ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка програмного забезпечення для автоматизації роботи з графіком чергувань персоналу підприємства»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Ірина ПЕРНАТА
(підпис)

Виконала: здобувачка вищої освіти групи ПД-41

Ірина ПЕРНАТА

Керівник: Оксана ЗОЛОТУХІНА
канд. техн. наук., доц.

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Пернатій Ірині Валеріївні

1. Тема кваліфікаційної роботи: «Розробка програмного забезпечення для автоматизації роботи з графіком чергувань персоналу підприємства»
керівник кваліфікаційної роботи канд. техн. наук, доц. Оксана ЗОЛОТУХІНА,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року № 51.

2. Строк подання кваліфікаційної роботи «29» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про методи та алгоритми комбінаторної оптимізації, опис евристичних методів автоматизованого складання розкладів та управління трудовими ресурсами підприємства, технічна документація з описом сучасних фреймворків для веброзробки та проєктування реляційних баз даних.

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

1. Огляд та аналіз існуючих методів та систем для автоматизованого планування графіків чергувань.
2. Проєктування програмної системи автоматизованого розподілу графіків робочих змін.

3. Програмна реалізація та опис функціонування веб-додатка для управління розкладами чергувань.
4. Тестування та оцінка ефективності розробленого програмного забезпечення.

5. Перелік графічного матеріалу: *презентація*
1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Технічні засоби реалізації.
4. Діаграма варіантів використання.
5. Діаграма архітектури системи.
6. Діаграми послідовності процесів.
7. ER-діаграма.
8. Діаграма діяльності генерації розкладу на вихідні.
9. Екранні форми.
10. Апробація результатів дослідження.

6. Дата видачі завдання «23» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	23.02-28.02.2026	
2	Аналіз та дослідження існуючих аналогів	01.03-07.03.2026	
3	Огляд існуючих алгоритмів та методів автоматизованого планування графіків чергувань персоналу	08.03-17.03.2026	
4	Проєктування програмної системи автоматизованого розподілу графіків робочих змін	18.03-29.03.2026	
5	Програмна реалізація вебдодатка для управління розкладами чергувань	30.03-19.04.2026	
6	Тестування розробленого програмного забезпечення	20.04-28.04.2026	
7	Оформлення роботи: вступ, висновки, реферат	29.04-05.05.2026	
8	Розробка демонстраційних матеріалів	06.05-10.05.2026	
9	Попередній захист роботи	11.05-22.05.2026	

Здобувачка вищої освіти _____ Ірина ПЕРНАТА
(підпис)

Керівник
кваліфікаційної роботи _____ Оксана ЗОЛОТУХІНА
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 57 стор., 1 табл., 18 рис., 28 джерел.

Мета роботи – підвищення ефективності роботи з графіком чергувань персоналу підприємства шляхом автоматизації формування та коригування графіка чергувань.

Об'єкт дослідження – процеси роботи з графіком чергувань персоналу підприємства.

Предмет дослідження – алгоритми, методи та програмні засоби автоматизації роботи з графіком чергувань персоналу підприємства.

Короткий зміст роботи: У роботі проаналізовано особливості організації робочого часу та виявлено ключові проблеми ручного формування графіків чергувань персоналу, зокрема значні часові витрати, ризики конфліктних призначень, відсутність сповіщень працівників та складність рівномірного розподілу навантаження. На основі огляду недоліків існуючих рішень сформульовано функціональні та нефункціональні вимоги до спеціалізованого вебзастосунку. Спроєктовано та розроблено програмну систему автоматизації планування з використанням сучасного стеку технологій: мови Python серверної частини, Vue.js для клієнтського інтерфейсу, а також ORM SQLAlchemy та СУБД PostgreSQL для управління базою даних. Програмно імплементовано евристичний алгоритм автоматизованої генерації розкладу, механізми перепризначення чергувань, погодження відпусток, систему сповіщень та збереження історії адміністративних дій. Реалізовано підсистему експорту розкладу та інтеграції із зовнішніми корпоративними календарями на базі стандарту RFC 5545. Проведене тестування підтвердило коректність роботи алгоритмів обробки змін та контролю конфліктів. Практичне впровадження розробленої системи забезпечує усунення людського фактора, підвищення прозорості процесів управління та суттєве скорочення часу

формування графіка чергувань – з 2–3 годин ручної роботи до 1,5 секунди. Сферою використання застосунку є операційне управління персоналом, оптимізація розподілу робочого часу та автоматизація кадрових процесів на підприємствах.

КЛЮЧОВІ СЛОВА: АВТОМАТИЗАЦІЯ ГРАФІКІВ ЧЕРГУВАНЬ, КОМБІНАТОРНА ОПТИМІЗАЦІЯ, ЕВРИСТИЧНІ АЛГОРИТМИ, ПЛАНУВАННЯ РОБОЧОГО ЧАСУ, FASTAPI, VUE.JS, POSTGRESQL, РЕЛЯЦІЙНІ БАЗИ ДАНИХ.

ЗМІСТ

ВСТУП

У сучасному світі управління трудовими ресурсами перетворилося з простої адміністративної функції на одну з ключових складових ефективної діяльності підприємства. Зі зростанням популярності гнучких графіків роботи та посиленням уваги до балансу робочого часу персоналу, зростає потреба у цифрових інструментах, що дозволяють не лише складати графіки чергувань автоматично, але й гарантувати справедливий розподіл навантаження, дотримання нормативних інтервалів відпочинку та прозорий облік компенсаційних днів. На практиці більшість команд досі планує чергування вручну в електронних таблицях, що поглинає значний час менеджерів та породжує системні помилки: нерівномірне навантаження, втрату обліку відгулів, порушення мінімальних інтервалів між робочими змінами. З точки зору обчислювальної складності задача автоматизованого складання розкладів належить до класу NP-складних проблем, ефективне розв'язання якої вимагає застосування алгоритмічних методів.

Мета роботи – з’ясування ефективності застосування алгоритмів для автоматизованого балансування робочого навантаження та мінімізації людського фактору в кадровому менеджменті за допомогою розробленого програмного забезпечення.

Об’єкт дослідження – процес організації робочого часу, формування розкладів чергувань та обліку компенсаційного навантаження персоналу в умовах підприємства.

Предмет дослідження – алгоритми і методи автоматизованого розподілу чергувань та їх вплив на рівномірність навантаження співробітників з урахуванням обмежень доступності персоналу.

Для реалізації поставленої мети виділено такі задачі:

1. Провести огляд та аналіз існуючих програмних рішень для управління графіками чергувань персоналу, визначити їх переваги та недоліки;

2. Сформулювати вимоги до розроблюваного web-застосунку на основі аналізу предметної галузі та операційних потреб підприємства;

3. Провести огляд та аналіз технологій та інструментів, що можуть бути використані для розробки web-застосунку;

4. Спроектувати та розробити web-застосунок з використанням обраних технологій, включно з евристичним алгоритмом генерації розкладу;

5. Провести тестування розробленого web-застосунку для виявлення та усунення помилок.

Для розробки web-застосунку було обрано мову програмування Python для серверної частини, що забезпечує можливості для реалізації обчислювальних алгоритмів. Використання асинхронного фреймворку FastAPI дозволяє створити високопродуктивний та масштабований бекенд завдяки підтримці стандарту ASGI. Бібліотека SQLAlchemy спрощує роботу з базою даних через об'єктно-реляційне відображення та гарантує транзакційну цілісність операцій. JavaScript-фреймворк Vue.js забезпечує побудову інтерактивного та динамічного користувацького інтерфейсу з компонентним підходом та ефективним оновленням DOM через механізм віртуального дерева елементів. Для зберігання даних обрано реляційну СУБД PostgreSQL, яка забезпечує підтримку принципів ACID, що є критичним для коректного обліку компенсаційних відгулів та історії адміністративних дій. Інтеграція з корпоративними календарями реалізована через стандарт RFC 5545 та Microsoft Graph API. Очікуваним результатом роботи є web-застосунок, який надасть менеджерам зручний інструмент для автоматизованого складання графіків чергувань, обліку компенсаційних днів та делегованого обміну змінами між співробітниками, сприяючи зменшенню часових витрат на адміністрування та підвищенню справедливості розподілу навантаження в команді.

1 ОГЛЯД ТА АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ ТА СИСТЕМ ДЛЯ АВТОМАТИЗОВАНОГО ПЛАНУВАННЯ ГРАФІКІВ ЧЕРГУВАНЬ

1.1 Аналіз предметної області та проблематики управління трудовими ресурсами

Операційне управління персоналом є складною математичною задачею [2, 11]. Складання робочих графіків вимагає одночасного узгодження десятків динамічних змінних, де доступність персоналу часто конфліктує з операційними потребами підприємства. Менеджери щомісяця витрачають значний час на коригування розкладів, що є непродуктивним використанням ресурсів. З точки зору теорії складності, задача розподілу чергувань належить до класу NP-складних проблем комбінаторної оптимізації [4, 17], що робить її важко розв'язною для людини через неможливість утримання в пам'яті всього масиву обмежень.

На рівні малих команд у великих організаціях спостерігається нестача автоматизації. Процеси планування здебільшого базуються на статичних електронних таблицях, що створює високі ризики помилок [2]. Зміна навіть одного параметра, наприклад, несподіваний вихідний або запит на відпустку призводить до необхідності ручного перерахунку всього розкладу. За відсутності транзакційної цілісності даних [10], помилка в одній комірці або випадково видалений рядок створюють розбіжність між фактично відпрацьованим часом та офіційно зафіксованим балансом годин працівника.

Специфіка підприємства має операційні правила, де ключовим є забезпечення обов'язкового інтервалу між вихідними чергуваннями. Ручний контроль цього параметра часто призводить до збоїв. Додати сюди необхідність безперервного обліку компенсаційних днів, бо згідно з правилами відпрацьована суботня або недільна зміна автоматично дає відгул. Цей механізм вимагає суворого

транзакційного зв'язку між графіком та профілем користувача, який неможливо реалізувати без реляційної бази даних.

Децентралізація інформаційних потоків провокує операційні збої. Статичні файли позбавлені механізмів системного сповіщення. Наслідком цього стають пропущені робочі зміни. Відсутність автоматизованих повідомлень про чергування змушує покладатися на пам’ять працівника.

Нерівномірний розподіл навантаження, спричинений суб'єктивними рішеннями адміністратора, підриває довіру співробітників до справедливості процесу. Недотримання мінімальних інтервалів відпочинку через людську помилку призводить до втрати працівниками контролю над власним часом, що негативно впливає на продуктивність та збільшує плинність кадрів.

Традиційні методи управління розкладами не забезпечують необхідного рівня точності та об'єктивності. Для вирішення цієї проблеми доцільно застосувати автоматизовану систему алгоритмів [1, 4], здатну здійснювати балансування навантаження без втручання людського фактору. Це дозволить підвищити прозорість процесу планування та гарантувати дотримання встановлених нормативних вимог.

1.2 Огляд існуючого програмного забезпечення для планування

Автоматизація кадрового планування починається з архітектурного вибору. Ринок пропонує три фундаментально різні парадигми управління ресурсами, які має власні межі масштабування. Проведено порівняльний аналіз існуючих підходів. Оцінку зведено до п'яти метрик представлених в таблиці 1.1, які впливають на технічну стійкість та життєздатність системи в умовах реального операційного навантаження.

Таблиця 1.1

Порівняльний аналіз систем автоматизації графіків чергувань

Характеристика	**Ручне ведення**	**Хмарні SaaS-системи**	**Корпоративні ERP**	**Система Shift Scheduler**
Автоматична генерація розкладу	Відсутня	Базова	Присутня	Присутня
Врахування специфічних бізнес-правил	Повна, але неконтрольована	Низька, жорстко задані рамки	Середня, вимагає доопрацювання коду	Повна
Автоматичний облік балансу компенсаційних відгулів	Відсутній	Присутній у преміум-модулях	Присутній	Вбудований спеціалізований модуль
Делегований обмін чергованнями із валідацією	Відсутній	Присутній	Залежить від конфігурації	Присутній
Інтеграція із зовнішніми календарями	Відсутня	Обмежена	Потребує налаштування шлюзів	Вбудована функція

1.2.1 Універсальні табличні процесори

Електронні таблиці залишаються найпоширенішим стартовим майданчиком. Програми рівня MS Excel [19] пропонують візуальну гнучкість як показано на рис. 1.1. Менеджер формує структуру розкладу, додає стовпці та змінює кольори, умовне форматування створюють імітацію автоматизації. На цьому переваги закінчуються.

Табличні процесори не мають вбудованого обчислювального ядра для автоматизації складної бізнес-логіки. Валідація даних обмежується регулярними виразами та перевіркою типів у комірках. Заповнення здійснюється виключно

вручну, що унеможливлює автоматичний контроль дотримання нормативних вимог, зокрема мінімальних інтервалів відпочинку між чергуваннями.

Відсутність реляційної моделі даних призводить до проблем цілісності інформації. Дані зберігаються у денормалізованому вигляді, без системних зв'язків між таблицями. Це унеможливлює автоматичний облік компенсаційних відгулів, бо інформація може втрачатися або дублюватися. Кожне оновлення локального файлу створює ризик втрати цілісності, наприклад, відпрацьована вихідна зміна може не бути зафіксована у відповідній таблиці обліку через людську помилку, що призводить до розриву логічних зв'язків між даними. Одна помилка провокує каскадні наслідки, які важко виявити та виправити.

Спроби розширення функціональності табличних процесорів через макроси VBA або скрипти Google Apps Script зіштовхуються з фундаментальним обмеженням: ці скриптові середовища не мають доступу до архітектурного шару зберігання у вигляді нормалізованих таблиць з зовнішніми ключами.

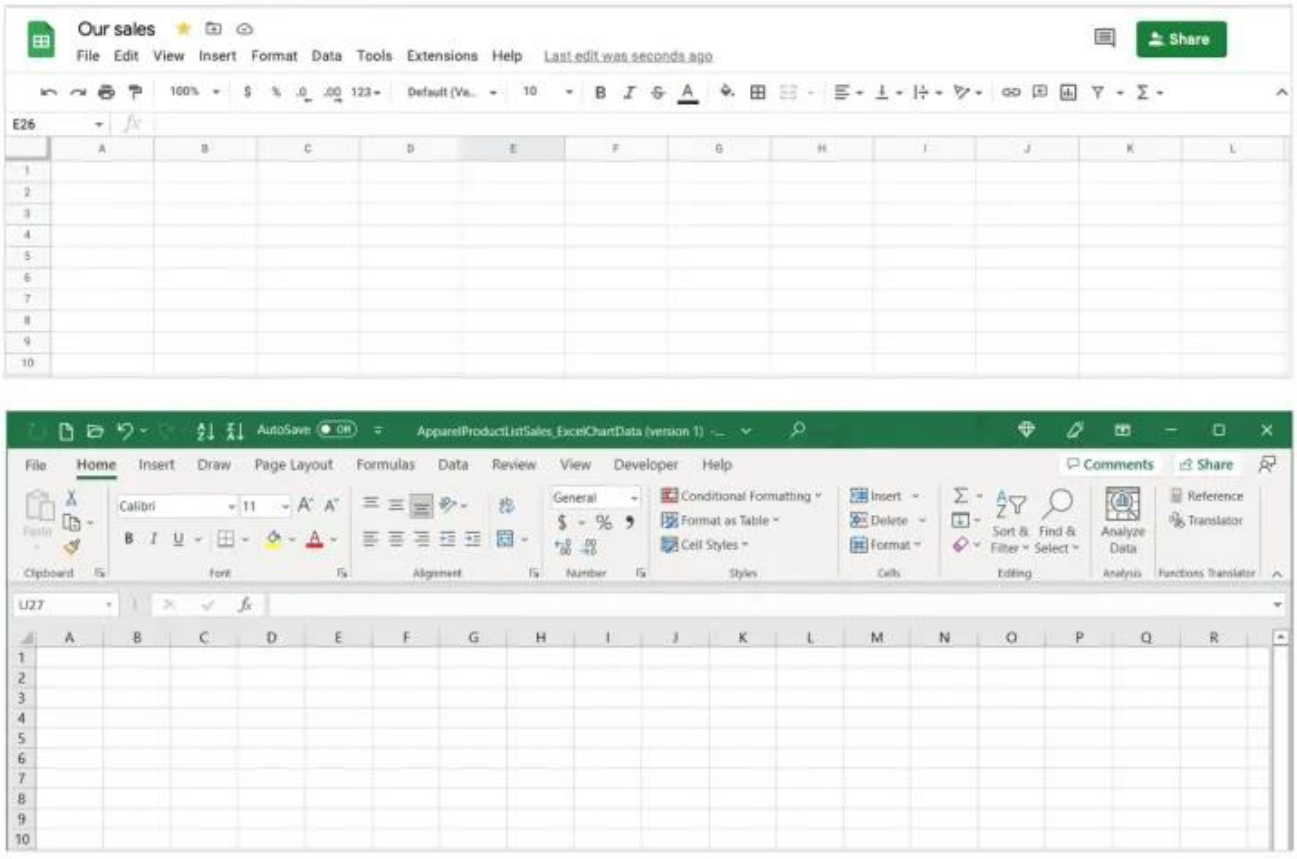


Рис. 1.1 Універсальні табличні процесори (MS Excel, Google Sheets)

Додатковим архітектурним бар'єром стає повна ізоляція таблиць від зовнішніх протоколів зв'язку. Синхронізація розкладу у реальному часі неможлива. Інтеграція з персональними календарями за стандартом RFC 5545 (iCalendar) [15] вимагає написання складних макросів на скриптових мовах. Персонал змушений постійно звертатися до статичного загального файлу.

1.2.2 Корпоративні системи управління ресурсами

На противагу таблицям існують комплексні корпоративні екосистеми. Системи управління ресурсами рівня SAP SuccessFactors [26] або спеціалізовані модулі обліку часу в BAS проєктуються під екстремальні навантаження. Такі рішення проєктуються для великого бізнесу і здатні закрити практично будь-яку потребу. Їхня технічна перевага полягає у високому рівні транзакційної надійності. База даних гарантує суворе дотримання принципів ACID при кожній операції [10]. В ERP-системі облік балансу відгулів відбувається повністю автоматично і часто одразу прив'язаний до бухгалтерських нарахувань. Також ці системи підтримують інтеграцію зі сторонніми календарями, що дозволяє синхронізувати графіки на рівні всієї компанії.

Водночас монолітна архітектура суттєво обмежує гнучкість локальних налаштувань. Користувацькі інтерфейси таких систем перевантажені бізнес-логікою, не пов'язаною безпосередньо з плануванням робочого часу. Впровадження одного специфічного правила вимагає рефакторингу базової конфігурації платформи, що під силу виключно профільним інженерам підтримки постачальника. Налаштування рольової моделі доступу є окремим проєктом впровадження з тривалим циклом узгодження. Зовнішня інтеграція реалізується через SOAP-протоколи або обтяжливі REST API, що ускладнює розробку клієнтських застосунків. Розгортання повноцінної ERP-платформи виключно для управління чергуваннями невеликої команди є економічно недоцільним та призводить до надмірного споживання апаратних і часових ресурсів на підтримку

інфраструктури. Вигляд інтерфейсу однієї з найпоширеніших корпоративних ERP-систем наведено на рис. 1.2.

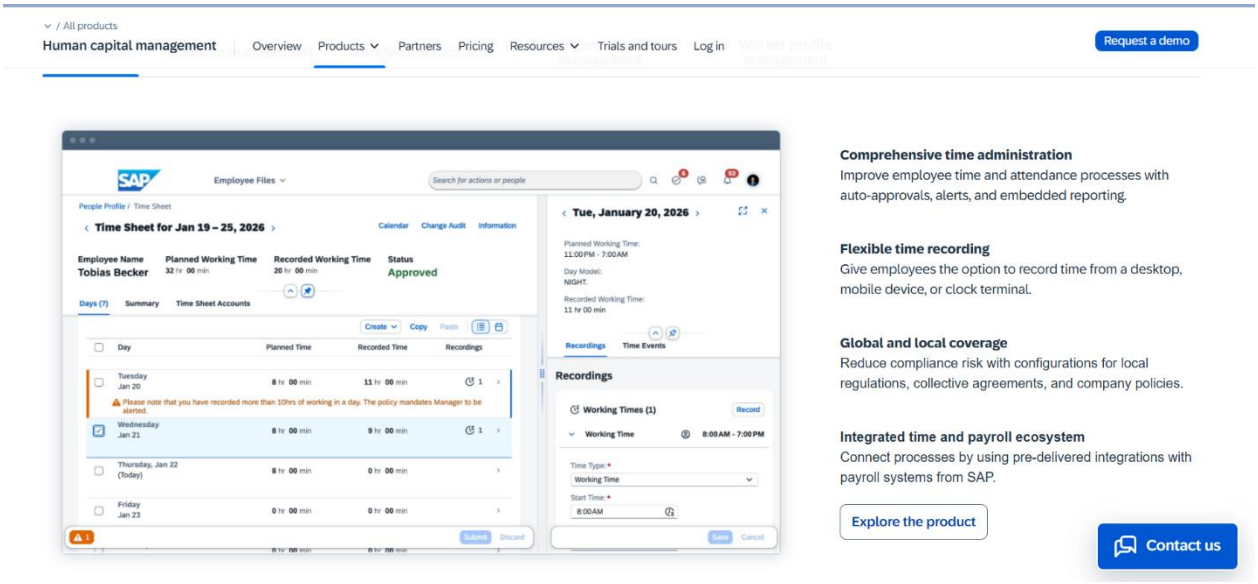


Рис. 1.2 Інтерфейс модуля обліку робочого часу в корпоративній системі SAP SuccessFactors

1.2.3 Хмарні SaaS-системи

Окремим і досить вагомим сегментом ринку є спеціалізовані хмарні сервіси, що працюють за моделлю SaaS (Software-as-a-Service). Головна перевага SaaS-рішень - це високий рівень автоматизації стандартних процесів. Платформи типу Deputy [9] роблять головну ставку на реактивний користувацький досвід. Мобільні додатки працюють швидко. Базові алгоритми автоматизації успішно покривають лінійні сценарії стандартного ритейлу або сфери обслуговування.

Однак ці алгоритми є закритими системами. Серверна частина архітектурно ізольована, інженер не має можливості втрутитися в логіку обчислювального ядра. Якщо підприємство вимагає нестандартної системи нарахування компенсаційних днів саме за специфічні типи змін, SaaS-рішення тут не відповідає вимогам. Налаштування жорстко обмежені параметрами панелі керування. Відбувається технологічна прив'язка до архітектури постачальника.

SaaS-продукти часто декларують широкі можливості зовнішньої інтеграції. Але публічні API-шлюзи мають суворі ліміти на кількість запитів. Побудова безшовної двосторонньої синхронізації між хмарним сервісом та внутрішньою інфраструктурою підприємства вимагає підняття додаткових проміжних серверів. Це повністю ламає початкову концепцію швидкого розгортання та додає нові точки відмови у загальну мережеву топологію. Інтерфейс модуля планування в хмарній SaaS-платформі Deputy проілюстровано на рис. 1.3.

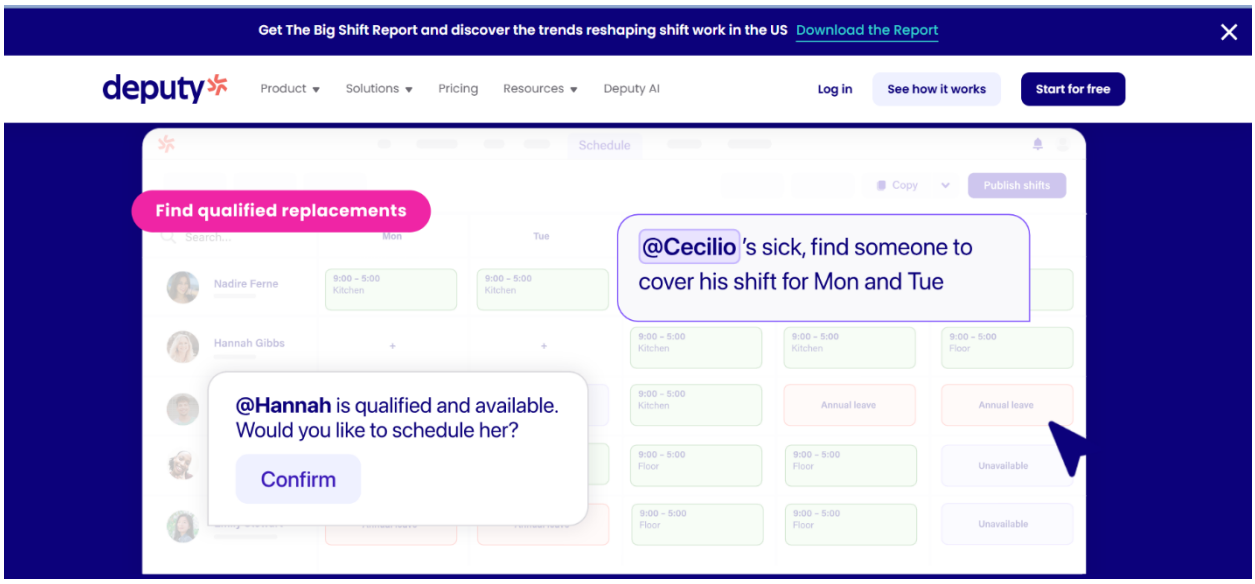


Рис. 1.3 Інтерфейс модуля планування розкладу в хмарній системі Deputy

1.2.4 Обґрунтування розробки кастомної системи «Shift Scheduler»

Аналіз існуючих інструментів виявляє системну проблему ринкових рішень. Програмні продукти пропонують полярні підходи: табличні процесори надають повну гнучкість інтерфейсу, але позбавлені алгоритмічної основи; корпоративні ERP-системи забезпечують надійність даних, проте є негнучкими та складними у налаштуванні; хмарні SaaS-платформи мають закриту архітектуру, що обмежує можливості адаптації. Кожне з цих рішень вимагає суттєвих компромісів при впровадженні в умовах специфічних операційних вимог підприємства.

Розробка кастомного програмного забезпечення «Shift Scheduler» вирішує цей дисбаланс. Власна архітектура гарантує повний контроль над обчислювальним

рушієм. Специфічні бізнес-правила реалізуються безпосередньо у модулях. API-first підхід дозволяє відокремити логіку генерації від клієнтського інтерфейсу. Реляційна модель бази даних нативно робить можливим транзакційний облік компенсаційних днів через механізми зовнішніх ключів та обмежень на рівні схеми. Інтеграція алгоритмів експорту iCalendar-файлів відбувається безпосередньо на рівні бекенду, без залучення сторонніх мікросервісів чи міделвару.

1.3 Формалізація бізнес-правил та нормативних обмежень

Будь-який алгоритм є лише математичним віддзеркаленням реальних обмежень предметної області. Програмний код не розуміє абстрактних понять на кшталт «справедливого графіка» чи «рівномірного навантаження». Він оперує виключно булевими значеннями та числовими коефіцієнтами штрафів. Для того, щоб перекласти операційні вимоги підприємства на мову Python, весь масив корпоративних правил було декомпозовано на два незалежні класи.

Перший клас становлять жорсткі обмеження [1, 11], порушення яких призводить до миттєвого відхилення згенерованого варіанту розкладу. Алгоритм відкидає цю гілку обчислень як некоректну. Базовим жорстким правилом є недоступність співробітника під час документально підтвердженої відпустки, лікарняного або офіційного відрядження. Реалізація цього вимагає безперервного транзакційного зв'язку між матрицею поточного планування та таблицею статусів користувачів у базі даних.

Система також повинна унеможливлювати перехресні призначення. Співробітник не може бути одночасно запланований на дві різні зміни, які перетинаються у часі. Щоб уникнути колізій, логіка перевірки базується на порівнянні часових штампів початку та кінця чергування.

При спробі призначити спеціаліста на зміну в суботу чи неділю, обчислювальне ядро має просканувати історичний лог бази даних у зворотному напрямку. Якщо у вікні мінус вісімнадцять діб знайдено інше вихідне чергування

цього ж користувача, функція валідації повертає False. Алгоритм змушений шукати іншого кандидата.

Другий клас формують м'які обмеження. Вони відповідають за оптимізацію та балансування загального навантаження команди [3, 11]. Замість бінарної логіки програма використовує евристичні штрафні коефіцієнти [1]. Головна мета обчислювального циклу є мінімізація сумарного вектора штрафів для всієї сітки розкладу. Якщо два працівники претендують на один часовий слот, пріоритет надається тому, хто відпрацював меншу кількість годин у поточному періоді, що вимагає динамічної агрегації даних з урахуванням специфіки кожної зміни.

Облік компенсаційних відгулів має бути реалізований як окремий транзакційний модуль. Відпрацьована вихідна зміна автоматично генерує компенсаційний день, збільшуючи баланс у реляційній таблиці. При ініціюванні запиту на відгул система виконує послідовність перевірок: наявність балансу, відсутність перевищення квот відсутності персоналу на обрану дату. Бізнес-логіка дозволяє авансове використання відгулів. Баланс може набувати від'ємного значення з подальшим погашенням при наступних вихідних чергуваннях. Лише після успішного проходження всіх перевірок транзакція фіксується в базі даних. У разі виявлення логічної помилки на будь-якому етапі система виконує відкат транзакції.

1.4 Постановка задачі на розробку кастомної системи

Головною метою розробки є створення автономної системи управління трудовими ресурсами з власним обчислювальним ядром. Задача полягає в автоматизації розподілу робочих чергувань на основі комбінаторних евристик. Математично проблему сформульовано як пошук оптимального розв'язку з мінімізацією штрафних коефіцієнтів [3, 6] при дотриманні всіх обмежень предметної області.

Архітектура будується за стандартом API-first. Присутнє розділення відповідальності між компонентами, тобто клієнтський інтерфейс відповідає

виключно за реактивну візуалізацію масивів даних. Будь-яка бізнес-логіка відсутня на фронтенді. Серверна частина інкапсулює алгоритми маршрутизації та генерації. Таке рішення унеможливлює вплив на алгоритми балансування навантаження через маніпуляції з локальним сховищем браузера або компрометацію клієнтської сесії.

Облік робочого часу формує транзакційний центр системи. Програмне забезпечення має підтримувати концепцію від'ємного балансу. Працівник може авансом використати відгул, що фіксується в реляційній моделі як борг без втручання адміністратора. Сценарії взаємодії користувачів з системою у вигляді діаграми прецедентів зображено на рис. 1.4.

Виходячи з формалізованої моделі, функціональні вимоги до програмного продукту зводяться до наступних алгоритмічних завдань:

- імплементація двопрохідного евристичного алгоритму генерації розкладу з окремими ітераціями для вихідних та будніх днів;
- автоматичне блокування призначень при виявленні порушень мінімального 18-денного інтервалу між чергуваннями;
- реалізація рольової моделі управління доступом з розмежуванням привілеїв адміністратора та співробітника;
- обробка та верифікація заявок на відпустки, відгули та обмін чергуваннями;
- автоматичний розрахунок та моніторинг балансу компенсаційних вихідних днів працівників;
- динамічна генерація файлів розкладу для синхронізації зі сторонніми клієнтами;
- фіксація та зберігання історії адміністративних дій у журналі аудиту.

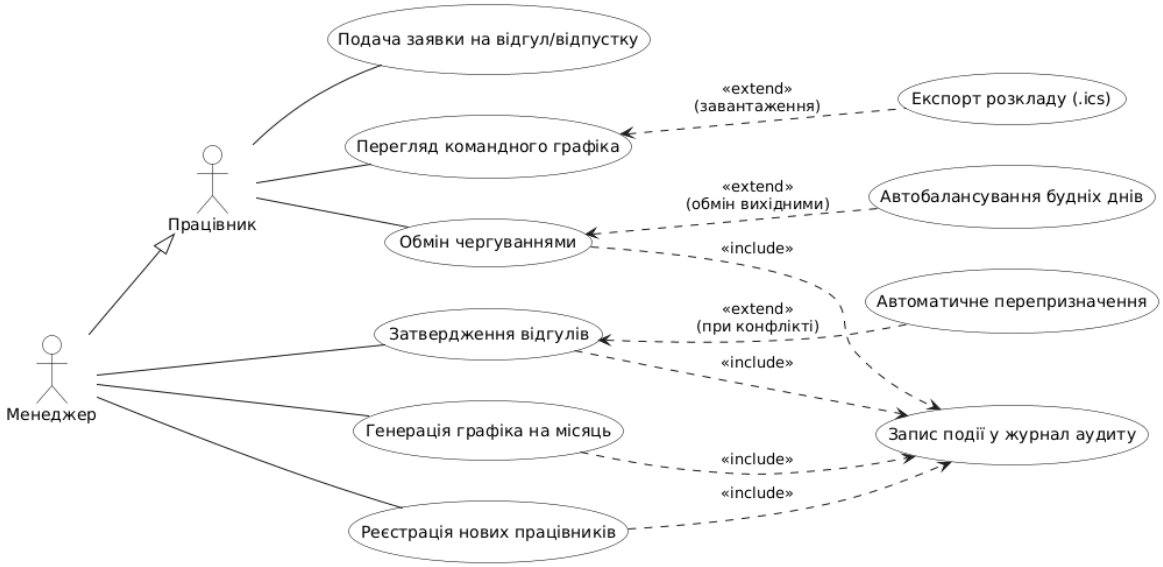


Рис. 1.4 Діаграма прецедентів

Проблема експоненціального зростання обчислювальної складності вирішується через систему вагових коефіцієнтів. Конфігурація пріоритетів маршрутизації виноситься у змінні середовища бази даних. Зміна операційних потреб підприємства вимагатиме лише оновлення констант, а не рефакторингу базових класів контролера.

Нефункціональні вимоги встановлюють інженерні межі. Продуктивність та послідовність даних виступають критичними метриками життєздатності розробки. Система повинна відповідати наступним технічним критеріям:

– час обробки стандартних REST API запитів не повинен перевищувати 200 мілісекунд;

– сесія користувача захищена JWT-токеном із часом життя 480 хвилин [16];

– паролі хешуються за допомогою криптографічного алгоритму Bcrypt;

– база даних зобов'язана гарантувати дотримання принципів ACID;

– алгоритм інтеграції календарів має відповідати специфікації RFC 5545 [15] без залучення зовнішніх проміжних мікросервісів.

Формалізована постановка задачі визначає вектор розробки. Наступним етапом є проєктування реляційної схеми даних та вибір конкретного технологічного стеку для реалізації затвердженої архітектури.

2 ПРОЄКТУВАННЯ ПРОГРАМНОЇ СИСТЕМИ АВТОМАТИЗОВАНОГО РОЗПОДІЛУ ГРАФІКІВ РОБОЧИХ ЗМІН

2.1 Вибір та обґрунтування технологічного стеку

Серверну частину побудовано на асинхронному фреймворку FastAPI [12, 18]. Цей вибір продиктований характером обчислень: генерація розкладу з перевіркою нормативних обмежень та обчисленням штрафних коефіцієнтів. Ця задача процесорно-важка і саме асинхронна модель ASGI дозволяє паралельно обробляти інші запити, поки алгоритм рахує. Pydantic [23] інтегрований з FastAPI і валідує вхідні JSON-тіла за анотаціями типів Python [25]. Якщо структура запиту не відповідає схемі, клієнт отримує помилку 422 з описом без звернення до бізнес-логіки. Додатковою перевагою є автоматична інтерактивна документація OpenAPI на ендпоінті /docs, яка стала основним інструментом для ручного тестування API під час розробки.

Для забезпечення реляційної цілісності даних обрано СУБД PostgreSQL [22]. Взаємодія серверного ядра з базою реалізована через ORM-бібліотеку SQLAlchemy [27]. Цей шар абстракції повністю ізолює бізнес-логіку програми від безпосередніх SQL-запитів. У файлі models.py розгорнуто декларативну схему сутностей: users, shifts, leaves. Кожен зв'язок описаний через зовнішні ключі з параметрами каскадного видалення. Це гарантує структурну стабільність на рівні бази. Облік компенсаційних відгулів вимагає жорстких ACID-транзакцій. У випадках коли працівник бере вихідний авансом і його баланс набуває від'ємного значення, SQLAlchemy дозволяє обертати такі багатоетапні операції у блоки try...except із примусовим викликом db.rollback() при будь-якій логічній колізії, унеможливлюючи часткові записи даних.

Підсистему безпеки та рольового доступу реалізовано в модулі auth.py з використанням стандарту OAuth2 [14] та JSON Web Tokens [16]. Під час виклику ендпоінту /api/auth/login клієнт передає облікові дані. Сервер перевіряє

криптографічний хеш пароля за допомогою бібліотеки passlib із алгоритмом Bcrypt. Успішна валідація ініціює генерацію токена. Він підписується серверним секретним ключем і містить зашифрований ідентифікатор користувача та його рольовий статус. Подальша маршрутизація захищається через систему залежностей фреймворку. Ін'єкції Depends(get_current_user) перехоплюють HTTP-заголовок Authorization, декодують токен і блокують виконання коду при недійсному підписі, віддаючи статус 401 Unauthorized.

Обчислювальне ядро винесено в окремий сервісний модуль scheduler.py, тоді як контролер у файлі schedules.py відповідає лише за прийом POST-запиту /generate та виклик відповідної функції. Усередині сервісу працює жадібний евристичний алгоритм. Він робить кілька запитів до PostgreSQL і підтягує перелік активних співробітників, затверджені відпустки та історичний лог чергувань. Усі звернення до бази проходять через await, тож головний потік сервера не блокується. Поки алгоритм обчислює, FastAPI спокійно віддає GET-запити іншим користувачам, які хочуть подивитися свій графік. Результат повертається у вигляді JSON-масиву з призначеннями.

Фронтенд побудовано на Vue.js третьої версії [28]. Ключова вимога до інтерфейсу це швидке оновлення без візуальних затримок, адже календарна сітка на місяць містить сотні DOM-елементів. З цим завданням Vue справляється завдяки віртуальному DOM: фреймворк рахує, які саме вузли реально змінилися, і оновлює тільки їх. За мережеві запити відповідає Axios [7]. JWT-токен підставляється у заголовок Authorization автоматично через інтерцептор, тож у коді бізнес-логіки не міститься авторизація. Коли менеджер змінює статус зміни, відправляється PATCH-запит, сервер оновлює запис у базі, а на клієнті оновлюється лише та конкретна комірка таблиці. Сторінка повністю не перезавантажується - це звична поведінка SPA-додатків.

Розгортання організовано через Docker [8]. Серверна частина, реляційна СУБД та клієнтський додаток ізольовані в незалежних віртуальних контейнерах. Управління внутрішньою мережею та взаємодією компонентів реалізовано через

docker-compose. Такий підхід гарантує повну ідентичність локального середовища розробки та робочого сервера. Залежності зафіксовані на рівні конфігураційних файлів. Ризики конфліктів системних бібліотек під час міграції коду зведені до мінімуму.

Пряма інтеграція через Microsoft Graph API [20] реалізована на рівні серверної логіки, проте її використання в продакшн-середовищі потребує додаткового налаштування корпоративної інфраструктури. Зокрема, необхідна реєстрація додатка в Microsoft Entra ID з наданням делегованих прав на рівні адміністратора організації. Для тестування та демонстрації цього функціоналу потрібні корпоративні облікові дані з відповідними правами доступу, що виходить за межі можливостей локального середовища розробки. Крім того, процес узгодження таких прав у реальному підприємстві може тривати тижнями, що ускладнює швидке впровадження системи.

З огляду на ці обмеження, основним механізмом інтеграції обрано автономну генерацію файлів календаря. Серверна частина нативно формує документи за відкритим стандартом RFC 5545 без залучення сторонніх бібліотек. Бекенд виконує запит до бази даних, отримує масив затверджених чергувань і конкатенує текстові інструкції iCalendar у форматі BEGIN:VEVENT … END:VEVENT, серіалізуючи часові мітки безпосередньо у відповідь HTTP. Результат повертається через клас PlainTextResponse з MIME-типом text/calendar, що дозволяє користувачу зберегти файл формату .ics для подальшого імпорту в MS Outlook, Google Calendar або інший сумісний календарний клієнт без необхідності налаштування OAuth-авторизації чи отримання корпоративних дозволів. Такий підхід забезпечує незалежність від конкретного постачальника хмарних послуг та гарантує технічну сумісність через загальноприйнятий відкритий стандарт.

2.2 Архітектура системи та клієнт-серверна взаємодія

Архітектура програмного комплексу базується на парадигмі API-first [13]. Відбувається розділення відповідальності між компонентами. Серверна частина

ізольована від клієнтського інтерфейсу і функціонує як незалежний обчислювальний вузол. Клієнтський додаток не містить бізнес-логіки, він виключно формує HTTP-запити та реактивно візуалізує отримані JSON-відповіді. Це дає змогу для масштабованості. Модифікація внутрішніх модулів бекенду не вимагає змін кодової бази фронтенду.

Топологія мережевої взаємодії побудована за моделлю відокремленого клієнта та сервера. Фронтенд та бекенд функціонують на різних мережевих портах. Це спричиняє автоматичне блокування прямих запитів браузером. Проблема вирішується налаштуванням політики CORS (Cross-Origin Resource Sharing). Дозволи конфігуруються під час ініціалізації додатка у файлі main.py. Запити маршрутизуються через ASGI-сервер. Мережеві байти конвертуються у Python-об'єкти. Далі фреймворк FastAPI виконує їхню диспетчеризацію та передає в асинхронний цикл подій.

Внутрішня структура бекенду розшарована на абстракції. Пряме звернення до бази даних із контролера заборонено. Обчислювальна логіка розділена на функціональні директорії. Рівень маршрутизації зосереджений у пакеті api, тут оголошені ендпоінти для управління ресурсами: auth.py, leaves.py, schedules.py, shifts.py. Ці модулі приймають запити та формують HTTP-відповіді і не виконують математичних розрахунків. Складні алгоритми винесені на рівень сервісів. Директорія services інкапсулює алгоритмічну складність системи. У файлі scheduler.py реалізовано алгоритм генерації та оптимізації розкладу. Модуль outlook.py форматує згенеровану матрицю даних за стандартом RFC 5545. Загальну архітектуру системи з розділенням на клієнтську частину, серверні модулі та сховище даних наведено на рис. 2.1.

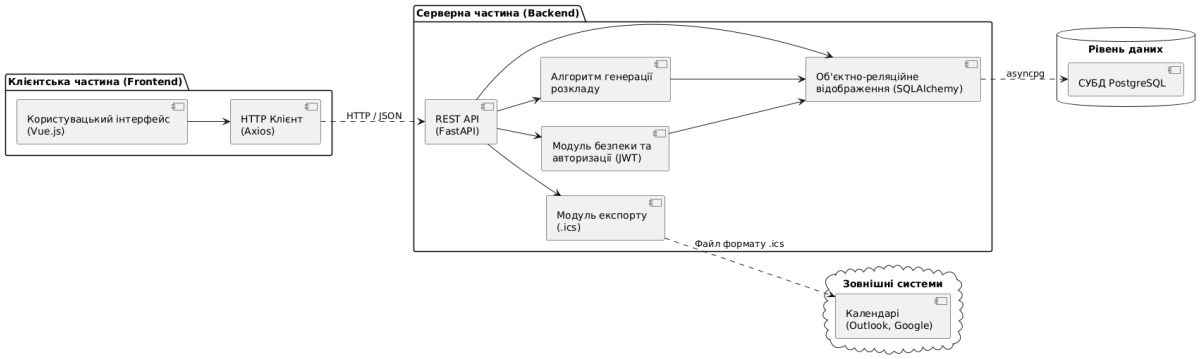


Рис. 2.1 Загальна архітектура системи

Цілісність даних забезпечується на рівні схем серіалізації через бібліотеку Pydantic [23]. Структура вхідних об'єктів типізована у файлі schemas.py. Сервер не обробляє неперевірений JSON, бібліотека Pydantic автоматично валідує типи та перевіряє наявність обов'язкових полів, конвертуючи вхідні дані у екземпляри класів Python. Якщо виявлено помилку типізації або пропущено поле, обробка переривається і сервер одразу повертає статус 422 Unprocessable Entity, що захищає бізнес-логіку від некоректних вхідних параметрів.

Систему безпеки побудовано за принципом stateless-архітектури без збереження стану сесій на сервері, що знижує навантаження на оперативну пам'ять. Ідентифікація користувача реалізована у модулі security.py. Авторизація відбувається через механізм JSON Web Token (JWT). Клієнт передає облікові дані, потім бекенд перевіряє криптографічний хеш пароля. Успішна валідація ініціює генерацію токена з цифровим підписом. На рівні фреймворку діє механізм впровадження залежностей. Файл deps.py містить перехоплювачі маршрутів. Перед виконанням захищеного ендпоінту залежність декодує токен та верифікує підпис. Якщо підпис недійсний, система блокує виконання та повертає код 401 Unauthorized.

Рівень доступу до даних ізольовано через ORM-інструментарій SQLAlchemy. Файл models.py містить декларативну схему реляційної бази PostgreSQL. Сутності користувачів, змін та відгулів описані як об'єктно-орієнтовані класи із зв'язками між таблицями, які встановлені через зовнішні ключі з правилами каскадного видалення. Підключення до бази керується конфігурацією з database.py. Пул

з'єднань оптимізовано для паралельних запитів. Транзакційна цілісність зберігається під час конкурентних записів. Ініціалізація бази та первинне наповнення винесені у скрипти init_db.py та seed.py. Повний цикл обробки HTTP-запиту від клієнта до бази даних та формування відповіді зображено на рис. 2.2.

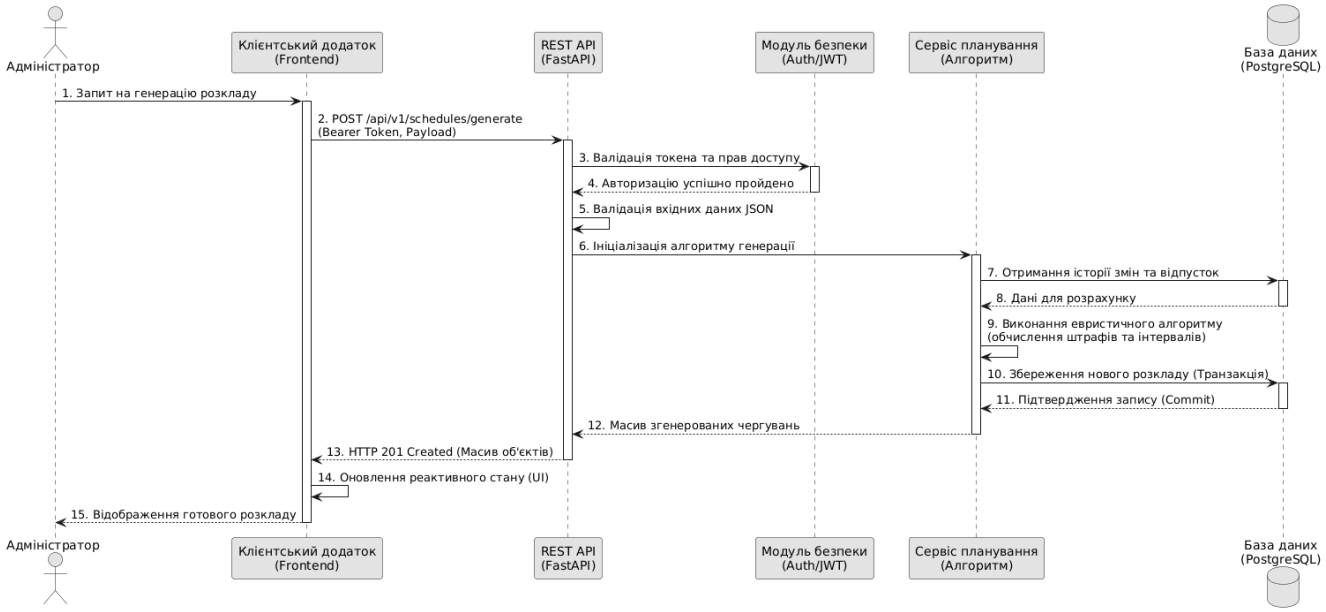


Рис. 2.2 Діаграма послідовності обробки HTTP-запиту

Конфігурація середовища рознесена за рівнем чутливості інформації з забороною явного задання паролів у вихідному коді. Модуль config.py відповідає за парсинг параметрів середовища. Він зчитує системні змінні з файлу .env, включаючи параметри підключення до бази та секретні ключі криптографії. Під час запуску додатка конфігураційний клас валідує наявність усіх критичних змінних. Відсутність параметра викликає помилку і запуск неконфігурованої системи автоматично блокується.

Маршрутизація процесів розподілена між спеціалізованими модулями. Ендпоінти shifts.py керують одиницями робочого часу, реалізуючи операції створення, читання, оновлення та видалення ресурсів. Запити на відгули обробляє модуль leaves.py, тут реалізовано транзакційну логіку валідації компенсаційних днів. Сервер перевіряє баланс співробітника. При недостатній кількості годин для

списання, база даних ініціює відкат операції. Структурна узгодженість цих сутностей виключає логічне накладання розкладів.

Клієнтську частину реалізовано за парадигмою Single Page Application: браузер завантажує базовий HTML один раз, подальший рендеринг виконує JavaScript-фреймворк. Зміна статусу заявки ініціює асинхронний PATCH-запит до API, далі бекенд оновлює запис у базі та повертає JSON. Стейт-менеджер локально оновлює відповідний масив даних. Інтерфейс оновлює лише змінений елемент DOM-дерева. Глобальне перезавантаження сторінки відсутнє.

Мережеві операції виконуються асинхронно. Технологічний стек ефективно обслуговує конкурентні з'єднання. Функції контролерів оголошені з використанням async def. Під час виконання тривалих I/O-запитів до PostgreSQL потоковий виконавець не блокується, оператор await повертає управління до Event Loop. Процесорний час використовується раціонально і сервер паралельно обробляє десятки інших HTTP-запитів.

Обробка виняткових ситуацій стандартизована на рівні бекенду. Системний стек викликів не передається клієнту. Налаштовано глобальні обробники помилок, тому будь-який виняток бізнес-логіки генерує кастомну подію. Фреймворк перехоплює її та конвертує у структуровану JSON-відповідь. Спроба видалення ресурсу без відповідних прав доступу перериває потік і клієнт отримує статус 403 Forbidden із полем опису помилки. Послідовність API-контракту зберігається за будь-яких умов експлуатації.

2.3 Проєктування бази даних та структури зберігання інформації

Фундаментом інформаційної архітектури слугує реляційна модель даних [10]. Вибір PostgreSQL обумовлений операційними вимогами до консистентності даних у контексті кадрового розкладу [22]. Використання NoSQL-рішень є архітектурно недоцільним через відсутність нативних механізмів гарантування цілісності зв'язків. Транзакційна надійність та повна підтримка вимог ACID

формують критичний базис. Кожна операція зміни статусу чергування виконується атомарно.

Взаємодія серверного ядра з фізичним сховищем абстрагована, прямі SQL-запити в коді контролерів повністю виключені. Рівень доступу до даних реалізовано через об'єктно-реляційне відображення SQLAlchemy. Конфігурація структури визначена у файлі models.py. Кожна таблиця бази даних проєктується як ізольований Python-клас. Цей шар автоматично нейтралізує загрозу SQL-ін'єкцій. Фреймворк самостійно забезпечує безпечну параметризацію запитів.

Центральним вузлом схеми є таблиця users. Вона зберігає облікові дані співробітників та параметри авторизації. Від цього ядра розгалужуються зовнішні ключі до всіх суміжних сутностей. Базові шаблони робочого часу інкапсульовано в таблиці shifts. Вона фіксує типи змін, їхню тривалість та час початку.

Матриця чергувань реалізована через нормалізацію. Таблиця schedule_periods задає часові рамки планування. Самі ж призначення зберігаються у транзакційній таблиці schedules. Кожен запис у ній пов'язує конкретного працівника user_id, обраний шаблон зміни shift_id та період period_id. Правила посилальної цілісності налаштовані, тому видалення шаблону зміни неможливе, якщо він вже залучений в активному розкладі. Схема реляційної моделі з усіма сутностями та зв'язками між ними подано на рис. 2.3

Реалізація логіки відгулів вимагає нетипових структурних рішень. База даних розмежовує сам факт заявки та облік накопичених днів. Таблиця leaves керує життєвим циклом запитів на відпочинок. Поле статусу використовує кастомний тип leavestatus, який фізично блокує запис невалідних станів на рівні рушія бази даних. Окремим прапорцем є булеве поле save_day_off. Воно сигналізує бекенду про необхідність залучення компенсаційної логіки.

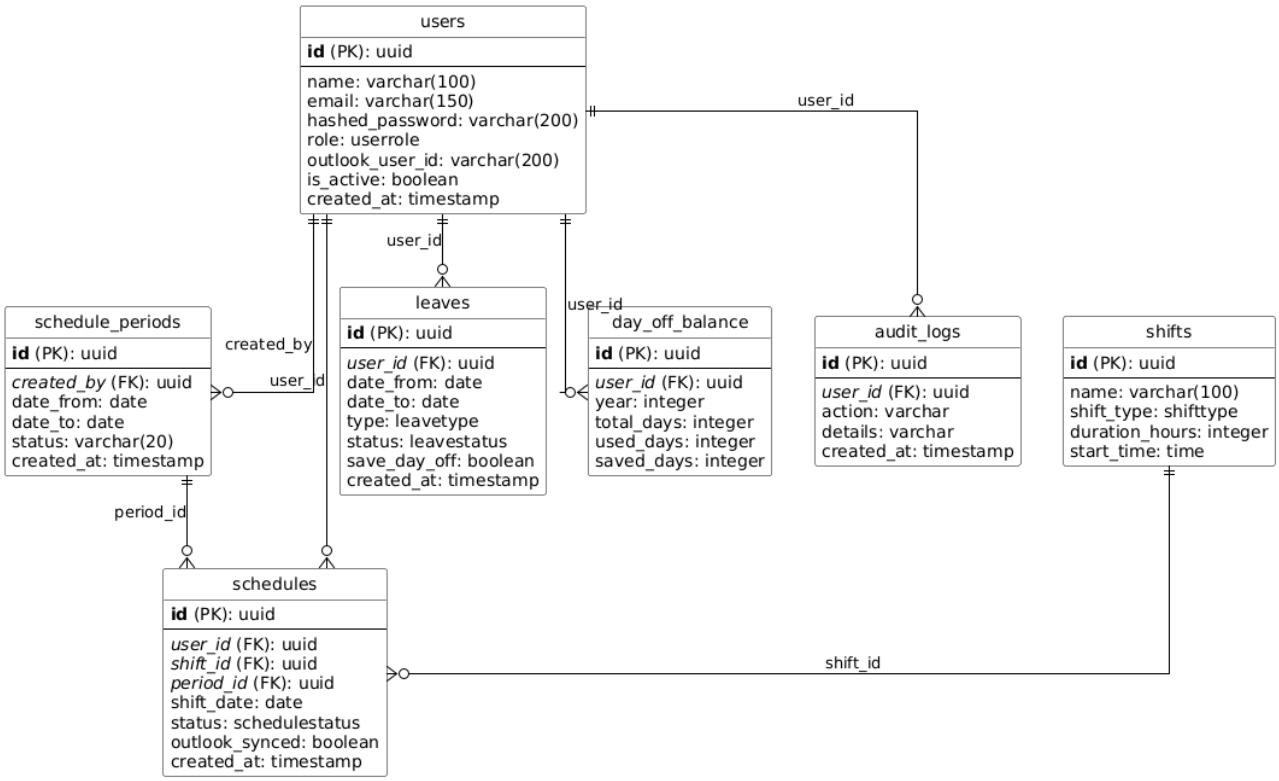


Рис. 2.3 ER-діаграма реляційної схеми бази даних

Для обліку компенсаційних днів створено окрему таблицю day_off_balance. Математика кредитного часу зберігається саме тут з сегментацією за роками. Реляційна модель фіксує загальну кількість днів total_days, використані used_days та накопичені saved_days. Бізнес-логіка дозволяє системі оперувати від'ємними значеннями. Працівник ініціює авансовий відгул, потім рахунок йде в мінус. Поле функціонує як динамічний кредитний ліміт. Наступна відпрацьована вихідна зміна запускає процес автоматичного оновлення і вирівнює баланс.

Система вимагає аудиту адміністративних дій. Для цього створено таблицю audit_logs. Вона записує кожну значущу подію в системі, фіксуючи час created_at, ініціатора user_id, тип операції action та деталі details. Така структура гарантує повну прозорість усіх змін розкладу.

Також обробка хронології потребує ізоляції від локальних часових налаштувань. Усі мітки зберігаються виключно у форматі UTC. Перетворення абсолютного часу на локальний київський формат відбувається під час рендерингу

в браузері. Цей архітектурний підхід повністю усуває логічні колізії при сезонному переході на літній чи зимній час.

Транзакційна ізоляція слугує механізмом запобігання від Race Condition. Якщо два адміністратори намагаються затвердити відгул на один часовий слот, СУБД бере управління паралелізмом на себе. Застосовуються механізми блокування на рівні рядків, SQLAlchemy огортає критичні операції оновлення балансів у транзакційні блоки. Конфлікт одночасного доступу викликає виняток бази даних, тому бекенд миттєво ініціює відкат через db.rollback(). Стан таблиць повертається до початкового консистентного вигляду.

2.4 Математичне та алгоритмічне забезпечення генерації розкладу

Формування матриці чергувань є задачею дискретної оптимізації [4, 17]. Застосування повного перебору варіантів недоцільне через велику кількість можливих комбінацій для групи працівників на місячному горизонті планування. Тому ядро системи побудовано на основі жадібної евристики з динамічним розрахунком штрафних коефіцієнтів [1, 4]. Цей алгоритм зосереджений у модулі scheduler.py. Він працює у два ізольовані етапи. Спочатку розподіляються вихідні дні, потім - будні. Це архітектурно розв'язує проблему конфлікту пріоритетів.

Жорсткі обмеження є основою і алгоритм має їх обов’язково дотримуватись [3, 11]. Перш ніж почати розподіл, система сканує таблицю leaves і всі затверджені відпустки і відгули перетворюються на множину заблокованих дат. Працівник фізично видаляється з пулу доступних кандидатів на конкретний день. Наступне правило - заборона дублювання. Механізм гарантує, що співробітник не отримає дві зміни протягом однієї календарної доби.

Розподіл вихідних днів є важливим етапом [5]. Тут працює математика справедливого навантаження. Для кожного доступного кандидата в ітераційному циклі обчислюється кортеж вагових коефіцієнтів. Сортування списку кандидатів відбувається за цільовою функцією: (weekend_count, too_soon_penalty, same_day_penalty, -days_since). Менше значення означає вищий пріоритет на

отримання зміни. Послідовність ітеративних кроків алгоритму при розподілі вихідних чергувань показано на рис. 2.4.

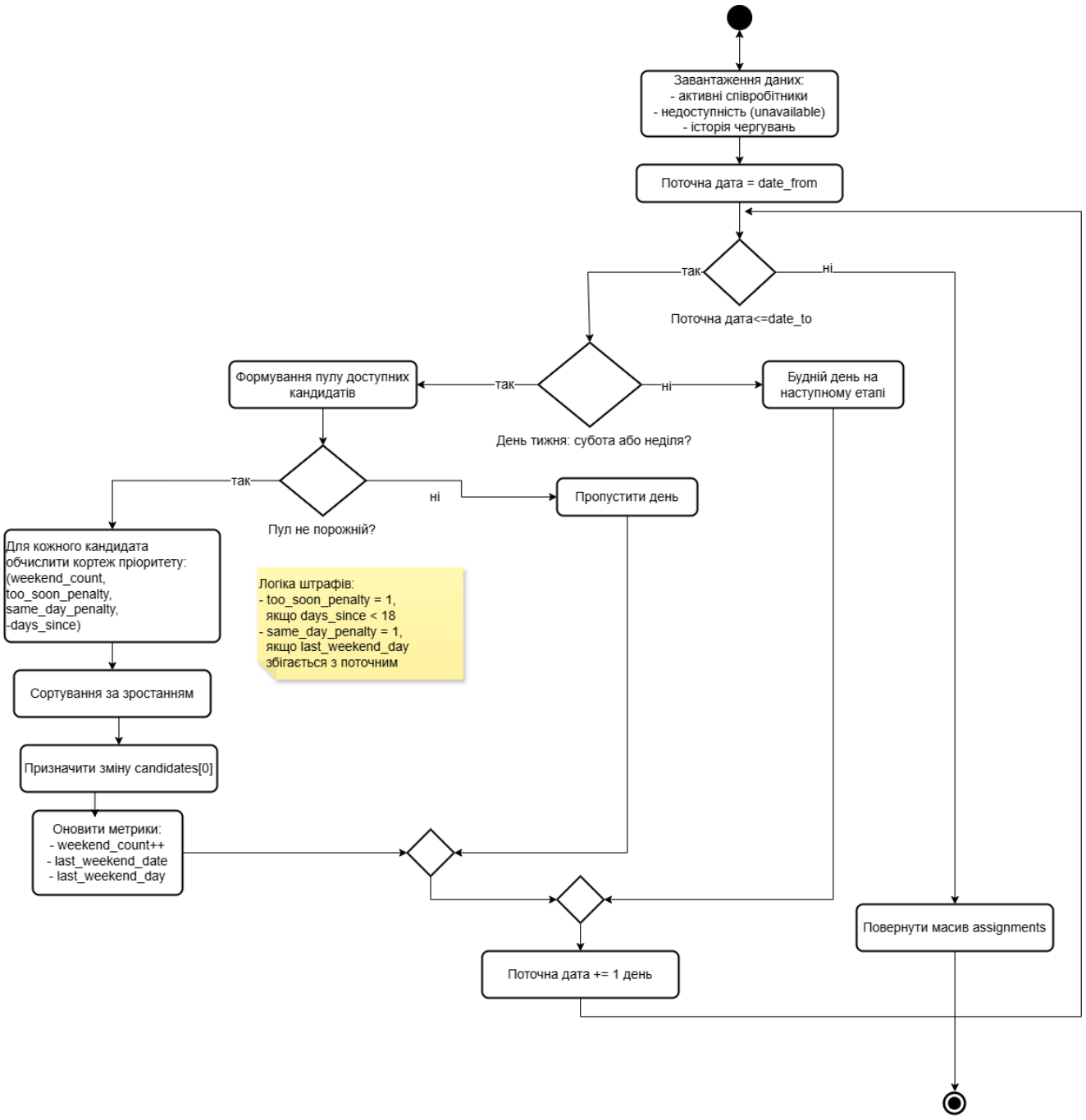


Рис. 2.4 Блок-схема алгоритму генерації розкладу на вихідні дні

Першим критерієм сортування є weekend_count - загальна кількість вже відпрацьованих вихідних. Алгоритм прагне зрівняти це значення для всіх членів команди. Другим і найважливішим маркером є too_soon_penalty - механізмом

запобігання надмірному навантаженню. Алгоритм рахує різницю у днях між поточною датою та останнім відпрацьованим вихідним. Якщо пройшло менше 18 діб, кандидат отримує штрафний бал і система блокує призначення. Співробітник не може працювати два вихідних поспіль. Третій параметр same_day_penalty відповідає за ротацію. Якщо минулого разу людина чергувала в суботу, сьогодні алгоритм намагатиметься поставити її в неділю. Останній параметр days_since використовується як тайбрейкер. Якщо всі інші коефіцієнти рівні, зміну отримає той, хто найдовше відпочивав.

Вибір саме лексикографічного сортування кортежів замість зваженої суми параметрів обумовлений різницею в логіці прийняття рішень. Зважена сума виду $w1 \cdot a + w2 \cdot b + w3 \cdot c + w4 \cdot d$ потребує емпіричного підбору вагових коефіцієнтів w_1, w_2, w_3, w_4 для кожного параметра. При цьому неоптимальний вибір ваг може призводити до неочікуваних ситуацій. Лексикографічне сортування Python усуває цю проблему. Порівняння кортежів виконується послідовно, спершу за першим елементом, потім за другим лише за умови рівності перших, і так далі.

Розподіл будніх днів підпорядкований іншій логіці. Тут пріоритетом є безперервність робочого процесу та балансування інтенсивності. Цикл проходить по кожному дню і формує пул доступних працівників. Далі система сканує всі доступні типи денних змін із таблиці shifts та призначає їх кандидатам на основі функції: (weekday_shifts_this_week, is_weekend, total_weekday_shifts, same_shift_penalty).

Працівник не повинен перепрацьовувати в межах одного тижня. Тому weekday_shifts_this_week стоїть на першому місці. Далі спрацьовує змінна is_weekend. Якщо людина отримала зміну у вихідний день на цьому тижні, то вона автоматично опускається в кінець черги на будні дні. Це превентивний механізм компенсації навантаження. Окрім цього, алгоритм враховує same_shift_penalty, тобто якщо вчора фахівець закривав "Ранкову" зміну, сьогодні система спробує дати йому інший шаблон, щоб уникнути монотонності операцій.

Динамічне оновлення розкладу є не менш складною задачею. Працівник раптово бере відпустку на затверджений тиждень. Адміністратор фіксує це в системі. Запускається асинхронна функція regenerate_future. Це не звичайне переписування, а механізм визначає точку перетину: максимальну дату між сьогоднішнім днем та початком активного періоду. Вся історія до цієї точки залишається недоторканою. Усі зміни після неї знищуються за допомогою ORM-викликів db.delete().

Цей процес має критичний побічний ефект - фінансовий баланс відгулів. Якщо видаляється майбутня зміна у вихідний день, система зобов'язана здійснити математичний відкат. Алгоритм звертається до таблиці day_off_balance, знаходить запис для конкретного користувача та року. Якщо значення saved_days більше нуля, воно декрементується. Лише після вирівнювання фінансового стану ініціюється виклик базової функції generate_schedule для перерахунку залишку місяця.

Ручне коригування розкладу здійснюється через ендпоінт обміну (/swap). Наприклад, якщо дві людини хочуть помінятися змінами, то буде відбуватись не просто переписування ідентифікаторів user_id, а контролер виконує валідацію: будні можна міняти лише на будні; вихідні на вихідні. База перевіряє відсутність подвійних змін у кандидатів на цільові дати. Лише після цього ініціюється транзакція обміну.

Якщо обмін стосувався вихідного дня, запускається внутрішня процедура rebalance_weekdays. Це каскадне балансування. Коли людина А віддає свій вихідний людині Б, вона має отримати натомість будній день людини Б у межах цього ж тижня. Алгоритм сканує поточний тиждень, знаходить робочу зміну людини Б, яка не конфліктує з розкладом людини А, і переписує ідентифікатор. Зберігає зміни через db.commit(). Структурна цілісність навантаження відновлюється миттєво.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ОПИС ФУНКЦІОНУВАННЯ ВЕБДОДАТКА ДЛЯ УПРАВЛІННЯ РОЗКЛАДАМИ ЧЕРГУВАНЬ

3.1 Реалізація серверної частини та налаштування REST API

Програмна імплементація серверного ядра базується на асинхронному вебфреймворку FastAPI [12, 18]. Монолітну архітектуру було відкинуто на етапі проєктування, тому маршрутизацію рознесена по ізольованих функціональних доменах. Файл головного додатка не містить бізнес-логіки, він лише агрегує підключення через клас APIRouter. Ендпоінти управління персоналом зосереджені в модулі users.py, авторизація винесена в auth.py, логіка матриці чергувань керується через schedules.py. Мережева взаємодія виконується виключно за протоколом HTTP/1.1. Диспетчеризацію сирих TCP-з'єднань бере на себе ASGI-сервер Uvicorn, який ефективно транслює вхідний потік мережевих байтів у нативні Python-корутини. Процес не блокується і потоковий виконавець паралельно обробляє десятки запитів під час очікування відповіді від бази даних.

Безпека реалізована за концепцією stateless-архітектури, тобто сервер не тримає сесій користувачів у своїй оперативній пам'яті. Механізм ідентифікації повністю спирається на передачу JSON Web Tokens (JWT) [16]. Маршрут /login приймає стандартизовану форму OAuth2PasswordRequestForm через механізм ін'єкції залежностей. Контролер асинхронно звертається до СУБД PostgreSQL і витягує цільовий запис співробітника, далі працює криптографічний шар. Функція verify_password з бібліотеки Passlib обчислює хеш введеного рядка і порівнює його з bcrypt-відбитком у базі даних [21]. Якщо збіг підтверджено, генерується новий токен доступу. У його корисне навантаження вшивається унікальний ідентифікатор sub та системна role. Клієнт отримує ключ аутентифікації.

Доступ до захищених маршрутів контролюється через механізм Dependency Injection . Функції get_current_user та require_manager працюють як фільтри безпеки на рівні роутера. Фреймворк перехоплює вхідний запит ще до входу в тіло

контролера, заголовок Authorization парситься, рядок формату Bearer розшифровується і токен декодується за криптографічним алгоритмом як показано на рис 3.1. Недійсний підпис або протермінований час життя токена викликає виняток HTTPException. Сервер віддає статус 401 Unauthorized. Будь-яке неавторизоване втручання заблоковано.

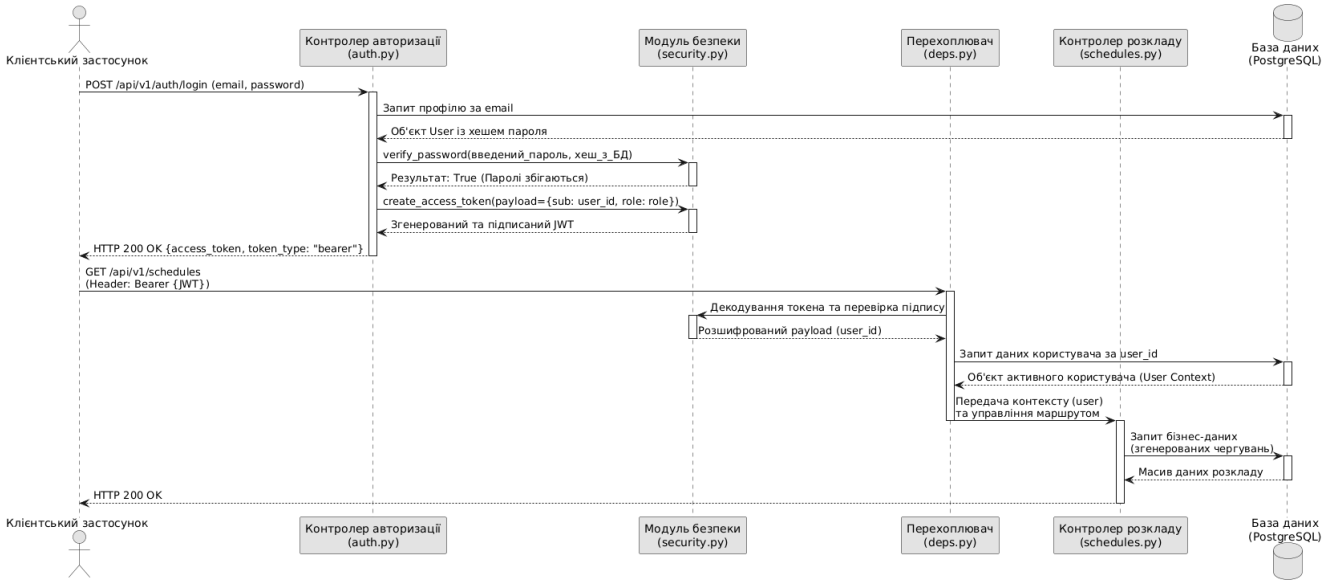


Рис. 3.1 Діаграма послідовності процесу автентифікації та валідації JWT-токена

Для захисту цілісності даних використано Pydantic. Сирі JSON-об'єкти від клієнта фізично ізольовані від бізнес-логіки бази даних. Модуль schemas.py декларує Data Transfer Objects. Процес створення або оновлення профілю працівника вимагає суворого дотримання типів полів. Особливу увагу приділено безпеці паролів. Ендпоінт PATCH /{user_id} реалізує складний механізм часткового оновлення. Адміністратор передає об'єкт UserUpdate, потім конвертує його в словник мутацій із застосуванням параметра exclude_unset=True. Оновлюються лише ті змінні, які фізично присутні в тілі запиту. Якщо виявлено поле password, контролер перехоплює його і введений рядок передається через hash_password. Оригінал видаляється зі словника операцій. База даних отримує виключно безпечний криптографічний хеш.

Взаємодія з базою даних підпорядкована правилам асинхронного об'єктно-реляційного відображення бібліотеки SQLAlchemy. Кожен контролер отримує ізольовану сесію через генератор get_db. Залежність відкриває транзакцію. Ліниве завантаження пов'язаних таблиць в асинхронному контексті неминуче спричиняє фатальну помилку MissingGreenlet, тому застосовується виключно явне жадібне завантаження. Виклик options(selectinload(Schedule.shift)) інструктує ядро бази даних підтягнути всі залежні шаблони змін одним SQL-запитом. Проблема N+1 запитів усунута архітектурно і продуктивність залишається передбачуваною і стабільною на будь-яких обсягах вибірок. Структура моделей спирається на використання uuid.UUID як первинних ключів, що ускладнює перебір ідентифікаторів зловмисниками та гарантує унікальність записів.

Найскладнішим транзакційним вузлом бекенду є обробка заяв на відпочинок. Маршрут PATCH /{leave_id}/status виконує багатоетапну каскадну мутацію стану системи. Адміністратор затверджує відгул, потім контролер ініціює ланцюг перевірок. Спочатку оновлюється фінансовий баланс вільного часу в таблиці DayOffBalance. Якщо запис для поточного року відсутній, він створюється динамічно. Поле used_days інкрементується і далі система сканує розклад на наявність конфліктів. Запит перевіряє таблицю schedules на предмет призначених змін у період затвердженої відпустки.

Сервер не обмежується простим видаленням конфліктної зміни, він намагається оптимізувати графік. Формується множина всіх активних співробітників і з неї програмно віднімаються ті фахівці, які вже мають зміну в цей день, і особи, які офіційно перебувають у відпустці. Залишається чистий пул доступних кандидатів. Перший вільний фахівець зі списку автоматично перехоплює цю зміну і ідентифікатор user_id перезаписується. Якщо це перепризначення випадає на вихідний день, бекенд виконує подвійну фінансову транзакцію. Баланс збережених відгулів попереднього власника декрементується. Баланс нового власника синхронно інкрементується, відпрацьовані години не

губляться. Якщо ж вільних людей у пулі не залишилося, зміна фізично видаляється оператором await db.delete(sch).

Усі критичні дії записуються у системний аудит. Таблиця AuditLog фіксує кожну дію керівництва і робітників. Створення екземпляра логу відбувається в рамках тієї ж сесії, що й основна транзакція. Якщо запис у розклад падає з помилкою валідації, лог також автоматично скасовується через механізм відкату бази даних.

Масова генерація розкладу інкапсульована в маршруті POST /generate. Процес стартує з примусового очищення поточного горизонту планування, тобто сервер видаляє всі майбутні записи на наступні 30 днів. Перед видаленням аналізуються типи змін. Скасований вихідний день повертається на рахунок співробітника. Далі створюється новий маркер періоду SchedulePeriod. Процесорний час передається евристичному алгоритму з модуля scheduler.py, він повертає збалансований масив призначень. Об'єкти Schedule масово завантажуються в пам'ять бази через db.add_all(). Транзакція закривається і графік фіксується.

Фінальним етапом генерації є інтеграція з корпоративною інфраструктурою підприємства. Модуль outlook.py робить можливим прямий зв'язок з Microsoft Graph API [20]. Запит виконується асинхронно через HTTP-клієнт httpx.AsyncClient. Авторизація на боці Microsoft відбувається за серверною схемою client_credentials. Бекенд обмінює облікові дані додатка на OAuth2-токен. Формується JSON-тіло календарної події. Запит відправляється на персональний ендпоінт користувача. Якщо зовнішній API Microsoft недоступний, система просто логує попередження. Основний потік виконання додатку не переривається. Розклад у локальній базі зберігається успішно за будь-яких умов.

Додатковий рівень зовнішньої інтеграції стосується статусу присутності персоналу. Затвердження відпустки в системі ініціює виклик функції create_outlook_ooo_event. Вона генерує подію на цілий день зі специфічним маркером showAs: oof (Out of Office). Календар співробітника блокується для колег.

Одночасно спрацьовує метод set_user_auto_reply, який надсилає PATCH-запит безпосередньо до конфігурації поштової скриньки. Автоматична відповідь активується. Внутрішні та зовнішні адресати отримують відповідне HTML-повідомлення про відсутність фахівця.

Ручне коригування розкладу адміністратором здійснюється через маршрут POST /swap. Клієнт передає два унікальні ідентифікатори існуючих чергувань. Контролер завантажує їх із бази. Далі працює жорстка бізнес-валідація: перевіряється сумісність категорій часу. Будні дні міняються виключно на будні, вихідні на вихідні. База окремо блокує спроби подвійних призначень. Якщо виявлено, що співробітник вже має зміну в цільову дату, операція відхиляється статусом 400 Bad Request.

Окрім прямої інтеграції через Graph API, реалізовано універсальний механізм синхронізації. Маршрут GET /export/me генерує файл стандарту RFC 5545 (iCalendar) [15]. Контролер збирає всі майбутні чергування та підтверджені відгули для активного профілю. Формується масив текстових інструкцій, змінні часу конвертуються у формат YYYYMMDDTHHMMSSZ, кожна подія огортається ідентифікаторами BEGIN:VEVENT та END:VEVENT. Відгули додатково маркуються директивою X-MICROSOFT-CDO-BUSYSTATUS:OOF. Спеціальний клас PlainTextResponse повертає сформований блок тексту. Сервер додає HTTP-заголовок Content-Disposition, що змушує браузер ініціювати пряме завантаження файлу .ics.

3.2 Програмна імплементація обчислювального ядра

Обчислювальне ядро системи інкапсульовано у модулі scheduler.py, який є автономним математичним рушієм. Його задача - розв'язання проблеми дискретної оптимізації при розподілі часових слотів. Якщо використовувати класичний brute-force, утворюється велика кількість комбінацій, які при генеруванні розкладу значно уповільнили би продуктивність бекенду. Застосовано жадібну евристику,

вона працює ітеративно [1, 3]. Рішення ухвалюються локально для кожного дня на основі динамічної Penalty Function.

Робота алгоритму починається з агрегації контексту. Ядро ініціює серію асинхронних запитів до бази даних PostgreSQL і витягує пул усіх активних співробітників, які мають системну роль employee. Далі система сканує реляційну таблицю leaves і усі затверджені відгули та відпустки перетворюються на структуру даних недоступності. У коді це імплементовано через defaultdict(set) з модуля collections. Вибір структури типу set є критичним архітектурним рішенням, яке гарантує перевірку входження дати за константний час O(1), запобігаючи деградації швидкодії від лінійного пошуку O(N) у стандартних списках на кожній ітерації генерації.

Також в системі формується історичний бекграунд, адже алгоритм не може справедливо розподілити майбутні зміни, не знаючи минулі. База даних повертає лог усіх чергувань до дати початку нового періоду планування. При цьому застосовується механізм завантаження selectinload(Schedule.shift). Це запобігає виникненню проблеми N+1 запитів під час аналізу типів змін в ORM SQLAlchemy. Цикл проходить по масиву past_schedules. Локальні змінні стану ініціалізуються також через defaultdict. Це захист від виключень KeyError. Новий працівник не має історії, замість помилки виконання він отримує базові значення: date(2000, 1, 1) для останнього вихідного та нуль для лічильників інтенсивності. Оперативна пам'ять сервера наповнюється точними метриками: хто скільки вихідних відпрацював (weekend_count), коли це було фізично (last_weekend_date) та на який день тижня припало (last_weekend_day).

Фаза перша: розподіл вихідних днів. Цикл while перебирає календар від date_from до date_to і алгоритм перевіряє умову current_date.weekday() >= 5. Поточний індекс вказує на суботу або неділю. Формується масив доступних кандидатів available, потім кожен об'єкт працівника пропускається через цільову функцію сортування weekend_key(u), вона повертає кортеж із чотирьох числових параметрів. Рушій Python нативно сортує кортежі лексикографічно. Елемент з

індексом нуль має найвищу вагу. Наступні індекси в кортежі працюють як Tie-breakers для вирішення конфліктів.

Перший параметр кортежу - weekend_count. Вирівнюється загальна кількість вихідних змін серед команди. Співробітник з меншою кількістю чергувань автоматично підіймається вгору по списку. Другий параметр - too_soon_penalty, який є запобіжником вигорання. Алгоритм вираховує змінну days_since. Якщо різниця між поточною датою генерації та останнім робочим вихідним працівника менша за 18 діб, кандидат отримує штрафний бал. Одиниця замість нуля. Цей математичний тригер фізично блокує призначення людини на два вихідних поспіль. Третій параметр - same_day_penalty відповідає за ротацію. Минулого разу працівник чергував у суботу, а сьогодні система намагатиметься дати йому неділю. Збіг днів генерує алгоритмічний штраф. Четвертий параметр виступає інвертованим часом відпочинку -days_since. У випадку, якщо всі попередні штрафи в кандидатів абсолютно однакові, зміну отримає той, хто найдовше не працював. Масив сортується через sorted(available, key=weekend_key). Перший елемент списку candidates[0] забирає зміну.

Результат обчислень не записується в базу даних відразу. Відкривати транзакцію на кожній ітерації занадто ресурсномістко, натомість формується словник з ключами user_id, shift_id, shift_date та додається в локальний масив оперативної пам'яті assignments. Одразу змінюється історичний стан кандидата і лічильник вихідних інкрементується. Спеціальна змінна works_weekend_this_week отримує прапорець True для конкретного тижня року week_num. Цей маркер важливий для наступного етапу обчислень.

Фаза друга: будні дні. Вказівник дати повертається на початок періоду і запускається цикл while з умовою weekday() < 5. Пріоритети сортування докорінно змінюються, оскільки логіка будніх днів відрізняється від вихідних. Цільова функція weekday_key(u) формує новий тип кортежу, де на першому місці стоїть weekday_shifts_this_week. Алгоритм рівномірно розподіляє чергування всередині одного тижневого спринту. Другим фактором йде is_weekend, збережений маркер

з попередньої фази. Працівник отримав суботню зміну на першому етапі розрахунків. Значення маркера дорівнює одиниці і кандидат падає в самий кінець черги на призначення будніх днів. Так ядро математично компенсує навантаження. Останнім фактором є same_shift_penalty, де перевіряється last_shift_id. Вчора була призначена ранкова зміна, тоді сьогодні алгоритм автоматично змістить пріоритет на користь вечірньої, таким чином варіюється час чергувань, адже в різний час доби різне навантаження. Якщо кандидат отримав призначення, тоді його об'єкт примусово видаляється з локального масиву current_available. Вкладений цикл переходить до наступного шаблону зміни на цей же день.

Після завершення двох фаз у пам'яті сервера зберігається готовий масив assignments. База даних досі залишається порожньою. Ініціюється фінальна конвертація. Генератор списку створює екземпляри ORM класів [Schedule(assign) for assign in assignments], і виконується метод db.add_all(). Всі записи записуються у реляційне сховище єдиним масивним мережевим пакетом і транзакція закривається.

Якщо працівник несподівано захворів або взяв авансовий відгул, у таблиці leaves з'являється новий затверджений запис. Система запускає процедуру часткового перерахунку regenerate_future і визначає regen_start. Знаходиться максимальна дата між сьогоднішнім днем та початком активного періоду графіка. Усі зміни, які знаходяться після точки перетину очищаються. Перед фізичним видаленням запису ORM перевіряє тип зміни. Зміна випадала на вихідний день, тоді система робить відкат. Виконується select запит до таблиці DayOffBalance і знаходить відповідний запис для конкретного року та користувача, тоді поле saved_days декрементується. Лише після вирівнювання балансу акумульованих відгулів, запис розкладу видаляється через виклик db.delete(sch). Звільнений діапазон дат заповнюється наново. Ядро викликає базову функцію generate_schedule для залишку періоду і згенеровані матриці знову проходять через цикл оновлення балансів. Нові вихідні інкрементують фінансовий показник saved_days. Тільки після цього відбувається фінальний db.commit().

Окремим логічним модулем обчислювального ядра є маршрутизатор обміну змінами /swap. Інтерфейс передає два об'єкти sch1 та sch2. Встановлюється перша валідація типізації: умова sch1.shift.shift_type != sch2.shift.shift_type. Будні дні міняються виключно на будні. Спроба обміняти робочий вівторок на суботу повертає статус 400 Bad Request. Наступним етапом є перевірка відсутності конфліктів, на якому алгоритм сканує базу даних на наявність інших активних змін у цільові дати для обох працівників. Після успішної валідації ідентифікатори user_id перезаписуються, реалізуючи обмін.

Якщо обмін стосується вихідних днів, система автоматично виконує балансування через функцію rebalance_weekdays. Співробітник А віддав своє вихідне чергування співробітнику Б. Оскільки передача вихідного дня порушує рівномірність тижневого навантаження, алгоритм вираховує хронологічні межі поточного робочого тижня week_start та week_end. Формується запит із застосуванням логічного оператора and_. Ядро шукає всі звичайні будні зміни співробітника Б на цьому ж тижні. Щойно зміну знайдено, алгоритм перевіряє, чи не призведе її передача співробітнику А до хронологічного конфлікту. Якщо конфлікту немає, зовнішній ключ user_id перезаписується в пам'яті. Будній день переходить від Б до А і тижнева рівновага годин відновлена без втручання адміністратора.

Кожна операція генерації розкладу, його перерахунку або обміну змінами фіксується в підсистемі аудиту через створення запису класу AuditLog з ідентифікатором ініціатора, типом дії та деталізованим описом. Запис аудиту додається в поточну транзакцію бази даних. У разі виникнення помилки на будь-якому етапі виконується відкат транзакції через db.rollback(), що повертає всі таблиці, включаючи журнал аудиту, до початкового стану. Така організація гарантує узгодженість даних при конкурентному доступі.

3.3 Реалізація підсистеми обліку відгулів та інтеграції

Підсистема обліку відгулів та зовнішніх інтеграцій є критичним компонентом системи, який ізольований в окремому домені leaves.py. Логіка роботи виходить за межі стандартних CRUD-операцій. Це комплексний механізм, який безпосередньо впливає на баланс розкладу та взаємодію з корпоративними системами через зовнішні API. Реалізація модуля вимагає дотримання принципів ACID для забезпечення атомарності кожної зміни статусу заявки [10, 22].

Процес починається з подання запиту співробітником через клієнтський інтерфейс на Vue.js. Ендпоінт POST / приймає дані LeaveCreate, які Pydantic автоматично валідує на відповідність типів дат та формату заявки (vacation, day_off). Після валідації запит зберігається в базі даних зі статусом pending без змін у поточному розкладі до отримання адміністративного рішення.

Основна бізнес-логіка виконується під час виклику маршруту PATCH /{leave_id}/status. Менеджер надсилає команду на затвердження. Контролер асинхронно ініціює з'єднання з базою даних, використовуючи ORM-метод selectinload для завантаження пов'язаних даних користувача. Якщо переданий статус має значення approved, запускається багатоступеневий каскадний процес.

Перший етап каскаду є фінансовим обліком вільного часу. Якщо тип заявки визначено як відгул day_off, система змінює агрегований баланс у таблиці DayOffBalance. Пошук виконується за композитним ключем: ідентифікатор користувача та поточний рік вибірки. Якщо запис для активного року ще не створено, бекенд ініціалізує його нульовими значеннями. Значення поля used_days інкрементується зі збереженням транзакційної цілісності. Працівник може подати запит навіть за умови відсутності накопичених днів. Система легально працює з авансовими відгулами, що закладає основу гнучкої корпоративної культури. Різниця балансів обчислюється динамічно під час рендерингу інтерфейсу на фронтенді.

Другий етап каскаду - вирішення конфліктів розкладу. Затверджена відпустка може перетинатися із заздалегідь згенерованим планом чергувань.

Сервер не повинен просто видалити запис, оскільки це порушить безперервність робочого процесу. Ендпоінт виконує select запит до таблиці schedules і сканує інтервал між date_from та date_to. Якщо знайдено конфліктні зміни, тоді алгоритм ініціює процедуру автоматичного перепризначення. Програмно формується множина busy_users, що агрегує ідентифікатори всіх співробітників, які вже залучені до роботи в конфліктний день або які самі перебувають у затвердженій відпустці. Ця множина віднімається від загального пулу активних працівників. Формується масив вільних робітників available. Якщо в резерві є вільний фахівець, система передає йому проблемну зміну, перезаписуючи зовнішній ключ user_id. Якщо делегована зміна припадає на вихідний день ShiftType.weekend, ініціюється додаткова балансувальна транзакція. Лічильник saved_days нового власника збільшується на одиницю, а лічильник попереднього власника зменшується. Тільки у випадку повної відсутності доступних кандидатів, зміна фізично знищується оператором await db.delete(sch).

Третій етап - протоколювання дій. Затвердження відгулу є обов'язковою адміністративною дією, також менеджер має знати день відгулу і недоступності працівника. Ядро ініціалізує створення запису в таблиці AuditLog і фіксує час транзакції, ідентифікатор автора підтвердження, категорія дії та текстовий коментар із вказанням цільового періоду. Лог-запис додається в ту саму сесію SQLAlchemy, що й основні мутації бази даних. Якщо на будь-якому з етапів (перерахунок балансу або делегування зміни) виникне виняток, вся глобальна транзакція повертається командою db.rollback() і скасовані або пошкоджені операції не потрапляють в аудит.

Четвертий етап - інтеграція з екосистемою Microsoft. Пряма інтеграція через Microsoft Graph API для встановлення статусів Out of Office та активації автовідповідей повністю реалізована на рівні серверної логіки в модулі outlook.py. Проте її практичне використання в продакшн-середовищі потребує додаткового налаштування корпоративної інфраструктури. Зокрема, необхідна реєстрація додатка в Microsoft Entra ID з наданням делегованих прав на рівні адміністратора

організації Service-to-Service автентифікація. Для тестування та повноцінної демонстрації цього функціоналу потрібні корпоративні облікові дані, що виходить за межі можливостей локального середовища розробки. Крім того, процес узгодження таких прав у реальному підприємстві може тривати тижнями. Тому мережеві виклики загорнуті в блоки try...except. За відсутності налаштованих ключів середовища система пропускає цей етап, генеруючи попередження в лог.

З огляду на ці корпоративні обмеження, основним механізмом інтеграції обрано автономну генерацію файлів календаря. Замість використання мережевих протоколів інтеграція базується на формуванні документів за відкритим стандартом RFC 5545. Бекенд не використовує сторонніх бібліотек: маршрут GET /export/me виконує запит до бази даних, отримує масив підтверджених чергувань і нативно серіалізує часові мітки. Формується текстовий масив інструкцій BEGIN:VEVENT, де відгули маркуються директивою X-MICROSOFT-CDO-BUSYSTATUS:OOF. Користувач завантажує готовий файл формату .ics через PlainTextResponse. Згенерований розклад імпортується в MS Outlook, Google Calendar або інший сумісний клієнт без необхідності налаштування OAuth-авторизації чи отримання корпоративних дозволів.

Інтерфейс користувача для управління цією системою ізольований у вкладках Vue-компонента. Клієнтська логіка використовує бібліотеку Axios. Метод requestLeave форматує дати та надсилає POST-запит. Локальний реактивний стан не оновлюється автоматично, додаток чекає підтвердження від сервера. Менеджер працює з вкладкою «Адмінка», де виклик функції approveLeave ініціює зміну статусу. Після успішного PATCH-запиту фронтенд виконує глобальне оновлення стану через допоміжну функцію loadPublicData(), і таблиця розкладу оновлюється. Реактивність гарантує узгодженість візуального відображення даних без необхідності перезавантаження сторінки.

3.4 Розробка користувацького інтерфейсу

Клієнтський інтерфейс системи побудовано як односторінковий додаток. Фундаментом слугує JavaScript-фреймворк Vue.js версії 3 з використанням парадигми Composition API [28]. Вона дозволяє інкапсулювати логіку всередині блоку <script setup>. Стан компонента є локальним і глобальні сховища даних на кшталт Pinia або Vuex не застосовуються. Браузер завантажує DOM-дерево лише один раз. Подальша маршрутизація та оновлення екранів відбуваються виключно силами віртуального DOM. Сторінка не перезавантажується.

Процес автентифікації реалізований окремим екраном додатка. Користувач не вводить логін вручну, інтерфейс рендерить випадаючий список з іменами співробітників. Пароль передається через стандартне поле з прихованим вводом. Для ініціалізації сесії програмний метод формує об'єкт типу FormData, такий формат передачі даних зумовлений вимогами специфікації авторизації OAuth2 [14]. Відправка облікових даних виконується асинхронним POST-запитом за допомогою HTTP-клієнта Axios [7]. У разі успішної перевірки бази даних сервер генерує та повертає клієнту JWT-токен доступу. Рядок токена записується у axios.defaults.headers.common['Authorization']. Завдяки такому підходу кожен наступний мережевий запит автоматично містить необхідний цифровий підпис, що дозволяє серверу ідентифікувати користувача, перевірити його системну роль та надати доступ до відповідних захищених ресурсів. Вигляд екрана входу з випадаючим списком користувачів та полем пароля наведено на рис. 3.2.

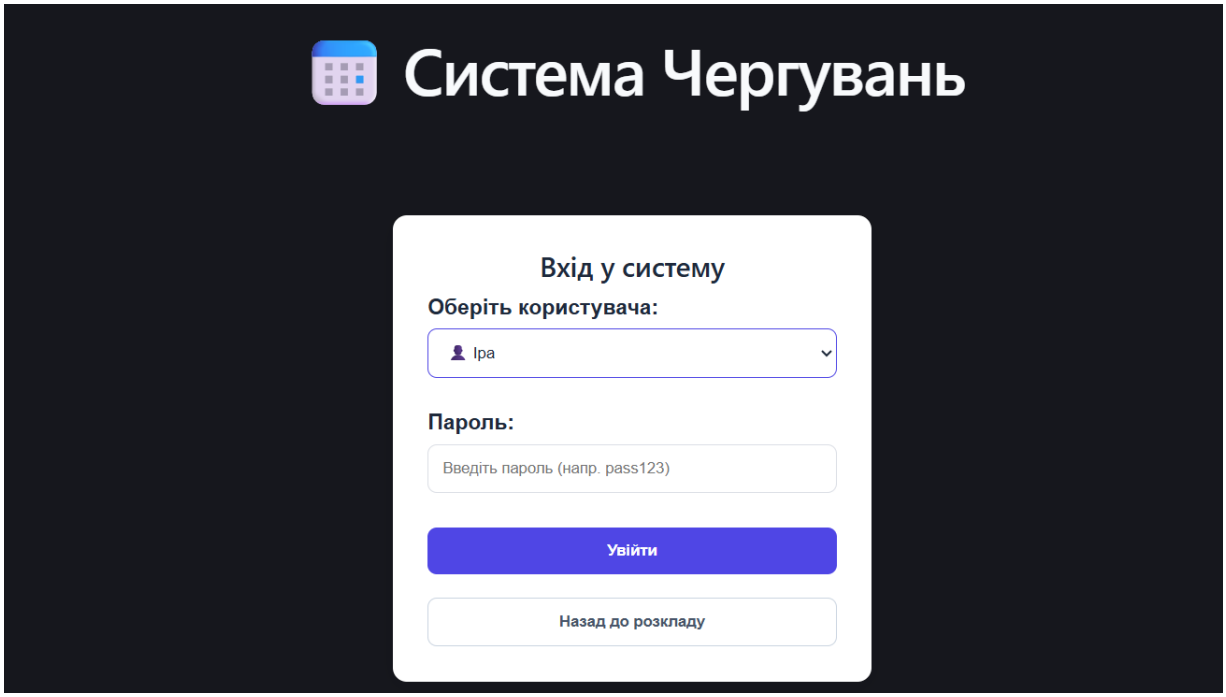


Рис. 3.2 Екран автентифікації користувача

Рольову модель доступу реалізовано на рівні клієнтського шаблону. Об'єкт currentUser містить поле role, на основі якого директиви v-if умовно рендерять елементи інтерфейсу. Працівник з роллю employee має доступ до розділу «Мій кабінет». Менеджер отримує доступ до адміністративної панелі. Перемикання між вкладками реалізовано через реактивну змінну activeTab, зміна якої оновлює CSS-класи кнопок та рендерить відповідний блок інтерфейсу без серверного рендерингу.

Ключовою задачею клієнтської частини є візуалізація матриці розкладу. Бекенд повертає одновимірний масив об'єктів schedules, де кожен елемент містить дату, зміну та користувача. Для побудови табличного представлення ці дані трансформуються у двовимірну структуру через обчислювані властивості Vue.js computed properties, які кешують результати та перераховуються лише при зміні вхідних даних.

Властивість uniqueDates формує горизонтальну вісь таблиці: витягує всі дати з масиву, видаляє дублікати через Set та сортує хронологічно. Властивість shiftRows формує вертикальну вісь: збирає унікальні назви змін та сортує їх з

урахуванням бізнес-логіки - зміни з підрядком «Вихідний» розміщуються внизу таблиці, будні дні залишаються зверху.

Рендеринг таблиці виконується через вкладені цикли v-for. Окремо обробляється рядок відпусток функцією getLeavesForDate, яка фільтрує затверджені заявки для кожної дати. Для заповнення комірок змін викликається getShiftUser, що повертає ім'я призначеного працівника для відображення у вигляді бейджу.

Для реалізації горизонтальної прокрутки з фіксованою першою колонкою використано CSS-властивості position: sticky та left: 0 для стовпця .shift-col. Контейнер .table-responsive обмежує ширину, а таблиця .excel-grid має width: max-content для виходу за межі видимої області. Псевдоелемент ::after з градієнтом створює візуальний ефект тіні для покращення сприйняття. Ергономіку інтерфейсу посилює функція isToday, яка порівнює системну дату клієнта з датами у таблиці та динамічно додає клас .today-highlight до стовпця поточного дня, виділяючи його кольором та рамками. Згенеровану матрицю розкладу у вигляді календарної сітки з підсвіченням поточного дня показано на рис. 3.3.

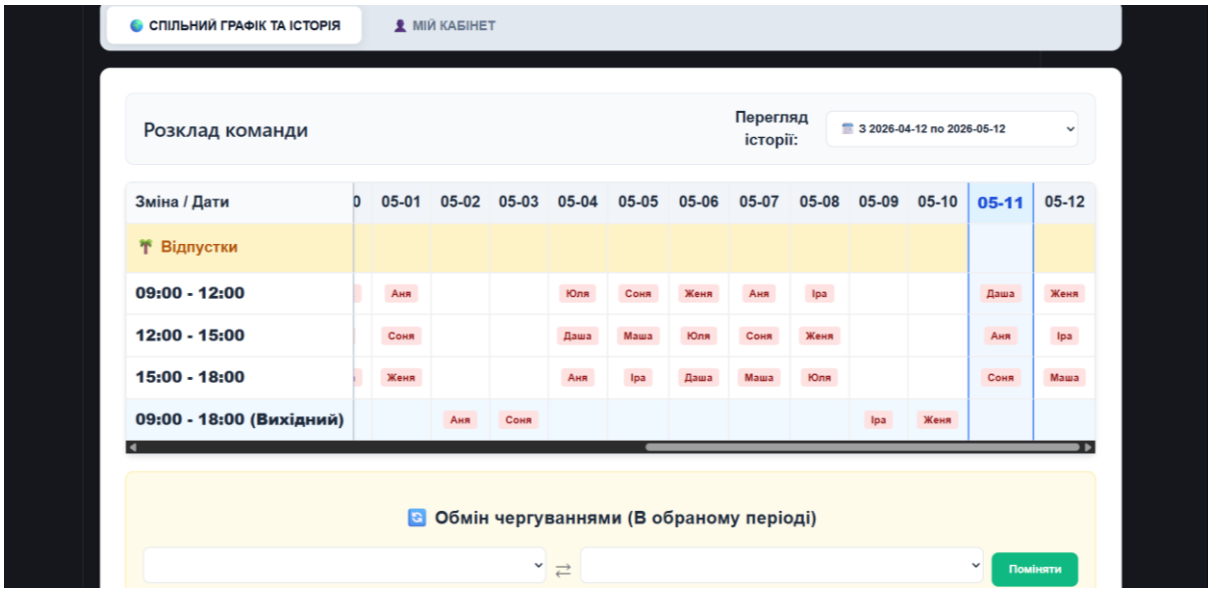


Рис. 3.3 Візуальне відображення згенерованої матриці розкладу

Експорт календаря реалізовано через завантаження .ics файлів. При натисканні кнопки функція exportCalendar надсилає запит з параметром responseType: 'blob', отримує бінарні дані від сервера, створює тимчасове посилання через window.URL.createObjectURL, програмно ініціює завантаження файлу та видаляє тимчасові елементи для запобігання витоку пам'яті. Загальний вигляд особистого кабінету працівника з власним графіком чергувань, балансом днів відгулу та формами заявок наведено на рис. 3.4.

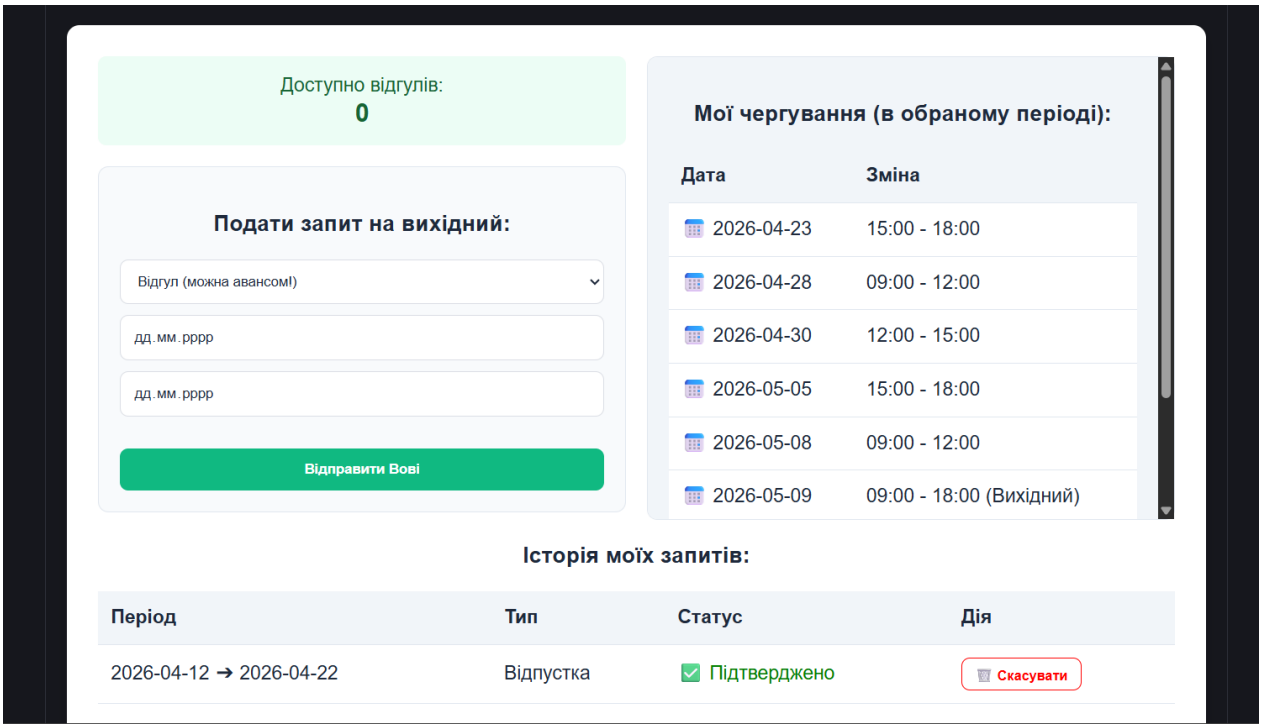


Рис. 3.4 Особистий кабінет працівника

Адміністративна панель агрегує критичні операції управління системою. Найбільш ресурсомісткою є генерація розкладу. Перед відправкою запиту реактивна змінна isGenerating перемикається у true, що блокує кнопку та змінює її текст на «Завантаження...». Цей патерн запобігає повторним викликам та паралельним генераціям, які могли б порушити транзакційну цілісність бази даних. Після завершення POST-запиту до /api/schedules/generate змінна скидається, а функція loadPublicData() виконує паралельні GET-запити для оновлення локальних масивів розкладів, користувачів, заявок та записів аудиту. Адміністративну панель

менеджера з елементами керування для запуску генерації розкладу та управління персоналом продемонстровано на рис. 3.5.

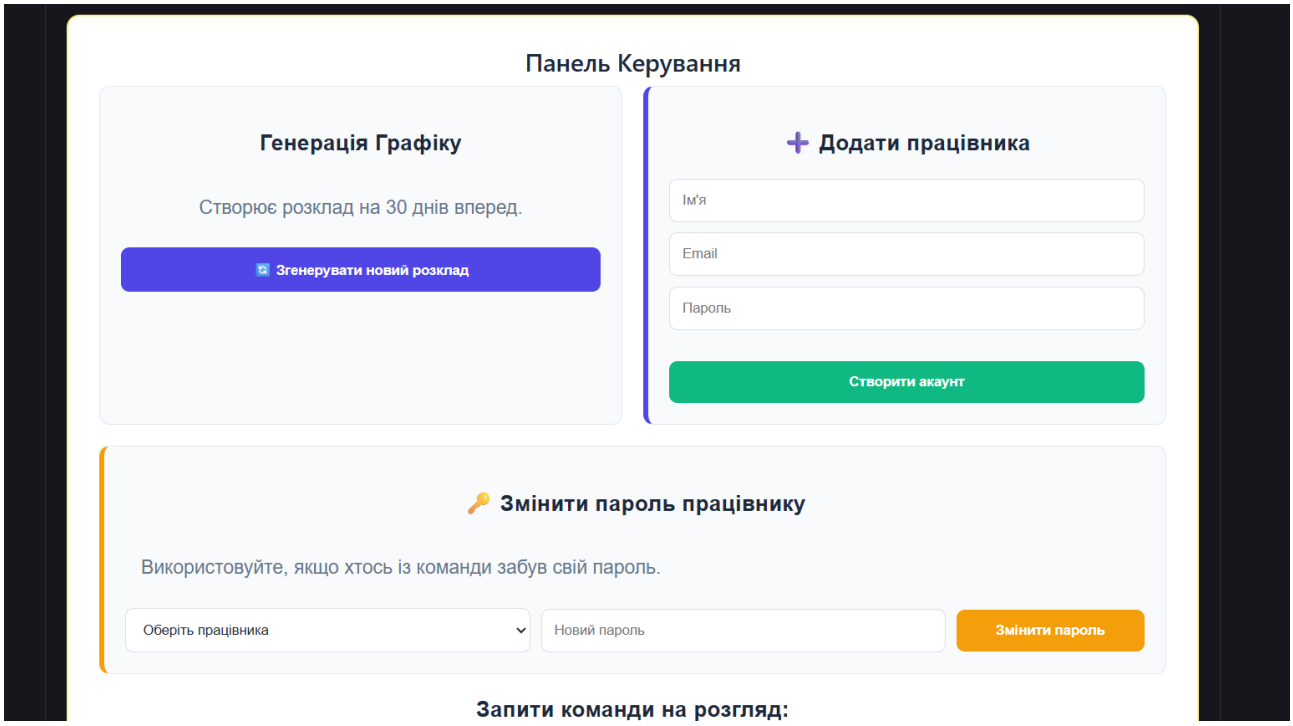


Рис. 3.5 Панель керування менеджера

Блок управління персоналом розділено засобами CSS Grid на дві зони. Форма створення облікового запису прив'язана через v-model до змінних newEmpName, newEmpEmail та newEmpPassword з локальною валідацією заповненості полів. Нижче частина реалізує адміністративне відновлення доступу: випадаючий список рендерить опції з масиву users через директиву v-for, обраний ідентифікатор зберігається у змінній selectedUserToUpdate, після чого PATCH-запит до /api/users/{user_id} оновлює виключно поле password згідно з принципом часткового оновлення ресурсу.

Реактивна модель Vue.js забезпечує автоматичне оновлення інтерфейсу при змінах даних через систему Proxy-об'єктів JavaScript. При використанні функцій ref() та reactive() фреймворк поміщає реактивні дані у Proxy-обгортки, які перехоплюють операції читання та запису. Це дозволяє відстежувати, які саме компоненти підписані на конкретні значення та оновлювати виключно потрібні

вузли DOM-дерева. У контексті матриці розкладу така архітектура є критичною, бо типова таблиця на місяць містить понад 600 окремих комірок. При зміні статусу однієї заявки після PATCH-запиту до /api/leaves/{leave_id}/status фронтенд оновлює локальний масив allLeaves, і Vue автоматично відображає лише ті комірки таблиці, які залежать від змінених даних.

Обробка заявок на відпустки та відгули виконується через ітерацію масиву allLeaves з директивою v-for. Кожна заявка візуалізується карткою з кольоровим маркуванням статусу: pending - помаранчевий, approved - зелений, rejected - червоний. Натискання кнопки «Затвердити» викликає approveLeave, яка надсилає PATCH-запит до /api/leaves/{leave_id}/status. Серверна частина виконує каскадну транзакцію оновлення статусу, перерахунку балансу компенсаційних днів та перепризначення конфліктних чергувань, після чого локальний стан інтерфейсу оновлюється. Інтерфейс перегляду активних заявок команди разом і журнал аудиту зображено на рис. 3.6.

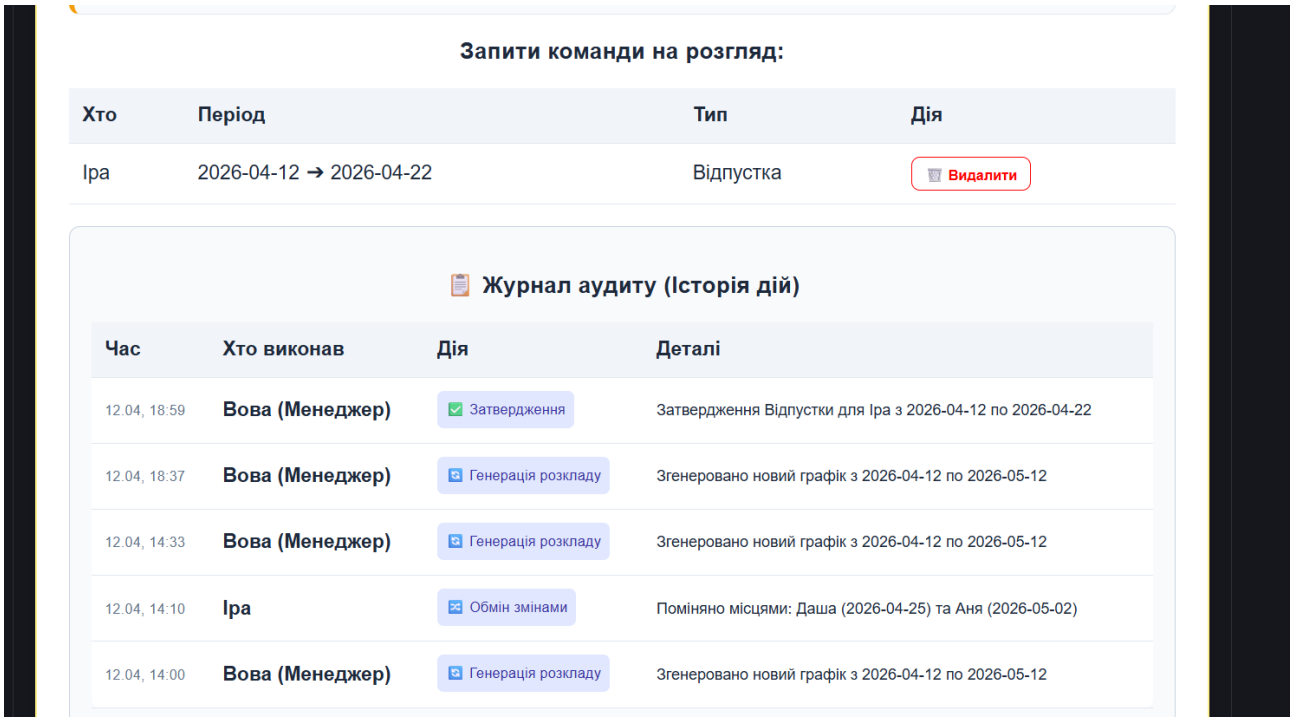


Рис. 3.6 Інтерфейс запитів команди і журнал аудиту на сторінці менеджера

Журнал аудиту розміщено у нижній частині адміністративної панелі. Серверні таймстампи у форматі ISO 8601 з прив'язкою до UTC конвертуються у локальне представлення функцією formatDateTime через нативний метод toLocaleString('uk-UA'). Записи відображаються у таблиці з виокремленням типу дії візуальними бейджами. Контейнер обмежено властивістю max-height: 400px з overflow-y: auto, що формує ізольовану зону прокрутки незалежно від обсягу історичних даних.

Механізм обміну чергуваннями реалізовано як компактний віджет з двома селектами, прив'язаними до змінних swapShift1 та swapShift2. Списки опцій фільтруються за належністю до активного періоду планування, значенням value є UUID-ідентифікатор запису. Функція swapSchedules надсилає POST-запит до /api/schedules/swap, серверна частина виконує валідацію типів змін та перевірку конфліктів. У разі помилки 400 Bad Request блок catch парсить структуру e.response?.data?.detail та виводить конкретну причину відмови, отриману від бекенду.

Візуальне оформлення інтерфейсу базується на анімаційних переходах через клас .fade-in з @keyframes, який дає плавну зміну прозорості за 0.4 секунди. Кольорова палітра побудована на сіро-синіх відтінках (#f8fafc) для фонів, індиго (#4f46e5) для основних дій та смарагдовому зеленому (#10b981) для підтверджень. Така функціональна диференціація кольорів знижує когнітивне навантаження адміністратора при роботі з системою.

4 ТЕСТУВАННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Методологія тестування та перевірка серверних компонентів

Архітектура «Shift Scheduler» розділена на клієнт, сервер і базу даних, тож тестування організовано на кількох рівнях. Тестування відбувалось від атомарних модульних перевірок до інтеграційних сценаріїв клієнт-серверної взаємодії. Базовим принципом верифікації обрано поєднання методів чорної та білої скриньки. Перший застосовувався для оцінки відповідності зовнішньої поведінки системи бізнес-вимогам. Другий використовувався для покриття внутрішніх гілок алгоритму генерації розкладу, де критично важливо перевірити коректність обчислення штрафних коефіцієнтів.

Автоматизоване модульне тестування реалізовано за допомогою фреймворку pytest з розширенням pytest-asyncio для роботи з асинхронними функціями [24]. Тестове середовище ізольовано від продакшн-конфігурації. Розгорнуто окремий екземпляр СУБД PostgreSQL з тестовою базою даних, яка перед кожним запуском набору перевірок ініціалізується скриптом init_db.py та наповнюється контрольними даними через seed.py. Повторний запуск однакового сценарію за тих самих вхідних даних обов'язково має генерувати ідентичну вихідну матрицю розкладу. Серверну частину запущено у конфігурації відладки з активним логуванням SQL-запитів SQLAlchemy, що дозволяє контролювати фактичні звернення до бази даних та виявляти проблему N+1 запитів.

Першим об'єктом верифікації став модуль безпеки security.py, який реалізує криптографічні операції. Тестова функція test_security_password_hashing виконує послідовну перевірку трьох ключових інваріантів: незворотність хешу, коректність валідації правильного пароля та відхилення помилкового. На вхід подається текстовий рядок «secure_password», який пропускається через функцію get_password_hash. Перше твердження assert підтверджує, що згенерований хеш не

збігається з вихідним паролем, що відповідає базовій вимозі однонаправленого перетворення. Друге твердження верифікує, що функція verify_password повертає True при порівнянні оригінального пароля з його хешем. Третя перевірка передає навмисно зіпсований рядок «wrong_password» та очікує отримати False, що засвідчує неможливість обходу криптографічного захисту. Усі три перевірки виконуються успішно, що підтверджує коректність роботи алгоритму bcrypt через бібліотеку Passlib [21].

Окремим об'єктом тестування виступили функції генерації та декодування JSON Web Tokens. Тест test_jwt_token_lifecycle перевіряє повний життєвий цикл сесійного токена. На вхід функції create_access_token передається словник з ідентифікатором користувача у форматі UUID та полем role зі значенням «manager». Перші два твердження верифікують, що результат є рядком ненульової довжини, а саме більшим за 20 символів, що відповідає мінімальній довжині валідного JWT-токена з підписом HS256. Згенерований токен передається у функцію decode_token, після чого виконується послідовна перевірка відновленого корисного навантаження. Поля sub та role мають точно збігатися з початковими значеннями, що передавалися у функцію генерації. Наявність автоматично доданого поля exp підтверджує коректність розрахунку часу закінчення дії токена згідно з константою ACCESS_TOKEN_EXPIRE_MINUTES.

Модуль валідації даних перевірено через тест test_pydantic_schema_validation, який охоплює як позитивні, так і негативні сценарії. У першій частині перевірки створюється об'єкт LeaveCreate з валідним набором полів: дві коректні дати у форматі date та значення type «vacation». Тест підтверджує, що екземпляр класу формується без помилок та зберігає передане значення типу заявки. У другій частині передається свідомо некоректна структура, де поля date_from та date_to містять рядки «not-a-date» замість об'єктів дати. Очікуваною поведінкою є генерація винятку ValidationError, що перехоплюється контекстним менеджером pytest.raises. Успішне завершення цього блоку засвідчує,

що Pydantic коректно відхиляє невалідні структури ще до моменту звернення до бази даних, формуючи перший рубіж захисту від некоректних вхідних даних.

Стійкість системи до зовнішніх збоїв перевірено асинхронним тестом test_outlook_service_graceful_degradation, що використовує декоратор pytest.mark.asyncio. Тест моделює критичну ситуацію відсутності корпоративних дозволів Microsoft Entra ID, що є типовим сценарієм при розробці без доступу до адміністратора організації. У межах тесту змінна MICROSOFT_CLIENT_ID примусово очищується на порожній рядок, після чого викликається асинхронна функція create_outlook_event з тестовою електронною адресою та часовими параметрами. Очікуваним результатом є повернення значення False без переривання основного потоку виконання та без генерації необроблених винятків.

Після завершення тесту початкове значення конфігураційної змінної відновлюється для запобігання впливу на інші тести. Розклад успішно зберігається в локальній базі даних незалежно від доступності корпоративних API Microsoft. Результат запуску всіх чотирьох модульних тестів зі статусом «4 passed» наведено на рис. 4.1.

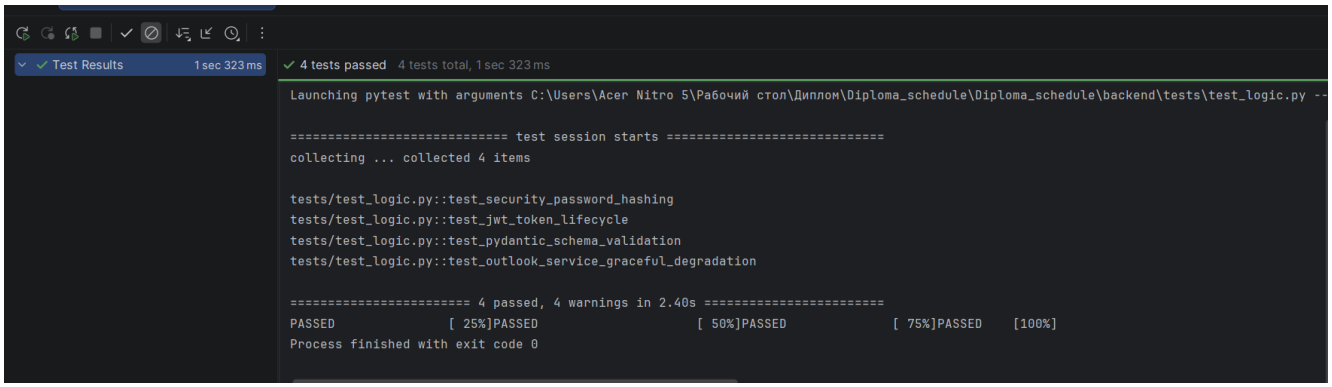


Рис. 4.1 Успішне проходження модульних тестів

4.2 Інтеграційне та функціональне тестування системи

Інтеграційне тестування перевіряє повний цикл обробки HTTP-запиту: від десеріалізації вхідного JSON через бізнес-логіку контролера до запису в базу даних та формування відповіді. Як інструмент тестування використано вбудовану інтерактивну документацію OpenAPI [12]. Фреймворк FastAPI автоматично генерує цей інтерфейс на основі анотацій типів Python та схем Pydantic, що дає змогу виконувати реальні API-виклики безпосередньо з браузера без залучення сторонніх інструментів. Інтерфейс інтерактивної документації Swagger UI зі списком усіх ендпоінтів системи показано на рис. 4.2.

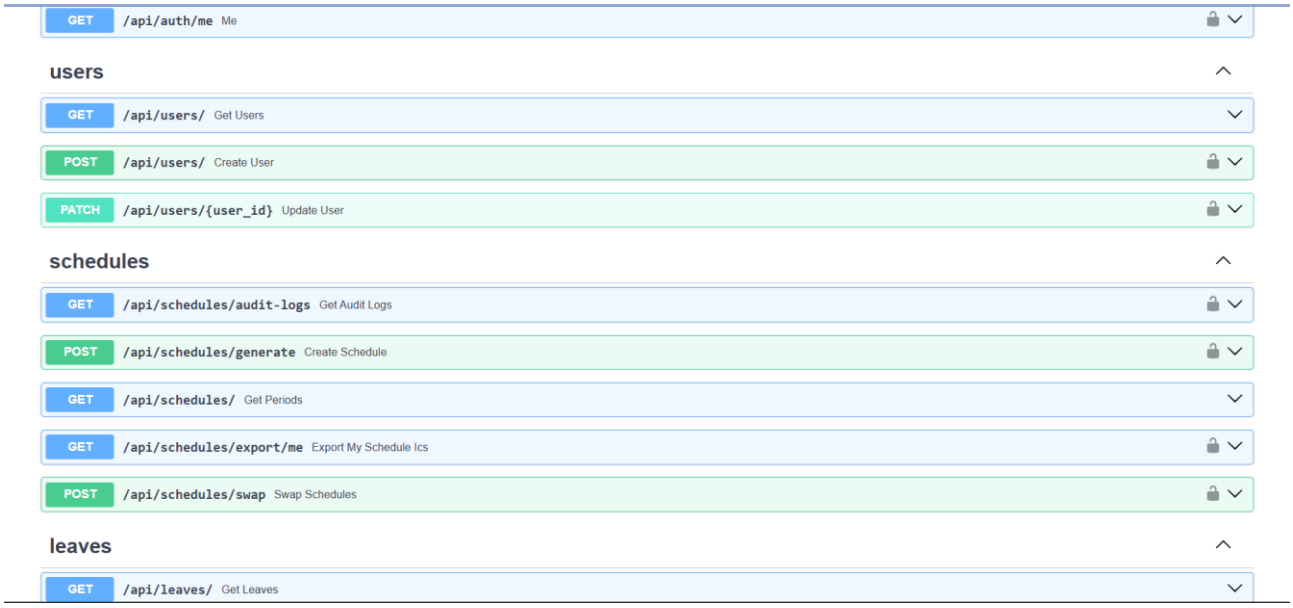


Рис. 4.2 Інтерактивна документація OpenAPI для тестування REST API

Перший інтеграційний сценарій охоплює процедуру автентифікації. У форму ендпоінта POST /api/auth/login передавалися облікові дані сід-користувача з адресою vova@company.com та паролем pass123. Сервер повернув HTTP-статус 200 OK з JSON-тілом, що містить поле access_token та об'єкт user зі структурою UserOut. Перевірено, що поле hashed_password відсутнє у відповіді - схема серіалізації коректно приховує криптографічно чутливі дані. Повторна авторизація з навмисно зіпсованим паролем повертає HTTP 401 Unauthorized з повідомленням

«Невірний email або пароль», що відповідає очікуваному захисному механізму. Спроба доступу до захищеного маршруту GET /api/auth/me без HTTP-заголовка Authorization так само блокується статусом 401, що підтверджує коректну роботу ін'єкції залежності get_current_user. Результат успішного виконання POST-запиту до /api/auth/login зі статусом 200 OK та згенерованим access_token проілюстровано на рис. 4.3.

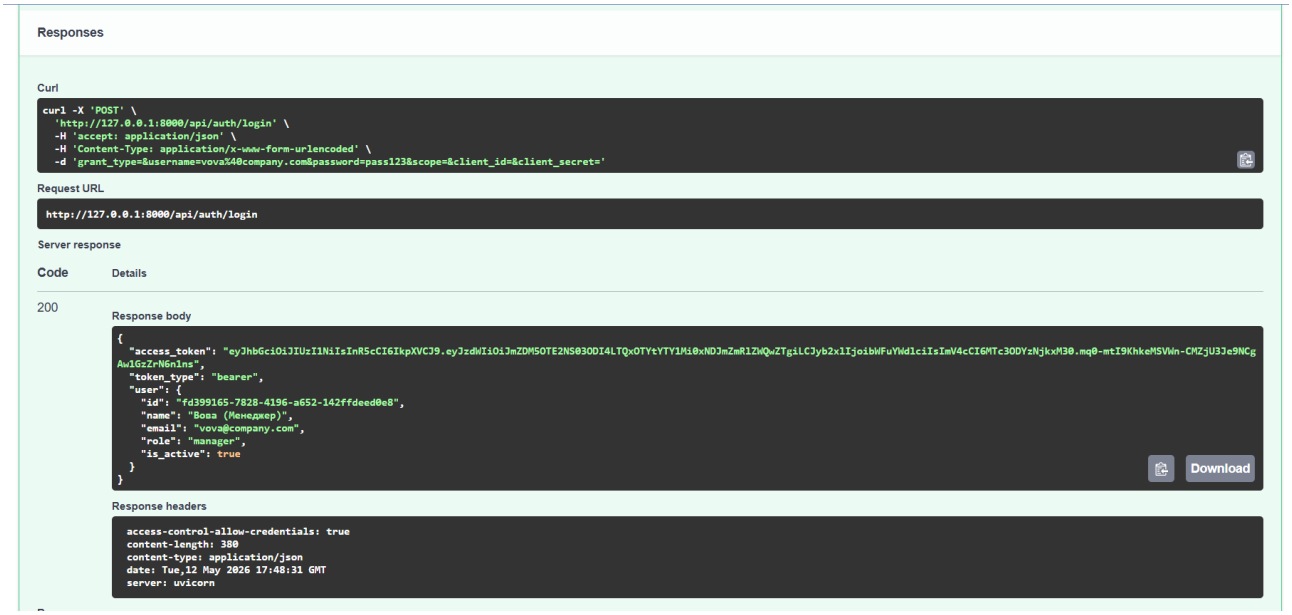


Рис. 4.3 Результат успішної автентифікації через ендпоінт /api/auth/login

Перевірка системи рольового розмежування доступу проводилася через виклик адміністративних ендпоінтів від імені звичайного співробітника. Здобуття токена для сід-користувача ira@company.com з роллю employee і подальший виклик POST /api/users/ для створення нового профілю повертає HTTP 403 Forbidden з повідомленням «Доступ лише для менеджерів». Залежність require_manager коректно перехоплює запит ще до виконання тіла контролера, аналізуючи поле role декодованого JWT-токена. Повторний виклик того самого маршруту з токеном менеджера успішно створює новий обліковий запис, повертаючи HTTP 201 Created та об'єкт нового користувача.

Окремим напрямком інтеграційного тестування виступила перевірка ендпоінта подання заявок на відгул. Звичайний користувач через POST /api/leaves/

подавав заявку з датами date_from та date_to, що охоплюють завтрашній день, та типом day_off. Сервер створював запис зі статусом pending без негайного списання балансу. Подальший виклик PATCH /api/leaves/{leave_id}/status від імені менеджера зі значенням approved ініціював каскадну транзакцію: інкрементація поля used_days у таблиці day_off_balance, створення запису у журналі аудиту та потенційне перепризначення конфліктних чергувань. Перевірка стану бази даних після затвердження підтвердила атомарність операції - усі три зміни були зафіксовані єдиною транзакцією, а відсутність будь-якої з них в попередніх тестових прогонах призводила до автоматичного відкату.

Для верифікації коректності евристичного алгоритму використано підхід на основі контрольних інваріантів: незалежно від конкретного розподілу імен по комірках, згенерована матриця має задовольняти певний набір математичних властивостей. Перший інваріант стосується дотримання жорстких обмежень. Після виклику POST /api/schedules/generate від імені менеджера система генерує розклад на 30 днів уперед. Запит до бази даних виконано через SQL-агрегацію для підрахунку кількості зайнятих слотів у вихідні дні протягом останніх 18 діб для кожного співробітника. Очікуваний результат - жоден працівник не отримує другої вихідної зміни в межах вікна 18 днів. Перевірка згенерованих матриць у серії з десяти послідовних прогонів підтвердила дотримання цього правила в усіх випадках, що засвідчує коректність роботи штрафу too_soon_penalty у функції сортування weekend_key.

Другий інваріант перевіряє відсутність подвійних призначень. Для кожної пари (user_id, shift_date) в таблиці schedules має існувати не більше одного запису. Цей факт підтверджено агрегаційним запитом SELECT user_id, shift_date, COUNT*()* з умовою HAVING COUNT*()* > 1, який повернув порожній результат. Спроба ручного впровадження дубліката через ендпоінт обміну змінами POST /api/schedules/swap коректно блокується контролером з HTTP-статусом 400 та повідомленням «Заборонено 2 зміни в день!», що підтверджує наявність валідації на прикладному рівні. Третій інваріант оцінює якість балансування навантаження.

Для кожного з шести сід-співробітників обчислено сумарну кількість призначень за період планування. Контрольні прогони засвідчили, що при наявності повного пулу доступних працівників різниця між найбільш та найменш завантаженим співробітником не перевищує одного чергування. У випадку, коли частина команди перебуває у затвердженій відпустці, навантаження автоматично перерозподіляється на доступних співробітників з відповідним зростанням дисбалансу, що є очікуваним наслідком зменшення пулу.

Окремий клас тестів охоплює сценарій ручного коригування розкладу. Симуляція ситуації екстреного вихідного відтворювалася через створення затвердженої заявки на відгул у середині згенерованого періоду. Викликом функції regenerate_future система мала перерахувати залишок графіка зі збереженням історичних записів. Перевірка стану бази даних до точки regen_start підтвердила непорушність попередніх записів, тоді як усі майбутні чергування були перерозподілені з урахуванням нової недоступності співробітника. Тестування механізму обміну змінами через POST /api/schedules/swap проводилося у двох сценаріях. У першому випадку оброблялася симетрична перестановка двох будніх змін між двома працівниками. У другому сценарії відбувався обмін вихідною зміною з подальшим автоматичним балансуванням буднього навантаження через функцію rebalance_weekdays. Алгоритм коректно знаходив найближчу будню зміну отримувача в межах поточного тижня і передавав її ініціатору обміну.

Перевірка модуля експорту календарних даних здійснювалася через ендпоінт GET /api/schedules/export/me. Згенерований файл імпортовано до Microsoft Outlook та Google Calendar для перевірки сумісності. Обидва клієнти коректно розпарсили структуру VCALENDAR з вкладеними подіями VEVENT, відобразивши призначені чергування у відповідних часових інтервалах. Події відпусток, помічені директивою X-MICROSOFT-CDO-BUSYSTATUS:OOF, у середовищі Outlook автоматично переходять у статус «Поза офісом». Результат імпорту згенерованого .ics-файлу до календарного клієнта Microsoft Outlook продемонстровано на рис. 4.4.

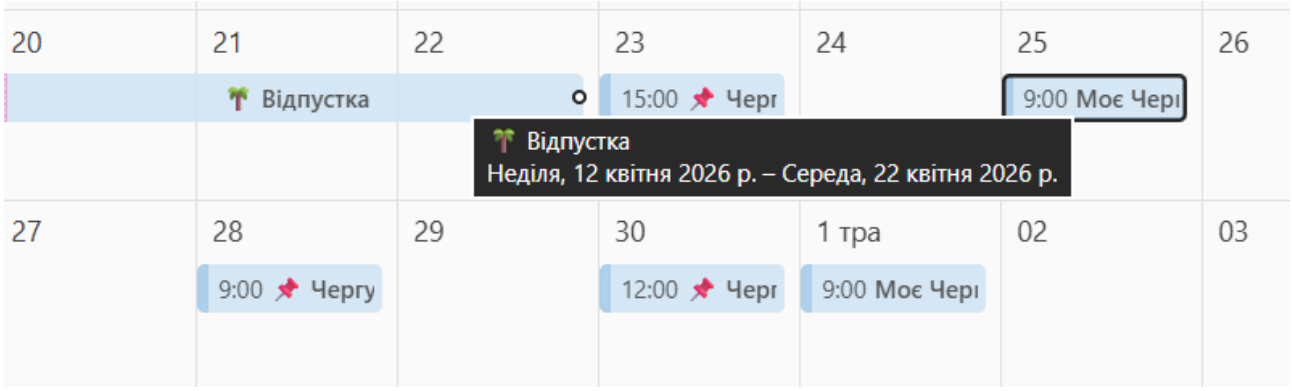


Рис. 4.4 Відображення згенерованого .ics-файлу в Microsoft Outlook

4.3 Оцінка ефективності розробленого програмного забезпечення

Перевірка клієнтської частини Vue.js проводилася в браузерах Google Chrome, Mozilla Firefox та Microsoft Edge актуальних версій. Сценарій авторизації перевірявся з обох ролей. Випадаючий список з іменами співробітників коректно завантажує дані з ендпоінта GET /api/users/. Після введення пароля та натискання кнопки входу клієнт виконує POST-запит з даними у форматі FormData, отримує JWT-токен і автоматично записує його у глобальні заголовки Axios через axios.defaults.headers.common['Authorization']. Подальші запити, виконані тим самим клієнтом, містять цифровий підпис без необхідності повторного вводу облікових даних. Реактивність відображення матриці розкладу перевірена через ручне моделювання сценарію обміну змінами. Після успішного PATCH-запиту до сервера фронтенд виконував повторне завантаження публічних даних через функцію loadPublicData(), що призводить до синхронізації локального стану з актуальними даними бази. Час оновлення інтерфейсу від моменту натискання кнопки до візуального оновлення комірок таблиці не перевищував 500 мілісекунд при роботі з локальним сервером.

Підсумкова оцінка ефективності системи проведена за двома групами критеріїв: технічні метрики продуктивності та функціональні показники впливу на бізнес-процеси кадрового планування. Усереднений час обробки REST API-запитів виміряно через вбудоване логування ASGI-сервера Uvicorn. Прості запити на

отримання списку користувачів через GET /api/users/ оброблялися за час менше 50 мілісекунд при роботі з тестовим набором даних. Запит на отримання повного списку розкладів через GET /api/schedules/ з завантаженням пов'язаних об'єктів через selectinload повертав відповідь за 80-120 мілісекунд. Найбільш ресурсомістка операція - генерація нового розкладу через POST /api/schedules/generate - займала від 800 мілісекунд до 1.5 секунди. Усі показники задовольняють вимогу нефункціонального обмеження часу відгуку.

Функціональна ефективність системи оцінювалася через порівняльний аналіз часових витрат менеджера на типові операції планування. Складання місячного графіка чергувань команди з шести осіб у середовищі електронної таблиці традиційно займає 2-3 години роботи з урахуванням ручної перевірки всіх обмежень та потенційних коригувань. Розроблена система виконує цю задачу за один HTTP-запит з часом обробки до 1.5 секунди, що означає скорочення часових витрат на декілька порядків. Подальше супроводження графіка - обробка раптових лікарняних та запитів на відгули - у ручному режимі вимагає 15-30 хвилин на кожен інцидент через необхідність перерахунку залежних показників. Автоматизоване перепризначення через каскадну транзакцію затвердження відгулу виконується миттєво без втручання адміністратора.

Транзакційна цілісність обліку компенсаційних відгулів верифікована через серію стрес-тестів. Послідовність із 50 операцій виконувалася в різних комбінаціях. Підсумковий стан таблиці day_off_balance після завершення всіх операцій точно відповідав очікуваному значенню, обчисленому незалежно через прямий перерахунок. Жодного випадку розсинхронізації балансу не зафіксовано, що підтверджує коректність використання механізмів ACID-транзакцій PostgreSQL та обгортки db.rollback() у блоках обробки винятків.

Запуск повного набору автоматизованих тестів виконано через команду pytest у директорії бекенду. Усі чотири модульні тести (test_security_password_hashing, test_jwt_token_lifecycle, test_pydantic_schema_validation та test_outlook_service_graceful_degradation)

завершилися успішно з результатом «4 passed», що підтверджує коректність роботи криптографічного модуля, механізму сесійних токенів, валідаційного шару Pydantic та підсистеми стійкості до зовнішніх збоїв. Сукупність проведених перевірок підтвердила, що розроблене програмне забезпечення відповідає сформульованим функціональним та нефункціональним вимогам. Модульне тестування засвідчило коректність роботи криптографічних функцій та механізмів валідації даних. Інтеграційні сценарії підтвердили правильність обробки повного циклу запитів від клієнта до бази даних, включаючи каскадні транзакції затвердження відгулів та автоматичного перепризначення конфліктних чергувань.

Функціональне тестування евристичного алгоритму через перевірку математичних інваріантів довело, що згенерована матриця чергувань завжди задовольняє жорсткі обмеження предметної області. Кількісна оцінка ефективності продемонструвала скорочення часових витрат менеджменту на складання графіків на декілька порядків. Отримані результати підтверджують досягнення мети кваліфікаційної роботи та доводять практичну цінність розробленого програмного забезпечення для автоматизації операційного управління трудовими ресурсами підприємства.

ВИСНОВКИ

У межах кваліфікаційної роботи розроблено програмне забезпечення «Shift Scheduler», призначене для автоматизації роботи з графіком чергувань персоналу підприємства. Створена система дає розв'язок задачі комбінаторної оптимізації при розподілі робочих змін з урахуванням бізнес-правил підприємства. Поставлена мета роботи досягнута. Всі сформульовані функціональні та нефункціональні вимоги реалізовані у вигляді працездатного веб-застосунку.

У ході виконання роботи проведено аналіз предметної області операційного управління трудовими ресурсами. Встановлено, що задача належить до класу NP-складних проблем дискретної оптимізації. Проаналізовано три ключові парадигми ринкових рішень і жодна з існуючих категорій не задовольняє одночасно вимоги бізнес-логіки і транзакційної надійності даних. На основі проведеного аналізу обґрунтовано необхідність розробки кастомного рішення.

На етапі проєктування здійснено формалізацію бізнес-правил через розділення обмежень на два класи. Спроєктовано архітектуру програмної системи і розроблено реляційну схему бази даних та двопрохідний евристичний алгоритм генерації розкладу.

Програмну реалізацію виконано на технологічному стеку, обраному з урахуванням специфіки обчислювальних завдань. Серверну частину побудовано на асинхронному фреймворку FastAPI з ORM-бібліотекою SQLAlchemy 2.0 та JWT-токенами для авторизації. Криптографічний захист паролів реалізовано через алгоритм bcrypt. Клієнтську частину розроблено на Vue.js версії 3 з парадигмою Composition API. Реалізовано систему інтеграції з корпоративними календарями через автономну генерацію файлів стандарту RFC 5545.

На етапі тестування проведено верифікацію розробленої системи на трьох рівнях. Модульні тести, реалізовані з використанням фреймворку pytest. Було перевірено криптографічний модуль, механізм генерації та декодування JWT-токенів, валідації Pydantic та підсистему стійкості. Інтеграційне тестування через

документацію OpenAPI підтвердило правильність обробки повного циклу HTTP-запитів від клієнта до бази даних. Функціональне тестування алгоритму генерації через перевірку математичних інваріантів довело, що згенерована матриця чергувань задовольняє жорсткі обмеження предметної області. Кількісна оцінка ефективності розробленого програмного забезпечення продемонструвала суттєві переваги у порівнянні з традиційними методами ручного планування. Розроблена система забезпечує скорочення часу формування графіка чергувань з 2–3 годин до 1,5 секунди, зменшення кількості помилок та підвищення прозорості процесів управління чергуваннями.

Практична цінність роботи полягає у створенні готового до впровадження програмного продукту, який може бути адаптований до потреб підприємств різних галузей, де актуальною є задача розподілу робочих змін з урахуванням нормативних обмежень. Контейнеризація через Docker дає змогу для розгортання у будь-якому виробничому середовищі.

Перспективи подальшого розвитку системи включають інтеграцію алгоритмів машинного навчання, для прогнозування пікових навантажень та проактивного коригування розкладу, реалізацію лікарняних і підключення до корпоративної системи для автоматичного оновлення календарів співробітників. Закладена архітектурна основа дозволяє нарощувати функціональність системи без переписування ключових компонентів обчислювального ядра.

Таким чином, поставлена у кваліфікаційній роботі мета щодо з'ясування ефективності застосування алгоритмів автоматизованого балансування робочого навантаження та мінімізації людського фактору в кадровому менеджменті досягнута. Розроблене програмне забезпечення підтверджує практичну цінність застосування комбінаторних евристик для розв'язання задач операційного управління трудовими ресурсами та демонструє переваги кастомної розробки над універсальними ринковими рішеннями в умовах специфічних вимог підприємства.

Робота пройшла апробацію. За її результатами було опубліковано наступні тези доповідей:

1. Перната І.В., Золотухіна О.А. Оптимізація графіку чергувань персоналу підприємства. Матеріали ІІ Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. К.: ДУІКТ, 2025. Збірник тез. С.503-504.

2. Перната І.В., Золотухіна О.А. Оптимізація графіку чергувань персоналу підприємства. Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2026. Збірник тез. С. 322-324.

ПЕРЕЛІК ПОСИЛАНЬ

1. Заневич Ю., Кухарський В. Про розв'язування задачі складання розкладу занять, використовуючи генетичний алгоритм. Вісник Львівського університету. Серія прикладна математика та інформатика. 2019. Вип. 27. С. 133–143. URL: http://publications.lnu.edu.ua/bulletins/index.php/ami/article/view/10725.

2. Косенко Н. В., Кравченко І. М. Цифровізація бізнес-процесів менеджменту персоналу. Галицький економічний вісник Тернопільського національного технічного університету. 2022. С. 91–101. URL: https://galicianvisnyk.tntu.edu.ua/pdf/74/1046.pdf.

3. Снитюк В. Є., Сіпко О. М. Технологія еволюційного формування розкладів у закладах вищої освіти : монографія. Київ : Видавець ФОП Піча Ю. В., 2022. 136 с. ISBN 978-966-969-152-1.

4. Кормен Т., Лейзерсон Ч., Рівест Р., Стайн К. Вступ до алгоритмів / пер. з англ. Київ : К.І.С., 2019. 1288 с. ISBN 978-617-684-239-2.

5. Жданова О. Г., Коваленко В. В. Задача складання розкладу виконання робіт з урахуванням їхніх часових вікон. Вісник Вінницького політехнічного інституту. 2023. № 2. С. 97–100. URL: https://visnyk.vntu.edu.ua/index.php/visnyk/article/view/2869.

6. Павлов О. А., Жданова О. Г., Сперкач М. О. Задача складання розкладу виконання завдань паралельними приладами з метою мінімізації максимуму відхилення від директивного терміну моментів завершення приладами усіх завдань. Вісник НТУУ "КПІ" 2014. С. 148–158. URL: http://dspace.nbuv.gov.ua/handle/123456789/86427.

7. Axios HTTP Client Documentation. Axios Authors. URL: https://axios-http.com/docs/intro.

8. Docker Documentation. Docker Inc. URL: https://docs.docker.com.

9. Deputy: Employee Scheduling Software & Time Tracking App. Deputy. URL: https://www.deputy.com/.

10. Codd E. F. A Relational Model of Data for Large Shared Data Banks. Communications of the ACM. 1970. Vol. 13, No. 6. P. 377–387. DOI: https://doi.org/10.1145/362384.362685.

11. Ernst A. T., Jiang H., Krishnamoorthy M., Sier D. Staff scheduling and rostering: A review of applications, methods and models. European Journal of Operational Research. 2004. Vol. 153, No. 1. P. 3–27. DOI: https://doi.org/10.1016/S0377-2217(03)00095-X.

12. FastAPI Documentation. FastAPI. URL: https://fastapi.tiangolo.com/.

13. Fowler M. Patterns of Enterprise Application Architecture. Boston : Addison-Wesley Professional, 2002. 533 p. ISBN 978-0-321-12742-6.

14. Hardt D. The OAuth 2.0 Authorization Framework : RFC 6749. Internet Engineering Task Force (IETF). 2012. URL: https://datatracker.ietf.org/doc/html/rfc6749.

15. Internet Calendaring and Scheduling Core Object Specification (iCalendar) : RFC 5545 / B. Desruisseaux. Internet Engineering Task Force (IETF). 2009. URL: https://datatracker.ietf.org/doc/html/rfc5545.

16. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT) : RFC 7519. Internet Engineering Task Force (IETF). 2015. URL: https://datatracker.ietf.org/doc/html/rfc7519.

17. Karp R. M. Reducibility among Combinatorial Problems. Complexity of Computer Computations. Boston, MA : Springer, 1972. P. 85–103. DOI: https://doi.org/10.1007/978-1-4684-2001-2_9.

18. Lubanovic B. FastAPI: Modern Python Web Development. Sebastopol : O'Reilly Media, 2023. 432 p. ISBN 978-1-098-13550-8.

19. Microsoft Excel Spreadsheet Software. Microsoft. URL: https://www.microsoft.com/en-us/microsoft-365/excel.

20. Microsoft Graph API Documentation. Microsoft Corporation. URL: https://learn.microsoft.com/en-us/graph/overview.

21. OWASP Top 10 – 2021. The OWASP Foundation. URL: https://owasp.org/Top10/.

22. PostgreSQL: The World's Most Advanced Open Source Relational Database. PostgreSQL Global Development Group. URL: https://www.postgresql.org/docs/.

23. Pydantic V2 Documentation. Pydantic Services Inc. URL: https://docs.pydantic.dev/latest/.

24. pytest Documentation. Holger Krekel and pytest-dev team. URL: https://docs.pytest.org/en/stable/.

25. Python 3.12.0 Documentation. Python Software Foundation. URL: https://docs.python.org/3.12/.

26. SAP SuccessFactors Employee Central. SAP. URL: https://www.sap.com/products/hcm/core-hr-and-payroll/employee-central.html.

27. SQLAlchemy 2.0 Documentation. SQLAlchemy Authors. URL: https://docs.sqlalchemy.org/en/20/.

28. Vue.js: The Progressive JavaScript Framework. Vue.js. URL: https://vuejs.org/guide/introduction.html.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка програмного забезпечення для автоматизації роботи з графіком чергувань персоналу підприємства

Виконала студентка 4 курсу
групи ПД-41
Перната Ірина Валеріївна
Керівник роботи
канд.техн.наук , доц. Оксана ЗОЛОТУХІНА

Київ – 2026

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи – підвищення ефективності роботи з графіком чергувань персоналу підприємства шляхом автоматизації формування та коригування графіка чергувань.

Об'єкт дослідження – процеси роботи з графіком чергувань персоналу підприємства .

Предмет дослідження – алгоритми, методи та програмні засоби автоматизації роботи з графіком чергувань персоналу підприємства .

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Провести аналіз особливостей організації робочого часу, формування графіків чергувань та балансування навантаження персоналу підприємства .
2. Провести огляд та аналіз існуючих програмних рішень для роботи з графіками чергувань персоналу, визначити їх переваги та недоліки .
3. Сформулювати функціональні і нефункціональні вимоги до розроблюваного web-застосунку на основі аналізу предметної галузі та операційних потреб підприємства .
4. Провести огляд та аналіз технологій та інструментів, що можуть бути використані для розробки web-застосунку.
5. Спроєктувати та розробити web-застосунок з використанням обраних технологій, включно з евристичним алгоритмом генерації розкладу.
6. Провести тестування розробленого web-застосунку для виявлення та усунення помилок.

АНАЛІЗ АНАЛОГІВ

Характеристика	**Ручне ведення (MS Excel)**	**Хмарні SaaS-системи**	**Корпоративні ERP**	**СистемаShift Scheduler**
Автоматична генерація графіку чергувань	Відсутня	Базова	Присутня	**Присутня**
Врахування специфічних бізнес-правил	Повна, але неконтрольована	Низька, жорстко задані рамки	Середня, вимагає доопрацювання коду	**Повна**
Автоматичний облік балансу компенсаційних відгулів	Відсутній	Присутній у преміум-модулях	Присутній	**Вбудований спеціалізований модуль**
Делегованийобмін черговання миіз валідацією	Відсутній	Присутній	Залежить від конфігурації	**Присутній**
Інтеграція із зовнішніми календарями	Відсутня	Обмежена бізнес правилами	Потребує налаштування шлюзів	**Вбудована функція**

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги

1. Автоматична генерація розкладу чергувань на основі заданих бізнес-правил та обмежень.
2. Керування обліковими записами користувачів та розмежування прав доступу.
3. Обробка та верифікація заявок на відпустки, відгули та обмін чергуваннями.
4. Автоматичний розрахунок та моніторинг балансу компенсаційних вихідних днів працівників.
5. Експорт персональних графіків у стандартизований формат iCalendar (.ics) для синхронізації з Outlook.
6. Фіксація та зберігання історії адміністративних дій у журналі аудиту.

5

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Нефункціональні вимоги

1. Дотримання мінімальних часових інтервалів між змінами при автоматичному плануванні.
2. Час обробки стандартних REST API запитів не повинен перевищувати 200 мілісекунд.
3. Сесія користувача захищена JWT-токеном із часом життя 480 хвилин.
4. Паролі хешуються за допомогою криптографічного алгоритму Bcrypt.
5. Система повинна застосовувати ліміти на завантаження даних для мінімізації навантаження на мережу.
6. Система повинна бути захищена від SQL-ін'єкцій шляхом використання технології об’єктно-реляційного відображення.
7. База даних повинна гарантувати принципи ACID.
8. Система повинна генерувати файли розкладу у стандартизованому форматі RFC 5545 (.ics) для забезпечення сумісності з корпоративними та персональними календарями.

6

ЗАСОБИ РОЗРОБКИ

Frontend

Vue.js 3

axios

Backend

PostgreSQL

FastAPI

Python 3.12

Безпека, Інтеграція та Інфраструктура

SQLAlchemy

git

GitHub

BCrypt HASH

RFC 5545 (ics)

Docker

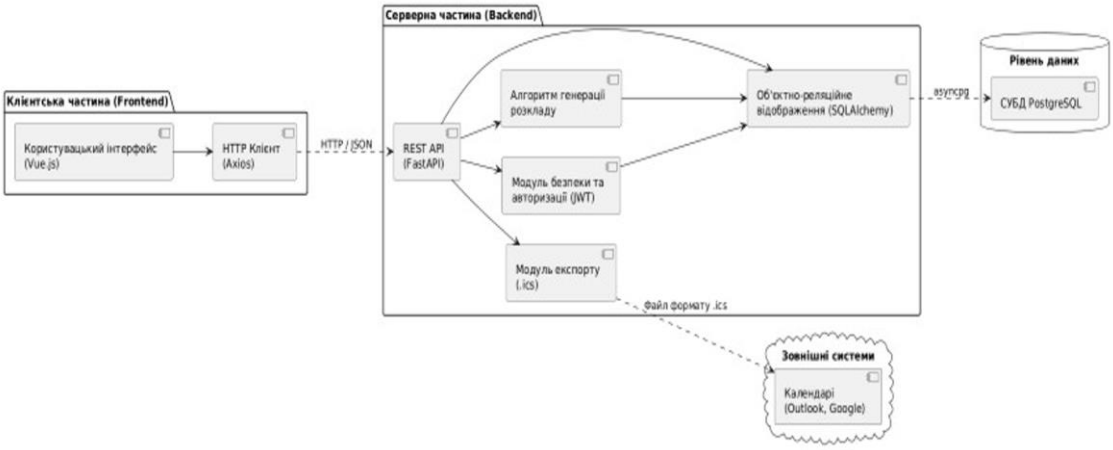
JWT

7

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ

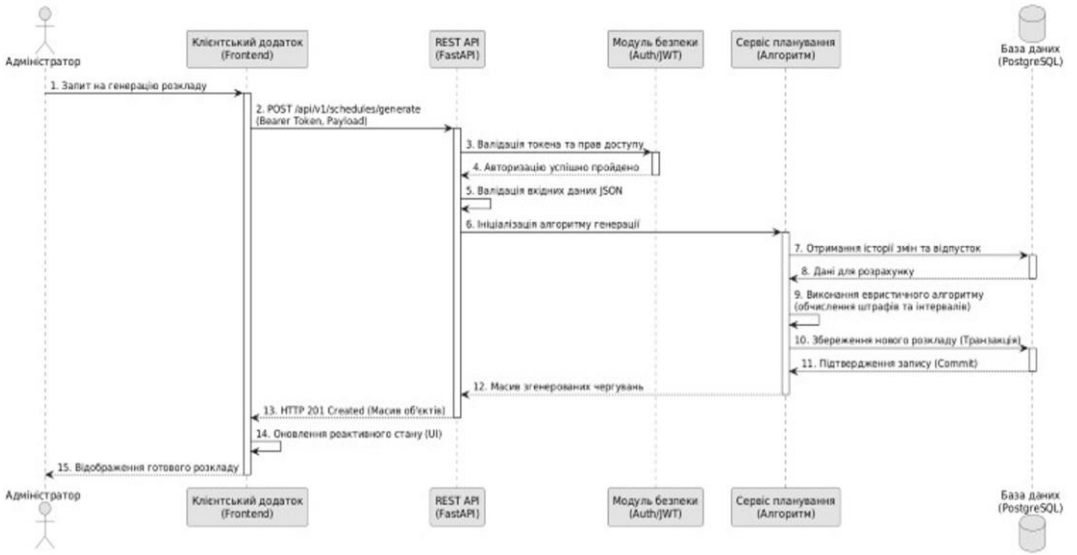
8

АРХІТЕКТУРА СИСТЕМИ



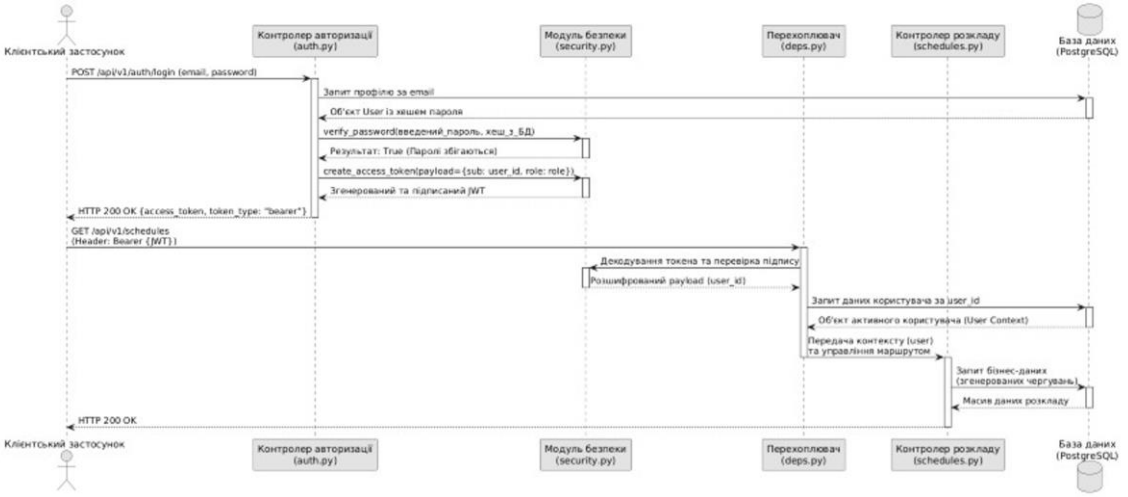
9

ДІАГРАМА ПОСЛІДОВНОСТІ ПРОЦЕСУ ГЕНЕРАЦІЇ РОЗКЛАДУ



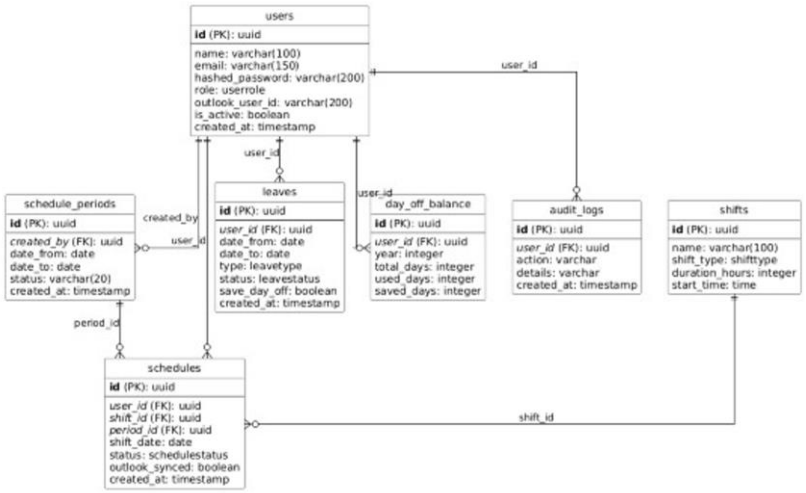
10

ДІАГРАМА ПОСЛІДОВНОСТІ ПРОЦЕСУ АВТЕНТИФІКАЦІЇ ТА ВАЛІДАЦІЇ JWT-ТОКЕНА



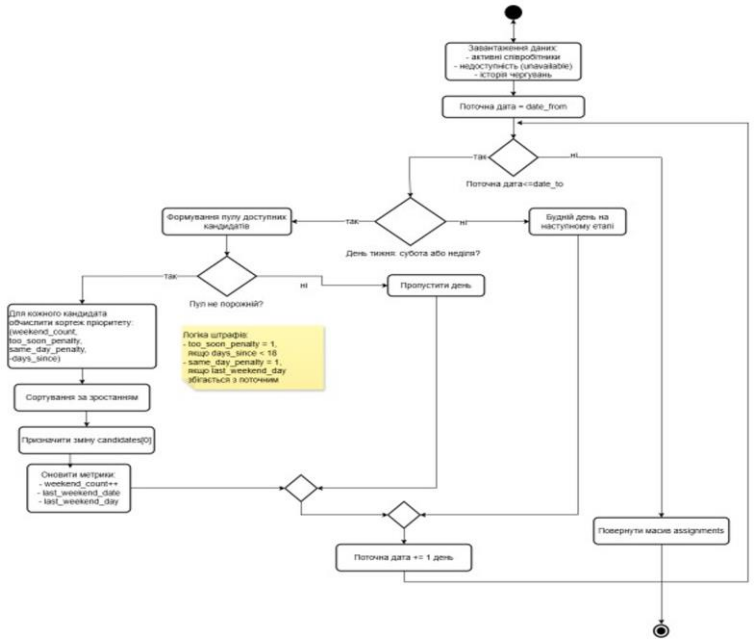
11

СТРУКТУРА БАЗИ ДАНИХ



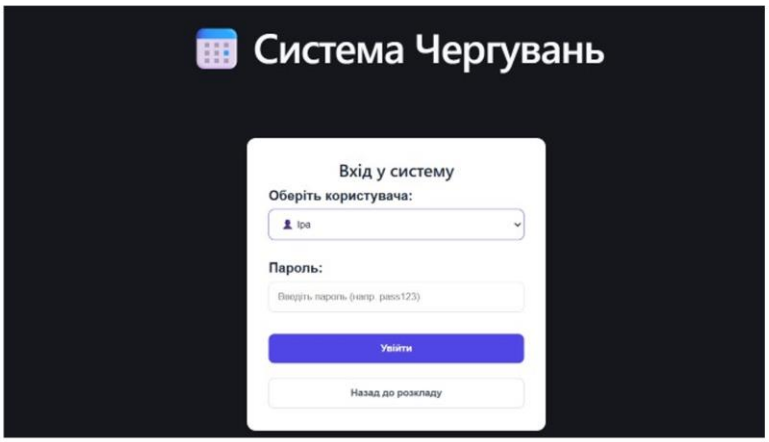
12

АЛГОРИТМ ГЕНЕРАЦІЇ РОЗКЛАДУ НА ВИХІДНІ ДНІ



13

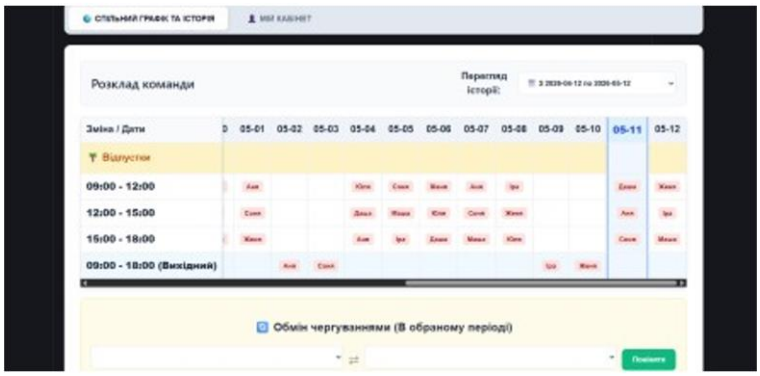
ЕКРАННІ ФОРМИ



Екран автентифікації користувача

14

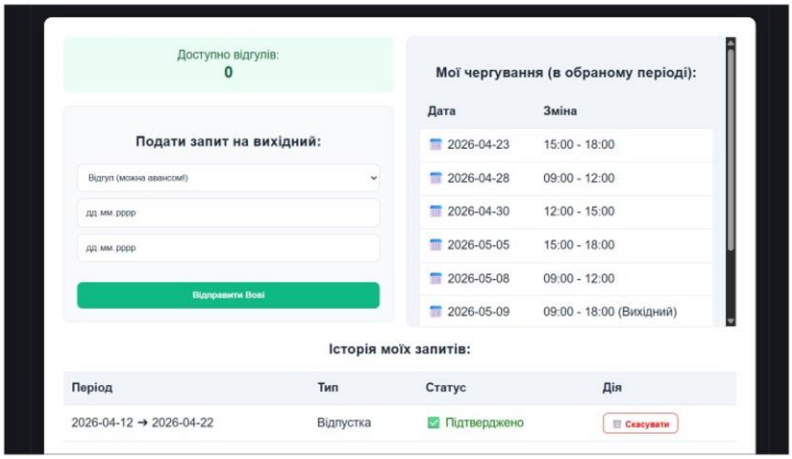
ЕКРАННІ ФОРМИ



Візуальне відображення згенерованої матриці розкладу

15

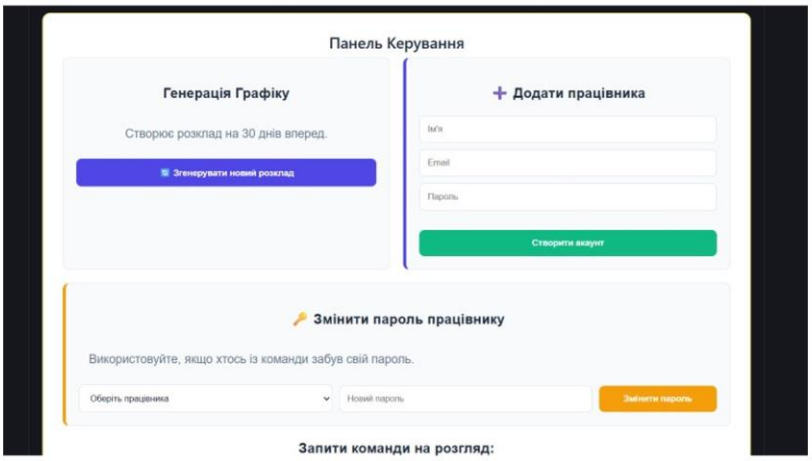
ЕКРАННІ ФОРМИ



Особистий кабінет працівника

16

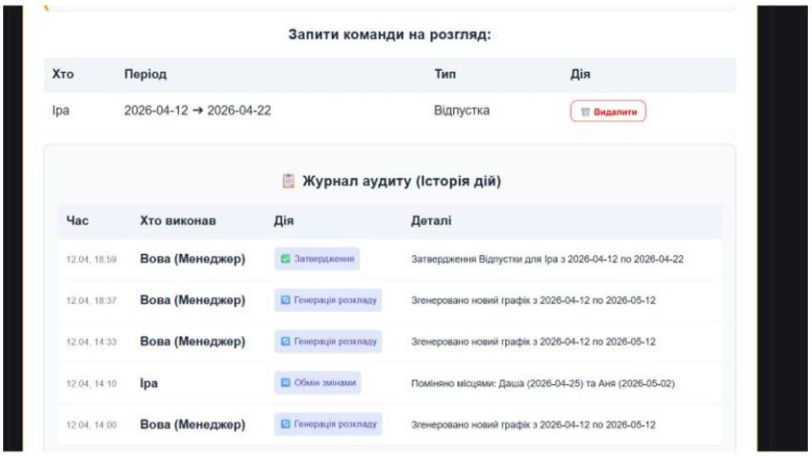
ЕКРАННІ ФОРМИ



Панель керування менеджера

17

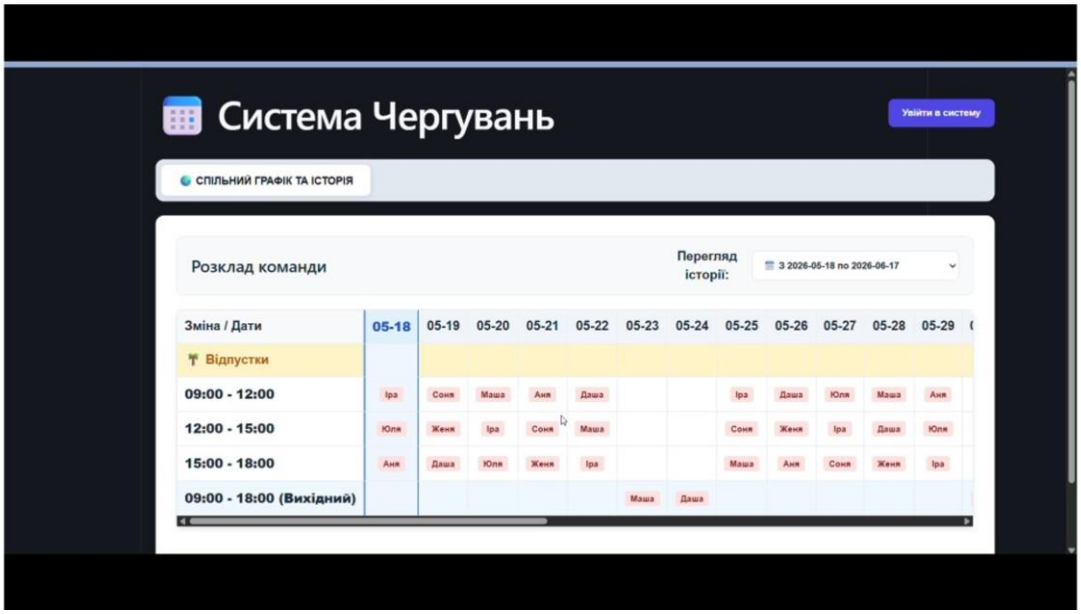
ЕКРАННІ ФОРМИ



Інтерфейс запитів команди і журнал аудиту на сторінці менеджера

18

ВІДЕОМАТЕРІАЛИ



19

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Перната І.В., Золотухіна О.А. Оптимізація графіку чергувань персоналу підприємства . Матеріали II Всеукраїнської науково -технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, Державний університет інформаційно -комунікаційних технологій . К.: ДУІКТ, 2025. Збірник тез. С.503-504.
2. Перната І.В., Золотухіна О.А. Оптимізація графіку чергувань персоналу підприємства . Матеріали VII Всеукраїнської науково -технічної конференції «Застосування програмного забезпечення в інформаційно -комунікаційних технологіях», 23 квітня 2026 року, Київ, Державний університет інформаційно - комунікаційних технологій . Збірник тез. К.: ДУІКТ, 2026. Збірник тез. С. 322-324.

20

ВИСНОВКИ

1. Проведено аналіз особливостей організації робочого часу, формування графіків чергувань та балансування навантаження персоналу підприємства. Визначено основні проблеми ручного формування графіків, зокрема значні часові витрати, ризик виникнення конфліктних призначень, складність оперативного внесення змін та необхідність забезпечення рівномірного розподілу навантаження між співробітниками.
2. Проведено огляд та аналіз існуючих програмних рішень для роботи з графіками чергувань персоналу. Визначено переваги та недоліки табличних процесорів, ERP-систем і SaaS-платформ.
3. Сформульовано функціональні та нефункціональні вимоги до web-застосунку для автоматизації роботи з графіком чергувань персоналу підприємства.
4. Проведено огляд та аналіз технологій та інструментів для розробки web-застосунку. Для серверної частини обрано мову програмування Python та фреймворк FastAPI, для клієнтської частини – Vue.js, для роботи з базою даних – SQLAlchemy та PostgreSQL. Визначено доцільність використання Microsoft Graph API та стандарту RFC 5545 для інтеграції із зовнішніми календарними системами.
5. Спроектовано та розроблено web-застосунок для автоматизації роботи з графіком чергувань персоналу підприємства. Реалізовано евристичний алгоритм генерації розкладу, механізми автоматизованого перепризначення чергувань, сповіщення співробітників, погодження відпусток та вихідних, експорт календарів і збереження історії адміністративних дій.
6. Проведено тестування розробленого web-застосунку. У результаті тестування підтверджено коректність роботи механізмів формування графіків, обробки змін, контролю конфліктних призначень, синхронізації календарів та системи сповіщень. Розроблена система забезпечує скорочення часу формування графіка чергувань з 2–3 годин до 1,5 секунди, зменшення кількості помилок та підвищення прозорості процесів управління чергуваннями, за рахунок усунення людського фактору при формуванні графіку і логування всіх дій користувачів.

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

```
@router.get("/audit-logs")
async def get_audit_logs(db: AsyncSession = Depends(get_db), current_user: User = Depends(require_manager)):
    result = await db.execute(
select(AuditLog).options(selectinload(AuditLog.user)).order_by(AuditLog.created_at.desc()).limit(50))
    return result.scalars().all()
@router.post("/generate", status_code=201)
async def create_schedule(db: AsyncSession = Depends(get_db), current_user: User = Depends(require_manager)):
    date_from = date.today()
    date_to = date_from + timedelta(days=30)
    old_schedules_res = await db.execute(
select(Schedule).options(selectinload(Schedule.shift)).where(Schedule.shift_date >= date_from))
    for sch in old_schedules_res.scalars().all():
        if sch.shift and sch.shift.shift_type == ShiftType.weekend:
            bal_res = await db.execute(select(DayOffBalance).where(DayOffBalance.user_id == sch.user_id,
                                                    DayOffBalance.year == sch.shift_date.year))
            bal = bal_res.scalar_one_or_none()
            if bal and bal.saved_days > 0: bal.saved_days -= 1
        await db.delete(sch)
    await db.commit()
    await db.execute(delete(SchedulePeriod).where(SchedulePeriod.date_from >= date_from))
    await db.commit()
    period = SchedulePeriod(date_from=date_from, date_to=date_to, created_by=current_user.id)
    db.add(period)
    await db.commit()
    await db.refresh(period)
    assignments = await generate_schedule(db, date_from, date_to, period.id, current_user.id)
    if not assignments: raise HTTPException(status_code=400, detail="Немає доступних співробітників.")
    schedules = [Schedule(**assign) for assign in assignments]
    db.add_all(schedules)
    db.add(AuditLog(user_id=current_user.id, action="Генерація розкладу",
                    details=f"Згенеровано новий графік з {date_from} по {date_to}"))
    await db.commit()
    from app.services.outlook import create_outlook_event
    shifts_res = await db.execute(select(Shift))
    all_shifts = {s.id: s for s in shifts_res.scalars().all()}
    users_res = await db.execute(select(User))
    all_users = {u.id: u for u in users_res.scalars().all()}
    local_bals = {}
    for sch in schedules:
        user = all_users.get(sch.user_id)
        shift = all_shifts.get(sch.shift_id)
        if shift and shift.shift_type == ShiftType.weekend:
            year = sch.shift_date.year
            key = f"{sch.user_id}_{year}"
            if key not in local_bals:
                bal_res = await db.execute(

select(DayOffBalance).where(DayOffBalance.user_id == sch.user_id, DayOffBalance.year == year))
                bal = bal_res.scalar_one_or_none()
                if not bal:
                    bal = DayOffBalance(user_id=sch.user_id, year=year, saved_days=0, used_days=0)
                    db.add(bal)
                local_bals[key] = bal
            local_bals[key].saved_days += 1
        if user and shift and shift.start_time:
            start_dt = datetime.combine(sch.shift_date, shift.start_time)
            end_dt = start_dt + timedelta(hours=shift.duration_hours)
            success = await create_outlook_event(
                user.email,
                start_dt.strftime("%Y-%m-%dT%H:%M:%S"),
                end_dt.strftime("%Y-%m-%dT%H:%M:%S")
            )
            if success: sch.outlook_synced = True
    await db.commit()
    result = await db.execute(select(SchedulePeriod).options(
```

```
selectinload(SchedulePeriod.schedules).selectinload(Schedule.user),
selectinload(SchedulePeriod.schedules).selectinload(Schedule.shift)
    ).where(SchedulePeriod.id == period.id))
    return result.scalar_one()
@router.get("/")
async def get_periods(db: AsyncSession = Depends(get_db)):
    result = await db.execute(

select(SchedulePeriod).options(selectinload(SchedulePeriod.schedules).selectinload(Schedule.user),

selectinload(SchedulePeriod.schedules).selectinload(Schedule.shift)).order_by(
            SchedulePeriod.date_from.desc()))
    return result.scalars().all()
# backend/app/api/schedules.py
@router.get("/export/me", response_class=PlainTextResponse)
async def export_my_schedule_ics(db: AsyncSession = Depends(get_db), current_user: User = Depends(get_current_user)):
    # 1. Завантажуємо чергування працівника
    sched_result = await db.execute(
        select(Schedule).options(selectinload(Schedule.shift))
        .where(Schedule.user_id == current_user.id, Schedule.shift_date >= date.today())
    )
    schedules = sched_result.scalars().all()
    leaves_result = await db.execute(
        select(Leave).where(Leave.user_id == current_user.id, Leave.status == "approved", Leave.date_to >= date.today())
    )
    leaves = leaves_result.scalars().all()
    ics_content = [
        "BEGIN:VCALENDAR",
        "VERSION:2.0",
        "PRODID:-//Shift Scheduler//UA",
        "METHOD:PUBLISH"
    ]
    now = datetime.utcnow().strftime("%Y%m%dT%H%M%SZ")
    for sch in schedules:
        if not sch.shift or not sch.shift.start_time: continue
        start_dt = datetime.combine(sch.shift_date, sch.shift.start_time)
        end_dt = start_dt + timedelta(hours=sch.shift.duration_hours)
        dtstart = start_dt.strftime("%Y%m%dT%H%M%S")
        dtend = end_dt.strftime("%Y%m%dT%H%M%S")
        ics_content.extend([
            "BEGIN:VEVENT",
            f"UID:shift-{sch.id}@shiftscheduler.local",
            f"DTSTAMP:{now}",
            f"DTSTART;TZID=Europe/Kyiv:{dtstart}",
            f"DTEND;TZID=Europe/Kyiv:{dtend}",
            f"SUMMARY: Чергування: {sch.shift.name}",
            f"DESCRIPTION:Тривалість: {sch.shift.duration_hours} год",
            "PRIORITY:5",
            "END:VEVENT"
        ])
    for l in leaves:
        start_str = l.date_from.strftime("%Y%m%d")
        end_str = (l.date_to + timedelta(days=1)).strftime("%Y%m%d")
        summary = " Відпустка" if l.type == "vacation" else " Відгул"
        ics_content.extend([
            "BEGIN:VEVENT",
            f"UID:leave-{l.id}@shiftscheduler.local",
            f"DTSTAMP:{now}",
            f"DTSTART;VALUE=DATE:{start_str}",
            f"DTEND;VALUE=DATE:{end_str}",
            f"SUMMARY:{summary}",
            "X-MICROSOFT-CDO-BUSYSTATUS:OOF",
            "TRANSP:TRANSPARENT",
            "END:VEVENT"
        ])
    ics_content.append("END:VCALENDAR")
    safe_name = current_user.email.split('@')[0]
    filename = f"schedule_{safe_name}.ics"
    return PlainTextResponse(
        "\n".join(ics_content),
        headers={"Content-Disposition": f"attachment; filename={filename}"},
        media_type="text/calendar"
    )
@router.post("/swap", status_code=200)
```

```
async def swap_schedules(data: SwapRequest, db: AsyncSession = Depends(get_db),
                         current_user: User = Depends(get_current_user)):
    sch1 = await db.get(Schedule, data.schedule_id_1,
                        options=[selectinload(Schedule.user), selectinload(Schedule.shift)])
    sch2 = await db.get(Schedule, data.schedule_id_2,
                        options=[selectinload(Schedule.user), selectinload(Schedule.shift)])
    if not sch1 or not sch2: raise HTTPException(status_code=404, detail="Чергування не знайдено")
    if sch1.shift.shift_type != sch2.shift.shift_type: raise HTTPException(status_code=400,

detail="Заборонено! Можна міняти лише будні на будні, а вихідні на вихідні.")
    existing_1 = await db.execute(
        select(Schedule).where(Schedule.user_id == sch1.user_id, Schedule.shift_date == sch2.shift_date,
                               Schedule.id != sch2.id))
    if existing_1.scalar_one_or_none(): raise HTTPException(status_code=400, detail="Заборонено 2 зміни в день!")
    existing_2 = await db.execute(
        select(Schedule).where(Schedule.user_id == sch2.user_id, Schedule.shift_date == sch1.shift_date,
                               Schedule.id != sch1.id))
    if existing_2.scalar_one_or_none(): raise HTTPException(status_code=400, detail="Заборонено 2 зміни в день!")
    old_user_1 = sch1.user_id
    old_user_2 = sch2.user_id
    sch1.user_id = old_user_2
    sch2.user_id = old_user_1
    log = AuditLog(user_id=current_user.id, action=" Обмін змінами",
                   details=f"Поміняно місцями: {sch1.user.name} ({sch1.shift_date}) та {sch2.user.name} ({sch2.shift_date})")
    db.add(log)
    await db.commit()
    if sch1.shift.shift_type == ShiftType.weekend:
        async def rebalance_weekdays(target_date, user_gaining_weekend, user_losing_weekend):
            week_start = target_date - timedelta(days=target_date.weekday())
            week_end = week_start + timedelta(days=4)
            shifts_res = await db.execute(
                select(Schedule).join(Shift).where(
                    and_(Schedule.user_id == user_gaining_weekend, Schedule.shift_date >= week_start,
                         Schedule.shift_date <= week_end, Shift.shift_type == ShiftType.weekday))
            )
            shifts_to_give = shifts_res.scalars().all()
            for s_to_give in shifts_to_give:
                conflict = await db.execute(select(Schedule).where(Schedule.user_id == user_losing_weekend,
                                                                   Schedule.shift_date == s_to_give.shift_date))
                if not conflict.scalar_one_or_none():
                    s_to_give.user_id = user_losing_weekend
                    db.add(s_to_give)
                    await db.commit()
                    return True
        await rebalance_weekdays(sch1.shift_date, user_gaining_weekend=old_user_2, user_losing_weekend=old_user_1)
        await rebalance_weekdays(sch2.shift_date, user_gaining_weekend=old_user_1, user_losing_weekend=old_user_2)
    return {"status": "success"}
```