Міністерство освіти і науки України
Рівненський державний гуманітарний університет
Кафедра інформаційних технологій та моделювання

Кваліфікаційна робота
за освітнім ступенем «бакалавр»
на тему:
«Інформаційна система підтримки електронного документообігу закладу вищої освіти. Централізована платформа взаємодії інтегрованих компонентів»

Виконав:
здобувач IV курсу
групи ІПЗ-41
спеціальності 121 «Інженерія програмного забезпечення»
Роман Ігорович Кузьмич

Науковий керівник:
старший викладач Тетяна Анатоліївна Кирик

М. Рівне, 2025 рік

ЗМІСТ

ВСТУП

У наш час заклади вищої освіти все частіше стикаються з проблемами, пов'язаними з обробкою та управлінням великими обсягами документації. Все більше закладів починають перехід від традиційного паперового документообігу до електронних систем, що значно полегшує роботу з документами та підвищує ефективність адміністративних процесів.

Актуальність. У сучасних закладах вищої освіти обсяг документів, що створюються, обробляються та зберігаються, постійно зростає, а вимоги до швидкості обробки інформації та доступу до неї стають все більш критичними. У зв’язку з цим виникає потреба у впровадженні ефективної системи електронного документообігу, яка здатна забезпечити автоматизацію процесів, скорочення паперової роботи та підвищення продуктивності співробітників.

Мета кваліфікаційної роботи — дослідити та обґрунтувати доцільність створення власної системи електронного документообігу для закладу вищої освіти, розробити інтерфейс користувача, що дозволить автоматизувати процеси обробки та генерації документів, а також впровадити надійний механізм автентифікації за допомогою Google OAuth.

Об’єктом кваліфікаційної роботи є процеси електронного документообігу в закладах вищої освіти та можливості їхньої автоматизації.

Предметом кваліфікаційної роботи є технології, що використовуються для розробки електронної системи документообігу. Вивчення особливостей інтеграції та розробки власних мікросервісів, побудови архітектури системи, а також розробка вебінтерфейсу для забезпечення зручності та продуктивності користувачів.

Завдання кваліфікаційної роботи:

- визначити переваги та вимоги до власної системи документообігу у закладі вищої освіти;
- провести порівняльний аналіз існуючих комерційних систем документообігу;

- спроєктувати масштабовану архітектуру для вебзастосунку;
- розробити інтуїтивно зрозумілий для користувача інтерфейс для взаємодії з документами;
- реалізувати власний сервіс авторизації з підтримкою Google OAuth 2.0;
- об’єднати мікросервіси авторизації, навчальних планів, студентів та генерації диплому у одному вебзастосунку.

Методи дослідження. У кваліфікаційної роботі застосовуються кілька методів дослідження для досягнення поставленої мети та завдань:

- Аналіз і порівняння: для вивчення існуючих рішень в сфері електронного документообігу та визначення переваг і недоліків різних комерційних систем.
- Моделювання: для розробки архітектурних елементів системи та діаграм взаємодії.
- Проєктування: для створення інтерфейсу користувача, враховуючи вимоги до зручності і продуктивності взаємодії; для розробки архітектури механізму авторизації з використанням Google OAuth та JWT.
- Експеримент: для реалізації та тестування власного механізму авторизації у системі.

Практичне значення дослідження. Розроблений вебзастосунок електронного документообігу допоможе працівникам закладу вищої освіти більш ефективно обробляти навчальні плани, дані студентів та генерувати дипломи для здобувачів освіти.

Апробація. Основні теоретичні та практичні результати дослідження доповідалися та обговорювалися на XVIII Всеукраїнській науково-практичній конференції здобувачів вищої освіти і молодих учених «Наука, освіта, суспільство очима молодих» (м. Рівне, 14 травня 2025 р.), звітній науковій конференції викладачів, співробітників і здобувачів вищої освіти Рівненського державного гуманітарного університету (м. Рівне, 15 травня 2025 року).

Структура роботи. Основна частина кваліфікаційної роботи складається з п’яти розділів.

Перший розділ розкриває тему потреби власної системи електронного документообігу для вищого навчального закладу та її порівняння з готовими комерційними рішеннями.

У другому розділі описуються обрані технології для розробки вебзастосунку, такі як TypeScript, React та допоміжні бібліотеки Redux, React Router та Axios.

Третій розділ описує архітектуру застосунку, включаючи діаграми взаємодії з різними мікросервісами та користувачами. Також представлено інтерфейс користувача для проведення операцій з документами.

Четвертий розділ присвячено розробці власного сервісу авторизації, який забезпечує захист та контроль доступу до системи. Розглянуто переваги інтеграції Google OAuth, використання JWT токенів для ідентифікації користувачів, а також реалізацію сервісу авторизації за допомогою Ktor. Наведено схему роботи авторизації у межах загальної архітектури вебзастосунку.

П’ятий розділ описує процес інтеграції додаткових мікросервісів із вебзастосунком. Зокрема, розглянуто обробку запитів для отримання інформації про студентів та їх групи, взаємодію з навчальними планами та системою оцінювання, а також механізм автоматизованої генерації диплому з урахуванням досягнень студента.

РОЗДІЛ 1. ПЕРЕВАГИ ВЛАСНОЇ СИСТЕМИ ДОКУМЕНТООБІГУ ТА АНАЛІЗ ГОТОВИХ РІШЕНЬ

1.1 Власна система документообігу

У новітніх закладах вищої освіти з часом виникає потреба оптимізації своєї адміністративної діяльності, яка включає обробку та зберігання великої кількості документів у архівах. У цьому випадку впровадження системи електронного документообігу стає неминучим кроком до цифровізації освітнього процесу. Така система дозволяє зберігати, керувати та обробляти документи, створюючи єдине середовище для швидкого доступу до будь-якої інформації, що спрощує та прискорює роботу, даючи змогу працівникам закладу освіти більше зосередитися на освітній діяльності замість виконання рутинних завдань.

Система документообігу, за свою історію, пройшла великий шлях еволюції від паперових архівів до сучасних цифрових платформ. Раніше обробка і зберігання документів потребувала значних сил та ресурсів, наприклад: спеціальних приміщень, витрат на папір та технічного обслуговування архівів. Проте, розвиток інформаційних технологій дозволяє оптимізувати роботу з паперовими документами та перенести їх в електронний формат, що скоротить витрати на їх зберігання та прискорить доступ до них.

Електронна система документообігу дозволяє зменшити паперову роботу, покращити стандартизацію даних та підвищити загальну продуктивність вищого навчального закладу. Наприклад, замість ручного оформлення й перевірки документів, працівники можуть використовувати автоматизовані шаблони, які забезпечують точність і стандартизацію даних, що є дуже важливим для закладів освіти.

У майбутньому, за допомогою такої системи, можливе впровадження додаткової аналітики, контролю ефективності освітніх процесів, моніторингу успішності студентів та розширення інтеграції з іншими системами. Наприклад, систему можна буде використовувати для збору і аналізу даних про відвідуваність і академічну успішність студентів, що дасть змогу оцінити

навчальний процес з іншого ракурсу та виявити можливі місця для автоматизації. Також, це дозволить більш ефективно планувати ресурси та приймати обґрунтовані управлінські рішення, спрямовані на покращення якості освітніх послуг.

1.1.1 Практична цінність системи для закладу вищої освіти

Важливим фактором є підвищення продуктивності роботи, оскільки автоматизація процесів з документами за допомогою електронної системи розв’язує проблему з нестачею часу у працівників на ведення паперів, дозволяючи зосередитись на основних освітніх завданнях.

Однією з переваг власної електронної системи документообігу є забезпечення високого рівня контролю над даними. А саме, можливість гнучкого налаштування доступу до різних типів документів залежно від рівня повноважень користувачів. Завдяки такій системі, заклад освіти отримує можливість захистити чутливу інформацію, наприклад, особисті дані студентів та співробітників, навчальні плани, фінансову документацію тощо.

Окрім цього, система сприяє зниженню витрат, пов’язаних із паперовою документацією. Зменшення загальної кількості паперових документів скорочує витрати на їх друк, копіювання та зберігання, а також знижує потребу у виділенні спеціальних приміщень для архівів. Це особливо актуально для університетів з великою кількістю паперових документів. Тому, перехід на електронні документи є не лише зручним, але й економічно вигідним рішенням.

Наступною перевагою є покращена швидкість обробки запитів від студентів, викладачів та адміністрації. Завдяки автоматизованим процесам система дозволяє швидко отримувати доступ до необхідних даних, виконувати їхній пошук, сортування чи аналіз. Наприклад, у разі необхідності внесення змін до навчального плану або отримання інформації про успішність студентів, система дозволяє виконати ці операції, буквально, у декілька натискань кнопок, що робить управління освітнім процесом набагато простішим. Це, в свою чергу, підвищує загальну ефективність роботи закладу, оскільки всі операції можуть

виконуватися у рази швидше, ніж у випадку використання традиційної паперової системи.

Використання електронної системи документообігу сприяє підвищенню рівня прозорості та взаємодії всередині закладу. Налагоджена система електронного документообігу дає можливість усім користувачам мати чітке уявлення про поточний стан навчального процесу, розклад, доступні ресурси та інші важливі аспекти. Наприклад, студенти зможуть отримувати своєчасну інформацію про зміни в розкладі, навчальні матеріали або інші оголошення щодо навчання. Для адміністрації та викладачів доступ до системи забезпечує можливість моніторингу навчального процесу та успішності здобувачів освіти та швидке прийняття рішень для поліпшення якості надання освітніх послуг.

1.1.2 Основні вимоги

Для ефективної та коректної роботи системи документообігу в закладі вищої освіти необхідно забезпечити відповідність ряду основних вимог. Ці вимоги охоплюють аспекти функціональності, безпеки, зручності у використанні та масштабованості системи. Тільки за умови відповідності усім цим вимогам система зможе забезпечити стабільне функціонування і повноцінно підтримувати навчальний процес та адміністративну діяльність закладу.

По-перше, система має бути зручною та інтуїтивно зрозумілою для різних груп користувачів, включаючи адміністрацію, викладачів та студентів. Інтерфейс користувача повинен бути розроблений так щоб можна було легко знаходити, завантажувати, переглядати та обробляти документи без необхідності проходити тривале навчання.

Другим важливим аспектом є високий рівень безпеки даних. Заклади вищої освіти досить часто мають справу з чутливою інформацією, такою як особисті дані студентів, інформацією про успішність та інші внутрішні документи. Тому система повинна забезпечувати надійний захист даних від несанкціонованого доступу. Наприклад, використовувати сучасні технології шифрування,

багатофакторну автентифікацію, контроль доступу та регулярне оновлення безпекових протоколів для уникнення витоків інформації.

Крім того, система має бути масштабованою та гнучкою, щоб легко адаптуватися до потреб навчального закладу в умовах зміни діяльності, впровадження нових процесів. Масштабованість важлива для інтеграції системи з іншими платформами, такими як бібліотеки, електронні журнали та онлайн-платформи для навчання на кшталт Moodle.

Наступною вимогою є забезпечення швидкої та безперебійної роботи. Система буде обробляти значні обсяги даних, важливо, щоб її продуктивність залишалася стабільною навіть при пікових навантаженнях. Тому що збій системи може призвести до зупинки навчального процесу та негативно вплинути на репутацію закладу. Для уникнення таких проблем потрібно використовувати високопродуктивне серверне обладнання, оптимізувати взаємодію між мікросервісами та проводити регулярне технічне обслуговування, як системи так і серверів.

Сумлінне дотримання вищезазначених вимог забезпечить надійність, ефективність та довготривалу актуальність електронної системи документообігу для закладу вищої освіти, що дозволить закладу отримати максимальну користь від впровадження цього інструменту.

1.2 Порівняння з комерційними рішеннями

Під час вибору електронної системи документообігу для закладу вищої освіти з’являється дилема розробляти власний продукт чи використовувати готове комерційне рішень. Обидва підходи мають свої плюси та мінуси, які потрібно проаналізувати, та визначитися з найбільш оптимальним варіантом, який краще відповідає специфічним вимогам закладу освіти.

Готовий продукт має перевагу в плані швидкості впровадження, оскільки, він, як правило, є готовим до використання з коробки. Комерційні системи створені із загальними функціями документообігу, які можуть задовольнити базові потреби багатьох організацій.

Оскільки комерційні продукти часто створюються для широкого кола різноманітних користувачів, вони можуть не враховувати специфічні вимоги, характерні саме для закладів вищої освіти. Наприклад, нам важливо щоб серед функціоналу було доступне формування навчальних планів, облік успішності студентів, автоматичне формування звітів про успішність тощо. Пошук або адаптація готового продукту може забрати набагато більше часу та ресурсів, ніж розробка власної системи. До того ж при проєктуванні власної системи можна додати усі необхідні функції і забезпечити наявність тільки найважливішого функціоналу для закладу освіти.

1.2.1 Аналіз існуючих продуктів

Megapolis.DocNet — це українська система документообігу, яка надає можливість централізованого зберігання документів, дозволяє користувачам здійснювати пошук, обмін та спільне редагування файлів [1]. Функціональні можливості Megapolis.DocNet включають узгодження та реєстрацію документів, роботу з документами у онлайн архіві та офлайн-менеджері, генерацію звітів, моніторинг виконання завдань та багато іншого. Крім того, система пропонує можливість налаштування доступу користувачів до документів залежно від їхніх ролей, що забезпечує безпеку й конфіденційність даних. Цінова політика Megapolis.DocNet залежить від масштабів організації та обсягу функціоналу. Вартість ліцензії розраховується індивідуально, і деякі університети можуть отримати спеціальні умови для впровадження системи. Наприклад, Київський політехнічний інститут імені Ігоря Сікорського використовує Megapolis.DocNet, що підтверджує її використання серед закладів вищої освіти України [1].

Alfresco — це система управління корпоративними інформаційними ресурсами і документообігом, один з лідерів на ринку вільного програмного забезпечення, серед програм для організації електронного документообігу [2]. Система підтримує функціонал для організації документів, керування доступом, налаштування робочих процесів та створення архівів. Також є версійність документів та відстеження дій користувачів у системі, що підвищує прозорість і

контроль за документацією. Дана система має вбудовані функції для інтеграції з іншими бізнес-додатками та підтримує налаштування індивідуальних інтеграцій. Головною перевагою цього продукту є відкритий код, що дає можливість краще і простіше інтегруватися з іншими, внутрішніми, системами навчального закладу. Alfresco надає можливість вибору між безкоштовною версією та комерційною, яка включає додаткові функції й підтримку від розробника.

DocuWare — це спеціалізована система документообігу, орієнтована на простоту використання та інтеграцію з бізнес-процесами. DocuWare дозволяє зберігати й впорядковувати документи, забезпечуючи зручний доступ до них для співробітників, а також пропонує функції автоматизації процесів і налаштування робочих процесів. Система містить функціонал для створення шаблонів і автоматичного заповнення документів, що прискорює роботу з типовими формами. Система є хмарною платформою та пропонує різні тарифні плани, які розраховуються на основі кількості користувачів і обсягу зберігання даних. Базовий пакет послуг буде вартувати 300 доларів на місяць, та може досягати 2000 доларів на місяць для великих організацій з розширеним функціоналом і збільшеними потребами в зберіганні.

1.2.2 Переваги власної системи

Розробка власної системи, порівняно з використанням стандартних комерційних рішень, має декілька переваг. Така система може бути повністю адаптована до специфічних потреб конкретного закладу, враховуючи його внутрішню структуру, правила документообігу та політику збереження даних. Це сприяє підвищенню ефективності використання системи, уникаючи обмежень, які можуть виникати при впровадженні сторонніх продуктів. Крім того, власна система може гнучко адаптуватися до майбутніх змін у вимогах освітньої діяльності та законодавства, що також забезпечує її довготривалу актуальність.

Не менш важливим фактором є ціна готової системи документообігу. Зазвичай такі системи мають механізм підписки на певний строк. Вартість може залежати від різних факторів, таких як обслуговування, обсяг збережених

документів, використовувані функції, тощо. З часом ці витрати можуть ще більше зрости, особливо для великих установ із великою кількістю даних. Хоча власна система також потребує витрат для функціонування чи зберігання даних, проте вони можуть бути набагато меншими, оскільки оплата здійснюється лише за використані ресурси, без додаткових переплат за непотрібні функції чи ліцензії.

Можна сказати, що вибір між власною системою документообігу та комерційними рішеннями залежить від потреб закладу вищої освіти. Комерційні системи можуть бути швидкими у впровадженні, але з надлишковою або недостатньою функціональністю, що може обмежувати заклад у можливостях налаштування та розвитку функціоналу. Власна система дозволяє отримати гнучке рішення, адаптоване до потреб конкретного закладу та оптимізоване для довгострокової роботи.

РОЗДІЛ 2. TYPESCRIPT ЯК АЛЬТЕРНАТИВА JAVASCRIPT ТА СУЧАСНИЙ ФРЕЙМВОРК ДЛЯ ВЕБЗАСТОСУНКІВ REACT

2.1 Огляд TypeScript

TypeScript — це строго типізована мова програмування, основою якої є JavaScript. Строго типізованою мовою програмування називають мову, яка має статичну типізацію даних. Це означає, що типи даних змінних визначаються на етапі компіляції, і система жорстко контролює, щоб тип змінних не змінювався протягом виконання програми. Така технологія дозволяє уникнути багатьох потенційних помилок, які можуть виникнути в динамічно типізованих мовах, таких як Python. Це і стало однією із причин для розробки TypeScript компанією Microsoft у 2012 році. Іншою причиною була відсутність потужних засобів для організації структури великих проєктів, а саме інтерфейсів, класів та модулів.

Проаналізуємо чому типізація у TypeScript є такою важливою перевагою, вона створює систему, яка приймає код JavaScript, але має типи. Це надає систему типів без необхідності додавати додаткові символи, щоб зробити типи явними у коді. Нижче наведено приклад як працює типізація даних (Див. рис. 2.1).

Рисунок. 2.1. Приклад типізованої змінної

Можна зауважити, що перед присвоєнням даних, змінній було надано тип *string*, що чітко вказує на те, що ця змінна може мати в собі тільки текстовий рядок. На рис. 2.2 зображено як виглядав би такий самий код, але написаний на JavaScript без використання мови типізації (Див. рис. 2.2).

Рисунок. 2.2. Приклад не типізованої змінної

Як видно на обох рисунках, змінні названі однаково, та містять одне і теж значення, але в першому випадку до змінної *myName* був присвоєний тип *string* тому при наведені курсором на цю змінну можна побачити який це тип і які дані містить змінна

TypeScript широко використовується для розробки як клієнтської, так і серверної частин застосунків. Наприклад, на клієнтській стороні TypeScript часто інтегрується з такими популярними фреймворками, як React або Angular, дозволяючи розробникам створювати динамічні інтерфейси користувача, а на серверній стороні він застосовується в поєднанні з Node.js для створення надійних серверних застосунків, забезпечуючи високу продуктивність та безпеку завдяки підтримці типів.

У 2020 році TypeScript посів 2-е місце у рейтингу найулюбленіших мов програмування за результатами опитування Stack Overflow 2020 Developer survey [3]. Відсутність строгої типізації у JavaScript була стримуючим фактором для багатьох, хто хотів би поринути у веброзробку.

Згідно з українським рейтингом мов програмування 2024 від порталу DOU, TypeScript показав найкращий темп зростання за рік. У загальному рейтингу TypeScript посідає друге, лише трохи поступаючись JavaScript. Майже половина респондентів обрала б TypeScript для свого наступного фронтенд проєкту, у той час, як JavaScript обрала б тільки четвертина [4].

Підтвердженням популярності TypeScript — є поступова міграція таких великих технологічних компаній, як Microsoft, Google, Slack, Airbnb на дану мову програмування. Одним із найвідоміших прикладів є фреймворк Angular,

Додано примітку [1]: Новий абзац

розроблений компанією Google. Починаючи з другої версії, Angular був повністю переписаний на TypeScript. Таким чином архітектура Angular активно використовує усі переваги TypeScript (інтерфейси, класи, декоратори та ін.). Завдяки цьому Angular став потужним інструментом для корпоративної розробки великих комерційних вебзастосунків.

Вищезазначені факти свідчать про те, що розробники дедалі більше цінують структуровану розробку у TypeScript, що робить його надійним і перспективним вибором для сучасної веброзробки.

2.2 Порівняння TypeScript з JavaScript

TypeScript є надбудовою над JavaScript, яка додає статичну типізацію, покращену структуру та можливості організації коду, що особливо корисно для великих проєктів та командної роботи.

TypeScript підтримує використання файлів визначень, які містять інформацію про типи для бібліотек [5]. Це подібно до того, як у мові C++ заголовні файли містять опис структури об'єктних файлів. Ці файли дозволяють іншим програмам використовувати значення, визначені бібліотеках, як статично типізовані сутності TypeScript, забезпечуючи кращу інтеграцію з бібліотеками.

TypeScript забезпечує зручність під час розробки, але потребує додаткового кроку компіляції у JavaScript код перед виконанням, оскільки браузери не здатні працювати безпосередньо з Typescript. У той час як JavaScript інтерпретується без необхідності компіляції [6, 7].

Оскільки TypeScript є зручною обгорткою JavaScript, тому їх синтаксис подібний. TypeScript можна розглядати як проміжний варіант між більш складними мовами програмування, такими як C# або Java, і простішими, як Python. Він підтримує модифікатори доступу (*public, private* та *protected*), має об'єктно-орієнтовану парадигму та використовує крапку з комою для завершення рядків.

Серед типів даних є всі основні, такі як *boolean, number, string, array, tuple, enum* [6]. Та є і більш просунуті та унікальні типи, такі як *union, interface, null, type* та інші [8].

Union типи дозволяють комбінувати та об'єднувати різні типи за допомогою логічних операторів [5]. Наприклад, зробити щоб змінна могла бути одночасно рядком або числом, це робить системи набагато гнучкішою.

У TypeScript існує два основні підходи для створення об'єктів: використання *Type* або *Interface* [8]. Type можна розглядати як аліас (псевдонім), оскільки його можна застосовувати для перейменування примітивних типів або *Union* типів. Це спрощує передачу аргументів у функції та робить код зручнішим для читання та використання. *Interface* дуже схожий до *Type*, але є і відмінності. *Interface* не може використовуватися як аліас для інших типів.

Однак, *Interface* підтримує розширення, що дозволяє додавати нові властивості в процесі виконання. Така гнучкість допомагає уникнути дублювання коду та покращує масштабованість.

Особливо потужними є такі типи, як *null, undefined* та *unknown*, за умови їх правильного використання. Завдяки строгій типізації, ці типи дозволяють більш точно контролювати порожні або невизначені значення [5]. Таке буде корисно в ситуаціях, коли змінні або об'єкти можуть не мати значення або бути відсутніми. Наприклад, у разі помилки запиту до бази даних, змінна залишиться невизначеною, що, в свою чергу, дозволить розробнику врахувати цей момент та додати додаткову перевірку з подальшим відображенням помилки для користувача.

2.2.1 Переваги TypeScript

Найбільш очевидною перевагою є статична типізація, яка допомагає уникати типових помилок та робить код надійнішим. Завдяки статичній типізації розробники можуть виявляти помилки ще на етапі компіляції, а не під час виконання коду. Серед інших переваг, завдяки яким TypeScript стає популярним вибором для великих проєктів, є [9]:

- Розширені можливості для створення *generics* (узагальнених методів чи класів), інтерфейсів і модулів, що полегшує створення складних архітектур у великих застосунках.
- Кращу підтримку для рефакторингу коду, включаючи перейменування змінних, виділення методів та інші операції. Крім того, інтегровані середовища розробки (IDE) надають корисні інструменти автодоповнення.
- Підтримка інструментів для побудови, тестування та відлагодження коду, таких як Webpack, Babel, Jest та інші.

2.2.2 Недоліки TypeScript

Попри значні переваги, TypeScript має й певні недоліки. Один із основних недоліків — це необхідність компіляції коду. TypeScript не виконується безпосередньо браузерами або іншими середовищами виконання, тому перед запуском він має бути перетворений у JavaScript. Але це додає додатковий етап у процесі розробки, який може збільшувати час компіляції, особливо у великих проєктах. Також можна зазначити, що [10]:

- TypeScript вимагає детального розуміння типізації та суворішого підходу до написання коду, що може потребувати додаткового часу для навчання й адаптації.
- Не всі бібліотеки та фреймворки повністю підтримують TypeScript, що іноді змушує розробників комбінувати його з JavaScript, ускладнюючи таким чином роботу.

2.3 Фреймворк React для веброзробки

Для розробки проекту було обрано бібліотеку React, яка є однією з найпопулярніших технологій у сфері фронтенд-розробки. Вибір обумовлений наявністю детальної документації та широкої спільноти розробників, яка забезпечує підтримку розробників-новачків і сприяє розвитку бібліотеки.

React — це популярний JavaScript-фреймворк, який був розроблений компанією Meta (Facebook) у 2013 році [11]. Головним призначення є створення комплексних інтерфейсів користувача. Особливістю, за яку багато хто полюбив React, є його орієнтованість на компонентний підхід, що дозволяє розробникам створювати вебдодатки зі складною логікою, легкою підтримкою та модульною архітектурою. React забезпечує високу швидкість і ефективність роботи застосунків завдяки своєму механізму віртуального Document Object Model (DOM), який оптимізує оновлення сторінки та знижує навантаження на браузер [12].

Основна мета React полягає в тому, щоб зробити процес розробки масштабованих інтерфейсів користувача легким і швидким. React має відкритий код, тому будь-хто бажаючий може долучитися до вже великої спільноти, яка створює додаткові інструменти та розширення [11]. Це дозволяє розробникам комбінувати React з користувацькими бібліотеками для створення ще більш потужних вебзастосунків.

React функціонує на принципах компонентного підходу, іншими словами вся структура вебдодатку складається з окремих, автономних компонентів. Кожен компонент в React являє собою незалежну частину інтерфейсу, що має свої властивості і стан. По-перше, можна повторно використовувати компоненти в різних частинах додатку, що сприяє модульності та економії часу на розробку. Компоненти в React можуть бути як функціональними, так і класовими, причому функціональні компоненти стали стандартом завдяки впровадженню хуків, таких як *useState*, *useEffect* та інші. По-друге, такий підхід робить підтримку великих проєктів простішою, оскільки дозволяє легко оновлювати чи змінювати окремі частини застосунку без необхідності зміни всієї кодової бази. Наприклад, завдяки компонентам можна мати загальну, звичайну, кнопку з однаковим розміром і стилем та використовувати цей компонент у всіх потрібних місцях. Коли потрібно буде внести якісь зміни, такі як зміна основного кольору, додавання заокруглення, тощо, тоді це можна буде зробити в одному місці і зміни торкнуться усього проєкту.

Віртуальний DOM є ще однією особливістю React. Коли користувач взаємодіє з інтерфейсом, наприклад, вводить щось чи натискає на кнопку, створюється копія основного DOM у пам'яті, яка відображає зміни, не оновлюючи відразу всю сторінку. Механізм порівнює попередню та нову версії віртуального DOM і вносить мінімальні зміни в основний DOM, який бачить користувач, що великою мірою оптимізує продуктивність. Такий підхід дозволяє створювати динамічні та інтерактивні інтерфейси з меншою кількістю навантажень на браузер, що є важливим для великих застосунків з великою кількістю елементів.

З впровадженням хуків у версії 16.8 React суттєво змінив підхід до управління станом і логікою компонента. Хуки, такі як *useState*, *useEffect*, *useContext* та інші, дозволяють функціональним компонентам виконувати ті ж завдання, що й класові компоненти, але з більш простим та зручним синтаксисом [11]. *useState* дозволяє встановлювати й оновлювати стан компонента, а *useEffect* — відстежувати зміни та керувати побічними ефектами, такими як запити до сервера або оновлення DOM. Хуки спрощують код і роблять його легшим для розуміння, що позитивно впливає на командну розробку та підтримку застосунків.

React зарекомендував себе як ефективний і потужний інструмент для веброзробки завдяки поєднанню гнучкості, високої продуктивності та простоти у розробці. Сьогодні він є стандартом для побудови сучасних вебзастосунків, що здатні швидко реагувати на дії користувача і забезпечувати високу продуктивність. У поєднанні з TypeScript, React стає ще потужнішим, дозволяючи створювати масштабовані, безпечні та надійні інтерфейси для будь-яких задач у веброзробці.

2.3.1 Допоміжні бібліотеки для React

Додано примітку [2]: Новий розділ

Попри потужність самого React у розробці вебзастосунків, він передусім сфокусований на візуальному представленні даних та не охоплює усіх аспектів повноцінної розробки. Наприклад, немає якісних вбудованих рішень для

маршрутизації між сторінками, глобального зберігання стану, або обробки HTTP запитів. Для вирішення даних задач існують такі бібліотеки як **Redux, React Router** та **Axios**. Вони спрощують реалізацію та пришвидшують розробку продукту, оскільки більшість рутинних задач вже вирішено великою спільнотою React розробників.

Redux — це бібліотека для управління глобальним станом застосунку, яка дозволяє централізовано зберігати всі важливі дані, що впливають на інтерфейс [13]. Ідея Redux базується на тому, що весь стан програми представлено у вигляді єдиного об'єкта *state*, який зберігається у своєму сховищі (*store*). Будь-які зміни стану здійснюються виключно через спеціально створені *reducers* (редуктори), які приймають попередній стан і дію, та зберігають новий стан. Якщо якась частина програми змінилась, програма про це не дізнається, а значить не зможе відреагувати, наприклад перемалювати потрібну частину екрана. Redux вирішує цю проблему. Зміна даних всередині сховища породжує події, на які можна підписуватись і виконувати довільну логіку.

Отже, незалежно від ієрархії вебзастосунку, будь-який компонент системи може мати доступ до загального *state,* без потреби передавання інформації по ланцюгу від батьківського до дочірнього елемента, зверху униз. Наприклад, користувач авторизується, і його дані мають бути доступні як у хедері домашньої сторінки, так і в особистому кабінеті. Замість передавання цієї інформації через параметри на кілька рівнів униз, компоненти можуть просто звертатися до глобального стану Redux.

React Router — це спеціалізована бібліотека для створення маршрутизації у React проєктах [14]. Вона дозволяє реалізовувати навігацію між компонентами за допомогою зміни посилання, але все одно зберігає односторінковий підхід. Бібліотека надає інструменти для визначення маршруту, відображення відповідного компонента, доступу до *url* параметрів, навігації через посилання або програмно, а також підтримки вкладених і динамічних маршрутів. За допомогою вкладених маршрутів можна компонувати структуру сторінок, де одна частина залишається сталою (наприклад, заголовок), а вміст змінюється залежно від

активного маршруту. Можна змінити тільки назву у заголовку, а інші компоненти у заголовку залишаються сталими.

Axios — це легка бібліотека на основі промісів, що дозволяє надсилати усі види REST запитів (GET, POST, DELETE та ін.). Хоча сам TypeScript має вбудований інтерфейс *fetch* для здійснення асинхронних інтернет запитів, у реальній розробці частіше використовується Axios через його простоту синтаксису та більший контроль над конфігурацією самого запиту. Також бібліотека дозволяє створювати загальний клієнт із загальними параметрами для подальшого здійснення запитів, що є дуже зручним рішенням у випадку великої кількості підключених сервісів.

```
1  const defaultOptions = {
2      withCredentials: true,
3      headers: {
4          'Content-Type': 'application/json',
5      },
6  };
7
8  const instance = axios.create(defaultOptions);
```

Рисунок. 2.3. Приклад ініціалізації Axios

На рис. 2.3 зображено приклад створення спеціалізованого екземпляра клієнта Axios з попередньо налаштованими параметрами. У змінній *defaultOptions* задаються два параметри. Перший — *withCredentials: true,* вказує, що при кожному запиті повинні надсилатися *cookies*, зазвичай це токени авторизації, що зберігаються на клієнтській стороні у HTTP-only cookies. Другий — *headers: Content-Type: application/json*, який повідомляє, що дані, передані у тілі запиту, мають формат JSON [15]. Ще однією перевагою є підтримка *interceptors* (перехоплювачів). Це спеціальні функції, які дозволяють виконувати певні дії перед безпосередньою обробкою запита або відповіді [15]. Найпопулярнішим випадком використання є відповідь 401 (не авторизовано) від сервера, тоді

interceptor може спробувати повторно авторизувати або перенаправити користувача на сторінку авторизації.

```
getAllStaff(listener: ApiCallListener<AllStaff>) {
    axios.get<AllStaff>(`${this.baseUrl}/staff/all`)
        .then(response => {
            listener.onSuccess(response.data);
        })
        .catch(error => {
            listener.onFailure(error.message);
        });
};
```

Рисунок. 2.4. Приклад використання Axios

На рис. 2.4 продемонстровано використання раніше створеного Axios клієнта для здійснення GET-запиту. Метод *axios.get()* має один обов'язковий параметр *url*, туди передається сформоване посилання *${this.baseUrl}/staff/all.* Типізація *<AllStaff>* вказує, що у відповіді очікується об'єкт AllStaff, що дозволяє TypeScript перевіряти структуру отриманих даних ще до виконання коду. Далі в ланцюжку викликів використовується *.then()* який виконується лише у випадку успішного завершення запиту [15]. У параметрі *response* зберігається об'єкт відповіді сервера, а саме тіло AllStaff знаходиться у *response.data*. У разі виникнення помилки або непередбаченої відповіді API, виконується блок *.catch()* із параметром *error.message*, який містить коротке текстове повідомлення про причину помилки. Дані приклади використання добре ілюструють, як зручно Axios дозволяє надсилати запити та обробляти їхню відповідь.

РОЗДІЛ 3. ПРОЄКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ ТА РОЗРОБКА КОРИСТУВАЦЬКОГО ІНТЕРФЕЙСУ

3.1 Архітектура вебзастосунку

Архітектура вебзастосунку є дуже важливим етапом перед початком розробки системи, оскільки правильна архітектура визначає, як різні компоненти системи взаємодіють між собою, обмінюються даними та забезпечують необхідну продуктивність і масштабованість. Таким чином можна створити гнучку систему, яка не лише підтримуватиме високі навантаження, але й забезпечуватиме легке оновлення та розширення функціоналу у майбутньому.

На етапі проєктування архітектури важливим інструментом є Unified Modeling Language (UML) — уніфікована мова моделювання, що дозволяє візуально представляти структуру та поведінку системи. Мова використовується для створення різних типів діаграм. Це допомагає команді розробників краще розуміти систему та взаємодію між її компонентами ще до початку реалізації.

UML має багато типів діаграм, які поділяються на дві категорії. Деякі типи представляють структурну інформацію, а решта представляють загальні типи поведінки [16]. Структурні діаграми представляють статичні аспекти системи, а саме те що повинно бути присутнім у системі для документування її архітектури. Діаграми поведінки представляють динамічний аспект системи та підкреслюють, що повинно відбуватися в системі, яка моделюється.

Діаграма пакетів належить до структурних діаграм та використовується для відображення логічної організації системи на високому рівні, групуючи класи та компоненти в пакети (модулі) для кращого розуміння структури проєкту. Така діаграма дозволяє чітко організувати проєкт, визначити залежності між різними модулями та забезпечити зрозуміле відображення архітектури системи. Кожен пакет містить відповідні класи та компоненти, що забезпечують функціональність цієї частини системи. Діаграма пакетів допомагає відстежувати залежності між різними частинами застосунку, вказуючи на важливі зв’язки та можливі точки інтеграції [16].

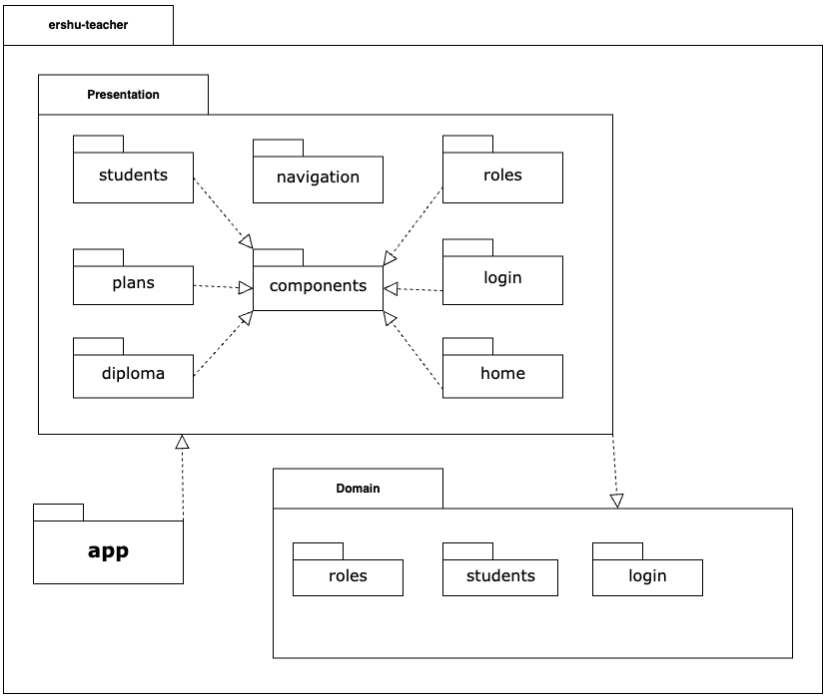

Рисунок. 3.1. Діаграма пакетів вебзастосунку

Діаграма на рис. 3.1 складається з двох основних рівнів — Presentation та Domain. Пакет *app* являється точкою відправлення і об’єднує все в єдине рішення.

Рівень *Presentation* відповідає за інтерфейс користувача та містить основні компоненти, які забезпечують взаємодію з користувачем. Такі пакети як *home, students, plans, diploma, roles ma login* є основними сторінками з якими безпосередньо взаємодіє користувач. *Navigation* забезпечує навігацію у системі, поєднуючи всі функції в єдине ціле. *Components* має спільний набір компонентів, який використовується іншими пакетами на рівні *Presentation* для уніфікації інтерфейсу та функціональності.

Рівень *Domain* містить основну бізнес-логіку та визначає ключові об’єкти, що спрощує взаємодію з усіма мікро сервісами.

Ця структура дозволяє розділити інтерфейс користувача та бізнес-логіку, що робить підтримку та масштабування системи простішою в майбутньому.

Діаграма прецедентів належить до поведінкових діаграм та відображає функціональні можливості системи з точки зору користувачів (акторів). Вона ілюструє, які дії можуть виконувати різні актори, визначаючи основні сценарії взаємодії з системою [16]. Діаграма дозволяє ідентифікувати вимоги до програмного забезпечення, визначити основні функції та забезпечити розуміння того, як система задовольняє потреби користувачів. Основна мета — візуально представити взаємодію між користувачами та системою, визначивши зв'язок між актором і випадками використання.

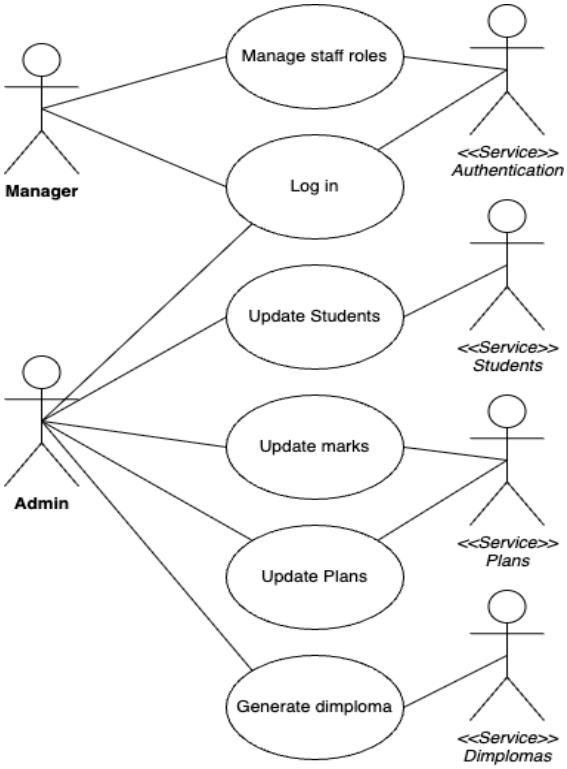

Рисунок 3.2. Діаграма прецедентів системи

На рис. 3.2 діаграма демонструє основні функції та взаємодії між користувачами та системою електронного документообігу. Основні актори цієї системи — це *Manager* та *Admin*, які мають різні права та можливості в межах

системи. Та мікросервіси - *Authentication*, *Students*, *Plans* та *Diplomas***,** які забезпечують швидку та коректну роботи всієї системи, як єдине ціле.

Manager може здійснювати вхід у систему та керувати ролями інших співробітників. Цей користувач може не тільки авторизуватись у системі, але й надає доступ іншим користувачам, призначає ролі та регулює права.

Admin має більше процесів, пов'язаних саме з документами. Як і *manager*, він може здійснювати вхід у систему. Основними функціями будуть оновлення внутрішньої бази даних з студентами та навчальними планами. Також доступна функція занесення балів у систему для кожного студента після завершення навчального семестру. Коли студент завершить навчання, можна буде легко згенерувати диплом на основі вже введених оцінок. Завдяки цьому можна значною мірою прискорити процес видачі диплому випускнику та мінімізує фактор людської помилки.

Додано примітку [3]: Видалено 3.2 Взаємодія з сервісами. Ширше описано в 4.3

3.2 Розробка вебінтерфейсу

Розробка вебінтерфейсу є відповідальним етапом у створенні сучасних систем, адже це саме та частина, яку безпосередньо бачить користувач. Тому інтерфейс має забезпечувати зручність та інтуїтивність у використанні, а також й ефективність у виконанні основних функцій системи. У освітньому середовищі, де кінцеві користувачі мають різні рівні технічної підготовки, зрозумілий інтерфейс є дуже важливим щоб можна було виконувати різноманітні задачі без довгої підготовки.

Рисунок. 3.3. Головна сторінка

На головній сторінці (Рис. 3.3) користувачу представлено три основні компоненти, а саме заголовок, вихід з системи та набір доступних функцій. Зверху по центру, у заголовку, зазначено ім’я та прізвище користувача, що забезпечує краще розуміння який акаунт зараз використовується, особливо у випадку, якщо їх декілька. Праворуч від заголовка розташована кнопка, яка дає змогу швидко вийти з системи за потреби. Під заголовком розміщені елементи з усіма доступними функціями. Кожна картка має свою назву, яка означає конкретну операцію, яку можна виконати натиснувши на цю кнопку. Також, кожна картка має свій рівень доступу і її видимість для користувача напряму залежить від його повноважень. Що дозволяє дуже гнучко налаштовувати рівні доступу для розподілення роботи між усіма співробітниками та забезпечує безпеку даних.

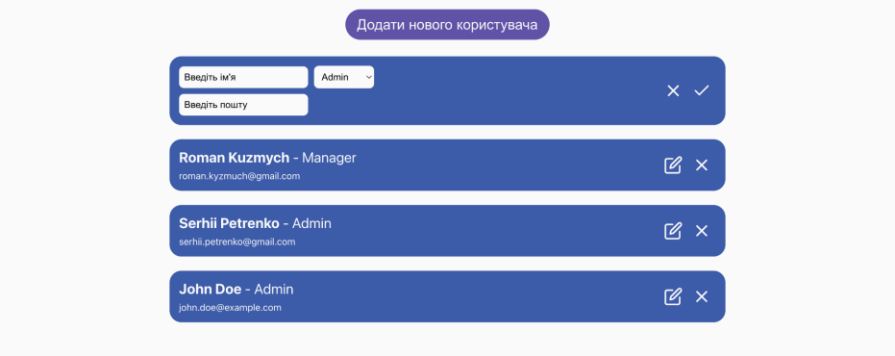

Рисунок. 3.4. Сторінка для редагування користувачів системи

На рис. 3.4 зображений функціонал для додавання нових користувачів, видалення старих та редагування вже існуючих користувачів. Кожен елемент списку складається з інформації про працівника та кнопок для маніпуляцій. У першому рядку зазначене ім'я та прізвище і роль користувача у системі. Нижче є пошта для кращої ідентифікації.

Праворуч від інформаційного блоку розташовані кнопки, що дозволяють виконувати різні операції з користувачем. Хрестик у даному випадку відповідає за видалення, а олівець за редагування існуючого користувача.

Після натискання на кнопку олівця ім'я користувача замінюється на поле для введення нових даних. А пошта залишається текстовим рядком, тому що вона є унікальним ідентифікатором користувача і не може бути змінена. Роль користувача замінюється на випадаючий список, який містить всі доступні ролі, що дозволяє легко змінювати рівень доступу співробітника в системі. При цьому, на місці попередніх кнопок з'являються нові елементи управління: кнопка хрестика, що скасовує редагування, та кнопка у вигляді галочки, що підтверджує внесені зміни. Завдяки цим кнопкам можна швидко відмовитись від редагування або зберегти нові налаштування без необхідності додаткових підтверджень чи переходу до інших етапів.

При додавання нового співробітника інтерфейс працює за аналогією з редагуванням вже існуючого користувача, надаючи ті самі поля для введення

даних, що забезпечує уніфікацію процесу. Така уніфікація інтерфейсу дозволяє знизити час, необхідний для освоєння функціоналу, оскільки користувачам не потрібно вивчати різні методи для редагування та додавання співробітників, що робить взаємодію з системою інтуїтивно зрозумілою та зручною.

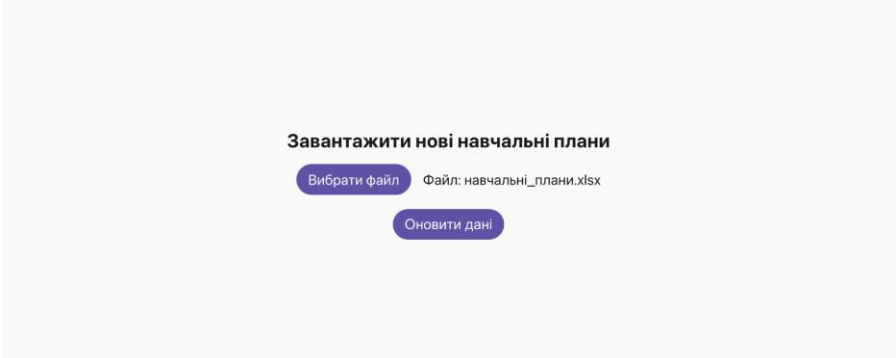

Рисунок. 3.5. Сторінка для завантаження нових документів

Функціонал, що дозволяє оновлювати документи, такі як навчальні плани, інформацію про студентів та інші важливі дані, без потреби ручного оновлення у бекенд сервісах. Такий підхід дозволяє спростити процеси адміністрування, знижуючи ризики помилок і забезпечуючи актуальність даних в режимі реального часу.

Інтерфейс для оновлення документів (рис. 3.5) надає співробітникам можливість завантажити оновлений файл з навчальними планами або іншою важливою інформацією для подальшого збереження в базі даних відповідного мікросервісу. У результаті можна миттєво використовувати нові дані для подальших операцій у системі.

Процес завантаження документів відбувається у два етапи. Спочатку співробітник вибирає необхідний файл для завантаження. Потім, після вибору файлу система виконує завантаження вибраного документа на сервер, де він піддається подальшій обробці та валідації перед збереженням у базі даних.

Отже, правильне проєктування архітектури та інтерфейсу вебзастосунку є фундаментом для створення ефективної та зручної системи. Вибір оптимальних

технологій, продумана організація взаємодії мікросервісів та забезпечення гнучкості у майбутньому дозволить забезпечити надійність та стабільність системи в умовах зростаючого навантаження.

РОЗДІЛ 4. РОЗРОБКА ВЛАСНОГО СЕРВІСУ АВТОРИЗАЦІЇ

4.1 Механізми захисту сервісу авторизації

Процес побудови безпечної системи електронного документообігу вимагає особливої уваги до способу автентифікації користувачів. Використання більш звичного підходу з локальними обліковими записами, де кожен користувач реєструється окремо, супроводжується низкою ризиків. Оскільки, логіни та паролі користувачів повинні зберігатися у певному сховищі, це створює потенційну вразливість у разі кібератаки. Крім того, користувачам доводиться запам’ятовувати ще один набір облікових даних, а у разі втрати облікових даних потрібно забезпечити процес відновлення логіна та пароля. Загалом, з’являється багато вразливостей для витоку конфіденційної інформації або компрометації всієї системи.

4.1.1 Переваги використання Google OAuth для авторизації

Проаналізуємо принципи роботи протоколу OAuth. Це протокол делегування, засіб, що дозволяє людині, що контролює ресурс, дозволити програмному додатку доступ до ресурсу від свого імені, не видаючи себе за нього. Програма запитує авторизацію у власника ресурсу та отримує маркери, які він може використовувати для доступу до ресурсу. Все це відбувається без необхідності додатку видавати себе за людину, яка контролює ресурс, оскільки токен явно представляє делеговане право доступу. Наприклад, ключ паркувальника автомобіля дозволяє власнику автомобіля надати обмежений доступ комусь, не передаючи при цьому повний контроль у вигляді ключа власника. Такі ключі можуть обмежити максимальну швидкість автомобіля і навіть заглушити його, якщо він проїде більше ніж на встановлену відстань від початкової точки, посилаючи попередження власнику. Так само токени OAuth можуть обмежити доступ клієнта тільки тими діями, які власник ресурсу делегував.

Google OAuth 2.0 дозволяє безпечно отримувати обмежений доступ до ресурсів Google-акаунта користувача без необхідності розкриття пароля. При використанні OAuth користувачі підтверджують свою особу через перевірений сервіс Google, після чого система отримує доступ лише до попередньо узгоджених даних, таких як електронна пошта чи ім’я користувача. Оскільки паролі не передаються та не зберігаються на стороні сервера документообігу, то зменшується можливість потенційної втрати чутливої інформації. Усі операції аутентифікації проводяться через інфраструктуру Google, яка підтримує багаторівневу автентифікацію: виявлення підозрілої активності, шифрування запитів і багато інших захисних заходів. Для самостійної реалізації таких механізмів потрібно було б витратити багато часу та ресурсів. Крім того, інтеграція з Google забезпечує високу зручність користування. Працівники навчального закладу можуть входити до системи за допомогою вже наявних корпоративних акаунтів.

Отже, використання Google OAuth дозволяє не лише підвищити рівень безпеки даних у системі, але й суттєво спростити процес входу для користувачів. Завдяки інтеграції з перевіреною інфраструктурою автентифікації, система електронного документообігу отримує надійний фундамент для подальшого розвитку з урахуванням сучасних вимог до захисту персональних даних та стійкості до кібератак.

4.1.2 JWT токен для надійного захисту запитів

JSON Web Token (JWT) — це відкритий стандарт передачі інформації у зашифрованому вигляді між сторонами, який дозволяє швидко і безпечно валідувати запити. Структура JWT токена складається з трьох частин: заголовка (header), корисного навантаження (payload) та цифрового підпису (signature). *Header* описує тип та метод шифрування, *payload* містить основні дані про користувача, а *signature* гарантує, що токен не був змінений сторонніми особами. Будь-яка спроба змінити дані токена призведе до невідповідності підпису, що буде негайно виявлено сервером і запит буде відхилено. Завдяки своїй самодостатності

та компактності JWT ідеально підходить для використання у системі з мікросервісною архітектурою, де важливо швидко та надійно перевіряти права доступу незалежно від сервісу [17].

Серед переваг використання JWT можна виділити ефективність перевірки автентичності користувачів. Оскільки вся необхідна інформація передається у тілі самого токена, система розпізнає авторизованого користувача без потреби здійснення податкових перевірок. Крім того, у мікросервісній архітектурі, де декілька серверів обробляють запити одночасно, кожен із них може незалежно перевіряти підпис токена у централізованому сервісі. У результаті з'являється можливість легко масштабувати систему горизонтально, додаючи нові сервери без ускладнень у процесі автентифікації.

Для забезпечення високого рівня безпеки при використанні JWT токенів у системі було впроваджено низку додаткових безпекових заходів. Ключовим, серед них, є встановлення обмеженого терміну дії *access* токена у дві години з моменту його видачі. Навіть якщо *access* токен буде перехоплено, його використання буде можливим лише протягом короткого проміжку часу, що суттєво підвищує загальну стійкість системи до атак. Для забезпечення безперервності роботи користувача, паралельно з *access* токеном видається *refresh* токен з довшим терміном дії — дві доби. Такий підхід дозволяє автоматично поновлювати короткочасний *access* токен без необхідності повторної автентифікації через Google OAuth. Таким чином можна безперервно користуватися системою електронного документообігу та виконувати свої завдання, а система зберігає безпеку, оскільки *refresh* токен також підлягає окремій перевірці та може бути відкликаним за необхідності. Після виходу користувача з системи спрацьовує механізм утилізації токенів користувача, інакше кажучи усі пов'язані з ним *access* та *refresh* токени негайно позначаються як недійсні у спеціальному сховищі, таким чином, навіть якщо термін дії токен ще не вичерпався, їх наступне використання буде неможливим. Тобто, якщо користувач дізнався, що його акаунт було перехоплено зловмисниками він може здійснити вихід з системи, що автоматично деактивує його токени та унеможливить їх подальше використання сторонніми особами.

З боку клієнтської частини, токени зберігаються у спеціальних HTTP-only cookies. Це означає, що доступ до токенів буде можливий лише через серверні запити і не дозволить отримати їх через JavaScript код у браузері. Такий підхід у зберіганні токенів суттєво обмежує ризики кібератак та гарантує потужний захист процесу автентифікації користувачів у електронній системі документообігу, що є критично важливим для забезпечення цілісності та конфіденційності даних у закладі вищої освіти.

4.2 Розробка сервісу авторизації за допомогою Ktor

Для реалізації авторизаційного мікросервісу було обрано фреймворк Ktor, розроблений JetBrains спеціально для створення асинхронних мікросервісів на мові Kotlin [18]. Вибір цього фреймворку обумовлений наявністю якісної документації, важливими перевагами над конкурентами та стрімким розвитком з великими перспективами у майбутньому. Його основною перевагою є висока гнучкість та низький поріг входу для Kotlin-розробників. Модульна структура дозволяє створювати продуктивні, легкі та масштабовані серверні додатки без надмірного навантаження на систему. На відміну від Spring Boot, який позиціонується як повноцінний і розширюваний фреймворк із численними рішеннями, з кількістю готових рішень для корпоративних систем та має велику екосистему, автоматичну конфігурацію, та численні вбудовані інструменти для безпеки, баз даних, моніторингу, логування тощо. Також, пришвидшується розробка, але може ускладнитися кастомізація і збільшитися обсяг ресурсоємності самого додатку. У свою чергу, Ktor дає змогу розробникам більш легку й мінімалістичну альтернативу. Його головна перевага полягає у високій гнучкості, що дозволяє самим визначати, які модулі потрібні, як саме обробляються запити, які механізми автентифікації та серіалізації використовуються. Це оптимізує надлишковий код та зменшує розмір самого проєкту. Серед основних переваг Ktor над Spring Boot [18]:

- Ktor написаний на Kotlin та забезпечує його підтримку з коробки, на відміну від Spring Boot який написаний на Java та немає повної підтримки Kotlin.

- Всі запити та обробка працюють неблокуюче один одного для підвищення продуктивності при великій кількості одночасних користувачів.
- Новий сервер можна запустити буквально з кількох рядків коду без складної конфігурації XML та YAML.
- Кількість залежностей буде менша, і тим самим буде вища продуктивність і гнучкість розгортання, тому що розробник сам контролює компоненти, які реально використовуються.

У рамках розробки було створено повноцінний мікросервіс авторизації, який реалізує вхід користувача через Google OAuth 2.0, генерацію JWT токенів, а також перевірку дозволів та захист доступу до закритих ресурсів.

Для зберігання інформації про неактуальні токени використовується сховище типу «ключ-значення» Redis, яке працює у пам’яті [19]. Воно підтримує встановлення часу життя записаних даних (TTL), що ідеально підходить для зберігання токенів з терміном дії. У сервісі авторизації Redis використовується для зберігання *access* та *refresh* токенів, після того як користувач вийшов з системи електронного документообігу. Токен зберігається як ключ, а його тип (*access/refresh*) — як значення. Під час обробки запиту система перевіряє, чи міститься цей токен у Redis. Якщо так — авторизація відхиляється, навіть якщо термін дії токена ще не закінчився. Цей підхід дозволяє реалізувати додатковий рівень безпеки, токен не можна повторно використати після завершення сесії.

Для зберігання критично важливих секретів у проєкті, таких як Google client id чи JWT sign secret було створено локальний файл .env. Використання такого файлу є сучасним стандартом у розробці бекенд сервісів. Воно дозволяє зберігати конфіденційні значення окремо від коду, не включаючи їх до Git репозиторію, що покращує безпеку. Крім того, це забезпечує гнучкість: одна й та сама система може запускатись у різних середовищах (*test* або *production*) лише з різними значеннями у .env файлі без жодної зміни в коді.

Дані про працівників закладу освіти, що безпосередньо працюють з електронною системою документообігу зберігаються у реляційній базі даних PostgreSQL. Таке рішення було обране з кількох причин. По-перше, реляційні бази

добре підходять для зберігання структурованих даних із чітко визначеними зв’язками та обмеженнями, наприклад кожен email повинен бути унікальним [20]. По-друге, вони забезпечують просту реалізацію перевірок, фільтрів та запитів, необхідних у процесі авторизації. У базі створена таблиця співробітників, яка містить електронну пошту (унікальний ключ), повне ім’я користувача та його роль у системі. Під час входу користувача у систему електронного документообігу система перевіряє чи присутній його email у таблиці з виданою роллю. Це дозволяє контролювати доступ лише для працівників, що були попередньо підтверджені. Тому, база даних також слугує як механізм верифікації та джерело даних для надання ролей у JWT токені.

Для ефективного керування ролями було розроблено набір ендпоїнтів, що забезпечують створення, редагування, перегляд та видалення записів про користувачів електронної системи документообігу.

1. **GET** (*/roles*) — повертає перелік доступних ролей у системі, які можуть бути призначені працівникам.
2. **GET** (*/staff*) — повертає інформацію про поточного користувача на основі переданого токену.
3. **GET** (*/staff/all*) — повертає список усіх зареєстрованих працівників у електронній системі документообігу.
4. **POST** (*/staff*) — записує нового працівника у базу даних з вказаними ролями, ім'ям та електронною поштою, якщо такої пошти ще немає.
5. **PATCH** (*/staff*) — дозволяє змінювати ім’я та роль існуючого працівника на основі його електронної пошти.
6. **DELETE** (*/staff/{email}*) — видаляє працівника з системи на основі його електронної пошти.

Дані запити використовуються інтерфейсом, зображеним на рис. 3.4, який забезпечує зручний доступ до основних операцій з користувачами.

4.3 Схема роботи авторизації у електронній системі документообігу

Добре спроєктована схема авторизації дозволяє точно контролювати, хто і в якій ролі має доступ до окремих функцій системи, забезпечує чітке розмежування повноважень, запобігає несанкціонованому доступу та зменшує ризик витоку конфіденційної інформації.

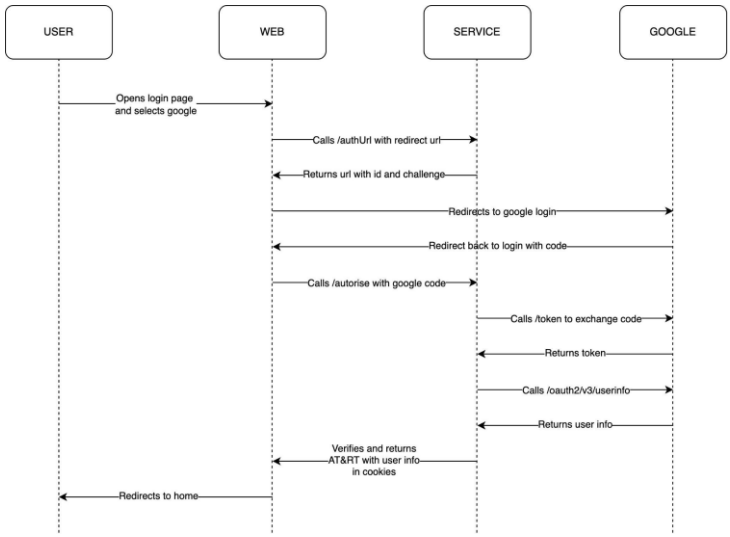

Рисунок. 4.1. Діаграма послідовності авторизації

На рис. 4.1 зображена детальна взаємодія між усіма задіяними елементами під час процесу авторизації, а саме: користувач (User), вебзастосунок (Web), внутрішній сервіс авторизації (Service) та зовнішній сервіс (Google). Послідовна діаграма відображає весь шлях — від першої та до останньої взаємодії користувача під час проходження авторизації у систему. Далі кожен кожен крок у діаграмі буде описано з відповідним рисунком інтерфейса.

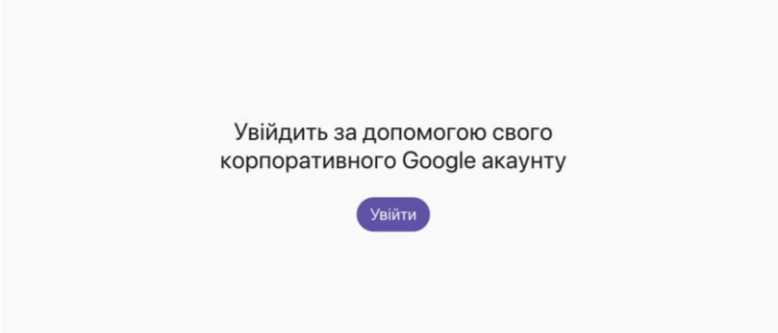

Рисунок. 4.2. Сторінка входу у систему

На рис. 4.2 зображено перший крок — сторінку, що просить користувача увійти за допомогою корпоративного акаунта закладу вищої освіти. Під час завантаження сторінки авторизації було здійснено GET-запит (*/googleAuthUrl*) на внутрішній сервіс авторизації з метою отримати посилання для автентифікації за допомогою Google OAuth. Посилання містить такі параметри:

- **Scope** — вказує, яка саме інформація нам буде доступна після успішної авторизації (openid+email+profile).
- **Response_type** — вибираємо, що нам має повернути Google у разі успішної авторизації (code).
- **Redirect_uri** — передаємо наше посилання на яке Google має повернути користувача після входу у акаунт, а також додати у посилання вище вказаний *code* з *response_type*, щоб система могла його зчитати та опрацювати.
- **Client_id** — секретний ключ від Google акаунта розробника, за допомогою якого налаштовується і працює Google OAuth.
- **Code_challenge** та **Code_challenge_method** — спеціальний зашифрований ключ, який перевіряється як на нашій стороні так і на стороні Google.

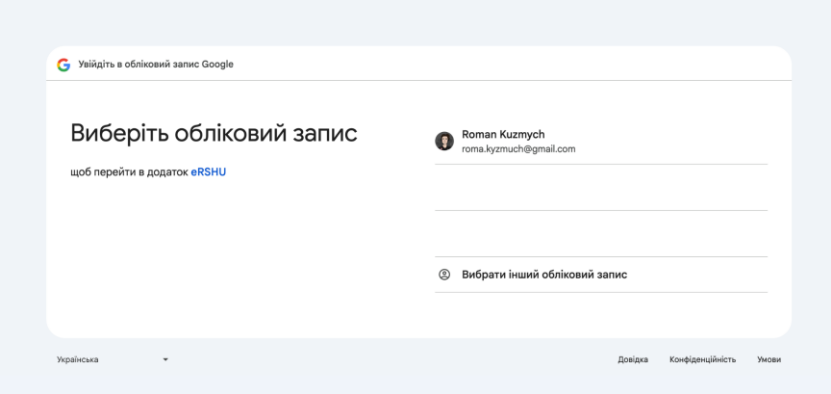

Рисунок. 4.3. Сторінка Google OAuth

На рис. 4.3 зображено вікно вибору Google акаунту, сюди юзер перенаправляється після натиску на кнопку увійти на рис. 4.2. Після підтвердження особи Google виконує редирект назад до вебзастосунку, додаючи *code* у параметрах вказаної *redirect_uri*.

Далі наш вебзастосунок автоматично зчитує цей код та відправляє на внутрішній сервіс для обміну на JWT токени. Для обміну вебзастосунок здійснює GET-запит (*/authorise*) на внутрішній сервіс разом з параметром *code*. Наступним кроком у процесі авторизації після отримання *authorization code* є його обмін на спеціальний *Google access token*. Це токен, який використовується для подальшої взаємодії з сервісами Google та отримання детальної інформації про автентифікованого користувача. Даний етап реалізується на стороні бекенд-сервісу та є критично важливим для завершення входу користувача у систему. Обмін коду на токен здійснюється через POST-запит (*/oauth2/token*) у тіло якого передаються кілька обов’язкових параметрів:

- **code** — авторизаційний код, що був переданий з вебзастосунку, після авторизації у Google;
- **client_id** та **client_secret** — секретні ключі від Google акаунта розробника;
- **redirect_uri** — посилання яке повинне співпадати з тим, яке використовувалося при авторизації у Google;

- **code_verifier** — код який був згенерований раніше і використовувався для отримання *code*.

У відповідь Google надсилає *access_token*, *id_token* (опціональний) та деякі додаткові поля, такі як *expires_in*. Отриманий *access* токен є короткочасним та діє протягом однієї години) і дозволяє надсилати авторизовані запити до Google API від імені користувача.

За допомогою отриманого токена, наш сервіс авторизації виконує GET-запит (*/oauth2/v3/userinfo*) з параметром *access_token*. У відповідь система отримує структуровану інформацію про користувача, що містить його повне ім’я, псевдонім та електронну адресу.

Наступний етап — це локальна перевірка користувача у системі. Насамперед, перевіряється, чи належить електронна адреса користувача до дозволеного домену закладу вищої освіти (у нашому випадку @rshu.edu.ua). Така перевірка дозволяє унеможливити доступ до системи стороннім користувачам, які використовують особисті Google акаунти. Другим етапом є перевірка наявності користувача в базі даних системи. Адміністратор повинен попередньо додати користувача до бази з відповідною поштою та роллю. Якщо запис відсутній — доступ до системи електронного документообігу не буде надано, навіть якщо автентифікація в Google пройшла успішно.

Завершальним етапом у процесі авторизації є генерація внутрішніх токенів. Система створює два JWT токени: *access* токен (2 годин) та refresh токен (2 доби). Access токен призначений для підтвердження автентичності користувача під час кожного запиту до захищених ресурсів. Усередині токена закодована пошта, ім’я користувача та його роль у системі. Ця інформація дозволяє серверу оперативно ідентифікувати користувача та визначити рівень доступу до функціоналу, не звертаючись щоразу до бази даних. Для забезпечення цілісності й неможливості підробки токен підписується на сервері за допомогою секретного ключа, відомого лише сервісу авторизації. *Refresh* токен не використовується безпосередньо для доступу до ресурсів, але слугує для отримання нового *access* токена після

завершення терміну його дії. Таким чином можна уникнути частих перенаправлень на авторизації через Google, зберігаючи безперервний користувацький досвід.

Сформовані токени надсилаються у відповідь клієнту у вигляді *HTTP-only cookies*. Такий спосіб зберігання є безпечним, оскільки cookie не доступна через JavaScript, що захищає її від XSS атак. Вони автоматично додаються до кожного запиту до сервера, спрощуючи архітектуру авторизації на фронтенді. Після цього користувач автоматично перенаправляється на головну сторінку системи, з вже активною сесією, де він може працювати відповідно до своєї ролі.

РОЗДІЛ 5. ІНТЕГРАЦІЯ ДОДАТКОВИХ МІКРОСЕРВІСІВ У ВЕБЗАСТОСУНОК

5.1 Обробка запитів з сервісу студентів

Першим етапом перед будь-якою взаємодією з даними студента є його вибір у системі, для однозначної ідентифікації конкретного здобувача освіти серед усіх, що навчаються у закладі освіти. Вибір студента виконується завдяки запитам до сервіса із студентськими даними та відбувається із послідовним вибором курсу, групи та студента. Така послідовна фільтрація полегшує роботу користувача системи коли потрібно швидко обрати одного із студентів. Після підтвердження вибору його унікальний ідентифікатор (*studentId*) буде використовується для подальших запитів, що стосуються зміни даних цього студента, наприклад отримання списку вибіркових предметів, або збереження оцінок.

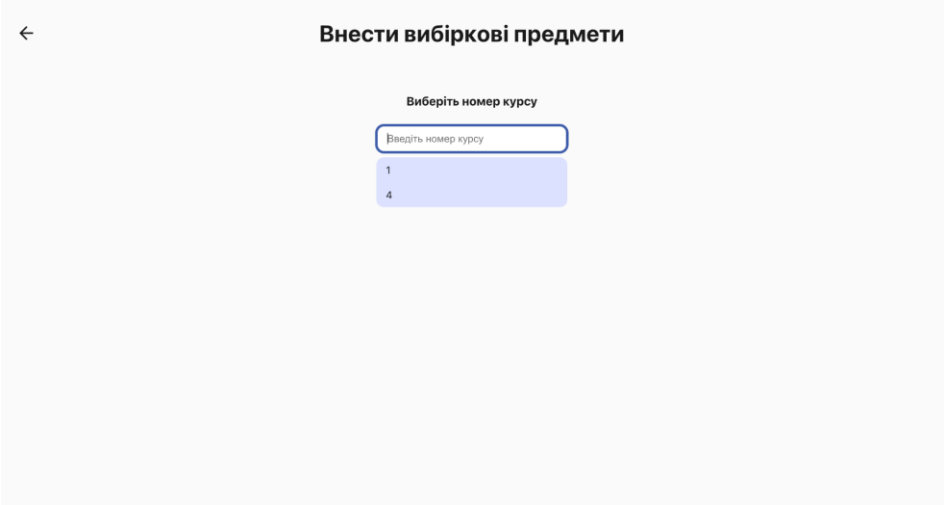

Рисунок. 5.1. Поле вибору номеру курсу

На рис 5.1 зображено перший крок, який дозволяє обрати номер курсу. Для цього відразу після відкриття сторінки, вебзастосунок надсилає GET-запит

(*/allyears*) для отримання списка усіх доступних курсів в університеті. Після отриманої відповіді від сервера, усі доступні роки навчання будуть додані у випадаючий список, як підказки. Вибір курсу дозволяє одразу відфільтрувати групи, доступні лише для цього року навчання, що, у свою чергу, зменшує обсяг даних, що передаються між клієнтом і сервером.

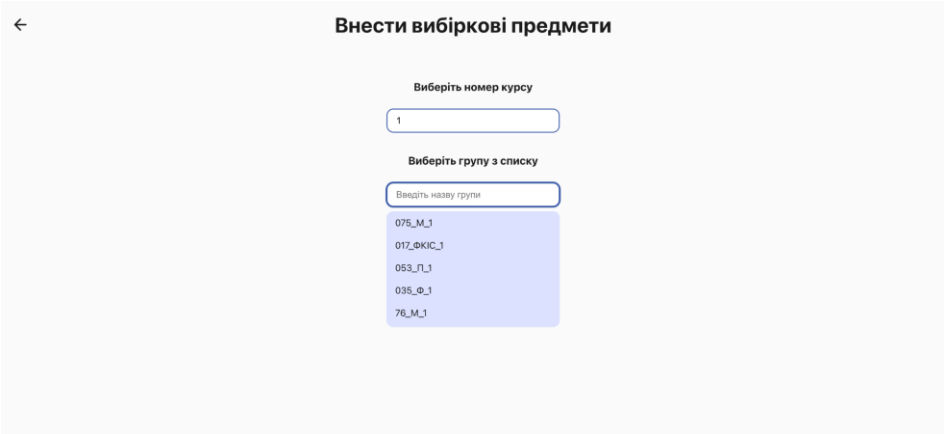

Рисунок. 5.2. Поле вибору групи

На рис 5.2 зображено наступний крок — вибір групи. Одразу після вибору курсу, надсилається ще одни GET-запит (*/groups/{year}*) з параметром *year* із попереднього кроку. У відповідь сервер відправляє перелік груп, що належать до обраного року навчання. Об’єкт групи складається із стандартних полів *id* та *group_name*, а також поля *students*, яке містить масив студентів, закріплених за цією групою у закладі вищої освіти. Такий підхід дозволив оптимізувати кількість запитів до сервера, що позитивно відобразилося на швидкості завантаження необхідного контенту у вебзастосунку. Після обробки отриманої відповіді, назви груп відображаються у вигляді підказок у випадаючому списку під полем для введення групи.

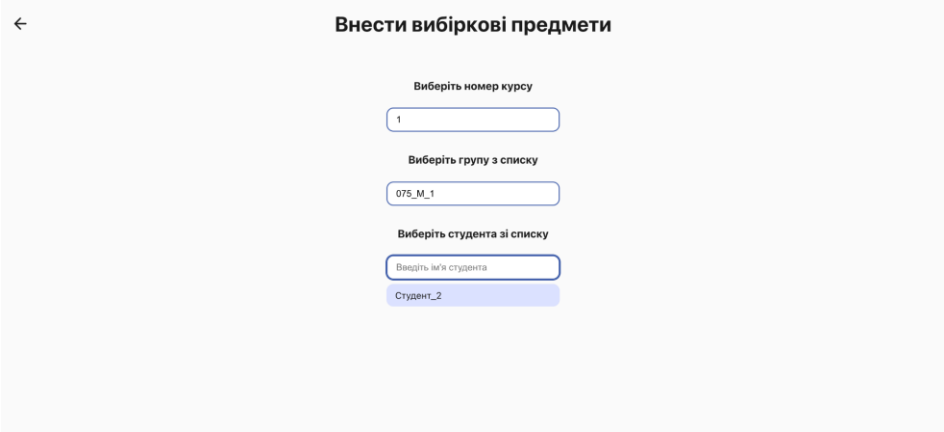

Рисунок. 5.3. Поле вибору студента

Після вибору групу, на екрані користувача з’являється останнє поле — вибір конкретного студента з групи (Див. рис. 5.3). Оскільки інформацію про студентів в групах ми вже отримали з попереднього запиту, відображення нового поля з підказками відбувається миттєво, що неабияк прискорює роботу з системою електронного документообігу. Отож, після остаточного вибору студента його дані, включаючи унікальний ідентифікатор будуть тимчасово збережені для подальших маніпуляцій, таких як вибір предметів, внесення оцінок або генерація диплому.

5.2 Інтеграція сервісу з навчальними планами та оцінками

Наступним етапом, після вибору студента, є внесення вибіркових предметів, які були обрані здобувачем освіти, у базу даних, що є обов’язковим етапом перед виставленням оцінок та подальшою генерацією диплома для здобувача освіти. Для реалізації функціоналу вибору предметів було використано два мікросервіса. Спочатку сервіс студентів для вибору унікального певного здобувача освіти та сервіс навчальних планів для отримання списків предметів, які доступні даному студенту.

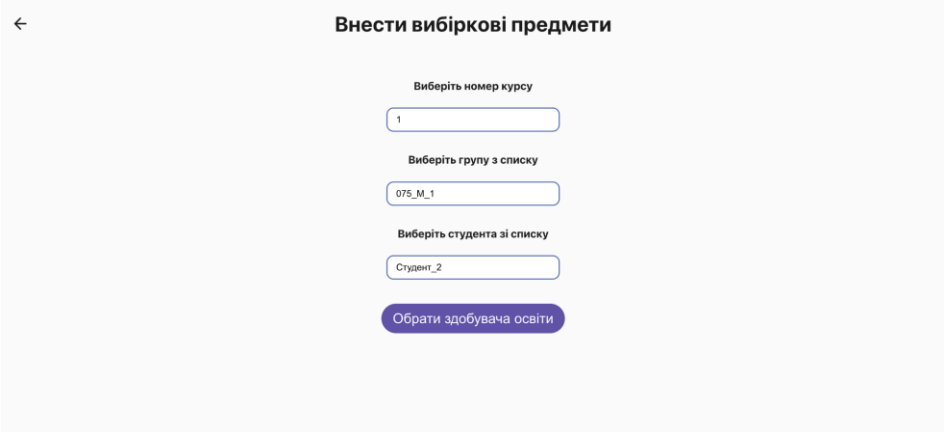

Рисунок. 5.4. Підтвердження вибору студента

Після натиску кнопки «Обрати здобувача освіти» (Див. рис. 5.4), на основі обраного здобувача освіти, відправляється GET-запит (*/optionalSubjects/{studentId}*) до сервісу навчальних планів з параметром *studentId*, завдяки цьому параметру можна індивідуально для кожного вносити обрані предмети. У відповідь приходять два списки:

1. доступні факультетські предмети;

2. загальноуніверситетських предмети.

З них формується загальний список таким чином: кожна група предметів, з якої буде обраний один предмет, має свій набір факультетських предметів та спільний набір загальноуніверситетських предметів.

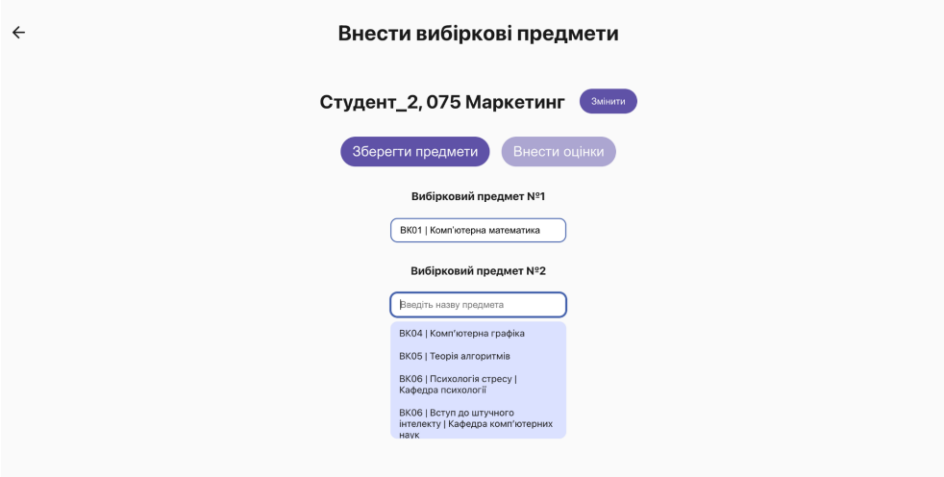

Рисунок. 5.5. Сторінка з вибірковими предметами

На рис. 5.5 зображено інтерфейс для заповнення вибіркових предметів, який складається з таких елементів:

- **Ім'я студента та назва групи** відображає поточного студента для якого будуть вноситися вибіркові предмети. Також є можливість швидко змінити студента.
- **Поля для введення вибіркових предметів** — кожне поле відповідає одному предмету, який студент може обрати в межах свого навчального плану. Поля підтримують автодоповнення на основі раніше сформованого списку з факультетських та загальноуніверситетських предметів. При виборі двох або більше однакових предметів, поле буде світитися червоним та вказувати на помилку.
- **Кнопка «Зберегти предмети»** зберігає поточні зміни у навчальному плані студента, надсилаючи PUT-запит (*/optionalSubjects/{studentId}*) разом з параметром *studentId*, що дозволить серверу індивідуально зберегти зміни.
- **Кнопка «Внести оцінки»** дозволяє перейти до наступного етапу, де будуть внесені оцінки для пройдених предметів за весь період навчання.

Після успішного заповнення вибіркових предметів стає доступним функціонал внесення всіх здобутих студентом оцінок протягом усіх років навчання. Варто зазначити, що перейти до заповнення оцінок також можна з головного екрану електронної системи документообігу. Ця можливість особливо корисна, якщо вибіркові предмети були внесені раніше або іншим відповідальним працівником, що дозволяє розподілити завдання між співробітниками — одна особа вносить оцінки, а інша — вибіркові предмети.

Коли студент був обраний, вебзастосунок надсилає GET-запит (*/grades/student/{studentId}*) до сервісу з навчальними планами. Тут також використовується унікальний ідентифікатор студента, як параметр для запиту. У відповідь ми отримуємо список всіх дисциплін з оцінками, для обраного студента.

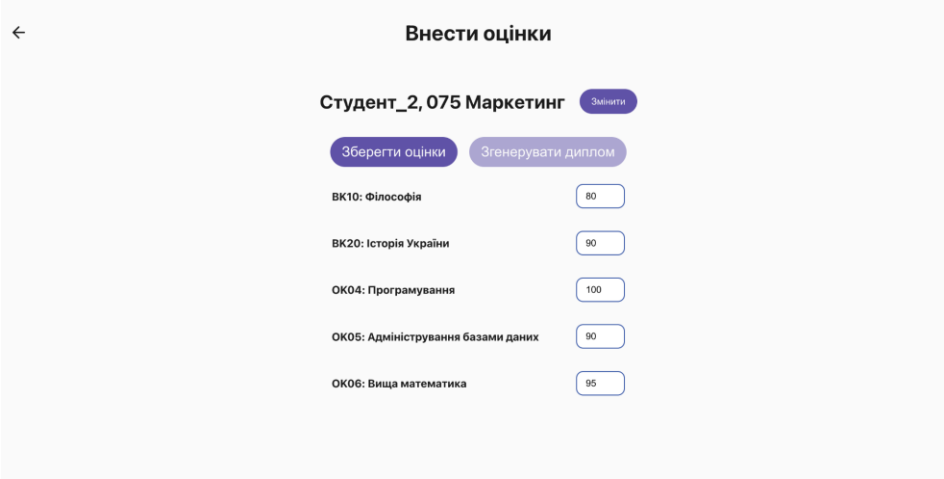

Рисунок. 5.6. Сторінка з внесенням оцінок

На рис. 5.6 зображено інтерфейс для внесення оцінок. Він є подібним до попередньої сторінки (рис 5.5), така уніфікації дозволить працівникам вищого навчального закладу швидше і простіше освоїтися у системі електронного документообігу. Серед відмінностей у інтерфейсі можна виділити:

- **Поля для введення оцінок** — кожен предмет представлений його кодом, повною назвою та полем з відповідним балом. Також, присутня валідація, яка не дозволяє внести оцінку менше 60 та більше 100.
- **Кнопка «Зберегти оцінки»** зберігає внесені зміни у базі даних, надсилаючи PUT-запит (*/optionalSubjects/{studentId}*) разом з параметром *studentId*, що дозволить серверу індивідуально зберегти зміни.
- **Кнопка «Згенерувати диплом»** дозволяє перейти до завершального етапу — генерація диплома для студента.

5.3 Процес генерації диплому та його додатку

Додано примітку [4]: Може змінитися

Завершальним етапом, після вибору предметів та внесення оцінок, є генерація диплому здобувача освіти. Автоматизація цього процесу усуває потребу у ручному введенні даних, що у свою чергу зменшує кількість помилок та скорочує час підготовки документів до друку. Для реалізації було використано окремий сервіс з дипломами, а також сервіс студентів для ідентифікації здобувача освіти.

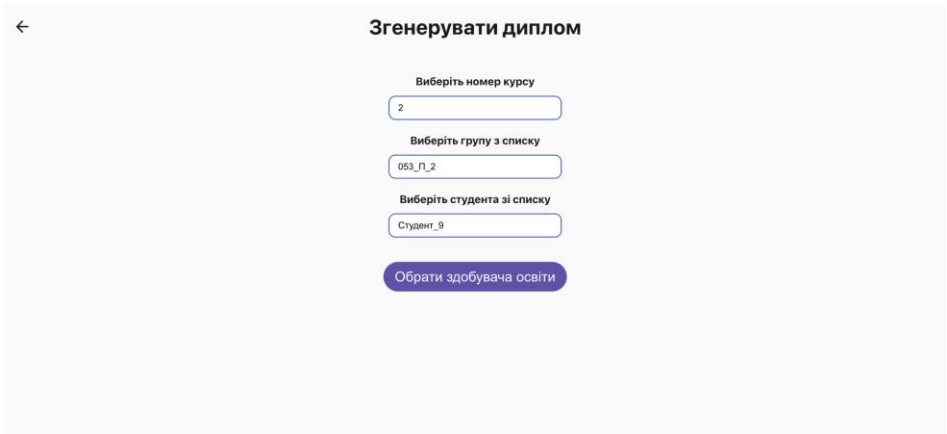

Рисунок. 5.7. Підтвердження вибору студента

Спершу, користувач обирає певного студента та підтверджує свій вибір натиснувши кнопку «Обрати здобувача освіти» (Див. рис. 5.7). Далі з’являється

інтерфейс, що допомагає працівникам закладу вищої освіти оптимізувати процес створення диплому для здобувача освіти (Див. рис. 5.8).

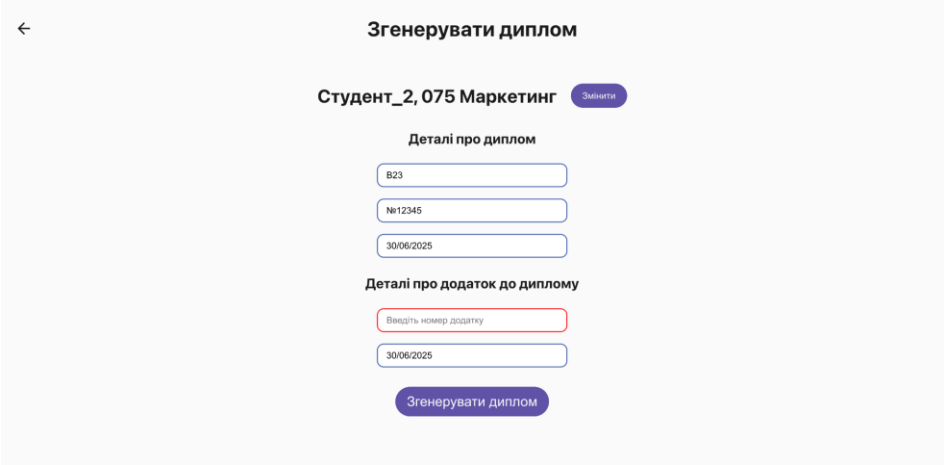

Рисунок. 5.8. Сторінка генерації диплома

У верхній частині відображається прізвище, ім’я та група поточного, вибраного студента для якого буде згенеровано диплом. Якщо студент був обраний помилково, є можливість змінити вибір, натиснувши на відповідну кнопку «Змінити», яка повертає на екран вибору меню вибору здобувача освіти. Наступний блок — «Деталі про диплом та його додаток». Багато інформації для заповнення бланку диплома, дипломний мікросервіс отримує під час мікросервісної комунікації з такими сервісами, як навчальні плани та студенти. Але але є така інформація яка недоступна у жодній наші базі даних, тому, відповідальний за це працівник повинен ввести вручну все чого не вистачає для успішної генерації диплома. Обов’язковою для введення інформації є:

- серія диплому (B23);
- номер диплому (№12345);
- дата видачі диплому (30/06/2025);
- номер додатку до диплома (№67890);
- дата видачі додатку до диплому (30/06/2025).

У вебзастосунку присутня мінімальна, потрібна валідація даних, якщо один із рядків порожній, то система не дасть відправити запит на генерацію диплому та підсвітить проблемне поле червоним кольором. У разі успішної верифікації цілісності даних та після натиску кнопки «Згенерувати диплом» здійснюється POST-запит (*/generateDiploma*) до дипломного мікросервіса. У тілі запита передається *studentId* та вищезазначена інформація про диплом та його додаток. Після успішної генерації та завантаження диплома на стороні сервера, вебзастосунок отримає відповідь з посиланням, перейшовши по якому працівник зможе запросто завантажити готові документи.

ВИСНОВКИ

Розробка інформаційної системи підтримки електронного документообігу закладу вищої освіти є важливим етапом у цифровізації процесів, пов’язаних з обробкою документів.

Під час виконання роботи було досягнуто поставлену мету — автоматизовано обробку навчальних планів і студентських даних та формування дипломів на основі попередньо оброблених даних. Було проведено детальний аналіз потреб навчального закладу та визначено функціональні вимоги до системи електронного документообігу. Досліджено переваги та недоліки існуючих комерційних рішень, що дозволило виявити численні переваги створення власної системи, яка більшою мірою відповідає специфічним потребам закладу.

Практична частина роботи реалізована з використанням сучасного технологічного стеку, що забезпечує надійність, масштабованість і продуктивність системи. У фронтенд-частині використано React з TypeScript у поєднанні з допоміжними бібліотеками Axios, React Router та Redux. Обрані технології у сукупності забезпечують гнучкий та швидкий інтерфейс для кінцевих користувачів, що підвищує зручність роботи із системою та мінімізує затримки при обробці даних. Для бекенд сервісу було використано Ktor з Kotlin, що дозволило створити високопродуктивний мікросервіс з мінімальними затримками. Для зберігання даних використовується реляційна база даних PostgreSQL та нереляційна — Redis для утилізації токенів. Надійний захист від несанкціонованого доступу забезпечують Google OAuth 2.0 та JWT токени, які дозволяють верифікувати надіслані запити.

У результаті виконання кваліфікаційної роботи було поглиблено практичні навички аналізу специфічних вимог для системи електронного документообігу у закладі вищої освіти та проведено її порівняння з комерційними рішеннями. Важливим досвідом стало проєктування масштабованої архітектури та

інтерфейсу для вебзастосунку та інтеграція мікросервісів авторизації, роботи зі студентськими даними, навчальними планами та дипломами.

Загалом, результати дослідження підтверджують доцільність розробки власної інформаційної системи підтримки електронного документообігу в закладах вищої освіти, адже вона краще адаптована до специфічних вимог. Розроблена система дозволяє автоматизувати етапи обробки навчальних планів та даних про здобувачів освіти та спрощує етап генерації дипломів.

У перспективі, система може бути доповнена мобільним застосунком, автоматичним формуванням розкладу або інших документів, та може бути масштабована на інші факультети чи кафедри, що підвищить ефективність роботи навчального закладу та сприятиме підвищенню якості надання освітніх послуг.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. "Мегаполіс" в університеті | КПІ ім. Ігоря Сікорського. КПІ ім. Ігоря Сікорського. URL: https://kpi.ua/megapolis (дата звернення: 27.10.2024).
2. Alfresco (ECM-система) – Вікіпедія. Вікіпедія. URL: https://uk.wikipedia.org/wiki/Alfresco_(ECM-система) (дата звернення: 27.10.2024).
3. Stack overflow developer survey 2020. Stack Overflow. URL: https://survey.stackoverflow.co/2020 (дата звернення: 01.11.2024).
4. Рейтинг мов програмування 2024. DOU. URL: https://dou.ua/lenta/articles/language-rating-2024/ (дата звернення: 01.11.2024).
5. Fain Y., Moiseev A. TypeScript quickly. Manning Publications Co. LLC, 2020.
6. Сучасний підручник з JavaScript. JavascriptInfo. URL: https://uk.javascript.info/ (дата звернення: 14.11.2024).
7. Handbook - The TypeScript Handbook. TypeScript: JavaScript With Syntax For Types. URL: https://www.typescriptlang.org/docs/handbook/intro.html (дата звернення: 14.11.2024).
8. Effective typescript: 62 specific ways to improve your typescript. O'Reilly Media, Incorporated, 2019. 250 p.
9. Understanding TypeScript’s benefits and pitfalls - LogRocket Blog. URL: https://blog.logrocket.com/understanding-typescripts-benefits-pitfalls/ (дата звернення: 20.11.2024).
10. TypeScript Pros, Cons, Faqs, and Myths. byby.dev - random bits from a developer. URL: https://byby.dev/ts-pros-cons (дата звернення: 20.11.2024).
11. Офіційне документація та базові можливості JavaScript бібліотеки React. Швидкий старт – React. URL: https://uk.react.dev/learn (дата звернення: 19.11.2024).
12. Griffiths D., David G. React Cookbook: Recipes for Mastering the React Framework. O'Reilly Media, Incorporated, 2021. 500 p.

13. Garreau M., Faurot W. Redux in Action. Manning Publications, 2018. 312 p.
14. React Router Home | React Router. React Router Official Documentation. URL: https://reactrouter.com/home (дата звернення: 24.11.2024).
15. Getting Started | Axios Docs. Axios. URL: https://axios-http.com/docs/intro (дата звернення: 25.11.2024).
16. Larman C. Applying UML and patterns: an introduction to object-oriented analysis and design. Upper Saddle River, N.J : Prentice Hall PTR, 1998. 507 p.
17. Madden N. API Security in Action. Manning Publications Company, 2020. 570 p.
18. How to create RESTful APIs in Kotlin with Ktor. Ktor Help. URL: https://ktor.io/docs/server-create-restful-apis.html (дата звернення: 27.02.2025).
19. Silva M. D. D., Tavares H. L. Redis Essentials: Harness the power of Redis to integrate and manage your projects efficiently. Packt Publishing, 2015. 230 p.
20. Conway J., Berkus J. Power Postgresql: Maximizing Postgresql Performance, Reliability, and Utility. Pearson Education, Limited, 2025. 600 p.

ДОДАТКИ

ДОДАТОК А

Код домашньої сторінки

```
import { useState, useEffect } from 'react';
import { useNavigate } from 'react-router-dom';
import RshuAppBar from '../components/RshuAppBar';
import RshuCardItem, { RshuCardItemProps } from '../components/RshuCardItem';
import './style/Home.css';
import GetHomeItemsUseCase from '../../domain/GetHomeItemsUseCase';
import { LoginRoute } from '../navigation/Routes';
import { Staff } from '../../domain/roles/GetAllStaffUseCase';
import GetCurrentStaffUseCase from '../../domain/auth/GetCurrentStaffUseCase';
import PageState, { usePageState } from '../components/state/PageState';
import LogOutStaffUseCase from '../../domain/roles/LogOutStaffUseCase';

const HomeScreen = () => {
   const [pageState, showLoader, pageIsReady, showError] = usePageState({ loading:
false, error: null });

   const navigate = useNavigate();
   const [staff, setStaff] = useState<Staff | null>(null);
   const [homeItems, setHomeItems] = useState<RshuCardItemProps[]>([]);

   useEffect(() => {
      showLoader();
      GetCurrentStaffUseCase({
         onSuccess: (staff) => {
            pageIsReady();
            setStaff(staff);
            setHomeItems(GetHomeItemsUseCase(staff.role));
```

```
        },
        onFailure: (error) => showError(error, () => window.location.reload()),
    })
}, []);

const logOut = () => {
    showLoader();
    LogOutStaffUseCase({
        onSuccess: () => {
            pageIsReady();
            navigate(LoginRoute, { replace: true })
        },
        onFailure: (error) => showError(error, () => window.location.reload()),
    })
}

const getTitle = () => {
    return staff?.fullName ? `Вітаємо, ${staff?.fullName}` : "eRSHU - Teachers";
}

return (
    <div>
        <PageState {...pageState} />

        <RshuAppBar
            title={getTitle()}
            logOut={logOut}
            hideBackButton={true} />

        <div className="items-container">
```

```
            {
              staff && homeItems?.map((item, index) => (
                <RshuCardItem key={index} title={item.title} route={item.route} />
              ))
            }
          </div>
        </div>
    );
};

export default HomeScreen;
```

Код сторінки керування ролей

```
import RshuButton from "../components/button/RshuButton";
import RoleItem from "./RoleItem";
import { useState, useEffect } from "react";
import "./styles/Role.css";
import GetAllStaffUseCase, { Staff } from "../../domain/roles/GetAllStaffUseCase";
import GetAllRolesUseCase from "../../domain/roles/GetAllRolesUseCase";
import { setUsers, setNewUser, setEditableUsers, deleteUser, editUser } from
"./RolesSlice";
import { useDispatch, useSelector } from "react-redux";
import EditStaffUseCase, { StaffAction } from "../../domain/roles/EditStaffUseCase";
import PageState, { usePageState } from "../components/state/PageState";
import RshuAppBar from "../components/RshuAppBar";
import { rolesTitle } from "../../domain/GetHomeItemsUseCase";

const RolesScreen = () => {
    const [pageState, showLoader, pageIsReady, showError] = usePageState();
```

```
    const dispatch = useDispatch();
    const userList: Array<Staff> = useSelector((state: any) => state.roles.userList);
    const editableUsers: Array<Staff> = useSelector((state: any) =>
state.roles.editableUsers);
    const newUser = useSelector((state: any) => state.roles.newUser);
    const [allRoles, setAllRoles] = useState<string[]>([]);

    useEffect(() => {
        GetAllRolesUseCase({
            onSuccess: (roles) => setAllRoles(roles.roles),
            onFailure: (error) => showError(error, () => {window.location.reload()}),
        });
        GetAllStaffUseCase({
            onSuccess: (staff) => {
                pageIsReady();
                dispatch(setUsers(staff.staff));
            },
            onFailure: (error) => showError(error, () => {window.location.reload()}),
        });
    }, [dispatch]);

    const addUser = () => {
        if (!newUser) {
            const user = {
                fullName: "",
                email: "",
                role: "ADMIN",
            };
            dispatch(setEditableUsers([...editableUsers, user]));
            dispatch(setNewUser(user));
```

```
    }
};

const onDeleteClick = (staff: Staff) => {
    showLoader();
    EditStaffUseCase(StaffAction.DELETE, staff, {
        onSuccess: () => {
            dispatch(deleteUser(staff.email));
            pageIsReady();
        },
        onFailure: (error) => showError(error, () => {window.location.reload()}),
    })
};

const cancelAddUser = () => {
    dispatch(setNewUser(null));
}

const addNewUser = (newUser: Staff) => {
    showLoader();
    EditStaffUseCase(StaffAction.ADD, newUser, {
        onSuccess: () => {
            dispatch(setUsers([newUser, ...userList]));
            dispatch(setNewUser(null));
            pageIsReady();
        },
        onFailure: (error) => showError(error, () => {window.location.reload()}),
    });
}
```

```
    const onEditDone = (oldUser: Staff, newUser: Staff) => {
        showLoader();
        EditStaffUseCase(StaffAction.EDIT, newUser, {
            onSuccess: () => {
                dispatch(editUser(
                    {
                        oldUser: oldUser,
                        newUser: newUser
                    }
                ));
                pageIsReady();
            },
            onFailure: (error) => showError(error, () => {window.location.reload()}),
        });
    }

    const setIsEditable = (user: Staff, isEditable: boolean) => {
        if (isEditable) {
            dispatch(setEditableUsers([...editableUsers, user]));
        } else {
            dispatch(setEditableUsers(editableUsers.filter((editableUser)                    =>
editableUser.email !== user.email)));
        }
    }

    return (
        <div>
            <RshuAppBar title={rolesTitle} />

            <div className="role-container">
```

```
            <PageState {...pageState} />

            <RshuButton className="role-add" label="Додати нового користувача"
onClick={addUser} />
            {
               newUser && <RoleItem key={0}
                  user={newUser}
                  allRoles={allRoles}
                  isEditable={editableUsers.includes(newUser)}
                  setIsEditable={(it) => setIsEditable(newUser, it)}
                  onDeleteUser={() => {}}
                  onEditDone={addNewUser}
                  onEditCancel={cancelAddUser} />
            }
            {
               userList.map((user, index) => (
                  <RoleItem key={index + 1}
                     user={user}
                     allRoles={allRoles}
                     isEditable={editableUsers.includes(user)}
                     setIsEditable={(it) => setIsEditable(user, it)}
                     onDeleteUser={() => onDeleteClick(user)}
                     onEditDone={(newUser) => onEditDone(user, newUser)}
                     onEditCancel={() => {}} />
               ))
            }
         </div>
      </div>
   );
};
```

```
export default RolesScreen;
```

Код сторінки оновлення навчальних планів

```
import { plansFileTitle } from '../../domain/GetHomeItemsUseCase';
import UploadSubjectsFileUseCase from
'../../domain/subject/UploadSubjectsFileUseCase';
import { useButtonState } from '../components/button/ButtonState';
import RshuButton from '../components/button/RshuButton';
import RshuAppBar from '../components/RshuAppBar';
import RshuFileUploader from '../components/RshuFileUploader';
import './styles/Plans.css';
import { useState } from 'react';

const PlansFileScreen = () => {
    const [file, setFile] = useState<File | null>(null);
    const buttonState = useButtonState();

    const handleFileChange = (e: React.ChangeEvent<HTMLInputElement>) => {
        if (e.target.files) {
            setFile(e.target.files[0]);
            buttonState.setState('initial');
        }
    };

    const uploadFile = () => {
        if (!file) return;

        buttonState.setState('progress');
```

```
    setTimeout(() => {
        UploadSubjectsFileUseCase(file, {
            onSuccess: () => {
                buttonState.setState('success');
            },
            onFailure: () => {
                buttonState.setState('fail');
            }
        });
    }, 2000);
}

const getUpdateLabel = () => {
    switch (buttonState.buttonState) {
        case 'progress':
            return 'Завантаження...';
        case 'success':
            return 'Успішно! Ще раз?';
        case 'fail':
            return 'Сталася помилка. Ще раз?';
        default:
            return 'Оновити дані';
    }
}

return (
    <div>
        <RshuAppBar title={plansFileTitle} />

        <div className='plans-container'>
```

```
            <RshuFileUploader file={file} onFileChange={handleFileChange} />

            <RshuButton
              className={'update-button ' + (!file || buttonState.isDisabled ? 'disabled' :
'')}
              label={getUpdateLabel()}
              onClick={uploadFile}
            />
          </div>
        </div>
    );
};

export default PlansFileScreen;
```