МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

ТАРАНОВСЬКИЙ ПАВЛО ЮРІЙОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій
канд. техн. наук, доцент
_____ О. В. Зелінська
«____»_____ 2024 р.

РОЗРОБКА ДОДАТКУ ДЛЯ ОБЛІКУ ДОКУМЕНТАЦІЇ КАФЕДРИ ЗВО

Спеціальність 122 «Комп’ютерні науки»

Кваліфікаційна (бакалаврська) робота

Керівник:
Р. М. Бабаков, професор кафедри
інформаційних технологій,
д. т. н., доцент
_____

Оцінка: _____ / _____ / _____
(бали/за шкалою ЄКТС/за національною шкалою)

Голова ЕК: _____

Вінниця – 2024

АНОТАЦІЯ

Тарановський П.Ю. Розробка додатку для обліку документації кафедри ЗВО. Спеціальність 122 «Комп’ютерні науки». Донецький національний університет імені Василя Стуса, Вінниця, 2024.

Мета кваліфікаційної роботи полягає у розробці ефективного та зручного у використанні додатку для обліку документації кафедри ЗВО з метою полегшення процесів зберігання, пошуку та управління документами, забезпечення систематизації та структурування інформації, а також підвищення загальної ефективності адміністративної роботи на кафедрі.

У вступі висвітлено актуальність саме розробки додатку. У першому розділі аналізуються головні конкуренти та проводиться аналіз цільової області. У другому розділі розглядаються інструменти та методи, які сприяли створенню цього додатку. У третьому розділі подано опис інтерфейсу користувача, а також реалізацію клієнтської та серверної частин програми.

Ключові слова: Додаток для обліку документації, Python, PostgresSql

54 с., 13 рис., 10 джерел.

ABSTRACT

Development of an Application for Accounting of Documentation of the Department of a Higher Educational Institution. Specialty 122 "Computer Science". Vasyl Stus Donetsk National University, Vinnytsia, 2024.

The aim of the qualification work is to develop an efficient and user-friendly application for managing documentation within the department of a higher education institution. This aims to facilitate processes such as storage, retrieval, and management of documents, ensuring systematic structuring of information and enhancing overall administrative efficiency within the department.

The introduction highlights the relevance of developing such an application. The first chapter analyzes the main competitors and conducts an analysis of the target area. The second chapter examines the tools and methods that contributed to the creation of

this application. The third chapter provides a description of the user interface, as well as the implementation of both the client and server sides of the program.

Keywords: Application for Accounting of Documentation, Python, PostgresSql
54 p., 13 figures, 10 sources.

ЗМІСТ

ВСТУП

У наш час, коли обсяг і склад документації зростає з кожним днем, потреба в ефективному управлінні цими даними стає надзвичайно важливою. Необхідність у швидкому доступі до різноманітних документів, які можуть бути розташовані в різних місцях та форматах, вимагає розробки інструментів, які спростять та зроблять більш ефективними процеси роботи з ними. Додаток для обліку документації кафедри у закладі вищої освіти може стати не лише зручним інструментом для співробітників кафедри, але й важливим кроком у покращенні якості освітніх послуг загалом.

Особлива увага у цій роботі буде приділена аналізу вже існуючих рішень та програмних продуктів, які застосовуються для управління документацією в освітніх установах. Шляхом порівняння їхніх можливостей та функціоналу з вимогами та потребами конкретної кафедри, ми зможемо визначити переваги та недоліки існуючих рішень та відповідно розробити оптимальний варіант для запропонованого додатку.

Завдяки розробці цього додатку, ми очікуємо досягнення значного покращення у сфері адміністративного управління кафедрою, зменшення часових затрат на пошук та обробку документів, а також підвищення загального рівня організації та ефективності роботи. Таким чином, ця робота не лише відповідає актуальним потребам сучасної освіти, а й відкриває нові перспективи для вдосконалення управління освітніми процесами в цілому.

Успішна реалізація цього додатку також сприятиме покращенню комунікації та співпраці між різними підрозділами кафедри, забезпечуючи зручний доступ до спільної бази даних. Це дозволить уникнути дублювання інформації та забезпечить єдність даних між усіма членами колективу.

Крім того, розробка цього додатку може стати першим кроком у впровадженні цифровізації та автоматизації адміністративних процесів в освітніх установах. Це сприятиме створенню сучасного та інноваційного середовища, яке

відповідатиме вимогам сучасної освіти та підвищить конкурентоспроможність установи.

Загалом, розробка додатку для обліку документації кафедри є важливим кроком у напрямку модернізації та покращення управління освітніми процесами. Це не лише сприятиме оптимізації рутинних завдань, але й створить основу для подальшого розвитку та впровадження інноваційних технологій у сфері освіти.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ ФУНКЦІОНУВАННЯ ДОДАТКУ ДЛЯ ОБІЛКУ ДОКУМЕНТАЦІЇ

1.1 Аналіз вимог до системи обліку документації кафедри ЗВО

Ідентифікація основних видів документації полягає у проведенні докладного аналізу різноманітності документів, які генеруються та обробляються на кафедрі ЗВО. Серед цих документів можуть бути навчальні програми, які визначають зміст та методику навчання; наукові дослідження, включаючи наукові статті, звіти, проекти тощо; різноманітні звіти, такі як звіти про науково-дослідну роботу, фінансові звіти, звіти про виконану роботу; матеріали лабораторних робіт, які містять результати експериментів та протоколи лабораторних занять; а також адміністративні документи, такі як розпорядження, накази, листи, довідки, протоколи засідань тощо. Цей аналіз є важливим етапом у розумінні обсягу та характеру документації кафедри, що дозволяє належним чином визначити потреби у системі обліку документів та розробити відповідні рішення для оптимального управління цими документами.

Після ідентифікації основних видів документації на кафедрі ЗВО, проводиться подальший аналіз, спрямований на визначення характеристик кожного типу документа та їхнього обсягу. Це включає в себе встановлення стандартів та форматів документів, їхніх специфічних вимог до зберігання, термінів зберігання та доступності для різних користувачів. Наприклад, навчальні програми можуть мати специфічні вимоги щодо формату та змісту, вимагаючи регулярного оновлення та перегляду, тоді як наукові дослідження можуть вимагати більшої конфіденційності та контролю доступу.

Детальний аналіз основних видів документації також допомагає визначити потреби користувачів у системі обліку документів. Різні типи документів можуть мати різні вимоги щодо швидкості доступу, рівня конфіденційності та можливості спільної роботи над ними. Наприклад, студентам може бути потрібний швидкий доступ до навчальних матеріалів, в той час як наукові праці

можуть вимагати більшого контролю доступу та можливості спільного редагування. Такий аналіз допомагає підготувати план розробки системи обліку документів, що належним чином враховує потреби та вимоги користувачів кафедри ЗВО. [1]

Оцінка потреб користувачів в системі обліку документації є важливим етапом у процесі розробки, оскільки вона дозволяє зрозуміти очікування та пріоритети користувачів щодо функціональності та використання системи. Для проведення цієї оцінки можуть використовуватися різноманітні методи, одним з яких є проведення опитування серед членів кафедри.

Опитування може включати запитання щодо поточних проблем з управлінням документацією, таких як час, витрачений на пошук необхідних документів, або складність спільної роботи над документами. Крім того, користувачі можуть висловлювати свої побажання щодо функціональності системи, такої як можливість створення категорій або тегів для організації документів, автоматичне створення резервних копій або можливість доступу до системи з різних пристроїв.

Після збору даних з опитування, їх можна проаналізувати з метою визначення головних потреб користувачів та розроблення відповідного функціоналу системи обліку документації. Це дозволить забезпечити, що розроблена система відповідає потребам користувачів та допомагає покращити їхню продуктивність та ефективність у роботі.

Окрім опитування, важливо також провести фокус-групи або інтерв'ю з ключовими представниками кафедри. Це дозволить отримати більш глибоке розуміння потреб користувачів, їхніх вподобань та проблем, з якими вони зіштовхуються при використанні поточних систем обліку документації.

Після збору відгуків від користувачів, слід провести аналіз, щоб ідентифікувати спільні та найбільш важливі потреби користувачів. Цей аналіз допоможе зрозуміти, які саме функції та можливості має надавати система обліку документації, щоб задовольнити потреби користувачів та покращити їх робочий процес.

Аналіз поточних проблем існуючих систем обліку документації на кафедрі ЗВО включає в себе виявлення та оцінку різних аспектів, які можуть бути недоліками чи недоліками у поточних процесах управління документами. Ось деякі з найпоширеніших проблем:

- Втрати даних: Поточна система може бути вразливою до втрати даних через технічні несправності, випадкове видалення або невірне зберігання, що може призвести до втрати важливої інформації.
- Незручність пошуку та доступу до інформації: Користувачам може бути важко знаходити необхідні документи через неякісну організацію документації або відсутність ефективних засобів пошуку.
- Недостатня систематизація: Документи можуть бути розташовані у незручних або непорозумілих структурах, що ускладнює їхнє знаходження та організацію.
- Недостатній рівень безпеки: Система може бути недостатньо захищеною від несанкціонованого доступу, що може призвести до витоку конфіденційної інформації.
- Обмежені можливості спільної роботи: Відсутність зручних інструментів для спільного редагування та обміну документами може гальмувати співпрацю між членами кафедри. [2]

Аналіз цих проблем допоможе визначити основні недоліки поточних систем обліку документації та визначити пріоритети для подальшого вдосконалення та розробки нової системи.

Визначення технічних вимог для системи обліку документації - це процес встановлення конкретних параметрів та характеристик, які система повинна відповідати для забезпечення ефективної та надійної роботи. Основні аспекти технічних вимог включають масштабованість, безпеку даних та можливість інтеграції з іншими програмними засобами.

Масштабованість визначає здатність системи працювати ефективно при зростанні обсягу даних чи користувачів. Система повинна бути здатна

масштабуватися горизонтально або вертикально, щоб забезпечити стабільну роботу навіть у випадку значного збільшення навантаження.

Безпека даних включає різні заходи для захисту конфіденційності, цілісності та доступності даних. Система повинна мати механізми шифрування, контролю доступу, резервне копіювання та інші заходи для запобігання несанкціонованому доступу або втраті даних.

Можливість інтеграції з іншими програмними засобами є також важливою характеристикою. Система повинна бути сумісною з існуючими програмами та платформами, що використовуються на кафедрі, для забезпечення зручності та ефективності використання.

Додатковим аспектом технічних вимог є ефективність системи обліку документації. Система повинна працювати швидко та ефективно, навіть при великому обсязі даних та одночасному доступі багатьох користувачів. Це включає в себе оптимізацію швидкості обробки запитів, реакції на користувацькі взаємодії та мінімізацію часу очікування.

Крім того, важливою характеристикою є гнучкість системи, що дозволяє вносити зміни та розширювати функціональність відповідно до змін у потребах користувачів чи технологій. Система повинна бути легко налаштовуваною та розширюваною, щоб враховувати нові вимоги та можливості, які можуть з'явитися у майбутньому.

Огляд правових та регуляторних вимог для системи обліку документації є важливим етапом в розробці, оскільки він визначає обов'язки та вимоги, які система повинна виконувати з точки зору законодавства та внутрішніх правил установи.

Цей процес включає проведення аналізу вимог, що стосуються зберігання та обробки документів відповідно до встановлених норм законодавства. Наприклад, деякі види документів можуть підпадати під особливі правові вимоги з питань зберігання та доступу до них, наприклад, медичні записи або особисті дані студентів.

Крім того, система обліку документації повинна відповідати внутрішнім правилам та процедурам установи, зокрема щодо конфіденційності, захисту особистих даних та документообігу. Це може включати в себе вимоги щодо обмеження доступу до певних категорій документів, ведення журналів доступу, а також встановлення процедур архівування та знищення документів у відповідності з правилами зберігання.

Під час огляду правових та регуляторних вимог також важливо враховувати міжнародні та національні стандарти зберігання та обробки даних, такі як Загальний регламент про захист персональних даних (GDPR) у Європейському Союзі або стандарти забезпечення інформаційної безпеки ISO/IEC 27001. Дотримання цих стандартів може бути важливим для забезпечення відповідності та міжнародного визнання системи.

Зважаючи на постійні зміни в законодавстві та регуляторному середовищі, система обліку документації повинна бути готовою до внесення змін та адаптації відповідно до нових вимог. Це може вимагати регулярного оновлення та аудитування системи з метою впевненості в її відповідності вимогам. Такий підхід допоможе забезпечити надійність, безпеку та відповідність системи обліку документації у всіх аспектах її функціонування.

1.2 Огляд теоретичних концепцій та методів розробки програмного забезпечення для обліку документації

Теорія баз даних є ключовою для розуміння та використання ефективних методів зберігання та обробки даних у системах обліку документації. Вивчення принципів реляційних та нереляційних моделей допомагає розробникам визначити найкращий підхід до структурування даних, що враховує специфіку кожного конкретного випадку.

Реляційна модель бази даних забезпечує логічне та організоване зберігання даних у вигляді таблиць, що дозволяє ефективно керувати інформацією та виконувати різноманітні операції, такі як пошук, сортування та фільтрація. Нереляційні моделі, з іншого боку, надають більшу гнучкість у роботі з

неструктурованими даними та дозволяють зберігати дані у вигляді ключ-значення, документів чи графів.

Вибір оптимальної моделі бази даних залежить від конкретних потреб системи обліку документації, таких як обсяг та характер даних, швидкість доступу до них, складність запитів та інші фактори. Правильно спроектована база даних забезпечує ефективне управління документами та їх метаданими, що робить роботу з системою зручною та продуктивною для користувачів.

Додатково, вивчення теорії баз даних допомагає врахувати різноманітні аспекти, такі як нормалізація даних, індексація та оптимізація запитів. Нормалізація даних спрощує процес зберігання та управління інформацією, зменшуючи дублювання даних та забезпечуючи їхню цілісність. Індексація дозволяє швидко здійснювати пошук та доступ до даних шляхом створення ефективних індексів для швидкого доступу до потрібної інформації. Оптимізація запитів дозволяє підвищити продуктивність системи шляхом ефективного виконання складних запитів до бази даних. [3]

Крім того, розуміння теорії баз даних є важливим для підтримки та розвитку системи у майбутньому. Правильно спроектована база даних може легко масштабуватися та адаптуватися до змін у вимогах користувачів або умов діяльності кафедри, забезпечуючи довготривалу та стабільну роботу системи обліку документації.

Використання методів аналізу вимог є ключовим етапом у процесі розробки програмного забезпечення для обліку документації. Використання методу викладки вимог дозволяє систематизувати та документувати потреби користувачів, уточнюючи їхні вимоги до функціональності та характеристик системи. Цей метод допомагає уникнути недорозумінь та невідповідностей у сприйнятті вимог між розробниками та замовниками.

Принципи SOLID (Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) є основними принципами об'єктно-орієнтованого програмування, які допомагають створювати системи, що є міцними, гнучкими та легко розширюваними. Використання цих принципів

дозволяє розробникам правильно організувати структуру програмного забезпечення для забезпечення його ефективності та підтримки у майбутньому.

Архітектурні шаблони, такі як MVC (Model-View-Controller), MVP (Model-View-Presenter), MVVM (Model-View-ViewModel) та інші, встановлюють основні принципи побудови програмного забезпечення та розділення його компонентів для забезпечення простоти, гнучкості та повторного використання коду. Використання цих шаблонів допомагає створювати системи, які є легкими у розробці, тестуванні та підтримці.

Крім цього, методи аналізу вимог також можуть включати проведення сеансів взаємодії з користувачами, спостереження за роботою та спілкування з потенційними користувачами для збору вимог та отримання повноцінного розуміння їхніх потреб. Це дозволяє отримати важливі відомості про реальне використання системи та відповідно налаштувати функціональність.

Також важливим етапом є аналіз існуючих аналогічних систем та їхніх недоліків. Шляхом вивчення конкурентів та їхніх рішень можна виявити можливості для покращення, а також уникнути повторення помилок чи недоліків власного продукту.

Принципи інтерфейсного дизайну включають в себе не лише естетичний аспект, але й функціональність та зручність користування системою. Врахування цих принципів допомагає створити інтуїтивно зрозумілий та приємний інтерфейс, що сприяє зручному та ефективному використанню системи.

Успішний інтерфейс повинен бути спрощеним та легким для сприйняття користувачем, забезпечуючи швидкий доступ до основних функцій та інформації. Наприклад, чітко розміщені кнопки та меню допомагають користувачам швидко зорієнтуватися в системі та здійснити необхідні дії.

Дотримання принципів UI/UX дизайну також включає уникнення перенасиченості інформацією та забезпечення належної інтерактивності. Відповідно побудований інтерфейс дозволяє користувачам з легкістю здійснювати різноманітні дії та отримувати необхідну інформацію без зайвих зусиль.

Для досягнення високої якості інтерфейсного дизайну також важливо враховувати вимоги різних категорій користувачів. Наприклад, система обліку документації на кафедрі використовується різними людьми з різним рівнем технічної підготовки та досвіду використання комп'ютерних програм. Тому інтерфейс повинен бути адаптований для різних користувачів, забезпечуючи зручність використання навіть для тих, хто не має великого досвіду в роботі з програмним забезпеченням. [4]

Додатково, важливо враховувати сучасні тенденції та стандарти в інтерфейсному дизайні. Використання таких концепцій, як "мінімалізм", "плоский дизайн" та "матеріальний дизайн", дозволяє створити сучасний та естетично привабливий інтерфейс, що відповідає сучасним вимогам користувачів.

Нарешті, важливо забезпечити постійне вдосконалення та оновлення інтерфейсу відповідно до змін потреб користувачів та розвитку технологій. Це включає в себе збір та аналіз фідбеку від користувачів, проведення тестувань з виявленням слабких місць інтерфейсу, а також впровадження нововведень та поліпшень для підвищення якості користувацького досвіду.

Методи тестування програмного забезпечення є ключовим етапом в процесі розробки системи обліку документації, оскільки вони дозволяють перевірити її функціональність, надійність та відповідність вимогам перед впровадженням.

Модульне тестування дозволяє перевірити окремі компоненти або модулі програмного забезпечення на коректність їх роботи. Цей тип тестування дозволяє виявити та усунути помилки на ранніх етапах розробки, забезпечуючи більшу стабільність та надійність системи.

Інтеграційне тестування спрямоване на перевірку взаємодії між різними модулями або компонентами програмного забезпечення. Цей процес дозволяє виявити та вирішити проблеми, пов'язані з інтеграцією різних частин системи, та переконатися в правильному функціонуванні системи як єдиної цілі.

Системне тестування проводиться на завершальному етапі розробки перед впровадженням системи в експлуатацію. Цей тип тестування охоплює весь функціонал системи та перевіряє його відповідність вимогам та очікуванням користувачів. Такий підхід дозволяє переконатися в готовності системи до використання в реальних умовах та уникнути непередбачених проблем після впровадження.

Методології розробки програмного забезпечення визначають підхід до організації та управління процесом розробки. Вибір відповідної методології впливає на ефективність команди, швидкість впровадження та якість продукту.

Методологія Waterfall передбачає послідовний процес розробки, де кожна фаза виконується послідовно і завершується перш ніж розпочнеться наступна. Цей підхід підходить для проектів з чітко визначеними вимогами, де можливі зміни мінімальні.

Scrum та Kanban - це ітеративні методології, які дозволяють гнучко виходити за межі стандартних фаз розробки. Scrum використовує суворі ролі, зустрічі та спринти для досягнення мети проекту, тоді як Kanban спрямований на неперервний потік роботи з підтримкою обмеження в роботі над завданнями.

Вибір методології повинен ґрунтуватися на особливостях проекту, таких як розмір команди, складність завдань, частота змін вимог та доступність ресурсів. Ефективне використання відповідної методології допомагає забезпечити вчасне виконання завдань, задоволення потреб користувачів та успішне завершення проекту. [5]

Крім того, важливо враховувати культурні та командні особливості при виборі методології розробки. Наприклад, Scrum може бути ефективним для команд з високим рівнем самоорганізації та гнучкості, тоді як Waterfall може підходити для проектів, де потрібна чітка структура та велика увага до деталей на кожному етапі розробки. Враховуючи ці аспекти, команди можуть здійснити оптимальний вибір методології, який відповідає конкретним умовам проекту та сприятиме досягненню його цілей.

1.3 Огляд існуючих систем обліку документації в навчальних закладах

В сучасному освітньому середовищі ефективне управління документацією в навчальних закладах відіграє ключову роль у забезпеченні якісного навчального процесу та зручного доступу до інформації. Однак, з ростом обсягу даних та збільшенням складності адміністративних процесів, виникає потреба у високоефективних системах обліку документації. В рамках даного розділу ми проведемо огляд існуючих систем обліку документації, що використовуються в навчальних закладах. [10]

Цей огляд допоможе нам з'ясувати переваги та недоліки різних програмних рішень, їхню ефективність у забезпеченні потреб користувачів, а також визначити найбільш підходящі варіанти для певних умов та вимог. Розгляд існуючих систем обліку документації дозволить нам глибше зрозуміти сучасні тенденції у цій галузі та визначити шляхи подальшого розвитку програмного забезпечення для оптимізації адміністративних процесів в навчальних закладах.

Moodle - це відкрите програмне забезпечення для управління навчальним процесом, яке включає можливості для створення курсів, завдань, форумів та інших інструментів.

Переваги: Гнучка система, можливість кастомізації, безкоштовна та активно підтримується спільнотою користувачів.

Недоліки: Складний у використанні для новачків, вимагає додаткових зусиль для налагодження та підтримки.

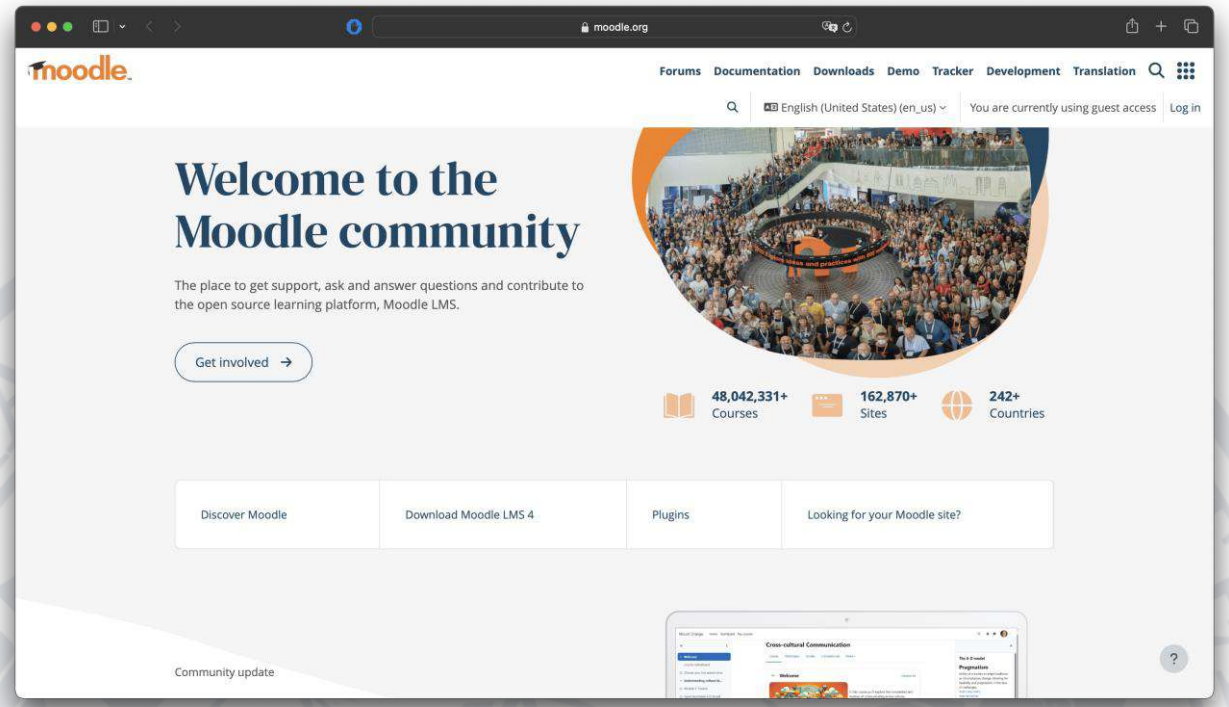

Рисунок 1.1 – головна сторінка платформи moodle

Google Classroom - це інструмент для створення та керування онлайн-класами, який використовується для спільної роботи над завданнями, обміну матеріалами та комунікації.

Переваги: Інтегрований з Google Apps, масштабований для великої кількості користувачів, простий у використанні.

Недоліки: Можливі обмеження в налаштуванні, не завжди ідеально підходить для великих організацій або складних проектів.

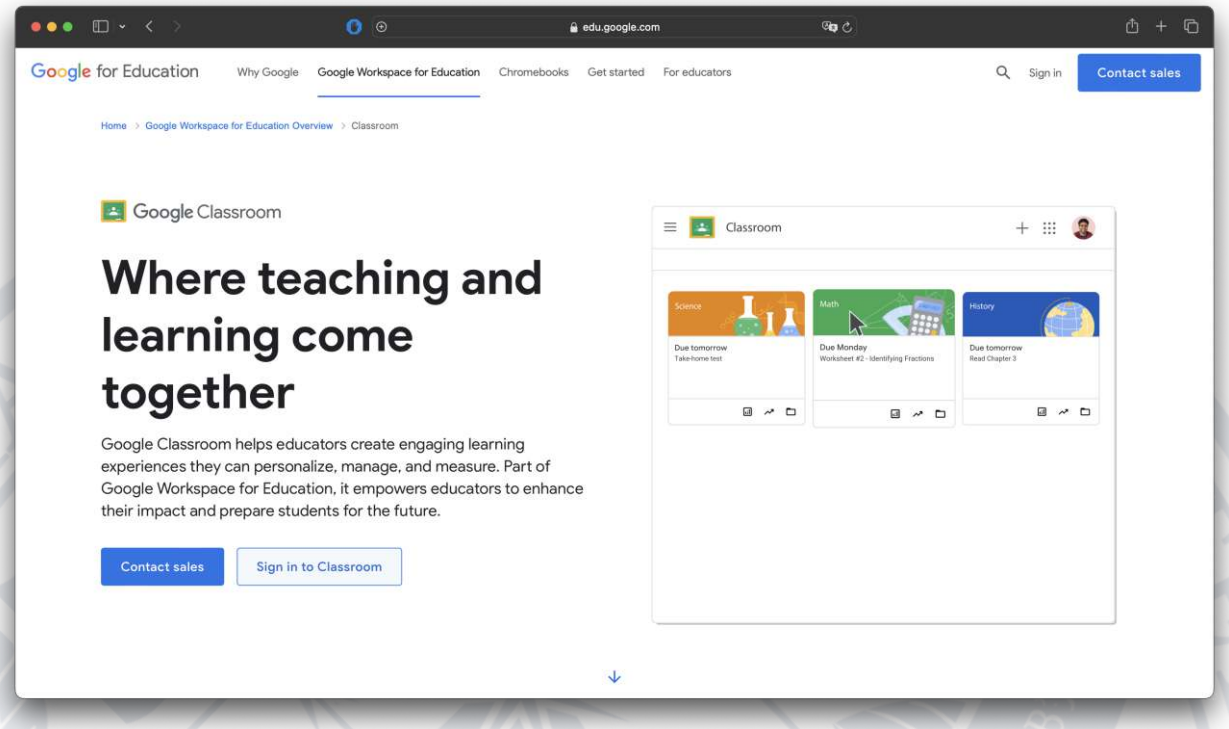

Рисунок 1.2 – інформація про Google Classroom

Edmodo - це соціальна мережа для навчальних закладів, яка дозволяє створювати віртуальні класи, спільно працювати над завданнями та спілкуватися з учителями та однокласниками.

Переваги: Легкий у використанні, має сучасний інтерфейс, дозволяє створювати зручні спільноти для обміну документами та спілкування.

Недоліки: Може бути обмежений у функціональності порівняно з іншими платформами, особливо для великих організацій.

Schoology – це інтегрована платформа для навчання та управління навчальним процесом, яка надає комплексний набір інструментів для ефективної організації навчання. Однією з ключових переваг Schoology є його універсальність і гнучкість, що дозволяє вчителям створювати різноманітні навчальні матеріали, завдання та тести, а також створювати віртуальні класи для спільної роботи та обміну досвідом.

Переваги: Інтегрована платформа з універсальними можливостями для управління навчанням та співпраці між учасниками навчального процесу.

Недоліки: Обмежена можливість кастомізації інтерфейсу та вимоги до технічної підтримки для налагодження та налаштування.

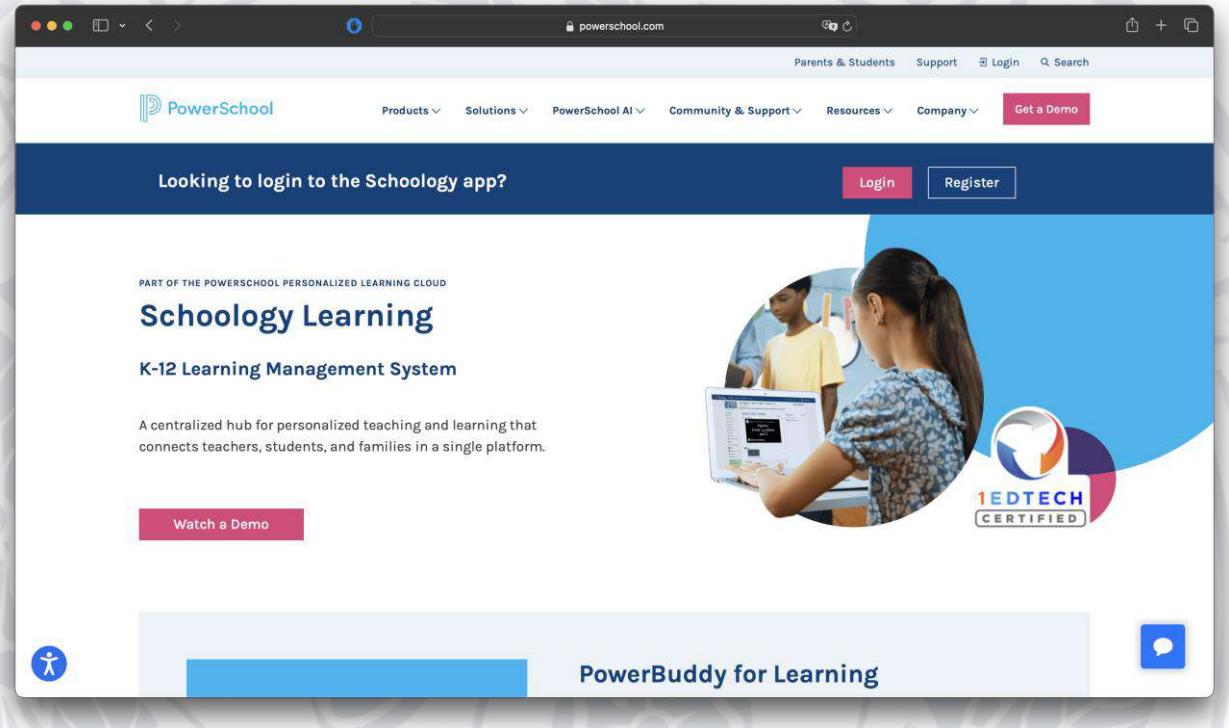

Рисунок 1.3 – головна сторінка платформи Schoology

Anthology - це інтегрована система для управління навчальним процесом, яка включає в себе можливості для створення курсів, завдань, форумів та інших інструментів.

Переваги: Розширені функціональні можливості для обліку документації та навчального процесу, включаючи відео-та аудіо-матеріали.

Недоліки: Система може бути складною для освоєння, потребує значних витрат на налаштування та підтримку.

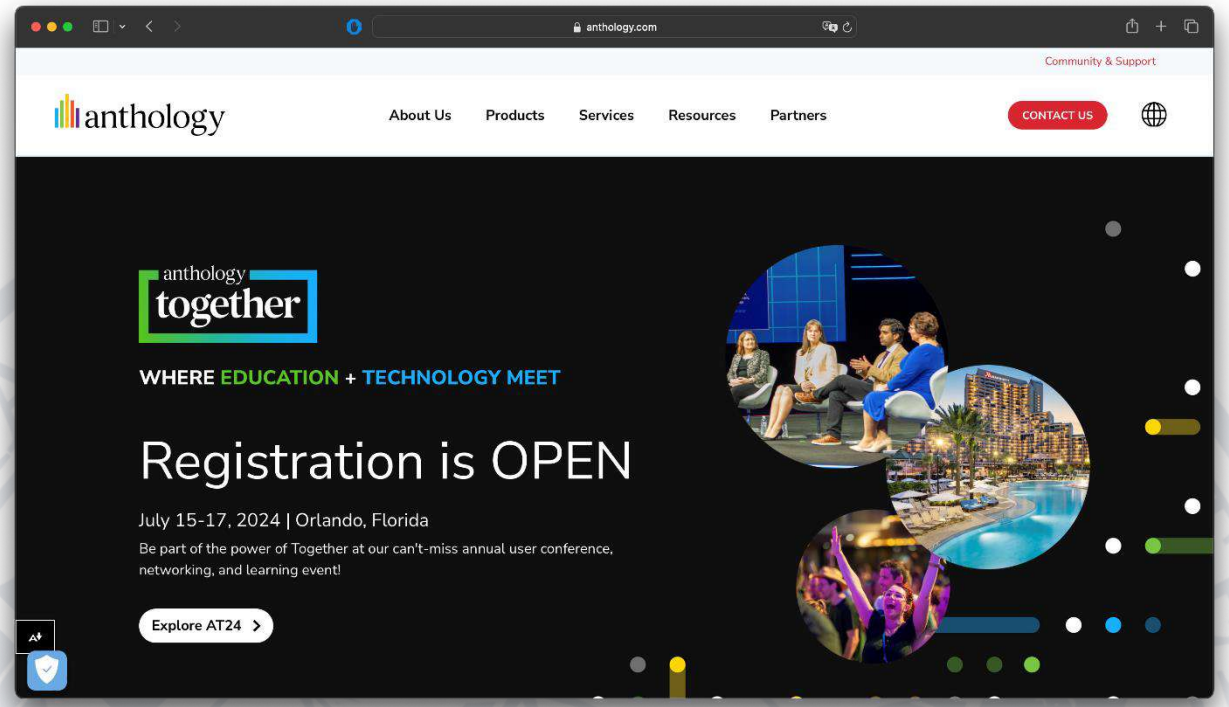

Рисунок 1.4 – сторінка регістрації платформи Schoology

РОЗДІЛ 2
АНАЛІЗ ТЕХНОЛОГІЧНИХ РІШЕНЬ ТА ІНСТРУМЕНТІВ ДЛЯ РОЗРОБКИ ДОДАТКУ

2.1 Підходи та методи розробки програмного забезпечення для обліку документації

Ітеративний підхід у розробці програмного забезпечення для обліку документації - це методологія, яка передбачає розбиття процесу розробки на невеликі ітерації або ітераційні цикли. Кожна ітерація представляє собою повний цикл розробки, який включає в себе кроки від планування до впровадження.

Планування ітерації: Починаючи з планування, команда розробників визначає, які функції чи частини функціональності будуть реалізовані протягом цієї ітерації. Визначаються обсяг робіт, ресурси, терміни та очікуваний результат.

Аналіз вимог і проектування: На цьому етапі розробники аналізують вимоги до системи обліку документації та визначають оптимальні шляхи їх реалізації. Планується архітектура системи та визначаються деталі реалізації.

Розробка та реалізація: Функціональність системи реалізується відповідно до визначених вимог та проекту. Розробники пишуть код, виконують тестування та вносять необхідні коригування.

Тестування та валідація: Після завершення розробки функціоналу проводиться тестування, включаючи функціональне тестування, тестування на злам та інші види тестів для перевірки коректності та надійності роботи системи.

Впровадження та збір відгуків: Після успішного завершення ітерації функціонал системи впроваджується та стає доступним для користувачів. Збираються відгуки від користувачів, які використовують систему, щоб врахувати їх у наступних ітераціях та вдосконалити продукт.

Цей процес повторюється через послідовні ітерації, де кожна нова ітерація вдосконалює функціональність системи та враховує відгуки користувачів та зміни у вимогах. Такий підхід дозволяє поступово розвивати та поліпшувати

систему обліку документації, забезпечуючи високу якість та відповідність потребам користувачів.

Модульна архітектура - це підхід до розробки програмного забезпечення, при якому система розбивається на невеликі, самодостатні модулі. Кожен модуль відповідає за виконання певної функції або набору пов'язаних функцій і може бути розвиваний, тестований та підтримуваний окремо від інших модулів. Такий підхід дозволяє забезпечити більшу гнучкість та ефективність у розробці програмного забезпечення.

Один із головних плюсів модульної архітектури - це розширені можливості розвитку. Розробники можуть працювати над окремими модулями, не змінюючи при цьому інші частини системи, що дозволяє ефективно додавати нові функції та вдосконалювати існуючі без ризику впливу на решту системи.

Крім того, модульна архітектура спрощує тестування програмного забезпечення, оскільки кожен модуль може бути протестований окремо. Це дозволяє виявляти та виправляти помилки на ранніх етапах розробки, що підвищує надійність та якість продукту.

Крім того, модульна структура дозволяє зберігати код системи в логічно відокремлених частинах, що полегшує управління та підтримку програмного забезпечення. Кожен модуль може бути розроблений, протестований та модифікований незалежно від інших, що забезпечує більшу гнучкість та швидкість реакції на зміни.

Тестування на ранніх етапах розробки є важливою практикою, яка полягає в використанні методів тестування ще на початкових стадіях створення програмного продукту. Це дозволяє виявляти та виправляти помилки ще до завершення проекту та перед впровадженням програми. Основна мета - зменшити вартість виправлення помилок та забезпечити високу якість програмного продукту.

Одним з переваг тестування на ранніх етапах є можливість виявлення проблем та недоліків в програмному продукті на ранніх стадіях розробки, коли їх виправлення виявляється менш складним та менш витратним. Це допомагає

зменшити кількість помилок, які потраплять у фінальну версію продукту, і підвищує його якість. [6]

Тестування на ранніх етапах також сприяє покращенню спілкування між членами команди розробки, оскільки воно вимагає активної взаємодії між розробниками та тестувальниками. Це сприяє зниженню ризиків та підвищенню ефективності розробки програмного забезпечення.

Нарешті, тестування на ранніх етапах дозволяє виявити та виправити помилки в програмному коді до того, як вони стануть критичними проблемами, що можуть виникнути під час використання програмного продукту реальними користувачами. Це дозволяє зберегти репутацію компанії та покращити задоволення користувачів від використання програмного продукту.

Принципи архітектури програмного забезпечення (ПЗ) - це основні принципи та підходи до організації структури програмного забезпечення з метою забезпечення його ефективності, масштабованості та гнучкості. При аналізі різних архітектурних шаблонів для створення системи обліку документації, важливо враховувати такі принципи:

Розділення відповідальностей (Separation of Concerns): Цей принцип передбачає розділення системи на окремі модулі або компоненти, кожен з яких відповідає за виконання конкретної функції. Наприклад, окремі компоненти можуть бути відповідальні за зберігання документів, обробку запитів користувачів, відображення інтерфейсу тощо.

Інкапсуляція (Encapsulation): Цей принцип передбачає обмеження доступу до внутрішньої структури компонентів системи, забезпечуючи доступ до них лише через визначені інтерфейси. Інкапсуляція дозволяє зменшити залежність між різними частинами системи і сприяє підвищенню надійності та безпеки програмного забезпечення.

Висока зграбність (High Cohesion): Цей принцип стверджує, що компоненти системи повинні бути спрямовані на виконання однієї конкретної задачі або функції. Висока зграбність допомагає забезпечити чіткість та розуміння структури системи, а також полегшує підтримку та модифікацію коду.

Низька залежність (Low Coupling): Цей принцип передбачає мінімізацію зв'язків між компонентами системи. Низька залежність дозволяє змінювати один компонент без необхідності вносити зміни в інші, що робить систему більш гнучкою та легше змінюваною.

Масштабованість (Scalability): Цей принцип передбачає можливість збільшення обсягу обробки даних або обсягу роботи системи без значних змін в її архітектурі. Масштабованість дозволяє системі ефективно працювати при зростанні обсягів даних або користувацького навантаження.

Управління версіями та контроль змін у програмному забезпеченні включає в себе використання систем контролю версій, таких як Git, для ефективного відстеження та керування розвитком проекту. Ці системи дозволяють розробникам зберігати копії коду, вносити зміни та відстежувати історію всіх змін.

Один з ключових аспектів управління версіями - це гілкування, що дозволяє розробникам створювати відокремлені гілки для розвитку конкретних функцій чи вирішення завдань. Це полегшує паралельну роботу та забезпечує безпеку відносно основної гілки.

Процес злиття (Merge) дозволяє об'єднувати зміни з різних гілок після завершення роботи. Це забезпечує гнучкість та контроль над об'єднанням різних функціональних частин коду.

Історія змін, яка зберігається в системі контролю версій, дозволяє відстежувати еволюцію коду та відновлювати попередні версії програми у випадку потреби.

Крім того, системи контролю версій можуть надавати можливість контролювати доступ до репозиторію коду, визначати права доступу для різних користувачів та груп розробників, що забезпечує безпеку та конфіденційність коду.

2.2 Модель та концепція функціонування додатку для обліку документації

Визначення об'єктів обліку - це процес ідентифікації основних сутностей, що будуть взаємодіяти в системі обліку документації. Основними об'єктами можуть бути, наприклад, документи, користувачі, відділи та інші сутності, які відображають основні аспекти діяльності організації або кафедри.

Документи: Це основні об'єкти обліку, які включають в себе різноманітну документацію, яка зберігається та обробляється на кафедрі або в навчальному закладі. Це можуть бути навчальні матеріали, наукові роботи, звіти, лабораторні роботи та інші.

Користувачі: Ці об'єкти представляють собою користувачів системи, таких як адміністратори, викладачі, студенти та інші особи, які мають доступ до системи обліку документації з різними ролями та правами.

Відділи: Ці об'єкти відображають структуру організації або навчального закладу та можуть включати різні підрозділи, кафедри, факультети тощо. Кожен відділ може мати свою власну документацію та користувачів, які з ним взаємодіють.

Додатково до основних об'єктів обліку можуть бути визначені додаткові сутності, які відображають конкретні аспекти діяльності кафедри чи навчального закладу. Наприклад, це може включати в себе групи студентів, наукові проекти, аудиторії для проведення занять, бібліотечний фонд та інші. Визначення цих об'єктів дозволяє більш повно налаштувати систему обліку документації під потреби конкретного закладу та забезпечити більш точне відображення його діяльності.

Крім того, важливим аспектом є визначення взаємозв'язків між об'єктами обліку. Наприклад, документи можуть бути пов'язані з конкретними користувачами або відділами, студенти можуть бути призначені до певних груп чи курсів, а наукові проекти можуть бути пов'язані з викладачами чи дослідницькими групами. Встановлення таких взаємозв'язків допомагає

побудувати більш комплексну систему обліку, яка враховує реальні взаємодії та процеси в навчальному закладі або кафедрі.

Визначення цих об'єктів допомагає побудувати базову структуру додатку та визначити необхідні функції для їх обробки. Наприклад, для об'єкта "Документи" можуть бути необхідні функції додавання, редагування та видалення документів, а для об'єкта "Користувачі" - функції управління користувачами та їхніми ролями.

Архітектурний огляд додає розуміння структури та організації додатку для обліку документації. Це включає в себе:

Модель архітектури: Вибір між розподіленою або централізованою моделлю. Розподілена архітектура передбачає розміщення функцій та даних на різних серверах, що може забезпечити вищу масштабованість та надійність. У централізованій архітектурі всі функції та дані зосереджені на одному сервері.

Тип бази даних: Вибір типу бази даних, такого як реляційна, нереляційна або гібридна. Реляційна база даних, як MySQL або PostgreSQL, забезпечує структурований підхід до зберігання даних, тоді як нереляційна база даних, така як MongoDB або Cassandra, може бути корисною для зберігання неструктурованих даних. [7]

Забезпечення безпеки: Розробка стратегії безпеки для захисту конфіденційності та цілісності даних. Це включає в себе механізми аутентифікації, авторизації, шифрування та інші заходи безпеки.

Масштабованість та продуктивність: Врахування можливостей масштабування додатку для підтримки зростаючої кількості користувачів та обсягів даних. Це може включати в себе розгалуженість серверів, кешування та інші методи оптимізації продуктивності.

Інтеграція з іншими системами: Визначення підходів до інтеграції з іншими програмними засобами, такими як системи управління навчанням, електронні бібліотеки та інші. Це дозволить покращити обмін даними та функціональність системи.

Функціональні вимоги визначають основні функції та можливості, які має надавати додаток для обліку документації. Основні аспекти цього пункту включаються в:

Додавання, редагування та видалення документів. Система повинна забезпечувати можливість додавання нових документів, редагування існуючих та видалення неактуальних документів. Це забезпечить актуальність та точність документації.

Пошук та фільтрація. Додаток повинен надавати можливість швидкого та зручного пошуку документів за різними критеріями, такими як назва, автор, дата створення тощо. Фільтрація документів за різними параметрами також може бути корисною для зручного організації та перегляду.

Забезпечення доступу різним користувачам. Система повинна мати механізми авторизації та аутентифікації, які забезпечать доступ до різних функцій та документів в залежності від ролі користувача. Наприклад, викладачі можуть мати право додавати та редагувати документи, тоді як студенти можуть мати лише право перегляду.

Відстеження версій та історія змін. Система може забезпечувати можливість відстеження версій документів та збереження історії змін, що дозволить відновити попередні версії документів та відслідковувати зміни, внесені користувачами.

Експорт та імпорт даних. Додаток може надавати можливість експорту та імпорту документів, що дозволить обмінюватися даними з іншими системами та зберігати резервні копії документів.

Нефункціональні вимоги враховують аспекти, які не стосуються прямо функціональності додатку, але мають важливе значення для його успішного функціонування та використання користувачами. Ці аспекти включають:

Продуктивність: Додаток повинен працювати ефективно та швидко, навіть при великому обсязі даних або навантаженні. Це важливо для забезпечення задоволення користувачів та забезпечення плавної роботи системи.

Масштабованість: Система повинна бути здатна масштабуватися, тобто збільшувати свої можливості для обробки зростаючої кількості користувачів або обсягу даних. Це важливо для забезпечення стійкості та ефективності системи у майбутньому.

Безпека: Додаток повинен забезпечувати високий рівень безпеки для захисту конфіденційної інформації та запобігання несанкціонованому доступу. Це включає в себе захист даних, механізми аутентифікації та авторизації, шифрування та інші заходи безпеки.

Зручність інтерфейсу користувача: Інтерфейс користувача повинен бути зрозумілим, зручним та інтуїтивно зрозумілим. Це допоможе забезпечити зручність використання додатку та підвищити задоволення користувачів від його використання.

Надійність: Система повинна бути надійною та стабільною, здатною працювати без збоїв або відмов протягом тривалого часу. Це важливо для забезпечення безперебійної роботи додатку та запобігання втрати даних чи непередбачуваних ситуацій.

Взаємодія з користувачем включає опис інтерфейсу користувача, в якому визначається структура екранів, розташування елементів, кольорова схема та інші деталі, що забезпечують зручність та ефективність використання програми. Також описуються процеси взаємодії з користувачем, включаючи кроки, які вони виконують для взаємодії з додатком, такі як створення, редагування та видалення документів, пошук інформації та інші дії.

Управління сеансами користувачів також є важливою складовою, включаючи механізми аутентифікації, авторизації та управління правами доступу, що забезпечують безпеку та конфіденційність даних. Обробка помилок також важлива, оскільки вона включає в себе виявлення, реєстрацію та відображення помилок користувачеві, а також механізми відновлення роботи програми після виникнення помилок.

Доступність та ергономіка також враховуються при розробці, для забезпечення того, щоб програма була доступною для користувачів з

обмеженими можливостями та мала комфортний інтерфейс використання для всіх користувачів.

Під час опису взаємодії з користувачем також важливо враховувати естетичні та емоційні аспекти інтерфейсу. Це означає розгляд впливу дизайну та способу взаємодії на емоційний стан користувача та його задоволення від використання програми. Правильно підібраний дизайн та зручність інтерфейсу можуть позитивно впливати на сприйняття та використання додатку.

Додатково, важливо забезпечити відповідність інтерфейсу користувача сучасним тенденціям та очікуванням аудиторії. Це може включати в себе використання інтерактивних елементів, анімації, адаптивного дизайну для різних пристроїв та інші сучасні технології, які зроблять взаємодію з додатком більш приємною та зручною для користувача.

Нарешті, розгляд взаємодії з користувачем також передбачає постійне вдосконалення додатку на основі зворотного зв'язку від користувачів. Збирання та аналіз повернених відгуків дозволяє виявляти слабкі місця додатку, а також ідентифікувати можливості для покращення і розширення функціональності з метою забезпечення оптимального досвіду користувача.

2.3 Вибір інструментів для розробки

При виборі інструментів для розробки додатку для комп'ютера важливо врахувати різні аспекти, такі як тип додатку, технічні вимоги проекту та інші фактори. Ось декілька ключових аспектів, що варто врахувати:

- Мова програмування: Вибір мови програмування залежить від ваших технічних знань, потреб проекту та екосистеми, з якою ви працюєте. Наприклад, для розробки настільних додатків часто використовують мови програмування, такі як Java, C++, C#, Python, а для веб-додатків можуть використовуватися JavaScript, HTML/CSS, PHP тощо.

- Інтегроване середовище розробки (IDE): Вибір IDE залежить від особистих вподобань та специфіки проекту. Наприклад, для розробки на Java

можна використовувати IntelliJ IDEA або Eclipse, для розробки на C# - Visual Studio, а для веб-розробки - Visual Studio Code або Sublime Text.

- Фреймворки та бібліотеки: Використання фреймворків та бібліотек може значно полегшити процес розробки. Наприклад, для розробки веб-додатків можна використовувати фреймворки, такі як React, Angular або Vue.js, для графічних додатків - бібліотеки, такі як Qt для C++ або JavaFX для Java.

Проаналізувавши дані аспекти я вирішив розроблять свій застосунок на Python. Давайте розглянемо чому я зробив такий бивір.

По-перше, Python вважається однією з найбільш доступних мов програмування завдяки своєму простому синтаксису і чіткій структурі, що полегшує вивчення навіть початківцям. Далі, велика кількість стандартних бібліотек Python робить розробку ефективною і швидкою, оскільки вони дозволяють виконувати багато різних завдань без додаткового написання коду.

Крім того, Python є крос-платформеною мовою програмування, тому ваш додаток може запускатися на різних операційних системах, що робить його більш універсальним. Також варто відзначити активну спільноту Python, яка постійно працює над вдосконаленням мови та створенням нових інструментів, що сприяє розвитку екосистеми мови і забезпечує доступ до безлічі корисних ресурсів для розробників. [8]

Завершуючи, Python має широкі можливості застосування, від розробки веб-сайтів та веб-додатків до аналізу даних, наукових обчислень та штучного інтелекту. Вибір Python дозволить вам легко адаптувати свій додаток до потреб проекту та впевнено рухатися вперед у своїх розробках.

Для бази даних я вибрав PostgreSQL. Гарна взаємодія між Python та PostgreSQL є одним з ключових факторів, що робить їх популярними інструментами для розробки програмного забезпечення. Ось деякі аспекти цієї взаємодії:

Бібліотеки доступу до бази даних: В Python існує кілька бібліотек, що дозволяють взаємодіяти з PostgreSQL, таких як Psycopg2, SQLAlchemy, psycopg2-binary тощо. Ці бібліотеки надають розробникам широкий функціонал для роботи з базою даних: від виконання простих запитів до складних операцій з транзакціями та ORM.

ORM (Об'єктно-реляційне відображення): SQLAlchemy, одна з найпопулярніших бібліотек ORM для Python, надає можливість працювати з базою даних PostgreSQL у більш об'єктно-орієнтованому стилі. ORM дозволяє вам використовувати класи Python для представлення таблиць у базі даних та виконувати з ними операції, що спрощує розробку та підтримку коду. [9]

Підтримка вбудованих типів даних: PostgreSQL має розширену підтримку для різноманітних типів даних, таких як JSON, XML, географічні дані тощо. Python також має багато бібліотек для роботи з цими типами даних, що дозволяє легко обробляти та аналізувати дані у вашому додатку.

Підтримка асинхронного програмування: В останні роки стала популярною асинхронна розробка програмного забезпечення. PostgreSQL має підтримку асинхронних запитів, а Python також має ряд бібліотек (наприклад, asyncpg), які дозволяють взаємодіяти з базою даних асинхронно, що забезпечує високу продуктивність вашого додатку.

Зручність розробки та тестування: Python має простий синтаксис та багато інструментів для тестування, що робить розробку та тестування взаємодії з PostgreSQL досить зручними. Крім того, наявність багатьох фреймворків для веб-розробки (наприклад, Django або Flask), які мають гарну підтримку для PostgreSQL, робить процес розробки ще ефективнішим.

РОЗДІЛ 3
РОЗРОБКА ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ДОДАТКУ ДЛЯ ОБЛІКУ ДОКУМЕНТАЦІЇ КАФЕДРИ ЗВО

3.1 Архітектурні рішення проекту додатку для обліку документації

В проекті є 2 частини це база даних та інтерфейс користувача. Давайте розглянемо ці 2 аспекти більш детально.

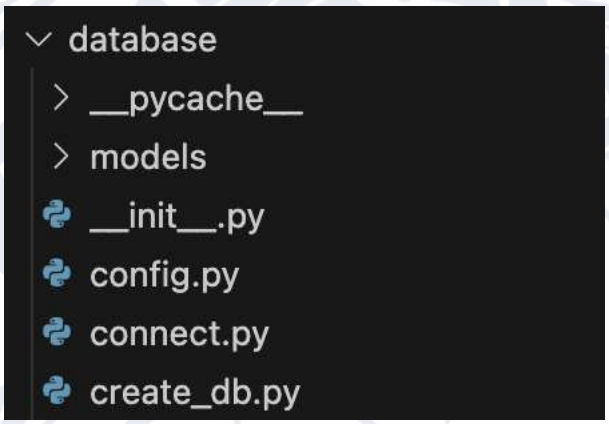

Рисунок 3.1 – вміст папки “database”

В папці з базою даних присутні файли для створення, конфігурації та під’єднання до бази. І також є папка з моделями таблиць.

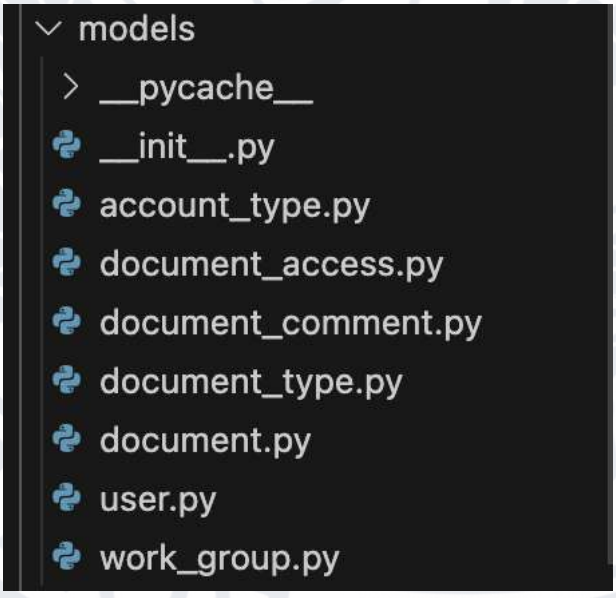

Рисунок 3.2 – вміст папки “database/models”

Далі перейдемо до графічної частини додатку.

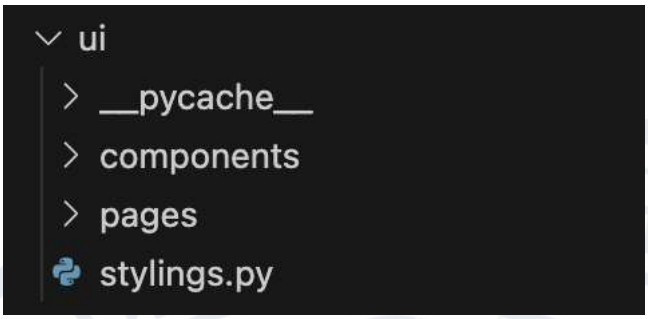

Рисунок 3.3 – вміст папки “ui”

В папці з UI присутні папки компонентів та сторінок, а також файл з стилями для компонентів.

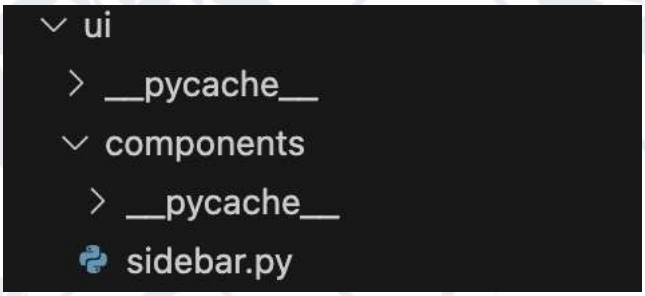

Рисунок 3.4 – вміст папки “ui/components”

В папці з компонентами присутній файл для відображення бокової панелі.

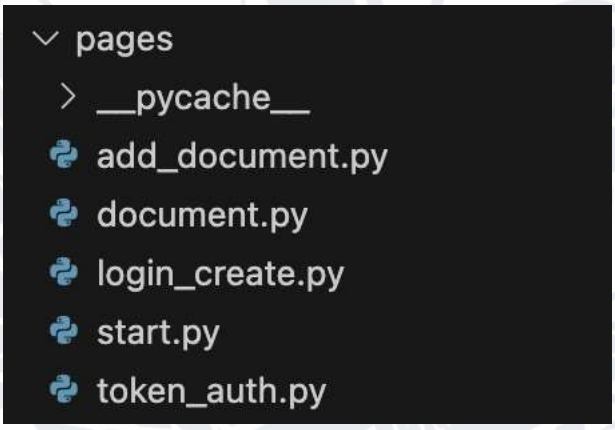

Рисунок 3.5 – вміст папки “ui/pages”

В папці з сторінками присутні файли всіх сторінок які є в додатку.

3.2 Реалізація серверної частини додатку

Почнемо з файлу для підключення до бази даних.

```
engine = create_engine("postgresql+psycopg2:///data.db", echo=False)
if not database_exists(engine.url):
    create_database(engine.url)
```

Тут використовується бібліотека SQLAlchemy для роботи з базою даних PostgreSQL. Давайте розберемо його:

- Створення підключення до бази даних: В першому рядку ми створюємо об'єкт engine, який представляє собою з'єднання з базою даних PostgreSQL. Ми використовуємо функцію create_engine з бібліотеки SQLAlchemy для створення цього з'єднання. У параметрі postgresql+psycopg2:///data.db ми вказуємо тип бази даних (PostgreSQL) та розширення для взаємодії з PostgreSQL (psycopg2). Строка ///data.db вказує на те, що база даних має назву data.db. Параметр echo=False вимикає виведення SQL-запитів у консоль.
- Перевірка існування бази даних і її створення: Наступний рядок містить умовний оператор if, який перевіряє, чи існує база даних, яку ми вказали у URL engine.url. Функція database_exists перевіряє наявність бази даних за URL-адресою. Якщо база даних не існує, то ми викликаємо функцію create_database, яка створює нову базу даних за URL-адресою.

Далі перейдемо до моделей сущностей.

Тип акаунту

```
class AccountType(Base):
    __tablename__ = "account_types"

    id: Mapped[int] = mapped_column(
        primary_key=True, nullable=False, autoincrement=True
    )
    name: Mapped[str] = mapped_column(nullable=False)
    creation_time: Mapped[str] = mapped_column(default=str(dt.datetime.today()))
    users = relationship("User")

    def __repr__(self) -> str:
        return f"AccountType(id={self.id!r}, name={self.name!r})"

    def create_account_type(name):
        session = Session(bind=engine)
        student = AccountType(name=name)
        session.add(student)
        session.commit()
```

Створюємо клас AccountType, який відображає таблицю account_types в базі даних за допомогою ORM бібліотеки SQLAlchemy. Давайте розберемо його:

Оголошення класу: Ми визначаємо клас AccountType, який успадковується від базового класу Base. Це типовий підхід у SQLAlchemy для використання ORM.

Опис таблиці: За допомогою атрибуту __tablename__ ми вказуємо назву таблиці в базі даних, з якою буде пов'язаний цей клас. У цьому випадку це таблиця account_types.

Колонки таблиці: Ми описуємо колонки таблиці за допомогою атрибутів класу AccountType. Наприклад, id, name та creation_time відображають колонки таблиці. Ми також вказуємо параметри колонок, такі як primary_key, nullable та default, за допомогою функції mapped_column.

Відношення: У класі ми вказуємо відношення між таблицею AccountType та іншою таблицею User. В даному випадку users - це зв'язок один до багатьох.

Метод repr: Цей метод визначає представлення об'єкту класу AccountType у вигляді рядка. Це допомагає при відладці, щоб бачити інформацію про об'єкти.

Метод create_account_type: Цей метод створює новий запис в таблиці account_types. Він створює об'єкт AccountType з заданим ім'ям, додає його до сесії та зберігає зміни у базі даних.

Доступ до дукументу

```
class DocumentAccess(Base):
    __tablename__ = "document_accesses"

    work_group_id: Mapped[int] = mapped_column(
        ForeignKey("work_groups.id"), primary_key=True
    )
    document_id: Mapped[int] = mapped_column(
        ForeignKey("documents.id"), primary_key=True
    )
    user_id: Mapped[int] = mapped_column(ForeignKey("users.id"), nullable=False)
    access_type: Mapped[str] = mapped_column(nullable=False)
    creation_time: Mapped[str] = mapped_column(default=str(dt.datetime.today()))

    def __repr__(self) -> str:
        return f"User(id={self.id!r}, name={self.name!r}, fullname={self.fullname!r})"
    def create_document_access():
    ...
```

Цей код визначає клас DocumentAccess, який відображає таблицю document_accesses в базі даних за допомогою ORM бібліотеки SQLAlchemy. Давайте розберемо кожен його елемент:

Оголошення класу: Визначається клас DocumentAccess, який успадковується від базового класу Base. Це типовий підхід у SQLAlchemy для використання ORM.

Опис таблиці: За допомогою атрибуту __tablename__ вказується назва таблиці в базі даних, яку представляє цей клас. У даному випадку це таблиця document_accesses.

Колонки таблиці: Кожен атрибут класу відповідає колонці таблиці. Наприклад, work_group_id, document_id, user_id, access_type та creation_time відображають відповідні колонки у таблиці. Використовуються функції mapped_column для визначення параметрів колонок, таких як primary_key, nullable, а також для визначення зовнішніх ключів (ForeignKey), які вказують на зв'язок з іншими таблицями.

Метод repr: Цей метод визначає представлення об'єкту класу DocumentAccess у вигляді рядка. Використовується для зручності відладки та виведення інформації про об'єкти.

Метод create_document_access: Цей метод, відповідає за створення доступу до документу. У ньому передбачено роботу з об'єктами сесії, додавання нового доступу та збереження змін у базі даних.

Коментар документа

```
class DocumentComment(Base):
    __tablename__ = "document_comments"

    id: Mapped[int] = mapped_column(
        primary_key=True, nullable=False, autoincrement=True
    )
    comment: Mapped[str] = mapped_column(nullable=False)
    owner: Mapped[int] = mapped_column(ForeignKey("users.id"))
    documnet_id: Mapped[int] = mapped_column(ForeignKey("documents.id"))
    creation_time: Mapped[str] = mapped_column(default=str(dt.datetime.today()))

    def __repr__(self) -> str:
        return f"DocumentComment(id={self.id!r}, comment={self.comment!r},
owner={self.owner!r}, document_id={self.documnet_id!r})"
```

Тут ми створюємо клас DocumentComment, який відображає таблицю document_comments в базі даних за допомогою ORM бібліотеки SQLAlchemy. Давайте розберемо кожен його елемент:

- Оголошення класу: Визначається клас DocumentComment, який успадковується від базового класу Base. Це типовий підхід у SQLAlchemy для використання ORM.
- Опис таблиці: За допомогою атрибуту __tablename__ вказується назва таблиці в базі даних, яку представляє цей клас. У даному випадку це таблиця document_comments.
- Колонки таблиці: Кожен атрибут класу відповідає колонці таблиці. Наприклад, id, comment, owner, documnet_id та creation_time відображають відповідні колонки у таблиці. Використовуються функції mapped_column для визначення параметрів колонок, таких як primary_key, nullable, autoincrement, а також для визначення зовнішніх ключів (ForeignKey), які вказують на зв'язок з іншими таблицями.
- Метод repr: Цей метод визначає представлення об'єкту класу DocumentComment у вигляді рядка. Використовується для зручності відладки та виведення інформації про об'єкти.

Тип документа

```
class DocumentType(Base):
    __tablename__ = "document_types"

    id: Mapped[int] = mapped_column(
        primary_key=True, nullable=False, autoincrement=True
    )
    name: Mapped[str] = mapped_column(nullable=False)
    description: Mapped[Optional[str]]
    creation_time: Mapped[str] = mapped_column(default=str(dt.datetime.today()))
    documents = relationship("Document", back_populates="document_type")

    def __repr__(self) -> str:
        return f"DocumentType(id={self.id!r}, name={self.name!r},
description={self.description!r})"

    def create_document_type(name=None):
        ...
```

Тут ми по аналогії з минулими таблицями визначаємо клас DocumentType в якому визначаємо необхідні поля та методи.

Користувач

```
class User(Base):
    __tablename__ = "users"

    id: Mapped[int] = mapped_column(
        primary_key=True, nullable=False, autoincrement=True
    )
    name: Mapped[Optional[str]]
    fullname: Mapped[Optional[str]]
    midlename: Mapped[Optional[str]]
    email: Mapped[str] = mapped_column(nullable=False)
    password: Mapped[str] = mapped_column(nullable=False)
    account_type_id: Mapped[int] = mapped_column(
        ForeignKey("account_types.id"), default=1
    )
    creation_time: Mapped[str] = mapped_column(default=str(dt.datetime.today()))
    documents: Mapped[List["Document"]] = relationship(backref="user")

    def __repr__(self) -> str:
        return f"User(id={self.id!r}, name={self.name!r}, fullname={self.fullname!r})"
```

Тут як вже робили раніше створюємо клас та методи.

Документ

Тут доволі об’ємний код тому давайте його більш детально розглянемо.

```
class Document(Base):
    __tablename__ = "documents"
    id: Mapped[int] = mapped_column(
        primary_key=True, nullable=False, autoincrement=True
    )
    title: Mapped[str] = mapped_column(nullable=False)
    content: Mapped[Optional[str]]
    owner: Mapped[int] = mapped_column(ForeignKey("users.id"))
    file_source: Mapped[str] = mapped_column(nullable=False)
    document_type_id: Mapped[int] = mapped_column(ForeignKey("document_types.id"))
    status: Mapped[str] = mapped_column(nullable=False)
    creation_time: Mapped[str] = mapped_column(default=str(dt.datetime.today()))
    document_type = relationship("DocumentType", back_populates="documents")
    def __repr__(self) -> str:
        return f"Document(id={self.id!r}, title={self.title!r}, owner={self.owner!r}, type={self.document_type_id})"
    def create_doc():
        session = Session(bind=engine)
        random_title = "".join(random.choices(string.ascii_lowercase, k=5))
        random_content = "".join(random.choices(string.ascii_lowercase, k=30))
        document = Document(
            title=random_title,
            ...
            status="new",
        )
        session.add(document)
        session.commit()
```

Цей код визначає клас Document, який відображає таблицю documents в базі даних за допомогою ORM бібліотеки SQLAlchemy. Розглянемо кожну його частину:

Оголошення класу: Визначається клас Document, який успадковується від базового класу Base. Це типовий підхід у SQLAlchemy для використання ORM.

Опис таблиці: За допомогою атрибуту __tablename__ вказується назва таблиці в базі даних, яку представляє цей клас. У даному випадку це таблиця documents.

Колонки таблиці: Кожен атрибут класу відповідає колонці таблиці. Наприклад, id, title, content, owner, file_source, document_type_id, status та creation_time відображають відповідні колонки у таблиці.

Використовуються функції mapped_column для визначення параметрів колонок, таких як primary_key, nullable, autoincrement, а також для визначення зовнішніх ключів (ForeignKey), які вказують на зв'язок з іншими таблицями.

Відношення до інших таблиць: Колонка document_type_id представляє зовнішній ключ до таблиці document_types, а document_type описує зв'язок з цією таблицею за допомогою відношення relationship.

Метод repr: Цей метод визначає представлення об'єкту класу Document у вигляді рядка. Використовується для зручності відладки та виведення інформації про об'єкти.

Метод create_doc(): Цей метод, відповідає за створення нового документа. У ньому передбачено роботу з об'єктами сесії, додавання нового документа та збереження змін у базі даних.

3.3 Реалізація клієнтської частини додатку

В даному проекті присутні наступні сторінки:

Вхід в акаунт та створення акаунту, головна з переліком документів, сторінка документу.

Вхід в акаунт

Визначаємо клас LoginPage, який є частиною веб-інтерфейсу та відповідає за сторінку входу користувача в систему. Розглянемо кожну його частину:

Оголошення класу: Визначається клас LoginPage, який успадковується від класу ft.View, що використовується у бібліотеці Fyne для створення графічних інтерфейсів.

Ініціалізація: У конструкторі класу задаються параметри для відображення сторінки в центрі екрана (vertical_alignment="center", horizontal_alignment="center") та маршрут (route="login-user"), за яким буде доступна сторінка.

Створення елементів інтерфейсу: У конструкторі також створюються елементи інтерфейсу, такі як заголовок сторінки (title="Doccu - Sign In User"), а також інші елементи, які не вказані у коді через знаки ....

Метод authenticate_user: Цей метод викликається при спробі аутентифікації користувача. Він отримує дані, введені користувачем (email та password), та перевіряє їх правильність за допомогою функції authenticate_credentials. Якщо аутентифікація пройшла успішно, користувач перенаправляється на стартову сторінку /start. Якщо ж аутентифікація не вдалася, виводиться повідомлення про помилку.

```
class LoginPage(ft.View):
    def __init__(self, page: ft.Page):
        super().__init__(
            route="login-user",
            vertical_alignment="center",
            horizontal_alignment="center",
        )
        self.page = page
        self.body = Body(
            title="Doccu - Sign In User",
            ...
            function=lambda _: self.authenticate_user(),
        )

        self.controls = [self.body]

    def authenticate_user(self):
        session = Session(bind=engine)
        email = self.body.email.value
        self.page.session.set("email", email)
        password = self.body.password.value
        if authenticate_credentials(email=email, password=password, session=session):
            self.body.invalid_credentials_alert.visible = "False"
            self.page.update()
            self.page.go("/start")
        else:
            self.body.invalid_credentials_alert.visible = "True"
            self.page.update()
```

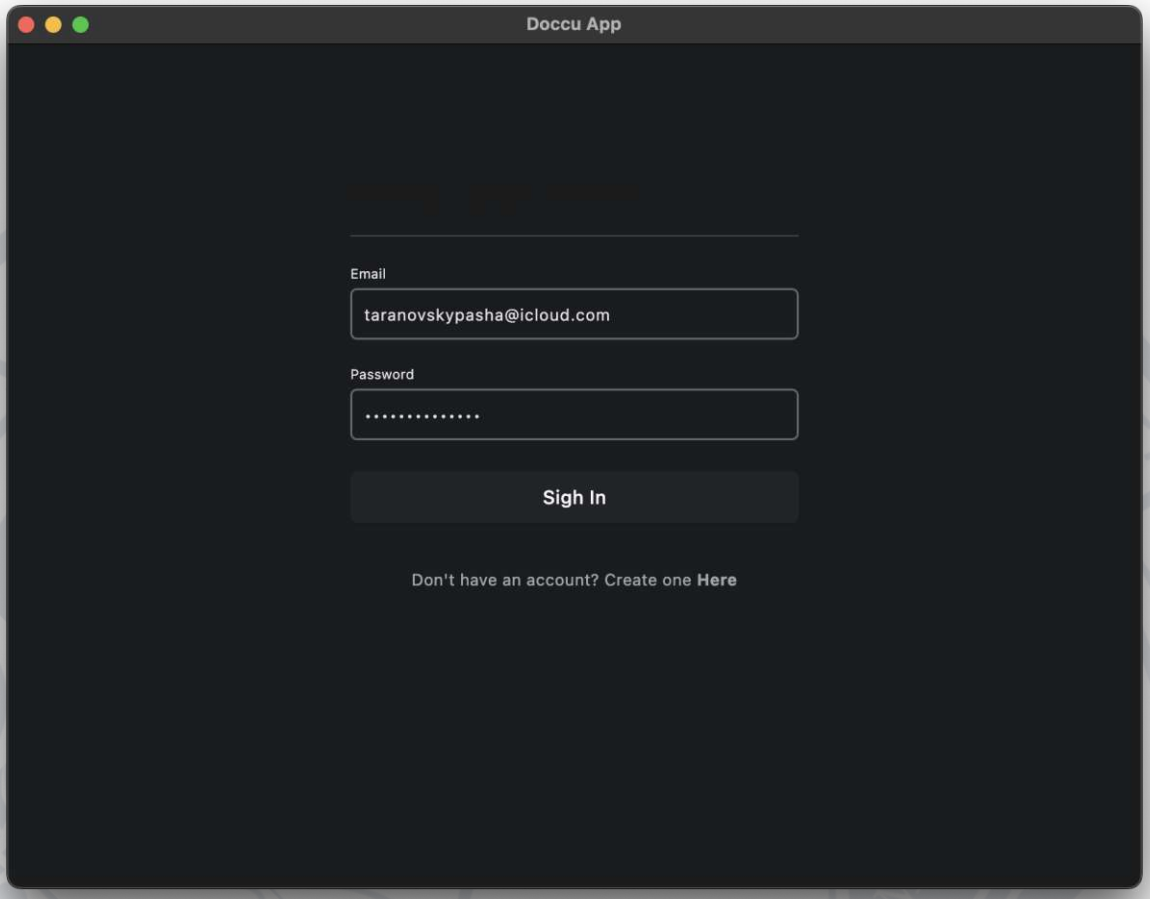

Рисунок 3.6 – вигляд сторінки для входу в акаунт

Створення акаунту

Визначаємо клас CreatePage, який відповідає за створення сторінки реєстрації користувача в системі. Розглянемо кожну його частину:

Оголошення класу: Визначається клас CreatePage, який успадковується від класу ft.View, що використовується у бібліотеці Fyne для створення графічних інтерфейсів.

Ініціалізація: У конструкторі класу задаються параметри для відображення сторінки в центрі екрана (vertical_alignment="center", horizontal_alignment="center") та маршрут (route="create-user"), за яким буде доступна сторінка.

Створення елементів інтерфейсу: У конструкторі створюються елементи інтерфейсу, такі як заголовок сторінки (title="Doccu - Create Account"), кнопка для створення акаунта, відповідні повідомлення та елементи введення.

Метод create_user: Цей метод викликається при спробі створення нового користувача. Він отримує дані, введені користувачем (email та password), та перевіряє їх відповідність вимогам за допомогою функцій validate_email та validate_password. Якщо введені дані відповідають вимогам, генерується та відправляється аутентифікаційний токен, після чого користувач перенаправляється на сторінку підтвердження реєстрації. Якщо ж введені дані не відповідають вимогам або користувач вже зареєстрований, відображається відповідне повідомлення.

```
class CreatePage(ft.View):
    def __init__(self, page: ft.Page):
        super().__init__(
            route="create-user",
            vertical_alignment="center",
            horizontal_alignment="center",
        )
        self.page = page
        self.body = Body(
            title="Doccu - Create Account",
            ...
            function=lambda _: self.create_user(),
        )

        self.controls = [self.body]

    def create_user(self):

        session = Session(bind=engine)
        email = self.body.email.value
        password = self.body.password.value
        if user_exists(email, session):
            print("You are already registered!")  # Create popup and clear the inputs
        else:
            email_validation_result = validate_email(email=email)
            password_validation_result = validate_password(len(password))
       ...
```

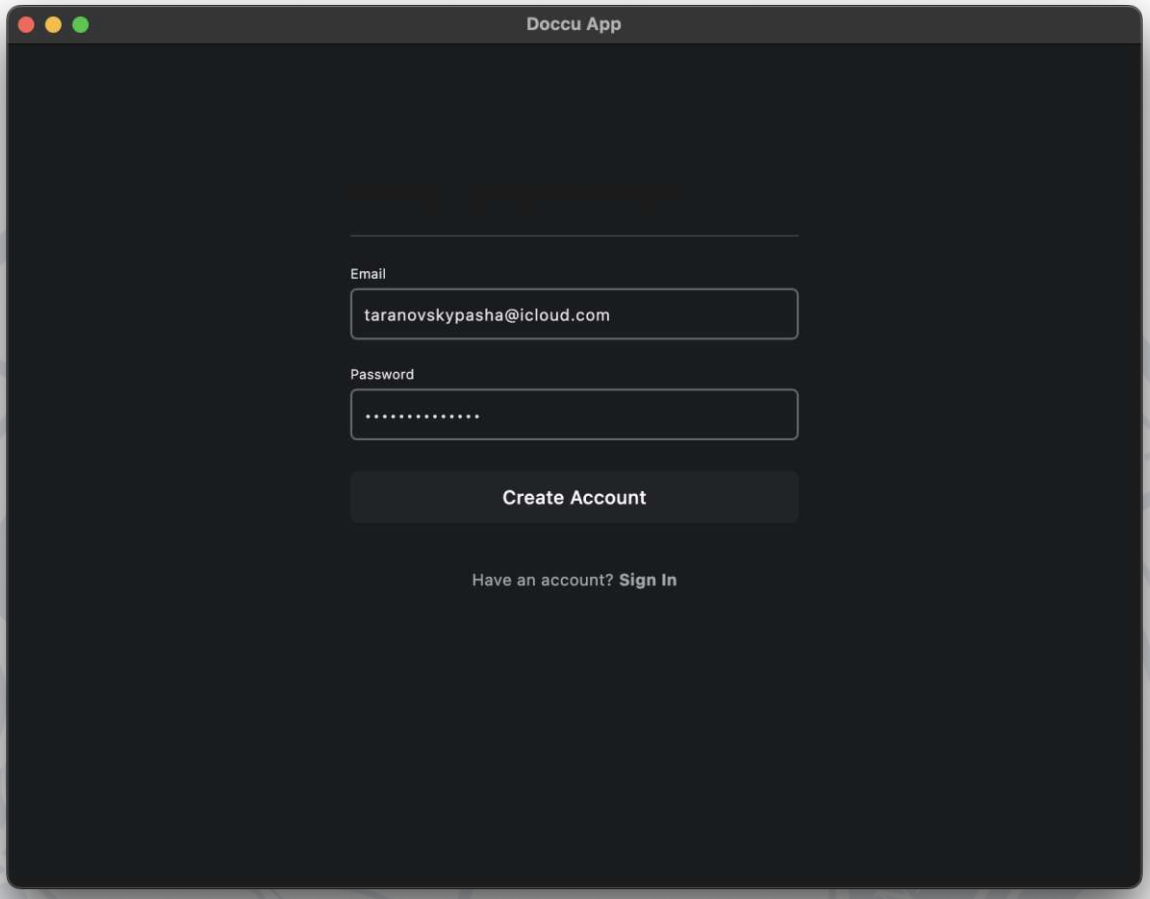

Рисунок 3.7 – вигляд сторінки для створення акаунту

Головна сторінка

Спочатку визначимо клас DocumentCard, який відображає карточку документа в інтерфейсі. Розглянемо кожну частину класу:

Ініціалізація класу: В конструкторі класу викликається конструктор батьківського класу ft.Container, а також виконуються додаткові операції для відображення карточки.

Отримання даних документа: Здійснюється отримання даних про документ з бази даних за допомогою об'єкта сесії. Витягуються інформація про тип документа, дату створення та інші необхідні дані.

Створення текстових полів: Створюються тексові поля для відображення інформації про документ, такі як назва документа, тип та дата створення.

Відображення в ряд: Елементи текстових полів розміщуються в ряд, використовуючи контейнер ft.Row. Також додатково відображається кнопка для зміни статусу документа.

```
class DocumentCard(ft.Container):
    def __init__(self, document, page):
        super().__init__(**document_card_style)

        session = Session(bind=engine)

        self.document = document

        self.type = (
            session.query(DocumentType)
            .filter(DocumentType.id == self.document.document_type_id)
            .first()
            .name
        )

        self.creation_time_obj = datetime.strptime(
            self.document.creation_time, "%Y-%m-%d %H:%M:%S.%f"
        )

        self.formated_creation_date = self.creation_time_obj.strftime("%Y-%m-%d")

        self.name = ft.Text(
            value=self.shorter_text(f" {self.document.id}    {self.document.title}"),
            color="#FAFAFF",
            bgcolor="#1C1C1C",
            height=20,
        )
       ...
```

Далі створюємо клас DocumentCardInfoTab, який представляє собою вкладку інформації для карточки документа. Давайте розглянемо його структуру та функціональність:

Ініціалізація класу: В конструкторі класу викликається конструктор батьківського класу ft.Container, а також виконується додаткова ініціалізація для відображення вкладки з інформацією.

Створення текстових полів: Створюються тексові поля для відображення заголовків колонок таблиці, такі як ID та назва, тип, дата створення та статус документа.

Створення контейнера для інформаційних полів: Кожне текстове поле згортається в окремий контейнер з відповідним розміром.

Відображення в ряд: Елементи текстових полів розміщуються в ряд, використовуючи контейнер ft.Row, що утворює загальний вигляд вкладки з інформацією про документ.

```
class DocumentCardInfoTab(ft.Container):
    def __init__(self):
        super().__init__(**document_card_style)

        self.alignment = ft.alignment.Alignment(-1, -1)
        self.id_title = self.InfoBar(
            Text=ft.Text(value=" ID    TITLE", color="white"), width=200
        )
        self.type = self.InfoBar(Text=ft.Text(value="TYPE", color="white"), width=120)
        self.creation_time = self.InfoBar(
            Text=ft.Text(value="CREATION TIME", color="white"), width=165
        )
        self.status = self.InfoBar(
            Text=ft.Text(value="STATUS", color="white"), width=80
        )
        self.content = ft.Row(
            controls=[
                self.id_title,
                self.type,
                self.creation_time,
                self.status,
            ]
        )

    def InfoBar(self, Text: ft.Text, width: int):
        return ft.Container(
            width=width,
            alignment=ft.alignment.center,
            content=ft.Row(controls=[Text]),
        )
```

І тепер визначено клас Body, який є контейнером для відображення головного вмісту сторінки. Розглянемо його складові:

Ініціалізація класу: В конструкторі викликається конструктор батьківського класу ft.Container. Специфічною особливістю цього класу є те, що він отримує список документів та об'єкт сторінки як аргументи.

Сайдбар: Створюється екземпляр класу SideBar, який представляє бічний бар.

Список документів: Створюється спискове представлення (ft.ListView), яке відображатиме документи. Для кожного документа створюється об'єкт DocumentCard і додається до спискового представлення.

Основний вміст сторінки: Створюється ряд (ft.Row), який містить сайдбар та колонку. Колонка складається з вкладки з інформацією про документи і спискового представлення документів.

```
class Body(ft.Container):

    def __init__(self, documents, page):
        super().__init__()
        self.sidebar = SideBar()
        self.doc_lv = ft.ListView(spacing=10, height=540, width=600)
        for i in range(len(documents)):
            self.doc_lv.controls.append(DocumentCard(documents[i], page))

        self.content = ft.Row(
            controls=[
                self.sidebar,
                ft.Column(controls=[DocumentCardInfoTab(), self.doc_lv]),
            ]
        )
```

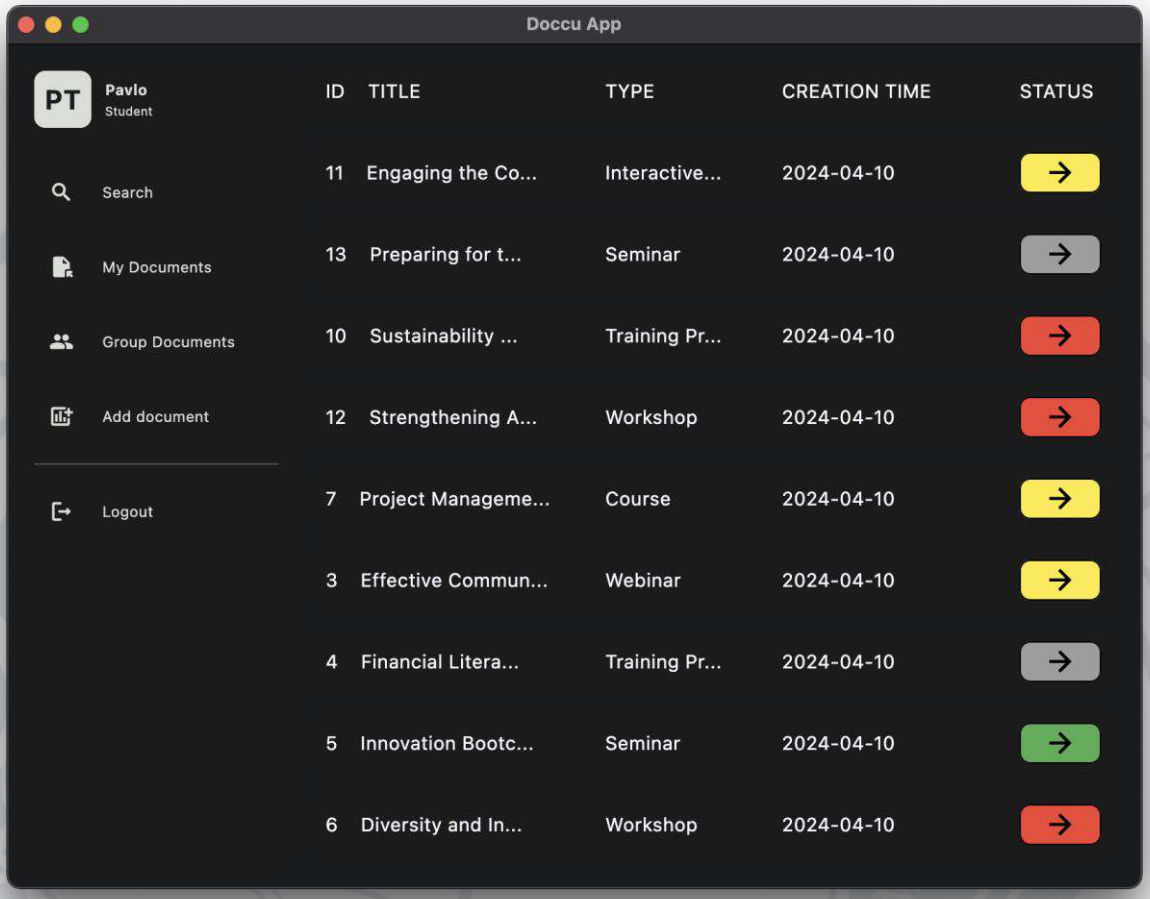

Рисунок 3.8 – вигляд головної сторінки

Сторінка документу

Визначено клас Body, який представляє форму редагування документу. Розглянемо його основні складові:

Ініціалізація класу: Конструктор отримує ідентифікатор документу, об'єкт сторінки та функцію, яка викликається при натисканні кнопки збереження. У конструкторі також викликається конструктор батьківського класу ft.Container.

Зчитування даних про документ: Здійснюється з'єднання з базою даних та отримання інформації про документ, включаючи назву, тип, статус, вміст тощо.

Створення візуальних елементів: Для відображення форми редагування створюються різні елементи, такі як текстові поля, випадаючі списки, кнопки тощо, і налаштовуються їх параметри.

Взаємодія з користувачем: Для деяких елементів визначаються обробники подій. Наприклад, при натисканні на кнопку видалення викликається метод delete_document, який видаляє документ з бази даних та перенаправляє користувача на початкову сторінку.

Організація вмісту сторінки: Всі елементи розташовуються на сторінці в певному порядку за допомогою контейнерів ft.Row та ft.Column. Таким чином, створюється зручний та логічний інтерфейс для редагування документів.

```
class Body(ft.Container):

    def __init__(self, document_id: str, page: ft.Page, function: callable):
        super().__init__()

        self.page = page

        session = Session(bind=engine)

        self.document_source = (
            session.query(Document.file_source)
            .filter(Document.id == document_id)
            .first()[0]
        )
        self.document_types = session.query(DocumentType.name).all()

        ...


    def delete_document(self, document_id: str):
        session = Session(bind=engine)
        document = session.query(Document).get(document_id)
        session.delete(document)
        session.commit()
        self.page.go("/start")
```

І на останок створюємо клас DocumentPage, який представляє сторінку для редагування конкретного документу. Розглянемо його основні складові:

Ініціалізація класу: Конструктор отримує об'єкт сторінки та ідентифікатор документу. Викликається також конструктор батьківського класу ft.View.

Створення тіла сторінки: В тілі сторінки створюється об'єкт класу Body, який представляє форму редагування документу. Для цього використовується

ідентифікатор документу та об'єкт сторінки. Також визначається функція, яка викликається при натисканні кнопки збереження змін.

Організація контролів сторінки: У властивості controls зберігається список контролів, який складається лише з тіла сторінки.

Збереження змін: Метод save_changes викликається при натисканні кнопки збереження змін. В ньому отримуються дані з форми редагування та зберігаються до бази даних.

```
class DocumentPage(ft.View):
    def __init__(self, page: ft.Page, document_id: str):
        super().__init__()

        self.page = page

        self.document_id = document_id

        self.body = Body(
            document_id=self.document_id,
            page=self.page,
            function=lambda _: self.save_changes(document_id=document_id),
        )

        self.controls = [self.body]

    def save_changes(self, document_id):
        session = Session(bind=engine)
        title = self.body.title.value
        type = self.body.document_type_dropdown.value
        type_id = (
            session.query(DocumentType.id).filter(DocumentType.name ==
type).first()[0]
        )
        status = self.body.status_dropdown.value.lower()
        document = session.query(Document).get(document_id)
        document.title = title
        document.document_type_id = type_id
        document.status = status
        session.commit()
        self.page.update()
```

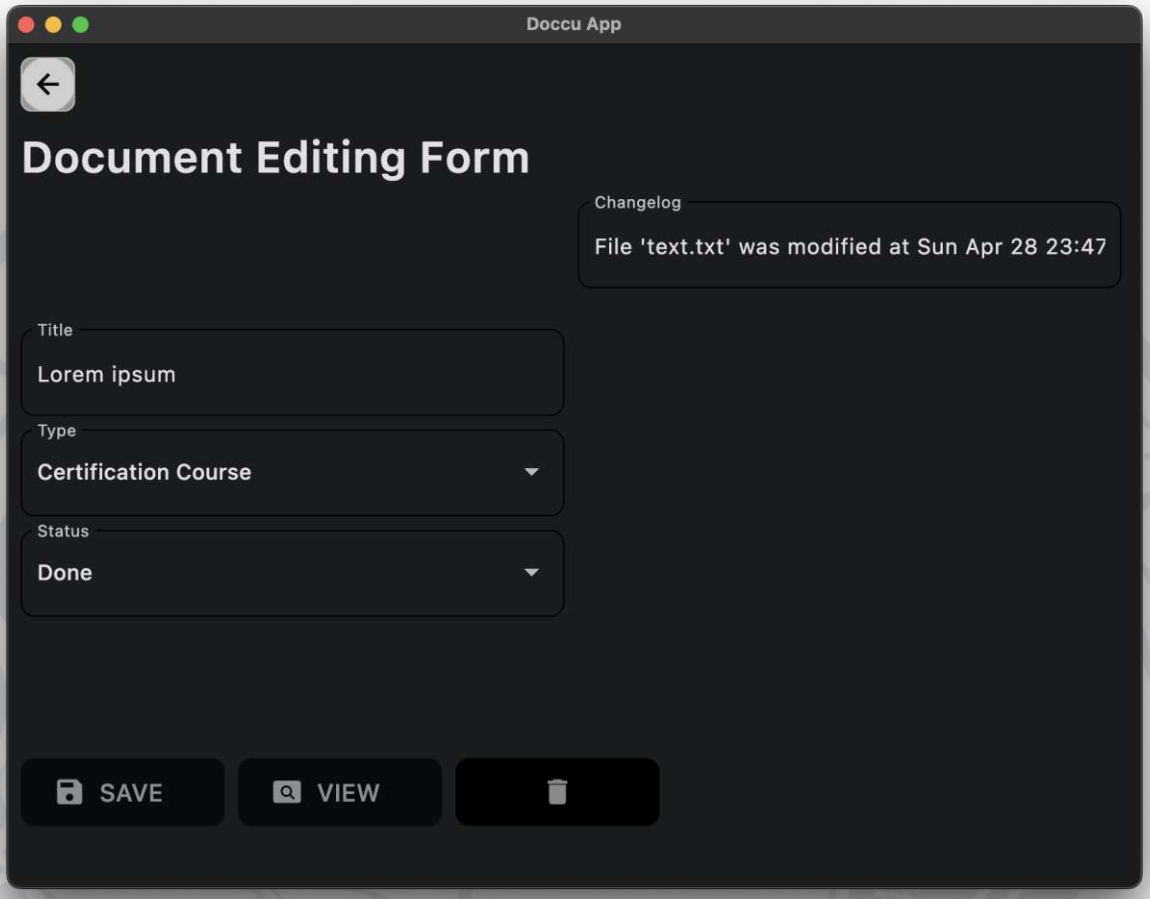

Рисунок 3.9 – вигляд сторінки файлу

ВИСНОВКИ

В рамках даної бакалаврської роботи був проведений дослідницький та практичний аналіз процесів обліку документації на кафедрі ЗВО. На основі цього аналізу була розроблена та впроваджена програмна система для автоматизації цих процесів.

У ході дослідження виявлено, що існуючі системи обліку документації на кафедрі мають ряд недоліків, таких як обмежена функціональність, складність у використанні та недостатня інтеграція з іншими програмними засобами. Ці недоліки створюють проблеми у швидкому доступі до інформації та управлінні документами.

Розроблена програмна система для обліку документації кафедри вирішує ці проблеми, надаючи користувачам зручний та ефективний інструмент для управління документами. Система має інтуїтивний інтерфейс, який дозволяє швидко знаходити та обробляти необхідну інформацію.

Впровадження програмної системи сприяє підвищенню продуктивності та оптимізації робочих процесів на кафедрі. Крім того, вона забезпечує надійне зберігання документів та забезпечує їх конфіденційність та цілісність.

Отже, розробка та впровадження програмної системи для обліку документації кафедри є актуальним та важливим кроком у напрямку модернізації управління документами на освітньому закладі. Вона сприяє покращенню робочих процесів, забезпечуючи ефективнішу роботу персоналу та підвищуючи загальний рівень якості навчального процесу.

СПИСОК ВИКОРИСТАНИХ ПОСИЛАНЬ

1. Документи і документування: поняття та види: https://osvita.ua/vnz/reports/dilovodstvo/24228/
2. Аналіз поточних проблем існуючих систем обліку документації: https://magazine.faaf.org.ua/oblikovo-analitichna-sistema-informaciyne-zabezpechennya-upravlinnya-pidpriemstvom.html
3. Типи баз даних: особливості, відмінності та приклади: https://dou.ua/lenta/articles/types-of-databases/
4. Що таке UX-дизайн та його принципи: https://outsourcing.team/uk/blog/design/shho-take-ux-dizajn-ta-jogo-printsipi/
5. Ключові методології розробки програмного забезпечення: робота команди зсередини: https://wezom.com.ua/ua/blog/metodologija-razrabotki-programmnogo-obespechenija
6. Раннє тестування та 4 причини, чому це важливо: https://qaukraine.online/rannie-testuvannia-ta-4-prychyny-chomu-tse-vazhlyvo/#:~:text=ЩО%20ТАКЕ%20РАННЄ%20ТЕСТУВАННЯ%3F,коду%20або%20на%20етапі%20проєктування.
7. 11 типів сучасних баз даних: короткий опис, схеми і приклади БД: https://senior.ua/articles/11-tipv-suchasnih-baz-danih-korotkiy-opis-shemi--prikladi-bd
8. Що таке Python?: https://acode.com.ua/intro-python/
9. SQLAlchemy Wikipedia: https://uk.wikipedia.org/wiki/SQLAlchemy
10. Ведення ділової документації ЗПО: https://naurok.com.ua/vedennya-dilovo-dokumentaci-zpo-149897.html
11. Martin Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002. 533 p.
12. Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2003. 560 p.

13. Robert C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008. 464 p.
14. Steve McConnell. Code Complete: A Practical Handbook of Software Construction. Microsoft Press, 2004. 960 p.
15. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1994. 395 p.
16. Ian Sommerville. Software Engineering 10th Edition. Pearson, 2015. 816 p.
17. Mike Cohn. User Stories Applied: For Agile Software Development. Addison-Wesley Professional, 2004. 304 p.
18. Scott Chacon, Ben Straub. Pro Git 2nd Edition. Apress, 2014. 456 p.
19. Andrew S. Tanenbaum, Maarten Van Steen. Distributed Systems: Principles and Paradigms 2nd Edition. Pearson, 2006. 608 p.
20. Rod Stephens. Beginning Software Engineering. Wrox, 2015. 480 p.
21. Bruce Eckel. Thinking in Java 4th Edition. Prentice Hall, 2006. 1150 p.
22. Sandi Metz. Practical Object-Oriented Design in Ruby: An Agile Primer. Addison-Wesley Professional, 2012. 272 p.
23. Kyle Simpson. You Don't Know JS: ES6 & Beyond. O'Reilly Media, 2015. 278 p.
24. Kent Beck. Extreme Programming Explained: Embrace Change 2nd Edition. Addison-Wesley Professional, 2004. 224 p.
25. Adam Freeman. Pro ASP.NET Core MVC 2 7th Edition. Apress, 2017. 1040 p.
26. Sam Newman. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, 2015. 280 p.
27. Gregor Hohpe, Bobby Woolf. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional, 2003. 736 p.
28. Roy Fielding. Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine, 2000. 162 p.

29. Joshua Bloch. Effective Java 3rd Edition. Addison-Wesley Professional, 2017. 416 p.
30. Jez Humble, David Farley. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Professional, 2010. 512 p.
31. Василь Мазур. "Програмування на мові Python: Практичний посібник". Київ: БІНОМ, 2018. 416 с.
32. Олег Бунина. "Python для дітей. Самовчитель по програмуванню". Дніпро: Буква, 2017. 352 с.

ДЕКЛАРАЦІЯ

про дотримання академічної доброчесності

Я, _____

_____

Повністю вказується ПІБ та статус (посада для працівників, освітня (освітньо-наукова) програма – для здобувачів вищої освіти)

що нижче підписалась/підписався, розуміючи та підтримуючи загальновизнані засади справедливості, доброчесності та законності,

ЗОБОВ'ЯЗУЮСЬ:

дотримуватися принципів та правил академічної доброчесності, що визначені законодавством України, локальними нормативними актами Донецького національного університету імені Василя Стуса, положеннями, правилами, умовами, визначеними іншими суб'єктами, та не допускати їх порушення.

ПІДТВЕРДЖУЮ:

що мені відомі положення статті 42 Закону України «Про освіту»; що у даній роботі не представляла/представляв чиїсь роботи повністю або частково як свої власні. Там, де я скористалася/скористався працею інших, я зробила/зробив відповідні посилання на джерела інформації; що дана робота не передавалась іншим особам і подається вперше, не порушує авторських та суміжних прав закріплених статтями 21-25 Закону України «Про авторське право та суміжні права», а дані та інформація не отримувались в недозволений спосіб.

УСВІДОМЛЮЮ:

що ця робота може бути перевірена університетом на плагіат або інші порушення академічної доброчесності, в тому числі з використанням спеціалізованих сервісів; що у разі порушення академічної доброчесності, до мене можуть бути застосовані процедури, передбачені законодавством України та Кодексом академічної доброчесності та корпоративної етики Донецького національного університету імені Василя Стуса, іншими локальними нормативними актами університету, та я можу бути притягнута/притягнутий до академічної відповідальності.

_____ _____

(дата) (підпис)