Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра обчислювальної техніки

Пояснювальна записка

до комплексної бакалаврської дипломної роботи на тему:

«Клієнт-серверна інформаційна система підрозділу навчального закладу з можливістю розгортання в хмарному середовищі. Частина 1. «Проектування розподіленої архітектури та менеджмент з використанням гнучких методологій»

Виконав: студент 4 курсу, групи 1КІ-18б
спеціальності 123 — Комп'ютерна інженерія
(шифр і назва напряму підготовки, спеціальності)

Станішевський Д.Ю.
(прізвище та ініціали)

Керівник к.т.н., доцент каф. ОТ

Войцеховська О.В.
(прізвище та ініціали)

«12» 06 2022 р.

Рецензент к.т.н., ст.викладач каф. ЗІ

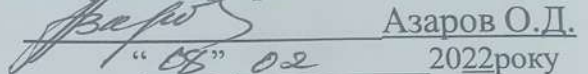
Лукічов В.В.
(прізвище та ініціали)

«13» 06 2022 р.

Допущено до захисту
Зав. кафедри ОТ Азаров О.Д.
«15» 06 2022 р.

Вінниця ВНТУ — 2022 рік

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки
Освітньо-кваліфікаційний рівень перший (бакалаврський)
Галузь знань — 12 — Інформаційні технології
(шифр і назва)
Спеціальність — 123— «Комп'ютерна інженерія»
(шифр і назва)
Освітньо — професійна програма — Комп'ютерна інженерія
(Назва освітньо — професійної програми)

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ, д.т.н, проф.

Азаров О.Д.
"08" 02 2022року

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Станішевський Дмитро Юрійович

1 Тема роботи — Клієнт-серверна інформаційна система підрозділу навчального закладу з можливістю розгортання в хмарному середовищі. Частина 1. «Проектування розподіленої архітектури та менеджмент з використанням гнучких методологій», керівник роботи Войцеховська О.В., затверджені наказом вищого навчального закладу від 24.03.2022 № 66

2 Строк подання студентом роботи 14.06.2022.

3 Вихідні дані до роботи: контент для наповнення веб-додатку інформаційної системи, вимоги до функціональності, швидкодії та безпеки.

4 Зміст текстової частини — аналіз сучасних підходів в розробці розподілених клієнт-серверних інформаційних систем з використанням веб технологій; обґрунтування вибору технологій та проектування інформаційної системи з урахуванням можливостей розширення та підтримки; реалізація високорівневої архітектури та технічний контроль її дотримання.

5 Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень): UML діаграма взаємодії редактора та адміністратора з авторизацією, структурна схема високорівневої архітектури, структурна схема архітектури потенційного розвитку додатку, структурна схема SQL бази даних для авторизації, UML діаграма структури SQL бази даних для авторизації.

6 Консультанти розділів роботи приведені в таблиці 1.

Таблиця 1 — Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-3	Войцеховська О. В., к.т.н., доцент каф. ОТ		

7 Дата видачі завдання 08.02.2022 р.

8 Календарний план виконання курсової роботи приведений в таблиці 2.

Таблиця 2 — Календарний план

№ з/	Назва етапів дипломного проекту (роботи)	Термін виконання		Примітка
		початок	закінчення	
1	Аналіз завдання	08.02.22		виконано
2	Аналіз тенденцій в розробці веб-додатків	09.02.22	24.02.22	виконано
3	Аналіз технологій для розробки веб-додатку	25.02.22	20.03.22	виконано
4	Розробка базової архітектури	21.03.22	05.04.22	виконано
5	Організація команди по сумісній роботі	06.04.22	10.04.22	виконано
6	Проектування та розробка авторизації	11.04.22	02.05.22	виконано
7	Проведення контролю над розробкою. Рефакторінг програмного коду.	03.05.22	25.05.22	виконано
8	Оформлення пояснювальної записки та ілюстративного матеріалу	26.05.22	10.06.22	виконано
9	Перевірка якості виконання бакалаврської роботи та усунення недоліків	13.06.22		виконано

Студент __________ Станішевський Д.Ю.

Керівник роботи __________ Войцеховська О.В.

АНОТАЦІЯ

Комплексна бакалаврська дипломна робота складається з 74 сторінок формату А4, на яких є 24 рисунків, перелік джерел посилань містить 15 найменувань.

В комплексній бакалаврській дипломні роботі проведено аналіз сучасних тенденцій, мов, підходів та технологій в розробці клієнт-серверних додатків з використанням веб-технологій.

Проведено порівняльний аналіз мов, та обрано найбільш відповідну. Також було обрано інші технології на основі аналізу та порівнянь. Спроектовано високорівневу архітектуру додатку. Створені умови для подальшого розвитку та розширення. Розроблено та реалізовано авторизацію згідно потреб додатку, що розробляється. Проведено рефакторінг з метою відповідності програмного коду до прийнятих стандартів, запобіганню проблем та забезпечення модульності проекту.

Ключові слова: розподілена архітектура, веб-додаток, модуль, інверсія залежностей, мови програмування, рефакторинг, багаторівнева високорівнева архітектура, багаторівнева архітектура.

ABSTRACT

The complex bachelor's thesis consists of 74 A4 pages, which contain 24 figures, the list of reference sources contains 15 titles.

In the complex bachelor's thesis analyzes current trends, languages, approaches and technologies in the development of client-server applications using web technologies.

A comparative analysis of languages was conducted, and the most appropriate one was chosen. Other technologies based on analysis and comparisons were also selected. The high-level architecture of the application is designed. Conditions are created for further development and expansion. Authorization developed and implemented according to the needs of the developed application. Refactoring was carried out in order to comply with the software code to the adopted standards, prevent problems and ensure the modularity of the project.

Keywords: distributed architecture, web application, module, dependency inversion, programming languages, refactoring, multilevel, high-level architecture, multilevel architecture.

ЗМІСТ

ВСТУП 8

1 АНАЛІЗ СУЧАСНИХ ПІДХОДІВ В РОЗРОБЦІ РОЗПОДІЛЕНИХ КЛІЄНТ-СЕРВЕРНИХ ІНФОРМАЦІЙНИХ СИСТЕМ З ВИКОРИСТАННЯМ ВЕБ ТЕХНОЛОГІЙ 11

1.1 Огляд домінуючих парадигм та підходів до розробки програмного забезпечення 12

1.2 Огляд сучасних методологій розробки 18

1.3 Аналіз хостингів та підходів до розгортання проекту 23

2 ОБГРУНТУВАННЯ ВИБОРУ ТЕХНОЛОГІЙ ТА ПРОЕКТУВАННЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ З УРАХУВАННЯМ МОЖЛИВОСТЕЙ РОЗШИРЕННЯ ТА ПІДТРИМКИ 25

2.1 Аналіз вимог та потенціалу розвитку інформаційної системи 25

2.2 Конкурентний аналіз мов програмування та технологій для розробки серверної сторони 25

2.3 Конкурентний аналіз мов програмування та технологій для розробки клієнтської сторони 30

2.4 Конкурентний аналіз баз даних 34

2.5 Стратегії та технології взаємодії між компонентами інформаційної системи........ 35

3 РЕАЛІЗАЦІЯ ВИСОКОРІВНЕВОЇ АРХІТЕКТУРИ ТА ТЕХНІЧНИЙ КОНТРОЛЬ ЇЇ ДОТРИМАННЯ 37

3.1 Розробка авторизації користувача 37

3.2 Рефакторінг API до стандарту REST 38

3.3 Перевірка якості та корегування програмного коду 41
3.4 Покращення якості, за рахунок застосування рішень підвищеної складності 50
3.5 Реалізація високорівневої архітектури та перспективи розвитку інформаційної системи 55
ВИСНОВКИ 58
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 60
ДОДАТОК А Технічне завдання 62
ДОДАТОК Б UML діаграма взаємодії редактора та адміністратора з авторизацією 66
ДОДАТОК В Структурна схема високорівневої архітектури 67
ДОДАТОК Г Структурна схема архітектури потенційного розвитку додатку 68
ДОДАТОК Д Структурна схема SQL бази даних для авторизації 69
ДОДАТОК Е UML діаграма структури SQL бази даних для авторизації 70
ДОДАТОК Ж Лістинг множинної реалізації інтерфейсу 71
ДОДАТОК К Протокол перевірки кваліфікаційної роботи на наявність текстових запозичень 74

ВСТУП

За останніх 20 років відбувся поступовий перехід від десктопних додатків до додатків на основі веб-технологій. Недоліками додатків для персональних комп’ютерів була складність з їх сумісністю з різними системами, вимоги до додаткового програмного забезпечення, та відсутності мобільності, як такою.

Інженери та науковці намагались розробити універсальні платформи які б вирішували ці проблеми, але спроби виявились невдалими. Рішення прийшло звідки не очікували — з веб технологій.

На початку свого існування веб виконував задачі статичних сторінок — заміни книгам та журналам. Та з появою JavaScript — з’явилась певна інтерактивність. Наступним поштовхом для розвитку стала поява AJAX, а пізніше цілих реактивних бібліотек та фреймворків. Це дозволило розробляти зручні, швидкі та багатофункціональні інтерфейси, які нічим не поступались десктопним. Також великою перевагою була пов’язаність на мережі Інтернет. Так, з часом потреба в колаборації різного роду корпоративних і особистих систем з сервером зростала. Нині не можна представити додаток, який реалізований лише на клієнтській стороні.

На даному етапі повноцінні додатки на основі веб-технологій є найбільшою нішею в IT сфері. Це породило велику кількість технологій, підходів, принципів та рішень в цій сфері. Також в цілому відбулось ускладнення проектів.

В наслідок цього зросла роль архітекторів та технічних лідерів, та й в цілому відбувався ріст вимог до знань в архітектурні програмного забезпечення серед розробників різного рівня.

Метою роботи є розробка клієнт-серверного веб-додатку офіційної сторінки кафедри обчислювальної техніки Вінницького національного технічного університету, яка дозволить швидко та продуктивно редагувати, видаляти та добавляти необхідну інформацію в клієнтський додаток і керувати правами доступу користувачів, розділяючи їх на гостей, викладачів та адміністраторів, які зможуть користуватись адміністративною панеллю.

Задачі дослідження бакалаврської роботи:

— проаналізувати популярні технології розробки клієнтських інтерфейсів;

— обрати оптимальний стек технологій для проектування розподіленої клієнт-серверної інформаційної системи;

— спроектувати архітектурне рішення;

— проводити контроль за дотриманням вимог та стандартів;

— особисто розробляти складні рішення;

— технічний контроль результатів роботи та рефакторінг;

— аналіз перспектив розвитку проекту.

Об'єкт дослідження — процес проектування та розробки розподіленої інформаційної системи структурного підрозділу навчального закладу з використанням веб технологій.

Предметом дослідження є технології, мови, принципи та підходи при розробці клієнт-серверної інформаційної системи з метою її адміністрування та подальшого розширення.

Апробація результатів бакалаврської роботи: зроблено доповіді на LI науково-технічній конференції підрозділів Вінницького національного технічного університету [1] та на Всеукраїнській науково-практичній інтернет-конференції «Молодь в науці: дослідження, проблеми, перспективи».

Публікації за темою роботи: Станішевський Д. Ю. Аналіз розвитку підходів до розробки веб-додатків та зростання впливу розподіленої архітектури / Д. Ю. Станішевський, О. В. Войцеховська // Тези доповіді. Матеріали LI наукової-технічної конференції підрозділів Вінницького національного технічного університету (Вінниця, 2022 р.) [Електронний ресурс]. Режим доступу: до ресурсу: https://conferences.vntu.edu.ua/index.php/all-fitki/all-fitki-2022/paper/view/15887/13334.

Станішевський Д. Ю. Оцінка ризиків технічного боргу та проведення рефакторінгу мовою програмування C# / Д. Ю. Станішевський, О. В. Войцеховська // Тези доповіді. Матеріали Всеукраїнської науково-практичної інтернет-конференції «Молодь в науці: дослідження, проблеми, перспективи", 2022 р. [Електронний ресурс]. Режим доступу: до ресурсу: https://conferences.vntu.edu.ua/index.php/mn/mn2022/paper/viewFile/16319/13728.

1 АНАЛІЗ СУЧАСНИХ ПІДХОДІВ В РОЗРОБЦІ РОЗПОДІЛЕННИХ КЛІЄНТ-СЕРВЕРНИХ ІНФОРМАЦІЙНИХ СИСТЕМ З ВИКОРИСТАННЯМ ВЕБ ТЕХНОЛОГІЙ

Ще десять років назад в сфері розробки серверних додатків панувала одна прийнятна архітектура. Існувало три компоненти: база даних, сервер та клієнтська частина.

База даних в більшості проектів використовувалась SQL — MSSQL для проектів, що розгортаються на операційних системах Windows Server, PostgreSQL для великих і навантажених проектів, Oracle для фінансового сектору та MySQL в усіх інших випадках. Тобто існувала універсальна схема вибору бази даних для проекту і окремого процесу вибору та розробки не було. Вся варіативність в цьому питанні збігалась до вибору де саме буде розгортатись ця база, на одному сервері чи на окремому відносно сервера.

Щодо сервера варіативність зберігалась на рівні вибору мови: існували мови для розробки сайтів (PHP, NodeJS) та для корпоративних систем (Java та C#). При цьому зазвичай вибір мови обмежувався наявністю відповідних спеціалістів або швидкості розробки. Технології API застосовувались не часто, а їхньою сферою застосування були мобільні додатки да програми для персонального комп'ютера. Основним способом взаємодії між клієнтською (фронтендом) та серверною (бекендом) частинами були так звані «шаблонізатори» — технології, що дозволяли фактично будувати веб-сторінки на стороні сервера та повертати їх у готовому вигляді. Це створювало жорсткі зв’язки між клієнтською та серверною частинами веб-додатка. Також це вимагало постійної взаємодії між розробниками різного стеку. Часто розробка взагалі виконувалась одним розробником. Як хостинг застосовувались або фізичні сервери або орендовані віртуальні машини на хостинг сервісах [1].

Клієнтська частина також мала типовий набір технологій, а саме JQuery для найпопулярніших задач та Bootstrap для адаптивної верстки.

Але за останнє десятиріччя в сфері веб-розробки відбулись докорінні зміни, які покладені побороти проблеми розробки.

1.1 Огляд домінуючих парадигм та підходів до розробки програмного забезпечення

Парадигма програмування — це система понять та підходів до розробки програмного забезпечення.

Існує велика кількість парадигм розробки, але не всі вони в рівній кількості використовуються у розробці. Зробимо огляд основних, які вплинули на основну частину сучасних мов програмування.

Імперативна парадигма — найперша парадигма, що з'явилась в часи перших обчислювальних машин. Більшість ЕОМ, починаючи від лампових першого покоління і до сьогодення мали стан (пам'ять, регістри, перемикання логічні вентилі, тощо) та програмувались за допомогою команд. Саме так, імперативна парадигма — це програмування за допомогою дій — команд, які міняють стан. Саме ця парадигма забезпечила появу термінів: «змінна», «присвоювання», «пам'ять», «вказівник». Певне всі сучасні мови програмування підтримують її в повному обсязі. Виключення можуть становити хіба що езотеричні і вузькоспеціалізовані мови. Прикладом мов, які підтримують виключно лише імперативну парадигму, без більш пізніх є мови асемблеру.

Структурна парадигма. Недоліком перших мов програмування є їхня громіздкість. Розробникам доводилось прописувати велику кількість команд для виконання елементарних дій. Над цією проблемою працювали Е. Дейкстра та Н. Вірт та сформували структурну парадигму. Суть цієї парадигми — створення структур, які інкапсулювали в собі найчастіше використовувані конструкції, такі як цикл, умовний оператор, оператор вибору. Парадигма також є дуже базовою для мов програмування і зараз підтримується більшістю популярних мов серед використовуваних, за виключенням асемблерних.

Процедурна парадигма. З часом проекти, що розроблялись, ускладнювались. З’явились проблеми з дублюванням великих блоків коду та з розумінням їх розробником в цілому. Процедурна парадигма вирішувала це питання додаючи нове поняття — підпрограма або процедура. Мови, що реалізують цю парадигму, повинні мати можливість поділу коду на окремі частини, які можуть викликатись з основного коду або іншої такої частини, та повертати керування назад. Процедура в різних мовах має різну назву, в залежності від парадигм (включаючи більш пізні), наприклад: методи, функції, макроси. Мови до появи процедурних парадигм не мали можливості передачі управління підпрограмам. Процедурна парадигма також має велику популярність, і представлена в більшості мов. Відмітимо, що асемблерні мови реалізують процедурну та імперативну парадигму, проте не мають структурної, яка історично знаходиться між ними.

Модульна парадигма. З розвитком технологій проблемою стало зростання складності проектів, а також - потреба перевикористання коду в різних проектах. Модульна парадигма є ідейним продовженням процедурної, та іноді взагалі не виділяється як окрема. Її суттю є створення «модулів» окремих блоків коду, які містять набір підпрограм, а також можливість імпорту (підключення) таких модулів до основної програми.

Розглянуті парадигми є базою всієї сучасної розробки, та їхня відсутність в тій чи іншій мові є скоріше виключенням чим тенденцією. Але їх недостатньо для розробки сучасної інформаційної системи, оскільки обмеження, що вони накладають можуть призвести до когнітивного ускладнення кодової бази. Тому в подальшому були розробленні додаткові парадигми, цілями яких було вирішення типових проблем та спроби структурувати кодову базу. В процесі еволюції ринок завоювало дві парадигми — об’єктно орієнтована та функціональна парадигми. Розглянемо їх детально.

Об’єктно-орієнтована. Є ідейним продовженням модульної парадигми. Вводить поняття об’єкту — певної логічної сутності, яка містить інформацію про себе (поля), та може виконувати певні дії (методи). Принципи об’єктно-

орієнтованого програмування (ООП) були запропоновані одним з його засновників Аланом Кеєм:

— все є об'єктами;

— всі дії та розрахунки виконуються шляхом взаємодії (обміну даними) між об'єктами, під час якої один об'єкт потребує, щоб інший об'єкт виконав деяку дію, при цьому об'єкти взаємодіють, надсилаючи й отримуючи повідомлення — запит на виконання дії, доповнений набором аргументів, які можуть знадобитися під час виконання дії;

— кожен об'єкт має незалежну пам'ять, яка складається з інших об'єктів;

— кожен об'єкт є представником (екземпляром) класу, який виражає загальні властивості об'єктів;

— у класі задається поведінка (функціональність) об'єкта, таким чином усі об'єкти, які є екземплярами одного класу, можуть виконувати одні й ті ж самі дії;

— класи організовані у єдину деревоподібну структуру з загальним корінням, яка називається ієрархією успадкування, а пам'ять та поведінка, зв'язані з екземплярами деякого класу, автоматично доступні будь-якому класу, розташованому нижче в ієрархічному дереві.

На основі цього було сформовано шість концепцій ООП:

— інкапсуляція — обмеження доступу одних компонентів програми до інших, а також приховування реалізацій методів від компонентів, що їх викликають;

— успадкування — можливість отримування класами властивостей інших класів за допомогою механізму наслідування;

— поліморфізм — має декілька проявів, а саме, можливість створювати декілька функцій з однаковою назвою але різною сигнатурою, можливість написання коду, який може працювати з різними типами за певним обмеженням

(узагальнення як популярна реалізація концепції), та поліморфізм підтипів — можливість використовувати похідний клас замість базового;

— абстракція — допоміжна концепція, суть якої полягає в тому, що при проектуванні класів необхідно враховувати та реалізовувати лише суттєву для програми функціональність;

— перевикористання коду — також допоміжна концепція, оскільки є очевидним наслідком попередніх;

— відправка повідомлень — повністю архаїчна концепція, реалізацію якої майже не можливо зустріти в сучасних ООП мовах, а її суть полягає в тому, що об’єкти взаємодіють між собою за допомогою відправлення спеціальних викликів та повідомлень, від концепції відмовились через її незручність.

Ця парадигма на сьогодні є найпопулярнішою. До мов, які її реалізують більшою мірою відносяться Java, C#, C++, Python, Ruby, тощо. Ряд мов реалізують її лише частково: JavaScript, F#, PHP, Lua.

Функціональна парадигма. На відміну від попередніх — є не продовженням, а переосмисленням попереднього процесу формування парадигм. Заперечує імперативну парадигму, тобто в цій парадигмі не існує станів та команд. Вся програма формується як вираз схожий на математичний, тобто є вхідні дані, є результат, та є функція перетворення даних [2].

До понять функціонального програмування відносять чисті функції.

Чистими називають функції, які не мають побічних ефектів введення-виведення і пам'яті (вони залежать тільки від своїх параметрів і повертають тільки свій результат). Чисті функції володіють декількома корисними властивостями, багато з яких можна використовувати для оптимізації коду:

— якщо результат чистої функції не використовується, він може бути видалений без шкоди для інших виразів;

— результат виклику чистої функції може бути кешований, тобто збережений в таблиці значень разом з аргументами виклику, при цьому якщо надалі функція викликається з цими ж аргументами, її результат може бути взятий

прямо з таблиці без її повторного обчислення (принцип прозорості посилань), а кешування, ціною невеликої затрати пам'яті, дозволяє істотно збільшити швидкодію і зменшити глибину рекурсії в деяких алгоритмах;

— якщо немає ніякої залежності серед даних між двома чистими функціями, то порядок їх обчислення можна змінити (тобто обчислення чистих функцій задовольняє принципам thread-safe);

— якщо вся мова не допускає побічних ефектів, то можна використовувати будь-яку політику обчислення, що надає свободу компілятору комбінувати і реорганізовувати обчислення виразів у програмі (наприклад, виключити деревоподібні структури).

Хоча більшість компіляторів імперативних мов програмування розпізнають чисті функції і видаляють загальні підвирази для викликів чистих функцій, вони не можуть робити це завжди для попередньо скомпільованих бібліотек, які, як правило, не надають цю інформацію. Деякі компілятори, такі як gcc, з метою оптимізації надають програмісту ключові слова для позначення чистих функцій. Fortran 95 дозволяє позначати функції як «pure».

Іншим поняттям є функціональні структури.

Чисто функціональні структури даних часто представлені інакше, ніж їх процедурні аналоги. Наприклад, масив з постійним доступом і часом оновлення є основним компонентом більшості імперативних мов і багато імперативних структур даних, таких як геш-таблиця і двійкова купа, засновані на масивах. Масиви можна замінити асоціативними масивами або списками випадкового доступу, які допускають суто функціональну реалізацію, але мають логарифмічний доступ і час оновлення. Чисто функціональні структури даних мають стійкість, тобто властивість зберігати попередні версії структури даних незмінними. У Clojure постійні структури даних використовуються як функціональні альтернативи їхнім імперативним аналогам.

Зараз функціональна парадигма хоч і є однією з двох популярних парадигм, проте сильно поступається ООП. Чисто функціональними мовами зараз є тільки

вузькоспеціалізовані мови, наприклад — APL. Більшість мов або реалізують цю парадигму частково JavaScript, F#, Scala, або використовують як допоміжну — C#, Java, Python, і більшість популярних ООП мов.

Отже, типовою сучасною мовою є ООП мова з елементами функціональної, або така, яка містить часткові реалізації обох парадигм. Хоча в цілому можна виділити тенденції до зростання популярності функціональної парадигми.

Стандартом для індустрії стали принципи SOLID — п'ять принципів, сформованих Робертом Мартіном. Усі розробники на об'єктно-орієнтованих мовах намагаються їх дотримуватись. Також відбувається полеміка щодо часткової реалізації принципів в функціональних мовах.

— принцип єдиного обов'язку — кожен клас або модуль повинен виконувати лише один обов'язок і повинна існувати лише одна причина для змінювання коду;

— принцип відкритості/закритості — сутності повинні бути розроблені таким чином, щоб у разі виникнення потреби в їх зміні, вона відбувалась за рахунок розширення, а не змін;

— принцип підстановки Лісков — будь-який нащадок може використовуватись замість батьківського класу без зміни існуючої функціональності, тобто дочірні класи повинні робити не менше ніж батьківські;

— принцип розподіленого інтерфейсу — краще декілька спеціалізованих інтерфейсів замість одного загального, оскільки це ускладнює їх реалізацію та може призводити до прив'язування непотрібного функціональності;

— принцип інверсії залежностей — класи повинні залежати від абстракцій а не конкретних реалізацій, модулі вищих рівнів не повинні залежати від модулів нижчих рівнів.

Іншим популярних підходів є KISS (Keep Is Simlpe), який полягає в спрощенні реалізації. Є важливим, оскільки захищає від так званого «оверінжиніринга» та непотрібного ускладнення системи.

Також є принцип DRY, або «Don’t repeat your self». Цей принцип вимагає від розробника не повторювати за собою. Багато розробників вважають це правило абсолютним і намагаються позбутись будь-якого повтору коду в проекті. Насправді повтор коду не є проблемою, якщо він випадковий, а не закономірний. Допустимо у нас є дві сутності з однаковою групою полів. Потрібно задати ряд запитань, а саме: «чи буде в подальшому їхній розвиток сумісним», «чи буде перевага в такому об’єднанні», «чи не ускладнить це зайвий раз систему». Якщо ці дві сутності справді пов’язанні, а таке об’єднання дасть переваги — це треба зробити. В іншому випадку це порушення принципу KISS для реалізації принципу DRY. Класичним прикладом такого порушення, є використання тих самих сутностей на різних рівнях проекту замість створення окремих на кожний.

1.2 Огляд сучасних методологій розробки

Існують різні підходи до організації процесу розробки програмного забезпечення. Розглянемо основні з них:

Модель водоспаду (waterfall) — це основна модель життєвого циклу, розроблена Вінстоном Ройсом у 1970 році. Ця модель представляє безліч етапів або процесів послідовно, що прогресує поступово вниз.

Цей підхід корисний, коли вимоги добре відомі, технології зрозумілі та ресурси, що мають необхідний досвід, є.

Модель водоспаду подано на рисунку 1.1.

В фазі збору та аналізу вимог потрібно зафіксувати та проаналізувати всі вимоги та переконатися, перевіряються вони чи ні.

В фазі дизайну системи потрібно створювати та оформляти документи на основі аналізу вимог. Визначте вимоги до обладнання та програмного забезпечення.

В фазі реалізації розробляють надійний код для компонентів відповідно до конструкції та інтегруйте їх.

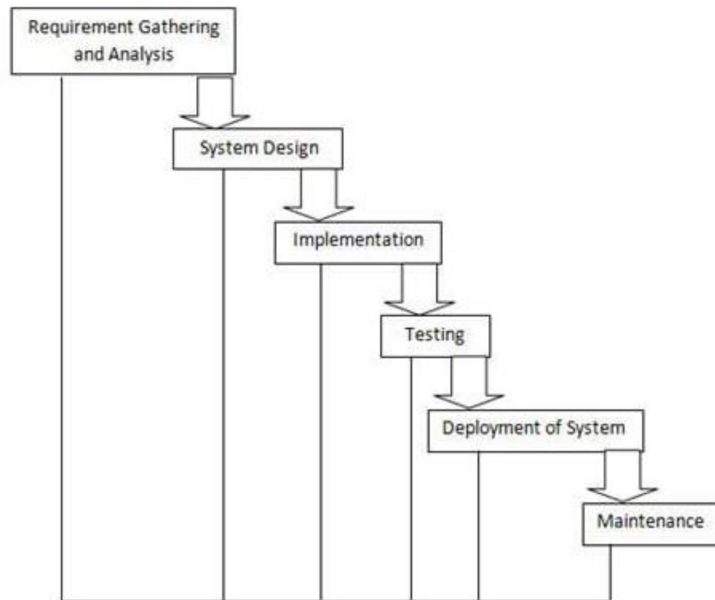


Рисунок 1.1 — Модель життєвого циклу waterfall

В фазі тестування системи, інтегровані компоненти утворюють цілу систему, цей етап виконується для того, щоб переконатися, чи працює система відповідно до вимог, відстежуючи та звітуючи про хід тестування.

В фазі розгортання системи переконуються, що система стабільна з нульовими помилками, усі критерії тесту були дотриманими.

В фазі використання, пересвідчуються що додаток працює ефективно відповідно до вимог із відповідним середовищем. У разі виявлення дефекту, його слід виправити та розгорнути (оновити) у середовищі.

Переваги моделі водоспаду:

— простота і зрозумілість;

— легко керувати, оскільки кожна фаза має свої конкретні результати;

— уникає перекриття етапів;

— відмінний для невеликих проектів.

Недоліки моделі водоспаду:

— збільшення ризику та невизначеності;

— розпочавши фазу тестування, ви не можете нічого змінити на попередніх етапах;

— не підходить для складних і великих проектів;

— не підходить там, де вимоги постійно змінюються.

Гнучка модель (Agile) показує ітераційний та інкрементальний підхід. Цей підхід розбиває продукт на невеликі додаткові одиниці для забезпечення ітерацій. Потім кожна ітерація включає такі етапи, як планування, аналіз вимог, проектування, кодування, тестування одиниць, перевірка прийнятності тощо (рисунок 1.2).

Цей підхід також дозволяє постійно взаємодіяти з клієнтом для його зворотного зв'язку та виправлення вимог через рівні проміжки часу.

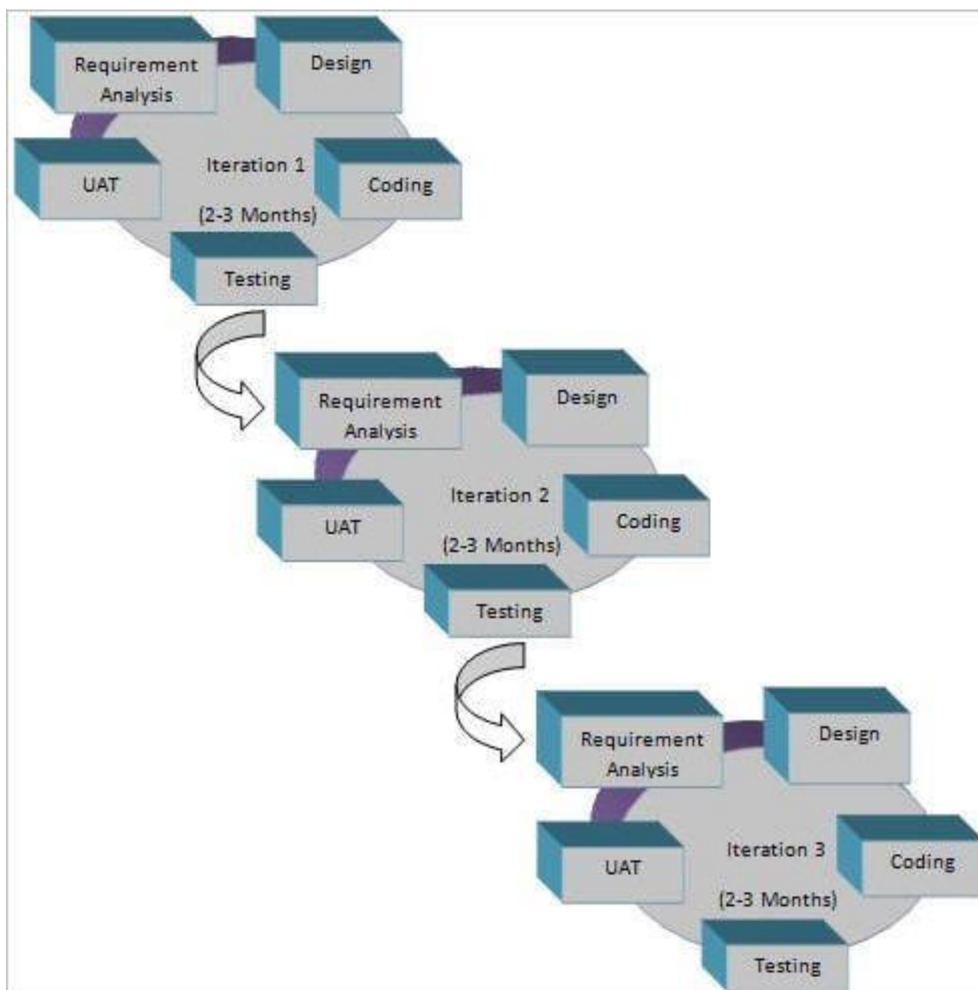


Рисунок 1.2 — Модель життєвого циклу гнучкої методології

Переваги моделі Agile:

— реалістичний підхід до розробки програмного забезпечення;

— сприяє роботі в команді;

— усуває невідповідність між вимогами та тестовими кейсами;

— швидкий і вимагає мінімальної кількості ресурсів;

— підходить для великих та довгострокових проектів;

— підходить для вимог що можуть змінитись;

— легкий у використанні.

Недоліки Agile моделі:

— не підходить для складних проектів;

— потрібна велика кількість взаємодії з клієнтом, що може спричинити затримку;

— неправильне введення вимог може спричинити неправильну розробку програмного продукту;

— підвищений ризик виникнення помилок;

— передача іншій команді може бути досить складною.

Спіральна модель включає ітеративний підхід до розвитку разом із систематичним підходом моделі водоспаду. Це схоже на додаткову модель та акцент на аналізі ризиків.

Спіральна модель має чотири етапи:

— етап планування;

— аналіз ризиків;

— інженерна фаза;

— етап оцінювання.

На етапі планування збираються та переглядаються вимоги для завершення тестування.

На етапі аналізу відбувається виявлення, моніторинг та оцінку ризиків управління. Вимоги аналізуються для виявлення ризиків за допомогою таких методів, як мозковий штурм, тощо.

На інженерному етапі програмне забезпечення розробляється та перевіряється наприкінці.

На етапі оцінювання, замовник оцінює результати проекту та надає свої відгуки для наступної спіралі або затвердження.

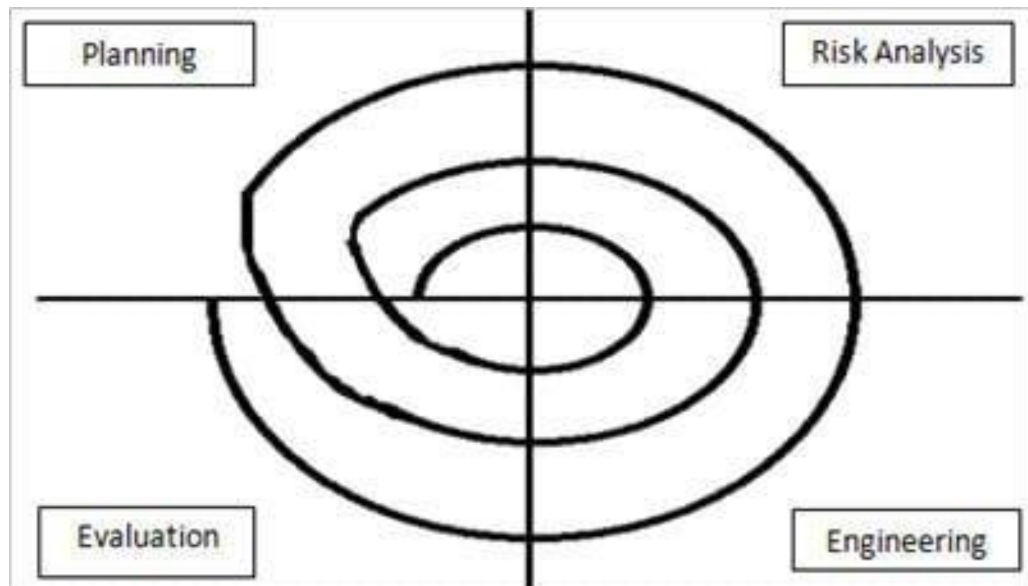


Рисунок 1.3 — Модель життєвого циклу спіральної моделі

Коли використовувати спіральну модель:

— для проектів з високим ризиком;

— коли вимоги складні;

— якщо проект великий;

— достатньо часу для отримання відгуку користувача про наступну ітерацію;

— потрібні суттєві зміни внаслідок проведення досліджень та аналізу;

— користувачі не впевнені у своїх потребах.

Переваги спіральної моделі:

— уникнення ризику, оскільки воно включає велику кількість аналізу ризиків;

— швидкий розвиток;

— змінии у вимогах легко усуваються;

— вимоги можна отримати більш точно.

Недоліки спіральної моделі:

— комплексне управління;

— не підходить для невеликих проектів;

— дорогий в запровадженні;

— потрібен великий обсяг аналізу ризиків та досвіду для успіху їх проекту.

1.3 Аналіз хостингів та підходів до розгортання проекту

Існує три підходи хостингу проекту. Вони з'являлись в хронологічному порядку та поступово витісняли один одного.

Історично першою з'явився On-Premise підхід, коли сервер, на якому все розгортається безпосередньо фізично знаходиться у офісі компанії або іншому приміщенні.

До переваг такого підходу відноситься повний контроль над сервером, його безпекою і т.п. Недоліків значно більше, оскільки такий сервер треба обслуговувати, забезпечувати комунікації, слідкувати за вологістю повітря та струмом.

Зараз такий підхід використовується лише в сфері інформаційної безпеки, для збереження чутливої інформації. Сферами, де найбільш розповсюджений такий підхід, є банки, державний та військовий сектор.

Наступний, та донедавна найпопулярніший підхід — віддалений сервер. Це віртуальна машина в дата центрі, яка орендується у компанії-провайдера.

З переваг — відсутність фізичного обслуговування, швидке придбання, можливість конфігурування. З недоліків — гірша безпека та потреба в самостійному розгортанні.

І найновіший підхід — хмарний хостинг. Це розгортання за допомогою спеціальних технологій проекту в хмарних сервісах. З переваг — зручність, автоматичність, велика кількість готових рішень, тощо. З недоліків — ціна та необхідність мати певну підготовку. Наведемо таблицю порівнянь цих підходів

Характеристики розглянутих хостінгів зведено в таблицю 1.1

Таблиця 1.1 — Характеристики підходів хостінгу

Характеристика	On-Premise	Remote host	Cloud
Складність обслуговування	Висока	Середня	Низька
Безпека	Висока	Середня	Середня
Час впровадження	Високий	Середня	Низька
Вартість	Висока	Низька	Середня
Потреба в окремих спеціалістах	Висока	Низька	Середня

Отже, було проаналізовані парадигми, що є домінуючими у сучасній розробці. Головною є об’єктно орієнтована парадигма. Розглянуто декілька основних методологій розробки, зокрема гнучку, яка є домінуючою у сучасних реаліях. Також розглянуті та порівняні три хостинги.

2 ОБГРУНТУВАННЯ ВИБОРУ ТЕХНОЛОГІЙ ТА ПРОЕКТУВАННЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ З УРАХУВАННЯМ МОЖЛИВОСТЕЙ РОЗШИРЕННЯ ТА ПІДТРИМКИ

2.1 Аналіз вимог та потенціалу розвитку інформаційної системи

Перед командою постала ціль розробити розподілений веб-додаток для підрозділу навчального закладу. Окрім наявності веб сайту необхідна можливість додавати та редагувати контент. Також є вимоги, щодо базової кібербезпеки. Проект повинен розгортатись як на віддаленому сервері так і в майбутньому з використанням хмарних технологій. В першій версії необхідна можливість динамічно заповнювати окремі сторінки, зокрема новини, та редагувати інформацію про викладачів.

В подальшому необхідна можливість додавати та редагувати інформацію про аспірантів, наукову літературу, видавничу діяльність, можливість розміщувати файли в спільний доступ. В подальшому — можливість розробки окремого кабінету для студентів та абітурієнтів.

Також необхідно врахувати можливість масштабування по горизонталі та використання інших технологій для покращення системи.

2.2 Конкурентний аналіз мов програмування та технологій для розробки серверної сторони

Мова програмування, на якій буде розроблятись клієнт-серверна інформаційна система для структурного підрозділу навчального закладу, повинна відповідати декільком критеріям.

По-перше, ця мова повинна підходити для розробки серверної частини, оскільки не всі мови мають потрібні бібліотеки або технічні можливості.

По-друге, мова має мати достатню розвинену документацію, спільноту та підтримку.

По-третє, ця мова повинна мати досить розвинену спільноту фахівців які вміють працювати з нею, оскільки необхідна подальша розробка та підтримка.

Отже ми маємо три критерія — актуальність, підтримка та популярність. Два з цих критерії, а саме підтримка та популярність можна оцінити по використанню в цілому в індустрії. Для цього звернемось до статистики ресурсу DOU (рисунок 2.1)

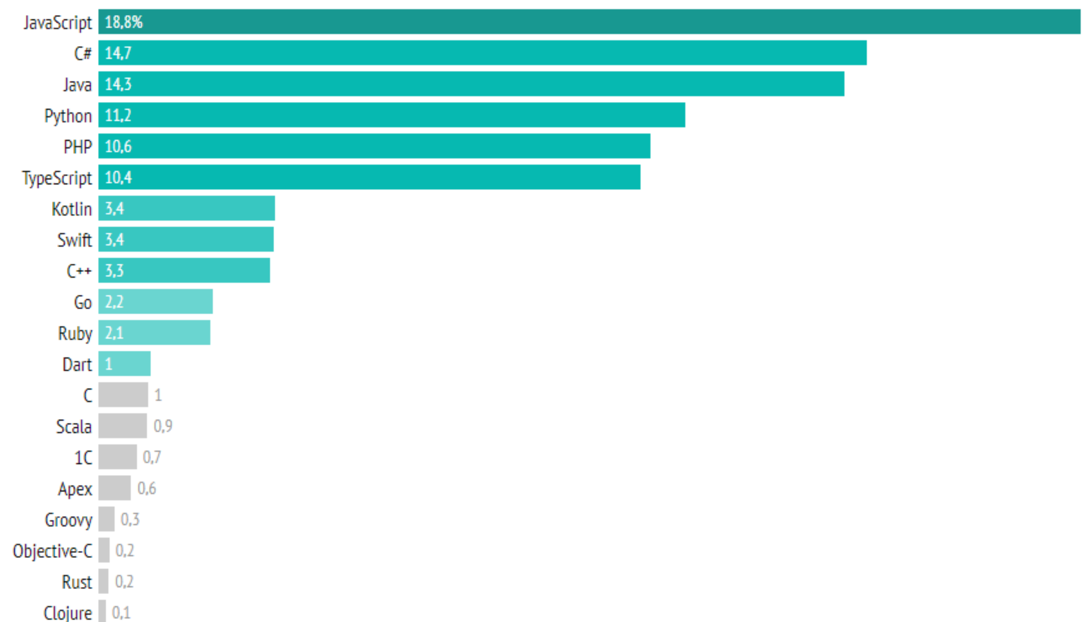


Рисунок 2.1 — Статистика використання мов програмування

Розглянемо мови з відсотком використання понад 1%. До них відноситься.

JavaScript, C#, Java, Python, PHP, TypeScript, Kotlin, Swift, C++, Go, Ruby, Dart. Мови C++ та Swift виключимо з цього списку оскільки вони не використовуються для розробки серверної частини. А мови JavaScript, TypeScript та Dart об’єднаємо під єдиною назвою NodeJs, оскільки всі вони є похідними від JavaScript та використовують ті ж самі технології, різниця між різними діалектами для нас несуттєва.

В подальшому виборі необхідно виставити оцінку за декількома критеріями, а саме:

— популярність — наскільки поширена мова;

— підтримка — розвиненість спільноти, наявність документації;

— легкість освоєння — наскільки швидко команда зможе розібратись в технології;

— продуктивність — швидкодія додатку написаного з застосування мови;

— безпечність — наявність захистів від популярних хакерських атак;

— суб'єктивна оцінка команди — оскільки проектування відбувається з вже існуючою командою, а не відбувається набір команди вже під проект, то необхідно враховувати їх думку;

— актуальність — наскільки технологіях актуальна для проекту такого типу.

NodeJS. Спільна технологія для JavaScript та його похідних діалектів Dart та TypeScript. Хоча JS розроблявся саме для розробки клієнтської частини — веб-сайтів. Про те був створений NodeJS для розробки серверів з використанням клієнтських технологій. Ця технологія спеціалізується для серверів звичайних не високонавантажений веб-сайтів. Серед переваг — швидкість розробки та розгортання, спеціалізація на взаємодії з клієнтськими JS (наприклад унікальна технологія рендеру реактивних фреймворків на стороні сервера — SSR), та можливість швидкого входження фронтенд розробника в бекенд, а також наявність асинхроності. До недоліків можна віднести відсутність багатопоточності, проблеми з продуктивністю, а також одвічна проблема з втратою пам'яті при довготривалій роботі.

В складних корпоративних системах також може використовуватись як проміжній рівень між клієнтською та серверною частиною, так званий Web-Specified Middleware.

Java. Мова яка створювалась для можливості запуску коду на будь-якій платформі за допомогою Java Virtual Machine (JVM). Найбільшої популярності

набула для мобільної розробки та корпоративної серверної. Також був період коли активно застосовувалась і в фронтенд частині, в так званих Java-аплетах.

Протягом довгого часу вважається лідером в бекенд-розробці корпоративних систем. Являє собою типову С-подібну ООП мову з незначною підтримкою функціональної парадигми. Перевагою є запуск під JVM, тобто повна кросплатформенність.

До недоліків можна віднести проблеми з продуктивністю, «важкість» проектів та наявність великої кількості архаїчних застарілих рішень. Існують спроби повернути до життя Java створенням нових мов на її основі. Наприклад Kotlin, який також розглядається в цьому порівнянні, що являє собою спрощення та усучаснення Java. А також Scala — функціональне розширення для Java, яке є дуже складною мовою.

C#. Постійний конкурент Java від Microsoft зі схожими можливостями та синтаксисом. Доволі довгий час не міг догнати Java в популярності, оскільки не мав кросплатформенності та був жорстко прив'язаний до платформи Windows. Хоча на відміну від Java отримував нові функціональні можливості раніше і якісніше. Також має кращу підтримку функціональної парадигми. З незвичних та ефективних переваг — можливість інтегруватись з проектами написаними іншими мовами .Net, до таких відносяться CLR-сумісні діалекти VisualBasic та C++, та повністю функціональний F#.

В 2016 з'явилась кросплатформенна платформа для .Net - .Net Core, яку пізніше перейменували в .Net. Окрім довгоочікуваної підтримки інших операційних систем, також була проведена велика робота з ефективністю та безпекою.

Одразу після виходу була знайдена суттєва кількість проблем та помилок, більшою частиною яких були виправленні в наступній версії 2.0.

В третій версії були виправленні більшість відомих проблем, та значно покращена швидкодія. Саме ця версія є найбільш популярною, серед існуючих проектів. Це призвело до відмови від паралельної платформи під Windows - .Net

Framework. Одразу після 3 версії вийшла 5 версії з назвою .Net, для усунення плутанини між різними платформами.

Python. Мова програмування з підтримкою і ООП і функціональної парадигми. На відміну від попередніх мов має не C-подібний синтаксис. Хоча має можливість розробки серверних додатків за допомогою фреймворка Django, про те спеціалізується на створенні систем штучного інтелекту та обробки великих об’ємів даних.

PHP. Історичний лідер в розробці серверної частини веб-сайтів. Був довгий час найпопулярнішою мовою для веб-розробника. Більшість сайтів в мережі Інтернет розробленні за допомогою нього. Проте мова є застарілою, з Perl-подібним синтаксисом. Має досить заплутану колекцію бібліотек та фреймворків, а також величезну проблему з безпекою. Були неодноразові спроби вирішення більшості проблем, більшість яких вдалось вирішити, але мова досі вважається застарілою, і розробка нових проектів з її використанням є малоймовірною.

Go. Спроба замінити Java та C# в корпоративному сегменті. Також мультипарадигменна мова з C-подібним синтаксисом. Великої популярності та динаміки її росту — немає.

Ruby. Духовний наслідник PHP, створений суто для розробки серверної частини. Також не набрав великої популярності. Використовується ентузіастами або як нова мова для розробників на PHP.

Зробимо коротку характеристику та проаналізуємо обрані мови, а потім проведемо оцінювання в зведеній таблиці 2.1.

В результаті оцінювання в лідери вийшов C#. Хоча він не спеціалізується саме на сайтах, а цьому сенсі краще підійшов би NodeJS або PHP, але він виявився найоптимальнішим по сукупності оцінок.

Таблиця 2.1 Порівняння різних мов програмування за обраними критеріями

Мова	Популярність	Підтримка	Легкість	Продуктивність	Безпека	Команда	Актуальність	Загальна оцінка
NodeJS	4	4	4.5	3	3	4	5	27.5
C#	4.5	5	4	5	5	5	4	32.5
Java	5	4	4	4	5	4	4	30
Python	4	4.5	3.5	4	4	3.5	2	25.5
PHP	4.5	4.5	4	3	2	1	4.5	23.5
Kotlin	1	3	3.5	4	5	3	4	23.5
Go	3	3	3	5	5	2	3	24
Ruby	3	3	3	3.5	4	2	5	23.5

2.3 Конкурентний аналіз мов програмування та технологій для розробки клієнтської сторони.

В сфері розробки домінує JavaScript. Ця мова єдина з якою може працювати сучасний веб-браузер, тому фронтенд-розробники жорстко прив’язанні до її використання. Ця мова має ряд недоліків, таких як нелогічний синтаксис та поведінка, проблема з оптимізацією, бідний функціонал. Хоча була спроба вирішення частини в оновленнях мови зокрема в стандарті ES5, розробникам потрібно була більша кількість можливостей.

Саме тому були створенні мови програмування з можливістю компіляції (перетворення) коду в JavaScript, до таких мов відносяться Dart та TypeScript, від Google та Microsoft відповідно. Ці мови додають повноцінну типізацію та реалізацію ООП.

Іншою мовою є Blazor, технологія що дозволяє розробляти фронтенд частину на C# з подальшим перетворенням в JS, але це досить нова технологія, і є досить мало проектів з її використанням.

Це не усі складнощі з фронтенд розробкою, оскільки базових можливостей не вистачає для повноцінної розробки. По перше такий код буде досить великий за об'ємом.

Також проблемою є підвантаження даних на вже завантажену сторінку. Для цього слугує технологія AJAX, яка дозволяє відправляти асинхронні запити до сервера без перезавантаження сторінки. Базовий JS вимагає прописувати весь процес запиту до сервера, а потім перемальовувати сторінки. Оскільки це досить часта потреба при розробці - були створенні цілі фреймоврки для роботи з AJAX, їх називають — реактивними. Найпопулярнішими є Angular, React, Vue.

Розглянемо переваги та недоліки кожної з них.

Angular. Це фреймворк для розробки фронтенд частини від розробників C#. Розроблявся одночасно з TypeScript. Реалізує підходи розробки схожі з розробкою бекенду на C# та ASP.NET.

Переваги Angular:

— Angular використовується разом із Typescript, оскільки вони розроблялись одночасно;

— angular-language-service — забезпечує інтелектуальні можливості та автозаповнення шаблону HTML-компонента;

— нові функції, такі як generation Angular, які використовують бібліотеки npm із CLI, generation та розробку компонентів, що користуються Angular;

— детальна документація, яка дозволяє отримати всю необхідну інформацію, однак вимагає багато часу на вивчення;

— одностороння прив'язка даних, яка забезпечує виключно поведінку додатків, що зводить до мінімуму ризик можливих помилок;

— MVVM (Model-View-ViewModel), яка дозволяє розробникам працювати над одним компонентом паралельно;

— DI контейнер для компонентів, пов'язаних з модулями і модульністю в цілому;

— структура та архітектура, спеціально створені для великої масштабності проекту.

Недоліки Angular:

— різноманіття різних структур (Injectables, Components, Pipes, Modules і т. п.) ускладнюють вивчення в порівнянні з React і Vue.js, у яких є тільки «Component»;

— відносно погана продуктивність, враховуються показники.

React. Бібілотека від Facebook розроблена в 2013 році. Дозволяє розробляти веб-сайти різної складності.

Переваги React:

— легко вивчити, завдяки простому дизайну, використанню JSX (HTML-подібний синтаксис) для шаблонів і дуже детальної документації;

— дуже швидкий, завдяки реалізації React Virtual DOM та різноманітним оптимізаціям рендеринга;

— відмінна підтримка рендеринга на сторонньому сервері, що робить його потужною платформою для контент-орієнтованих додатків;

— першокласна підтримка Progressive Web App (PWA) завдяки генератору доданого create-react-app;

— одностороння привязка, що означає менше небажаних побічних ефектів;

— Redux, найбільш популярна платформа для управління станом додатків у React, її легко вчити та використовувати;

— React реалізує концепцію функціонального програмування (FP);

— додатки можуть бути створені за допомогою TypeScript або Facebook's Flow, що мають вбудовану підтримку JSX;

— перехід між версіями, як правило, дуже простий: Facebook надає «кодові модулі» для автоматизації більшої частини процесу;

— навички, отримані в React, можуть бути корисні для розробки на React Native.

Недоліки React:

— React не однозначний і залишає розробникам можливість вибирати найкращий спосіб розробки;

— спільнота ділиться за способами написання CSS в React, які розділяються на традиційні таблиці стилів (CSS Modules) і CSS-in-JS (т.е. Emotion і Styled Components);

— React відмовляється від компонентів на основі класів, що може стати проблемою для розробників, яким більш комфортно працювати з об'єктно-орієнтованим програмуванням (ООП);

— змішування шаблонів з логікою (JSX) не зрозуміле для деяких розробників при перших знайомствах з React;

VueJS. Легкий та швидкий фрейворк для веб-сайтів та веб-додатків. Підходить для невеликих проектів.

Переваги Vue.js:

— посилений HTML, це означає, що Vue.js має багато параметрів подібних до Angular, а це, завдяки використанню різних компонентів, допомагає оптимізації HTML-блоків;

— детальна документація, що означає що Vue.js має дуже докладну документацію, яка може прискорити процес навчання для розробників та заощадити багато часу на розробку програми, використовуючи лише базові знання HTML та JavaScrip;

— адаптивність, може бути здійснений швидкий перехід від інших фреймворків до Vue.js через схожість з Angular та React з точки зору дизайну та архітектури;

— чудова інтеграція, Vue.js можна використовувати як для створення односторінкових програм, так і для більш складних веб-інтерфейсів програм, при цьому, що невеликі інтерактивні елементи можна легко інтегрувати до існуючої інфраструктури без негативних наслідків;

— масштабування, Vue.js може допомогти в розробці досить великих шаблонів багаторазового використання, які можуть бути зроблені майже за той самий час, що і простіші;

— крихітний розмір, Vue.js займає близько 20 КБ, зберігаючи при цьому свою швидкість і гнучкість, що дозволяє досягти кращої продуктивності в порівнянні з іншими платформами.

Недоліки Vue.js:

— нестача ресурсів, Vue.js, як і раніше, займає досить невелику частку ринку в порівнянні з React або Angular, що означає, що обмін знаннями в цьому середовищі все ще знаходиться на початковій стадії.

— ризик надмірної гнучкості, іноді можуть виникнути проблеми при інтеграції у величезні проекти, і поки що немає досвіду можливих рішень.

Чітких параметрів, за яким можна було б обрати фрейморк немає. Тому для вирішення цього питання було проведено обговорення з командою.

Одразу відмовились від Angular, оскільки він важкий та повільний і буде надлишковим. З точки зору проекту, краще підходить VueJs, але враховуючи розширення проекту в майбутньому та досвід команди — зупинились на React.

2.4 Конкурентний аналіз баз даних

Існує дві основних групи типів баз даних — SQL, та NoSQL. Перші це реляційні бази даних на основі жорстких таблиць та зв’язків між ними. Інші — це

загальна назва усіх інших баз даних, які не відповідають стандартам SQL.

Проведемо аналіз чи підходять нам реляційні бази.

Усі сутності мають постійну структуру, відсутня мінливість. Між ними є зв'язки. Є необхідність в швидкому пошуку по полях (застосування індексів). Також для нас перевагою буде застосування ORM підходу. Це технологія, яка дозволяє передати створення бази даних спеціальній бібліотеці. Це спростить процес розробки, оскільки команді не потрібно самостійно розробляти базу даних. Також грамотний підхід дозволить локалізувати залежність від бази даних та швидко змінити технологію. В майбутньому не виключено застосуванні інших баз даних. Наприклад для кешування, балансування навантаження, збереження секретів, тощо.

2.5 Стратегії та технології взаємодії між компонентами інформаційної системи

Взаємодія між різними компонентами інформаційної системи може відбуватись за двома стратегіями: push, або синхронна, та pull або асинхронна.

Перша базується на безпосередньому виклику іншого компонента. Це схоже на телефонний дзвінок: коли ви телефонуєте до когось з запитанням, то та людина залишає всі справи, щоб відповісти вам і залишається зайнятою до кінця розмови. Зазвичай цей підхід застосовується при потребі в негайні відповіді — виконанні команди, отримання даних та інше. Подібна взаємодія зазвичай базується на мережевих протоколах. Найчастіше це HTTP запити, але можливе використання сокетів TCP/UDP або telnet.

Pull стратегія діаметрально інша до неї. Компонент, що звертається до іншого не викликає інший тут і зараз, а розміщає запит на обробку. Це схоже швидше на повідомлення в месенджері: повідомлення, яке було написано, співрозмовник прочитає та відреагує коли в нього на те буде час. Такий підхід краще, коли не потрібно очікувати результатів роботи іншого компонента. Наприклад, коли відбувся запит на оновлення певних даних. Наприклад,

банківські додатки, з їхнім оновленням через деякий час — в цьому можна побачити асинхронний підхід. Оскільки в банківській сфері дуже складна логіка роботи, тому запити опрацьовуються повільно.

Цей підхід реалізується за допомогою черг, спільних баз даних, брокерів запитів, моделі підписник-публікатор.

Також важливою різницею між синхронним та асинхронним викликом є рівень пов’язаності між собою компонентів. При синхронному виклику той, що викликає, повинен бути пов’язаним з тим кого викликають. При асинхронному виклику між ними немає зв’язку, оскільки їх пов’язує певний механізм по середині.

3 РЕАЛІЗАЦІЯ ВИСОКОРІВНЕВОЇ АРХІТЕКТУРИ ТА ТЕХНІЧНИЙ КОНТРОЛЬ ЇЇ ДОТРИМАННЯ

3.1 Процес авторизації користувача

Класична авторизація, що застосовується для веб додатків, а саме — самостійна реєстрація користувачів з подальшим підтвердженням пошти не підходить для нашого проекту. У нас немає потреби в авторизації звичайних користувачів. В додатку з точки зору функціональності існує три ролі. Звичайний користувач, який не авторизується та має доступ до публічної інформації. Редактор, який додає та редагує вміст сайту. Та адміністратор, що надає або забирає права в редактора та інших адміністраторів.

Авторизація потрібна тільки для двох останніх ролей. Також важливо зауважити що реєстрацію можуть проходити тільки дозволенні користувачі.

Отже, було розроблено схему роботи реєстрації та авторизації з наступними кроками (додаток Б):

— адміністратор отримує пошту майбутнього менеджера по інших захищених каналах (усно, через меседжер, емейл тощо);

— адміністратор реєструє користувача та отримує тимчасовий пароль;

— адміністратор передає цей пароль менеджеру по захищеному каналу зв’язку;

— менеджер логіниться в системі та змінює пароль.

Також адміністратор може надавати і віднімати права адміністратора в користувачів (але не можна забрати права в останнього адміністратора), видаляти користувачів.

Підхід з генерацією тимчасового паролю може здатись не безпечним в плані IT безпеки. Проте розробка цілком безпечного підходу вимагала б значних ресурсів та часу. Також було враховано те, що подібних реєстрацій планується не

велика кількість, та її будуть виконувати спеціалісти з високою кіберкультурою. Тому було вирішено зупинитись на такому підході.

3.2 Рефакторінг API до стандарту REST

В попередньому розділі було вирішено, що буде використовуватись підхід RESTfull API для нашого проекту. В обов’язки технічного керівника, архітектора або іншої людини відповідальній за технічний розвиток проекту входить необхідність проводити контроль за дотриманням підходів та стандартів запроваджених у команді.

API, що розробила команда ми можемо побачити на рисунку 3.1.

Рисунок 3.1 — API до приведення його до стандарту REST

Ми маємо чотири сутності та набір дій, які можна виконати над ними, створених за допомогою кінцевих точок. З позитивних сторін даної роботи можна

зазначити доречний поділ на сутності, логічне іменування кінцевих точок, та правильне використання різних методів HTTP протоколу.

До проблем можна віднести неправильний підхід до іменування, а саме відхід від принципу кінцевої точки як ресурсу. Коли у нас є звернення до ресурсу по першій ланці в імені, є дія закодована в методі, та параметри, які конкретизують дію або звернення.

Необхідно зазначити, що проблема наявна лише для Teacher та News оскільки інші були розроблені особисто, з врахуванням цих вимог.

На рисунку 3.2 Зображене API після виправлень.

Окрім виправлення імен та параметрів кінцевих точок, було додано префікс «admin» для усіх кінцевих точок. Це з однієї сторони дозволить краще орієнтуватись, яка частина відповідає за публічну сторону, а яка за адміністрування. З іншої сторони, це «перфорована лінія» за Робертом Мартіном — місце де в майбутньому можна провести розділення на окремі сервіси клієнтської та серверної частини. Клієнтська частина у свою чергу повинна працювати з адміністративними і клієнтськими кінцевими точками як окремими сервісами.

Шлях, який відповідав за отримання даних по сторінках, перетворився з «PagedNews» з двома параметрами, розміром сторінки та номером сторінки за порядком, на «news/page/{page}», з одним параметром — кількістю записів на сторінці. Керуючись логікою RESPfull API маємо ресурс News, його підресурс (представлення) — Page, з порядковим номером, який вказується як частина шляху. Додатковим параметром вказується розмір сторінки, як «уточнення» запиту. З професійної точки зору такий параметр не є елегантним, та його можна було б винести в заголовок запиту, проте після консультацій з командою було вирішено застосувати необов’язковий параметр (зі значенням за замовчуванням).

Auth

POST /admin/auth/login

POST /admin/auth/logout

News

GET /news/page/{page}

GET /news

GET /admin/news

POST /admin/news

PUT /admin/news

GET /admin/news/{id}

DELETE /admin/news/{id}

Teacher

GET /teacher/page/{page}

GET /teacher

GET /admin/teacher

POST /admin/teacher

PUT /admin/teacher

DELETE /admin/teacher

GET /admin/teacher/{id}

User

POST /admin/user

GET /admin/user

DELETE /admin/user/{email}

PUT /admin/user/{email}/password

POST /admin/user/{email}/role/{role}

DELETE /admin/user/{email}/role/{role}

Рисунок 3.2 — API після приведення до стандарту REST

Розглянемо конкретні перетворення кінцевих точок.

Важливим моментом є рефакторінг пошуку об’єктів за рядком. Розробник створив кінцеву точку «SearchedNewsString». Це є помилкою з точки зору стандартів, що було обрано. Сам пошуковий рядок не є ресурсом у нашій системі. Також це не схоже на назву команд (вони дозволяються в REST). Розглянемо тоді з точки зору ресурсу, що робить цей запит — він повертає колекцію ресурсів, з певним уточненням, отже це має бути просто запит /news з параметром, в який передається пошуковий рядок. Так, але може виникнути питання: що робити якщо з’явиться потреба в ще одній кінцевій точці, яка повертає такі ж самі дані але за

іншими параметрами. Ключове слово тут «параметри». Можна створювати декілька кінцевих точок, з різними наборами параметрів, і це буде правильно з точки зору проектування API, оскільки буде існувати єдина кінцева точка з різними параметрами. Тобто поліморфізм методів на рівні API.

Останніх п’ять кінцевих точок розглянемо разом. Це звичайні CRUD операції над ресурсом-сутністю. Прибираємо зайве з шляху кінцевої точки, залишаємо лише ім’я ресурсу.

Аналогічний рефакторинг відбувся і з іншою сутністю — «Teacher».

3.3 Перевірка якості та корегування програмного коду

Також в обов’язки технічного керівника, архітектора, та й старшого розробника входить проведення огляду результатів роботи інших розробників. Не обов’язково рев’ю проводить старший по посаді. Нормальною практикою є коли початківець перевіряє досвідченого працівника, для самонавчання, та запобігання виникнення ефекту авторитета.

Але з іншої точки зору — фінальний огляд повинен проводити саме технічний керівник проекту, оскільки він несе головну частину відповідальності за якість проекту а також має найбільший досвід та експертизу.

Проведемо огляд коду розробленого командою. Це серверна частина проекту, розроблена мовою C#.

Перше на що звернемо, це файлова структура (рисунок 3.3).

Одразу видно першу проблему — некоректний кореневий простір імен. Згідно стандарту розробки мови C# не допускається застосування в іменах проектів використання знаку «пробіл». Також гарною практикою є розділ імені проекту та його частини через точку. Тобто необхідно замінити «ot-api» на «Ot.Api» Для цього використано засіб швидкого рефакторингу в VisualStudio.

Також є багато зауважень щодо файлової структури в цілому. Відсутність поділу на рівні: представлень, логіки та даних. Змішування підходів іменування як по типу класів (Models, Entities, Services, Repositories), так і по функціональності

(CloudStorage, GraphQL) і тд. Також присутнє невірне розміщення класів: сутності для запитів та відповідей від сервера розміщувались в папці Models.

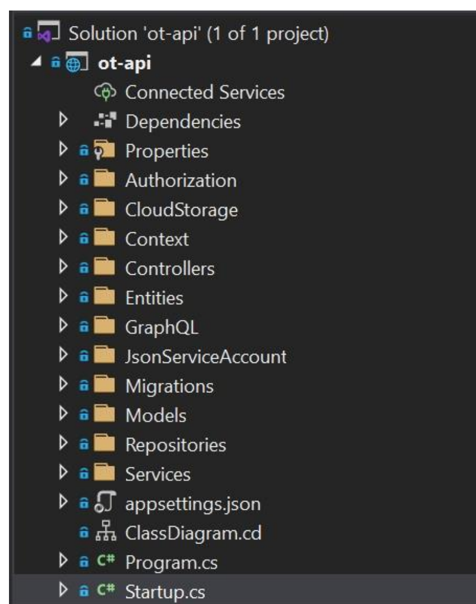


Рисунок 3.3 — Файлова структура до рефакторингу

Але найбільшою проблемою є повна монолітність та сильна залежність класів між собою. Таку архітектуру називають істинним монолітом. Проблемою її є складність в подальшій розробці, та фактична неможливість поступового переходу до мікросервісної архітектури.

На рисунку 3.4 зображено файлову та проектну структуру вже після всього рефакторінгу.

Перше на що необхідно звернути увагу це поділ на проекти. У нас є два проекти з бізнес логікою, а саме Logic.Admin та Logic.Client, як не мають залежностей від інших проектів. Інтерфейси залежностей знаходяться в цих же проектах. Це є гарною демонстрацією інверсії залежностей на рівні модулів.

Є два проекти з реалізаціями репозиторіїв, відповідно до технологій, а саме Data.SqlServer та Data.GoogleCloud. Про ці проекти логіка нічого не знає, а отже можна їх з легкістю переписати, коли зміниться технологія, а також зміни в репозиторії не будуть вимагати змін в логіці.

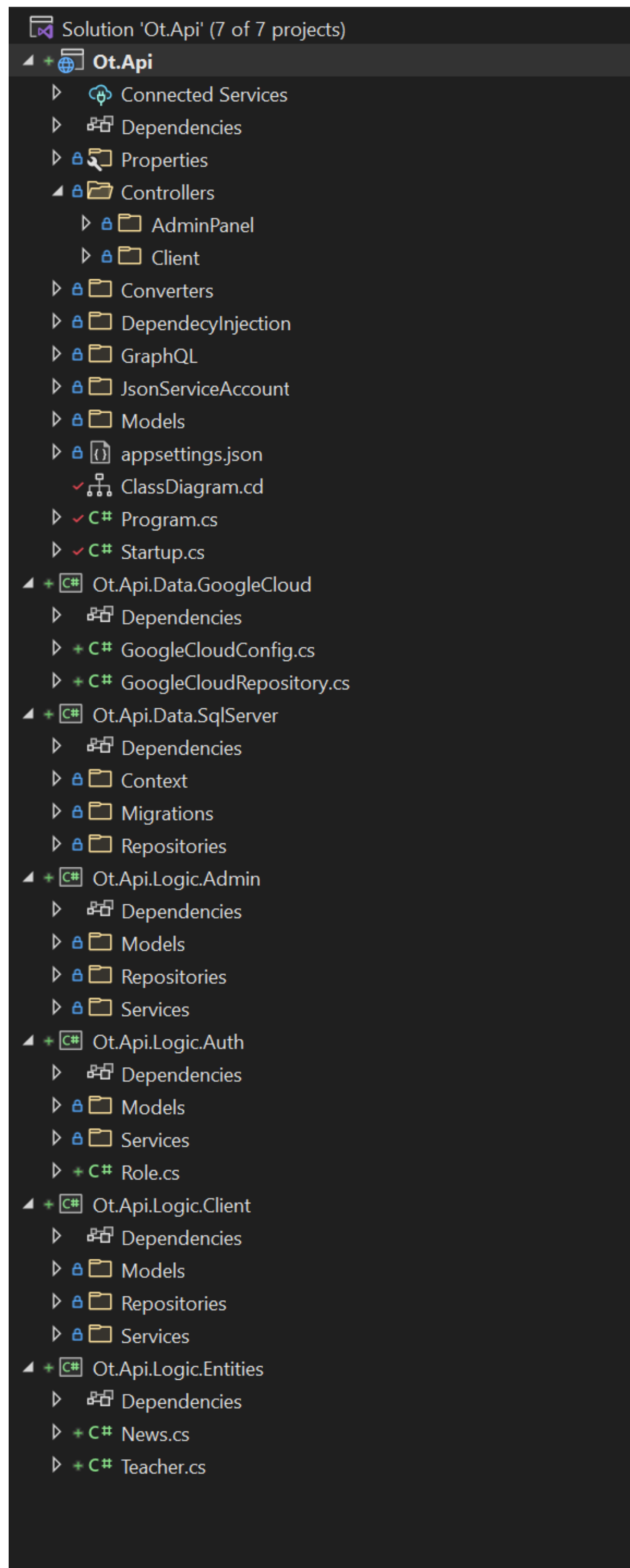


Рисунок 3.4 — Файлова структура після рефакторингу

Також є окремий проект для сутностей. Він був створений окремо, оскільки багато інших проектів мають посилання на нього. Це є гарним прикладом модуля з великою кількістю залежностей та з закритістю до змін, оскільки сутності зазвичай додаються, і вкрай рідко змінюються.

Окремо винесена авторизація. Вона ніяк не пересікається з бізнес логікою, та має багато залежностей на стороні бібліотеки.

І довершує все API проект, який містить мінімум логіки та максимум конфігурації. Це єдиний проект який знає про існування всіх інших та працює з ними. Він найбільше здатний до мінливості.

Розглянемо проведення рев’ю окремих класів.

На рисунку 3.5 є клас PageViewModel. Червоним виділено проблемні місця.

Перша проблема — це не видалення зайвих посилань на зовнішні простори імен. Так, в більшості випадків сучасний компілятор прибире їх самостійно та не буде підвантажувати простори імен, що не використовуються. Але не потрібно покладатись лише на компілятор. Подібні «рудименти» понижують зручність читання коду.

Наступне — неправильне ім’я. ViewModel використовуються в архітектурних патернах MVC та MVVM, у нас же звичайне API та View відсутні (веб сторінки які будуються серверним додатком). Доречніше було б назвати просто Model.

Наступна проблема — конструктор, з логікою. Для класів-сутностей, з використанням «бідного підходу» конструктор не потрібен, окрім як для присвоювання значень за замовчуванням. Тим більше проблемою є конструктор з логікою, оскільки ця логіка може додавати неочікуваної поведінки.

І останнє виправлення, яке має більш естетичний характер, це перетворення обрахункових властивостей, на лямбда функції. Це дозволить спростити вигляд виразу. Кінцевий результат можна побачити на рисунку 3.6.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace Ot.Api.Models.Pagination
{
    1 reference
    public class PageViewModel<T>
    {
        3 references
        public int PageNumber { get; private set; }
        2 references
        public int TotalPages { get; private set; }
        1 reference
        public List<T> ItemsAtPage { get; private set; }

        0 references
        public PageViewModel(int itemsCount, int pageNumber, int pageSize, List<T> itemsAtPage)
        {
            ItemsAtPage = itemsAtPage;
            PageNumber = pageNumber;
            TotalPages = (int)Math.Ceiling(itemsCount / (double)pageSize);
        }

        0 references
        public bool HasPreviousPage
        {
            get
            {
                return (PageNumber > 1);
            }
        }

        0 references
        public bool HasNextPage
        {
            get
            {
                return (PageNumber < TotalPages);
            }
        }
    }
}
```

Рисунок 3.5 — Клас PagedViewModel

```
3 references
public class PagedModel<T>
{
    2 references
    public int PageNumber { get; set; }

    1 reference
    public int TotalPages { get; set; }

    0 references
    public List<T> ItemsAtPage { get; set; }

    0 references
    public bool HasPreviousPage => PageNumber > 1;

    0 references
    public bool HasNextPage => PageNumber < TotalPages;
}
```

Рисунок 3.6 — Клас PagedViewModel після виправлень

Наступний метод — це, FilterByText (рисунок 3.7). Проблеми є три. Перша — важкість в читанні, необхідно виправити табуляцію. Друга — повтор виклику методів ToLower().Trim(), та остання — подвійне виконання тієї самої дії — нормалізації вхідного параметру.

```
1 reference
public ICollection<News> FilterByText(string searchedText)
{
    return repository
        .GetList()
        .Where(news =>
            news.Title.ToLower().Trim().Contains(searchedText.ToLower().Trim())
        || news.Description.ToLower().Trim().Contains(searchedText.ToLower().Trim())).ToList();
}
```

Рисунок 3.7 — Метод FilterByText

Проводимо винесення методу, створення змінної з нормалізованим вхідним параметром та виправляємо табуляцію і отримуємо результат як на рисунку 3.8.

```
1 reference
public ICollection<News> FilterByText(string searchedText)
{
    var normalizedSearchedText = NormalizeString(searchedText);

    return repository
        .GetList()
        .Where(news =>
            NormalizeString(news.Title).Contains(normalizedSearchedText)
            || NormalizeString(news.Description).Contains(normalizedSearchedText))
        .ToList();
}

3 references
private string NormalizeString(string text)
{
    return text.ToLower().Trim();
}
```

Рисунок 3.8 — Демонстрація прийому «винесення метода»

Наступний проблемний клас, до реалізації якого потрібно внести зміни, — ImagesService (рисунок 3.9).

```
5 references
public class ImagesService
{
    IConfiguration configuration;
    ICloudStorage cloudStorage;

    2 references
    public ImagesService(IConfiguration config)
    {
        configuration = config;
        cloudStorage = new GoogleCloudStorage(config);
    }

    2 references
    public async Task<ImageInfo> SaveImagesAsync(IFormFile imageFile)
    {
        string filePath = null;
        string cloudLink = null;

        if (imageFile != null)
        {
            Guid id = Guid.NewGuid();
            string ext = Path.GetExtension(imageFile.FileName);

            filePath = id + ext;

            cloudLink = await cloudStorage.UploadFileAsync(imageFile, filePath);
        }

        return new ImageInfo(filePath, cloudLink);
    }
```

Рисунок 3.9 — Клас ImageService

Проблемами є відсутність абстракції — необхідно створити відповідний інтерфейс. Залежність від інтерфейсу бібліотеки для розробки API, що є порушенням інверсії залежностей. І нарешті — створення залежності прямо в конструкторі, це робить неможливим тестуванні цього тесту.

```
2 references
public class ImageService : IImageService
{
    private IFileRepository fileRepository;

    0 references
    public ImageService(IFileRepository fileRepository)
    {
        this.fileRepository = fileRepository;
    }
```

Рисунок 3.10 — Клас ImageService після виправлень

Для виправлення створюємо новий інтерфейс та переводимо залежність в параметри конструктора. Щодо IConfiguration, то в результаті глибокого рефакторінгу класу потреба в ньому пропала, хоча класичним рішенням було б

створення класу-конфігурації яка б стала залежністю. Результат рефакторінгу можна побачити на рисунку 3.10.

На рисунку 3.11 зображена досить нетипова помилка в методі UpdateNews.

```
1 reference
public void UpdateNews(News updatedNews, News news)
{
    news.Title = updatedNews.Title;
    news.Description = updatedNews.Description;
    news.Content = updatedNews.Content;

    if (updatedNews.ImageStorageUrl != null)
    {
        news.ImageStorageUrl = updatedNews.ImageStorageUrl;
        news.ImageName = updatedNews.ImageName;
    }

}
```

Рисунок 3.11 — Нетипова помилка в методі UpdateNews

Дана функція, а також інші в цьому класі займались лише модифікацією моделей, без збереження в базу. Це є порушенням правил чистоти функції, оскільки чиста функція не повинна вимагати виконання дій ні до ні після її виклику. Виправляється додаванням роботи з базою даних. Побічним ефектом стала відмова від одного з вхідних параметрів, оскільки попередній стан можна отримати з бази даних. Також додатковим виправленням стало видалення слова News з назви функції, оскільки не потрібно уточнювати ім’я сутності в кожному методі, якщо він зазначений в імені класу. Результат виправлення подано на рисунку 3.12.

Класичний приклад порушення принципу єдності відповідальності показано на рисунку 3.13. В класі TeacherService є метод для конвертації сутностей. Для цієї цілі було створено окремий клас-конвертор.

Іншим виправлення, є заміна циклу на LINQ операцію перетворення. Результат показано на рисунку 3.14.

```
2 references
public void UpdateNews(News updatedNews)
{
    var news = repository.GetById(updatedNews.Id);

    news.Title = updatedNews.Title;
    news.Description = updatedNews.Description;
    news.Content = updatedNews.Content;

    if (updatedNews.ImageStorageUrl != null)
    {
        news.ImageStorageUrl = updatedNews.ImageStorageUrl;
        news.ImageName = updatedNews.ImageName;
    }

    repository.Update(news);
}
```

Рисунок 3.12 — Виправлений метод UpdateNews

Останній приклад помилки, але не останній за важливістю — порушення ізоляції рівнів (рисунок 3.15). Контролер не повинен залежати від репозиторіїв напряму.

```
/// <summary>
/// Get all list of teachers from db
/// </summary>
/// <returns>List of teachers</returns>
[HttpGet("Teachers")]
0 references
public ActionResult Get()
{
    var teachers = service.GetAll();
    if (teachers == null)
    {
        return BadRequest();
    }

    List<TeacherResponse> teacherViewModels = new List<TeacherResponse>();

    foreach (var item in teachers)
    {
        teacherViewModels.Add(_teacherService.CreateTeacherViewModel(item));
    }

    return Ok(teacherViewModels);
}
```

Рисунок 3.13 — Порушення принципу єдності відповідальності SOLID

```
/// <summary>
/// Get all list of teachers from db
/// </summary>
/// <returns>List of teachers</returns>
[HttpGet("")]
0 references
public ActionResult Get()
{
    var teachers = service.GetAll();
    if (teachers == null)
    {
        return BadRequest();
    }

    List<TeacherResponse> responses = teachers.Select(x => x.Convert()).ToList();

    return Ok(responses);
}
```

Рисунок 3.14 — Використання конвертора в LINQ виразі

```
private IRepository<News> _newsRepository;
private NewsService _newsService;

0 references | 0 changes | 0 authors, 0 changes
public NewsController(IRepository<News> newsRepository, NewsService newsService)
{
    _newsService = newsService;
    _newsRepository = newsRepository;
}
```

Рисунок 3.15 — Використання конвертора в LINQ виразі

3.4 Покращення якості, за рахунок застосування рішень підвищеної складності

Архітектор зобов'язаний додавати до проекту кращі рішення в складних місцях. Тобто вирішувати самі нетривіальні задачі. Він повинен робити це не лише з ціллю вирішення проблеми, але й для передачі досвіду команді.

Розглянемо декілька запроваджених прикладів «Найкращого досвіду» або як ще їх називають «Best Practices».

Для початку розглянемо підхід в організації Startup класу в ASP.Net додатку, який слугує для конфігурування проекту.

За замовчуванням він має досить громіздкий вигляд. До впровадження підходу він мав вигляд як показано на рисунку 3.16. А це відносно маленький

проект, і вже є труднощі в читанні та розумінні його.

```
2 references
public class Startup
{
    0 references
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    4 references
    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to the container.
    0 references
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers();
        services.AddSession();
        services.AddControllersWithViews();

        services.AddCors(o => o.AddPolicy("AllowAnyOrigin",
            builder =>
            {
                builder.AllowAnyOrigin()
                    .AllowAnyMethod()
                    .AllowAnyHeader();
            }));
        services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
        services.AddPooledDbContextFactory<Context.Context>(provider => provider.UseSqlServer(Configuration.GetConnectionString("VNTU_ComputingTechnologiesDepartmeent")));
        services.AddScoped<Context.Context>(p => p.GetRequiredService<IDbContextFactory<Context.Context>>().CreateDbContext());

        services
            .AddGraphQLServer()
            .AddType<NewsType>()
            .AddType<TeacherType>()
            .AddQueryType<Query>()
            .AddFiltering()
            .AddSorting();

        services.AddTransient<IRepository<News>, NewsRepository>();
        services.AddTransient<IRepository<Teacher>, TeachersRepository>();
        services.AddTransient<NewsService>(provider => new NewsService(Configuration));
        services.AddTransient<TeacherService>(provider => new TeacherService(Configuration));
        services.AddSwaggerGen(c =>
        {
            c.SwaggerDoc("v1", new OpenApiInfo {
                Title = "Ot.Api",
                Version = "v1",
                Description = "Документація для інформаційної системи кафедри обчислювальної техніки ВНТУ" });
        });

        services.AddIdentity<User, IdentityRole>(opts => {
            opts.Password.RequiredLength = 8;   //
            opts.Password.RequireNonAlphanumeric = false;
            opts.Password.RequireLowercase = true;
            opts.Password.RequireUppercase = true;
            opts.Password.RequireDigit = true;
        })
            .AddEntityFrameworkStores<Context.Context>();

        services.AddTransient<IUserService, UserService>();
        services.AddTransient<IRoleService, RoleService>();
        services.AddTransient<IAuthService, AuthService>();
    }

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    0 references
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();

        }

        app.UseCors("AllowAnyOrigin");

        app.UseSwagger();
        app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "Ot.Api v1"));

        app.UseSession();
        app.UseHttpsRedirection();

        app.UseRouting();

        app.UseAuthentication();
        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
            endpoints.MapGraphQL();
        });

        app.UseGraphQLVoyager(new GraphQLVoyagerOptions()
        {
            GraphQLEndPoint = "/graphql",
            Path = "/graphql-voyager",
        });
    }
}
```

Рисунок 3.16 — Громіздкий Startup до запровадження декомпозиції

Гарним вирішенням цієї проблеми є винесення конфігурування залежностей в окремі класи, наприклад з методами розширювання. Приклад класу Startup з його використанням зображено на рисунку 3.17. Логіка поділу на класи конфігуратори може бути різна. На цьому проекті я застосував поділ по аспекту додатку. Також використовував підхід поділу по предметній області, тобто коли є конфігурування роботи з користувачами, і конфігурування автоматизації складу, як приклад.

```
// This method gets called by the runtime. Use this method to add services to the container.
0 references
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    services.AddSession();
    services.AddControllersWithViews();

    services.ConfigureAuthorization();
    services.ConfigureDataContext(Configuration);
    services.ConfigureGraphQL();
    services.ConfigureServices();
    services.ConfigureSwagger();
}
```

Рисунок 3.17 — Startup із запровадженою декомпозицією

Але, можна допустити помилку яка на перший погляд є гарною ідеєю. А саме розміщення класів - конфігураторів залежностей разом з класами які вони конфігурують, тобто в середині проекту з бізнес логікою. Це неправильно, оскільки в такому випадку появиться скрізна залежність на бібліотеку для організації залежностей.

Додатковою гарною практикою, є поділ всередині конфігуратора на окремі методи а також поділ на логічні блоки пустими рядками. Прикладом конфігуратора є клас, що додає сервіси та репозиторії. Він зображений на рисунку 3.18. Інші конфігуратори є однотипними.

Ще одне корисне виправлення коду зображено на рисунку 3.19. Жорстке прописування роутів позбавить від проблем зі зміною версій, неакуратними змінами в конфігураторі роутів, оскільки не виникне проблема з випадковою поломкою всіх шляхів. Також це полегшить пошук контролерів в середині проекту.

```
0 references
public static class ConfigureServices
{
    1 reference
    public static void ConfigureCustomServices(this IServiceCollection services)
    {
        services.AddRepositories();
        services.AddServices();
    }

    1 reference
    private static void AddRepositories(this IServiceCollection services)
    {
        services.AddTransient<Admin.Repositories.IRepository<News>, NewsRepository>();
        services.AddTransient<Admin.Repositories.IRepository<Teacher>, TeacherRepository>();

        services.AddTransient<Client.Repositories.IRepository<News>, NewsRepository>();
        services.AddTransient<Client.Repositories.IRepository<Teacher>, TeacherRepository>();

        services.AddTransient<Admin.Repositories.IFileRepository, GoogleCloudRepository>();
    }

    1 reference
    private static void AddServices(this IServiceCollection services)
    {
        services.AddTransient<Admin.Services.INewsService, Admin.Services.NewsService>();
        services.AddTransient<Admin.Services.ITeacherService, Admin.Services.TeacherService>();
        services.AddTransient<Admin.Services.IImageService, Admin.Services.ImageService>();

        services.AddTransient<Client.Services.INewsService, Client.Services.NewsService>();
        services.AddTransient<Client.Services.ITeacherService, Client.Services.TeacherService>();
    }
}
```

Рисунок 3.18 — Приклад конфігуратора

```
[Route("admin/teacher")]
1 reference
public class TeacherController : Controller
```

Рисунок 3.19 — Прописаний шлях, що обробляє контролер

Наступним «гарним рішенням» є множинна реалізація схожих інтерфейсів. У нас є інтерфейс IRepository одразу в двох проектах — Logic.Client та Logic.Admin. Але для обох цих проектів існує лише одна реалізація рівня даних. Можна було б розробити два репозиторії незалежно, розмістивши в різних папках або давши різні імена. Але це не є елегантним рішенням. В цьому проекті використали саме множинну реалізацію (рисунок 3.20).

Також було використано ще дві «кращих практики». У нас був конфлікт імен, оскільки в обох проектах інтерфейс мав спільну назву. Цей конфлікт був вирішений через псевдоніми. Таке рішення не вимагає змін в проекті з логікою.

Іншим рішенням — є конфлікт методів між інтерфейсами. Це було вирішено через створення загального методу, який одночасно належав першому інтерфейсу та додаткового метода, що викликає перший. Фактично у нас є два різних інтерфейса метода які посилаються на ту саму реалізацію.

```
using Ot.Api.Logic.Client.Models;
using Ot.Api.Logic.Entities;
using System;
using System.Collections.Generic;
using System.Linq;
using Admin = Ot.Api.Logic.Admin.Repositories;
using Client = Ot.Api.Logic.Client.Repositories;

namespace Ot.Api.Data.SqlServer.Repositories
{
    3 references
    public class NewsRepository : Admin.IRepository<News>, Client.IRepository<News>
    {
        private Context context;

        0 references
        public NewsRepository(Context context)...

        4 references
        public int Count()...

        3 references
        public void Create(News item)...

        3 references
        public void Delete(int id)...

        5 references
        public News GetById(int id)...

        4 references
        public List<News> GetList()
        {
            return context.News.ToList();
        }

        3 references
        public PagedModel<News> GetPagedItems(int pageSize, int pageNumber)...

        3 references
        public void Update(News item)...

        3 references
        ICollection<News> Client.IRepository<News>.GetList()
        {
            return GetList();
        }
    }
}
```

Рисунок 3.20 — Реалізація трьох «кращих рішень» в одному файлі

Останній, так і не реалізований, в зв’язку з відсутністю потреби, але все таки розроблений підхід — відмова від анотацій Entity Framework на користь FluentAPI.

На перший погляд це не логічно, оскільки FluentAPI більш громіздкий, та не зручний. Але істина перевага в тому що модель сутності нічого не знає про реалізацію бази даних. Важливість рішення полягає в тому, що у нас на проект з сутностями зав'язані всі інші проекти. Тобто у випадку якщо буде «скрізна» залежність на бібліотеку для бази даних, у нас порушиться принцип інкапсуляції модулів в цілому.

В даному проекті цей підхід так і не був реалізований, оскільки не було потреби в коригування побудови бази даних EntityFramework. Базової побудови було досить.

3.5 Реалізація високорівневої архітектури та перспективи розвитку інформаційної системи

Детальна схема високорівневої архітектури наведена в додатку В. З однієї сторони її не можна назвати істинним монолітом оскільки є поділ на проекти, інкапсуляція модулів, інверсія залежностей. Але і до мікросервісної її не можна віднести, бо є лише один проект серверної сторони, вся логіка підіймається одночасно.

Це модульна архітектура з наявністю потенційних модулів. Так, мікросервісна — це також модульна архітектура, але ці поняття не тотожні. Мікросервіси — це окрема реалізація поділів на модулі.

На даному етапі у нас немає потреби в окремих сервісах під блоки логіки, що обумовлено невеликим навантаженням, малою кількістю користувачів, невеликими функціональними можливостями, обмеженими людськими, технічними та часовими ресурсами. Розробка мікросервісів на даному етапі життєвого циклу проекту не виправдана.

Але вже зараз можна розраховувати, що у випадку успішності проекту ситуація зміниться, та знадобляться переваги мікросервісів. Будуть нові

функціональні можливості, більше навантаження, необхідність в нових механізмах, таких як SEO оптимізація.

Елегантним рішенням є модульно-монолитна архітектура. Було розроблено монолітний додаток, але прораховано, як кожне рішення буде переноситись на мікросервіси. При розробці, залежності повинні були працювати так, щоб було однаково, яка в них реалізація — репозиторія SQL бази, кеш в пам’яті чи окремий сервіс. Проекти повинні були мінімально залежати один від одного. Більшість проектів можна було б виділити з основного рішення, перенести в нове, дописати йому введення та виведення та використовувати як новий сервіс.

Розглянемо детально модулі, які є на даному етапі, а також зовнішні частини системи.

Logic.Entities — невеликий проект зі списком усіх сутностей, які застосовуються в проекті. Оскільки сутності міняються дуже рідко, підходить під велику кількість посилань на неї.

Logic.Auth — інкапсулює всю авторизацію. Має посилання на зовнішні бібліотеки, а саме IdentityService з пакета ASP.Net Core.

Logic.Client — має логіку для отримання та перетворення даних, що застосовуються на клієнті. Має інтерфейси для репозиторіїв, проте ніяк не залежить від конкретних реалізацій.

Logic.Admin — має логіку для оновлення, додавання та видалення даних. Також має інтерфейси репозиторіїв.

Data.SqlServer — реалізує репозиторії за допомогою MSSQL Server.

Data.GoogleCloud — реалізація файлового репозиторію.

API — інтерфейсна та конфігураційна частина, знає про всю структуру додатку окрім клієнтської частини.

SQLServer — певна база даних, що буде використовуватись цим проектом. Її розміщення не визначено, це додає певної гнучкості.

GoogleCloud — хмарне сховище для файлів (фотографій).

Website та AdminPanel — клієнтська частина.

Отже маємо хорошу інкапсуляцію модулів. Операції та технології вводу та виводу залежать від логіки, а не навпаки (як зазвичай).

Розроблюваний проект на даному етапі — це лише основа для подальшого розвитку. Враховувалась можливість появи нових елементів, що зараз не розглядається. Правильно розроблена архітектура дозволяє відкладати на потім вирішення таких питань, а також дозволяє без проблем розширятись та покращуватись.

Потенційні можливості розвитку проекту схематично зображенні в додатку Г. Розглянемо їх детально:

— перехід на мікросервіси, для чого, як було написано вище, достатньо дописати код для операцій вводу та виводу на кожен проект і такий перехід стане можливим;

— перехід в хмарний хостинг, для якого немає ніяких завад, а в подальшому, у випадку ускладнення проекту це буде необхідно для організації балансування навантаження та маршрутизації;

— використання SSR (Server Side Rendering) — технології, яка дозволяє виконувати побудову клієнтської частини на стороні сервера, для цього розгортається додатковий сервіс створений на основі Node JS.

ВИСНОВКИ

За останніх десять років ситуація в сфері розробки веб-додатків сильно змінилась.. Замість стандартного набору з ASP .Net MVC, EF6 і т.п. прийшло велика кількість нових технологій та підходів. А класична трикомпонетна архітектура залишилась, в основному, в старих проектах. Часто стали траплятись сервісні проекти, які не мають клієнтської частини, як такої, та слугують допоміжними в більш складній системі.

У зв’язку з цим виросла значимість ролі архітектора, технічного керівника або іншого спеціаліста, відповідального за проектування архітектурної компоненти. Вкладенні час та зусилля на правильне проектування, підбір технологій, команди, планування дій дозволяють зекономити багато ресурсів та позбутись потенційних проблем.

У даній комплексній бакалаврській дипломній роботі розроблено розподілений веб-додаток за допомогою гнучких методологій. Як мову розробки обрано C#, як кращий кандидат на основі порівняння за багатьма критеріями. Як база даних використовується MS SQL Server, хоча обрана технологія не кінцева та може змінюватись. Для розробки клієнтської частини використовується фреймворк React разом з мовою TypeScript.

При розробці було застосовано модульний підхід, створенні можливості для подальшого переходу на мікросервісну архітектуру, додавання підходу CQRS, SSR, баз даних для кеша, нових функціональних можливостей.

Для збереження фотографій було обрано технологію Google Cloud Bucket. Також була створена можливість розгортання в хмарному середовищі та поступової міграції на технології Azure або Amazon Web Services.

При розробці інформаційної системи було створено нетипову логіку авторизації та аутенфікації.

Проведено детальне код рев’ю та рефакторинг, розглянуто та описано типові помилки. Відбулось приведення API до стандарту RESTfull, а також запроваджено

та описано ряд кращих рішень.

Проект, що був розроблений, не є завершеним та може розвиватись в подальшому, для чого при його розробці врахована можливість появи нових елементів.

Результати комплексної бакалаврської дипломної роботи впроваджено мережі ВНТУ як сайт кафедри обчислювальної техніки (додаток Л)

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Аналіз розвитку підходів до розробки веб-додатків та зростання впливу розподіленої архітектури [Електронний ресурс] / Д. Ю. Станішевський, О. В. Войцеховська — 2022. — conferences.vntu.edu.ua/index.php/all-fitki/all-fitki-2022/paper/view/15887/13334

2. Р. Мартін, Чиста архітектура. Харків, Україна : Фабула, 2021, 368 с. ISBN: 978-617-09-5286-8.

3. REST documentation [Електроний ресурс] – Режим доступу до ресурсу: https://docs.github.com/en/rest.

4. Most popular database management system [Електроний ресурс] – Режим доступу до ресурсу: https://www.stackscale.com/blog/popular-database-management-systems/

5. What is Entity Framework Core [Електронний ресурс]. Режим доступу: https://www.entityframeworktutorial.net/what-is-entityframework.aspx.

6. Creating Entity Framework Data Model. [Електронний ресурс]. Режим доступу:https://www.entityframeworktutorial.net/entityframework6/create-entity-data-model.aspx. ADO.NET Overview. [Електронний ресурс]. Режим доступу: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-overview.

7. What is REST. [Електронний ресурс]. Режим доступу: https://restfulapi.net/.

8. API Documentation & Design Tools with Swagger. [Електронний ресурс]. Режим доступу: https://swagger.io/

9. Differences between MySQL and SQL Server [Електронний ресурс]. Режим доступу: https://www.geeksforgeeks.org/difference-between-mysql-and-ms-sql-server/

10. What is WEB-API [Електронний ресурс]. Режим доступу: https://www.tutorialsteacher.com/webapi/what-is-web-api

11. What are HTTP Status Code [Електронний ресурс]. Режим доступу:

https://umbraco.com/knowledge-base/http-status-codes/

12. Create A Storage Bucket In Google Cloud Platform [Електронний ресурс]. Режим доступу: https://www.c-sharpcorner.com/article/create-a-storage-bucket-in-google-cloud-platform/

13. The benefits of ReactJS [Електронний ресурс]. Режим доступу: https://www.peerbits.com/blog/reasons-to-choose-reactjs-for-your-web-development-project.html.

14. Fetch API [Електронний ресурс]. Режим доступу: https://www.jscamp.app/docs/javascript27/.

15. React і Redux: функціональна веб-розробка. Алекс Бенкс, Єва Порселло 2018. – 336с.

ДОДАТОК А

Технічне завдання

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп’ютерної інженерії
Кафедра обчислювальної техніки

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ
_____ проф., д.т.н. О.Д. Азаров
«__» ________ 2022 р.

ТЕХНІЧНЕ ЗАВДАННЯ
на виконання комплексної бакалаврської дипломної роботи на тему:
«Клієнт-серверна інформаційна система підрозділу навчального закладу з можливістю розгортання в хмарному середовищі. Частина 1. «Проектування розподіленої архітектури та менеджмент з використанням гнучких методологій»
08-23.КБДР.045.00.000 ІЗ

Науковий керівник: к.т.н.,доц. каф. ОТ
________ Войцеховська О.В.

Виконав: студент групи 2КІ-18б
________ Станішевський Д.Ю.

Вінниця 2022 р.

1 Підстава для виконання дипломної роботи

Підстава для виконання дипломної роботи (ДР): актуальність досліджень полягає у необхідності розробки розподіленого клієнт-серверного додатку з використанням гнучких методологій, та хмарних технологій, з можливістю розширення та масштабування та наказ про затвердження теми дипломної роботи.

2 Мета ДР і призначення розробки

Мета ДР — розробка клієнт-серверного веб-додатку офіційної сторінки кафедри обчислювальної техніки Вінницького національного технічного університету. Призначення ДР полягає в виконанні комплексної бакалаврської дипломної роботи для подальшого використання.

3 Вихідні дані для виконання ДР

Вихідними даними для виконання ДР є веб-сайт кафедри обчислювальної техніки.

4 Технічні вимоги до виконання ДР

Технічними вимогами до виконання ДР є контент для наповнення веб-додатку, можливість його редагувати, додавати та видаляти. Можливість реєструвати нових менеджерів по контенту. Доступність веб-сайту, та його швидкодія.

5 Етапи ДР та очікувані результати приведені в таблиці А.1.

Таблиця А.1 — Етапи ДР

№ етапу	Назва етапу	Термін виконання		Очікувані результати
		початок	кінець	
1	Аналіз тенденцій в розробці веб-додатків	09.02.22	24.02.22	Розділ 1
2	Аналіз технологій для розробки веб-додатку	25.02.22	20.03.22	Розділ 1
3	Розробка базової архітектури	21.03.22	05.04.22	Розділ 2
4	Організація команди по сумісній роботі	06.04.22	10.04.22	Розділ 2
5	Проектування та розробка авторизації	11.04.22	02.05.22	Розділ 3
6	Проведення контролю над розробкою. Рефакторінг	03.05.22	25.05.22	Кінцевий веб-додаток
7	Оформлення пояснювальної записки та ілюстративного матеріалу	26.05.22	10.06.22	ПЗ, презентація

6 Матеріали, що подаються до захисту ДР

Пояснювальна записка ДР, графічні і ілюстративні матеріали, протокол попереднього захисту БДР на кафедрі, відзив наукового керівника, рецензія, анотації до МКР українською та іноземною мовами, нормоконтроль про відповідність оформлення ДР діючим вимогам.

7 Порядок контролю виконання та захисту ДР

Виконання етапів графічної та розрахункової документації ДР контролюється науковим керівником згідно зі встановленими термінами. Захист ДР відбувається на засіданні Державної екзаменаційної комісії, затвердженою наказом ректора.

8 Вимоги до оформлення ДР

Вимоги викладені в методичних вказівках до дипломного проектування, ДСТУ_ 3008-95, ДСТУ 3974-2000 «Правила виконання дослідно-конструкторських робіт. Загальні положення» та діючого ГОСТ 2.114-95 ЕСКД.

Технічне завдання до виконання прийняв ______ Станішевський Дмитро

ДОДАТОК Б

UML діаграма взаємодії редактора та адміністратора з авторизацією

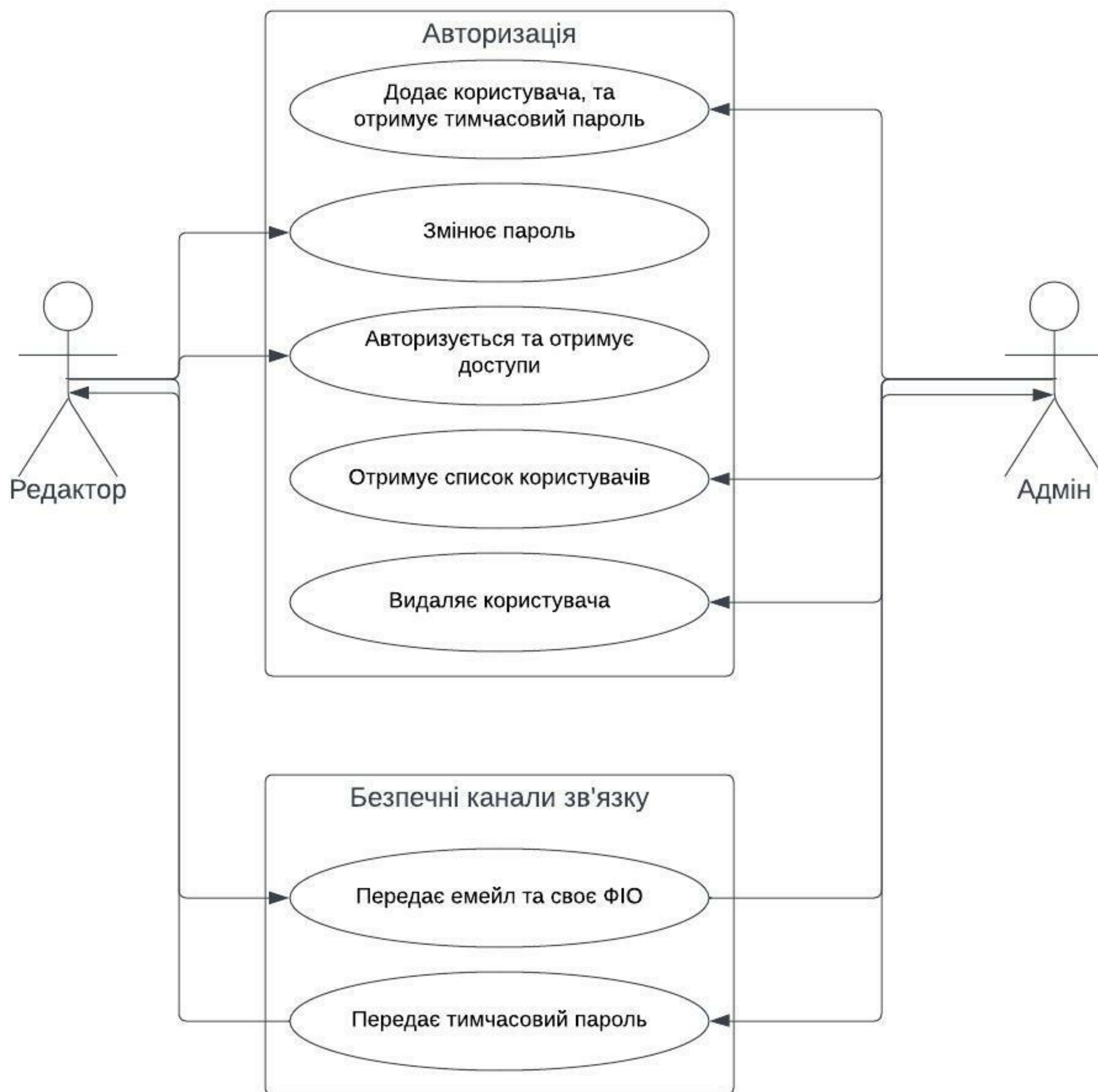


Рисунок Б.1 — UML діаграма взаємодії редактора та адміністратора з авторизацією

ДОДАТОК В

Структурна схема високорівневої архітектури

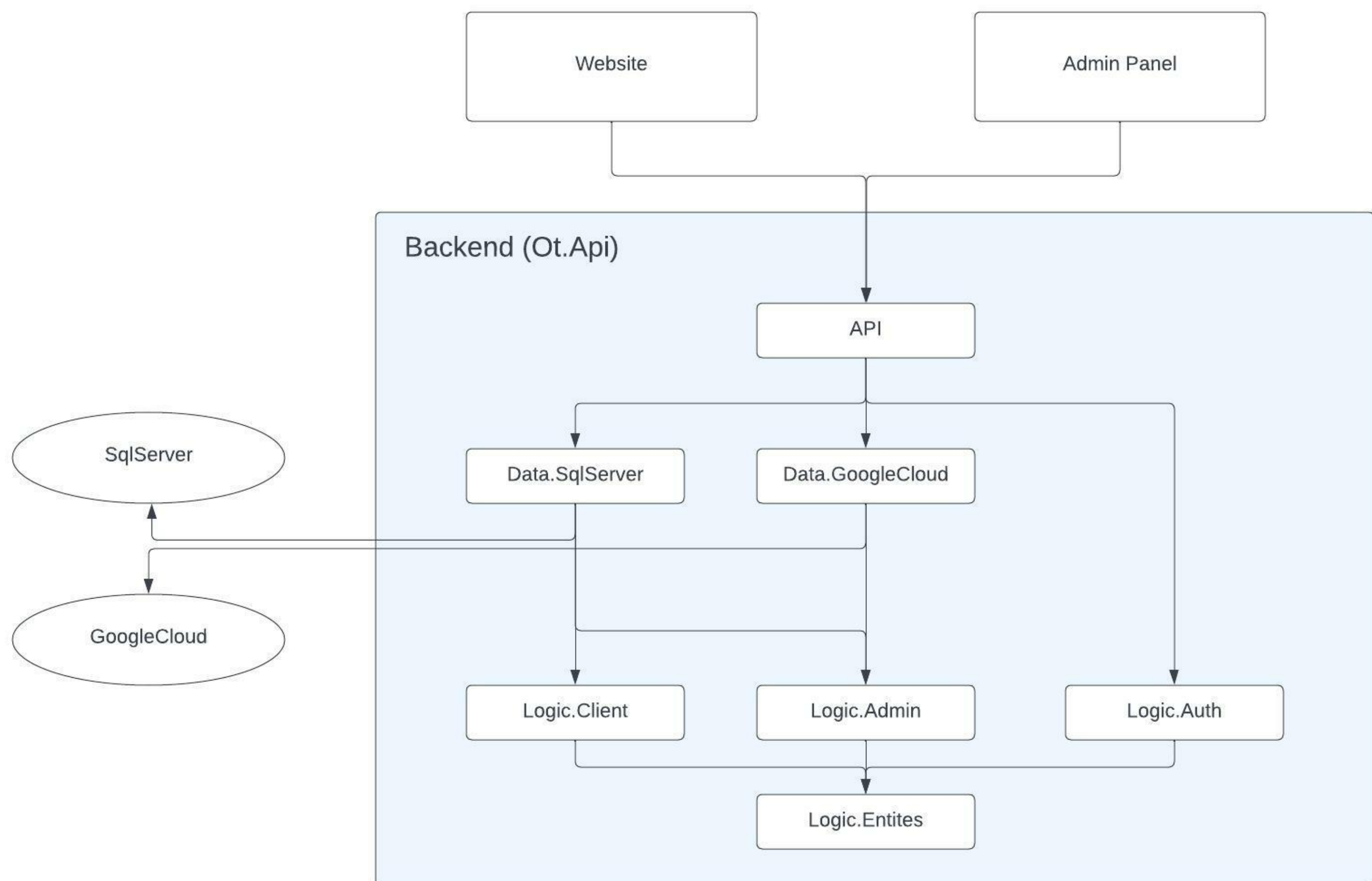


Рисунок В.1 — Структурна схема високорівневої архітектури

ДОДАТОК Г

Структурна схема архітектури потенційного розвитку додатку

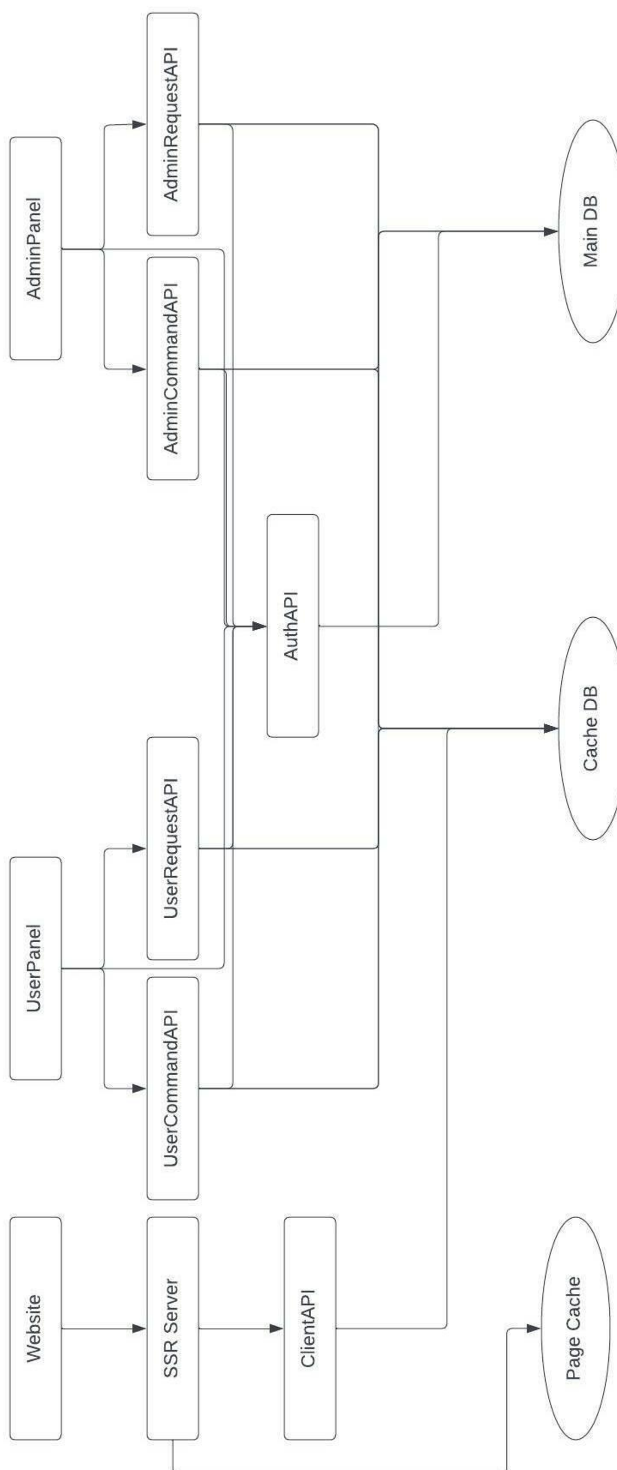


Рисунок Г.1 — Структурна схема архітектури потенційного розвитку додатку

ДОДАТОК Д

Структурна схема SQL бази даних для авторизації

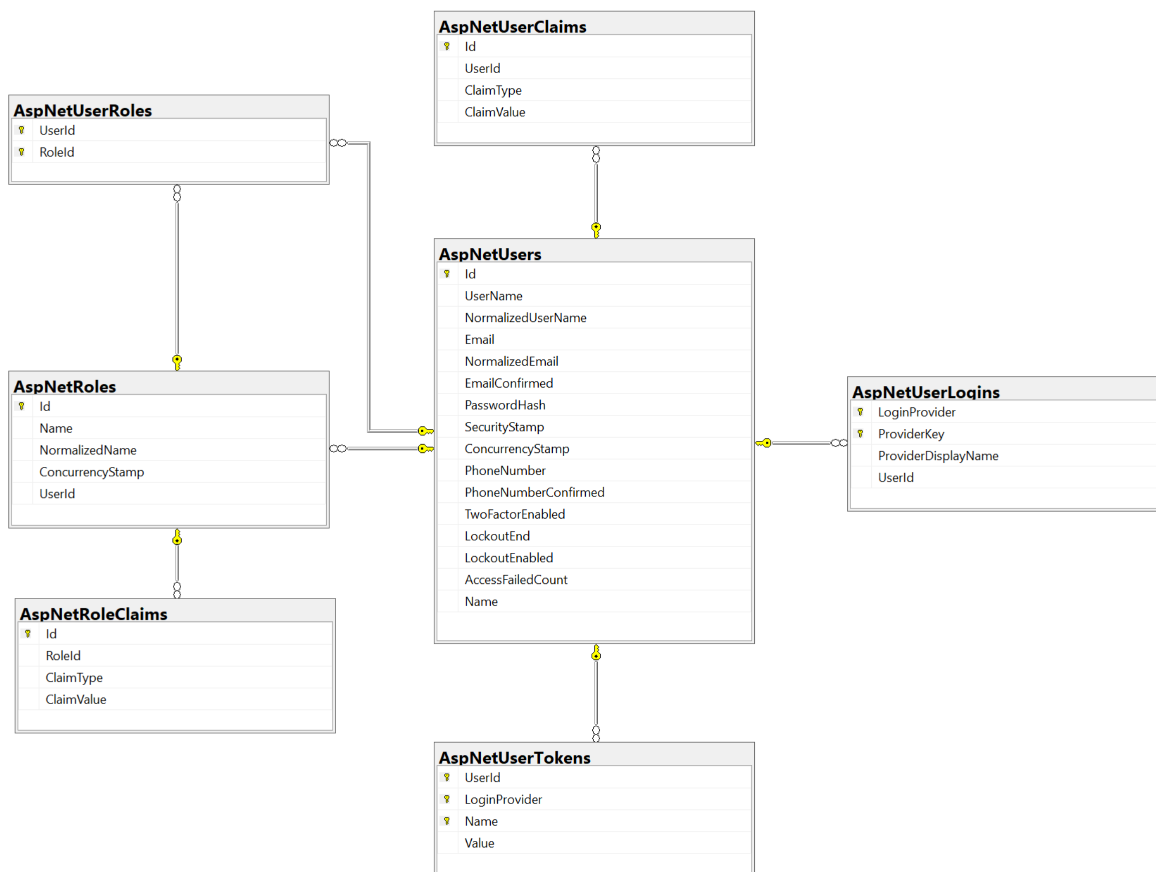


Рисунок Д.1 — Структурна схема SQL бази даних для авторизації

ДОДАТОК Е

UML діаграма структури SQL бази даних для авторизації

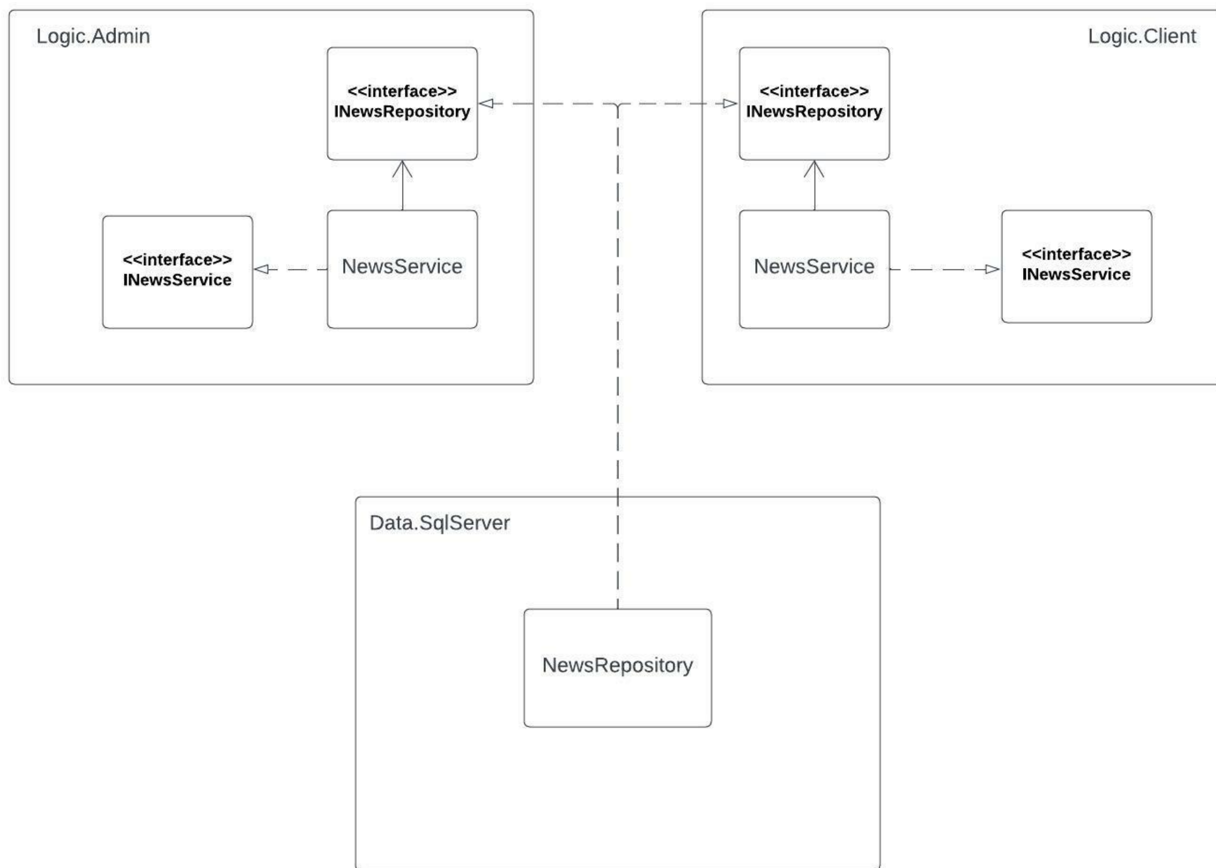


Рисунок Е.1 — UML діаграма структури SQL бази даних для авторизації

ДОДАТОК Ж

Лістинг множинної реалізації інтерфейсу

```
using Ot.Api.Logic.Client.Models;
using Ot.Api.Logic.Entities;
using System;
using System.Collections.Generic;
using System.Linq;
using Admin = Ot.Api.Logic.Admin.Repositories;
using Client = Ot.Api.Logic.Client.Repositories;

namespace Ot.Api.Data.SqlServer.Repositories
{
   public class NewsRepository : Admin.IRepository<News>, Client.IRepository<News>
   {
      private Context context;

      public NewsRepository(Context context)
      {
         this.context = context;
      }

      public int Count()
      {
         return context.News.Count();
      }

      public void Create(News item)
      {
```

```
            context.News.Add(item);
            context.SaveChanges();
        }

        public void Delete(int id)
        {
            var newsToDelete = context.News.Find(id);
            context.News.Remove(newsToDelete);

            context.SaveChanges();
        }

        public News GetById(int id)
        {
            return context.News.FirstOrDefault(x => x.Id == id);
        }

        public List<News> GetList()
        {
            return context.News.ToList();
        }

        public PagedModel<News> GetPagedItems(int pageSize, int pageNumber)
        {
            var newsInPage = context.News.Skip(pageSize * (pageNumber -
1)).Take(pageSize).ToList();

            return new PagedModel<News>
```

```
            {
                TotalPages = (int)Math.Ceiling(Count() / (double)pageSize),
                PageNumber = pageNumber,
                ItemsAtPage = newsInPage
            };
        }

        public void Update(News item)
        {
            Create(item);
        }

        ICollection<News> Client.IRepository<News>.GetList()
        {
            return GetList();
        }
    }
}
```

ДОДАТОК К

ПРОТОКОЛ
ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ НА
НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи Клієнт-серверна інформаційна система підрозділу навчального закладу з можливістю розгортання в хмарному середовищі. Частина 1. «Проектування розподіленої архітектури та менеджмент з використанням гнучких методологій

Тип роботи: бакалаврська дипломна робота
(БДР, МКР)

Підрозділ кафедра обчислювальної техніки
(кафедра, факультет)

Показники звіту подібності Unicheck

Оригінальність 92,7% Схожість 7,3%

Аналіз звіту подібності (відмітити потрібне):

- ✓ Запозичення, виявлення у роботі, оформлені коректно і не містять ознак плагіату.
- □ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її виконання автором. Роботу направити на розгляд експертної комісії кафедри.
- □ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку ___________ Захарченко С.М.
(підпис) (прізвище, ініціали)

Ознайомлені з повним звітом подібності, який був згенерований системою Unicheck щодо роботи.

Автор роботи ___________ Станішевський Д. Ю.
(підпис) (прізвище, ініціали)

Керівник роботи ___________ Войцеховська О. В.
(підпис) (прізвище, ініціали)

ДОДАТОК Л

УЗГОДЖЕНО
Декан факультету інформаційних технологій та комп'ютерної інженерії, к.пед.н., доцент
_______ Світлана КИРИЛАЩУК
«___» _______ 2022 р.

ЗАТВЕРДЖУЮ
Проректор з науково-педагогічної роботи та організації освітнього процесу ВНТУ, к.т.н., доцент
_______ Олександр ПЕТРОВ
«___» _______ 2022 р.

АКТ ВПРОВАДЖЕННЯ № ____

результатів комплексної бакалаврської дипломної роботи

Замовник Вінницький національний технічний університет
(найменування організації)

Цим актом підтверджується, що результати роботи – «Клієнт-серверна інформаційна система підрозділу навчального закладу з можливістю розгортання в хмарному середовищі. Частина 1. «Проектування розподіленої архітектури та менеджмент з використанням гнучких методологій»,
що виконана студентом гр. 2КІ – 18б, ВНТУ, Станішевським Д. Ю.
(виконавець)
впроваджено у Вінницькому національному технічному університеті
(найменування організації, де здійснювалося впровадження)

1.Вид впроваджених результатів сайт кафедри обчислювальної техніки ВНТУ
(експлуатація виробу, роботи, технології)

2. Характеристика масштабу впровадження одиничне
(унікальне, одиничне, партія, масове, серійне)

3. Форма впровадження програмний продукт

4. Новизна результатів роботи модернізація старих розробок
(піонерські, принципово нові, якісно нові, модифікації, модернізація старих розробок)

5. Впроваджені: в мережі ВНТУ

6. Річний економічний ефект -

Від виконавця:
Студент групи 2КІ-18б
_______ Дмитро СТАНІШЕВСЬКИЙ

Від організації:
Завідувач кафедри обчислювальної техніки ВНТУ, д.т.н., професор
_______ Олексій АЗАРОВ

Науковий керівник
_______ доц. Олена ВОЙЦЕХОВСЬКА