамках реалізації серверної частини було організовано структуру програмних модулів, спроєктовано та реалізовано базу даних MongoDB, а також розроблено REST API, включаючи ключові ендпоінти та логіку їх обробки. Було реалізовано механізми взаємодії сервісних модулів з API соціальних мереж, що охоплюють управління OAuth 2.0 токенами, формування запитів для публікації контенту та обробку відповідей. Окрему увагу було приділено реалізації механізмів безпеки серверної частини, зокрема використанню JWT для автентифікації, та розробці процесу обробки завантаження медіафайлів за допомогою бібліотеки Multer.

Клієнтську частину було розроблено як односторінковий застосунок на базі React та TypeScript. У ході реалізації було структуровано програмні модулі, розроблено ключові сторінки та компоненти користувацького інтерфейсу, впроваджено підходи до управління станом за допомогою React Context API та організовано навігацію. Також було реалізовано механізми взаємодії клієнтської частини з серверним API та клієнтську логіку OAuth 2.0 авторизації для підключення акаунтів соціальних мереж.

РОЗДІЛ 4. ТЕСТУВАННЯ ТА АНАЛІЗ РОЗРОБЛЕНОГО ПРОЄКТУ

4.1 Демонстрація роботи та функціональне тестування веб-застосунку

Для підтвердження коректності роботи створеного програмного продукту та наочної ілюстрації його основного функціоналу покроково розглянуто основні процеси взаємодії користувача з системою. Цей процес також слугуватиме для функціонального тестування ключових аспектів системи.

4.1.1 Реєстрація та автентифікація користувача

При першому відвідуванні веб-застосунку користувач бачить головний екран, де йому пропонується обрати тип контенту для створення. У верхній частині сторінки розташована кнопка для ініціації входу в систему через обліковий запис Google (рис. 4.1).

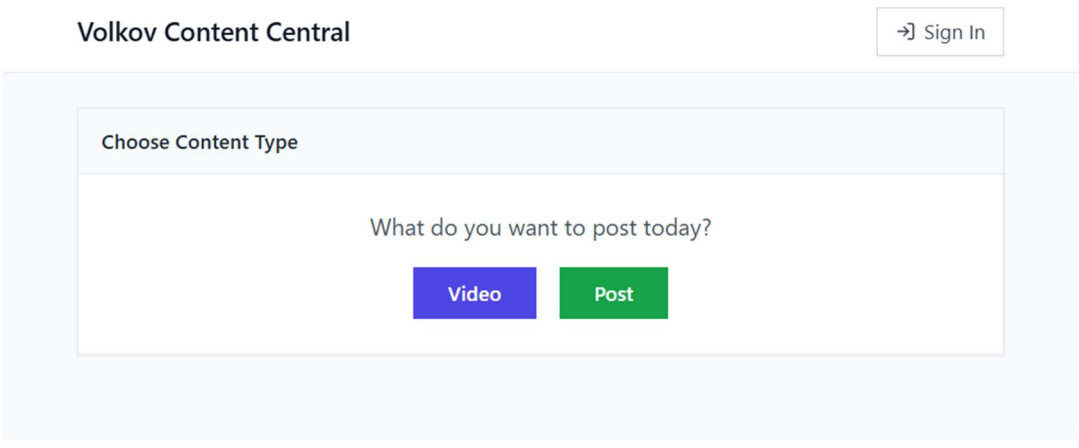


Рисунок 4.1 – Початковий екран для неавтентифікованого користувача

Після натискання користувачем кнопки входу, ініціюється процес автентифікації через Google OAuth. Застосунок відкриває спливаюче вікно, де користувач проходить стандартну процедуру вибору акаунту Google та надання необхідних дозволів на базову інформацію про користувача.

У випадку успішного завершення автентифікації на стороні Google та коректної обробки зворотного виклику серверною частиною, користувач залишається на тому ж головному екрані застосунку. Про успішний вхід свідчить оновлення інтерфейсу у заголовку сторінки: кнопка входу замінюється на елемент, що відображає профіль користувача. Додатково, користувач отримує відповідне сповіщення про успішну автентифікацію (рис. 4.2).

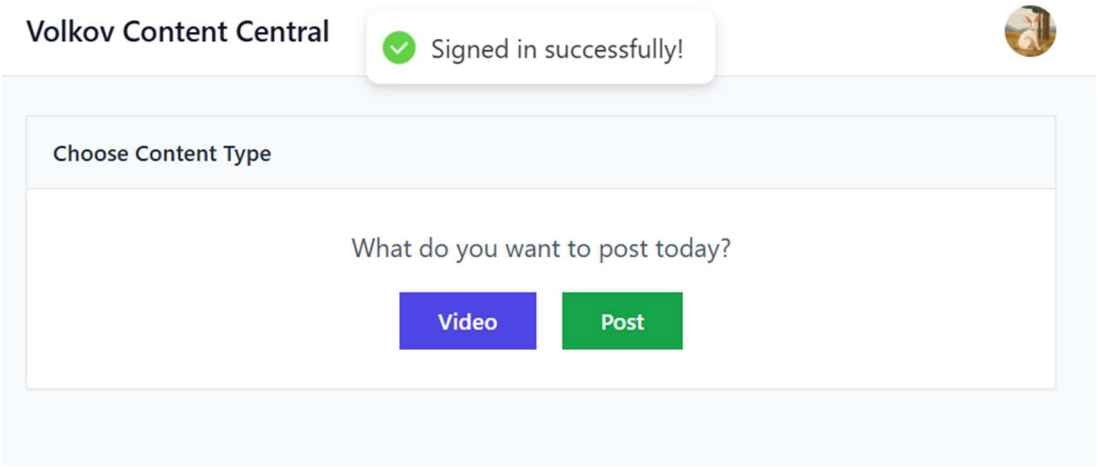


Рисунок 4.2 – Вигляд сторінки після успішної автентифікації користувача

4.1.2 Налаштування та публікація відео

Після автентифікації та вибору типу контенту «Відео» на головному екрані, користувач переходить до багатоетапного процесу створення відео-публікації. Спочатку відбувається завантаження відеофайлу. Для цього в інтерфейсі передбачено спеціальну форму, що підтримує як перетягування файлу, так і його вибір через стандартний діалог операційної системи (рис. 4.3).

Після вибору файлу користувач бачить його назву, розмір та область для перегляду.

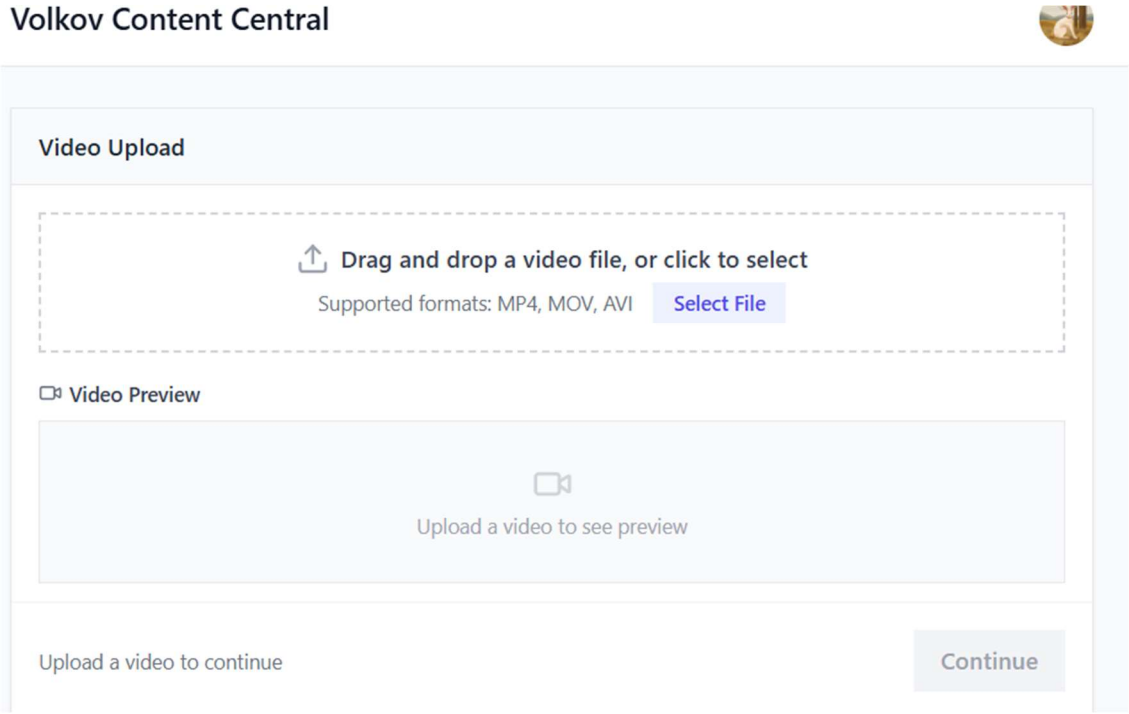


Рисунок 4.3 – Інтерфейс завантаження відеофайлу

Наступним кроком є вибір соціальних платформ для публікації. Користувач потрапляє на екран, де представлено список доступних платформ (рис. 4.4). Тут він може обрати одну або декілька соціальних мереж.

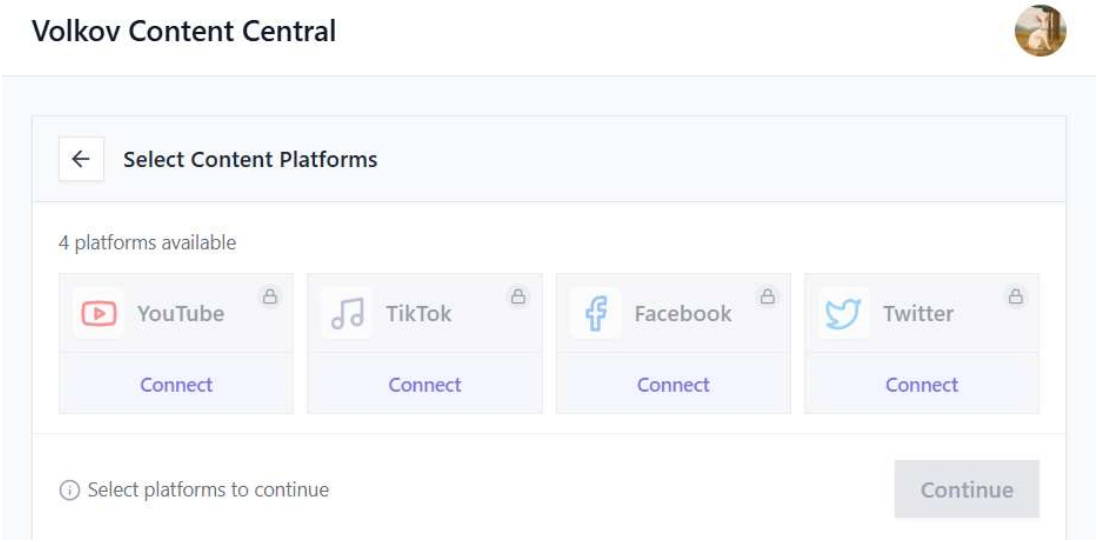


Рисунок 4.4 – Інтерфейс вибору соціальних платформ та підключення акаунтів

Якщо для обраної платформи акаунт ще не підключено, користувач натискає кнопку "Підключити". Система ініціює OAuth 2.0 потік авторизації: відкривається спливаюче вікно, яке перенаправляє користувача на сторінку авторизації відповідної соціальної мережі (наприклад, Google для YouTube). Для кожної платформи відкривається її власний, наданий API, інтерфейс авторизації, де користувач надає веб-застосунку необхідні дозволи. Після успішного надання дозволів та обробки даних сервером, спливаюче вікно закривається, а інтерфейс на екрані вибору платформ оновлюється (рис. 4.5).

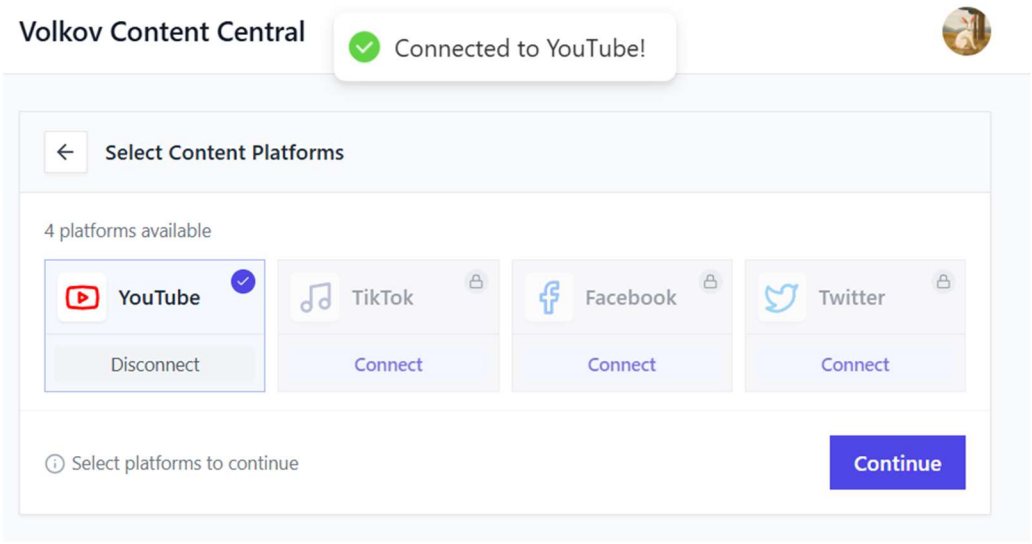


Рисунок 4.5 – Відображення успішно підключеного акаунту YouTube

Після вибору платформ та успішного підключення необхідних акаунтів, користувач переходить до фінального етапу – налаштування параметрів публікації. Інтерфейс цього етапу містить вкладки для кожної обраної соціальної мережі, що дозволяє індивідуально конфігурувати метадані, такі як назва відео, опис та теги (рис. 4.6). Також користувач може обрати налаштування видимості та, у випадку запланованої публікації, вказати точну дату та час публікації контенту. Передбачена можливість копіювання налаштувань з однієї платформи на інші.

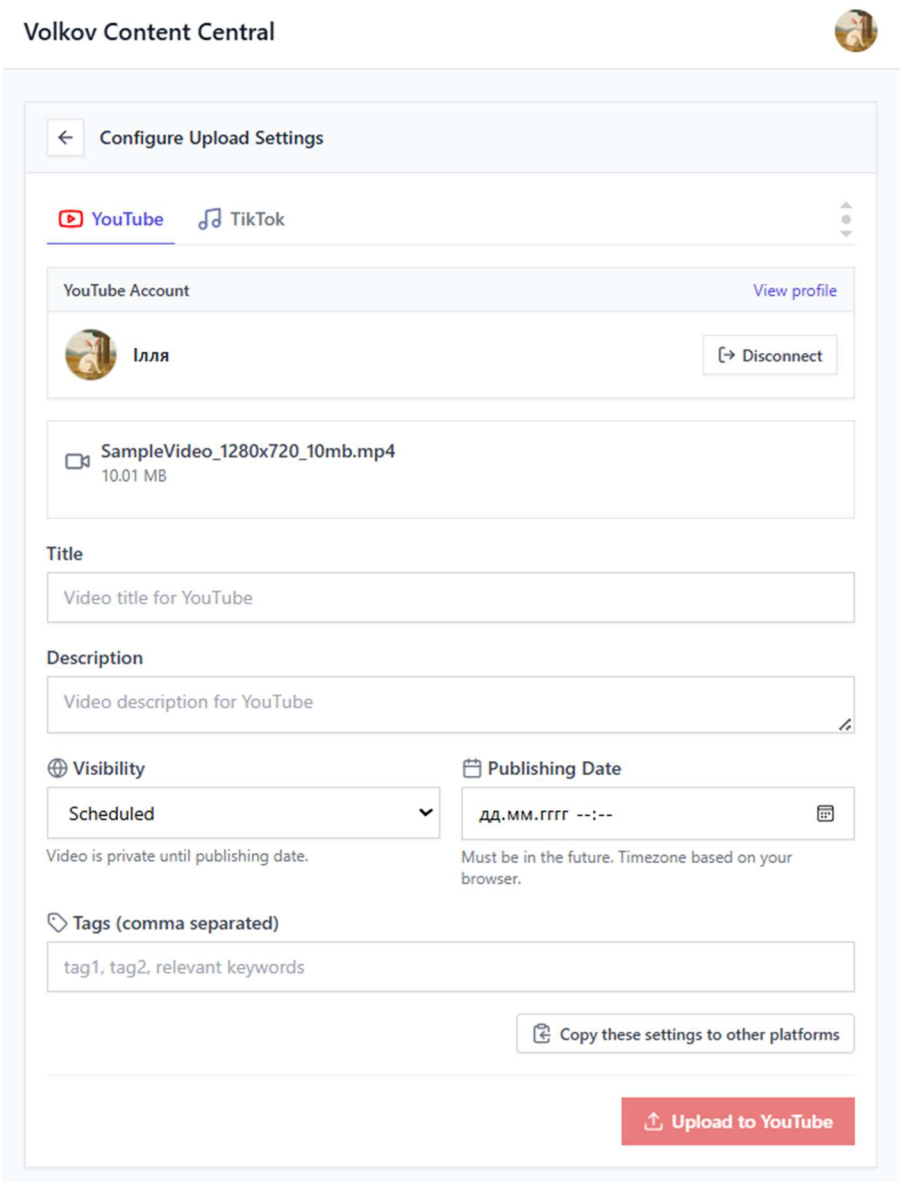


Рисунок 4.6 – Інтерфейс налаштування параметрів відео-публікації

Для запуску процесу публікації або планування для кожної налаштованої платформи передбачена окрема кнопка. Після її натискання відображається візуальний індикатор процесу завантаження відео. У випадку успішного завантаження відображається посилання на опублікований контент (рис. 4.7). На кожному етапі створення відео-публікації користувач має можливість повернутися до попереднього кроку.

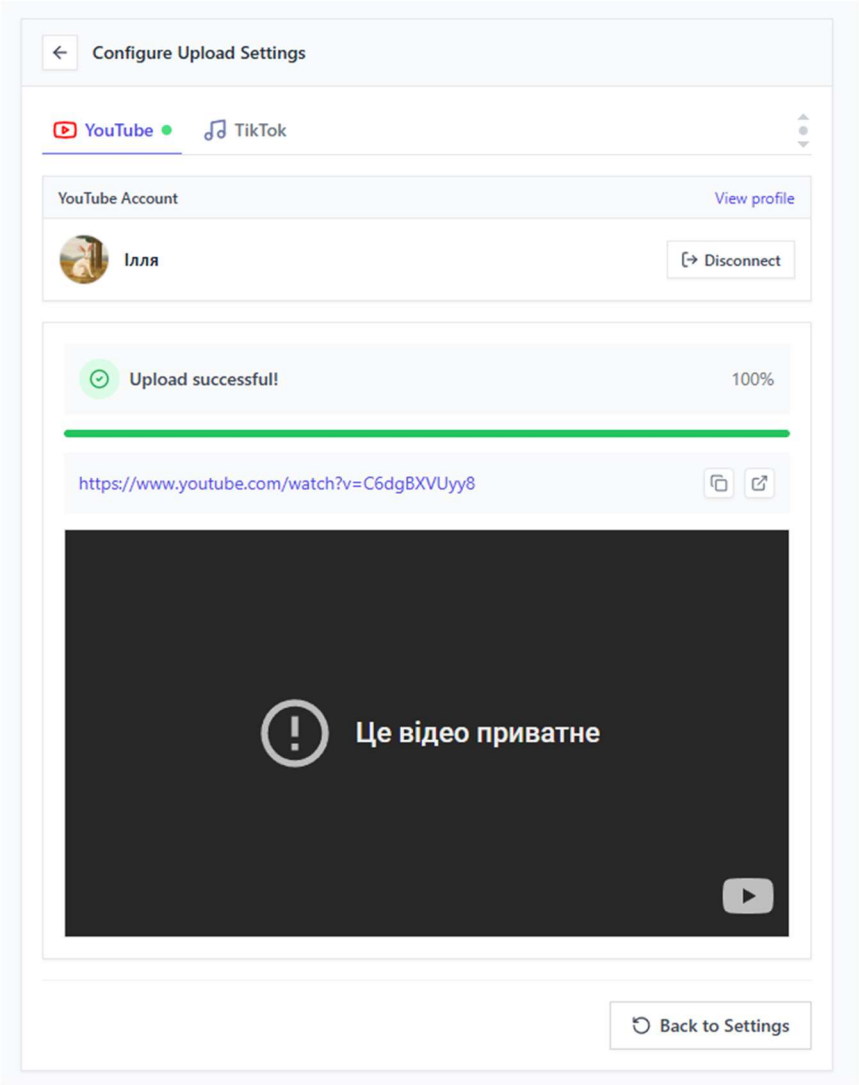


Рисунок 4.7 – Відображення результату успішної публікації відео

4.1.3 Створення текстової публікації

Веб-застосунок також надає користувачам можливість створювати та публікувати текстові пости. Після вибору відповідного типу контенту «Пост» на головному екрані, користувач переходить до екрану вибору соціальних платформ, доступних для текстових публікацій.

Після вибору однієї або декількох платформ та натискання кнопки для продовження, користувач потрапляє на екран налаштування текстового поста. Цей екран містить вкладки для кожної обраної соціальної мережі (рис. 4.8).

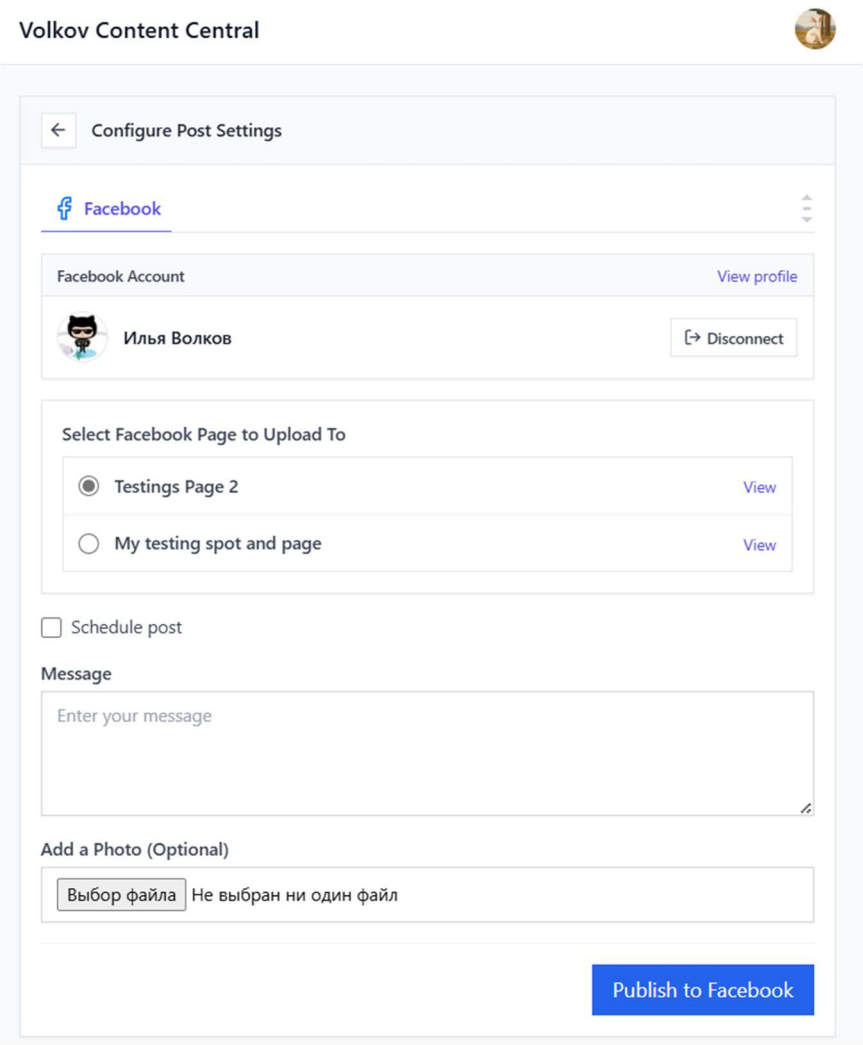


Рисунок 4.8 – Інтерфейс налаштування параметрів текстової публікації

Для кожної платформи користувач вводить текст повідомлення. Опціонально можна додати зображення.

Після налаштування всіх параметрів, користувач ініціює публікацію за допомогою відповідної кнопки «Опублікувати». Система обробляє запит, відправляючи дані на сервер для подальшої взаємодії з API соціальних мереж. Користувач отримує інформацію про статус публікації для кожної обраної платформи.

ВИСНОВОК ДО РОЗДІЛУ 4

У четвертому розділі було проведено демонстрацію роботи та функціональне тестування розробленого веб-застосунку для централізованої публікації контенту. Було розглянуто та перевірено ключові користувацькі сценарії, включаючи реєстрацію та автентифікацію користувачів, підключення акаунтів соціальних мереж, а також процеси створення, налаштування та публікації відео- та текстового контенту.

Проведене тестування підтвердили коректну реалізацію основного функціоналу застосунку. Механізми автентифікації та авторизації працюють належним чином, забезпечуючи безпечний доступ та управління підключеними акаунтами. Система успішно обробляє завантаження медіафайлів та передає дані для публікації на обрані соціальні платформи, враховуючи специфічні налаштування для кожної з них. На основі проведеного огляду та тестування можна зробити висновок, що розроблений веб-застосунок функціонує відповідно до поставлених завдань.

ВИСНОВКИ

У ході виконання дипломної роботи було успішно розроблено веб-застосунок, призначений для централізованої публікації контенту в популярних соціальних мережах. Метою роботи було створення інструменту, що надасть користувачам єдиний інтерфейс для ефективного завантаження, налаштування та планування публікації відео- та текстового контенту через офіційні API соціальних мереж.

У першому розділі було проведено аналіз актуальності проблеми управління контентом у соціальних мережах та огляд існуючих програмних рішень. На основі порівняння аналогів та виявлених потреб користувачів було сформульовано детальні функціональні та нефункціональні вимоги до розроблюваного веб-застосунку. Ключовими вимогами стали підтримка інтеграції з YouTube, TikTok, Facebook та Twitter, можливість публікації відео- та текстових постів, включення механізму планування, забезпечення безпечноъ автентифікації та авторизації користувачів через OAuth 2.0, а також створення простого та інтуїтивно зрозумілого інтерфейсу.

Другий розділ було присвячено обґрунтуванню вибору технологічного стеку та розгляду теоретичних аспектів розробки. Було обрано клієнт-серверну архітектуру з реалізацією серверної частини на Node.js та Express.js, клієнтської – на React та TypeScript, а для зберігання даних – документо-орієнтовану СУБД MongoDB. Детально розглянуто механізми автентифікації на базі JWT, протокол OAuth 2.0 для взаємодії з API соціальних мереж, а також проаналізовано специфіку роботи з API кожної з обраних платформ.

У третьому розділі було детально описано процес проєктування та реалізації веб-застосунку. Розглянуто загальну архітектуру системи, спроєктовано структуру бази даних та реалізовано моделі даних. Було

розроблено REST API серверної частини, включаючи ендпоінти для автентифікації користувачів, управління підключеними акаунтами соціальних мереж та публікації контенту. Описано механізми взаємодії сервісних модулів з API соціальних мереж, реалізацію безпеки на сервері та обробку завантаження медіафайлів. Також було представлено реалізацію клієнтської частини, структуру її програмних модулів, ключові компоненти та сторінки, управління станом та навігацію, а також взаємодію з серверним API.

У четвертому розділі було проведено демонстрацію роботи та функціональне тестування розробленого веб-застосунку. Шляхом покрокового відтворення основних користувацьких сценаріїв, таких як реєстрація, підключення акаунтів соціальних мереж, створення та планування відео- і текстових публікацій, було підтверджено працездатність ключового функціоналу системи.

У результаті виконання дипломної роботи було розроблено функціональний веб-застосунок, що надає користувачам інструмент для централізованого управління публікаціями в соціальних мережах та відповідає усім поставленим вимогам. Розроблена система має потенціал для подальшого вдосконалення та розширення функціональних можливостей.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. The 10+ Best Social Media Publishing Tools and Their Benefits [Електронний ресурс] // Social Champ Blog – 2024. – Режим доступу до ресурсу: https://www.socialchamp.com/blog/social-media-publishing-tools/
2. 9 Best Hootsuite Alternatives [Електронний ресурс] // HopperHQ – 2025. – Режим доступу до ресурсу: https://www.hopperhq.com/blog/hootsuite-alternatives/
3. Client-server overview [Електронний ресурс] // MDN Web Docs – 2025. – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/First_steps/
4. Про Node.js [Електронний ресурс] // Node.js – 2025. – Режим доступу до ресурсу: https://nodejs.org/uk/about/
5. React [Електронний ресурс] // React – 2025. – Режим доступу до ресурсу: https://react.dev/
6. TypeScript [Електронний ресурс] // Вікіпедія – 2025. – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/TypeScript
7. Tailwind CSS [Електронний ресурс] // Tailwind Labs – 2025. – Режим доступу до ресурсу: https://tailwindcss.com/
8. MongoDB [Електронний ресурс] // MongoDB, Inc. – 2025. – Режим доступу до ресурсу: https://www.mongodb.com/
9. Mongoose ODM [Електронний ресурс] // MongooseJS – 2025. – Режим доступу до ресурсу: https://mongoosejs.com/
10. Axios Introduction [Електронний ресурс] // Axios – 2025. – Режим доступу до ресурсу: https://axios-http.com/docs/intro
11. Multer [Електронний ресурс] // Express.js – 2025. – Режим доступу до ресурсу: https://expressjs.com/en/resources/middleware/multer.html

12. Автентифікація та Авторизація: у чому різниця? [Електронний ресурс] // CyberCalm – 2025. – Режим доступу до ресурсу: https://cybercalm.org/novyny/avtentyfikatsiya-ta-avtoryzatsiya/
13. JSON Web Token [Електронний ресурс] // Вікіпедія – 2024. – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/JSON_Web_Token
14. Що таке OAuth? [Електронний ресурс] // Microsoft Security – 2025. – Режим доступу до ресурсу: https://www.microsoft.com/uk-ua/security/business/security-101/what-is-oauth
15. Що таке API і як він працює? [Електронний ресурс] // Acode – 2025. – Режим доступу до ресурсу: https://acode.com.ua/what-is-api/
16. YouTube Data API v3 Getting Started [Електронний ресурс] // Google Developers – 2025. – Режим доступу до ресурсу: https://developers.google.com/youtube/v3/getting-started
17. API Overview [Електронний ресурс] // TikTok for Developers – 2025. – Режим доступу до ресурсу: https://developers.tiktok.com/
18. Graph API Overview [Електронний ресурс] // Meta for Developers – 2025. – Режим доступу до ресурсу: https://developers.facebook.com/docs/graph-api/
19. Introduction to the X API [Електронний ресурс] // X Developer Platform – 2025. – Режим доступу до ресурсу: https://docs.x.com/x-api/introduction

ДОДАТОК А

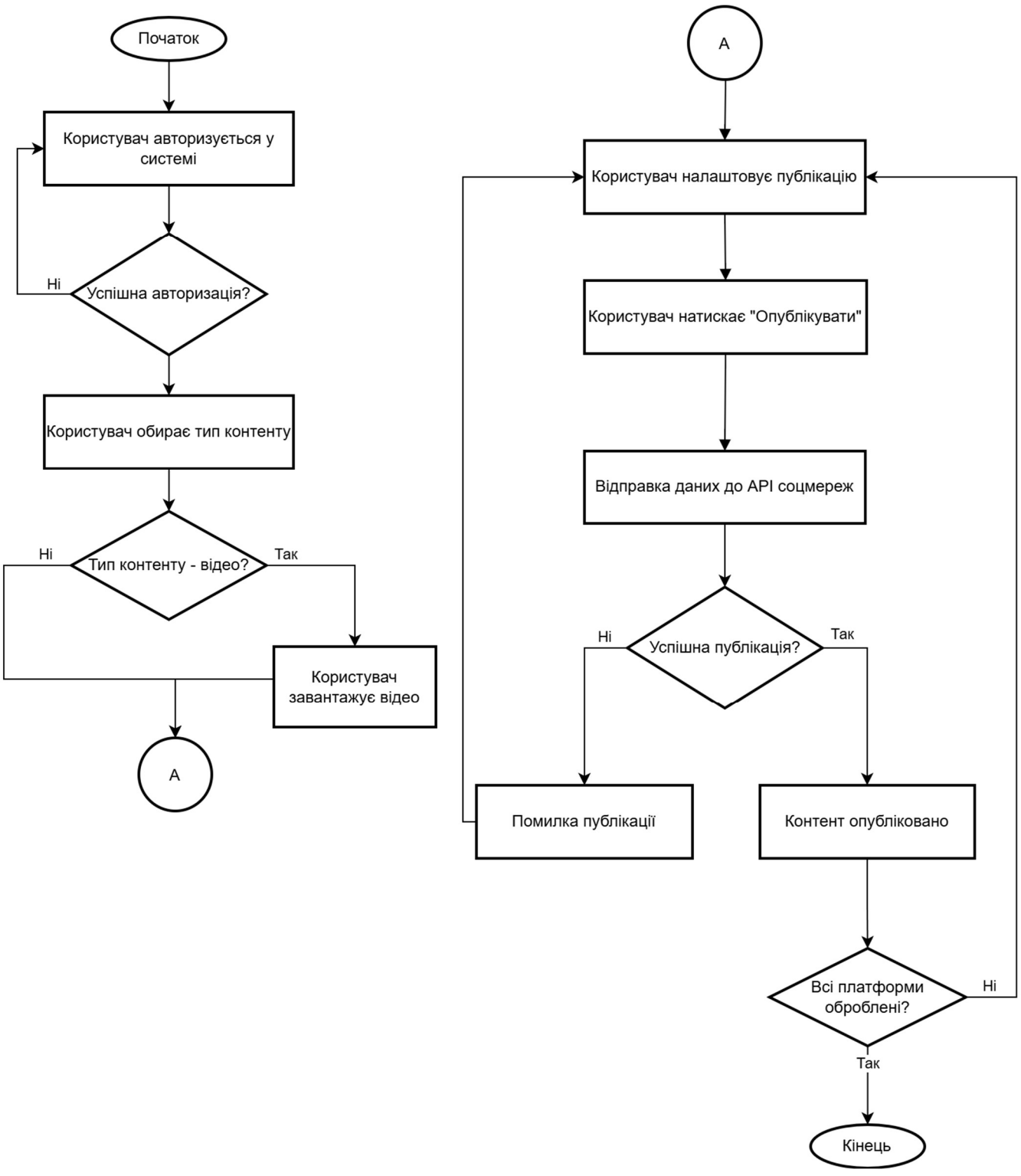
Веб-застосунок для централізованої публікації контенту в соціальних мережах

Принципова схема (блок-схема алгоритму)

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2025 р



					ІАЛЦ.467200.004 Д1	
		№ докум.	Підпис	Дата		
Розробив		Волков І. А.			**Веб-застосунок для централізованої публікації контенту в соціальних мережах** Принципова схема (блок-схема алгоритму)	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Гайдай А.Р.				КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-11
Н. Контр.		Нікольський С.С.				
Затвердив						

ДОДАТОК Б

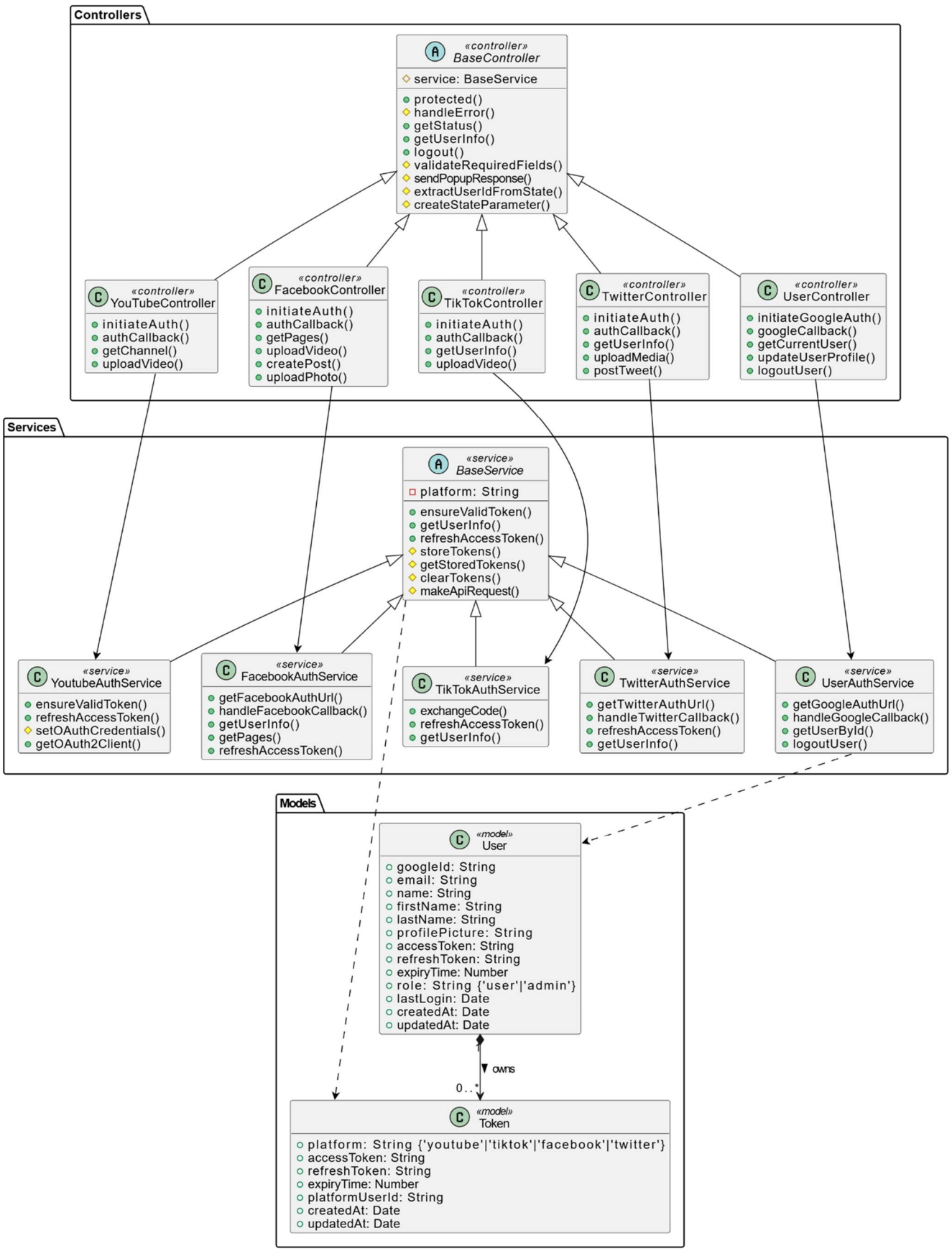
Веб-застосунок для централізованої публікації контенту в соціальних мережах

Функціональна схема (діаграма класів)

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2025 р



					ІАЛЦ.467200.005 Д2	
		№ докум.	Підпис	Дата		
Розробив		Волков І. А.			**Веб-застосунок для централізованої публікації контенту в соціальних мережах** Функціональна схема (діаграма класів)	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Гайдай А.Р.				КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-11
Н. Контр.		Нікольський С.С.				
Затвердив						

ДОДАТОК В

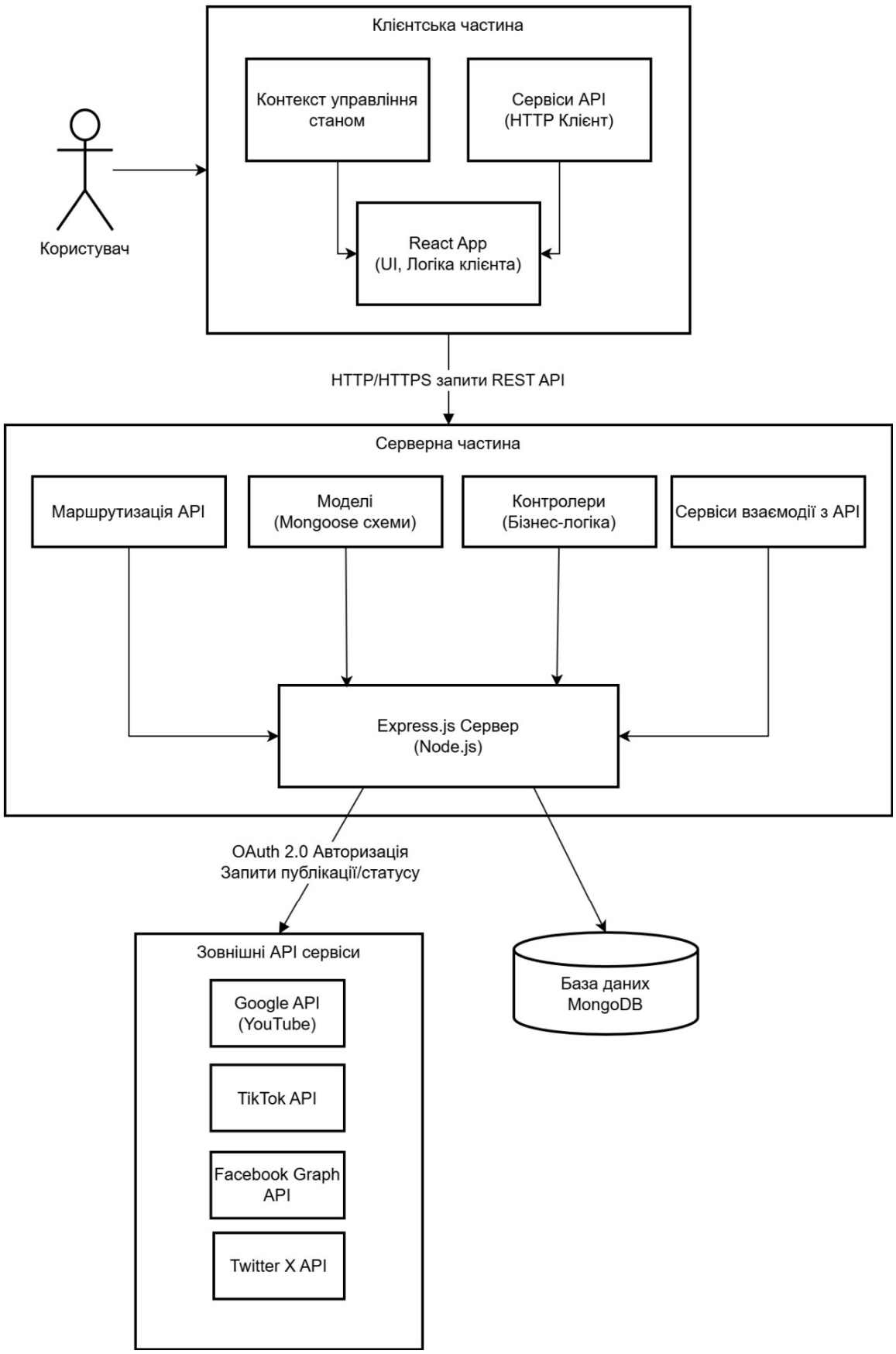
Веб-застосунок для централізованої публікації контенту в соціальних мережах

Структурна схема системи

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2025 р



					ІАЛЦ.467200.006 Д3	
		№ докум.	Підпис	Дата		
Розробив		Волков І. А.			**Веб-застосунок для централізованої публікації контенту в соціальних мережах** Структурна схема системи	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Гайдай А.Р.				КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-11
Н. Контр.		Нікольський С.С.				
Затвердив						

ДОДАТОК Г

Веб-застосунок для централізованої публікації контенту в соціальних мережах

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 60

Київ 2025 р

```
// server.js

require('dotenv').config();
const express = require('express');
const cors = require('cors');
const { connectToDatabase } = require('./lib/database');

const authRoutes = require('./routes/authRoutes');
const youtubeRoutes = require('./routes/youtubeRoutes');
const tiktokRoutes = require('./routes/tiktokRoutes');
const userRoutes = require('./routes/userRoutes');
const facebookRoutes = require('./routes/facebookRoutes');
const twitterRoutes = require('./routes/twitterRoutes');

const app = express();
const port = process.env.PORT || 8080;
const host = process.env.HOST || '0.0.0.0';

const frontendOrigin = process.env.FRONTEND_URL;

// Налаштування CORS для взаємодії з фронтендом
app.use(cors({
  origin: frontendOrigin,
  credentials: true
}));
app.use(express.json());

app.use('/auth', authRoutes);
app.use('/api/youtube', youtubeRoutes);
app.use('/api/tiktok', tiktokRoutes);
app.use('/api/users', userRoutes);
app.use('/api/facebook', facebookRoutes);
app.use('/api/twitter', twitterRoutes);

// Запуск сервера після підключення до бази даних
async function startServer() {
  try {
    await connectToDatabase();
    app.listen(port, host, () => {
      console.log(`Backend server running on http://${host}:${port}`);
    });
  } catch (error) {
    console.error('Failed to start server:', error);
    process.exit(1);
  }
}

startServer();

// database.js

const mongoose = require('mongoose');
require('dotenv').config();

const MONGODB_URI = process.env.MONGODB_URI;

async function connectToDatabase() {
  try {
    await mongoose.connect(MONGODB_URI);
```

```
    console.log('Connected to MongoDB successfully');

    process.on('SIGINT', async () => {
      await mongoose.connection.close();
      console.log('MongoDB connection closed due to app termination');
      process.exit(0);
    });
  } catch (error) {
    console.error('Error connecting to MongoDB:', error);
    process.exit(1);
  }
}

module.exports = { connectToDatabase };

// jwt.js

const jwt = require('jsonwebtoken');

const JWT_SECRET = process.env.JWT_SECRET;

exports.generateToken = (user) => {
  const payload = {
    userId: user.id || user._id,
    email: user.email,
    role: user.role || 'user',
  };
  return jwt.sign(payload, JWT_SECRET, { expiresIn: '7d' });
};

exports.verifyToken = (token) => {
  try {
    return jwt.verify(token, JWT_SECRET);
  } catch (error) {
    return null;
  }
};

exports.generateRefreshToken = (user) => {
  const payload = {
    userId: user.id || user._id,
    type: 'refresh',
  };
  return jwt.sign(payload, JWT_SECRET, { expiresIn: '30d' });
};

exports.authenticateToken = (req, res, next) => {
  const authHeader = req.headers['authorization'];

  if (!authHeader?.startsWith('Bearer ')) {
    return res.status(401).json({ error: 'Unauthorized: Missing or invalid token format' });
  }

  const token = authHeader.split(' ')[1];
  const decoded = this.verifyToken(token);
  if (!decoded) {
    return res.status(401).json({ error: 'Unauthorized: Invalid or expired token' });
  }

  req.user = decoded;
  next();
};
```

```
// authRoutes.js

const express = require('express');
const youtubeController = require('../controllers/youtubeController');
const tiktokController = require('../controllers/tiktokController');
const facebookController = require('../controllers/facebookController');
const twitterController = require('../controllers/twitterController');

const router = express.Router();

// YouTube auth routes
router.get('/youtube', youtubeController.initiateAuth);
router.get('/youtube/callback', youtubeController.authCallback);
router.post('/youtube/logout', youtubeController.logout);

// TikTok auth routes
router.get('/tiktok', tiktokController.initiateAuth);
router.get('/tiktok/callback', tiktokController.authCallback);
router.post('/tiktok/logout', tiktokController.logout);

// Facebook auth routes
router.get('/facebook', facebookController.initiateAuth);
router.get('/facebook/callback', facebookController.authCallback);
router.post('/facebook/logout', facebookController.logout);

// Twitter auth routes
router.get('/twitter', twitterController.initiateAuth);
router.get('/twitter/callback', twitterController.authCallback);
router.post('/twitter/logout', twitterController.logout);

module.exports = router;

// userRoutes.js

const express = require('express');
const userController = require('../controllers/userController');
const router = express.Router();

router.get('/login/google', userController.initiateGoogleAuth);
router.get('/login/google/callback', userController.googleCallback);
router.get('/profile', userController.getCurrentUser);

module.exports = router;

// facebookRoutes.js

const express = require('express');
const facebookController = require('../controllers/facebookController');
const multer = require('multer');
const router = express.Router();

const storage = multer.memoryStorage();
const upload = multer({ storage, limits: { fileSize: 1024 * 1024 * 1024 } }); // 1GB

router.get('/status', facebookController.getStatus);
router.get('/userinfo', facebookController.getUserInfo);
router.get('/pages', facebookController.getPages);
router.post('/upload', upload.single('videoFile'), facebookController.uploadVideo);
router.post('/photo', upload.single('photo'), facebookController.uploadPhoto);
router.post('/post', facebookController.createPost);

module.exports = router;
```

```
// baseController.js

const { authenticateToken } = require('../lib/jwt');

const frontendOrigin = process.env.FRONTEND_URL;

class BaseController {
  constructor(service) {
    this.service = service;
    this.authenticateToken = authenticateToken;
  }

  // Додає middleware аутентифікації до обробника
  protected(handler) {
    return [this.authenticateToken, handler];
  }

  handleError(error, res, userId, operation, statusCode = 500) {
    console.error(`Error ${operation} for user ${userId}:`, error.response ?
error.response.data : error.message);
    if (
      error.message?.includes('re-authenticate') ||
      error.message?.includes('token') ||
      error.message?.includes('authentication') ||
      statusCode === 401
    ) {
      return res.status(401).json({ error: `Authentication error: ${error.message}. Please
reconnect.` });
    }
    return res.status(statusCode).json({
      error: error.response?.data?.error?.message || error.message || 'An unknown error
occurred'
    });
  }

  // Перевіряє чи користувач автентифікований
  getStatus = async (req, res) => {
    const userId = req.user.userId;
    try {
      await this.service.ensureValidToken(userId);
      return res.status(200).json({ isAuthenticated: true });
    } catch (error) {
      console.log(`Auth check failed for user ${userId}:`, error.message);
      return res.status(200).json({ isAuthenticated: false });
    }
  };

  // Повертає інформацію про користувача
  getUserInfo = async (req, res) => {
    const userId = req.user.userId;
    try {
      const profile = await this.service.getUserInfo(userId);
      res.json({ user: profile });
    } catch (error) {
      this.handleError(error, res, userId, `fetching user info`, 500);
    }
  };

  // Вихід користувача та очищення токенів
  logout = async (req, res) => {
    const userId = req.user.userId;
    if (!userId) {
      return res.status(401).json({ error: 'User not authenticated.' });
    }
    try {
```

```
      await this.service.clearTokens(userId);
      console.log(`Successfully cleared tokens from database for user ${userId}.`);
      res.status(200).json({ success: true, message: 'Successfully logged out.' });
    } catch (err) {
      this.handleError(err, res, userId, 'logout');
    }
  };

  // Валідує наявність обов'язкових полів у запиті
  validateRequiredFields(req, res, requiredFields) {
    for (const field of requiredFields) {
      if (field === 'file' && !req.file) {
        res.status(400).json({ error: 'No file uploaded.' });
        return false;
      } else if (field !== 'file' && (!req.body[field] || req.body[field].trim() === '')) {
        res.status(400).json({ error: `${field} is required.` });
        return false;
      }
    }
    return true;
  }

  // Відправляє відповідь у вигляді popup для OAuth flow
  sendPopupResponse(res, messageType, data) {
    const scriptContent = `
        if(window.opener) {
            window.opener.postMessage(${JSON.stringify({ type: messageType, ...data })},
'${frontendOrigin}');
        } else { console.error("window.opener not found!"); }
        window.close();
    `;
    const htmlBody = messageType.endsWith('_SUCCESS')
      ? `<h1>Authentication Success</h1><p>Closing...</p>`
      : `<h1>Authentication Error</h1><p>${data.error || 'Unknown error'}. Closing...</p>`;
    res.status(messageType.endsWith('_SUCCESS') ? 200 : 400).send(
      `<!DOCTYPE
html><html><head><title>Auth</title></head><body><script>${scriptContent}</script>${htmlBody
}</body></html>`
    );
  }

  // Дістає userId із state параметра (OAuth)
  extractUserIdFromState(state) {
    if (!state) {
      return { error: 'State parameter missing' };
    }
    try {
      const stateData = JSON.parse(Buffer.from(state, 'base64').toString('ascii'));
      if (!stateData.userId) {
        return { error: 'userId missing in state' };
      }
      return { userId: stateData.userId };
    } catch (error) {
      return { error: 'Invalid state parameter' };
    }
  }

  // Створює state параметр для OAuth flow
  createStateParameter(userId) {
    const stateData = JSON.stringify({
      userId: userId,
      nonce: Math.random().toString(36).substring(7)
    });
    return Buffer.from(stateData).toString('base64');
  }
```

```
}

module.exports = BaseController;

// userController.js

const BaseController = require('./baseController');
const userAuthService = require('../services/userAuthService');
const { generateToken, generateRefreshToken } = require('../lib/jwt');
const User = require('../models/User');

class UserController extends BaseController {
  constructor() {
    super(userAuthService);
  }

  sendPopupResponse(res, messageType, data) {
    res.send(`
      <script>
        window.opener.postMessage(${JSON.stringify({ type: messageType, ...data })}, "*");
        window.close();
      </script>
      <p>${messageType.endsWith('_SUCCESS') ? 'Authentication successful' : 'Authentication
failed'}</p>
    `);
  }

  initiateGoogleAuth = (req, res) => {
    try {
      const authUrl = this.service.getGoogleAuthUrl();
      res.redirect(authUrl);
    } catch (error) {
      this.handleError(error, res, 'unknown', 'initiating Google authentication');
    }
  };

  googleCallback = async (req, res) => {
    const { code, error } = req.query;

    if (error) {
      return this.sendPopupResponse(res, 'USER_AUTH_ERROR', { error });
    }

    if (!code) {
      return this.sendPopupResponse(res, 'USER_AUTH_ERROR', { error: 'Authorization code
missing' });
    }

    try {
      const { user, tokens } = await this.service.handleGoogleCallback(code);

      const accessToken = generateToken(user);
      const refreshToken = generateRefreshToken(user);

      this.sendPopupResponse(res, 'USER_AUTH_SUCCESS', {
        user,
        accessToken,
        refreshToken
      });
    } catch (error) {
      console.error('Error in Google callback:', error);
      this.sendPopupResponse(res, 'USER_AUTH_ERROR', { error: error.message ||
'Authentication failed' });
    }
  };
```

```
  getCurrentUser = async (req, res) => {
    try {
      const userId = req.user.userId;
      const user = await this.service.getUserById(userId);

      res.status(200).json({
        user: {
          id: user.id,
          name: user.name,
          email: user.email,
          profilePicture: user.profilePicture,
          role: user.role
        }
      });
    } catch (error) {
      this.handleError(error, res, req.user.userId, 'getting current user', 404);
    }
  };

  updateUserProfile = async (req, res) => {
    try {
      const userId = req.user.userId;
      const { name } = req.body;

      const user = await User.findById(userId);
      if (!user) {
        return res.status(404).json({ error: 'User not found' });
      }

      if (name) user.name = name;
      await user.save();

      res.status(200).json({
        user: {
          id: user._id,
          name: user.name,
          email: user.email,
          profilePicture: user.profilePicture,
          role: user.role
        }
      });
    } catch (error) {
      this.handleError(error, res, req.user.userId, 'updating user profile');
    }
  };

  logoutUser = async (req, res) => {
    try {
      const userId = req.user.userId;
      await this.service.logoutUser(userId);

      res.status(200).json({ message: 'Successfully logged out' });
    } catch (error) {
      this.handleError(error, res, req.user.userId, 'logging out user');
    }
  };
}

const userController = new UserController();

module.exports = {
  initiateGoogleAuth: userController.initiateGoogleAuth,
  googleCallback: userController.googleCallback,
  getCurrentUser: userController.protected(userController.getCurrentUser),
```

```
  updateUserProfile: userController.protected(userController.updateUserProfile),
  logoutUser: userController.protected(userController.logoutUser)
};

// youtubeController.js

const { google } = require('googleapis');
const stream = require('stream');
const axios = require('axios');
const { URLSearchParams } = require('url');
const BaseController = require('./baseController');
const youtubeAuthService = require('../services/youtubeAuthService');

const GOOGLE_CLIENT_ID = process.env.GOOGLE_CLIENT_ID;
const GOOGLE_CLIENT_SECRET = process.env.GOOGLE_CLIENT_SECRET;
const YOUTUBE_REDIRECT_URI = process.env.YOUTUBE_BACKEND_REDIRECT_URI;

const YOUTUBE_API = {
  AUTH: (params) => `https://accounts.google.com/o/oauth2/v2/auth?${params}`,
  TOKEN: 'https://oauth2.googleapis.com/token',
  VIDEO_URL: (videoId) => `https://www.youtube.com/watch?v=${videoId}`
};

class YouTubeController extends BaseController {
  constructor() {
    super(youtubeAuthService);
  }

  initiateAuth = async (req, res) => {
    if (!GOOGLE_CLIENT_ID || !YOUTUBE_REDIRECT_URI) {
      return res.status(500).json({ error: 'Google credentials or redirect URI not
configured on backend.' });
    }
    const userId = req.user.userId;
    if (!userId) {
      return res.status(401).json({ error: 'User not authenticated.' });
    }

    const state = this.createStateParameter(userId);

    const scopes = [
      'https://www.googleapis.com/auth/youtube.upload',
      'https://www.googleapis.com/auth/youtube',
      'https://www.googleapis.com/auth/youtube.readonly'
    ];
    const params = new URLSearchParams({
      client_id: GOOGLE_CLIENT_ID,
      redirect_uri: YOUTUBE_REDIRECT_URI,
      response_type: 'code',
      scope: scopes.join(' '),
      access_type: 'offline',
      prompt: 'consent',
      state: state
    });
    const authUrl = YOUTUBE_API.AUTH(params);
    res.status(200).json({ authUrl });
  };

  // Обробка callback після OAuth авторизації YouTube
  authCallback = async (req, res) => {
    const { code, error, state } = req.query;
    let userId;

    if (state) {
      const result = this.extractUserIdFromState(state);
```

```
      if (result.error) {
        return this.sendPopupResponse(res, 'YT_AUTH_ERROR', { error: result.error });
      }
      userId = result.userId;
    } else {
      return this.sendPopupResponse(res, 'YT_AUTH_ERROR', { error: 'State parameter missing'
});
    }

    if (error) {
      return this.sendPopupResponse(res, 'YT_AUTH_ERROR', { error });
    }

    if (!code) {
      return this.sendPopupResponse(res, 'YT_AUTH_ERROR', { error: 'Authorization code
missing' });
    }

    if (!GOOGLE_CLIENT_ID || !GOOGLE_CLIENT_SECRET || !YOUTUBE_REDIRECT_URI) {
      return this.sendPopupResponse(res, 'YT_AUTH_ERROR', { error: 'Backend configuration
error' });
    }

    try {
      // Обмін коду на токени YouTube
      const tokenResponse = await axios.post(YOUTUBE_API.TOKEN, new URLSearchParams({
        code: code,
        client_id: GOOGLE_CLIENT_ID,
        client_secret: GOOGLE_CLIENT_SECRET,
        redirect_uri: YOUTUBE_REDIRECT_URI,
        grant_type: 'authorization_code'
      }));

      const { access_token, refresh_token, expires_in } = tokenResponse.data;

      if (!access_token) {
        throw new Error("Access token missing in Google's response.");
      }

      // Зберігаємо токени для користувача
      await this.service.storeTokens(userId, {
        accessToken: access_token,
        refreshToken: refresh_token,
        expiryTime: Date.now() + expires_in * 1000
      });

      this.sendPopupResponse(res, 'YT_AUTH_SUCCESS', { platform: 'YouTube' });

    } catch (err) {
      this.sendPopupResponse(res, 'YT_AUTH_ERROR', { error: 'Token exchange failed' });
    }
  };

  // Отримати інформацію про YouTube канал користувача
  getChannel = async (req, res) => {
    const userId = req.user.userId;
    try {
      // Перевіряємо токен і створюємо клієнт YouTube API
      const authenticatedClient = await this.service.ensureValidToken(userId);
      const youtube = google.youtube({ version: 'v3', auth: authenticatedClient });

      const response = await youtube.channels.list({
        part: 'snippet',
        mine: true,
        maxResults: 1
```

```
      });

      if (response.data.items && response.data.items.length > 0) {
        const channel = response.data.items[0];
        const snippet = channel.snippet;
        const channelInfo = {
          id: channel.id,
          title: snippet.title,
          thumbnailUrl: snippet.thumbnails?.default?.url || snippet.thumbnails?.medium?.url
|| null
        };
        res.status(200).json(channelInfo);
      } else {
        res.status(404).json({ error: 'No YouTube channel found for the authenticated user.'
});
      }
    } catch (error) {
      this.handleError(error, res, userId, 'fetching YouTube channel info');
    }
  };

  // Завантаження відео на YouTube
  uploadVideo = async (req, res) => {
    const userId = req.user.userId;
    try {
      // Перевіряємо токен користувача
      const authenticatedClient = await this.service.ensureValidToken(userId);

      if (!this.validateRequiredFields(req, res, ['file', 'title'])) {
        return;
      }

      const { title, description, tags, visibility, scheduledDate } = req.body;

      // Формуємо метадані для відео
      const metadata = {
        snippet: {
          title: title,
          description: description || '',
          tags: tags ? tags.split(',').map(tag => tag.trim()).filter(tag => tag) : [],
        },
        status: {
          privacyStatus: (visibility === 'scheduled') ? 'private' : (visibility ||
'private'),
        },
      };

      // Якщо відео має бути заплановане
      if (visibility === 'scheduled' && scheduledDate) {
        if (isNaN(Date.parse(scheduledDate))) {
            metadata.status.privacyStatus = 'private';
        } else {
            metadata.status.publishAt = scheduledDate;
        }
      }

      // Створюємо потік для завантаження файлу
      const bufferStream = new stream.PassThrough();
      bufferStream.end(req.file.buffer);

      const youtube = google.youtube({ version: 'v3', auth: authenticatedClient });

      // Відправляємо запит на завантаження відео
      const youtubeResponse = await youtube.videos.insert({
          part: 'snippet,status',
```

```
          requestBody: metadata,
          media: {
            mimeType: req.file.mimetype,
            body: bufferStream,
          },
        });

      res.status(200).json({
        message: 'Successfully uploaded to YouTube!',
        videoId: youtubeResponse.data.id,
        videoUrl: YOUTUBE_API.VIDEO_URL(youtubeResponse.data.id),
      });

    } catch (error) {
      this.handleError(error, res, userId, 'YouTube upload');
    }
  };
}

const youtubeController = new YouTubeController();

module.exports = {
  getStatus: youtubeController.protected(youtubeController.getStatus),
  getChannel: youtubeController.protected(youtubeController.getChannel),
  uploadVideo: youtubeController.protected(youtubeController.uploadVideo),
  initiateAuth: youtubeController.protected(youtubeController.initiateAuth),
  authCallback: youtubeController.authCallback,
  logout: youtubeController.protected(youtubeController.logout)
};

// tiktokController.js

const BaseController = require('./baseController');
const tiktokAuthService = require('../services/tiktokAuthService');
const axios = require('axios');
const stream = require('stream');
const { URLSearchParams } = require('url');

const TIKTOK_CLIENT_ID = process.env.TIKTOK_CLIENT_ID;
const TIKTOK_REDIRECT_URI = process.env.TIKTOK_BACKEND_REDIRECT_URI;

const TIKTOK_API = {
  AUTH: (params) => `https://www.tiktok.com/v2/auth/authorize/?${params}`,
  VIDEO_INIT: 'https://open.tiktokapis.com/v2/post/publish/inbox/video/init/',
};

class TikTokController extends BaseController {
  constructor() {
    super(tiktokAuthService);
  }

  initiateAuth = async (req, res) => {
    if (!TIKTOK_CLIENT_ID || !TIKTOK_REDIRECT_URI) {
      return res.status(500).json({ error: 'TikTok Client ID or Redirect URI not configured
on backend.' });
    }
    const userId = req.user.userId;
    if (!userId) {
      return res.status(401).json({ error: 'User not authenticated.' });
    }

    const scopes = [
      "user.info.basic",
      "video.upload",
    ];
```

```
    const csrfState = this.createStateParameter(userId);

    const params = new URLSearchParams({
      client_key: TIKTOK_CLIENT_ID,
      scope: scopes.join(','),
      response_type: 'code',
      redirect_uri: TIKTOK_REDIRECT_URI,
      state: csrfState,
    });
    const authUrl = TIKTOK_API.AUTH(params);

    res.status(200).json({ authUrl });
  };

  authCallback = async (req, res) => {
    const { code, state, error, error_description } = req.query;
    let userId;

    if (state) {
      const result = this.extractUserIdFromState(state);
      if (result.error) {
        return this.sendPopupResponse(res, 'TT_AUTH_ERROR', { error: result.error });
      }
      userId = result.userId;
    } else {
      return this.sendPopupResponse(res, 'TT_AUTH_ERROR', { error: 'State parameter missing'
});
    }

    if (error) {
      return this.sendPopupResponse(res, 'TT_AUTH_ERROR', { error: error_description ||
error });
    }
    if (!code) {
      return this.sendPopupResponse(res, 'TT_AUTH_ERROR', { error: 'Authorization code
missing' });
    }

    try {
      await this.service.exchangeCode(userId, code);
      this.sendPopupResponse(res, 'TT_AUTH_SUCCESS', { platform: 'TikTok' });
    } catch (err) {
      this.sendPopupResponse(res, 'TT_AUTH_ERROR', { error: err.message || 'Token exchange
failed' });
    }
  };

  getUserInfo = async (req, res) => {
    const userId = req.user.userId;
    try {
      const userInfo = await this.service.getUserInfo(userId);
      res.status(200).json(userInfo);
    } catch (error) {
      if (error.message?.includes('re-authenticate') ||
          error.message?.includes('token') ||
          error.message?.includes('PERMISSION_DENIED')) {
        await this.service.clearTokens(userId);
        res.status(401).json({ error: `Authentication error: ${error.message}. Please
reconnect TikTok.` });
      } else {
        this.handleError(error, res, userId, 'fetching TikTok user info');
      }
    }
  };
```

```
  uploadVideo = async (req, res) => {
    const userId = req.user.userId;
    try {
      const accessToken = await this.service.ensureValidToken(userId);

      if (!req.file) {
        return res.status(400).json({ error: 'No video file uploaded.' });
      }
      if (!req.file.buffer) {
        return res.status(500).json({ error: 'Internal server error: Upload buffer
unavailable.' });
      }

      const initEndpoint = TIKTOK_API.VIDEO_INIT;

      const initResponse = await axios.post(
        initEndpoint,
        {
          source_info: {
            source: "FILE_UPLOAD",
            video_size: req.file.size,
            chunk_size: req.file.size,
            total_chunk_count: 1
          }
        },
        {
          headers: {
            'Authorization': `Bearer ${accessToken}`,
            'Content-Type': 'application/json; charset=UTF-8'
          }
        }
      );

      const initError = initResponse.data.error;
      if (initError && initError.code !== 'ok') {
        throw new Error(`Failed to initialize TikTok upload: ${initError.message || 'Unknown
init error'} (Code: ${initError.code})`);
      }

      if (!initResponse.data.data?.publish_id || !initResponse.data.data?.upload_url) {
        throw new Error("Incomplete initialization response from TikTok (publish_id or
upload_url missing in data object)");
      }

      const { publish_id, upload_url } = initResponse.data.data;

      const bufferStream = new stream.PassThrough();
      bufferStream.end(req.file.buffer);

      const uploadResponse = await axios.put(
        upload_url,
        bufferStream,
        {
          headers: {
            'Content-Type': req.file.mimetype,
            'Content-Range': `bytes 0-${req.file.size - 1}/${req.file.size}`
          },
          maxBodyLength: Infinity,
          maxContentLength: Infinity,
          timeout: 300000
        }
      );

      if (uploadResponse.status < 200 || uploadResponse.status >= 300) {
```

```
        throw new Error(`Video file upload to TikTok storage failed with status
${uploadResponse.status}`);
      }

      res.status(200).json({
        message: 'Successfully initiated video upload to TikTok Inbox!',
        upload_id: publish_id,
        details: "Video is processing in TikTok inbox. Check the TikTok app."
      });

    } catch (error) {
      let statusCode = 500;
      let errorMessage = 'An unknown error occurred during TikTok upload.';

      if (axios.isAxiosError(error)) {
        statusCode = error.response?.status || 500;
        const tiktokError = error.response?.data?.error;
        const uploadErrorData = error.response?.data;

        errorMessage = tiktokError?.message ||
                       (typeof uploadErrorData === 'string' ?
uploadErrorData.substring(0,200) : null) ||
                       error.message;

        if ((statusCode === 401 || statusCode === 403 || tiktokError?.code ===
'permission_denied' || tiktokError?.code === 2003) &&
error.config.url.includes('video/init')) {
          try {
            await this.service.clearTokens(userId);
          } catch (clearErr) { console.error("Failed to clear tokens:", clearErr); }
          errorMessage = `TikTok Auth/Permission Error during init (${tiktokError?.code}):
${errorMessage}. Please reconnect.`;
          statusCode = statusCode === 403 ? 403 : 401;
        } else {
          errorMessage = `TikTok API Error (${tiktokError?.code || statusCode}):
${errorMessage}`;
        }
      } else if (error.message?.includes('re-authenticate')) {
        statusCode = 401;
        errorMessage = `Authentication error: ${error.message}. Please reconnect TikTok.`;
      } else {
        errorMessage = error.message || errorMessage;
      }

      return res.status(statusCode).json({ error: errorMessage });
    }
  };
}

const tiktokController = new TikTokController();

module.exports = {
  getStatus: tiktokController.protected(tiktokController.getStatus),
  getUserInfo: tiktokController.protected(tiktokController.getUserInfo),
  uploadVideo: tiktokController.protected(tiktokController.uploadVideo),
  initiateAuth: tiktokController.protected(tiktokController.initiateAuth),
  authCallback: tiktokController.authCallback,
  logout: tiktokController.protected(tiktokController.logout)
};

// facebookController.js

const BaseController = require('./baseController');
const facebookAuthService = require('../services/facebookAuthService');
const axios = require('axios');
```

```
const FormData = require('form-data');

const GRAPH = 'https://graph.facebook.com/v22.0';
const FACEBOOK_API = {
  GRAPH,
  VIDEO: (pageId) => `${GRAPH}/${pageId}/videos`,
  FEED: (pageId) => `${GRAPH}/${pageId}/feed`,
  PHOTO: (pageId) => `${GRAPH}/${pageId}/photos`,
  VIDEO_URL: (pageId, videoId) => `https://www.facebook.com/${pageId}/videos/${videoId}`
};

class FacebookController extends BaseController {
  constructor() {
    super(facebookAuthService);
  }

  initiateAuth = async (req, res) => {
    try {
      const userId = req.user.userId;
      const authUrl = this.service.getFacebookAuthUrl(userId);
      res.status(200).json({ authUrl });
    } catch (error) {
      this.handleError(error, res, req.user.userId, 'initiating Facebook auth');
    }
  };

  authCallback = async (req, res) => {
    const { code, state, error } = req.query;
    if (error) {
      return this.sendPopupResponse(res, 'FB_AUTH_ERROR', { error });
    }
    if (!code || !state) {
      return this.sendPopupResponse(res, 'FB_AUTH_ERROR', { error: 'Missing code or state'
});
    }

    const result = this.extractUserIdFromState(state);
    if (result.error) {
      return this.sendPopupResponse(res, 'FB_AUTH_ERROR', { error: result.error });
    }
    const userId = result.userId;

    try {
      const { user, pages } = await this.service.handleFacebookCallback(userId, code);
      this.sendPopupResponse(res, 'FB_AUTH_SUCCESS', { user, pages });
    } catch (err) {
      this.sendPopupResponse(res, 'FB_AUTH_ERROR', { error: err.message || 'Facebook
authentication failed' });
    }
  };

  getPages = async (req, res) => {
    const userId = req.user.userId;
    try {
      const pages = await this.service.getPages(userId);
      res.json({ pages });
    } catch (error) {
      this.handleError(error, res, userId, 'fetching Facebook pages');
    }
  };

  uploadVideo = async (req, res) => {
    const userId = req.user.userId;
    const { title, description, selectedPageId } = req.body;
```

```
    if (!this.validateRequiredFields(req, res, ['file', 'selectedPageId'])) {
      return;
    }

    try {
      const pages = await this.service.getPages(userId);
      const page = pages.find(p => p.id === selectedPageId);
      if (!page || !page.access_token) {
        return res.status(403).json({ error: 'Not authenticated or missing page access
token' });
      }
      const pageToken = page.access_token;

      const form = new FormData();
      form.append('source', req.file.buffer, { filename: req.file.originalname, contentType:
req.file.mimetype });
      if (title) form.append('title', title);
      if (description) form.append('description', description);
      form.append('access_token', pageToken);

      const fbRes = await axios.post(
        FACEBOOK_API.VIDEO(selectedPageId),
        form,
        { headers: form.getHeaders() }
      );

      const videoId = fbRes.data.id;
      const videoUrl = FACEBOOK_API.VIDEO_URL(selectedPageId, videoId);
      res.json({ videoId, videoUrl });
    } catch (error) {
      const message = error.response?.data?.error?.message || error.message;
      res.status(500).json({ error: message });
    }
  };

  createPost = async (req, res) => {
    const userId = req.user.userId;
    const { pageId, message, link, published, scheduledPublishTime } = req.body;

    if (!this.validateRequiredFields(req, res, ['pageId', 'message'])) {
      return;
    }

    try {
      const pages = await this.service.getPages(userId);
      const page = pages.find(p => p.id === pageId);
      if (!page || !page.access_token) {
        return res.status(403).json({ error: 'Not authenticated or missing page access
token.' });
      }
      const params = { access_token: page.access_token, message };
      if (link) params.link = link;
      if (typeof published !== 'undefined') params.published = published;
      if (scheduledPublishTime) params.scheduled_publish_time = scheduledPublishTime;
      const fbRes = await axios.post(
        FACEBOOK_API.FEED(pageId),
        params
      );
      return res.json({ postId: fbRes.data.id });
    } catch (error) {
      const msg = error.response?.data?.error?.message || error.message;
      res.status(500).json({ error: msg });
    }
  };
```

```
  uploadPhoto = async (req, res) => {
    const userId = req.user.userId;
    const { pageId, message, published, scheduledPublishTime } = req.body;

    if (!this.validateRequiredFields(req, res, ['pageId', 'file'])) {
      return;
    }

    try {
      const pages = await this.service.getPages(userId);
      const page = pages.find(p => p.id === pageId);
      if (!page || !page.access_token) {
        return res.status(403).json({ error: 'Not authenticated or missing page access
token.' });
      }
      const form = new FormData();
      form.append('source', req.file.buffer, { filename: req.file.originalname, contentType:
req.file.mimetype });
      form.append('access_token', page.access_token);
      if (message) form.append('caption', message);
      if (typeof published !== 'undefined') form.append('published', published);
      if (scheduledPublishTime) form.append('scheduled_publish_time', scheduledPublishTime);
      const fbRes = await axios.post(
        FACEBOOK_API.PHOTO(pageId),
        form,
        { headers: form.getHeaders() }
      );
      res.json({ photoId: fbRes.data.id, postId: fbRes.data.post_id });
    } catch (error) {
      const errMsg = error.response?.data?.error?.message || error.message;
      res.status(500).json({ error: errMsg });
    }
  };
}

const facebookController = new FacebookController();

module.exports = {
  getStatus: facebookController.protected(facebookController.getStatus),
  getUserInfo: facebookController.protected(facebookController.getUserInfo),
  getPages: facebookController.protected(facebookController.getPages),
  uploadVideo: facebookController.protected(facebookController.uploadVideo),
  createPost: facebookController.protected(facebookController.createPost),
  uploadPhoto: facebookController.protected(facebookController.uploadPhoto),
  initiateAuth: facebookController.protected(facebookController.initiateAuth),
  authCallback: facebookController.authCallback,
  logout: facebookController.protected(facebookController.logout)
};

// twitterController.js

const BaseController = require('./baseController');
const twitterAuthService = require('../services/twitterAuthService');
const axios = require('axios');
const FormData = require('form-data');

const TWITTER_API = {
  MEDIA_UPLOAD_INITIALIZE: 'https://api.twitter.com/2/media/upload/initialize',
  MEDIA_UPLOAD_APPEND: 'https://api.twitter.com/2/media/upload/{id}/append',
  MEDIA_UPLOAD_FINALIZE: 'https://api.twitter.com/2/media/upload/{id}/finalize',
  MEDIA_UPLOAD_STATUS: 'https://api.twitter.com/2/media/upload',
  TWEET: 'https://api.twitter.com/2/tweets',
  USER_INFO: 'https://api.twitter.com/2/users/me'
};
```

```
const CHUNK_SIZE = 4 * 1024 * 1024;

class TwitterController extends BaseController {
  constructor() {
    super(twitterAuthService);
  }

  initiateAuth = async (req, res) => {
    try {
      const userId = req.user.userId;
      const authUrl = this.service.getTwitterAuthUrl(userId);
      res.status(200).json({ authUrl });
    } catch (error) {
      this.handleError(error, res, req.user.userId, 'initiating Twitter auth');
    }
  };

  authCallback = async (req, res) => {
    const { code, state, error } = req.query;
    if (error) {
      return this.sendPopupResponse(res, 'TW_AUTH_ERROR', { error });
    }
    if (!code || !state) {
      return this.sendPopupResponse(res, 'TW_AUTH_ERROR', { error: 'Missing code or state'
});
    }

    const result = this.extractUserIdFromState(state);
    if (result.error) {
      return this.sendPopupResponse(res, 'TW_AUTH_ERROR', { error: result.error });
    }
    const userId = result.userId;

    try {
      await this.service.handleTwitterCallback(userId, code);
      this.sendPopupResponse(res, 'TW_AUTH_SUCCESS', { platform: 'Twitter' });
    } catch (err) {
      this.sendPopupResponse(res, 'TW_AUTH_ERROR', { error: err.message || 'Twitter
authentication failed' });
    }
  };

  getUserInfo = async (req, res) => {
    try {
      const userId = req.user.userId;
      const user = await this.service.getUserInfo(userId);

      res.json({
        success: true,
        user
      });
    } catch (error) {
      this.handleError(error, res, req.user.userId, 'fetching Twitter user info');
    }
  };

  async getTwitterAuth(userId) {
    try {
      const accessToken = await this.service.ensureValidToken(userId);
      return { accessToken };
    } catch (error) {
      return { error: error.message || 'Not authenticated with Twitter' };
    }
  }
```

```
  async uploadMediaToTwitter(file, accessToken) {
    try {
      const totalBytes = file.size;
      const mediaType = file.mimetype;
      const isVideo = mediaType.startsWith('video/');

      const mediaCategory = mediaType.startsWith('image/')
        ? 'tweet_image'
        : 'tweet_video';

      const initPayload = {
        media_type: mediaType,
        total_bytes: totalBytes,
        media_category: mediaCategory
      };

      if (isVideo) {
        initPayload.additional_owners = [];
      }

      const initResp = await axios.post(TWITTER_API.MEDIA_UPLOAD_INITIALIZE, initPayload, {
        headers: {
          Authorization: `Bearer ${accessToken}`,
          'Content-Type': 'application/json'
        }
      }).catch(err => {
        throw err;
      });

      const mediaId = initResp.data.data.id;

      const totalChunks = Math.ceil(totalBytes / CHUNK_SIZE);

      for (let i = 0; i < totalBytes; i += CHUNK_SIZE) {
        const chunk = file.buffer.slice(i, i + CHUNK_SIZE);
        const segmentIndex = Math.floor(i / CHUNK_SIZE);

        const appendForm = new FormData();
        appendForm.append('segment_index', segmentIndex);
        appendForm.append('media', chunk, {
          filename: file.originalname,
          contentType: mediaType
        });

        await axios.post(TWITTER_API.MEDIA_UPLOAD_APPEND.replace('{id}', mediaId),
appendForm, {
          headers: {
            Authorization: `Bearer ${accessToken}`,
            ...appendForm.getHeaders()
          }
        }).catch(err => {
          throw err;
        });
      }

      const finalizePayload = {
        total_chunks: Math.ceil(totalBytes / CHUNK_SIZE)
      };

      const finalizeResp = await
axios.post(TWITTER_API.MEDIA_UPLOAD_FINALIZE.replace('{id}', mediaId), finalizePayload, {
        headers: {
          Authorization: `Bearer ${accessToken}`,
          'Content-Type': 'application/json'
        }
```

```
      }).catch(err => {
        throw err;
      });

      let processingInfo = finalizeResp.data.data.processing_info;
      if (!processingInfo) {
        return mediaId;
      }

      let state = processingInfo.state;
      let checkAfterSecs = processingInfo.check_after_secs || 1;

      while (state && state !== 'succeeded' && state !== 'failed') {
        await new Promise(resolve => setTimeout(resolve, checkAfterSecs * 1000));

        const statusResp = await axios.get(TWITTER_API.MEDIA_UPLOAD_STATUS, {
          params: {
            media_id: mediaId,
            command: 'STATUS'
          },
          headers: {
            Authorization: `Bearer ${accessToken}`,
            'Content-Type': 'application/json'
          }
        }).catch(err => {
          throw err;
        });

        processingInfo = statusResp.data.data.processing_info;
        if (!processingInfo) {
          break;
        }

        state = processingInfo.state;
        checkAfterSecs = processingInfo.check_after_secs || 1;

        if (state === 'failed') {
          throw new Error(`Twitter media processing failed:
${JSON.stringify(processingInfo)}`);
        }
      }

      return mediaId;
    } catch (error) {
      if (error.response) {
        throw new Error(`Twitter API error (${error.response.status}):
${JSON.stringify(error.response.data)}`);
      }
      throw error;
    }
  }

  async postTweet(accessToken, text, mediaIds = []) {
    const payload = { text: text || '' };

    if (mediaIds.length > 0) {
      payload.media = { media_ids: mediaIds };
    }

    const response = await axios.post(TWITTER_API.TWEET, payload, {
      headers: {
        Authorization: `Bearer ${accessToken}`,
        'Content-Type': 'application/json'
      }
    });
```

```
    return response.data;
  }

  uploadToTwitter = async (req, res) => {
    try {
      const userId = req.user.userId;
      const { error, accessToken } = await this.getTwitterAuth(userId);
      if (error) {
        return res.status(401).json({ success: false, error });
      }
      const file = req.file;
      const text = req.body.text || '';
      let mediaIds = [];
      if (file) {
        try {
          const mediaId = await this.uploadMediaToTwitter(file, accessToken);
          mediaIds.push(mediaId);
        } catch (err) {
          return res.status(500).json({
            success: false,
            error: 'Media upload failed',
            details: err.message
          });
        }
      }
      try {
        const tweetResponse = await this.postTweet(accessToken, text, mediaIds);
        res.json({
          success: true,
          tweet: tweetResponse
        });
      } catch (err) {
        return res.status(500).json({
          success: false,
          error: 'Tweet failed',
          details: err.response?.data || err.message
        });
      }
    } catch (error) {
      this.handleError(error, res, req.user.userId, 'posting to Twitter');
    }
  };
}

const twitterController = new TwitterController();

module.exports = {
  getStatus: twitterController.protected(twitterController.getStatus),
  getUserInfo: twitterController.protected(twitterController.getUserInfo),
  uploadToTwitter: twitterController.protected(twitterController.uploadToTwitter),
  initiateAuth: twitterController.protected(twitterController.initiateAuth),
  authCallback: twitterController.authCallback,
  logout: twitterController.protected(twitterController.logout)
};

// baseService.js

const Token = require('../models/Token');
const axios = require('axios');

class BaseService {
  constructor(platform, config = {}) {
    this.platform = platform;
    this.config = config;
```

```
  }

  async storeTokens(userId, { accessToken, refreshToken, expiryTime, platformUserId }) {
    try {
      let token = await Token.findOne({ userId, platform: this.platform });

      if (token) {
        token.accessToken = accessToken;
        if (refreshToken) {
          token.refreshToken = refreshToken;
        }
        token.expiryTime = expiryTime;
        if (platformUserId) {
          token.platformUserId = platformUserId;
        }
      } else {
        token = new Token({
          userId,
          platform: this.platform,
          accessToken,
          refreshToken,
          expiryTime,
          platformUserId: platformUserId || null
        });
      }

      await token.save();
    } catch (error) {
      throw new Error(`Failed to store authentication tokens for ${this.platform}`);
    }
  }

  async getStoredTokens(userId) {
    if (!userId) {
      return { accessToken: null, refreshToken: null, expiryTime: null, platformUserId: null
};
    }

    try {
      const token = await Token.findOne({ userId, platform: this.platform });

      if (!token) {
        return { accessToken: null, refreshToken: null, expiryTime: null, platformUserId:
null };
      }

      return {
        accessToken: token.accessToken,
        refreshToken: token.refreshToken,
        expiryTime: token.expiryTime,
        platformUserId: token.platformUserId
      };
    } catch (error) {
      return { accessToken: null, refreshToken: null, expiryTime: null, platformUserId: null
};
    }
  }

  async clearTokens(userId) {
    if (!userId) {
      return;
    }

    try {
      await Token.deleteOne({ userId, platform: this.platform });
```

```
    } catch (error) {
      console.error(`Error clearing ${this.platform} tokens for user ${userId}:`, error);
    }
  }

  async makeApiRequest(method, url, options = {}) {
    try {
      const response = await axios({
        method,
        url,
        ...options
      });

      return response.data;
    } catch (error) {
      if (error.response) {
        const errorData = error.response.data;
        const errorMessage = errorData?.error?.message ||
                             errorData?.error_description ||
                             errorData?.error ||
                             'API request failed';

        throw new Error(`${this.platform} API Error (${error.response.status}):
${errorMessage}`);
      }

      throw error;
    }
  }

  async ensureValidToken(userId) {
    const currentTokens = await this.getStoredTokens(userId);

    if (!currentTokens.accessToken) {
      throw new Error(`User not authenticated with ${this.platform}`);
    }

    const isTokenExpired = !currentTokens.expiryTime || Date.now() >=
currentTokens.expiryTime;

    if (isTokenExpired && currentTokens.refreshToken) {
      return await this.refreshAccessToken(userId);
    }

    return currentTokens.accessToken;
  }

  async refreshAccessToken(userId) {
    throw new Error('refreshAccessToken must be implemented by subclass');
  }

  async getUserInfo(userId) {
    throw new Error('getUserInfo must be implemented by subclass');
  }
}

module.exports = BaseService;

// userAuthService.js

const { google } = require('googleapis');
const User = require('../models/User');
const BaseService = require('./baseService');

const PLATFORM = 'google';
```

```
const clientId = process.env.GOOGLE_CLIENT_ID;
const clientSecret = process.env.GOOGLE_CLIENT_SECRET;
const userAuthRedirectUri = process.env.USER_AUTH_REDIRECT_URI;

class UserAuthService extends BaseService {
  constructor() {
    super(PLATFORM, { clientId, clientSecret, redirectUri: userAuthRedirectUri });
    this.userAuthClient = new google.auth.OAuth2(clientId, clientSecret,
userAuthRedirectUri);
  }

  getGoogleAuthUrl() {
    const scopes = [
      'https://www.googleapis.com/auth/userinfo.profile',
      'https://www.googleapis.com/auth/userinfo.email',
      'openid',
      'profile',
      'email'
    ];

    return this.userAuthClient.generateAuthUrl({
      access_type: 'offline',
      prompt: 'consent',
      scope: scopes,
      include_granted_scopes: true
    });
  }

  async handleGoogleCallback(code) {
    try {
      const { tokens } = await this.userAuthClient.getToken(code);
      this.userAuthClient.setCredentials(tokens);

      const oauth2 = google.oauth2({ verssion: 'v2', auth: this.userAuthClient });
      const { data } = await oauth2.userinfo.get();

      let user = await User.findOne({ googleId: data.id });

      if (user) {
        user.name = data.name;
        user.firstName = data.given_name;
        user.lastName = data.family_name;
        user.profilePicture = data.picture;
        user.accessToken = tokens.access_token;
        user.lastLogin = new Date();

        if (tokens.refresh_token) {
          user.refreshToken = tokens.refresh_token;
        }

        if (tokens.expiry_date) {
          user.expiryTime = tokens.expiry_date;
        }
      } else {
        user = new User({
          googleId: data.id,
          email: data.email,
          name: data.name,
          firstName: data.given_name,
          lastName: data.family_name,
          profilePicture: data.picture,
          accessToken: tokens.access_token,
          refreshToken: tokens.refresh_token,
          expiryTime: tokens.expiry_date,
        });
```

```
    }

    await user.save();

    return {
      user: {
        id: user._id,
        googleId: user.googleId,
        name: user.name,
        email: user.email,
        profilePicture: user.profilePicture,
        role: user.role
      },
      tokens
    };
  } catch (error) {
    throw new Error('Failed to authenticate with Google');
  }
}

async getUserById(userId) {
  try {
    const user = await User.findById(userId);
    if (!user) {
      throw new Error('User not found');
    }

    return {
      id: user._id,
      googleId: user.googleId,
      name: user.name,
      email: user.email,
      profilePicture: user.profilePicture,
      role: user.role
    };
  } catch (error) {
    throw error;
  }
}

async refreshUserToken(userId) {
  try {
    const user = await User.findById(userId);
    if (!user || !user.refreshToken) {
      throw new Error('User not found or missing refresh token');
    }

    const isTokenExpired = !user.expiryTime || Date.now() >= user.expiryTime - (5 * 60 *
1000);

    if (isTokenExpired) {
      this.userAuthClient.setCredentials({
        refresh_token: user.refreshToken
      });

      const { credentials } = await this.userAuthClient.refreshAccessToken();

      user.accessToken = credentials.access_token;
      if (credentials.refresh_token) {
        user.refreshToken = credentials.refresh_token;
      }
      user.expiryTime = credentials.expiry_date;

      await user.save();
    }
```

```
      return {
        accessToken: user.accessToken,
        expiryTime: user.expiryTime
      };
    } catch (error) {
      throw error;
    }
  }

  async logoutUser(userId) {
    try {
      const user = await User.findById(userId);
      if (!user) {
        throw new Error('User not found');
      }

      user.accessToken = null;
      user.expiryTime = null;

      await user.save();

      return { success: true };
    } catch (error) {
      throw error;
    }
  }
}

const userAuthService = new UserAuthService();
module.exports = userAuthService;

// youtubeAuthService.js

const { google } = require('googleapis');
const BaseService = require('./baseService');

const PLATFORM = 'youtube';
const clientId = process.env.GOOGLE_CLIENT_ID;
const clientSecret = process.env.GOOGLE_CLIENT_SECRET;
const redirectUri = process.env.YOUTUBE_BACKEND_REDIRECT_URI;

class YoutubeAuthService extends BaseService {
  constructor() {
    super(PLATFORM, { clientId, clientSecret, redirectUri });
    this.oauth2Client = new google.auth.OAuth2(clientId, clientSecret, redirectUri);
  }

  async setOAuthCredentials(userId) {
    const tokens = await this.getStoredTokens(userId);
    if (tokens.accessToken && tokens.refreshToken) {
      this.oauth2Client.setCredentials({
        access_token: tokens.accessToken,
        refresh_token: tokens.refreshToken,
        expiry_date: tokens.expiryTime,
      });
    } else {
      this.oauth2Client.setCredentials({});
    }
  }

  async ensureValidToken(userId) {
    await this.setOAuthCredentials(userId);

    const currentTokens = await this.getStoredTokens(userId);
```

```
    if (!currentTokens.accessToken || !currentTokens.refreshToken) {
      throw new Error('User not authenticated with YouTube or refresh token missing.');
    }

    const isTokenExpired = !currentTokens.expiryTime || Date.now() >=
currentTokens.expiryTime;

    if (isTokenExpired) {
      try {
        const { credentials } = await this.oauth2Client.refreshAccessToken();

        await this.storeTokens(userId, {
          accessToken: credentials.access_token,
          refreshToken: credentials.refresh_token || currentTokens.refreshToken,
          expiryTime: credentials.expiry_date
        });

        this.oauth2Client.setCredentials(credentials);

      } catch (refreshError) {
        await this.clearTokens(userId);
        this.oauth2Client.setCredentials({});
        throw new Error('Failed to refresh YouTube token. Please re-authenticate.');
      }
    }

    if (!this.oauth2Client.credentials.access_token) {
      throw new Error('Missing access token after check/refresh.');
    }

    return this.oauth2Client;
  }

  async refreshAccessToken(userId) {
    const currentTokens = await this.getStoredTokens(userId);
    this.oauth2Client.setCredentials({
      refresh_token: currentTokens.refreshToken
    });

    const { credentials } = await this.oauth2Client.refreshAccessToken();

    await this.storeTokens(userId, {
      accessToken: credentials.access_token,
      refreshToken: credentials.refresh_token || currentTokens.refreshToken,
      expiryTime: credentials.expiry_date
    });

    return credentials.access_token;
  }

  getOAuth2Client() {
    return this.oauth2Client;
  }
}

const youtubeAuthService = new YoutubeAuthService();
module.exports = youtubeAuthService;

// tiktokAuthService.js

const axios = require('axios');
const { URLSearchParams } = require('url');
const BaseService = require('./baseService');

const PLATFORM = 'tiktok';
```

```
const clientId = process.env.TIKTOK_CLIENT_ID;
const clientSecret = process.env.TIKTOK_CLIENT_SECRET;
const redirectUri = process.env.TIKTOK_BACKEND_REDIRECT_URI;
const tokenEndpoint = 'https://open.tiktokapis.com/v2/oauth/token/';

class TikTokAuthService extends BaseService {
  constructor() {
    super(PLATFORM, {
      clientId,
      clientSecret,
      redirectUri,
      tokenEndpoint
    });
  }

  async exchangeCode(userId, code) {
    if (!this.config.clientId || !this.config.clientSecret || !this.config.redirectUri) {
      throw new Error("TikTok Client ID, Client Secret, or Redirect URI not configured on
backend");
    }

    try {
      const params = {
        client_key: this.config.clientId,
        client_secret: this.config.clientSecret,
        code: code,
        grant_type: 'authorization_code',
        redirect_uri: this.config.redirectUri
      };
      const formBody = new URLSearchParams(params).toString();

      const response = await axios.post(this.config.tokenEndpoint, formBody, {
        headers: {
          'Content-Type': 'application/x-www-form-urlencoded',
          'Cache-Control': 'no-cache'
        }
      });

      if (response.data.error || !response.data.access_token) {
        throw new Error(response.data.error_description || response.data.error || 'Token
exchange failed');
      }

      const { access_token, refresh_token, expires_in, open_id } = response.data;
      const expiryTime = Date.now() + expires_in * 1000;

      await this.storeTokens(userId, {
        accessToken: access_token,
        refreshToken: refresh_token,
        expiryTime: expiryTime,
        platformUserId: open_id
      });

      return { success: true };

    } catch (error) {
      const errorMsg = error.response?.data?.error_description ||
error.response?.data?.error || error.message;
      throw new Error(`TikTok token exchange failed: ${errorMsg}`);
    }
  }

  async refreshAccessToken(userId) {
    const currentTokens = await this.getStoredTokens(userId);
```

```
    if (!this.config.clientId || !this.config.clientSecret) {
      throw new Error("TikTok Client ID or Client Secret not configured");
    }

    if (!currentTokens.refreshToken) {
      throw new Error(`No TikTok refresh token available for user ${userId}`);
    }

    try {
      const params = {
        client_key: this.config.clientId,
        client_secret: this.config.clientSecret,
        refresh_token: currentTokens.refreshToken,
        grant_type: "refresh_token"
      };
      const formBody = new URLSearchParams(params).toString();

      const response = await axios.post(this.config.tokenEndpoint, formBody, {
        headers: {
          'Content-Type': 'application/x-www-form-urlencoded',
          'Cache-Control': 'no-cache'
        }
      });

      if (response.data.error || !response.data.access_token) {
        throw new Error(response.data.error_description || response.data.error || 'Token
refresh failed');
      }

      const { access_token, refresh_token, expires_in, open_id } = response.data;
      const expiryTime = Date.now() + expires_in * 1000;

      await this.storeTokens(userId, {
        accessToken: access_token,
        refreshToken: refresh_token || currentTokens.refreshToken,
        expiryTime: expiryTime,
        platformUserId: open_id || currentTokens.platformUserId
      });

      return access_token;

    } catch (error) {
      const errorMsg = error.response?.data?.error_description ||
error.response?.data?.error || error.message;
      await this.clearTokens(userId);
      throw new Error(`Failed to refresh TikTok token: ${errorMsg}. Please re-
authenticate.`);
    }
  }

  async getUserInfo(userId) {
    const accessToken = await this.ensureValidToken(userId);
    const userInfoEndpoint = process.env.VITE_TIKTOK_USERINFO_ENDPOINT ||
'https://open.tiktokapis.com/v2/user/info/';

    try {
      const response = await axios.get(userInfoEndpoint, {
        headers: {
          'Authorization': `Bearer ${accessToken}`
        },
        params: {
          fields: 'open_id,display_name,avatar_url'
        }
      });
```

```
      const tikTokError = response.data.error;
      if (tikTokError && tikTokError.code !== 'ok') {
        throw new Error(tikTokError.message || 'Failed to fetch user info from TikTok');
      }

      if (!response.data.data?.user) {
        throw new Error('User data not found in TikTok response.');
      }

      return response.data;
    } catch (error) {
      if (error.message?.includes('re-authenticate') ||
          error.message?.includes('token') ||
          error.message?.includes('PERMISSION_DENIED')) {
        await this.clearTokens(userId);
        throw new Error(`Authentication error: ${error.message}. Please reconnect TikTok.`);
      }
      throw error;
    }
  }
}

const tiktokAuthService = new TikTokAuthService();
module.exports = tiktokAuthService;

// facebookAuthService.js

const axios = require('axios');
const BaseService = require('./baseService');

const PLATFORM = 'facebook';
const clientId = process.env.FACEBOOK_APP_ID;
const clientSecret = process.env.FACEBOOK_APP_SECRET;
const redirectUri = process.env.FACEBOOK_REDIRECT_URI;

class FacebookAuthService extends BaseService {
  constructor() {
    super(PLATFORM, { clientId, clientSecret, redirectUri });
  }

  getFacebookAuthUrl(userId) {
    if (!this.config.clientId || !this.config.redirectUri) {
      throw new Error('Facebook OAuth config missing');
    }

    const stateData = Buffer.from(JSON.stringify({ userId })).toString('base64');
    const params = new URLSearchParams({
      client_id: this.config.clientId,
      redirect_uri: this.config.redirectUri,
      state: stateData,
      scope: 'email,public_profile,pages_show_list',
      response_type: 'code'
    });

    return `https://www.facebook.com/v22.0/dialog/oauth?${params}`;
  }

  async handleFacebookCallback(userId, code) {
    const tokenRes = await this.makeApiRequest('get',
'https://graph.facebook.com/v22.0/oauth/access_token', {
      params: {
        client_id: this.config.clientId,
        redirect_uri: this.config.redirectUri,
        client_secret: this.config.clientSecret,
        code
```

```
      }
    });

    const accessToken = tokenRes.access_token;

    const longRes = await this.makeApiRequest('get',
'https://graph.facebook.com/v22.0/oauth/access_token', {
      params: {
        grant_type: 'fb_exchange_token',
        client_id: this.config.clientId,
        client_secret: this.config.clientSecret,
        fb_exchange_token: accessToken
      }
    });

    const longToken = longRes.access_token;
    const expiresInRaw = longRes.expires_in;
    const expiresInSec = typeof expiresInRaw === 'number' && !isNaN(expiresInRaw) ?
expiresInRaw : 60 * 24 * 60 * 60;
    const expiryTime = Date.now() + expiresInSec * 1000;

    await this.storeTokens(userId, {
      accessToken: longToken,
      refreshToken: longToken,
      expiryTime
    });

    const profileRes = await this.makeApiRequest('get', 'https://graph.facebook.com/me', {
      params: {
        access_token: longToken,
        fields: 'id,name,email,picture'
      }
    });

    const pagesRes = await this.makeApiRequest('get',
'https://graph.facebook.com/me/accounts', {
      params: {
        access_token: longToken
      }
    });

    return { user: profileRes, pages: pagesRes.data };
  }

  async getUserInfo(userId) {
    const accessToken = await this.ensureValidToken(userId);

    try {
      const res = await this.makeApiRequest('get', 'https://graph.facebook.com/me', {
        params: {
          access_token: accessToken,
          fields: 'id,name,email,picture'
        }
      });

      return res;
    } catch (err) {
      throw new Error('Failed to fetch Facebook profile');
    }
  }

  async getPages(userId) {
    const accessToken = await this.ensureValidToken(userId);

    try {
```

```
      const res = await this.makeApiRequest('get', 'https://graph.facebook.com/me/accounts',
{
        params: {
          access_token: accessToken
        }
      });

      return res.data;
    } catch (err) {
      throw new Error('Failed to fetch Facebook pages');
    }
  }

  async refreshAccessToken(userId) {
    const tokens = await this.getStoredTokens(userId);
    return tokens.accessToken;
  }
}

const facebookAuthService = new FacebookAuthService();
module.exports = facebookAuthService;

// twitterAuthService.js

const axios = require('axios');
const BaseService = require('./baseService');

const PLATFORM = 'twitter';
const clientId = process.env.TWITTER_CLIENT_ID;
const clientSecret = process.env.TWITTER_CLIENT_SECRET;
const redirectUri = process.env.TWITTER_REDIRECT_URI;

class TwitterAuthService extends BaseService {
  constructor() {
    super(PLATFORM, { clientId, clientSecret, redirectUri });
  }

  getTwitterAuthUrl(userId) {
    const stateData = JSON.stringify({ userId, nonce:
Math.random().toString(36).substring(7) });
    const state = Buffer.from(stateData).toString('base64');
    const scopes = [
      'tweet.read',
      'tweet.write',
      'users.read',
      'offline.access',
      'media.write'
    ];
    const params = new URLSearchParams({
      response_type: 'code',
      client_id: this.config.clientId,
      redirect_uri: this.config.redirectUri,
      scope: scopes.join(' '),
      state,
      code_challenge: 'challenge',
      code_challenge_method: 'plain',
    });
    return `https://twitter.com/i/oauth2/authorize?${params.toString()}`;
  }

  async handleTwitterCallback(userId, code) {
    const params = new URLSearchParams({
      code,
      grant_type: 'authorization_code',
      client_id: this.config.clientId,
```

```
      redirect_uri: this.config.redirectUri,
      code_verifier: 'challenge',
    });

    const basicAuth =
Buffer.from(`${this.config.clientId}:${this.config.clientSecret}`).toString('base64');

    try {
      const response = await axios.post('https://api.twitter.com/2/oauth2/token', params, {
        headers: {
          'Content-Type': 'application/x-www-form-urlencoded',
          'Authorization': `Basic ${basicAuth}`,
        },
      });

      const { access_token, refresh_token, expires_in } = response.data;

      await this.storeTokens(userId, {
        accessToken: access_token,
        refreshToken: refresh_token,
        expiryTime: Date.now() + expires_in * 1000
      });

      return { access_token, refresh_token };
    } catch (error) {
      throw new Error(error.response?.data?.error || 'Twitter authentication failed');
    }
  }

  async refreshAccessToken(userId) {
    const tokens = await this.getStoredTokens(userId);
    if (!tokens.refreshToken) {
      throw new Error('No refresh token available for Twitter');
    }

    const params = new URLSearchParams();
    params.append('refresh_token', tokens.refreshToken);
    params.append('grant_type', 'refresh_token');
    params.append('client_id', this.config.clientId);

    const basicAuth =
Buffer.from(`${this.config.clientId}:${this.config.clientSecret}`).toString('base64');

    try {
      const response = await axios.post(
        'https://api.twitter.com/2/oauth2/token',
        params,
        {
          headers: {
            'Content-Type': 'application/x-www-form-urlencoded',
            'Authorization': `Basic ${basicAuth}`
          }
        }
      );

      const { access_token, refresh_token, expires_in } = response.data;

      await this.storeTokens(userId, {
        accessToken: access_token,
        refreshToken: refresh_token || tokens.refreshToken,
        expiryTime: Date.now() + expires_in * 1000
      });

      return access_token;
    } catch (error) {
```

```
      await this.clearTokens(userId);
      throw new Error('Twitter token expired or invalid. Please re-authenticate.');
    }
  }

  async getUserInfo(userId) {
    const tokens = await this.getStoredTokens(userId);
    if (!tokens.accessToken) {
      throw new Error('Not authenticated with Twitter');
    }

    try {
      const response = await
axios.get(`https://api.twitter.com/2/users/me?user.fields=profile_image_url`, {
        headers: { Authorization: `Bearer ${tokens.accessToken}` }
      });

      const user = response.data.data;
      return {
        id: user.id,
        name: user.name,
        username: user.username,
        avatarUrl: user.profile_image_url
      };
    } catch (error) {
      throw new Error(error.response?.data || error.message || 'Failed to fetch Twitter user
info');
    }
  }
}

const twitterAuthService = new TwitterAuthService();
module.exports = twitterAuthService;

// User.js

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  googleId: {
    type: String,
    required: true,
    index: true,
  },
  email: {
    type: String,
    required: true,
    unique: true,
    index: true,
  },
  name: {
    type: String,
    required: true,
  },
  firstName: String,
  lastName: String,
  profilePicture: String,
  accessToken: String,
  refreshToken: String,
  expiryTime: Number,
  role: {
    type: String,
    enum: ['user', 'admin'],
    default: 'user',
  },
```

```
  lastLogin: {
    type: Date,
    default: Date.now,
  },
  createdAt: {
    type: Date,
    default: Date.now,
  },
  updatedAt: {
    type: Date,
    default: Date.now,
  },
});

userSchema.pre('save', function(next) {
  this.updatedAt = new Date();
  next();
});

const User = mongoose.model('User', userSchema);

module.exports = User;

// Token.js

const mongoose = require('mongoose');

const tokenSchema = new mongoose.Schema({
  userId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    required: true,
    index: true,
  },

  platform: {
    type: String,
    required: true,
    enum: ['youtube', 'tiktok', 'facebook', 'twitter'],
    index: true,
  },

  accessToken: {
    type: String,
    required: true,
  },

  refreshToken: {
    type: String,
    required: true,
  },

  expiryTime: {
    type: Number,
    required: true,
  },

  platformUserId: {
    type: String,
    sparse: true,
  },

  createdAt: {
    type: Date,
    default: Date.now,
```

```
  },

  updatedAt: {
    type: Date,
    default: Date.now,
  },
});

tokenSchema.index({ userId: 1, platform: 1 }, { unique: true });

tokenSchema.pre('save', function(next) {
  this.updatedAt = new Date();
  next();
});

const Token = mongoose.model('Token', tokenSchema);

module.exports = Token;

// App.tsx

import { useState } from 'react';
import toast, { Toaster } from 'react-hot-toast';

import VideoUploadForm from './components/VideoUploadForm';
import PlatformSelection from './components/PlatformSelection';
import VideoSettings, { DEFAULT_COMMON_SETTING, DEFAULT_FACEBOOK_SETTING } from
'./components/VideoSettings';
import ContentTypeSelection from './components/ContentTypeSelection';
import PostFlow from './components/PostFlow';

import { VideoProvider } from './context/VideoContext';
import { UserAuthProvider } from './context/UserAuthContext';
import platforms from './data/platforms';
import { SelectedPlatform } from './types';
import Header from './components/Header';

function App() {
  const [contentType, setContentType] = useState<'video'|'post'|null>(null);
  const [currentStep, setCurrentStep] = useState(1);
  const [selectedPlatforms, setSelectedPlatforms] = useState<SelectedPlatform[]>([]);

  const startPost = () => setContentType('post');
  const startVideo = () => {
    setContentType('video');
    setCurrentStep(1);
  };

  const handleVideoSubmit = () => {
    setCurrentStep(2);
  };

  const handlePlatformSelect = (platformId: string, isAuthenticated?: boolean) => {
    const platform = platforms.find(p => p.id === platformId);
    if (!platform) return;

    if (selectedPlatforms.some(p => p.id === platformId)) {
      return;
    }

    const defaultSettings = platformId === 'facebook'
        ? { ...DEFAULT_FACEBOOK_SETTING }
        : { ...DEFAULT_COMMON_SETTING };

    setSelectedPlatforms(prev => [
```

```
      ...prev,
      {
        id: platform.id,
        name: platform.name,
        iconName: platform.iconName,
        settings: defaultSettings,
        isAuthenticated: isAuthenticated
      }
    ]);
  };

  const handlePlatformDeselect = (platformId: string) => {
    setSelectedPlatforms(prev => prev.filter(p => p.id !== platformId));
  };

  const handlePlatformsSubmit = () => {
    if (selectedPlatforms.length === 0) {
      toast.error("Please select at least one platform");
      return;
    }
    setCurrentStep(3);
  };

  const goBack = () => {
    if (currentStep > 1) {
      setCurrentStep(currentStep - 1);
    }
  };

  return (
    <div className="min-h-screen bg-gray-50 flex flex-col">
      <Toaster position="top-center" />

      <Header />

      <main className="flex-1 py-6 mb-8">
        <div className="mx-auto max-w-2xl px-4">
          {contentType === null && <ContentTypeSelection onSelect={type => type === 'video'
? startVideo() : startPost()} />}
          {contentType === 'post' && <PostFlow />}

          {contentType === 'video' && (
            <div className="space-y-6">
              {currentStep === 1 && (
                <VideoUploadForm onSubmit={handleVideoSubmit} />
              )}
              {currentStep === 2 && (
                <PlatformSelection
                  platforms={platforms}
                  selectedPlatforms={selectedPlatforms}
                  onSelect={handlePlatformSelect}
                  onDeselect={handlePlatformDeselect}
                  onNext={handlePlatformsSubmit}
                  isActive={true}
                  onBack={currentStep > 1 ? goBack : undefined}
                />
              )}
              {currentStep === 3 && (
                <VideoSettings selectedPlatforms={selectedPlatforms} isActive={true}
onBack={goBack} />
              )}
            </div>
          )}
        </div>
      </main>
```

```
    </div>
  );
}

export default function AppWithProvider() {
  return (
    <UserAuthProvider>
      <VideoProvider>
        <App />
      </VideoProvider>
    </UserAuthProvider>
  );
}

// UserAuthContext.tsx

import { createContext, useState, useEffect, ReactNode, useContext } from 'react';
import toast from 'react-hot-toast';
import axios from 'axios';

export interface User {
  id: string;
  name: string;
  email: string;
  profilePicture: string;
  role: string;
}

interface UserAuthContextType {
  user: User | null;
  isAuthenticated: boolean;
  isLoading: boolean;
  login: () => void;
  logout: () => void;
}

const UserAuthContext = createContext<UserAuthContextType>({
  user: null,
  isAuthenticated: false,
  isLoading: true,
  login: () => {},
  logout: () => {},
});

export const UserAuthProvider = ({ children }: { children: ReactNode }) => {
  const [user, setUser] = useState<User | null>(null);
  const [isLoading, setIsLoading] = useState(true);

  useEffect(() => {
    const checkAuth = async () => {
      const token = localStorage.getItem('accessToken');

      if (!token) {
        setIsLoading(false);
        return;
      }

      try {
        const response = await
axios.get(`${import.meta.env.VITE_API_BASE_URL}/api/users/profile`, {
          headers: {
            Authorization: `Bearer ${token}`,
          },
        });
```

```
      setUser(response.data.user);
    } catch (error) {
      localStorage.removeItem('accessToken');
      localStorage.removeItem('refreshToken');
      toast.error('Your session has expired. Please sign in again.');
    } finally {
      setIsLoading(false);
    }
  };

  checkAuth();
}, []);

useEffect(() => {
  const handleAuthMessage = (event: MessageEvent) => {
    if (!event.data || !event.data.type) return;

    if (event.data.type === 'USER_AUTH_SUCCESS') {
      localStorage.setItem('accessToken', event.data.accessToken);
      localStorage.setItem('refreshToken', event.data.refreshToken);
      setUser(event.data.user);
      toast.success('Signed in successfully!');
    } else if (event.data.type === 'USER_AUTH_ERROR') {
      toast.error(`Login failed: ${event.data.error || 'Unknown error'}`);
    }
  };

  window.addEventListener('message', handleAuthMessage);
  return () => window.removeEventListener('message', handleAuthMessage);
}, []);

const login = () => {
  const width = 600;
  const height = 700;
  const left = window.innerWidth / 2 - width / 2;
  const top = window.innerHeight / 2 - height / 2;

  window.open(
    `${import.meta.env.VITE_API_BASE_URL}/api/users/login/google`,
    'Google Login',
    `width=${width},height=${height},left=${left},top=${top}`
  );
};

const logout = async () => {
  const token = localStorage.getItem('accessToken');

  if (token) {
    try {
      await axios.post(`${import.meta.env.VITE_API_BASE_URL}/api/users/logout`, {}, {
        headers: {
          Authorization: `Bearer ${token}`,
        },
      });
    } catch (error) {
      console.error('Error logging out user:', error);
    }
  }

  localStorage.removeItem('accessToken');
  localStorage.removeItem('refreshToken');
  setUser(null);
  toast.success('Signed out successfully');
};
```

```
  return (
    <UserAuthContext.Provider
      value={{
        user,
        isAuthenticated: !!user,
        isLoading,
        login,
        logout,
      }}
    >
      {children}
    </UserAuthContext.Provider>
  );
};

export const useUserAuth = () => useContext(UserAuthContext);

// VideoContext.tsx

import { createContext, useState, ReactNode, useContext } from 'react';

interface VideoContextType {
  videoFile: File | null;
  setVideoFile: (file: File | null) => void;
  isUploading: boolean;
  setIsUploading: (isUploading: boolean) => void;
  uploadProgress: number;
  setUploadProgress: (progress: number) => void;
}

const VideoContext = createContext<VideoContextType | undefined>(undefined);

export const VideoProvider = ({ children }: { children: ReactNode }) => {
  const [videoFile, setVideoFile] = useState<File | null>(null);
  const [isUploading, setIsUploading] = useState(false);
  const [uploadProgress, setUploadProgress] = useState(0);

  const value = {
    videoFile,
    setVideoFile,
    isUploading,
    setIsUploading,
    uploadProgress,
    setUploadProgress,
  };

  return <VideoContext.Provider value={value}>{children}</VideoContext.Provider>;
};

export const useVideo = (): VideoContextType => {
  const context = useContext(VideoContext);
  if (context === undefined) {
    throw new Error('useVideo must be used within a VideoProvider');
  }
  return context;
};

// PlatformSelection.tsx

import { Check, Info, ArrowLeft, Lock } from 'lucide-react';
import { Platform, SelectedPlatform } from '../types';
import { useVideo } from '../context/VideoContext';
import IconsByName from './PlatformIcons';
import toast from 'react-hot-toast';
import { useYouTubeAuth } from '../hooks/useYouTubeAuth';
```

```
import { useTikTokAuth } from '../hooks/useTikTokAuth';
import { useFacebookAuth } from '../hooks/useFacebookAuth';
import { useTwitterAuth } from '../hooks/useTwitterAuth';

interface PlatformSelectionProps {
  platforms: Platform[];
  selectedPlatforms: SelectedPlatform[];
  onSelect: (platformId: string, isAuthenticated?: boolean) => void;
  onDeselect: (platformId: string) => void;
  onNext: () => void;
  isActive: boolean;
  onBack?: () => void;
  requireVideoSource?: boolean;
}

const renderPlatformIcon = (iconName: string) => {
  const IconComponent = IconsByName[iconName];
  return IconComponent ? (
    <IconComponent className="h-full w-full object-contain" />
  ) : (
    <Info className="h-6 w-6 text-gray-400" />
  );
};

const PlatformSelection = ({
  platforms,
  selectedPlatforms,
  onSelect,
  onDeselect,
  onNext,
  isActive,
  onBack,
  requireVideoSource = true
}: PlatformSelectionProps) => {
  const { videoFile } = useVideo();
  const { isAuthenticated: isYouTubeAuthenticated, login: loginYouTube, logout:
logoutYouTube } = useYouTubeAuth();
  const { isAuthenticated: isTikTokAuthenticated, login: loginTikTok, logout: logoutTikTok }
= useTikTokAuth();
  const { isAuthenticated: isFacebookAuthenticated, login: loginFacebook, logout:
logoutFacebook } = useFacebookAuth();
  const { isAuthenticated: isTwitterAuthenticated, login: loginTwitter, logout:
logoutTwitter } = useTwitterAuth();

  const hasVideoSource = !!videoFile;

  const getAuthStatus = (platformId: string): boolean => {
    if (platformId === 'youtube') return isYouTubeAuthenticated;
    if (platformId === 'tiktok') return isTikTokAuthenticated;
    if (platformId === 'facebook') return isFacebookAuthenticated;
    if (platformId === 'twitter') return isTwitterAuthenticated;
    return false;
  };

  const handleAuthenticate = async (platformId: string) => {
    const platform = platforms.find(p => p.id === platformId);
    if (!platform) return;

    let loginFunction: (() => Promise<void> | void) | null = null;

    if (platformId === 'tiktok') {
      loginFunction = loginTikTok;
    } else if (platformId === 'facebook') {
      loginFunction = loginFacebook;
    } else if (platformId === 'youtube') {
```

```
    loginFunction = loginYouTube;
  } else if (platformId === 'twitter') {
    loginFunction = loginTwitter;
  } else {
    toast.error(`Authentication for ${platform.name} not implemented.`);
    return;
  }

  if (!loginFunction) {
    toast.error(`Login function not found for ${platform.name}.`);
    return;
  }

  const loadingToast = toast.loading(`Connecting to ${platform.name}...`);

  try {
    await loginFunction();
    toast.dismiss(loadingToast);
    setTimeout(() => {
      const nowAuthenticated = getAuthStatus(platformId);
      if (nowAuthenticated && !isPlatformSelected(platformId)) {
        onSelect(platformId, true);
      }
    }, 100);
  } catch (error: any) {
    toast.dismiss(loadingToast);
  }
};

const handleDisconnect = (platformId: string) => {
  const platform = platforms.find(p => p.id === platformId);
  if (!platform) return;

  let logoutFunction: (() => void) | null = null;
  if (platformId === 'youtube') logoutFunction = logoutYouTube;
  else if (platformId === 'tiktok') logoutFunction = logoutTikTok;
  else if (platformId === 'facebook') logoutFunction = logoutFacebook;
  else if (platformId === 'twitter') logoutFunction = logoutTwitter;

  if (logoutFunction) {
    logoutFunction();
    if (isPlatformSelected(platformId)) {
      onDeselect(platformId);
    }
  } else {
    toast.error(`Disconnect for ${platform.name} not implemented.`);
  }
};

const isPlatformSelected = (platformId: string) => {
  return selectedPlatforms.some(p => p.id === platformId);
};

const isPlatformAuthenticated = (platformId: string): boolean => {
  return getAuthStatus(platformId);
};

const handleTogglePlatform = (platformId: string) => {
  const platform = platforms.find(p => p.id === platformId);
  if (!platform) return;

  const requiresAuth = platform.requiresAuth;
  const isAuthenticated = isPlatformAuthenticated(platformId);

  if (requiresAuth && !isAuthenticated) {
```

```
      return;
    }

    if (isPlatformSelected(platformId)) {
      onDeselect(platformId);
    } else {
      onSelect(platformId, requiresAuth ? isAuthenticated : undefined);
    }
  };

  if (!isActive) return null;

  return (
    <div className="overflow-hidden rounded-lg border border-gray-200 bg-white shadow-sm">
      <div className="border-b border-gray-200 bg-gray-50 px-4 py-3">
        <div className="flex items-center justify-between">
          <div className="flex items-center">
            {onBack && (
              <button
                onClick={onBack}
                className="mr-3 flex h-7 w-7 items-center justify-center rounded-md border
border-gray-200 bg-white text-gray-700 shadow-sm hover:bg-gray-50 disabled:cursor-not-
allowed disabled:opacity-50"
                aria-label="Go back"
              >
                <ArrowLeft size={16} />
              </button>
            )}
            <h2 className="text-sm font-medium text-gray-800">Select Content Platforms</h2>
          </div>
        </div>
      </div>

      <div className="relative pb-16">
        <div className="p-4">
          <div className="mb-3 flex justify-between items-center">
            <p className="text-xs text-gray-500">
              {platforms.length} platform{platforms.length !== 1 ? 's' : ''} available
            </p>
          </div>

          {platforms.length > 0 ? (
            <div className="grid grid-cols-2 gap-2 sm:grid-cols-3 md:grid-cols-4">
              {platforms.map((platform) => {
                const isSelected = isPlatformSelected(platform.id);
                const isAuthenticated = isPlatformAuthenticated(platform.id);
                const requiresAuth = platform.requiresAuth;
                const isDisabled = requiresAuth && !isAuthenticated;

                return (
                  <div
                    key={platform.id}
                    className={`group relative flex flex-col rounded-md border transition-
all ${
                      isSelected && !isDisabled
                        ? 'border-indigo-300 bg-indigo-50/50'
                        : isDisabled
                        ? 'border-gray-200 bg-gray-100 opacity-70 cursor-not-allowed'
                        : 'border-gray-200 bg-white hover:border-gray-300 hover:shadow-sm'
                    }`}
                  >
                    <div
                      className={`relative p-2 ${!isDisabled ? 'cursor-pointer' : ''}`}
                      onClick={!isDisabled ? () => handleTogglePlatform(platform.id) :
undefined}
```

```
                    >
                      <div className="flex items-center">
                        <div className={`mr-2 flex h-8 w-8 flex-shrink-0 items-center
justify-center overflow-hidden rounded bg-white p-1 shadow-sm ${
                          isDisabled ? 'opacity-60' : ''
                        }`}>
                          {renderPlatformIcon(platform.iconName)}
                        </div>
                        <div className="overflow-hidden">
                          <h3 className={`truncate text-sm font-medium ${
                            isDisabled ? 'text-gray-500' : 'text-gray-800'
                          }`}>
                            {platform.name}
                          </h3>
                        </div>

                        <div className="absolute right-1.5 top-1.5 flex items-center space-
x-1">
                          {requiresAuth && !isAuthenticated && (
                            <div className="flex h-4 w-4 items-center justify-center
rounded-full bg-gray-200 text-gray-500" title="Connection required">
                              <Lock className="h-2.5 w-2.5" />
                            </div>
                          )}

                          {!isDisabled && (
                            <div className={`flex h-4 w-4 items-center justify-center
rounded-full border transition-colors ${
                              isSelected
                                ? 'border-indigo-600 bg-indigo-600 text-white'
                                : 'border-gray-300 bg-white text-transparent group-
hover:border-gray-400'
                            }`} title={isSelected ? 'Selected' : 'Select'}>
                              <Check className="h-2.5 w-2.5" />
                            </div>
                          )}
                        </div>
                      </div>
                    </div>

                    {requiresAuth && (
                      <div className="mt-auto border-t border-gray-200 p-1.5">
                        {isAuthenticated ? (
                          <button
                            type="button"
                            onClick={(e) => { e.stopPropagation();
handleDisconnect(platform.id); }}
                            className="w-full rounded bg-gray-100 px-2 py-1 text-xs text-
gray-600 hover:bg-gray-200 transition-colors"
                            title={`Disconnect ${platform.name}`}
                          >
                            Disconnect
                          </button>
                        ) : (
                          <button
                            type="button"
                            onClick={(e) => { e.stopPropagation();
handleAuthenticate(platform.id); }}
                            className="w-full rounded bg-indigo-50 px-2 py-1 text-xs font-
semibold text-indigo-700 hover:bg-indigo-100 transition-colors"
                            title={`Connect ${platform.name}`}
                          >
                            Connect
                          </button>
                        )}
```

```
                  </div>
                )}
              </div>
            );
          })}
        </div>
      ) : (
        <div className="flex flex-col items-center justify-center rounded-md bg-gray-50
p-6 text-center">
          <Info className="mb-2 h-8 w-8 text-gray-400" />
          <p className="text-sm text-gray-500">No platforms available to select.</p>
        </div>
      )}
    </div>

    <div className="absolute bottom-0 left-0 right-0 bg-white border-t border-gray-100
p-4 flex items-center justify-between shadow-sm">
      <div className="flex items-start space-x-1">
        <Info className="mt-0.5 h-3.5 w-3.5 flex-shrink-0 text-gray-400" />
        <p className="text-xs text-gray-500">
          Select platforms to continue
        </p>
      </div>
      {/* кнопка "Continue" неактивна якщо нічого не вибрано або нема відео */}
      <button
        type="button"
        onClick={onNext}
        disabled={selectedPlatforms.length === 0 || (requireVideoSource &&
!hasVideoSource)}
        className={`rounded-md px-4 py-2 text-sm font-medium transition-colors shadow-sm
${
          selectedPlatforms.length === 0 || (requireVideoSource && !hasVideoSource)
            ? 'cursor-not-allowed bg-gray-200 text-gray-400'
            : 'bg-indigo-600 text-white hover:bg-indigo-700 focus:outline-none
focus:ring-2 focus:ring-indigo-500 focus:ring-offset-2'
        }`}
      >
        Continue
      </button>
    </div>
  </div>
</div>
);
};

export default PlatformSelection;

// VideoSettings.tsx

import React, { useState, useEffect, useCallback } from 'react';
import { ArrowLeft, Info } from 'lucide-react';
import { useVideo } from '../context/VideoContext';
import {
  SelectedPlatform,
  UploadSettings,
} from '../types';
import IconsByName from './PlatformIcons';
import toast from 'react-hot-toast';

import { useVideoUploader } from '../hooks/useVideoUploader';

import YouTubeSettings from './settings/YouTubeSettings';
import TikTokSettings from './settings/TikTokSettings';
import FacebookSettings from './settings/FacebookSettings';
import TwitterSettings from './settings/TwitterSettings';
```

```
type AllPlatformSettings = Record<string, UploadSettings>;

interface VideoSettingsProps {
  selectedPlatforms: SelectedPlatform[];
  isActive: boolean;
  onBack?: () => void;
}

export const DEFAULT_COMMON_SETTING: UploadSettings = {
  title: '',
  description: '',
  visibility: 'public',
  tags: [],
  scheduledDate: null,
};

export const DEFAULT_FACEBOOK_SETTING: UploadSettings = {
    ...DEFAULT_COMMON_SETTING,
    selectedPageId: ''
};

const VideoSettings: React.FC<VideoSettingsProps> = ({
  selectedPlatforms,
  isActive,
  onBack,
}) => {
  const { videoFile } = useVideo();
  const [platformSettings, setPlatformSettings] = useState<AllPlatformSettings>({});
  const [activePlatform, setActivePlatform] = useState<string | null>(null);
  const [videoPreviewUrl, setVideoPreviewUrl] = useState<string | null>(null);

  const {
    uploadSinglePlatform,
    platformResults,
    resetPlatformStatus,
  } = useVideoUploader();

  // Оновлює налаштування для конкретної платформи
  const handleSettingChange = useCallback((platformId: string, key: keyof UploadSettings,
value: any) => {
    setPlatformSettings(prev => ({
      ...prev,
      [platformId]: {
        ...(prev[platformId] || (platformId === 'facebook' ? DEFAULT_FACEBOOK_SETTING :
DEFAULT_COMMON_SETTING)),
        [key]: value,
      },
    }));
  }, []);

  useEffect(() => {
    const newSettings: AllPlatformSettings = {};
    selectedPlatforms.forEach(platform => {
      if (!platformSettings[platform.id]) {
         if (platform.id === 'facebook') {
             newSettings[platform.id] = { ...DEFAULT_FACEBOOK_SETTING };
         } else {
             newSettings[platform.id] = { ...DEFAULT_COMMON_SETTING };
         }
      } else {
         newSettings[platform.id] = platformSettings[platform.id];
      }
    });
    if (Object.keys(newSettings).some(id => !platformSettings[id])) {
```

```
      setPlatformSettings(prev => ({ ...prev, ...newSettings }));
    }

    if (!activePlatform || !selectedPlatforms.some(p => p.id === activePlatform)) {
      setActivePlatform(selectedPlatforms[0]?.id || null);
    }
  }, [JSON.stringify(selectedPlatforms.map(p => p.id))]);

  // Створює URL для перегляду відео та автозаповнює заголовок
  useEffect(() => {
    let objectUrl: string | null = null;
    if (videoFile) {
      objectUrl = URL.createObjectURL(videoFile);
      setVideoPreviewUrl(objectUrl);

      const initialActivePlatformId = selectedPlatforms[0]?.id;
      if (initialActivePlatformId && platformSettings[initialActivePlatformId]) {
          const currentSettings = platformSettings[initialActivePlatformId];
          if (!currentSettings.title) {
              const nameWithoutExtension = videoFile.name.split('.').slice(0, -1).join('.');
              const formattedName = nameWithoutExtension.replace(/[-_]/g, ' ');
              handleSettingChange(initialActivePlatformId, 'title', formattedName);
          }
      }
    }

    return () => {
      if (objectUrl) {
        URL.revokeObjectURL(objectUrl);
        setVideoPreviewUrl(null);
      }
    };
  }, [videoFile, handleSettingChange, selectedPlatforms]);

  // Обробник змін для загальних налаштувань
  const handleCommonSettingsChange = useCallback(<K extends keyof UploadSettings>(key: K,
value: UploadSettings[K]) => {
      if (activePlatform) {
          handleSettingChange(activePlatform, key, value);
      }
  }, [activePlatform, handleSettingChange]);

  // Обробник змін для Facebook-специфічних налаштувань
  const handleFacebookSettingsChange = useCallback(
    <K extends keyof UploadSettings>(key: K, value: UploadSettings[K]) => {
       if (activePlatform === 'facebook') {
           handleSettingChange(activePlatform, key, value);
       }
   },
   [activePlatform, handleSettingChange]
  );

  // Копіює налаштування з активної платформи на всі інші
  const copySettingsToAll = useCallback(() => {
    if (!activePlatform) return;

    const sourceSettings = platformSettings[activePlatform];
    if (!sourceSettings) return;

    const updateToast = toast.loading('Copying settings...')
    setPlatformSettings(prev => {
      const newSettings = { ...prev };
      selectedPlatforms.forEach(platform => {
         if (platform.id !== activePlatform) {
```

```
            const defaultSetting = platform.id === 'facebook' ? DEFAULT_FACEBOOK_SETTING :
DEFAULT_COMMON_SETTING;
            newSettings[platform.id] = {
               ...(prev[platform.id] || defaultSetting),
               title: sourceSettings.title,
               description: sourceSettings.description,
               visibility: sourceSettings.visibility,
               tags: [...(sourceSettings.tags || [])],
               scheduledDate: sourceSettings.scheduledDate,
               selectedPageId: platform.id === 'facebook' ?
(prev[platform.id]?.selectedPageId || '') : undefined
            };
         }
      });
      return newSettings;
    });
    toast.success('Settings copied to other platforms.', { id: updateToast });

  }, [activePlatform, platformSettings, selectedPlatforms]);

  const renderPlatformIcon = (iconName: string) => {
    const IconComponent = IconsByName[iconName];
    return IconComponent ? (
      <IconComponent className="h-full w-full" />
    ) : (
      <Info className="h-full w-full text-gray-700" />
    );
  };

  if (!isActive) return null;

  const currentActivePlatformData = selectedPlatforms.find(p => p.id === activePlatform);

  return (
    <div className="overflow-hidden rounded-lg border border-gray-200 bg-white shadow-sm">
      <div className="border-b border-gray-200 bg-gray-50 px-4 py-3">
        <div className="flex items-center">
          {onBack && (
            <button
              onClick={onBack}
              className="mr-3 flex h-7 w-7 items-center justify-center rounded-md border
border-gray-200 bg-white text-gray-700 shadow-sm hover:bg-gray-50 disabled:cursor-not-
allowed disabled:opacity-50"
              aria-label="Go back"
            >
              <ArrowLeft size={16} />
            </button>
          )}
          <h2 className="text-sm font-medium text-gray-800">Configure Upload Settings</h2>
        </div>
      </div>

      <div className="p-4">
        {selectedPlatforms.length > 0 ? (
          <div className="flex flex-col space-y-4">
            <div className="overflow-x-auto border-b border-gray-200">
              <nav className="-mb-px flex space-x-2" aria-label="Tabs">
                {selectedPlatforms.map(platform => (
                  <button
                    key={platform.id}
                    type="button"
                    onClick={() => setActivePlatform(platform.id)}
                    className={`flex items-center whitespace-nowrap border-b-2 px-2 py-2
text-sm font-medium transition disabled:cursor-not-allowed disabled:opacity-70 ${
                      activePlatform === platform.id
```

```
                        ? 'border-indigo-500 text-indigo-600'
                        : 'border-transparent text-gray-500 hover:border-gray-300
hover:text-gray-700'
                    }`}
                    aria-current={activePlatform === platform.id ? 'page' : undefined}
                  >
                    <div className="mr-1.5 h-5 w-5 flex-shrink-0 overflow-hidden rounded-
sm">
                      {renderPlatformIcon(platform.iconName)}
                    </div>
                    <span>{platform.name}</span>
                    {platformResults[platform.id]?.status &&
platformResults[platform.id]?.status !== 'pending' && (
                        <span className={`ml-1.5 h-2 w-2 rounded-full ${
                            platformResults[platform.id]?.status === 'success' ? 'bg-green-
400' :
                            platformResults[platform.id]?.status === 'error' ? 'bg-red-400'
:
                            'bg-blue-400'
                        }`}></span>
                    )}
                  </button>
                ))}
              </nav>
            </div>

            <div className="min-h-[200px]">
               {currentActivePlatformData && activePlatform &&
platformSettings[activePlatform] ? (
                    <>
                        {activePlatform === 'youtube' && (
                            <YouTubeSettings
                                platform={currentActivePlatformData}
                                settings={platformSettings[activePlatform]}
                                onSettingsChange={handleCommonSettingsChange}
                                videoFile={videoFile}
                                videoPreviewUrl={videoPreviewUrl}
                                uploadResult={platformResults.youtube}
                                onCopySettings={copySettingsToAll}
                                uploadSinglePlatform={uploadSinglePlatform}
                                resetPlatformStatus={resetPlatformStatus}
                                renderEmbed={(platformId, result) =>
                                   result.videoId ? (
                                     <div className="mt-3 aspect-video w-full overflow-
hidden rounded-md border border-gray-200 bg-black">
                                       <iframe
                                         width="100%"
                                         height="100%"

src={`https://www.youtube.com/embed/${result.videoId}`}
                                         title="YouTube video player"
                                         frameBorder="0"
                                         allow="accelerometer; autoplay; clipboard-write;
encrypted-media; gyroscope; picture-in-picture"
                                         allowFullScreen
                                       ></iframe>
                                     </div>
                                   ) : null
                                }
                            />
                        )}
                        {activePlatform === 'tiktok' && (
                            <TikTokSettings
                                platform={currentActivePlatformData}
                                videoFile={videoFile}
```

```
                                videoPreviewUrl={videoPreviewUrl}
                                uploadResult={platformResults.tiktok}
                                uploadSinglePlatform={uploadSinglePlatform}
                                resetPlatformStatus={resetPlatformStatus}
                            />
                        )}
                        {activePlatform === 'facebook' && (
                            <FacebookSettings
                                platform={currentActivePlatformData}
                                settings={platformSettings[activePlatform]}
                                onSettingsChange={handleFacebookSettingsChange}
                                videoFile={videoFile}
                                videoPreviewUrl={videoPreviewUrl}
                                uploadResult={platformResults.facebook}
                                uploadSinglePlatform={uploadSinglePlatform}
                                resetPlatformStatus={resetPlatformStatus}
                                onCopySettings={copySettingsToAll}
                            />
                        )}
                        {activePlatform === 'twitter' && (
                            <TwitterSettings
                                platform={currentActivePlatformData}
                                settings={platformSettings[activePlatform]}
                                onSettingsChange={handleCommonSettingsChange}
                                videoFile={videoFile}
                                videoPreviewUrl={videoPreviewUrl}
                                uploadResult={platformResults.twitter}
                                uploadSinglePlatform={uploadSinglePlatform}
                                resetPlatformStatus={resetPlatformStatus}
                                onCopySettings={copySettingsToAll}
                            />
                        )}
                    </>
                ) : !activePlatform ? (
                    <div className="flex h-full items-center justify-center text-gray-
500">Select a platform tab.</div>
                ) : (
                    <div className="flex h-full items-center justify-center text-gray-
500">Loading settings...</div>
                )}
            </div>
          </div>
        ) : (
          <div className="flex flex-col items-center justify-center rounded-md bg-gray-50 p-
6 text-center">
            <Info className="mb-2 h-8 w-8 text-gray-400" />
            <p className="text-sm text-gray-500">No platforms selected.</p>
            <p className="mt-1 text-xs text-gray-400">Go back to select platforms to upload
to.</p>
          </div>
        )}
      </div>
    </div>
  );
};

export default VideoSettings;

// PostFlow.tsx

import React, { useState } from 'react';
import PlatformSelection from './PlatformSelection';
import PlatformUploadStatus from './settings/PlatformUploadStatus';
import { ArrowLeft, RotateCcw } from 'lucide-react';
import { useFacebookAuth } from '../hooks/useFacebookAuth';
```

```
import { useTwitterAuth } from '../hooks/useTwitterAuth';
import AuthStatusDisplay from './settings/AuthStatusDisplay';
import IconsByName from './PlatformIcons';
import { usePostPublisher } from '../hooks/usePostPublisher';
import platformsList from '../data/platforms';
import { SelectedPlatform } from '../types';
import FacebookPostForm from './post/FacebookPostForm';
import TwitterPostForm from './post/TwitterPostForm';

const PostFlow: React.FC = () => {
  const postPlatforms = platformsList.filter(p => p.id === 'facebook' || p.id === 'twitter');
  const [step, setStep] = useState<'select'|'compose'>('select');
  const [selectedPlatforms, setSelectedPlatforms] = useState<SelectedPlatform[]>([]);
  const { publishSingle, results } = usePostPublisher();
  const [activePlatform, setActivePlatform] = useState<string | null>(null);
  const handleSelect = (platformId: string, isAuthenticated?: boolean) => {
    const platform = postPlatforms.find(p => p.id === platformId);
    if (platform) {
      setSelectedPlatforms(prev => {
        if (prev.some(p => p.id === platformId)) return prev;
        return [
          ...prev,
          {
            ...platform,
            settings: {
              title: '',
              description: '',
              visibility: 'public',
              tags: [],
              scheduledDate: null
            },
            isAuthenticated
          }
        ];
      });
      setActivePlatform(prev => prev || platformId);
    }
  };
  const handleDeselect = (platformId: string) => {
    setSelectedPlatforms(prev => prev.filter(p => p.id !== platformId));
    setActivePlatform(prev => (prev === platformId ? (selectedPlatforms.length > 1 ?
selectedPlatforms.find(p => p.id !== platformId)?.id || null : null) : prev));
  };
  const handlePostSubmit = (payload: any) => {
    if (activePlatform) {
      const platform = selectedPlatforms.find(p => p.id === activePlatform);
      if (platform) {
        publishSingle(platform, payload);
      }
    }
  };
  const { isAuthenticated: isFacebookAuth, isLoading: isFacebookLoading, userInfo:
facebookUserInfo, error: facebookError, logout: facebookLogout } = useFacebookAuth();
  const { isAuthenticated: isTwitterAuth, isLoading: isTwitterLoading, userInfo:
twitterUserInfo, error: twitterError, logout: twitterLogout } = useTwitterAuth();
  const FacebookIcon = IconsByName['facebook'];
  const TwitterIcon = IconsByName['twitter'];
  const facebookAuthInfo = facebookUserInfo
    ? { name: facebookUserInfo.name, id: facebookUserInfo.id, imageUrl:
facebookUserInfo.picture?.data?.url, profileUrl:
`https://facebook.com/${facebookUserInfo.id}` }
    : null;
  const twitterAuthInfo = twitterUserInfo
```

```
    ?     {     name:     twitterUserInfo.name,     id:     twitterUserInfo.id,     imageUrl:
twitterUserInfo.avatarUrl,          profileUrl:          twitterUserInfo.username          ?
`https://twitter.com/${twitterUserInfo.username}` : undefined }
    : null;

  if (step === 'select') {
    return (
      <PlatformSelection
        platforms={postPlatforms}
        selectedPlatforms={selectedPlatforms}
        onSelect={handleSelect}
        onDeselect={handleDeselect}
        onNext={() => setStep('compose')}
        isActive={true}
        requireVideoSource={false}
      />
    );
  }

  return (
    <div className="overflow-hidden rounded-lg border border-gray-200 bg-white shadow-sm">
      <div className="border-b border-gray-200 bg-gray-50 px-4 py-3 flex items-center">
        <button onClick={() => setStep('select')} className="mr-3 flex h-7 w-7 items-center
justify-center rounded-md border border-gray-200 bg-white text-gray-700 shadow-sm hover:bg-
gray-50">
          <ArrowLeft size={16} />
        </button>
        <h2 className="text-sm font-medium text-gray-800">Configure Post Settings</h2>
      </div>
      <div className="p-4 space-y-4">
        <div className="overflow-x-auto border-b border-gray-200">
          <nav className="-mb-px flex space-x-2" aria-label="Tabs">
            {selectedPlatforms.map(platform => (
              <button
                key={platform.id}
                type="button"
                onClick={() => setActivePlatform(platform.id)}
                className={`flex items-center whitespace-nowrap border-b-2 px-2 py-2 text-sm
font-medium transition ${activePlatform === platform.id ? 'border-indigo-500 text-indigo-600'
: 'border-transparent text-gray-500 hover:border-gray-300 hover:text-gray-700'}`}
                aria-current={activePlatform === platform.id ? 'page' : undefined}
              >
                <div className="mr-1.5 h-5 w-5 flex-shrink-0 overflow-hidden rounded-sm">
                  {IconsByName[platform.iconName]                                          &&
React.createElement(IconsByName[platform.iconName], { className: 'h-full w-full' })}
                </div>
                <span>{platform.name}</span>
              </button>
            ))}
          </nav>
        </div>
        {activePlatform === 'facebook' && (
          <AuthStatusDisplay
            platformName="Facebook"
            platformIcon={FacebookIcon ? <FacebookIcon className="h-full w-full"/> : null}
            isAuthenticated={isFacebookAuth}
            isLoading={isFacebookLoading}
            authInfo={facebookAuthInfo}
            error={facebookError}
            onLogout={facebookLogout}
          />
        )}
        {activePlatform === 'twitter' && (
          <AuthStatusDisplay
            platformName="Twitter"
```

```
            platformIcon={TwitterIcon ? <TwitterIcon className="h-full w-full"/> : null}
            isAuthenticated={isTwitterAuth}
            isLoading={isTwitterLoading}
            authInfo={twitterAuthInfo}
            error={twitterError}
            onLogout={twitterLogout}
          />
        )}
        <div className="min-h-[200px]">
          {activePlatform && (() => {
            const result = results[activePlatform];
            if (!result || result.status === 'pending') {
              if (activePlatform === 'facebook') {
                return <FacebookPostForm onSubmit={handlePostSubmit} isSubmitting={false} />;
              }
              if (activePlatform === 'twitter') {
                return <TwitterPostForm onSubmit={handlePostSubmit} isSubmitting={false} />;
              }
              return null;
            }
            return (
              <React.Fragment>
                <PlatformUploadStatus
                  platformId={activePlatform}
                  result={result}
                />
                {result && (
                  <div className="mt-4 flex items-center justify-end border-t border-gray-100
pt-4">
                    <button
                      type="button"
                      onClick={() => {
                        results[activePlatform] = { ...result, status: 'pending', progress:
0, error: undefined };
                        setActivePlatform(null);               setTimeout(()               =>
setActivePlatform(activePlatform), 0);
                      }}
                      className="flex items-center justify-center rounded-md border border-
gray-300 bg-white px-4 py-2 text-sm font-medium text-gray-700 shadow-sm hover:bg-gray-50"
                    >
                      <RotateCcw className="mr-1.5 h-4 w-4" /> Back to Settings
                    </button>
                  </div>
                )}
              </React.Fragment>
            );
          })()}
        </div>
      </div>
    </div>
  );
};

export default PostFlow;

// usePlatformAuth.ts

import { useState, useEffect, useCallback, useRef } from 'react';
import toast from 'react-hot-toast';
import axios from 'axios';

interface PlatformAuthConfig<TUserInfo> {
  statusUrl: string;
  userInfoUrl: string;
  loginUrl: string;
```

```
  logoutUrl: string;
  postMessageSuccessType: string;
  postMessageErrorType: string;
  extractUserInfo: (data: any) => TUserInfo | null;
  platformName: string;
}

export function usePlatformAuth<TUserInfo>(config: PlatformAuthConfig<TUserInfo>) {
  const popupRef = useRef<Window | null>(null);
  const [isAuthenticated, setIsAuthenticated] = useState(false);
  const [isLoading, setIsLoading] = useState(true);
  const [userInfo, setUserInfo] = useState<TUserInfo | null>(null);
  const [error, setError] = useState<string | null>(null);

  const fetchStatus = useCallback(async () => {
    setIsLoading(true);
    setError(null);
    try {
      const token = localStorage.getItem('accessToken');
      const headers: Record<string, string> = {};
      if (token) {
        headers['Authorization'] = `Bearer ${token}`;
      }

      const response = await axios.get(config.statusUrl, { headers });
      setIsAuthenticated(response.data.isAuthenticated);

      if (response.data.isAuthenticated) {
        await fetchUserInfo();
      }
    } catch (err) {
      let errorMessage = 'Failed to check status';
      if (axios.isAxiosError(err)) {
        errorMessage = err.response?.data?.error || err.message || errorMessage;
      } else if (err instanceof Error) {
        errorMessage = err.message;
      }
      setError(errorMessage);
      setIsAuthenticated(false);
    } finally {
      setIsLoading(false);
    }
  }, []);

  const fetchUserInfo = useCallback(async () => {
    try {
      const token = localStorage.getItem('accessToken');
      const headers: Record<string, string> = {};
      if (token) {
        headers['Authorization'] = `Bearer ${token}`;
      }

      const response = await axios.get(config.userInfoUrl, { headers });
      setUserInfo(config.extractUserInfo(response.data));
    } catch (err) {
      let errorMessage = 'Failed to fetch user';
      if (axios.isAxiosError(err)) {
        errorMessage = err.response?.data?.error || err.message || errorMessage;
      } else if (err instanceof Error) {
        errorMessage = err.message;
      }
      setError(errorMessage);
      setUserInfo(null);
    }
  }, []);
```

```
  const logout = useCallback(async () => {
    try {
      const token = localStorage.getItem('accessToken');
      const headers: Record<string, string> = {};
      if (token) {
        headers['Authorization'] = `Bearer ${token}`;
      }

      await axios.post(config.logoutUrl, {}, { headers });
      setIsAuthenticated(false);
      setUserInfo(null);
      toast.success(`Logged out from ${config.platformName}`);
    } catch (err) {
      toast.error(`Logout failed, but cleared local state`);
      setIsAuthenticated(false);
      setUserInfo(null);
    }
  }, []);

  const login = useCallback(async () => {
    setError(null);
    try {
      const token = localStorage.getItem('accessToken');
      const headers: Record<string, string> = {};
      if (token) {
        headers['Authorization'] = `Bearer ${token}`;
      }

      const response = await axios.get(config.loginUrl, { headers });
      const { authUrl } = response.data;

      const width = 600, height = 700;
      const left = window.screenX + (window.outerWidth - width) / 2;
      const top = window.screenY + (window.outerHeight - height) / 2;

      if (popupRef.current && !popupRef.current.closed) {
        popupRef.current.close();
      }

      popupRef.current       =       window.open(authUrl,       `${config.platformName}Auth`,
`width=${width},height=${height},top=${top},left=${left}`);
    } catch (err) {
      let errorMessage = `Failed to initiate ${config.platformName} login`;
      if (axios.isAxiosError(err)) {
        errorMessage = err.response?.data?.error || err.message || errorMessage;
      }
      toast.error(errorMessage);
    }
  }, []);

  useEffect(() => {
    fetchStatus();
  }, [fetchStatus]);

  useEffect(() => {
    const handleMessage = (event: MessageEvent) => {
      if (event.origin !== import.meta.env.VITE_API_BASE_URL) return;
      if (event.data?.type === config.postMessageSuccessType) {
        fetchStatus();
        toast.success(`Connected to ${config.platformName}!`);
      } else if (event.data?.type === config.postMessageErrorType) {
        setError(event.data.error);
        toast.error(`${config.platformName} login failed: ${event.data.error}`);
      }
```

```
      if (popupRef.current && !popupRef.current.closed) popupRef.current.close();
    };
    window.addEventListener('message', handleMessage);
    return () => { window.removeEventListener('message', handleMessage); };
  }, [fetchStatus]);

  return {
    isAuthenticated,
    isLoading,
    userInfo,
    error,
    login,
    logout,
    fetchStatus,
  };
}

// useVideoUploader.ts

import { useState, useCallback } from 'react';
import toast from 'react-hot-toast';
import { useVideo } from '../context/VideoContext';
import {
  SelectedPlatform,
  UploadSettings,
  PlatformUploadResult
} from '../types';
import { uploadTikTokVideo } from '../services/tiktokApiService';
import { uploadTwitterPost } from '../services/twitterApiService';
import { uploadYouTubeVideo } from '../services/youtubeApiService';
import { uploadFacebookVideo } from '../services/facebookApiService';

type PlatformID = 'youtube' | 'tiktok' | 'facebook' | 'twitter';

export function useVideoUploader() {
  const { videoFile } = useVideo();
  const      [platformResults,      setPlatformResults]      =      useState<Record<string,
PlatformUploadResult>>({});

  const updateResult = (platformId: string, data: Partial<PlatformUploadResult>) => {
    setPlatformResults(prev => {
      const current = prev[platformId] || { status: 'pending', progress: 0 };
      const updated = { ...current, ...data };
      return { ...prev, [platformId]: updated };
    });
  };

  const updateProgress = (platformId: string, progress: number) => {
    updateResult(platformId, { progress });
  };

  const uploadSinglePlatform = useCallback(async (
    platform: SelectedPlatform,
    platformSetting: UploadSettings
  ) => {
    if (!videoFile) {
      toast.error('No video file selected for upload.');
      return;
    }

    const platformId = platform.id as PlatformID;

    if   (platformResults[platformId]?.status   &&   platformResults[platformId].status   !==
'pending' && platformResults[platformId].status !== 'error') {
      toast.error(`Upload already processing for ${platform.name}.`);
```

```
      return;
    }

    let settingsValid = true;
    if (!platformSetting.title) {
      toast.error(`Title is missing for ${platform.name}.`);
      settingsValid = false;
    }
    if (platformId === 'facebook' && !platformSetting.selectedPageId) {
      toast.error(`Facebook Page not selected.`);
      settingsValid = false;
    }
    if (platformId === 'youtube' && platformSetting.visibility === 'scheduled' &&
!platformSetting.scheduledDate) {
      toast.error(`Scheduled date missing for YouTube.`);
      settingsValid = false;
    }
    if (!settingsValid) {
      updateResult(platformId, { status: 'error', error: 'Configuration validation failed'
});
      return;
    }

    const token = localStorage.getItem('accessToken');
    const sessionId = token || undefined;

    updateResult(platformId, { status: 'uploading', progress: 0, error: undefined });
    const uploadToastId = toast.loading(`Starting upload to ${platform.name}...`);

    try {
      let videoId: string | undefined;
      let videoUrl: string | undefined;

      if (platformId === 'youtube') {
        const result = await uploadYouTubeVideo(
          videoFile,
          {
            title: platformSetting.title,
            description: platformSetting.description,
            tags: platformSetting.tags,
            visibility: platformSetting.visibility as 'public' | 'private' | 'unlisted' |
'scheduled',
            scheduledDate: platformSetting.scheduledDate || undefined
          },
          sessionId,
          (progress: number) => updateProgress(platformId, progress)
        );

        if ('error' in result) {
          throw new Error(result.error);
        }

        videoId = result.videoId;
        videoUrl = result.videoUrl;

      } else if (platformId === 'tiktok') {
        const result = await uploadTikTokVideo(
          videoFile,
          sessionId,
          platformSetting.title,
          platformSetting.description || '',
          (progress: number) => updateProgress(platformId, progress)
        );

        if ('error' in result) {
```

```
        throw new Error(result.error);
      }

      toast.success(result.message || 'Successfully sent video to TikTok Inbox!', { id:
uploadToastId });
      videoId = `tiktok_inbox_${Date.now()}`;

    } else if (platformId === 'facebook') {
      const result = await uploadFacebookVideo(
        videoFile,
        {
          title: platformSetting.title,
          description: platformSetting.description,
          selectedPageId: platformSetting.selectedPageId || ''
        },
        sessionId,
        (progress: number) => updateProgress(platformId, progress)
      );

      if ('error' in result) {
        throw new Error(result.error);
      }

      videoId = result.videoId;
      videoUrl = result.videoUrl;

    } else if (platformId === 'twitter') {
      const tweetText = platformSetting.title || '';
      const result = await uploadTwitterPost(
        videoFile,
        tweetText,
        sessionId,
        (progress: number) => updateProgress(platformId, progress)
      );

      if ('error' in result) {
        throw new Error(result.error);
      }

      toast.success('Successfully uploaded video to Twitter!', { id: uploadToastId });
      videoId = result.tweet?.data?.id || undefined;
      videoUrl = result.tweet?.data?.id ?
`https://twitter.com/i/web/status/${result.tweet.data.id}` : undefined;
    } else {
      toast.error(`Platform ${platform.name} uploads not implemented yet.`, { id:
uploadToastId });
      throw new Error(`Platform ${platformId} not supported yet.`);
    }

    updateResult(platformId, { status: 'success', videoId, videoUrl, progress: 100 });
    toast.success(`Successfully uploaded to ${platform.name}!`, { id: uploadToastId });

  } catch (error) {
    const errorMessage = error instanceof Error ? error.message : 'Unknown upload error';
    updateResult(platformId, { status: 'error', error: errorMessage, progress: 0 });
    toast.error(`Failed to upload to ${platform.name}: ${errorMessage}`, { id: uploadToastId
});
  }

}, [videoFile, platformResults, updateResult, updateProgress]);

const resetPlatformStatus = useCallback((platformId: string) => {
  updateResult(platformId, {
    status: 'pending',
    progress: 0,
```

```
      videoId: undefined,
      videoUrl: undefined,
      error: undefined
    });
  }, [updateResult]);

  return {
    uploadSinglePlatform,
    platformResults,
    resetPlatformStatus
  };
}

// usePostPublisher.ts

import { useState, useCallback } from 'react';
import { SelectedPlatform, PlatformUploadResult } from '../types';
import { uploadTwitterPost } from '../services/twitterApiService';
import { createFacebookPost, uploadFacebookPhoto } from '../services/facebookApiService';

interface PostPayload {
  pageId?: string;
  message: string;
  imageFile?: File;
  videoFile?: File;
  published?: boolean;
  scheduledPublishTime?: number;
}

export function usePostPublisher() {
  const [results, setResults] = useState<Record<string, PlatformUploadResult>>({});

  const updateResult = (platformId: string, data: Partial<PlatformUploadResult>) => {
    setResults(prev => {
      const current = prev[platformId] || { status: 'pending', progress: 0 };
      return { ...prev, [platformId]: { ...current, ...data } };
    });
  };

  const publishSingle = useCallback(async (
    platform: SelectedPlatform,
    payload: PostPayload
  ) => {
    const platformId = platform.id;
    updateResult(platformId, { status: 'uploading', progress: 0, error: undefined });
    try {
      const token = localStorage.getItem('accessToken');

      if (platformId === 'facebook') {
        if (payload.imageFile) {
          const result = await uploadFacebookPhoto(
            payload.imageFile,
            {
              pageId: payload.pageId || '',
              message: payload.message,
              published: payload.published,
              scheduledPublishTime: payload.scheduledPublishTime
            },
            token || undefined
          );

          if ('error' in result) {
            throw new Error(result.error);
          }
```

```
          updateResult(platformId, {
            status: 'success',
            videoId: result.photoId,
            videoUrl: result.postId ? `https://facebook.com/${result.postId}` : undefined,
            progress: 100
          });
        } else {
          const result = await createFacebookPost(
            {
              pageId: payload.pageId || '',
              message: payload.message,
              published: payload.published,
              scheduledPublishTime: payload.scheduledPublishTime
            },
            token || undefined
          );

          if ('error' in result) {
            throw new Error(result.error);
          }

          updateResult(platformId, {
            status: 'success',
            videoId: result.postId,
            videoUrl: `https://facebook.com/${result.postId}`,
            progress: 100
          });
        }
      } else if (platformId === 'twitter') {
        const file = payload.imageFile || payload.videoFile;
        const twitterRes = await uploadTwitterPost(file, payload.message, token || undefined);

        if ('error' in twitterRes) {
          throw new Error(twitterRes.error);
        }

        updateResult(platformId, {
          status: 'success',
          videoUrl:twitterRes.tweet?.data?.id?
`https://twitter.com/i/web/status/${twitterRes.tweet.data.id}` : undefined,
          videoId: twitterRes.tweet?.data?.id,
          progress: 100
        });
      }
    } catch (e: any) {
      updateResult(platformId, { status: 'error', error: e.message });
    }
  }, []);

  return { publishSingle, results };
}
```