Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем та програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

Бакалавр
(назва освітнього ступеня)

на тему: Розробка веб-застосунку для пошуку та замовлення послуг на базі Python Django

Виконав(ла): студент(ка) 4 курсу, групи СПз-41
спеціальності 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

		Бабій Р. І.
	(підпис)	(прізвище та ініціали)
Керівник		Пастух О. А.
	(підпис)	(прізвище та ініціали)
Нормоконтроль		Стоянов Ю. М.
	(підпис)	(прізвище та ініціали)
Завідувач кафедри		Петрик М. Р.
	(підпис)	(прізвище та ініціали)
Рецензент		
	(підпис)	(прізвище та ініціали)

Тернопіль 2025

АНОТАЦІЯ

Кваліфікаційна робота бакалавра. Тернопільський національний технічний університет імені Івана Пулюя, кафедра програмної інженерії, спеціальність 121 «Інженерія програмного забезпечення». ТНТУ, 2025, Сторінок 59, таблиць 0, додатків 3, джерел 12, рисунків 11, презентація.

Ключові слова: ВЕБ-ЗАСТОСУНОК, DJANGO, ЗАМОВЛЕННЯ ПОСЛУГ, ІНТЕГРАЦІЯ API, ПОШУК ПОСЛУГ.

Тема: Розробка веб-застосунку для пошуку та замовлення послуг на базі Python, Django.

У даній кваліфікаційні роботі розглядається процес створення веб-застосунку для пошуку та замовлення послуг із використанням сучасних технологій веб-розробки на основі фреймворку Django та мови програмування Python. Метою роботи є розробка зручного та функціонального сервісу, який дозволяє користувачам знаходити необхідні послуги, переглядати інформацію про виконавців, оформляти замовлення, а також взаємодіяти через інтерфейс платформи.

У роботі представлено детальний аналіз предметної області, описано структуру та архітектуру веб-застосунку, процес проєктування моделей бази даних, реалізацію впровадження сторонніх APIs, а також механізми керування залежностями та слідуванням кращим практикам розробки. Особливу увагу приділено питанням безпеки, захисту даних користувачів, а також забезпеченню масштабованості та зручності інтерфейсу.

Результатом роботи є повнофункціональний веб-застосунок, який забезпечує ефективну комунікацію між замовниками та виконавцями послуг, дозволяє управляти профілями, створювати гнучкий графік для виконавців, переглядати замовлення та інтеграцію з Google календарем для покращення досвіду користувачів. Застосунок створений з урахуванням сучасних вимог до написання коду, керування залежностями та архітектури програмного забезпечення.

ABSTRACT

Bachelor's qualification work. Ternopil National Technical University named after Ivan Puluj, Department of Software Engineering, specialty 121 "Software Engineering". TNTU, 2025, 59 pages, 0 tables, 3 appendices, 12 references, 11 figures, presentation.

Keywords: WEB APPLICATION, DJANGO, ORDERING SERVICES, API INTEGRATION, SEARCHING SERVICES.

Topic: Development of a web application for searching and ordering services based on Python, Django.

This qualification work considers the process of creating a web application for searching and ordering services using modern web development technologies based on the Django framework and the Python programming language. The purpose of the work is to develop a convenient and functional service that allows users to find the necessary services, view information about performers, place orders, and also interact through the platform interface.

The work presents a detailed analysis of the subject area, describes the structure and architecture of the web application, the process of designing database models, the implementation of third-party APIs, as well as mechanisms for managing dependencies and following best development practices. Particular attention is paid to security issues, user data protection, as well as ensuring scalability and user-friendliness of the interface.

The result of the work is a fully functional web application that provides effective communication between customers and service providers, allows you to manage profiles, create a flexible schedule for providers, view orders and integrate with Google Calendar to improve user experience. The application is created taking into account modern requirements for writing code, managing dependencies and software architecture.

ЗМІСТ

ВСТУП

В сучасному цифровому середовищі надання та замовлення послуг стрімко переміщується в онлайн простір. Все більше користувачів шукають зручні та швидкі способи знайти потрібного спеціаліста, забронювати послугу або залишити відгук — і все це без необхідності дзвінків по телефону чи особистому відвідування закладів, які спеціалізуються на наданні таких послуг. З іншого боку, виконавцям важливо мати інструмент для управління графіком, локаціями, профілем, а також пошуку нових клієнтів. Тому виникає актуальна потреба у розробці веб-застосунку, який би з'єднував обидві сторони процесу — замовника і виконавця — в єдиній зручній екосистемі.

Метою цієї дипломної роботи є створення веб-застосунку для пошуку та замовлення послуг з використанням мови програмування Python та фреймворку Django. Django обрано через його розвинену структуру, що дозволяє ефективно реалізувати повнофункціональний веб-застосунок зі зручною адміністративною панеллю, підтримкою авторизації, роботою з базами даних, маршрутизацією, формами та шаблонами. Крім того, Django добре підходить для реалізації ролей користувачів, таких як замовники та виконавці, що є критично важливим у даному проєкті.

Застосунок, розроблений у межах цього проєкту, дозволяє замовникам реєструватися на платформі, переглядати список виконавців, ознайомлюватися з їхніми профілями та розкладами, і бронювати послуги через інтерактивну форму. Виконавці ж мають змогу керувати своїм профілем, вказувати доступні локації надання послуг, створювати розклад роботи, синхронізувати події з Google Calendar та переглядати список своїх замовлень. Особливу увагу приділено логіці взаємодії між користувачами: для виконавців реалізовано функціональність створення локацій та графіків, редагування й видалення записів, а також обмеження доступу до даних інших користувачів, що реалізовано за допомогою міксинів авторизації.

Для замовників доступна система бронювання послуг з вибором часу та місця, після чого всі дані зберігаються у відповідній моделі, а за необхідності синхронізуються з зовнішніми сервісами. Також у рамках роботи розглянуто процес організації архітектури веб-застосунку, використання принципів інверсії залежностей та підключення зовнішніх APIs, зокрема Google Calendar, що підвищує зручність роботи виконавців і дозволяє уникнути накладок у розкладі, а також Google Oauth, який надає зручну аутентифікацію у кілька кліків. Окрім цього проведена робота над автоматизацією процесу перевірки коду за допомогою сучасних інструментів для форматування автоматичного форматуванню коду та зручному керуванню залежностями пакетів, необхідних для застосунку, та зручному розгортанню.

Цей застосунок не лише демонструє технічну реалізацію ключових компонентів, але й орієнтований на реальні потреби користувачів, забезпечуючи зручний доступ до послуг у будь-який момент. У роботі проаналізовано аналогічні рішення на ринку, виявлено їхні обмеження та обґрунтовано потребу у розробці гнучкішої та масштабованішої системи. Особливий акцент зроблено на простоті інтерфейсу, безпеці персональних даних та оптимізації швидкодії застосунку.

Таким чином, дана дипломна робота демонструє не лише вміння працювати з сучасними технологіями, а й здатність аналізувати бізнес-потреби, трансформуючи їх у ефективний програмний продукт. Проєкт розроблено самостійно, з урахуванням вимог до якості програмного забезпечення, слідування кращим практикам програмування, масштабованості архітектури та реальних сценаріїв використання у сфері онлайн-сервісів.

1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Огляд конкурентів

У контексті сучасного ринку онлайн-сервісів в Україні існує кілька популярних застосунків, що забезпечують пошук та замовлення послуг. Серед них найбільш популярними є Kabanchik.ua, YouDo та Profi.ua. У цьому розділі зроблено порівняння основних конкурентів з точки зору функціональності, зручності використання та масштабування, що дозволяє виявити як їх сильні та слабкі сторони, а також окреслити вимоги до розробленого веб-застосунку.

Kabanchik.ua – один із лідерів ринку надання послуг в Україні, який спеціалізується на замовленні різного роду послуг від ремонту квартир та приладів до організації різноманітних заходів.

Основні переваги платформи:

- Широкий асортимент послуг: Сайт охоплює велику кількість категорій, що дозволяє користувачеві швидко знайти потрібного спеціаліста чи послугу.
- Система рейтингів та відгуків: Для збільшення довіри користувачів реалізовано функціонал оцінювання якості послуг і комунікації між замовником та виконавцем.
- Мобільний додаток: Наявність мобільного застосунку забезпечує зручний доступ до сервісу з будь-якого пристрою.

Недоліки платформи можуть бути пов’язані з іноді складною системою сортування завдань або велика кількість деталей інтерфейсу та великою кількістю інформації, що ускладнює швидку навігацію у застосунку.

YouDo — ще один популярний онлайн-застосунок для пошуку виконавців послуг.

Особливості платформи включають:

- Інтуїтивний інтерфейс: Зручна навігація та простий процес розміщення завдань, що допомагає використовувати платформу як для замовників, так і для виконавців.
- Реальний час комунікації: Інтеграція онлайн чату дозволяє, пришвидшує комунікацію та допомагає вирішувати питання щодо завдань, уточнювати деталі та координувати графік виконання робіт.
- Гнучка система завдань: Сервіс орієнтований на різні категорії послуг, від побутових ремонтів до професійних, що робить його універсальним для широкого кола осіб.

Серед недоліків можна відзначити деяку нестабільну роботу мобільного застосунку, а також суперечності в алгоритмах розподілу завдань, що іноді можуть призвести до затримок у відгуках від виконавців.

Profi.ua орієнтований на пошук висококваліфікованих фахівців у різних сферах послуг.

Основні сильні сторони цієї платформи:

- Професійність виконавців: Акцент на відборі виконавців за допомогою детального профілю та перевірки навиків сприяє підвищенню якості наданих послуг та досвіду користувачів.
- Оперативність обробки запитів: Система швидко формує замовлення та миттєво надає зворотній зв’язок, що дозволяє зменшити час очікування відповіді.
- Аналітика та звітність: Деякі рішення застосунку забезпечують розширені механізми для аналізу виконання робіт, що сприяє формуванню постійних клієнтських відносин.

Однак, недоліком Profi.ua є обмежена кількість спеціалістів у певних регіонах, що знижує доступність сервісу для користувачів поза великими містами.

Порівняно з конкурентами, розроблюваний веб-застосунок для пошуку та

замовлення послуг має потенціал забезпечити додаткові переваги за рахунок:

- Модульної архітектури: Використання Python та Django дозволяє реалізувати чіткий розподіл функціональних компонентів, що забезпечує легкість масштабування та підтримки застосунку [1, 2].
- Інтеграції з зовнішніми сервісами: Інтеграція з API таких сервісів, як Google Calendar та Google Oauth, для синхронізації графіків виконавців та швидкої аутентифікації у застосунок, покращує досвід користування застосунком [3].
- Підвищення безпеки даних: Реалізація сучасних підходів до захисту даних користувачів дозволить знизити ризики несанкціонованого доступу та зловживань.
- Зручного та інтуїтивно зрозумілого інтерфейсу: Застосунок максимально спрощує інтерфейс, що дозволить легко орієнтуватися як замовникам, так і виконавцям.

Отже, аналіз конкурентів показує, що ринок онлайн-сервісів в Україні має широкий потенціал для впровадження нових технологічних рішень. Розробка веб-застосунку з сучасною архітектурою, інтегрованою комунікацією та високими стандартами безпеки може стати конкурентною перевагою на фоні існуючих платформ, відповідаючи сучасним вимогам користувачів щодо зручності, оперативності та надійності.

1.2 Обґрунтування вибору напрямку дослідження

В сучасних умовах розробки програмного забезпечення важливо вибрати правильні технології, які забезпечують потрібне виконання поставлених задач та відноситись до них як до інструментів, які забезпечують зручність розробки, масштабованість і підтримку сучасних практик та підходів в розробці. У цьому

розділі я обґрунтую вибір напрямку дослідження і ключових технологій, зокрема Python, Django, Dependency Injector Framework, Poetry, Pydantic та системи для автоматичного форматування коду наприклад, Ruff.

Python обрано як основну мову програмування завдяки її високій читабельності, простоті синтаксису та великій кількості бібліотек, що значно прискорює розробку затосунків. Гнучкість цієї мови дозволяє ефективно розробляти як невеликі проєкти, так і масштабовані системи, що відповідають сучасним вимогам замовників та ринку як такого. Велика спільнота забезпечує надійну підтримку під час розробки, численні приклади використання, які легко можна знайти на просторах інтернету, та багато репозитаріїв з кодом які легко найти, що є значною мірою важливим під час розробки та підтримки проєкту в довгостроковій перспективі.

Django – це високорівневий фреймворк для Python, який забезпечує швидкий старт розробки завдяки готовим рішенням для роботи з базами даних, системами аутентифікації, маршрутизації, шаблонізації та інших елементів типових веб-застосунків, це дозволяє концентруватися на впровадженні бізнес-логіки, тому що розробники цього фреймворку подбали про більшість типових проблем, з якими нам довелося б зустрітися під час розробки без наявності такого інструменту. Серед основних переваг Django виділяють:

- Вбудовану систему безпеки, захист від CSRF, XSS, SQL-ін'єкцій.
- Чіткий шаблон проєктування, що сприяє розділенню бізнес-логіки від представлення даних.
- Широкі можливості для масштабування та підтримки великого обсягу запитів завдяки оптимізованій роботі з ORM.

Ці характеристики роблять Django хорошим вибором для проєкту, спрямованого на забезпечення стабільної роботи та високої ефективності обробки запитів користувачів.

Використання Dependency Injector Framework дозволяє впровадити патерн інверсії залежностей, що сприяє створенню правильної архітектури та допомагає

відділити бізнес-логіку, взаємодію з базами даних, інтеграцію з сторонніми APIs та з самим фреймворком [4]. Застосування цього підходу дозволяє:

- Покращити тестованість коду, оскільки окремі компоненти системи можна ізольовано тестувати.
- Спрощувати підтримку та розширення функціоналу за рахунок чітко визначених залежностей між модулями.
- Підвищувати гнучкість системи, дозволяючи легко змінювати або доповнювати окремі компоненти без необхідності внесення змін до всієї бази коду
- Уникати типових помилок під час імпортації різних модулів, що часто призводить до помилки кругового імпорту.
- Забезпечує структуру проєкту навігація у якій інтуїтивно зрозуміла.

Poetry – це сучасний інструмент управління залежностями та пакування бібліотек для Python, який використовується для:

- Забезпечення ізольованого середовища розробки, що знижує ризик конфліктів між різними бібліотеками чи проєктами в цілому.
- Автоматизації вирішення проблем з залежностями версій пакетів, що сприяє створенню середовища, у якому легко додавати нові чи оновлювати існуючі бібліотеки

Poetry дозволяє автоматично керувати всіма залежностями проєкту, що сприяє стабільності та підтримці коду на всіх етапах розробки проєкту.

Pydantic забезпечує ефективну валідацію даних і керування конфігураціями за допомогою анотацій типів Python. Основні переваги його використання:

- Надійність і безпомилковість обробки вхідних даних, що особливо важливо при роботі з API та формами.
- Автоматичне перетворення даних у потрібні типи, що суттєво знижує ризик помилок у роботі системи.
- Зручність в інтеграції з іншими компонентами застосунку, що дозволяє

створювати структуровані та передбачувані об'єкти даних.

- Сприяє підвищенню якості коду та надійності роботи системи, що безпосередньо впливає на підтримуваність застосунку.
- Спрощує процес налагодження коду.

Інструменти для автоматичного форматування коду та підтримки кращих практик та стандартів написання коду, такі як Ruff, Bantit та MyPy.

Ці інструмент забезпечує швидку і ефективну перевірку якості коду, виявляючи потенційні помилки та неточності на ранніх етапах розробки та дозволяють ідентифікувати помилки та в окремих випадках автоматично відформатувати код. Підтримує сучасні стандарти Python, сприяючи підтримці єдиного стилю написання проєкту. Допомагає підтримувати чистоту та зручність читання коду, що важливо для довгострокового супроводу проєкту. Використання MyPy, який автоматично перевіряє анотацію типів та дозволяє мінімізувати ризики появи критичних помилок з типами даних та сприяє постійному вдосконаленню якості програмного застосунку.

Обрані технології та інструменти сприяють розробці якісного, масштабованого та зручного застосунку, який у майбутньому легко підтримувати, а також забезпечувати конкурентноздатність на ринку, так і потреби кінцевих користувачів. Використання Python, Django, Dependency Injector Framework, Poetry, Pydantic та різного роду лінтерів гарантує ефективну організацію процесу розробки, спрощує написання тестів та підтримки коду, що є важливими чинниками успіху проєкту. Цей підхід дозволяє створити застосунок, який відповідає сучасним стандартам розробки програмного забезпечення та забезпечує високий рівень зручності та безпеки для користувачів.

1.3 Формування вимог

Розробка веб-застосунку для пошуку та замовлення послуг вимагає комплексного підходу до визначення вимог. На цьому етапі здійснюється аналіз

бізнес-логіки, ідентифікація потреб кінцевих користувачів, визначення основних функціональних та інших характеристик системи, а також специфікація інтеграційних точок із сторонніми сервісами. Формування вимог включає декілька ключових моментів.

Функціональні вимоги описують основні можливості системи та взаємодії користувачів з платформою:

- Аутентифікація та авторизація: Реєстрація нових користувачів з розподілом ролей замовник та виконавець. Механізми входу та виходу з застосунку. Підтримка відновлення паролю та верифікація облікового запису.
- Керування профілем користувача: Можливість редагування особистих даних, налаштування сповіщень.
- Окремий функціонал для виконавців: управління графіком роботи, інформацією про послуги та локацією де надається послуга.
- Замовлення послуг та бронювання: Можливість оформлення замовлення, включаючи вибір дати, часу та локації послуги. Інтеграція з Google Calendar для синхронізації графіків виконавців та замовників. Система підтвердження замовлень та зворотного зв’язку.

Нефункціональні вимоги формують основу якості системи, забезпечуючи її продуктивність, безпеку та зручність використання.

- Продуктивність та масштабованість: Забезпечення високої швидкодії при обробці запитів та великій кількості одночасних користувачів. Можливість горизонтального та вертикального масштабування системи у відповідь на збільшення навантаження.
- Безпека: Реалізація механізмів захисту від несанкціонованого доступу, SQL-ін'єкцій, XSS та інших веб-атак. Шифрування передачі конфіденційних даних, а також зберігання критичної інформації (наприклад, особистих даних користувачів, токенів аутентифікації).
- Надійність та відмовостійкість: Забезпечення безперебійної роботи системи з можливістю швидкого відновлення після збоїв.

- Зручність використання: Інтуїтивно зрозумілий інтерфейс користувача Мінімізація кількості кліків для виконання основних операцій, що сприяє покращенню користувацького досвіду.
- Підтримка та розширюваність: Модульна архітектура, яка дозволяє додавати нові функції без суттєвих змін в існуючому коді. Документованість коду та використання перевірених фреймворків і бібліотек для забезпечення стабільності та легкості підтримки.

Для забезпечення повної функціональності застосунку важливо чітко визначити як внутрішні, так і зовнішні взаємодії. Взаємодія між модулями аутентифікації, управління профілями, замовленнями та повідомленнями через чітко визначені API та внутрішні сервіси. Використання принципів інверсії залежностей для полегшення тестування та підтримки коду.

Документування вимог Оформлення вимог проводиться згідно з прийнятими стандартами розробки ПЗ. Документ має містити:

- Повний опис кожної вимоги з обґрунтуванням її необхідності.
- Прототипи основних інтерфейсів користувача.
- Функціональні діаграми та опис сценаріїв використання.
- Технічні специфікації для інтеграції з зовнішніми сервісами.

2 РОЗРОБКА СИСТЕМНОЇ АРХІТЕКТУРИ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Створення якісних програмного забезпечення, веб-застосунків чи то комплексних систем починається з ретельного проєктування архітектури, що є неодмінно фундаментом для подальшої розробки та підтримки. На цьому етапі визначаються ключові структурні компоненти відповідно задуманої бізнес логіки, їх взаємодія та принципи обміну даними, що дозволяє забезпечити як ефективну роботу всіх підсистем, так і легкість в додаванні нового функціоналу з мінімальними змінами в існуючому коді.

Правильно спроєктована архітектура гарантує високу масштабованість, надійність та безпеку всієї системи, що особливо важливо для продуктів, орієнтованих на вирішення складних бізнес-задач. Розробка програмного забезпечення є логічним продовженням цього етапу: вона включає написання коду, створення інтерфейсів окремих об'єктів, тестування і інтеграцію окремих модулів у єдину систему. Впровадження сучасних методологій розробки, а також використання передових технологічних інструментів сприяє не лише підвищенню якості кінцевого продукту, але й зменшенню часу розробки та спрощенню подальшої підтримки.

Для забезпечення кращих практик проєктування необхідно чітко розуміти початкові потреби програмного забезпечення та приймати рішення, спираючись на функціональні та нефункціональні вимоги, особливості предметної області,

Таким чином, обґрунтування вибору технологічних рішень, бібліотек, підходів та побудова системної архітектури є безумовно критично важливим етапом для досягнення високої ефективності, зручності використання та конкурентоспроможності у побудові сучасних веб-застосунків.

2.1 Вибір архітектури проєкту

На етапі проєктування архітектури даного застосунку одним із основних рішень є вибір між монолітною та мікросервісною архітектурою. У контексті даного проєкту вибір зупинився саме на монолітній архітектурі, через ряд практичних та технічних чинників, які полегшують розробку. Монолітна архітектура передбачає реалізацію одного цілісного застосунку, в межах якого реалізуються всі основні компоненти: бізнес-логіка, інтерфейс користувача, доступ до даних та взаємодія з зовнішніми сервісами. Такий підхід забезпечує простоту в розробці, тестуванні, розгортанні, підтримці та особливо в пошуку та виправленню помилок.

Перевагою моноліту є також швидкість розробки та зменшення накладних витрат на налаштування взаємодії між усіма сервісами застосунку, управління інфраструктурою та обслуговуванні. В рамках даного проєкту, орієнтованого на створення веб-застосунку для пошуку та замовлення послуг, використання монолітного підходу дозволяє зосередитись на реалізації ключових функцій без ускладнення архітектури зайвою сервісною декомпозицією. Крім того, обраний технологічний стек — Python з використанням фреймворку Django — природно сприяє розробці саме монолітних застосунків завдяки своїй структурі та архітектурному патерну MVC, який реалізує фреймворк Django, розшифровується як Model View Controller. Цей патерн зручно розбиває застосунок на моделі даних, представлення та контролер, який відповідає за дії користувача.

Наявності вбудованого ORM, системи маршрутизації, шаблонізатора, засобів аутентифікації та адміністративної панелі, що надається розробником прямо з “коробки” та дозволяє концентрувати увагу на реалізації бізнес-логіки застосунку і не перейматися іншими деталями, які вже підготували за нас. Таким чином, монолітна архітектура в рамках даного проєкту є обґрунтованим вибором, що відповідає його масштабам, обраним технологіям та надає достатню гнучкість для

майбутнього розвитку і розширення функціоналу [5].

2.2 Пошук акторів та варіантів використання

На цьому етапі розробки веб-застосунку потрібно визначити основних акторів та варіанти використання, що відображають функціональні можливості застосунку з погляду взаємодії з кінцевими користувачами та їхніми можливостями. Визначено три основних типа акторів:

- User (Користувач) — узагальнений актор, який охоплює всі базові та спільні дії різних акторів, пов’язані з автентифікацією та налаштуванням облікового запису. До функціоналу цього користувача належать реєстрація як клієнта або працівника, вхід до застосунку, перегляд та оновлення налаштувань облікового запису.
- Customer (Клієнт) — користувач, який шукає та бронює послуги. Його взаємодія у застосунку включає перегляд списку працівників, ознайомлення з їх деталями, бронювання зустрічей, перегляд та скасування власних записів.
- Worker (Працівник) — користувач, який надає послуги. До його можливостей входить керування інформацією про клієнтів, розташуваннями, графіками роботи та прийнятими записами. Працівник також може скасовувати зустрічі з клієнтами.

У системі також передбачено інтеграцію з Google Calendar для обох типів користувачів. Усі користувачі мають можливість під’єднати або від’єднати свій обліковий запис Google Calendar для синхронізації подій.

Функціональність системи структуровано за блоками:

- Authentication — охоплює всі дії, пов’язані з авторизацією та управлінням профілем користувача.
- Calendar Integration — функції підключення та відключення Google Calendar

користувача.

- Customer Functionality — дії клієнта щодо пошуку та бронювання послуг працівників.
- Worker Functionality — дії працівника з адміністрування особистого графіка та локацій надання послуг.

Побудована діаграма варіантів використання дозволяє чітко окреслити можливості кожного з типів користувачів, зрозуміти обсяг функціональності та забезпечити подальше коректне проєктування архітектури застосунку. Такий підхід сприяє чіткої валідації вимог до застосунку та ефективній реалізації необхідного функціоналу, а також спрощує додавання нового функціоналу у майбутньому.

Звичайний користувач. Усі користувачі, незалежно від своєї ролі (клієнт або працівник), мають доступ до базових функцій системи:

- Sign Up as Customer / Worker – можливість зареєструватися в застосунку як клієнт або як працівник.
- Login – автентифікація в застосунок для отримання доступу до персоналізованого функціоналу.
- View Account – перегляд власної сторінки облікового запису.
- Update Settings – редагування персональних даних та налаштувань.

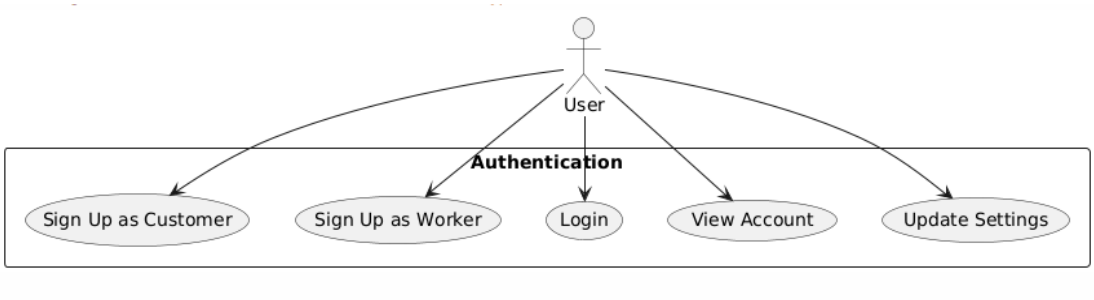


Рис. 2.1 – Діаграма варіантів використання базового функціоналу

- Connect Google Calendar / Disconnect Google Calendar – інтеграція з Google

Calendar для автоматичної синхронізації запланованих послуг або від'єднання від сервісу, для припинення створення подій у Google календарі користувача.

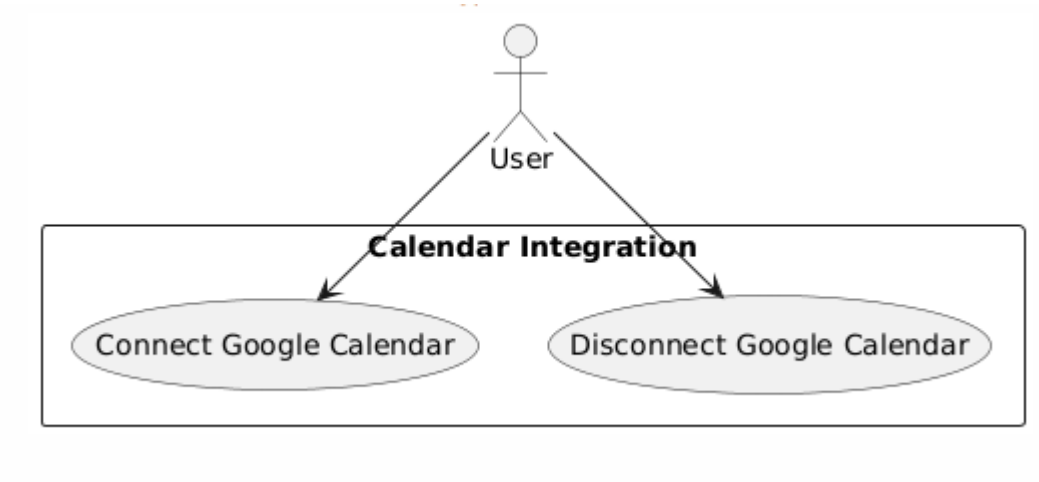


Рис. 2.2 – Діаграма варіантів використання взаємодії користувача з Google Calendar інтеграцією

Клієнт (Customer). Користувач із роллю клієнта має змогу:

- Browse Workers – переглядати список працівників у застосунку.
- View Worker Details – переглядати детальну інформацію про конкретного працівника.
- Book Appointment – забронювати послугу із вибраним працівником.
- View Appointments – переглядати власні поточні та майбутні записи послуг.
- Cancel Appointment – скасовувати раніше заброньовані послуги.

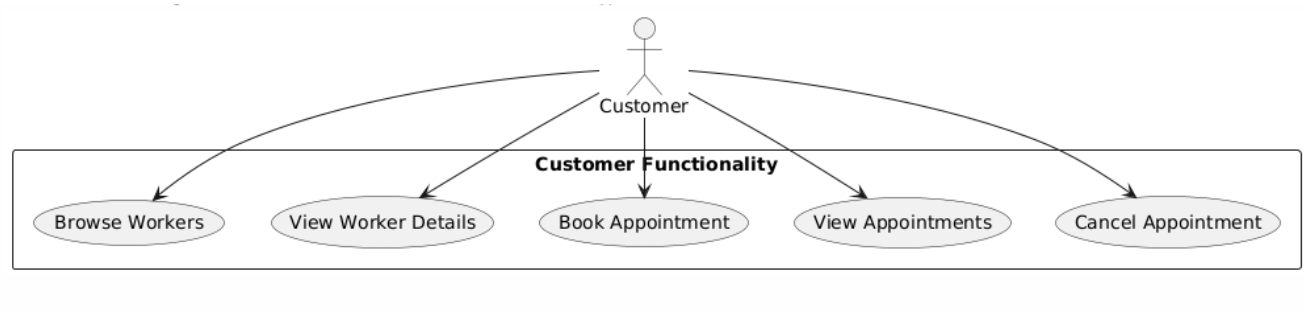


Рис. 2.3 – Діаграма варіантів використання функціоналу клієнта

Працівник (Worker). Користувач із роллю працівника має доступ до більш розширених адміністративних можливостей налаштування своєї послуги:

- View Customers – переглядати список список своїх клієнтів.
- Manage Locations (Create, Edit, Delete) – додавати, редагувати або видаляти локації, в яких надається послуга.
- Manage Schedules (Create, Edit, Delete) – створювати та оновлювати розклад роботи, а також видаляти непотрібні графіки.
- View Appointments – переглядати майбутні записи на послугу.
- Cancel Appointment – скасовувати призначені записи у разі необхідності.

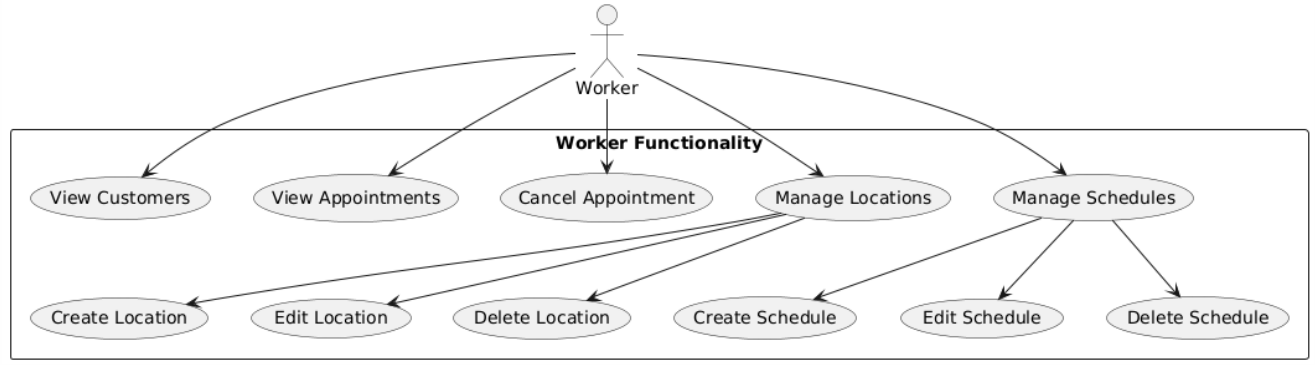


Рис. 2.4 – Діаграма варіантів використання функціоналу працівника

Такий розподіл функціональності між ролями дозволяє системі залишатися простою та зручною для клієнтів, забезпечуючи при цьому достатній рівень контролю та управління для працівників, які надають послуги. Крім цього даний розподіл не ускладнює добавлення нового функціоналу відповідно до майбутніх потреб.

2.3 Аналіз логіки виконання: Діаграми послідовностей

У цьому розділі проаналізовано чотири основні сценарії взаємодії користувачів із застосунком за допомогою діаграм послідовностей, які демонструють динаміку обміну сигналами між акторами, компонентами та основними класами застосунку. Діаграми послідовностей використовуються для відображення порядку викликів і повернень між об’єктами в застосунку, що дозволяє наочно побачити реалізацію кожного бізнес-сценарію. Для кожного з наведених сценаріїв наведено опис ролей акторів, задіяних компонентів та послідовність сигналів, що виконуються та обробляються.

Розглянемо діаграму послідовностей реєстрації нового користувача з обраною роллю.

- Актор: New User.
- Компоненти: SignUpView на базі TemplateView, форми CustomerSignUpForm або WorkerSignUpForm, модель User.
- Користувач надсилає GET запит до /signup/, що викликає SignUpView для відображення форми реєстрації.
- Після заповнення форми і відправки POST-запиту відбувається валідація даних у form.is_valid(), далі створюється новий екземпляр User через виклик form.save().
- При успішному створенні облікового запису користувач перенаправляється на сторінку входу /login/ із параметром success_url.

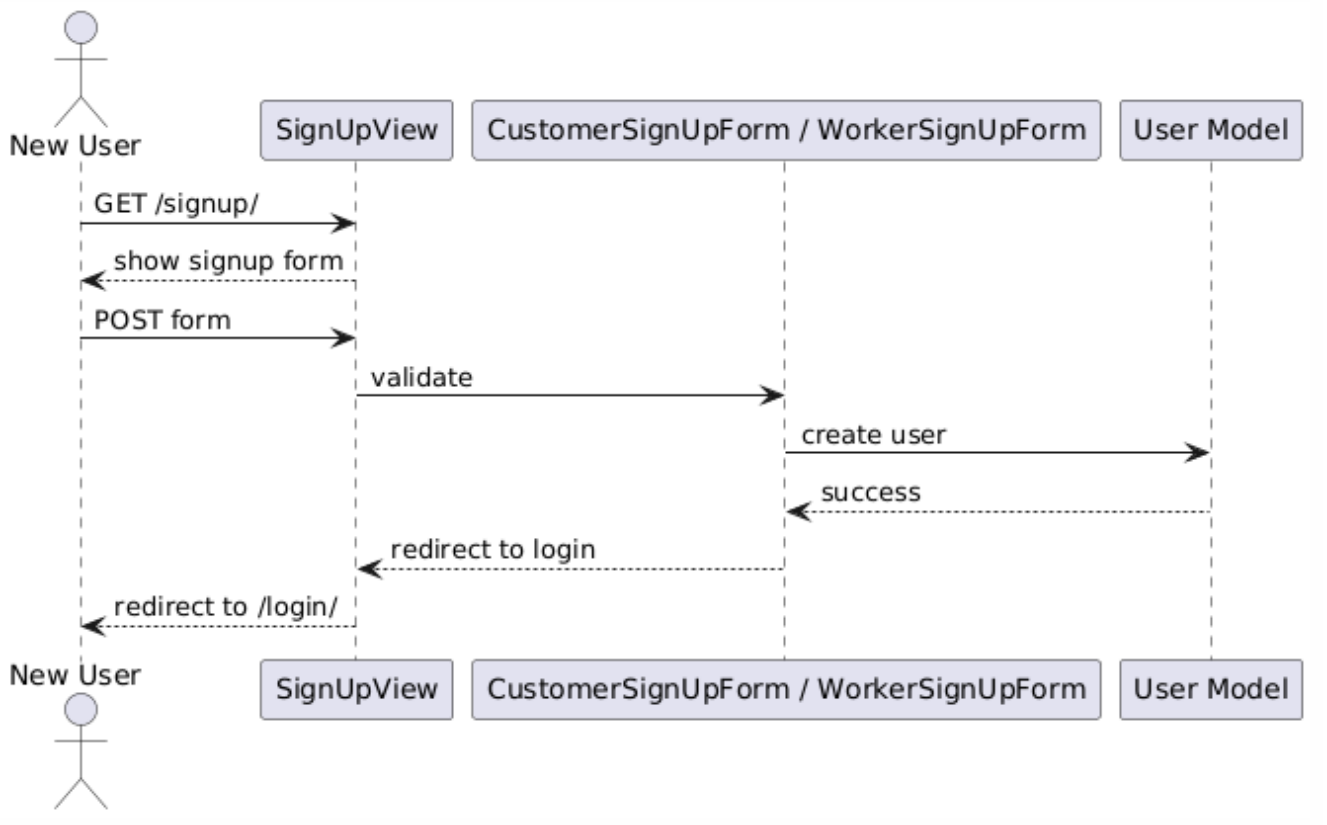


Рис. 2.5 – Діаграма послідовностей реєстрація нового користувача

Таке докладне відображення динаміки взаємодії компонентів застосунку за допомогою діаграм послідовностей дозволяє уточнити порядок обробки сигналів та забезпечити відповідність реалізації початковим вимогам і полегшити подальше тестування та підтримку архітектури.

Розглянемо діаграму послідовностей підключення користувача до Google Calendar з метою подальшого використання даних користувача у створенні подій у його календарі.

- Актор: User.
- Компоненти: функції google_calendar_init та google_calendar_redirect, Google OAuth Flow, Session, модель User.
- Користувач робить GET запит до /google-calendar/init/, що запускає google_calendar_init — створення потоку OAuth за допомогою flow.from_client_secrets_file() і формування authorization_url.
- Параметри state зберігаються в сесії, після чого користувача перенаправлено на сторінку авторизації Google, для підтвердження надання даних з

подальшим використанням їх для створення подій в календарі користувача.

- Після успішного входу Google надсилає запит на /google-calendar/redirect/, де google_calendar_redirect перевіряє стан із сесії і викликає flow.fetch_token(), отримуючи об’єкт credentials.
- Отримані облікові дані зберігаються в полі google_calendar_credentials моделі User, та виконується переадрисація назад на сторінку облікового запису /account/.

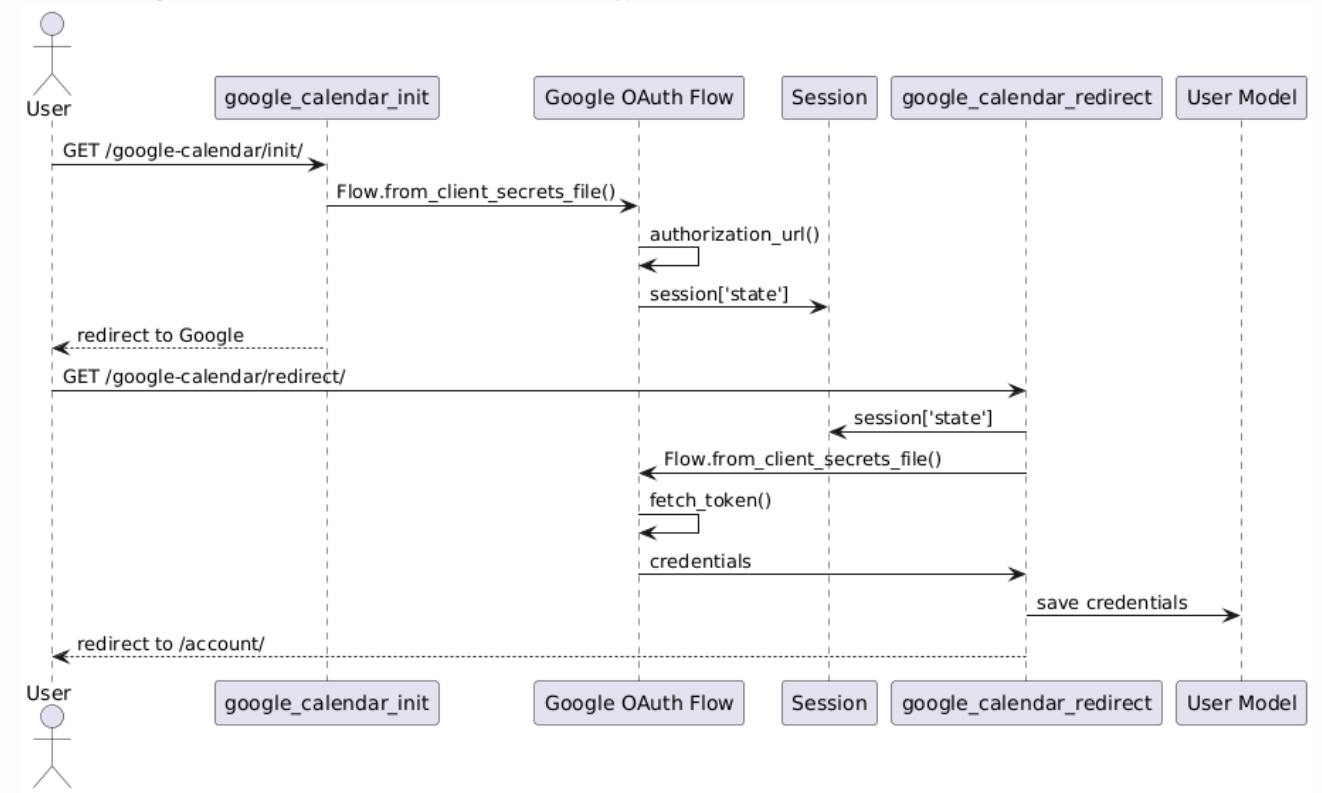


Рис. 2.6 – Діаграма послідовностей інтеграції користувача з Google Calendar

Розглянемо діаграму послідовностей створення запису на прийом до працівника клієнтом.

- Актор: Customer.
- Компоненти: AppointmentCreateView (CBV на базі CreateView), AppointmentForm, Session, CreateAppointmentUseCase, модель Appointment.
- Клієнт відправляє HTTP POST запит до /appointment/<pk>/, викликаючи метод post() з AppointmentCreateView.
- Представлення отримує форму через get_form(), після чого відображає

заповнені дані форми.

- Виконується перевірка коректності форми через form.is_valid(), і з сесії витягується збережений час замовлення.
- Через впроваджений сценарій використання create_appointment_u_c створюється новий запис у таблиці Appointment.
- Після успішного створення клієнт перенаправляється на сторінку /appointments/ із кодом 302 Redirect.

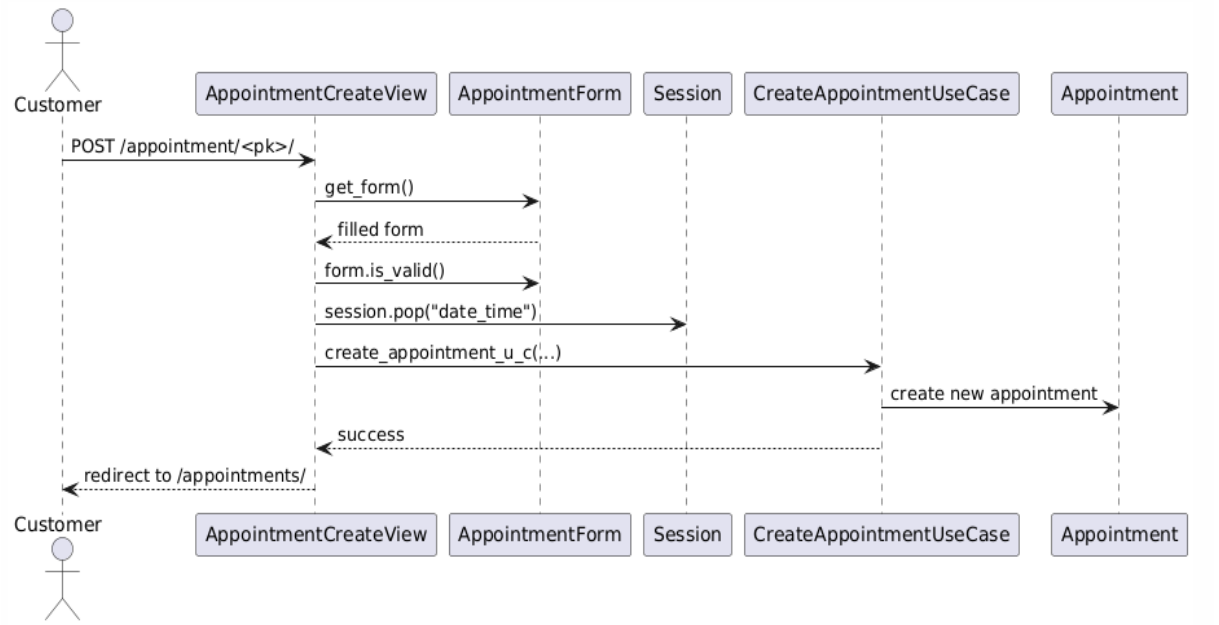


Рис. 2.7 – Діаграма послідовностей створення запису на прийом

Розглянемо діаграму послідовностей перегляду розкладу працівника, у даній діаграмі актором представлено працівника, проте переглядати розкалд може також клієнт з існуючим обліковим записом або без нього.

- Актор: Worker, Customer або не аутентифікований користувач.
- Компоненти: WorkerDetailView (CBV на базі DetailView), RetrieveWorkerScheduleUseCase, модель Schedule, шаблон user/worker_detail.html
- Користувач виконує GET запит до /workers/<pk>/?day=..., викликаючи метод get() у WorkerDetailView.

- Представлення через впроваджений сценарій використання retrieve_worker_schedule_u_c формує тижневий розклад працівника використовуючи дані з моделі Schedule.
- Отриманий розклад (week_schedule) повертається до представлення, яке передає дані в контекст і викликає рендеринг шаблону render() для формування HTML-сторінки.

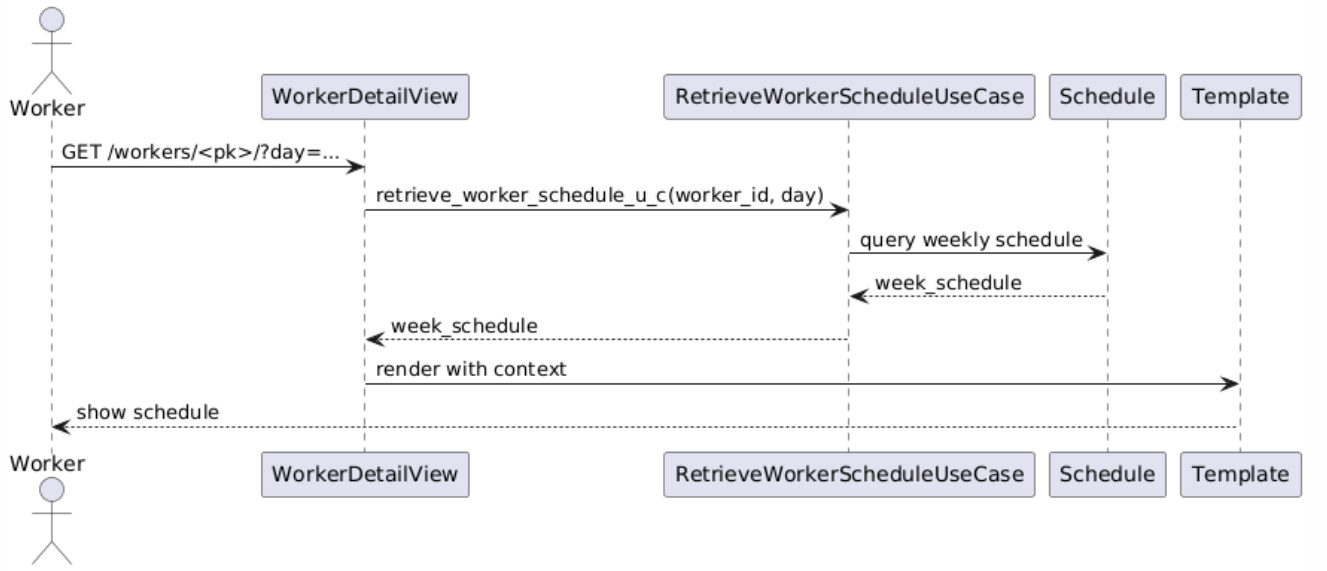


Рис. 2.8 – Діаграма послідовностей перегляду розкладу працівника

2.4 Проєктування моделей бази даних

Діаграма класів є важливим інструментом для наочності структури застосунку, що дозволяє зрозуміти, наявні об'єкти у застосунку та те як вони взаємодіють між собою. У нашому випадку, діаграма класів відображає основні сутності застосунку та їх взаємозв'язки.

Дана UML-діаграма класів реалізує статичну модель застосунку, де чотири основні моделі — User, Location, Schedule та Appointment — поєдані різними зв’язками, що відображають ролі працівника та клієнта, місце де надається послуга, індивідуальні графіки роботи та записи на прийом до працівника. Структурний

підхід UML забезпечує зрозуміле розділення відповідальності: User містить інформацію про дані користувача та його роль, клас Location містить адресні атрибути, Schedule визначає інтервали часу коли працівник надає свої послуги, та Appointment містить інформацію про запис з прив'язкою до користувача, локації та часу. Така архітектура гарантує зрозумілість реалізації, зручність підтримки та подальшого розширення застосунку.

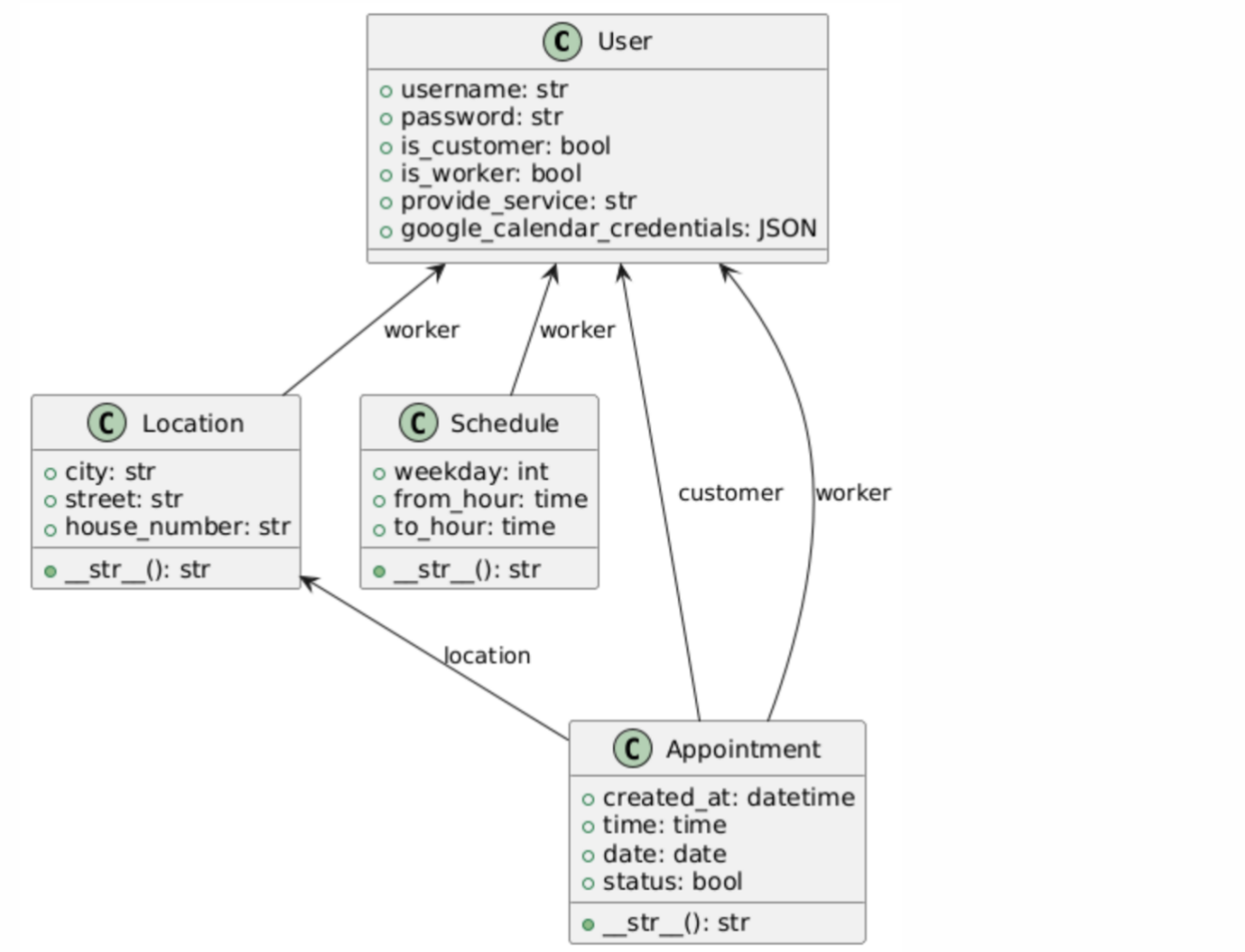


Рис. 2.9 – Діаграма класів

Клас User представляє користувача застосунку, що може одночасно виконувати роль клієнта та працівника. Логічні прапорці is_customer та is_worker визначають, чи має даний користувач можливість користуватися застосунком як клієнт або обслуговувати клієнтів як працівник. Атрибут provide_service зберігає

послугу, що надається даним працівником, а google_calendar_credentials забезпечує зберігання облікових даних для інтеграції із зовнішнім сервісом Google Calendar.

Клас Location описує місце надання послуг і включає атрибути city, street та house_number, що формують повну поштову адресу. Спеціальний метод str() реалізовано для зручного текстового представлення адреси користувача.

Клас Schedule містить робочий графік працівника за допомогою атрибутів weekday, from_hour та to_hour, які визначають день тижня та часові інтервали коли працівник надає свої послуги. Метод str() виводить повертає зрозуміле текстове представлення розкладу, що полегшує його відображення в інтерфейсі або в логах застосунку.

Клас Appointment відповідає за запис на прийом та зберігає інформацію про created_at, date, time і статус запису до працівника. Метод str() повертає дані у зрозумілому текстовому представленні для виведення в інтерфейсах або логах.

Клас User пов’язаний із Location зв’язком одни до багатьох, що відображає ситуацію, коли один працівник може обслуговувати клієнтів у кількох місцях надання послуг. Така організація дозволяє гнучко додавати нові локації без лишніх змін.

Через зв’язок один до багатьох кожен працівник User може мати декілька записів у класі Schedule, що реалізує гнучкий графік із різними днями та часовими інтервалами, які можуть містити перерви у роботі працівника.

Відображено дві асоціації між User та Appointment для ролей клієнта та працівника: клієнт створює записи, а працівник їх опрацьовує. Обидві зв’язки реалізовані за допомогою один до багатьох і дозволяють фіксувати історію звернень з обох точок зору.

Зв’язок Location один до багатьох з Appointment визначає, що одне місце обслуговування може бути пов’язане з багатьма зверненнями клієнтів.

Використання атрибута google_calendar_credentials у класі User дозволяє синхронізувати розклад прийомів із Google Calendar, що підвищує зручність планування та нагадувань для користувачів, що забезпечує кращий досвід

користування застосунком в цілому.

Запропонована UML-діаграма класів формує надійну основу для подальшої розробки застосунку керуванні прийомами до працівника, інтеграції з календарними сервісами та розширення функціоналу. Чітке відокремлення сутностей і асоціацій гарантує зрозумілість коду та спрощує роботу над проєктом.

3 РОЗРОБКА ТА ТЕСТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ

3.1 Первинні налаштування залежностів проєкту та середовищем розробки

Poetry — сучасний менеджер залежностей та інструмент побудови пакунків для Python, який забезпечує зручне та надійне керувати середовищем розробки, також допомагає автоматизувати процес оновлення бібліотек та гарантувати відтворюваність проєкту. Використання Poetry особливо важливо на початковому етапі налаштування проєкту, оскільки:

- Забезпечує чітке керування залежностей у єдиному файлі pyproject.toml.
- Підтримує прості команди для додавання, видалення та оновлення бібліотек (poetry add, poetry remove, poetry update).
- Автоматично вирішує конфлікти версій та формує файл poetry.lock, що фіксує точні версії залежностей, що запобігає помилок.
- Створює ізольоване віртуальне середовище під проєкт, що запобігає конфліктам з глобальними пакетами та іншими наявними проєктами.

```
[tool.poetry] name = "customer-service"
version = "0.1.0"
description = "A web application has been created for service
providers and customers looking to schedule a service."
authors = ["Roman Babii"]
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.11"
django = "^5.1.6"
psycopg = "^3.2.4"
django-environ = "^0.12.0"
django-allauth = "^65.4.1"
google-auth-oauthlib = "^1.0"
google-api-python-client = "^2.87"
poetry = "^2.1.1" update = "^0.0.1"
pydantic = "^2.10.6"
pydantic-settings = "^2.8.1"
dependency-injector = "^4.46.0"
pyjwt = "^2.10.1"
```

```
[tool.poetry.group.dev.dependencies]
mypy = "^1.15.0"
types-pyyaml = "^6.0.12.20241230"
types-requests = "^2.32.0.20250301"
types-setuptools = "^75.8.2.20250301"
pre-commit = "^4.1.0"
ruff = "^0.9.9" bandit = "^1.8.3"
mypy-extensions = "^1.0.0"

[build-system]
requires = ['poetry-core']
build-backend = 'poetry.core.masonry.api'
```

Лістинг коду 3.1 – Початкової конфігурації проєкту

Таким чином, інтеграція Poetry у даному проєкті дозволяє значно підвищити продуктивність розробки та спростити підтримку та оновлення проєкту.

3.2 Налаштування інструментів автоматичного форматування

При розробці сучасного програмного забезпечення важливо, щоб програма не тільки виконувала свою функціональність, але й слідувала кращим практикам написання коду та зберігала єдиний стиль проєкту, що полегшує підтримку та подальшу розробку застосунків. Тому для підтримки високих стандартів коду та автоматичного виправлення помилок у проєкті налаштовано систему pre-commit, яка виконує набір перевірок перед кожним оновленням коду у віддаленому репозиторію. Нижче представлено конфігурацію у файлі .pre-commit-config.yaml.

```
repos:
 - repo: https://github.com/pre-commit/pre-commit-hooks
   rev: v5.0.0
   hooks:
     - id: check-yaml
       args:
         - --unsafe
     - id: end-of-file-fixer
     - id: trailing-whitespace
     - id: check-ast
     - id: check-case-conflict
```

```
    - id: check-merge-conflict
    - id: check-toml
    - id: debug-statements
    - id: detect-private-key

- repo: https://github.com/charliermarsh/ruff-pre-commit
  rev: v0.9.9
  hooks:
    - id: ruff
      args: [ "--fix", "--unsafe-fixes", "--exit-non-zero-on-fix" ]
    - id: ruff-format

- repo: https://github.com/PyCQA/bandit
  rev: 1.8.3
  hooks:
    - id: bandit
      args: ["-r", "."]

- repo: https://github.com/pre-commit/mirrors-mypy
  rev: v1.15.0
  hooks:
    - id: mypy
      additional_dependencies:
        - "pydantic>=2.10.6"
```

Лістинг коду 3.2 – Початкової конфігурації налаштування інструментів автоматичного форматування коду

Основні переваги використання pre-commit та інструментів автоматичного форматування:

- Автоматичне виявлення та виправлення форматування: end-of-file-fixer, trailing-whitespace, ruff-format забезпечують єдиний стиль коду без зайвих пробілів та не консистентних відступів.
- Статичний аналіз та linting: інтеграція Ruff та Bandit з автоматичним виправленням помилок дозволяє вчасно виявляти порушення стилю, потенційні помилки та вразливості.
- Перевірка типів: mypy із додатковими залежностями гарантує відповідність анотації типів, зменшуючи ризик винятків.
- Контроль цілісності конфігурацій: перевірки check-yaml, check-toml та check-case-conflict попереджають неправильні формати файлів і конфлікти злиттів.

- Підвищення якості коду: автоматичний pre-commit процес гарантує, що в репозиторій потрапляє лише вже перевірений і якісний код.

Впровадження pre-commit разом із зазначеними інструментами значно скорочує час на перевірку коду вручну та сприяє стабільності та підтримуваності проєкту, а також формує єдиний стандарт якості.

3.3 Структура проєкту та поділ на компоненти

Структура програмного забезпечення та чіткий поділ на компоненти є безумовно важливою частиною розробки, що забезпечує чисту архітектуру та спрощує навігацію в проєкті. Бізнес-логіка застосунку повинна бути відокремлена від фреймворку, взаємодії з базою даних повинна мати свій прошарок в архітектурі та не бути переплетеною у різних компонентах застосунку [6]. У даному проєкті можна виділити кілька основних архітектурних ланок, уся взаємодія з базою даних проводиться в каталозі data_access, бізнес-логіка розташована в папці core, інтеграція з сторонніми APIs у папці integrations та кожен компоненти фреймворку має свою окрему директорію. Це все забезпечує чітку структуру та організацію проєкту, що сприяє зрозумілій та чистій архітектурі веб-застосунку. Наочно побачити структуру проєкту можна у додатку А. Нижче наведено огляд основних каталогів і файлів проєкту, що реалізує принципи розділення відповідальності та підтримує масштабованість і зручність розробки:

- Пакет core: Містить реалізацію основних правил і сценаріїв роботи застосунку — створення та обробка записів (appointment) та отримання розкладу (schedule) працівників, інтеграція з календарем (google_calendar), а також загальні утиліти (utils).
- Пакет data_access містить рівень доступу до бази даних. Реалізує шаблон

репозиторій, інкапсулюючи всі операції з базою даних. Забезпечує чіткий інтерфейс для CRUD-операцій над моделями Appointment, Schedule та User, ізольовуючи їх від безпосереднього використання ORM Django у бізнес-логіці.

- Пакет integrations вміщує зовнішні сервіси. Містить код для інтеграції з API сторонніх провайдерів, зокрема Google Calendar. Структура включає інтерфейси та сервіси реалізації та допоміжні файли.
- Інші компоненти, так як appointment/, authentication/, worker/ — компоненти Djago, що відповідають за маршрути, моделі, форми і представлення користувачів та замовлень.
- common/ — спільні утиліти, типи та налаштування, що використовуються у всьому проєкті.
- static/ та templates/ — відповідно статичні ресурси та HTML-шаблони.
- Конфігураційні файли .env, pyproject.toml, .pre-commit-config.yaml тощо забезпечують керування середовищем та інструментами розробки.

Це все забезпечує чітке розділення відповідальностей між бізнес-логікою, доступом до даних і зовнішніми інтеграціями. Окрім цього, така структура спрощення тестування та можливість незалежного оновлення окремих модулів.

3.4 Керування інверсії залежностей проєкту

У сучасних веб-застосунках важливо забезпечити правильне управління залежностями між компонентами системи. Це дозволяє зробити архітектуру коду гнучкою, масштабованою і забезпечити легкість в тестуванні.Принцип інверсії залежностей (Dependency Inversion Principle, DIP) належить до п’яти SOLID-принципів об’єктно-орієнтованого проєктування. Суть DIP полягає в перенаправленні зв’язків між модулями: замість прямої залежності високорівневих

модулів від низькорівневих, обидва класи зв’язуються через інтерфейси чи абстрактні класи.

Логіка взаємодії модулів розробляється навколо абстракцій, що зменшує жорстке зв’язування компонентів. Тобто високорівневий модуль не знає про конкретні класи низькорівневих модулів, а оперує лише їхньою поведінкою, заданою абстракцією. За таким принципом, сучасні застосунки широко використовують DI (Dependency injection) для забезпечення DIP [7].

Застосування DIP забезпечує слабке зв’язування між модулями, що підвищує гнучкість архітектури. Наприклад, за DIP класи бізнес-логіки інкапсулюють залежності від технічних сервісів через інтерфейси, тож будь-яку конкретну реалізацію сервісу можна замінити новою, не змінюючи код бізнес-логіки. Ця явна інверсія залежностей робить програму більш зрозумілою та керованою. Усі залежності визначені в одному місці, через контейнери або фабрики, і змінювати їх простіше.

У Python одним із інструментів для забезпечення DIP є бібліотека Dependency Injector. Це фреймворк для організації dependency injection (DI), що дозволяє створення об’єктів і визначення залежностей через оголошення контейнера з залежностями. Dependency Injector надає механізми провайдерів фабрик для створення екземплярів і налагоджувачів для автоматичної ін’єкції у функції чи класи.

Централізоване керування створення об’єктів спрощує забезпечення консистентності конфігурацій, наприклад, єдиний спосіб заданого параметра api_key у сервісі який інтегрує стороній API і зменшує ризики непослідовності. Це зменшує ймовірність появи об’єкта з невірним парамeнтом. Крім того, така архітектура дозволяє вводити додаткові рівні захисту або перевірки при створенні об’єктів, адже вся генерація проходить через єдиний налагоджений шлях. Врешті-решт, очевидна відповідальність контейнера за збірку об’єктів підвищує стійкість системи до помилок і ускладнює появу таємних залежностей, сприяючи загальній безпеці коду.

Завдяки явній ін’єкції залежностей, бібліотека Dependency Injector покращує масштабованість проєкту. Якщо потрібно додати новий сервіс або модуль, достатньо описати його у контейнері, як новий провайдер, і передати функціям чи класам через ін’єкцію, не змінюючи існуючий код.

Наприклад, якщо з’являється новий клас сервісу, його просто додають як providers.Factory або providers.Singleton у декларативний контейнер. Після цього інші частини програми можуть отримувати цей сервіс через DI, наприклад, за допомогою декоратора @inject. Існуючим модулям не потрібно знати про деталі нового сервісу – достатньо оновити контейнер. Такий підхід суттєво полегшує розвиток застосунку, зі зростанням числа компонентів керованість залежностями не деградує, оскільки всі визначення знаходяться в одному місці.

Dependency Injector суттєво полегшує процес тестування коду. Оскільки, під час модульного тестування можна легко змінювати реальні сервіси на штучні об’єкти які створені спеціально для тестування. Бібліотека надає механізм override, будь-який провайдер можна замінити іншим під час виконання коду.

У наявному проєкті реалізувано керування залежностями за допомогою бібліотеки dependency-injector. Реалізація інверсії залежностей у проєкті знаходиться у файлі dependency_container.py створено клас контейнера залежностей та визначено існуючі сервіси, фабрики та сценарії використання.

```
WIRING_MODULES: tuple[str, ...] = (
   'appointment.views',
   'authentication.views',
)


class DependencyContainer(DeclarativeContainer):
   wiring_config = WiringConfiguration(modules=WIRING_MODULES)
   config = Configuration()

   # External integrations
   google_service_factory: Factory[GoogleServiceFactory] = Factory(
       GoogleServiceFactoryImpl
   )
   # Repositories
   appointment_reader_service: Factory[BaseAppointmentReaderService]
= Factory(
```

```
        AppointmentReaderService, appointment_model=Appointment
    )
    appointment_writer_service: Factory[BaseAppointmentWriterService]
= Factory(
        AppointmentWriterService, appointment_model=Appointment
    )

    schedule_reader_service: Factory[BaseScheduleReaderService] =
Factory(
        ScheduleReaderService, schedule_model=Schedule
    )
    user_reader_service: Factory[BaseUserReaderService] = Factory(
        UserReaderService, user_model=User
    )

    # Services
    google_calendar_event_service:
Factory[BaseGoogleCalendarEventService] = Factory(
        GoogleCalendarEventService,
        google_service_factory=google_service_factory,
    )

    # Use Cases
    # Appointment
    create_appointment_u_c: Factory[CreateAppointmentUseCase] =
Factory(
        CreateAppointmentUseCaseImpl,
        user_reader_service=user_reader_service,
        appointment_writer_service=appointment_writer_service,
        google_calendar_event_service=google_calendar_event_service,
    )

    # Schedule
    retrieve_worker_schedule_u_c:
Factory[RetrieveWorkerScheduleUseCase] = Factory(
        RetrieveWorkerScheduleUseCaseImpl,
        appointment_reader_service=appointment_reader_service,
        schedule_reader_service=schedule_reader_service,
    )
```

Лістинг коду 3.3 – Реалізація інверсії залежностей

Тут визначено фабрики (Factory) для створення екземплярів репозиторіїв, сервісів та варіантів використання (Use Cases). Це забезпечує централізоване управління залежностями та їх конфігурацією.

Ініціалізація контейнера відбувається у методі ready() застосунку customer_service, що забезпечує коректне завантаження залежностей одразу після запуску Django.

```
from django.apps import AppConfig
from django.conf import settings


class CustomerServiceConfig(AppConfig):
   default_auto_field = 'django.db.models.BigAutoField'
   name = 'customer_service'

   def ready(self) -> None:
       """Initialize Dependency Injection after Django is fully
loaded."""
       from customer_service.dependency_container import
DependencyContainer

       container = DependencyContainer()
       container.config.from_dict(settings.__dict__)
       container.init_resources()
```

Лістинг коду 3.4 – Ініціалізація контейнеру під час запуску застосунку

Використання ін’єкції залежностей у представленнях. Щоб отримати залежності безпосередньо у представленнях (Views), використовується декоратор @inject та механізм Provide. Таким чином, необхідний варіант використання retrieve_worker_schedule_u_c автоматично передається в метод контроллера без явної ініціалізації класу RetrieveWorkerScheduleUseCaseImpl вручну, реалізація якого наведено у Додатку Б.

```
@inject
def get(
   self,
   request: HttpRequest,
   retrieve_worker_schedule_u_c: RetrieveWorkerScheduleUseCase =
Provide[
       'retrieve_worker_schedule_u_c'
   ],
   *args: Args,
   **kwargs: Kwargs,
) -> HttpResponse:
   worker_id = WorkerID(int(self.kwargs['pk']))
   start_day_str = request.GET.get('day', None)
   start_day = (
       datetime.strptime(start_day_str, '%Y-%m-%d').date()
```

```
        if start_day_str
        else None
    )
    week_schedule = retrieve_worker_schedule_u_c(
        worker_id=worker_id,
        **({'start_day': start_day} if start_day else {}),  # type:
ignore[arg-type]
    )
    context = {'worker': self.get_object(), 'week': week_schedule}
    return render(request, 'user/worker_detail.html', context)
```

Лістинг коду 3.5 – Використання ін’єкції залежностей у представленні розкладу працівника

3.5 Реалізація проєкту

Раніше був наведений фрагмент коду з представлення, який викликає сценарій використання retrieve_worker_schedule_u_c. Тепер розглянемо сервіси, що залучені в його реалізації: appointment_reader_service та schedule_reader_service. Ці сервіси забезпечують доступ до бази даних і реалізують патерни Repository та Data Transfer Object (DTO).

DTO (Data Transfer Object) — це патерн програмування, який передбачає створення окремих об’єктів для передачі даних між різними архітектурними шарами застосунку. DTO не містять бізнес-логіки: їх основна мета — забезпечити структуровану передачу даних між компонентами застосунку.

Основні характеристики DTO:

- Містять лише поля для даних і, зазвичай, мінімальну кількість допоміжних методів.
- Допомагають розділити внутрішню структуру об'єктів та зовнішні представлення даних у коді.
- Підвищують безпеку, оскільки дозволяють явно контролювати, які саме дані передаються.

В архітектурі з розділенням на шари, наприклад, доступ до даних, бізнес-логіка, представлення, використання DTO є критично важливим для забезпечення гнучкості та незалежності компонентів системи.

У межах реалізованого застосунку паттерн DTO активно застосовується для розмежування бази даних і бізнес-логіки. Наприклад, для представлення даних про розклад працівника було створено окремий DTO-клас ScheduleHours.

```
class ScheduleHours(BaseModel):
    from_hour: time
    to_hour: time

    model_config = ConfigDict(from_attributes=True)
```

Лістинг коду 3.6 – DTO-клас представлення розкладу працівника

Паттерн Repository — це один із ключових архітектурних патернів у розробці програмного забезпечення, мета якого — абстрагувати доступ до джерела даних від бізнес-логіки застосунку. Repository створює проміжний шар між бізнес-логікою та базою даних, завдяки чому:

- Код бізнес-логіки не залежить від конкретної реалізації зберігання даних.
- Стає легше проводити модульне тестування за допомогою спеціально реалізованих репозитаріїв для тестування, що замінюють справжні реалізації.
- Зростає масштабованість проєкту: зміну бази даних можна реалізувати без змін у бізнес-логіці.

У проєкті було створено окремі сервіси для читання і запису даних для кожної сутності. Для прикладу розглянемо роботу з моделлю Schedule через читання даних.

```
class BaseScheduleReaderService(ABC):
    @abstractmethod
    def get_worker_schedules_hours_for_weekday(
        self, worker_id: WorkerID, weekday_number: int
    ) -> list[ScheduleHours]:
        """Retrieves worker schedule hours data for a given worker ID
and weekday number."""
class ScheduleReaderService(BaseScheduleReaderService):
```

```
    __slots__ = ('schedule_model',)

    def __init__(self, schedule_model: type[Schedule]) -> None:
        self.schedule_model = schedule_model
    def get_worker_schedules_hours_for_weekday(
        self, worker_id: WorkerID, weekday_number: int
    ) -> list[ScheduleHours]:
        return [
            ScheduleHours.model_validate(schedule)
            for schedule in self.schedule_model.objects.filter(
                worker=worker_id, weekday=weekday_number
            ).only('from_hour', 'to_hour')
        ]
```

Лістинг коду 3.7 – Абстракція репозиторію BaseScheduleReaderService та її реалізація у сервісі ScheduleReaderService

ScheduleReaderService реалізує метод отримання даних з бази через ORM Django. Дані перетворюються у DTO об'єкти типу ScheduleHours для подальшого безпечного використання в застосунку. Ін'єкція залежності schedule_model через конструктор __init__ дозволяє легко підміняти модель у тестах.

Аналогічним чином були реалізовані й інші сервіси для роботи з моделями. AppointmentReaderService, AppointmentWriterService — для читання та запису зустрічей. UserReaderService — для отримання даних про користувачів. Реалізації та абстракції класів AppointmentReaderService та AppointmentWriterService, наведено у Додатку Б.

Для побудови ресурсів взаємодії із API Google було створено окрему фабрику — GoogleServiceFactoryImpl, що реалізує протокол GoogleServiceFactory.

```
class GoogleServiceFactory(Protocol):
    __slots__ = ('user_reader_service',)

    def __call__(
        self,
        user_google_credentials: dict[str, Any],
        google_service_name: GoogleServiceNameEnum,
        service_version: str = 'v3',
    ) -> Resource:
        """Retrieves constructed a resource for interacting with
Google API.

        Args:
```

```
            user_google_credentials: The dictionary with user google
calendar credentials data.
            google_service_name: The name of the Google API service
to construct.
            service_version: The version of the API service to
construct.

        Returns: The constructed resource.

        """
class GoogleServiceFactoryImpl(GoogleServiceFactory):
    def __call__(
        self,
        user_google_credentials: dict[str, Any],
        google_service_name: GoogleServiceNameEnum,
        service_version: str = 'v3',
    ) -> Resource:
        return build(
            serviceName=google_service_name,
            version=service_version,
            credentials=self._load_credentials_for_user(
                user_credentials=user_google_credentials
            ),
        )

    @staticmethod
    def _load_credentials_for_user(user_credentials: dict[str, Any])
-> Credentials:
        return Credentials(
            token=user_credentials.get('token'),
            refresh_token=user_credentials.get('refresh_token'),
            token_uri=user_credentials.get('token_uri'),
            client_id=user_credentials.get('client_id'),
            client_secret=user_credentials.get('client_secret'),
            scopes=user_credentials.get('scopes'),
        )
```

Лістинг коду 3.7 – Протокол GoogleServiceFactory та реалізація фабрики GoogleServiceFactoryImpl

Основні функції фабрики:

- Формування підключення до необхідного Google API-сервісу Calendar через бібліотеку google-api-python-client.
- Завантаження користувацьких облікових даних Credentials із переданого словника.
- Абстрагування складнощів побудови клієнтів для зовнішніх API.

Це рішення спрощує бізнес-логіку, оскільки сервіси в застосунку не взаємодіють із фреймворком Django, та делегується спеціалізованій фабриці.

У модулі бізнес-логіки core створено сервіс GoogleCalendarEventService, який реалізує абстракцію BaseGoogleCalendarEventService, фрагмент коду якої наведено у Додатку Б.

Функціональність сервісу:

- Приймає дані користувача та події через спеціальний об’єкт EventData.
- Створює структуру події відповідно до вимог Google Calendar API.
- Викликає побудований через фабрику ресурс Google API для запису події до календаря користувача.

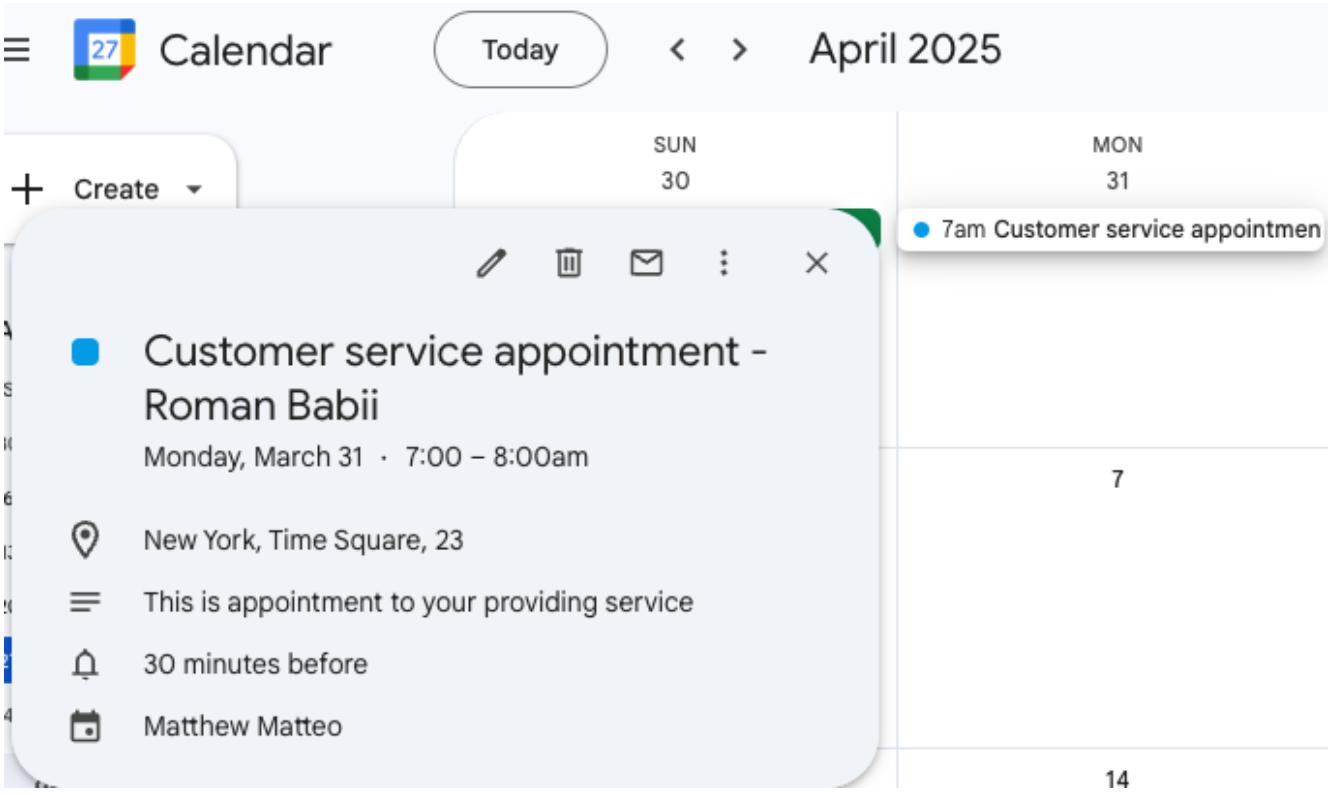


Рис. 3.1 – Створена подія в Google календарі користувача

Таким чином, користувачі сервісу автоматично отримують подію в Google Calendar після успішного бронювання послуги без необхідності вручну додавати її, що покращує їхній досвід користування.

Одним із ключових процесів застосунку є створення запису на послугу. Цей функціонал реалізовано у модулі core відповідно до принципів чистої архітектури, із чітким розділенням бізнес-логіки та зовнішніх залежностей.

Клас CreateAppointmentUseCaseImpl реалізує бізнес-правила створення нового запису користувача на послугу, фрагмент коду наведено у Додатку Б
Основні залежності, що передаються через конструктор:

- BaseUserReaderService — сервіс для отримання даних користувачів, наприклад, імені працівника.
- BaseAppointmentWriterService — сервіс для запису даних у базу даних.
- BaseGoogleCalendarEventService — сервіс для інтеграції з Google Calendar.

Логіка виконання сценарію використання:

- Збереження даних про запис у БД.
  З використанням сервісу appointment_writer_service створюється новий об'єкт Appointment у базі даних.
- Створення події у Google Calendar, за наявності підключення цього Google сервісу користувачем раніше. Через сервіс google_calendar_event_service створюється подія у його календарі.

Таким чином, бізнес-логіка повністю інкапсульована в CreateAppointmentUseCaseImpl і не залежить безпосередньо від Django або зовнішніх сервісів.

У Django-шарі, у класі AppointmentCreateView, за допомогою механізму ін'єкції залежностей викликається CreateAppointmentUseCaseImpl.

```
class AppointmentCreateView(LoginRequiredMixin, CreateView):
   model = Appointment
   form_class = AppointmentForm
   template_name = 'appointment/booking.html'
   redirect_field_name = reverse_lazy('login')

   @inject
   def post(
       self,
       request: HttpRequest,
       create_appointment_u_c: CreateAppointmentUseCase = Provide[
           'create_appointment_u_c'
       ],
       *args: Args,
```

```
        **kwargs: Kwargs,
    ) -> HttpResponseRedirect | HttpResponseBadRequest:
        appointment_form = self.get_form()
        appointment_form.is_valid()
        if not (location :=
appointment_form.cleaned_data.get('location')):
            return HttpResponseBadRequest('Missing required
parameter')
        create_appointment_u_c(
            appointment_input_data=AppointmentInputData(
                worker_id=WorkerID(int(self.kwargs['pk'])),
                location_data=LocationData(
                    location_id=LocationID(location.id),
address=(str(location))
                ),
                date_time=self.request.session.pop('date_time'),

customer_credentials=CustomerCredentialsModel.model_validate(
                    self.request.user
                ),
            ),
            time_zone=APPOINTMENT_TIME_ZONE_KYIV,
        )
        return
HttpResponseRedirect(reverse_lazy('appointment:appointments'))
```

Лістинг коду 3.8 – Представлення Django для створення запису до працівника

Послідовність дій у представленні:

- Отримання та валідація даних із форми бронювання.
- Формування об'єкта AppointmentInputData, який містить всю необхідну інформацію для створення запису.
- Виклик create_appointment_u_c із підготовленими даними та часовим поясом (APPOINTMENT_TIME_ZONE_KYIV).
- Перенаправлення користувача на сторінку перегляду записів після успішного створення.

Безпека доступу до об'єктів у веб-застосунку є критично важливою частиною будь-якого сучасного програмного забезпечення. У проєкті було реалізовано механізм перевірки прав доступу на рівні бізнес-логіки без прив'язки до стандартних Django permissions, що забезпечує більшу гнучкість і контроль.

Для централізованої перевірки прав доступу до об'єктів було створено окремий клас IsObjectOwner.

```
class IsObjectOwner:
   """Ensure that user is the owner of the object."""

   @staticmethod
   def has_object_permission(request: HttpRequest, obj: Any) ->
None:
       if obj.user != request.user:
           msg = 'You do not have permission to access this item.'
           raise PermissionDenied(msg)

   @staticmethod
   def ensure_customer_or_worker_access(request: HttpRequest, obj:
Any) -> None:
       user = request.user
       if user not in {obj.customer, obj.worker}:
           msg = 'You do not have permission to access this item.'
           raise PermissionDenied(msg)
```

Лістинг коду 3.9 – Клас перевірки прав доступу до об'єктів

Основні методи:

- has_object_permission

  Перевіряє, чи є поточний користувач власником об'єкта через атрибут user.

- ensure_customer_or_worker_access

  Забезпечує доступ тільки тим користувачам, які є або замовниками (customer), або виконавцями (worker) відповідного запису.

У разі порушення правил доступу викликається виняток PermissionDenied, що автоматично повертає відповідь із кодом 403 (Forbidden).

Механізм перевірки прав інтегрований безпосередньо у представлення, де необхідний контроль доступу до запису про зустріч.

Логіка:

- Після отримання об'єкта з бази даних через стандартний механізм Django super().get_object(), викликається метод IsObjectOwner.ensure_customer_or_worker_access.
- Якщо поточний користувач не є власником або працівником, який обслуговує цю зустріч, система повертає помилку доступу.

3.6 Тестування застосунку

Тестування є невід'ємною частиною процесу розробки програмного забезпечення, яка націлена на забезпечення якості, стабільності застосунку та перевіряє встановлені вимоги. Основна мета тестування полягає у виявленні та виправленні можливих помилок ще на етапі розробки, наприклад, тестування зменшує ризики поломки існуючого функціоналу при додаванні нового.

Функціональне тестування перевірило правильність реалізації основних функцій застосунку, таких як створення і видалення записів на прийом, забезпечення доступу до об'єктів тільки їх власниками, а також взаємодію з зовнішніми сервісами, наприклад, інтеграція з Google Calendar. Застосунок продемонстрував правильну роботу усіх основних сценаріїв без виявлення критичних помилок.

Навантажувальне тестування не проводилось у рамках цього етапу, однак архітектура застосунку побудована таким чином, щоб масштабування сервісу було можливим за допомогою стандартних засобів Django та розширення інфраструктури.

Процес тестування починається з аналізу вимог до застосунку, на основі яких формуються тестові сценарії. У нашому випадку було проведене умовне тестування, що включало як статичний аналіз коду, так і перевірку відповідності основним стандартам розробки. Для автоматизації процесу тестування використовувалися спеціалізовані інструменти, такі як Ruff, Bandit, що дозволило швидко виявити та усунути можливі проблеми на ранніх етапах [8, 9].

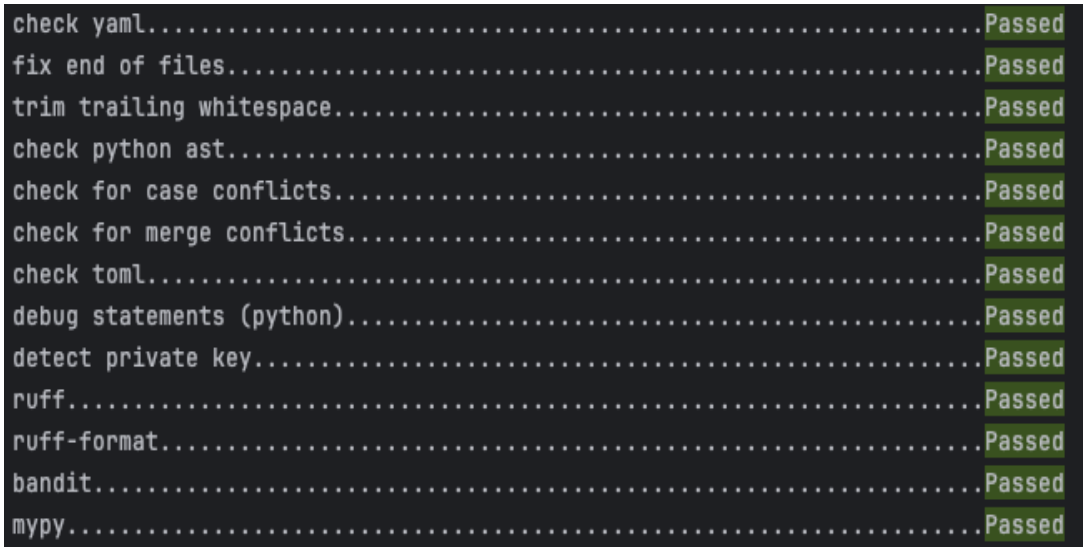


Рис. 3.1 – Результати проведених перевірок за допомогою інструментів аналізу коду

Результати проведених перевірок:

- check yaml: Перевірка правильності структури YAML-файлів у проєкті. Жодних помилок у форматуванні або структурі конфігурацій не виявлено.
- fix end of files: Всі файли завершуються одним порожнім рядком, як того вимагають правила оформлення коду.
- trim trailing whitespace: Вилучення зайвих пробілів у кінці рядків. Підтримка чистоти коду забезпечує кращу читабельність та запобігає виникненню непередбачуваних помилок при обробці файлів.
- check python ast: Перевірка на коректність синтаксису Python-файлів за допомогою побудови абстрактного синтаксичного дерева (AST). Всі файли успішно пройшли аналіз без синтаксичних помилок.
- check for case conflicts: Аналіз конфліктів найменувань файлів, які можуть виникати у файлових системах з чутливістю до регістру. Жодних проблем не виявлено.
- check for merge conflicts: У репозиторії відсутні залишки конфліктів після злиття гілок, що підтверджує правильність ведення історії змін у системі контролю версій.

- check toml: Перевірка правильності структури файлів конфігурації у форматі TOML, що застосовуються, зокрема, для налаштувань лінтерів та інструментів форматування.
- debug statements (python): Код не містить залишених операторів налагодження, що забруднюють код.
- detect private key: Було перевірено, що в коді або у файловій системі проєкту відсутні випадково закомічені приватні ключі.
- ruff: Лінтер ruff перевірив код на відповідність правилам PEP8, а також на інші поширені помилки. Всі знайдені раніше зауваження було виправлено.
- ruff-format: Весь код був автоматично відформатований за допомогою ruff-format, що забезпечує уніфікований стиль оформлення проєкту.
- bandit: Статичний аналізатор безпеки bandit не виявив критичних вразливостей у Python-коді. Особливу увагу було приділено безпечній роботі з даними користувачів та конфіденційною інформацією.
- mypy: Статична перевірка типів за допомогою mypy продемонструвала, що всі анотації типів коректні, типи змінних використовуються відповідно до очікувань, що зменшує ризики виникнення помилок під час виконання застосунку [10].

Таким чином, процес тестування підтвердив, що застосунок відповідає основним вимогам до якості, стабільності та безпеки. Статичний аналіз дозволили ефективно виявляти потенційні проблеми, а успішне проходження усіх перевірок свідчить про високу якість реалізації проєкту.

4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ ТА ОХОРОНА ПРАЦІ

4.1 Долікарська допомога при кровотечах

Надання домедичної допомоги при зовнішніх кровотечах є критично важливим етапом, що вимагає швидкого розпізнавання типу кровотечі, негайного застосування ефективних методів її зупинки та якомога швидшої передачі постраждалого до медиків.

Розуміння того, який тип кровотечі перед вами, допоможе правильно обрати метод допомоги:

- Артеріальна кровотеча: Це найнебезпечніший тип. Кров яскраво-червона і б'є сильним, пульсуючим струменем, синхронно з серцебиттям. Втрата крові може бути дуже швидкою.
- Венозна кровотеча: Кров темно-червона або вишневого кольору, витікає безперервно, рівномірним струменем, зазвичай без пульсації.
- Капілярна кровотеча: Проявляється повільним, рівномірним просочуванням крові по всій поверхні рани. Зазвичай не загрожує життю, якщо пошкодження невелике, але потребує обробки для запобігання інфікуванню.

Сучасні підходи та методи зупинки кровотечі. Пріоритетним є швидке та ефективне реагування, що відповідає міжнародним стандартам. Основні методи зупинки кровотечі включають:

- Прямий тиск на рану: Це першочерговий і найважливіший метод для зупинки більшості кровотеч. Сильно натисніть безпосередньо на джерело кровотечі чистою тканиною (марлевою серветкою, частиною одягу) або рукою в захисній рукавичці. Тиск має бути безперервним щонайменше 3-5 хвилин або до прибуття медиків чи накладання іншої пов'язки/турнікета.
- Накладання тиснучої пов’язки: Використовується для підтримання прямого тиску, особливо якщо ви надаєте допомогу самотужки. На рану кладуть стерильну або чисту серветку, а зверху туго накладають бинтову пов'язку.

- Тампонування рани: Цей метод ефективний при глибоких ранах і кровотечах з порожнин (наприклад, шия, пахви, пах), де неможливо накласти турнікет. Рана щільно заповнюється стерильним бинтом, гемостатичним бинтом або чистою тканиною. Після тампонування необхідно продовжувати прямий тиск на рану.
- Гемостатичні засоби: Це пов'язки або бинти, просочені спеціальними речовинами, що прискорюють згортання крові. Їх варто використовувати при тампонуванні, якщо вони є.
- Накладання кровоспинного турнікета (джгута): Застосовується виключно при масивній, загрозливій для життя, артеріальній кровотечі з кінцівок, коли прямий тиск та тампонування неефективні або неможливі.

Рекомендовано використовувати турнікети типу C-A-T або аналогічні сертифіковані вироби. Правила накладання: Накладається на кінцівку (плече, передпліччя, стегно, гомілка) на 5-7 см вище місця кровотечі. Затягнути до повної зупинки кровотечі, так, щоб пульс нижче турнікета не прощупувався. Обов'язково записати точний час накладання турнікета (на ньому самому або на видному місці на постраждалому). Турнікет не можна послаблювати або знімати до прибуття кваліфікованих медиків.

Дотримання базового алгоритму надання домедичної допомоги при зовнішній кровотечі збільшує шанси постраждалого на виживання:

- Переконайтеся у відсутності небезпеки для себе, оточуючих та постраждалого.
- Оцініть стан постраждалого: перевірте свідомість та дихання.
- Викличте бригаду екстреної медичної допомоги за номером 103.
- Одягніть засоби індивідуального захисту: медичні рукавички, за можливості захисні окуляри та маску.
- Огляньте місце кровотечі: Швидко визначте джерело та інтенсивність кровотечі. За необхідності, обережно розріжте або зніміть одяг над раною.

- При масивній кровотечі з кінцівки негайно накладіть турнікет. Запишіть час накладання.
- При інших кровотечах, або якщо турнікет не потрібен/недоступний:
- Здійсніть прямий тиск на рану руками.
- Якщо кровотеча продовжується, виконайте тампонування рани (пріоритетно гемостатичним бинтом).
- Після тампонування продовжуйте тиск на рану або накладіть тиснучу пов'язку.
- Забезпечте постраждалому протишокове положення: у горизонтальному положенні з піднятими ногами, якщо немає підозри на травму хребта, голови чи переломи кісток тазу/стегон.
- Вкрийте постраждалого для запобігання переохолодженню.
- Постійно контролюйте стан постраждалого (свідомість, дихання, ефективність зупинки кровотечі) до прибуття медиків.
- Будьте готові повторно застосувати методи зупинки кровотечі або накласти другий турнікет (вище першого), якщо кровотеча відновилася або продовжується.
- Передайте медичним працівникам всю важливу інформацію: обставини травми, тип кровотечі, вжиті заходи та особливо час накладання турнікета.

Дотримання цих правил допоможе врятувати життя та мінімізувати ризик ускладнень до моменту надання кваліфікованої медичної допомоги [11].

4.2 Заходи, що забезпечують рішення питань електробезпеки

Електробезпека – це система організаційних та технічних заходів і засобів, що забезпечують захист людей та майна від шкідливого і небезпечного впливу електричного струму, електричної дуги, електромагнітного поля та статичної

електрики. Дотримання вимог ДСТУ Б В.2.5-82:2016 є основою для проектування, монтажу та експлуатації безпечних електроустановок в офісних приміщеннях.

Основні технічні заходи захисту поділяються на ті, що забезпечують захист від прямого дотику, та ті, що забезпечують захист у разі пошкодження Розглянемо ключові з них.

Автоматичне вимикання живлення це основний захід захисту від непрямого дотику, а також захисту від пожеж, спричинених струмами замикання на землю, та захисту від перевантаження і короткого замикання.

Захист від перевантаження і короткого замикання здійснюється за допомогою автоматичних вимикачів або запобіжників. Вони розривають коло при перевищенні струмом заданого значення протягом певного часу або миттєво при короткому замиканні. Це запобігає перегріву провідників, пошкодженню обладнання та виникненню пожежі.

Захист від непрямих дотиків або пошкодження ізоляції реалізується за допомогою пристроїв захисного вимкнення (ПЗВ), також відомих як диференційні вимикачі або вимикачі диференційного струму (ВДС). ПЗВ реагує на появу диференційного струму у провідниках, що живлять електроустановку. Такий струм виникає, коли частина струму починає протікати поза основними провідниками, наприклад, через тіло людини, що доторкнулася до пошкодженого обладнання, або через пошкоджену ізоляцію на землю. ПЗВ забезпечує відключення живлення за долі секунди при досягненні струмом витоку небезпечного для людини значення (зазвичай 10 мА, 30 мА).

Існують також диференційні автоматичні вимикачі (дифавтомати), які поєднують функції АВ та ПЗВ в одному корпусі.

Стандарт ДСТУ вимагає, щоб час автоматичного відключення живлення не перевищував значень, встановлених для відповідних напруг та систем заземлення, щоб уникнути ураження людини при непрямому дотику.

Захисне заземлення призначене для усунення небезпеки ураження струмом у разі появи напруги на металевих неструмопровідних частинах обладнання,

наприклад, корпусах, внаслідок пошкодження ізоляції. Струм замикання в такому випадку піде в землю, що призведе до спрацювання пристроїв захисного відключення.

Електричне з'єднання струмопровідних частин для досягнення рівності їхніх потенціалів. Це запобігає виникненню небезпечної різниці потенціалів (напруги дотику) між одночасно доступними для дотику відкритими провідними частинами електрообладнання, сторонніми провідними частинами та землею.

Основна система зрівнювання потенціалів повинна з'єднувати між собою головний заземлювальний затискач, захисні провідники, заземлювальний провідник, металеві труби комунікацій (водопровід, опалення, газ – за узгодженням), металеві частини каркасу будівлі тощо.

Подвійна або посилена ізоляція. Забезпечення захисту від ураження електричним струмом без необхідності захисного заземлення відкритих провідних частин. Це захід захисту як від прямого, так і від непрямого дотику (у разі пошкодження основної ізоляції).

Подвійна ізоляція складається з основної ізоляції та додаткової ізоляції. Основна ізоляція забезпечує нормальну роботу обладнання та захист від прямого дотику. Додаткова ізоляція захищає у випадку пошкодження основної ізоляції.

Посилена ізоляція це єдина система ізоляції струмопровідних частин, яка забезпечує ступінь захисту від ураження електричним струмом, еквівалентний подвійній ізоляції.

Додаткові важливі заходи та аспекти:

- Правильний вибір та монтаж електрообладнання: Відповідно до умов експлуатації (вологість, температура, наявність агресивних середовищ).
- Кваліфікований персонал: Проєктування, монтаж, налагодження, експлуатація та ремонт електроустановок повинні виконуватися персоналом з відповідною кваліфікацією та групою з електробезпеки.
- Регулярні огляди та випробування: Періодична перевірка стану ізоляції, заземлювальних пристроїв, спрацювання захисної апаратури.

- Використання засобів індивідуального захисту (ЗІЗ): Діелектричні рукавички, килимки, інструмент з ізольованими ручками тощо при роботі в електроустановках.

Комплексне застосування зазначених заходів, ретельне проєктування відповідно до вимог ДСТУ Б В.2.5-82:2016 та інших чинних нормативних документів, якісний монтаж та належна експлуатація є запорукою створення безпечного технічного середовища. Це дозволяє мінімізувати ризики ураження електричним струмом для людей, запобігти пошкодженню обладнання та виникненню пожеж, забезпечуючи надійну та безпечну роботу електроустановок [12].

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було розроблено повнофункціональний веб-застосунок для пошуку та замовлення послуг на базі Python і Django, який інтегрує ключові компоненти сучасних онлайн-сервісів:

- багаторівневу аутентифікацію та розподіл ролей на клієнт або виконавець.
- бронювання зустрічей із автоматичною синхронізацією в Google Calendar.
- керування профілями, локаціями та графіками роботи виконавців.
- захищений механізм перевірки прав доступу до даних.
- чітке розділення бізнес-логіки, доступу до даних та інтеграційних модулів за допомогою сучасних патернів, як от Repository та DTO, і фреймворків, таких як Dependency Injection.

Запропонована архітектура та реалізація довели свою працездатність і зручність як для кінцевого користувача, так і для процесу розробки. Використання Poetry, pre-commit-hooks та статичного аналізу коду забезпечує високу якість та однорідність коду, а впровадження Dependency Injector підвищило тестованість та гнучкість застосунку.

У роботі деталізовано основні сценарії взаємодії — від створення та відміни записів до інтеграції з календарем і безпечного керування об’єктами. При цьому в межах проєкту було продемонстровано ключові сценарії використання, які демонструють фундаментальні можливості застосунку та підтверджують відповідність заявленим вимогам.

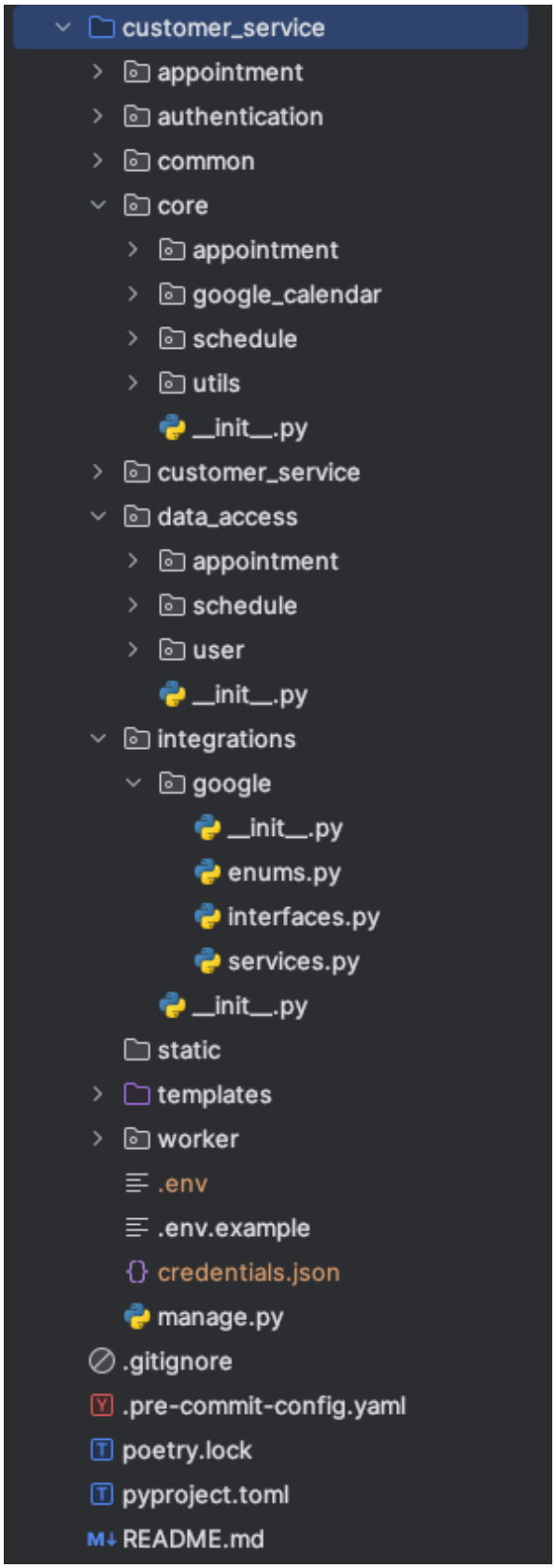
Отже, реалізований застосунок є міцною основою для подальшого розширення функціоналу, додавання нових сценаріїв використання та масштабування відповідно до потреб ринку послуг.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Django Software Foundation. Документація Django [Електронний ресурс]. – Режим доступу до ресурсу: https://docs.djangoproject.com/
2. Python Software Foundation. Документація Python 3.12 [Електронний ресурс]. – Режим доступу до ресурсу: https://docs.python.org/3/
3. Google Developers. Документація Google Calendar API [Електронний ресурс]. – Режим доступу до ресурсу: https://developers.google.com/calenda
4. Baptiste M. Документація Dependency Injector [Електронний ресурс]. – Режим доступу до ресурсу: https://python-dependency-injector.ets-labs.org/
5. Martin R.C. Чиста архітектура: посібник майстра зі структури та проєктування програмного забезпечення. – Prentice Hall, 2017. – 432 с.
6. Gamma E., Helm R., Johnson R., Vlissides J. Шаблони проєктування: елементи багаторазового об'єктно-орієнтованого програмного забезпечення. – Addison-Wesley, 1994. – 395 с
7. Freeman E., Robson E. Head First Design Patterns: створення розширеного та підтримуваного об’єктно-орієнтованого програмного забезпечення. – O'Reilly Media, 2020. – 704 с.
8. Bandit – статичний аналізатор безпеки для коду на Python [Електронний ресурс]. – Режим доступу до ресурсу: https://bandit.readthedocs.io/en/latest/
9. Ruff – надзвичайно швидкий лінтер для Python, написаний мовою Rust [Електронний ресурс]. – Режим доступу до ресурсу: https://docs.astral.sh/ruff/
10. Муру – статична типізація для Python [Електронний ресурс]. – Режим доступу до ресурсу: http://mypy-lang.org/
11. Про затвердження порядків надання домедичної допомоги особам при невідкладних станах. [Електронний ресурс] Міністерство охорони здоров’я України. – 2022. – Режим доступу: https://zakon.rada.gov.ua/laws/show/z0356-22
12. Желібо Є.П. Безпека життєдіяльності : підручник / В. В. Зацарний. Київ : Каравела, 2023. 344

ДОДАТКИ

ДОДАТОК А – Структура проєкту та її компоненти



ДОДАТОК Б – Лістинг коду веб-застосунку

Лістинг коду 1 – Абстракція сценарію використання RetrieveWorkerScheduleUseCase та її реалізація в сервісі RetrieveWorkerScheduleUseCaseImpl.

```
class RetrieveWorkerScheduleUseCase(Protocol):
   def __call__(
       self, worker_id: WorkerID, start_day: datetime =
timezone.now()
   ) -> WeeklySchedule:
       """Retrieves a weekly schedule for a given start day and
worker ID.

       Args:
           worker_id: The worker ID.
           start_day: The start day of week to retrieve weekly
schedule for.

       Returns:
           The weekly schedule for the given start day and worker
ID.

       """
class
RetrieveWorkerScheduleUseCaseImpl(RetrieveWorkerScheduleUseCase):
   __slots__ = ('appointment_reader_service',
'schedule_reader_service')

   def __init__(
       self,
       appointment_reader_service: BaseAppointmentReaderService,
       schedule_reader_service: BaseScheduleReaderService,
   ) -> None:
       self.appointment_reader_service = appointment_reader_service
       self.schedule_reader_service = schedule_reader_service

   def __call__(
       self, worker_id: WorkerID, start_day: datetime =
timezone.now().date()
   ) -> WeeklySchedule:
       return WeeklySchedule(
           previous_week_day=str(start_day -
timedelta(WEEKLY_SCHEDULE_DAYS)),
           next_week_day=str(start_day +
timedelta(WEEKLY_SCHEDULE_DAYS)),
           days=self._formate_daily_schedule(worker_id=worker_id,
start_day=start_day),
       )
   def _formate_daily_schedule(
```

```
        self, worker_id: WorkerID, start_day: date
    ) -> list[DailySchedule]:
        return [
            DailySchedule(
                date=str(current_day),

               weekday=current_day.strftime('%A').upper(),
                hours=self._formate_hours_list(
                    worker_id=worker_id, current_day=current_day
                ),
            )
            for day_offset in range(WEEKLY_SCHEDULE_DAYS)
            if (current_day := start_day +
timedelta(days=day_offset))
        ]

    def _formate_hours_list(self, worker_id: WorkerID, current_day:
date) -> list[str]:
        weekday_number = current_day.weekday()
        current_day_booked_hours = (

self.appointment_reader_service.get_worker_appointment_times_for_dat
e(
                worker_id=worker_id, date=str(current_day)
            )
        )
        worker_schedules_hours = (

self.schedule_reader_service.get_worker_schedules_hours_for_weekday(
                worker_id=worker_id, weekday_number=weekday_number
            )
        )
        return [
            hour
            for schedule_hours in worker_schedules_hours
            for hour in self._hour_list_generator(
                from_hour=schedule_hours.from_hour,
                to_hour=schedule_hours.to_hour,
                booked_hours=current_day_booked_hours,
            )
        ]
    @staticmethod
    def _hour_list_generator(
        from_hour: time, to_hour: time, booked_hours: list[str]
    ) -> Generator[str, None, None]:
        while from_hour <= to_hour:
            if (
                next_hour := from_hour.strftime(STR_FORMAT_HOURS)
            ) and next_hour not in booked_hours:
                yield next_hour
            from_hour = time(from_hour.hour + 1, from_hour.minute)
```

Лістинг коду 2 – Абстракція репозиторія BaseAppointmentReaderService та BaseAppointmentWriterService разом із їхніми реалізаціями в сервісах AppointmentReaderService та AppointmentWriterService

```
class BaseAppointmentReaderService(ABC):
   @abstractmethod
   def get_worker_appointment_times_for_date(
       self, worker_id: WorkerID, date: str
   ) -> list[str]:
       """Retrieves appointment times for specified worker ID and
date."""


class BaseAppointmentWriterService(ABC):
   @abstractmethod
   def create_appointment(self, appointment_data:
AppointmentDataType) -> None:
       """Creates appointment from provided appointment data."""

class AppointmentReaderService(BaseAppointmentReaderService):
   __slots__ = ('appointment_model',)

   def __init__(self, appointment_model: type[Appointment]) -> None:
       self.appointment_model = appointment_model

   def get_worker_appointment_times_for_date(
       self, worker_id: WorkerID, date: str
   ) -> list[str]:
       return [
           appointment.time.strftime(STR_FORMAT_HOURS)
           for appointment in self.appointment_model.objects.filter(
               worker=worker_id, date=date
           ).only('time')
       ]


class AppointmentWriterService(BaseAppointmentWriterService):
   __slots__ = ('appointment_model',)

   def __init__(self, appointment_model: type[Appointment]) -> None:
       self.appointment_model = appointment_model

   def create_appointment(self, appointment_data:
AppointmentDataType) -> None:

self.appointment_model.objects.create(**appointment_data.model_dump(
))
```

Лістинг коду 3 – Абстракція BaseGoogleCalendarEventService та її реалізація у сервісі GoogleCalendarEventService

```
class BaseGoogleCalendarEventService(ABC):
   @abstractmethod
   def create_event(
       self, user_google_credentials: dict[str, Any], event_data:
EventData
   ) -> None:
       """Creates event in user Google Calendar based on event data.

       Args:
           user_google_credentials: The dictionary with user google
calendar credentials data.
           event_data: The event data.

       """

class GoogleCalendarEventService(BaseGoogleCalendarEventService):
   __slots__ = ('google_service_factory',)

   def __init__(self, google_service_factory: GoogleServiceFactory)
-> None:
       self.google_service_factory = google_service_factory

   def create_event(
       self, user_google_credentials: dict[str, Any], event_data:
EventData
   ) -> None:
       event = {
           'summary': f'Customer service appointment -
{event_data.worker_name}',
           'location': event_data.address,
           'description': 'This is appointment to your providing
service',
           'start': {
               'dateTime':
f'{event_data.date}T{event_data.start_time.strftime("%H:%M:%S")}{eve
nt_data.time_zone}',
           },
           'end': {
               'dateTime':
f'{event_data.date}T{event_data.end_time.strftime("%H:%M:%S")}{event
_data.time_zone}',
           },
       }
       self.google_service_factory(
           user_google_credentials=user_google_credentials,
           google_service_name=GoogleServiceNameEnum.CALENDAR,
       ).events().insert(calendarId='primary', body=event).execute()
```

Лістинг коду 4 — Абстракція сценарію використання CreateAppointmentUseCase та його реалізація у CreateAppointmentUseCaseImpl.

```
class CreateAppointmentUseCase(Protocol):
   def __call__(
       self, appointment_input_data: AppointmentInputData,
time_zone: str
   ) -> None:
       """Creates appointment for user from input data.

       Args:
           appointment_input_data: The appointment input data to
create appointment for.
           time_zone: The time zone for the appointment.

       """
class CreateAppointmentUseCaseImpl(CreateAppointmentUseCase):
   __slots__ = (
       'appointment_writer_service',
       'google_calendar_event_service',
   )

   def __init__(
       self,
       user_reader_service: BaseUserReaderService,
       appointment_writer_service: BaseAppointmentWriterService,
       google_calendar_event_service:
BaseGoogleCalendarEventService,
   ) -> None:
       self.user_reader_service = user_reader_service
       self.appointment_writer_service = appointment_writer_service
       self.google_calendar_event_service =
google_calendar_event_service

   def __call__(
       self, appointment_input_data: AppointmentInputData,
time_zone: str
   ) -> None:
       date, time = appointment_input_data.date_time.split('/')
       self.appointment_writer_service.create_appointment(
           appointment_data=AppointmentDataType(
               worker_id=appointment_input_data.worker_id,
               customer_id=CustomerID(

appointment_input_data.customer_credentials.user_id
               ),

location_id=appointment_input_data.location_data.location_id,
               date=date,
               time=time,
           )
```

```
        )
        self._create_google_calendar_event(
            time_zone=time_zone,
            date=date,
            time=time,
            appointment_input_data=appointment_input_data,
        )

    def _create_google_calendar_event(
        self,
        time_zone: str,
        date: str,
        time: str,
        appointment_input_data: AppointmentInputData,
    ) -> None:
        if (

appointment_input_data.customer_credentials.google_calendar_credenti
als
            is not None
            and (
                worker_name :=
self.user_reader_service.get_worker_name(
                    worker_id=appointment_input_data.worker_id
                )
            )
        ):
            start_time, end_time =
self._prepare_appointment_hours(time=time)
            self.google_calendar_event_service.create_event(

user_google_credentials=appointment_input_data.customer_credentials.
google_calendar_credentials,
                event_data=EventData(
                    worker_name=worker_name,

address=appointment_input_data.location_data.address,
                    date=date,
                    start_time=start_time,
                    end_time=end_time,
                    time_zone=time_zone,
                ),
            )

    @staticmethod
    def _prepare_appointment_hours(time: str) -> tuple[datetime,
datetime]:
        start_time = datetime.strptime(time, STR_FORMAT_HOURS)
        end_time = start_time + timedelta(hours=1)
        return start_time, end_time
```

ДОДАТОК В – Диск із кваліфікаційною роботою бакалавра