Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

Розробка UI/UX особистого кабінету викладача в системі управління університетом

Текстова частина до курсової роботи
за спеціальністю “Інженерія програмного забезпечення”

Керівник курсової роботи
доктор технічних наук, доцент,
Глибовець А. М.

_____
(підпис)
“___” _____ 2024 р.
Виконав студент Давидов Д. О.
“___” _____ 2025 р.

Київ 2025

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри інформатики,
доцент, кандидат наук
_____ Гороховський С.С
(підпис)
“____” _____ 2025 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту Давидову Денису факультету інформатики 3 курсу

ТЕМА: Розробка UI/UX особистого кабінету викладача в системі управління університетом

Зміст ТЧ до курсової роботи:

Індивідуальне завдання
Вступ

1. Аналіз предметної області та постановка задачі
2. Особливості інтеграції кабінету викладача
3. Реалізація кабінету викладача

Висновок
Список використаних джерел
Додатки

Дата видачі “___” _____ 2025 р.
Керівник _____ Завдання отримав _____

Календарний План Виконання Курсової Роботи

Тема: Розробка системи управління освітніми програмами університету

№ з/п	ПЕРЕЛІК РОБОТ	Термін Виконання	Примітка
1	Отримання теми кваліфікаційної роботи	Листопад 2024	
2	Створення плану роботи	Кінець грудня 2024	
3	Ознайомлення з проєктом	Січень 2025	
4	Розробка практичної частини проєкту	Січень – Квітень 2025	
5	Написання першого розділу роботи	Початок квітня 2025	
6	Описання практичної частини роботи	Середина квітня 2025	
7	Перегляд змісту роботи з керівником	Початок травня 2025	
8	Внесення змін відповідно до зауважень наукового керівника	Початок травня 2025	
9	Створення презентації	Початок травня 2025	
10	Захист дипломної роботи	13.05.2024	

Студент Давидов Д. О. _____

Керівник Глибовець А.М. _____

“_____” _____ 2025 р.

ЗМІСТ

АНОТАЦІЯ

Курсова робота присвячена проєктуванню та реалізації особистого кабінету викладача в інформаційній системі SMART Ukma. Метою роботи є створення інтерфейсу, який дозволяє зручно керувати персональними, освітніми та науковими даними викладача.

У першому розділі розглянуто основи UX/UI, їхню роль у створенні ефективних інтерфейсів, а також сучасні підходи до дизайну в контексті цифрових освітніх систем.

У другому розділі проаналізовано цілі, функціональні можливості та структуру особистого кабінету викладача, його значення в межах SMART Ukma.

У третьому розділі описано технічні аспекти реалізації: архітектуру фронтенду, принцип єдиної відповідальності, використання Angular, Angular Material, Reactive Forms, RxJS та інтеграцію з бекендом через OpenAPI Generator. Також детально розглянуто підхід до форм, валідації та організації структури застосунку.

Ключові слова: SMART Ukma, UX/UI, особистий кабінет викладача, Angular, веб-інтерфейс, валідація, цифрова освіта.

ВСТУП

У сучасних умовах цифровізації вищої освіти особливого значення набуває створення інформаційних систем, які забезпечують ефективне управління освітніми процесами, кадровими ресурсами та адміністративними процедурами. Одним із ключових елементів таких систем є особистий кабінет викладача — цифровий інтерфейс, що забезпечує доступ до актуальної професійної інформації, дозволяє вносити зміни, а також автоматизує значну частину документообігу.

Наразі у багатьох університетах використовуються загальні або запозичені рішення, як-от платформи Moodle, які мають обмежену гнучкість у частині адміністрування академічної інформації викладачів. Системи, створені на базі сторонніх продуктів, не завжди повною мірою враховують специфіку внутрішніх процесів закладів вищої освіти. Відсутність зручного і зрозумілого інтерфейсу, дублювання інформації у різних підсистемах, складність у підтримці актуальності даних — усе це створює додаткове навантаження як для працівників, так і для адміністративного персоналу.

У цьому контексті актуальним є створення індивідуального рішення, яке дозволить викладачам самостійно керувати власною інформацією: додавати дані про освіту, публікації, досягнення, наукові ступені, заповнювати профіль тощо. Такий підхід сприяє децентралізації управління даними, підвищенню прозорості та ефективності внутрішньої взаємодії, а також зменшенню адміністративного навантаження на інші підрозділи

Курсова робота присвячена розробці фронтенд-частини особистого кабінету викладача в рамках інформаційної системи Smart Ukma. У межах цього проєкту розглядаються питання проєктування архітектури клієнтської частини, реалізації форм для внесення й редагування даних, побудови інтерфейсу на основі сучасних підходів до UX/UI та забезпечення інтеграції з серверною

частиною засобами OpenAPI Generator. Робота охоплює як структурну побудову інтерфейсу, так і його візуальну реалізацію відповідно до сучасних стандартів.

Новизна розробки полягає у поєднанні типових вимог до цифрового профілю викладача з використанням сучасних інструментів фреймворку Angular, застосуванням принципів модульності, розділення відповідальності та реактивного програмування. Результати цієї роботи можуть бути безпосередньо використані для масштабування функціональності системи Smart Ukma, а також слугувати основою для подальшої автоматизації академічних процесів у закладах вищої освіти.

Призначення курсової роботи полягає у реалізації функціонального, доступного та масштабованого особистого кабінету викладача, що відповідає потребам сучасного університетського середовища та сприяє підвищенню цифрової зрілості академічної спільноти.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1. Поняття UX/UI

У сучасних цифрових системах, особливо у сфері вищої освіти, проєктування інтерфейсу користувача має вирішальне значення для ефективності та прийнятності продукту. Від того, наскільки зручно та інтуїтивно організована взаємодія користувача із системою, залежить частота її використання, рівень довіри до неї та якість введених даних. Це особливо актуально у випадку таких модулів, як особистий кабінет викладача, де користувачі часто не є фахівцями з ІТ, але активно взаємодіють із інформаційною системою. Підходи, до взаємодії з користувачами мають назву – UX та UI. UX (user experience, користувацький досвід) - орієнтується на взаємодію, яку має користувач із повсякденними продуктами та послугами. Мета UX дизайну – зробити продукти чи послуги, простими, логічними та веселими. UI (user interface, користувацький інтерфейс) - це підхід, орієнтований на користувача, для проектування естетики цифрового продукту [1]. По суті, вони створюють зовнішній вигляд користувальницького інтерфейсу веб-сайту чи програми.

Підхід до побудови інтерфейсу повинен ґрунтуватися не лише на візуальній естетиці, а й на глибокому розумінні потреб, можливостей та обмежень користувача. Основною метою такого проєктування є створення середовища, у якому користувач швидко орієнтується, може без труднощів виконувати необхідні дії та не відчуває фрустрації через надмірну складність або непослідовність взаємодії із системою.

У сучасному проєктуванні інтерфейсів використовуються перевірені принципи, що забезпечують логічність, простоту та ефективність взаємодії. До таких принципів належать консистентність, передбачуваність, зворотний зв’язок, мінімізація когнітивного навантаження та адаптивність до різних сценаріїв

використання. Послідовність відіграє ключову роль: елементи інтерфейсу повинні бути стандартизованими в межах усієї системи. Якщо, наприклад, кнопка «Зберегти» завжди розташована в одному і тому ж місці на всіх сторінках, а сповіщення мають однаковий вигляд і колірне оформлення — користувач адаптується до системи значно швидше.

Іншим важливим аспектом є видимість стану системи. Кожна дія користувача має супроводжуватись зворотним сигналом — індикатором завантаження, повідомленням про успіх або помилку, візуальними підказками. Такі механізми формують довіру до системи, створюють відчуття передбачуваності та запобігають повторним або помилковим діям. Гнучкість взаємодії передбачає надання можливості як швидкої роботи (через шаблони, автозаповнення, контекстні підказки), так і поетапного введення даних у зручному темпі — залежно від досвіду користувача.

1.2. Гарні практики побудови інтерфейсів

До перевірених практик, що довели свою ефективність у різноманітних цифрових продуктах, належить чітке візуальне структурування сторінок. Воно включає поділ на блоки, логічне групування пов’язаних полів, використання вкладок або акордеонів для зменшення візуального навантаження. Така організація полегшує навігацію, особливо на сторінках з великою кількістю інформації, як-от особистий кабінет викладача.

Єдність кольорової палітри, типографіки та іконографіки допомагає сформувати цілісне і передбачуване середовище. Стилістична послідовність — один з важливих аспектів візуальної ієрархії, що формує комфорт користувача при роботі з системою. Успішною практикою також є виведення контекстних підказок та прикладів введення даних, що зменшує кількість помилок та підвищує впевненість у правильності виконуваних дій.

Інтерфейс не повинен містити зайвих елементів: лише ті поля й функції, які потрібні на конкретному етапі. Водночас система повинна надавати можливість перегляду та редагування даних у розрізі одного інтерфейсу без необхідності переходу на інші сторінки, що значно скорочує кількість дій і підвищує ефективність [2].

Особливо ефективними є інтерфейси з адаптивною поведінкою. Наприклад, якщо користувач обирає тип досягнення, то система динамічно змінює форму, показуючи лише релевантні поля. Це не лише економить простір, а й зменшує вірогідність помилки. У реалізації особистого кабінету викладача такі практики втілено через багаторівневу навігацію, редагування без зміни маршруту, використання акордеонів, а також ретельно продуману систему валідації.

1.3. Інструменти для проєктування та реалізації UX/UI

Створення зручного та ефективного інтерфейсу передбачає не лише дотримання теоретичних принципів, а й використання практичних інструментів, які забезпечують реалізацію, перевірку та вдосконалення інтерфейсів. Залежно від етапу розробки (прототипування, реалізація, тестування), використовуються різні програмні засоби.

На етапі створення концепції та структури інтерфейсу важливу роль відіграють інструменти прототипування. Найпоширенішим серед них є Figma — потужний інструмент для спільного проектування для команд. Він допомагає створювати реалістичні прототипи та оптимізуйте розробку продукту за допомогою систем проектування [3]. Figma дає змогу моделювати переходи, демонструвати макети замовникам і координувати роботу з командою. Подібні функції також реалізовані в Adobe XD та Sketch, які популярні серед фахівців UI-дизайну.

В освітніх проєктах, де графічне моделювання може бути мінімальним або здійснюватися паралельно з розробкою, Figma часто використовується як засіб координації між дизайнерами, розробниками та замовниками. Саме таким чином вона застосовувалась у межах створення інтерфейсу кабінету викладача в SMART Ukma.

Для реалізації UI-компонентів використовуються засоби фреймворку Angular, зокрема Angular Material, який забезпечує доступ до уніфікованих компонентів інтерфейсу: форм, кнопок, повідомлень, таблиць. У поєднанні з реактивними формами та кастомними компонентами ці інструменти дозволяють забезпечити високий рівень інтерактивності та узгодженості інтерфейсу.

Під час реалізації особистого кабінету викладача як частини системи SMART Ukma застосовується Angular — сучасний фронтенд-фреймворк, який підтримує побудову компонентно-орієнтованої архітектури, типізовану логіку та ефективну роботу з формами. У проєкті використовуються такі інструменти:

Angular Material — набір UI-компонентів, побудованих відповідно до принципів Material Design. Забезпечує єдиний стиль інтерфейсу, підтримку адаптивності, готові компоненти (вкладки, форми, діалоги, таблиці) та гнучку візуальну настройку [4].

Reactive Forms — забезпечують керований підхід до обробки вхідних даних форми, значення яких змінюються з часом [5].

RxJS — бібліотека для реактивного програмування, яка базується на концепції реактивного програмування та дозволяє зручно керувати асинхронними потоками даних (наприклад, HTTP-запитами, подіями введення) [6].

Уся візуальна логіка організована у вигляді окремих Angular-компонентів, що дає змогу чітко розмежовувати відповідальність, повторно використовувати елементи та адаптувати інтерфейс до різних категорій користувачів.

1.4. Типові помилки та небажані рішення

Неструктурований або неусвідомлений підхід до розробки UX/UI може суттєво знизити ефективність використання веб-застосунку, незалежно від технічної складності або функціонального наповнення. У випадку освітніх платформ, таких як особистий кабінет викладача, це має ще більшу вагу, оскільки основними користувачами є не фахівці з ІТ, а викладачі, які очікують логіки, простоти та передбачуваності у взаємодії з системою.

Однією з поширених помилок є перевантаження форм великою кількістю обов’язкових полів без відповідних пояснень. Це створює ситуацію, коли користувач не розуміє, чому саме він має заповнити те чи інше поле, і до чого може призвести його ігнорування. В результаті виникає розгубленість, фрустрація та небажання користуватись системою повторно. Оптимальний UX передбачає надання контекстних підказок і логічне групування полів, щоб процес заповнення не виглядав перевантаженим або хаотичним.

Ще однією суттєвою помилкою є використання технічної термінології без адаптації для кінцевого користувача. Якщо, наприклад, у полі потрібно вказати "ідентифікатор навчального ресурсу", але користувач не має уявлення, що це означає і де його взяти — така форма стає бар’єром, а не інструментом. Правильна практика полягає в заміні або поясненні термінів через приклади, тултіпи чи альтернативні назви, які є зрозумілими з контексту.

Важливою, але часто недооціненою проблемою є відсутність повідомлень про результати дій користувача. Якщо система не повідомляє, що дані успішно збережено, або не вказує на помилку в разі невдалої операції, користувач може кілька разів натискати кнопку «Зберегти», вважаючи, що система не відреагувала. Це породжує дублікати, викривлення даних і недовіру до платформи.

Окрему загрозу для зручності становить слабка або відсутня адаптація до мобільних пристроїв. У сучасних умовах користувачі дедалі частіше взаємодіють із системами через смартфони або планшети, особливо в освітньому середовищі. Якщо особистий кабінет не масштабований або має елементи, що «випадають» за межі екрану, це обмежує коло користувачів і створює враження застарілого продукту.

Нарешті, недотримання внутрішньої логіки в розміщенні елементів інтерфейсу — ще одна критична помилка. Якщо кнопки або блоки змінюють своє розташування залежно від сторінки, користувач змушений щоразу адаптуватися до нових умов, що збільшує когнітивне навантаження. Особливо шкідливими є ситуації, коли елементи, схожі за виглядом, виконують зовсім різні функції. Це суперечить принципу передбачуваності інтерфейсу й ускладнює навігацію, особливо для нових користувачів.

1.5. Порівняння skeuomorphism та flat design підходів

Історичний розвиток підходів до побудови інтерфейсів демонструє поступовий перехід від візуально насичених стилів до мінімалізму та функціональності. Одним із ключових прикладів такої трансформації є зміна парадигми від скевоморфного дизайну (skeuomorphism) до плоского дизайну (flat design), що значно вплинуло на сучасні практики UX/UI.

Скевоморфізм передбачає використання в інтерфейсі візуальних елементів, які імітують реальні об’єкти, з усіма їхніми фізичними властивостями: тінями, об’ємами, текстурами [7]. Такі рішення допомагали користувачам, не знайомим із цифровими технологіями, легше зрозуміти, як працює інтерфейс. Наприклад, блокнот виглядав як реальний блокнот із лініями, а кнопки — як випуклі поверхні, які можна «натиснути». Цей підхід широко використовувався у 2000-х роках, зокрема в ранніх версіях операційної системи iOS.

Втім, із розвитком цифрової грамотності та поширенням мобільних пристроїв акценти змістилися. Поява flat design стала відповіддю на зростаючу потребу у простих, адаптивних і технічно легких інтерфейсах. Плоский дизайн відмовляється від декоративності на користь чіткої структури, контрастної типографіки, простих кольорів і логічного розміщення елементів [8]. Такий стиль зменшує візуальне навантаження на користувача, забезпечує швидше завантаження сторінок і краще масштабування на різних пристроях. Прикладами систем, що активно використовують flat design, є Google Material Design та Microsoft Fluent UI.

Сьогодні flat design став стандартом для більшості цифрових платформ. Його популярність пояснюється не лише технічними перевагами, а й високим рівнем доступності: елементи стають зрозумілими незалежно від рівня підготовки користувача. Крім того, мінімалістичний підхід полегшує реалізацію адаптивного дизайну, що є особливо важливим у контексті роботи з мобільними пристроями.

1.6. Тестування зручності інтерфейсу

Після розробки користувацького інтерфейсу особливо важливо перевірити, наскільки він справді зручний, зрозумілий і ефективний для кінцевого користувача. Тестування зручності (usability testing) — це процес вивчення того, як реальні користувачі взаємодіють із системою в умовах, наближених до реального використання [9]. Його метою є виявлення бар’єрів у взаємодії, помилок, надмірного навантаження чи фрустрації.

Одним із найпоширеніших підходів є спостереження за користувачем (user observation). У цьому форматі користувачеві пропонують виконати типові завдання (наприклад, внести публікацію або змінити особисті дані), а дослідник спостерігає за діями, фіксує труднощі, помилки або неочікувану поведінку. Цей

метод є ефективним навіть при невеликій кількості учасників, адже дозволяє виявити найпоширеніші UX-проблеми.

Іншим популярним методом є опитування користувачів (user surveys), зокрема анкетування після використання системи. У кабінеті викладача можливим є опитування щодо задоволеності інтерфейсом, зрозумілості форм, складності навігації або відчуття впевненості при роботі. Застосовуються шкали від 1 до 5 або відкриті запитання. Отримані дані дозволяють оцінити суб'єктивне враження та пріоритетність проблем.

Технічно складнішим, але дуже показовим методом є аналіз теплових карт (heatmaps). За допомогою відповідного ПЗ (наприклад, Hotjar, Microsoft Clarity) візуалізуються дані, які показують, як користувачі веб-сайту натискають, прокручують та переміщують у межах сторінки [10]. Такі дані дозволяють виявити «мертві зони» інтерфейсу, неправильно розташовані елементи або поля, які користувачі ігнорують. У випадку SMART Ukma цей метод може бути застосований у майбутньому для оптимізації інтерфейсу на основі реальних сценаріїв використання.

Також використовуються юзабіліті-тестування з метриками, такими як час на виконання завдання, кількість кліків, кількість помилок. Наприклад, якщо середній час на додавання досягнення перевищує допустиму межу — це сигнал, що частину інтерфейсу слід спростити або перебудувати.

Загалом, тестування зручності є важливою частиною життєвого циклу інтерфейсу. Воно дозволяє переходити від припущень до фактичного розуміння потреб користувачів і забезпечує постійне вдосконалення системи на основі реальних даних.

1.7. Актуальність UX/UI для освітніх цифрових платформ

У випадку університетських інформаційних систем UX/UI не є другорядним елементом, а структурною частиною системи управління. Від зручності інтерфейсу залежить рівень заповненості особистих кабінетів, точність внесених даних, своєчасність оновлення, а також загальне ставлення працівників до цифрових інструментів в університеті.

Якщо система сприймається як складна та незрозуміла — вона фактично перестає виконувати свої функції, незалежно від технічного рівня реалізації. Саме тому при створенні особистого кабінету викладача акцент було зроблено не лише на функціональність, а й на відповідність базовим принципам UX/UI-дизайну.

2. ОСОБЛИВІСТЬ ІНТЕГРАЦІЇ КАБІНЕТУ ВИКЛАДАЧА

2.1. Визначення кабінету викладача

Особистий кабінет викладача у складі системи SMART Ukma є одним із ключових модулів внутрішньо-цифрового середовища університету. Його основне призначення - надати кожному викладачу індивідуальний простір для ведення, оновлення та перегляду власної академічної інформації, а також забезпечити адміністративним підрозділам централізований доступ до актуальних, структурованих і верифікованих даних про викладацький склад.

Система SMART Ukma - це комплексна інформаційна платформа, яка охоплює всі основні напрями університетського управління: освітній процес, кадрові ресурси, документообіг, моніторинг, аналітику, акредитаційну звітність. У межах цієї системи особистий кабінет викладача виконує функції цифрового інтерфейсу взаємодії між працівником університету та адміністративною інфраструктурою.

Цей модуль має на меті стати інструментом самообслуговування та персонального управління. Він дає можливість викладачеві не лише переглядати внесені адміністративні відомості, а й самостійно ініціювати додавання чи оновлення даних, що стосуються його академічної кар’єри. Таким чином, зменшується кількість проміжних ланок у взаємодії між користувачем і структурними підрозділами (кадровими, навчальними, аналітичними), що підвищує оперативність, знижує навантаження на адміністрацію та мінімізує ризики дублювання або втрати даних.

2.2. Порівняння з існуючим рішенням

На момент впровадження системи SMART Ukma вже функціонувала окрема платформа для організації навчального процесу - система дистанційної освіти distedu.ukma.edu.ua, побудована на основі Moodle. Moodle (Modular Object-Oriented Dynamic Learning Environment) - це модульне об'єктно-орієнтоване динамічне навчальне середовище, яке називають також системою управління навчанням (Learning Management System, LMS), призначена для створення, організації та супроводу освітніх онлайн-курсів або дистанційного навчання [11]. Однак функціональні можливість поточної платформи, розробленої на базі Moodle – обмежені. У distedu.ukma.edu.ua викладач може створити профіль, однак він є переважно формальним і не охоплює жодних елементів кар’єрного або кадрового обліку. Такі розділи, як інформація про освіту, наукові ступені, публікації, професійні досягнення, підвищення кваліфікації, — відсутні. У такому форматі, альтернатив кабінету викладача в Smart Ukma немає, а отже, користь від подібного нововведення – максимальна.

2.3. Основні цілі та значення

Централізація інформації в межах єдиного цифрового середовища має вирішальне значення для ефективного функціонування сучасного університету. Завдяки особистому кабінету досягаються такі цілі:

- Упорядкування даних про кадровий склад, його кваліфікацію, наукову активність, участь у проєктах тощо.
- Забезпечення достовірності інформації, оскільки викладач сам відповідає за її актуальність.
- Зменшення бюрократичного навантаження, оскільки основні дії (редагування, підтвердження, завантаження документів) виконуються без залучення додаткових посадових осіб.

Окрім практичного значення для інституційного адміністрування, кабінет викладача виконує також репрезентативну функцію — він формує цифровий академічний профіль, що може бути використаний для внутрішнього позиціонування, публічних рейтингів, подання на гранти, участі у конкурсах тощо.

2.4. Основні функції кабінету

Особистий кабінет реалізує широкий спектр можливостей, які охоплюють різні напрями професійної та освітньої діяльності викладача. До функціонального переліку входить:

- Перегляд і редагування персональних даних, таких як ПІБ, посада, науковий ступінь, вчене звання, факультет/кафедра, службові контакти, біографічна довідка, а також додаткова інформація - військовий облік, склад сім'ї, інвалідність, соціальний статус.
- Облік освіти, включаючи заклади вищої освіти, спеціальності, кваліфікації, серії та номери дипломів, шифри та коди спеціальностей.
- Фіксація наукових ступенів і вчених звань - як українських, так і міжнародних - із зазначенням тем дисертацій, дат захисту, наказів та інших офіційних даних.
- Внесення публікацій - наукових, методичних, публіцистичних, із можливістю вказати підтверджуючі дані, наукову платформу, прикріпити електронну версію або посилання.
- Документування підвищення кваліфікації - із зазначенням форми навчання, тривалості, кількості кредитів, організації, що проводила навчання, та результатів (сертифікатів, рішень вчених рад).

- Формування переліку інших професійних досягнень, таких як участь у грантах, міжнародна мобільність, нагороди, конкурси, членство в експертних комісіях, редакційних колегіях, професійних спілках.
- Завантаження документів, що підтверджують внесену інформацію: дипломи, сертифікати, витяги з наказів, результати курсів, скани дипломів тощо.

Інтерфейс кабінету побудовано з урахуванням потреб користувачів, які не є технічними фахівцями. Він містить вкладки за тематичними напрямами, структуровані форми, динамічні повідомлення про успіх чи помилки, елементи акордеонів для згортання/розгортання інформації, а також автоматичну валідацію введених даних. Усі ці рішення сприяють покращенню користувацького досвіду, зменшенню кількості помилок при внесенні інформації та загальному підвищенню довіри до системи.

2.5. Візуальна організація інтерфейсу кабінету викладача

Візуальний вигляд особистого кабінету викладача в системі SMART Ukma є прикладом структурованого, сучасного та функціонально орієнтованого інтерфейсу, що відповідає базовим принципам UX/UI-дизайну. Інтерфейс створено з урахуванням потреб звичайних користувачів, а саме — академічного персоналу, який працює з системою не щодня, але очікує логічної структури, простоти дій та зрозумілої навігації.

На рисунку 2.1 показан кабінет викладача, який маєвкладкову організацію, де кожен розділ (загальні дані, дисципліни, публікації, освіта, досягнення тощо) винесено у вигляді окремої горизонтальної вкладки у верхній частині сторінки. Така модель дозволяє логічно розділити ресурси за необхідністю по блоках, забезпечує швидкий доступ до конкретної категорії

інформації та знижує навантаження на користувача, оскільки одночасно відображається лише релевантна частина даних.

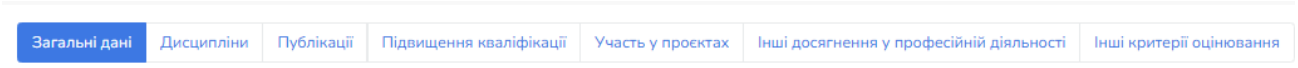


Рисунок 2.1 – Горизонтальні вкладки кабінету викладача

Візуальна ієрархія інтерфейсу особистого кабінету викладача реалізована на основі чітких і послідовних принципів. Кожен підрозділ має логічну структуру, яка забезпечує швидке орієнтування в межах розділу. Заголовки відображаються напівжирним шрифтом і візуально виокремлюються з основного тексту, що дозволяє одразу зчитати назву блоку або розділу. Вторинна інформація, наприклад, частка ставки, ідентифікатори акаунтів у наукових мережах чи службові ID, подається сірим або зменшеним шрифтом, що можна побачити на рисунку 2.2. Це дає змогу зберегти акцент на основних даних, водночас залишаючи допоміжну інформацію доступною. Межі між функціональними блоками чітко відокремлено за допомогою візуальних контейнерів (карток), відступів, ліній розділення та акордеонів, які дозволяють згортати або розгортати вміст за потреби. Такий підхід створює ієрархічну читабельність, навіть якщо в одному розділі зібрано десятки записів — наприклад, освіти чи публікацій.

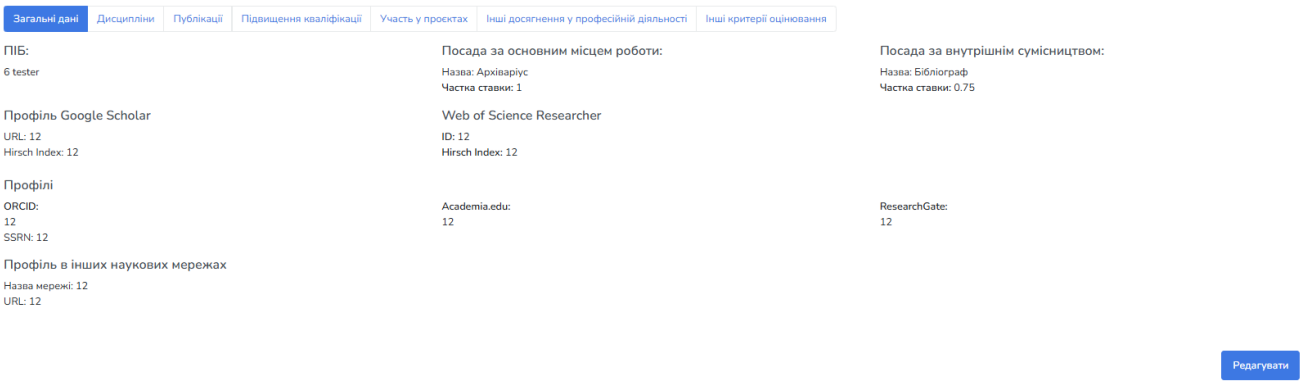


Рисунок 2.2 – Вигляд викладацької картки кабінету викладача

Елементи взаємодії з формами додавання та редагування інформації реалізовані із застосуванням компонентів Angular Material, таких як mat-form-field, mat-select, mat-datepicker та стандартні кнопки. Інтерфейс забезпечує інформативне маркування: у разі некоректного введення або незаповнених обов’язкових полів користувач бачить підсвічування поля червоним і пояснювальне повідомлення (наприклад «Поле обов’язкове»). Також передбачено єдину логіку взаємодії з діалоговими вікнами, які відкриваються для додавання нової інформації: у кожному з них присутні кнопки «Зберегти» та «Скасувати», що виведені в одному стилі та на одному місці, завдяки чому взаємодія з різними формами є передбачуваною і не викликає дезорієнтації.

Поведінка інтерфейсу при заповненні форм також демонструє відповідність сучасним вимогам до UX. Обов’язкові для заповнення поля чітко позначені зірочкою (*), а у разі їх ігнорування користувач негайно отримує візуальний зворотний зв’язок у вигляді рамки та тексту з помилкою, як вказано на рисунку 2.3. Вкладена структура дозволяє уникнути перевантаження інтерфейсу: при великій кількості блоків (наприклад, кількох записів про освіту або досягнення) користувач бачить лише заголовки з можливістю розгорнути зміст кожного з них. Додатково, вибір типів записів реалізовано через випадаючі списки (селектори), що не лише спрощує введення, а й запобігає орфографічним помилкам, варіативності формулювань і порушенню стандартизованої структури даних.

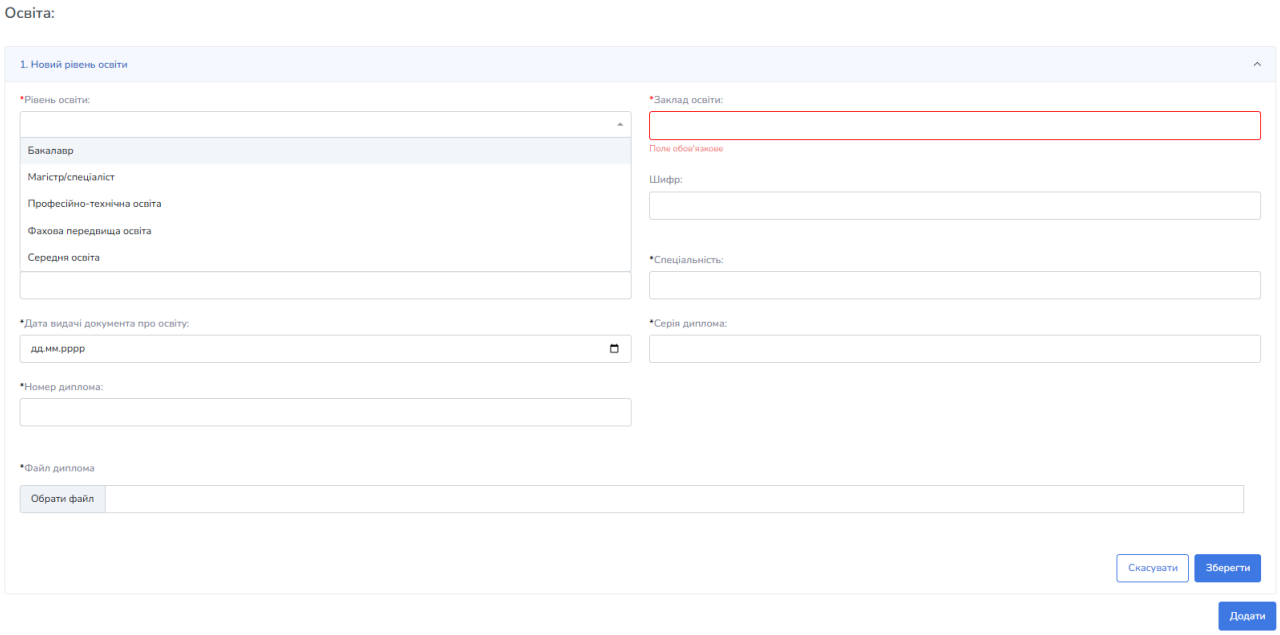


Рисунок 2.3 – Форма нового рівня освіти з усіма вдалими поведінками інтерфейсу

Реалізований візуальний інтерфейс поєднує зрозумілість, формальну строгість і технічну доступність, що відповідає специфіці цільової аудиторії — викладачів, які потребують швидкої, надійної та прогнозованої взаємодії з цифровими формами університетського середовища.

Усі елементи інтерфейсу витримані в єдиному стилі: шрифт, розміри, відступи, кнопки мають стандартизоване оформлення. Кольорова палітра — стримана, професійна (переважають білі та сині тони), що відповідає академічному контексту. Використання іконок, модальних діалогів і спливаючих повідомлень забезпечує додаткову інформативність і формує позитивне враження про систему.

3. РЕАЛІЗАЦІЯ КАБІНЕТУ ВИКЛАДАЧА

3.1. Архітектурні виклики та використання SRP

При написанні будь-якої комплексної роботи треба продумати правильне проєктування та використання ресурсів. Еволюція розробки програмного забезпечення дійшло до висновку, що принцип єдиної відповідальності має багато переваг при розробці великих сервісів. У технічній літературі такий підхід має назву - Single Responsibility Principle - клас повинен мати лише один обов’язок і лише одну причину для зміни [12].

Подібна технологія була й використана під час розробки програми. Створений клас має мати лише одну задачу, це може бути або перетворення вхідного значення після деяких маніпуляцій в інше, або виконання конкретно дії, пов’язаної з інтерфейсом чи обробкою даних. Наприклад, у рамках цього проєкту було реалізовано pipe OtherAchievementsTypeI18Pipe, який відповідає лише за одну задачу перетворення ключа типу наукового досягнення на відповідний текст для відображення в інтерфейсі (рисунок 3.1). Цей pipe не містить жодної сторонньої логіки, не залежить від зовнішніх сервісів і може бути повторно використаний у будь-якому компоненті. Таким чином, він ілюструє чітке дотримання принципу єдино відповідальності: має одну точку зміни лише у випадку, якщо зміниться спосіб відображення типів досягнень.

```
@Pipe({
  name: 'otherAchievementsI18',
  standalone: true
})
export class OtherAchievementsTypeL18Pipe implements PipeTransform {

  transform(type: string): string {
    return OtherScientificAchievementsI18[type];
  }

}
```

Рисунок 3.1 – Реалізація одного з пайпів

Аналогічно, усі компоненти, що реалізовані у проєкті, побудовані відповідно до принципу єдиної відповідальності. Для кожного окремого типу дії - перегляду, редагування або створення - використовуються різні компоненти. Компоненти, які відповідають за створення або редагування об’єктів, містять лише відповідну форму та логіку взаємодії з API, не дублюючи при цьому функціональність перегляду. Натомість відображення вже наявних даних реалізується окремими, вузькоспеціалізованими компонентами, які не включають форм або контролів введення. Таке розмежування дозволяє уникнути конфліктів логіки, зменшує зв’язність коду та спрощує його супровід.

Окремо винесено й повторно використовувані елементи інтерфейсу, що не є повноцінними сторінками. Вони організовані в окрему структуру та використовуються як складові частини більшої UI-сценаріїв. Завдяки цьому такі елементи не залежать від логіки маршрутизації чи завантаження даних, які належать до відповідальності сторінкових компонентів.

Також для попередньої підготовки даних використовуються спеціальні об'єкти, що не займаються відображенням або форматуванням, а лише виконують функцію завантаження інформації перед активацією компонента. Завдяки

чіткому розподілу обов’язків між структурними одиницями коду — компонентами, сервісами, трансформаторами та підготовчими модулями — вдалось досягти високого рівня організованості. Кожен фрагмент програми виконує лише ту функцію, для якої він був створений. Дотримання принципу єдиної відповідальності у структурі фронтенду забезпечує чисту архітектуру, зменшення дублювання логіки та можливість ефективного масштабування застосунку. Це суттєво покращує підтримуваність проєкту в довгостроковій перспективі.

3.2. Розділення відповідальності між шарами застосунку

При розробці фронтенд-застосунків особливо важливо дотримуватись принципу розділення відповідальності між різними шарами (Separation of Concerns). Це дозволяє кожному структурному елементу програми відповідати лише за свою частину логіки: інтерфейс користувача, обробку даних, завантаження інформації або її трансформацію. Під час реалізації проєкту такий підхід використовувався системно, що забезпечило високу організованість і легкість у супроводі коду [13].

Інтерфейс користувача (UI-шар) реалізовано за допомогою Angular-компонентів, які відповідають виключно за відображення елементів інтерфейсу, взаємодію з користувачем та емісію подій (наприклад, натискання кнопки або заповнення форми). Такі компоненти не містять логіки збереження даних або їх обробки. Натомість усі операції з API винесено в окремі сервіси, які інжектуються в компонент та викликаються при потребі.

Бізнес-логіка та взаємодія з серверною частиною застосунку реалізована через окремі сервіси, які відповідають за створення, оновлення або отримання об’єктів. Такий підхід дозволяє ізолювати взаємодію з API від інтерфейсної частини, а також спрощує тестування кожного шару окремо.

Шар трансформації даних реалізовано через пайпи, які виконують лише одну функцію — перетворення вхідного значення в зручний для відображення формат. Завдяки цьому в шаблонах компонентів відсутня зайва логіка, а форматування даних здійснюється централізовано.

Шар підготовки даних до маршрутизації представлено резолверами, які відповідають за завантаження та структурування даних до того, як сторінка буде відображена. Це дозволяє зменшити час очікування користувача та уникнути порожніх елементів інтерфейсу під час початкового рендерингу.

Окрім цього, в окрему структуру винесено обробники обмежень, які виконують лише одну задачу — визначають специфіку фільтрації або доступності даних певного підрозділу. Це ще один приклад ізольованої відповідальності, що забезпечує розширюваність проєкту.

Загалом, така архітектура дозволяє кожному елементу програми мати чітку межу відповідальності, мінімізує зв’язність між модулями та спрощує майбутнє розширення системи без ризику порушення існуючої логіки. Дотримання принципу розділення шарів стало ключовим фактором у забезпеченні якості та стабільності розробленого застосунку.

3.3. Використання standalone-компонентів у реалізації інтерфейсу

При реалізації особистого кабінету викладача проєкт було побудовано з використанням сучасного підходу до компонування в Angular — standalone-компонентів. Цей підхід дозволяє спростити структуру застосунку, підвищити гнучкість повторного використання компонентів і зменшити залежність від модульної системи Angular (NgModule), яка домінувала в попередніх версіях фреймворку [14].

На відміну від класичного підходу до створення компонентів в Angular, який передбачає обов’язкову реєстрацію кожного компонента, пайпа чи

директиви в межах відповідного модуля (NgModule), концепція standalone-компонентів пропонує інший підхід до організації коду. Самостійні компоненти (standalone components) в Angular є самодостатніми одиницями — вони самі визначають свої залежності, включаючи імпортовані компоненти, директиви та модулі, які їм необхідні для роботи. Завдяки цьому вони не потребують додаткової реєстрації у полі declarations чи imports модуля, як це було необхідно раніше.

Такий підхід має низку переваг з точки зору архітектури та підтримки проєкту. По-перше, він дозволяє суттєво скоротити кількість коду, необхідного для створення нового компонента, що особливо помітно при масштабуванні великої системи. Кожен standalone-компонент може бути створений і використаний із мінімальною конфігурацією, без необхідності втручання у структуру модуля. По-друге, використання самостійних компонентів сприяє зменшенню дублювання імпортів, оскільки залежності компонента описуються безпосередньо в його визначенні. Це підвищує прозорість структури та спрощує читання коду.

Ще однією важливою перевагою є полегшення реалізації механізмів ледачого завантаження (lazy loading), оскільки standalone-компоненти можуть бути завантажені автономно, без потреби у створенні проміжних модулів. Це робить систему більш динамічною та гнучкою у розгортанні окремих частин інтерфейсу. Нарешті, такий підхід підвищує загальну підтримуваність коду: компоненти стають більш ізольованими, легше тестуються, а також краще підходять для повторного використання в межах різних частин застосунку.

У рамках реалізованого проєкту standalone-компоненти використовуються не лише для побудови сторінок, а й для реалізації вкладених функціональних секцій, що дозволило знизити зв’язаність коду, спростити структуру застосунку та забезпечити високу модульність. Такий підхід

узгоджується з принципами сучасної компонентної архітектури й виявився особливо ефективним у проєктах із великою кількістю повторюваних інтерфейсних елементів та окремих зон відповідальності. Реалізація в межах проєкту

Прикладом використання standalone-компонента є AcademicStatusComponent — окрема сторінка, яка відповідає за відображення та редагування відомостей про вчені звання викладача. Компонент оголошено зі специфікацією standalone: true (рисунок 3.2).

Рисунок 3.2 – Демонстрація оголошення standalone компоненту

Він містить повний перелік усіх компонентів, які необхідні для його функціонування (AccordionContainerComponent, CreateAcademicStatusAccordionComponent, AcademicStatusUpsertComponent тощо), що дозволяє використовувати його автономно, без оголошення в NgModule.

Аналогічно оформлено сторінку TeacherGeneralInfoComponent, яка також є standalone-компонентом та імпортує інші standalone-компоненти, зокрема й той самий AcademicDegreeComponent (рисунок 3.3).

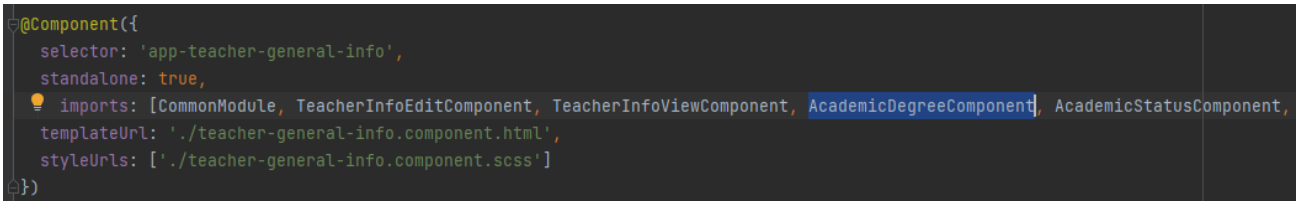


Рисунок 3.3 – Використання standalone компоненту в іншому компоненті

Використання standalone-компонентів виявилося особливо корисним у контексті складної багатосторінкової структури кабінету викладача, бо надало легший підхід для повторного використання елементів у різних вкладках та зменшило обсяг декларативного коду, покращило маштабованість та підвищало швидкість онбордингу нових розробників, оскільки кожен компонент чітко сепарований залежностями у класі.

3.4. Логічна структура та розподіл проєкту за каталогами

Для забезпечення високої підтримуваності, масштабованості та зручності навігації у фронтенд-частині застосунку було реалізовано продуману модульну структуру проєкту. Її основу складає папка pages, у якій зосереджено ключовий функціонал інтерфейсу. Такий підхід базується на принципах доменно-орієнтованого проєктування, розділення відповідальності та компонентної організації коду.

Усередині pages зберігаються окремі модулі, які відповідають за різні частини системи: модуль автентифікації (auth), модулі курсових робіт, документаційного менеджменту, панелі керування тощо. Найбільший обсяг має підрозділ user-profile-card, а саме його секція teacher-card, яка реалізує функціонал особистого кабінету викладача.

Всередині teacher-card структура побудована за принципом поділу на доступ до даних (data-access) та відображення інтерфейсу (pages). Такий розподіл дозволяє розмежовувати логіку взаємодії з API (сервіси, репозиторії) та компоненти, які безпосередньо відповідають за рендеринг UI. Це відповідає принципу Separation of Concerns і сприяє кращій модульності системи.

У папці pages кожен окремий тип даних (наприклад, educations, personal-achievements, projects, publications, qualification-improvement) представлений

власним підкаталогом. Кожен з них може містити вкладені папки view для візуальних компонентів, constants для фіксованих значень, type для опису типів даних, а також спеціалізовані підрозділи для створення, перегляду або редагування інформації. Наприклад, у модулі projects наявні окремі компоненти для створення (create-project-accordion), оновлення (project-upsert-card), перегляду (project-view-card) та виведення списку (projects-list) проєктів.

Кожен із цих компонентів реалізує принцип єдиної відповідальності (Single Responsibility Principle) та може бути повторно використаний в інших частинах системи. Наприклад, компонент project-upsert-card застосовується як для редагування існуючих записів, так і для створення нових, змінюючи лише вхідні параметри.

Крім того, у кожному модулі реалізовано власний файл маршрутизації routes.ts, що дозволяє централізовано визначати маршрути, полегшуючи реалізацію lazy loading та підвищуючи узгодженість у навігації між модулями. Уся система побудована так, щоб компоненти можна було легко переміщувати або повторно використовувати без порушення загальної архітектури.

3.5. Інтеграція з серверною частиною за допомогою OpenAPI

Одним із важливих аспектів реалізації сучасного вебзастосунку є забезпечення ефективної та безпечної взаємодії між фронтендом і бекендом. У цьому проєкті така взаємодія реалізована за допомогою інструменту OpenAPI Generator, який дозволяє автоматично створювати клієнтський код на основі специфікації REST API, описаної відповідно до стандарту OpenAPI [15].

Специфікація OpenAPI є формалізованим описом усіх можливих HTTP-запитів до сервера. Вона включає інформацію про методи, маршрути, параметри, структуру запитів і відповідей, а також типи даних. Такий підхід дозволяє мінімізувати непорозуміння між клієнтом і сервером, оскільки обидві сторони

взаємодіють на основі одного, уніфікованого опису. Завдяки цьому зростає прозорість API і спрощується підтримка застосунку на етапах масштабування або оновлення функціональності.

У проєкті на основі OpenAPI опису автоматично згенеровано Angular-сервіси, які реалізують логіку HTTP-запитів до серверної частини, а також моделі DTO (Data Transfer Object), що описують структуру вхідних і вихідних даних (рисунок 3.4). Окрім того, створено допоміжні утиліти, які відповідають за кодування параметрів, налаштування заголовків запитів, обробку параметрів шляху тощо. Кожен із ресурсів API має свій окремий сервіс, у якому реалізовано повний набір CRUD-операцій: створення, читання, оновлення й видалення. Наприклад, існує окремий сервіс, що відповідає за роботу з науковими ступенями, де кожна операція типізована й чітко визначена.

```
/**
 * merged spec
 * merged spec
 *
 * The version of the OpenAPI document: 1.0.0
 *
 *
 * NOTE: This class is auto generated by OpenAPI Generator (https://openapi-generator.tech).
 * https://openapi-generator.tech
 * Do not edit the class manually.
 */


export interface AboutViewDto {
    id?: number;
    missionUa: string;
    missionEn: string;
    visionUa: string;
    visionEn: string;
    historyUa: string;
    historyEn: string;
    strategyUa: string;
    strategyEn: string;
}
```

Рисунок 3.4 – Використання OpenAPI Generator для створення DTO

Застосування OpenAPI Generator дало змогу значно пришвидшити фронтенд-розробку. Розробник не витрачає час на ручне створення сервісів чи моделей, адже ці компоненти вже готові й типізовані. Крім того, у разі внесення змін на сервері, достатньо просто оновити OpenAPI опис і повторно згенерувати клієнтський код — це забезпечує повну синхронізацію між клієнтською та серверною частинами. Така інтеграція гарантує надійність і зменшує кількість помилок, пов’язаних із невірною структурою даних, оскільки типи перевіряються ще на етапі компіляції. Водночас зменшується дублювання коду, оскільки описані типи можуть використовуватись у багатьох модулях одночасно. Використання OpenAPI Generator стало не лише інструментом інтеграції, а й ключовим елементом забезпечення стабільності та підтримуваності всього застосунку. Це дозволило зосередитися на розробці інтерфейсної логіки, не витрачаючи зайвих ресурсів на технічне обслуговування базових структур.

3.6. Реалізація форм та валідації даних на основі Reactive Forms

Одним із ключових аспектів реалізації функціоналу особистого кабінету викладача є створення гнучкої системи форм для внесення та редагування даних. Це стосується, зокрема, введення інформації про наукові ступені, освіту, досягнення, публікації та інші складові академічної діяльності. Під час реалізації проєкту для побудови форм було обрано реактивний підхід (Reactive Forms), запропонований Angular. Такий підхід забезпечує широкі можливості для керування станом, структурою та валідацією введених даних.

Reactive Forms — це один із двох основних підходів до створення форм в Angular. Він базується на програмному описі структури форми у логіці компонента, що передбачає створення та керування окремими елементами форми (FormControl) та їх об’єднання у групи (FormGroup). Така архітектура дає змогу централізовано визначати валідатори, динамічно змінювати склад полів,

відстежувати зміни значень у реальному часі та мати повний контроль над внутрішнім станом форми — зокрема, valid, invalid, touched, dirty тощо [5].

У рамках компонента введення наукового ступеня було реалізовано три окремі форми: commonDegreeForm, ukrainianDegreeForm та internationalDegreeForm (рисунок 3.5). Перша з них містить загальні поля, як-от назва ступеня, установа, тема дисертації. Друга призначена для специфіки українських дипломів і охоплює такі поля, як галузь знань, спеціальність, дата видачі документа, серія та номер диплома. Третя форма — для міжнародних відзнак або ступенів, що мають іншу структуру.

Кожне поле має свої валідаційні обмеження, наприклад, поле name у загальній формі визначено як обов'язкове (за допомогою Validators.required), а поле diplomaSeries — із додатковими обмеженнями по довжині (Validators.maxLength(2) та Validators.minLength(2)). Така система дозволяє виявити помилки до моменту надсилання форми та вивести користувачеві відповідні повідомлення про некоректне заповнення.

```
commonDegreeForm = new FormGroup<IAcademicDegreeForms['commonDegreeForm']>( controls: {
    name: new FormControl( value: null, Validators.required),
    university: new FormControl( value: '', Validators.required),
    dissertationTopic: new FormControl( value: '', Validators.required),
    isUkrainianDiploma: new FormControl( value: true, Validators.required),
    userEmail: new FormControl( value: null, Validators.required)
});
readonly commonControlNames = getFormControlsNames(this.commonDegreeForm);

ukrainianDegreeForm = new FormGroup<IAcademicDegreeForms["ukrainianDegreeForm"]>( controls: {
    codeOfScientificSpecialty: new FormControl( value: {value: null, disabled: true}, validatorOrOpts: [Validators.required]),
    fieldOfStudy: new FormControl( value: {value: null, disabled: true}, validatorOrOpts: [Validators.required]),
    specialty: new FormControl( value: {value: null, disabled: true}, validatorOrOpts: [Validators.required]),
    date: new FormControl( value: null, validatorOrOpts: [Validators.required, pastDateValidator()]),
    diplomaNumber: new FormControl( value: null, Validators.required),
    diplomaSeries: new FormControl( value: '', validatorOrOpts: [Validators.required, Validators.maxLength( maxLength: 2), Validators.minLength( minLength: 2)]),
    documentWasIssuedBy: new FormControl( value: '', Validators.required)
});

get ukrainianDegreeFormControls() {
    return this.ukrainianDegreeForm.controls
}

readonly ukrainianControlNames = getFormControlsNames(this.ukrainianDegreeForm);

internationalDegreeForm = new FormGroup<IAcademicDegreeForms["internationalDegreeForm"]>( controls: {
    award: new FormControl( value: '', Validators.required)
});
```

Рисунок 3.5 – Використання FormGroup та FormControl для валідації форм

Особливістю реалізації є адаптивна логіка валідації, яка змінюється залежно від контексту. Наприклад, якщо користувач обирає, що має український диплом, активується ukrainianDegreeForm, тоді як для міжнародних даних активується internationalDegreeForm. Це забезпечує гнучкість інтерфейсу та виключає потребу заповнювати зайві поля. Крім того, реалізовано кастомний валідатор pastDateValidator(), що перевіряє, аби введена дата не була пізнішою за поточну — критично важливий механізм для заповнення історичних відомостей.

Форми також мають реактивну поведінку. Наприклад, при зміні спеціальності може автоматично оновлюватися пов’язана інформація про галузь знань або шифр спеціальності. Це реалізується через механізм підписки на події valueChanges, який надається RxJS — стандартною бібліотекою реактивного програмування в Angular.

HTML-шаблон реалізовано за допомогою конструкції <form [formGroup]="...">, з використанням таких елементів, як <ng-select> для вибору значень зі списку, <input> і <textarea> для текстових полів, а також спеціальних компонентів для завантаження сканів документів. Усі поля мають чіткі підписи, марковані обов’язковість та пояснення помилок валідації. Окремі секції форми логічно відокремлено: наприклад, блок «Інформація про диплом» візуально відділений від теми дисертації, що підвищує зручність навігації.

Поведінка при збереженні форми також структурована. У методі submit() здійснюється перевірка валідності всіх активних форм. Якщо виявлено помилки, вони підсвічуються, а поля позначаються як touched, що забезпечує зрозуміле інформування користувача про необхідність корекції введених даних.

3.7. Використання RxJS у роботі з асинхронними подіями

Одним із важливих інструментів, що використовуються у фронтенд-розробці з Angular, є бібліотека RxJS (Reactive Extensions for JavaScript). При

реалізації особистого кабінету викладача RxJS забезпечує ефективну обробку асинхронних подій, таких як HTTP-запити, взаємодія з формами та реакція на зміну стану.

RxJS ґрунтується на парадигмі реактивного програмування, у якій дані представлені у вигляді потоків (Observable), що дозволяє легко реагувати на зміни та комбінувати різні джерела подій. У проєкті особистого кабінету викладача це використовується для роботи з API-сервісами, завантаженням та видаленням файлів, а також керуванням станом форм та валідації [6].

Зокрема, у компоненті ABaseInfoAccordionComponent передбачено кілька сценаріїв застосування RxJS. Один із них — метод uploadFile$(), який відповідає за завантаження файлів до сервера. Якщо файл ще не прикріплено, викликається метод uploadFile() з подальшим використанням оператора tap() — для реакції після успішного завершення завантаження, наприклад, оновлення інформації про файл. Якщо ж файл уже існує, спочатку виконується його видалення методом removeFile$(), після чого новий файл завантажується (рисунок 3.6). Для цього використовується оператор switchMap(), який дозволяє виконати послідовну зміну потоків: спочатку видалити файл, а потім завантажити новий.

```
protected uploadFile$(file: File) {
    const observer = this.fileStorageService.uploadFile(this.fileEntityType, this.$state().id, file)
        .pipe(
            tap( next: id => this.fetchFileInfo())
        )
    if (!this.fileInfo?.id)
        return observer

    return this.removeFile$()
        .pipe(
            switchMap( project: () => observer)
        )

}

protected removeFile$() {
    return this.fileStorageService.deleteFile(this.fileEntityType, this.$state().id, this.fileInfo.id)
        .pipe(tap( next: () => {
            this.fileInfo = null
        }))
}
```

Рисунок 3.6 – Використання observer, pipe та switchMap, як демонстрація RxJS

Інший приклад — використання методу decorateHttpRequest() сервісу ToastService, який обгортає запити до сервера і дозволяє легко додавати обробку результатів (наприклад, відображення повідомлень про успішне збереження даних). Цей механізм базується на підписці (subscribe()), що надає змогу реагувати на результат виконання запиту та оновити внутрішній стан форми (this.$state.set(...)).

RxJS у проєкті виступає ключовим інструментом для побудови асинхронної взаємодії з бекендом, забезпечуючи зручне об’єднання послідовностей дій, контроль за станом і реакцією на події. Завдяки операторам tap, switchMap, subscribe тощо, вдалося реалізувати чисту й контрольовану логіку взаємодії з API, що відповідає сучасним підходам до архітектури Angular-застосунків.

ВИСНОВКИ

Під час виконання даної роботи було проведено дослідження актуальної проблеми сучасної системи освіти – її цифровізація. Були розглянуті гарні та погані практики в розробці користувацького інтерфейсу з метою вдалого використання цих практик та розробки максимально доступного веб-застосунку.

Було розглянуто цілі та функції особистого кабінету викладача, з метою покриття всіх необхідних сценаріїв під час розробки

Особливу увагу приділено валідації форм, візуальній ієрархії інтерфейсу, поведінці при введенні даних та адаптації до потреб звичайних користувачів. Було використано сучасні інструменти — Angular, Angular Material, Reactive Forms та RxJS, що дало змогу реалізувати масштабовану архітектуру з модульною побудовою компонентів за рахунок використання сучасних принципів та підходів до розробки великих веб-застосунків. Інтеграція з бекендом здійснювалась за допомогою OpenAPI Generator, що забезпечило повну типізацію та узгодженість клієнтської логіки з API сервера.

Так-як програмування постійно еволюціонує та з’являється більше інструментів для веб-розробки, дотримання базових принципів та практик під час цієї розробки здатне оптимізувати використаний час та запобігти негативним наслідкам у майбутньому.

СПИСОК ЛІТЕРАТУРИ

1. Van Deusen A. URL: https://flatironschool.com/blog/what-is-ux-ui-design/#anchor1 (дата звернення: 01.05.2025).
2. Nielsen J. URL: https://www.nngroup.com/articles/ten-usability-heuristics/ (дата звернення: 01.05.2025).
3. Figma. URL: https://www.figma.com/design/ (дата звернення: 01.05.2025).
4. Creating Beautiful Apps with Angular Material. URL: https://auth0.com/blog/creating-beautiful-apps-with-angular-material/ (дата звернення: 03.05.2025).
5. Reactive forms. URL: https://angular.dev/guide/forms/reactive-forms (дата звернення: 22.04.2025).
6. RxJS оператори. URL: https://medium.com/@jstify.community/rxjs-оператори-5cc8f0bdb0f6 (дата звернення: 10.04.2025).
7. Interaction Design Foundation. URL: https://www.interaction-design.org/literature/topics/skeuomorphism (дата звернення: 11.04.2025).
8. Interaction Design Foundation. URL: https://www.interaction-design.org/literature/article/flat-design-an-introduction (дата звернення: 11.04.2025).
9. Usability Testing: what it is, its benefits, and why it matters. URL: https://contentsquare.com/guides/usability-testing/ (дата звернення: 16.04.2025).
10. Using heatmaps to improve your website’s UX: 5 ways to get started. URL: https://contentsquare.com/guides/heatmaps/ux/ (дата звернення: 13.04.2025).
11. Що таке Moodle. URL: https://moodle.org/mod/page/view.php?id=8174 (дата звернення: 13.04.2025).

12.ПАТЕРНИ та ПРИНЦИПИ Програмування. URL: https://medium.com/@tatyanareshetnik/патерни-та-принципи-програмування-6772c8e7607c (дата звернення: 01.05.2025).
13. Separation of Concerns. URL: https://www.geeksforgeeks.org/separation-of-concerns-soc/ (дата звернення: 01.05.2025).
14. Ahmad A. Angular Modules vs. Standalone Components: A Detailed Comparison. URL: https://medium.com/@ausaf.cs/angular-modules-vs-standalone-components-a-detailed-comparison-7b28c803cdd2 (дата звернення: 13.04.2025).
15. OpenAPI Generator Tutorial. URL: https://apidog.com/blog/openapi-generator-tutorial/ (дата звернення: 15.04.2025).

ДОДАТКИ

Рисунок А.1 – Горизонтальні вкладки кабінету викладача

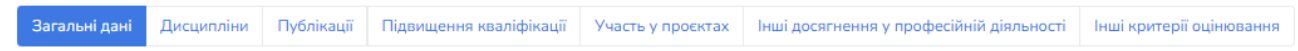


Рисунок Б.2 – Вигляд викладацької картки кабінету викладача

Рисунок В.3 – Форма нового рівня освіти з усіма вдалими поведінками інтерфейсу

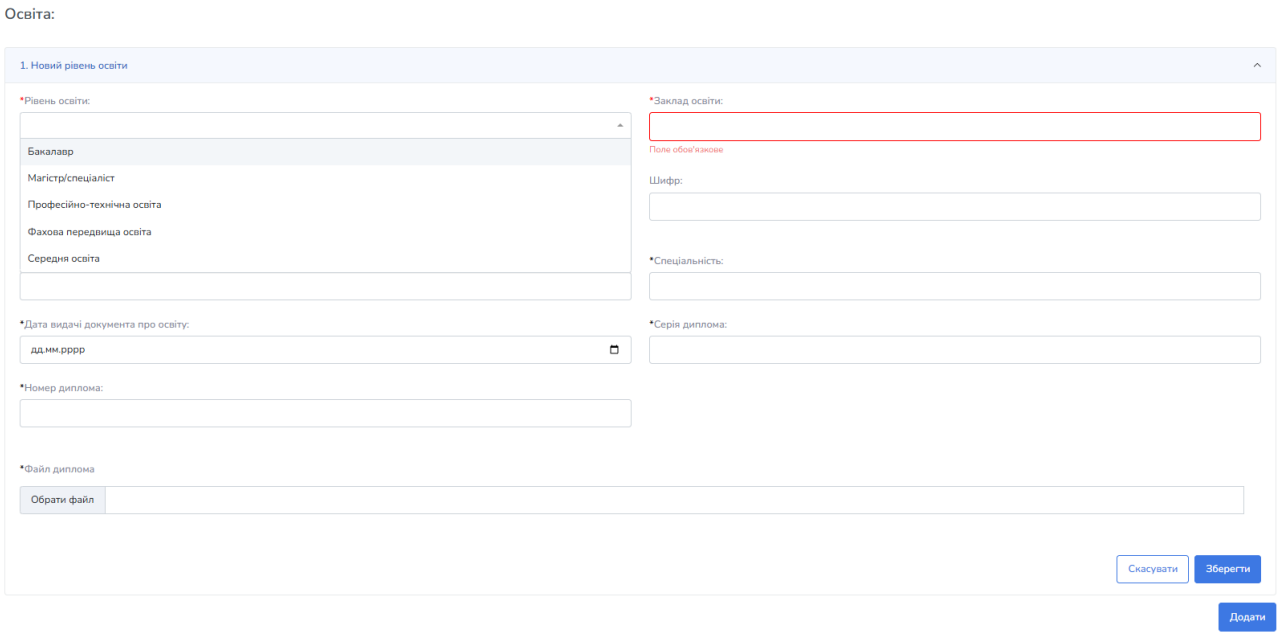


Рисунок Ґ.4 – Реалізація одного з пайпів

```
@Pipe({
  name: 'otherAchievementsI18',
  standalone: true
})
export class OtherAchievementsTypeL18Pipe implements PipeTransform {

  transform(type: string): string {
    return OtherScientificAchievementsI18[type];
  }

}
```

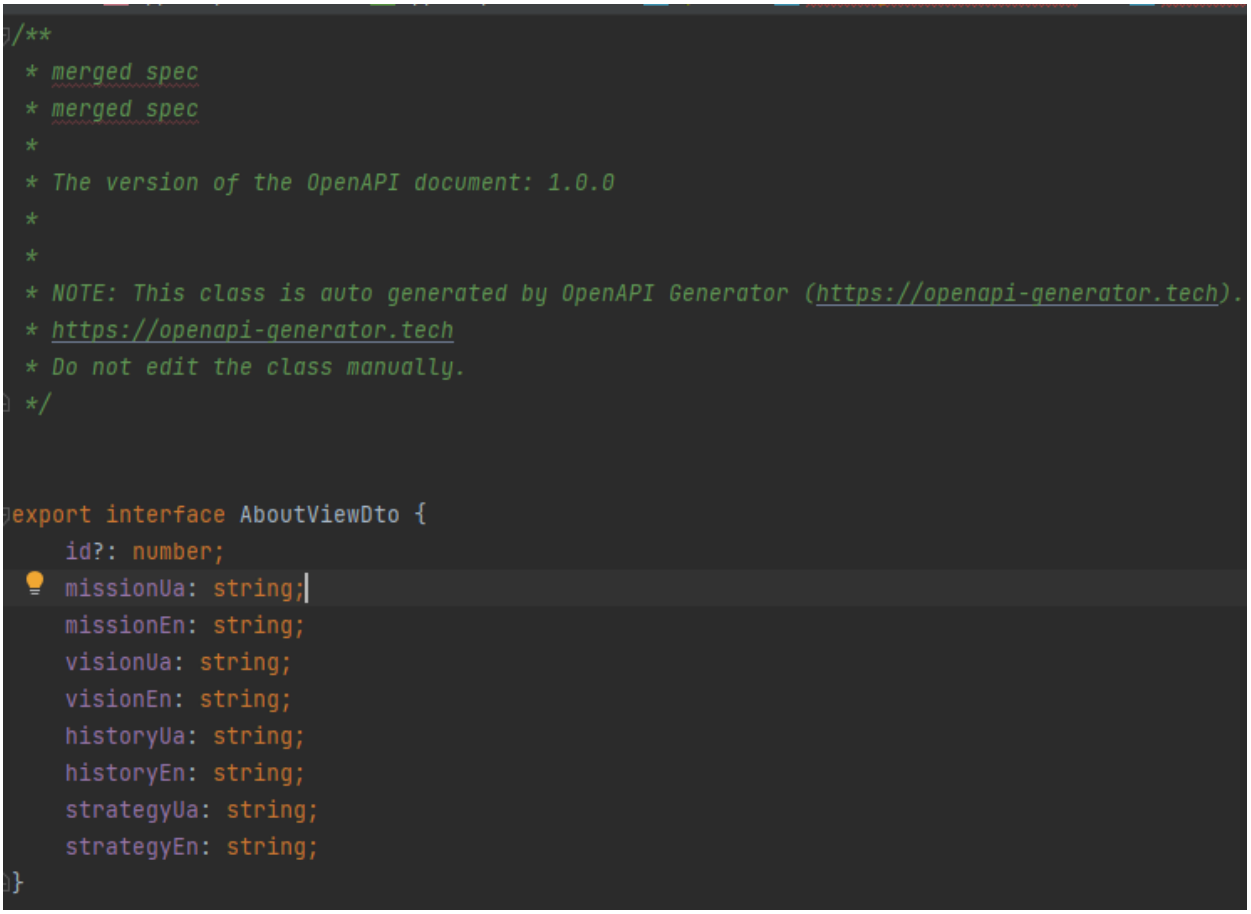
Рисунок Д.5 – Демонстрація оголошення standalone компоненту

```
@Component({
  selector: 'app-scientific-degree',
  standalone: true,
  imports: [CommonModule, AccordionContainerComponent, AcademicStatusViewComponent, AcademicStatusUpsertComponent, SavedAcademicStatusAccordionComponent,
  templateUrl: './academic-status.component.html',
  styleUrls: ['./academic-status.component.scss']
})
```

Рисунок Е.6 – Використання standalone компоненту в іншому компоненті

```
@Component({
  selector: 'app-teacher-general-info',
  standalone: true,
  imports: [CommonModule, TeacherInfoEditComponent, TeacherInfoViewComponent, AcademicDegreeComponent, AcademicStatusComponent,
  templateUrl: './teacher-general-info.component.html',
  styleUrls: ['./teacher-general-info.component.scss']
})
```

Рисунок Є.7 – Використання OpenAPI Generator для створення DTO



```
/**
 * merged spec
 * merged spec
 *
 * The version of the OpenAPI document: 1.0.0
 *
 *
 * NOTE: This class is auto generated by OpenAPI Generator (https://openapi-generator.tech).
 * https://openapi-generator.tech
 * Do not edit the class manually.
 */


export interface AboutViewDto {
    id?: number;
    missionUa: string;
    missionEn: string;
    visionUa: string;
    visionEn: string;
    historyUa: string;
    historyEn: string;
    strategyUa: string;
    strategyEn: string;
}
```

Рисунок Ж.8 – Використання FormGroup та FormControl для валідації форм

```
commonDegreeForm = new FormGroup<IAcademicDegreeForms['commonDegreeForm']>( controls: {
    name: new FormControl( value: null, Validators.required),
    university: new FormControl( value: '', Validators.required),
    dissertationTopic: new FormControl( value: '', Validators.required),
    isUkrainianDiploma: new FormControl( value: true, Validators.required),
    userEmail: new FormControl( value: null, Validators.required)
});
readonly commonControlNames = getFormControlsNames(this.commonDegreeForm);

ukrainianDegreeForm = new FormGroup<IAcademicDegreeForms["ukrainianDegreeForm"]>( controls: {
    codeOfScientificSpecialty: new FormControl( value: {value: null, disabled: true}, validatorOrOpts: [Validators.required]),
    fieldOfStudy: new FormControl( value: {value: null, disabled: true}, validatorOrOpts: [Validators.required]),
    specialty: new FormControl( value: {value: null, disabled: true}, validatorOrOpts: [Validators.required]),
    date: new FormControl( value: null, validatorOrOpts: [Validators.required, pastDateValidator()]),
    diplomaNumber: new FormControl( value: null, Validators.required),
    diplomaSeries: new FormControl( value: '', validatorOrOpts: [Validators.required, Validators.maxLength( maxLength: 2), Validators.minLength( minLength: 2)]),
    documentWasIssuedBy: new FormControl( value: '', Validators.required)
});

get ukrainianDegreeFormControls() {
    return this.ukrainianDegreeForm.controls
}

readonly ukrainianControlNames = getFormControlsNames(this.ukrainianDegreeForm);

internationalDegreeForm = new FormGroup<IAcademicDegreeForms["internationalDegreeForm"]>( controls: {
    award: new FormControl( value: '', Validators.required)
});
```

Рисунок 3.9 – Використання observer, pipe та switchMap, як демонстрація RxJS

```
protected uploadFile$(file: File) {
    const observer = this.fileStorageService.uploadFile(this.fileEntityType, this.$state().id, file)
        .pipe(
            tap( next: id => this.fetchFileInfo())
        )
    if (!this.fileInfo?.id)
        return observer

    return this.removeFile$()
        .pipe(
            switchMap( project: () => observer)
        )

}

protected removeFile$() {
    return this.fileStorageService.deleteFile(this.fileEntityType, this.$state().id, this.fileInfo.id)
        .pipe(tap( next: () => {
            this.fileInfo = null
        }))
}
```