МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інженерії програмного забезпечення

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри інженерії програмного забезпечення, канд. техн. наук, доцент,

_____Є.О. Давиденко

«____»_____2022 р.

КОМПЛЕКСНА КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ ЗНАНЬ. BACK-END ЧАСТИНА

Том 2

Спеціальність «Інженерія програмного забезпечення»

121 – ККРБ.2 – 409.21920801

Студент _____М. С. Кіяшко

підпис

«__» _____ 2022 р.

Керівник канд. техн. наук, доцент _____Є. О. Давиденко

підпис

«__» _____ 2022 р.

Консультант канд. техн. наук, доцент кафедри екології

__А. О. Алексєєва

підпис

«___» _____ 2022 р.

Миколаїв – 2022

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет імені Петра Могили
Факультет комп'ютерних наук
Кафедра інженерії програмного забезпечення

ЗАТВЕРДЖУЮ

Завідувач кафедри інженерії програмного

забезпечення, канд.техн.наук, доцент,

_____Є.О. Давиденко

«____»_____2022 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи бакалавра

Видано студенту групи 409 факультету комп'ютерних наук

Кіяшку Максиму Сергійовичу.
(прізвище, ім'я, по батькові студента)

1. Тема кваліфікаційної роботи

«Програмне забезпечення автоматизації тестування знань. Back-end частина»

Затверджена наказом по ЧНУ від «25» травня 2022 р. № 95

2. Строк представлення кваліфікаційної роботи «27» червня 2022 р.

3. Очікуваний результат роботи та початкові дані, якщо такі потрібні

Вхідні дані до роботи − функціональні та нефункціональні вимоги до програмного забезпечення автоматизації тестування знань. Результат − функціонуюча система автоматизації тестування знань.

4. Перелік питань, що підлягають розробці:

- аналіз принципів та підходів до тестування знань, тобто дослідження предметної області;
- огляд існуючих систем тестування знань;
- розробка вимог до системи на основі зібраної інформації про предметну область та аналоги;
- проєктування автоматизованої системи тестування знань;

– програмна реалізація серверної частини вебзастосунку для тестування знань.

5. Перелік графічних матеріалів:

Презентація.

6. Завдання до спеціальної частини

Питання охорони праці в трудовій діяльності програміста.

7. Консультанти:

Консультант	Кафедра (організація)	Частина роботи
Алєксєєва А. О.	Кафедра екології	Спеціальна частина з охорони праці

Керівник роботи канд. техн. наук, доцент, Є. О. Давиденко
(посада, прізвище, ім'я, по батькові)

_____
(підпис)

Завдання прийнято до виконання

Кіяшко Максим Сергійович
(прізвище, ім'я, по батькові студента)

_____
(підпис)

Дата видачі завдання «____» _____ 2022 р.

КАЛЕНДАРНИЙ ПЛАН
виконання кваліфікаційної роботи

Тема: «Програмне забезпечення автоматизації тестування знань. Back-end частина» .

№	Найменування роботи	Початок	Закінчення	Примітки
1.	Розробка та затвердження завдання на виконання ККРБ	03.10.2021р.	04.10.2021р.	виконано
2.	Складання календарного плану ККРБ	05.10.2021р.	09.10.2021р.	виконано
3.	Огляд літератури за темою ККРБ	10.10.2021р.	10.11.2021р.	виконано
4.	Аналіз предметної області	11.11.2021р.	20.12.2021р.	виконано
5.	Розробка проєктних рішень	21.12.2021р.	14.01.2022р.	виконано
6.	Моделювання та конструювання ПЗ	15.01.2022р.	14.03.2022р.	виконано
7.	Кодування, тестування та апробація розробленого ПЗ	15.03.2022р.	25.05.2022р.	виконано
8.	Написання спеціальної частини з охорони праці	26.05.2022р.	01.06.2022р.	виконано
9.	Відгук керівника ККРБ	02.06.2022р.	05.06.2022р.	виконано
10.	Оформлення ККРБ та презентації	06.06.2022р.	12.06.2022р.	виконано
11.	Попередній захист	13.06.2022р.	13.06.2022р.	виконано
12.	Завершення оформлення ККРБ та презентації	14.06.2022р.	20.06.2022р.	виконано
13.	Рецензування	21.06.2022р.	26.06.2022р.	виконано
14.	Захист кваліфікаційної роботи	27.06.2022р.	27.06.2022р.	

Розробив студент Кіяшко Максим Сергійович _____
(прізвище, ім'я, по батькові студента) (*підпис*)

«____» _____ 2022 р.

Керівник роботи канд. техн. наук, доцент, Є. О. Давиденко _____
(посада, прізвище, ім'я, по батькові) (*підпис*)

«____» _____ 2022 р.

АНОТАЦІЯ

до комплексної кваліфікаційної роботи бакалавра

«Програмне забезпечення автоматизації тестування знань. Back-end частина»

Студент 409 гр.: Кіяшко Максим Сергійович

Керівник: зав. кафедри ІПЗ, канд. техн. наук, доцент Давиденко Є. О.

Даний том комплексної роботи спрямований на розробку серверної частини застосунку автоматизації тестування знань, принципи роботи якого розглянуто у загальній частині дипломного проєкту.

Об'єкт: процес проходження тестування для перевірки знань в закладах освіти.

Предмет: засоби автоматизації процесу тестування, підрахунку та аналізу результатів.

Мета роботи полягає у підвищенні ефективності перевірки знань здобувачів освіти за рахунок зменшення кількості помилок при перевірці тестів, підвищення швидкості та точності підрахунку результатів шляхом розробки автоматизованої системи тестування.

Даний том складається зі вступу, двох розділів висновків та додатків, а також спеціальної частини «Охорона праці».

У вступі викладається актуальність теми, мета та поставлені завдання щодо реалізації back-end частини застосунку.

В першому розділі обґрунтовується вибір технологій для вирішення задачі, наводиться їх опис. Другий розділ охоплює процес створення серверної частини застосунку з використанням мови програмування C# та слабко зв'язаної архітектури.

У висновках проводиться аналіз роботи та отриманих результатів.

В спеціальній частині «Охорона праці» розглянуто питання безпеки праці в ході процесу розробки програмного забезпечення.

Том ККРБ викладений на 34 сторінки, містить 2 розділи, 29 ілюстрацій, 2 таблиці, 9 джерел в переліку посилань.

Ключові слова: *тестування знань, автоматизована система, C#, слабко зв'язаний моноліт, база даних SQL.*

ABSTRACT

of the Bachelor's Thesis

"Software for automation of knowledge testing. Backend"

Student of group 409: Kiiashko Maksym Serhiiovych

Supervisor: Deputy Dean, Associate Professor of the Department of software engineering Davydenko Ye. O.

This volume of the complex work is aimed at the development of the server side of the application for automating knowledge testing, the operating principles of which are discussed in the common volume of this work.

The object of the work is the process of test taking for the assessment of knowledge in education institutions.

The subject of the work are the means for automating the process of testing as well as calculating and analyzing test results.

The purpose of this work is to increase the efficiency of knowledge assessment of students by reducing the number of errors in checking tests, increasing the speed and accuracy of calculating results through the development of an automated testing system.

This volume consists of an introduction, three sections, conclusions and appendixes, as well as a special “Workplace safety” section.

In the first section the choice of technologies for implementing the system is justified and those technologies are described. The third section encompasses the process of developing the server side of the application using the C# programming language and loosely coupled architecture.

In the special section titled “Workplace safety” the questions of the safety of labor during the process of software development are discussed.

In the conclusion we analyze the work and the obtained results.

This volume is 34 pages long and includes 2 sections, 29 illustrations, 2 tables, and 9 sources.

Keywords: *knowledge assessment, automated system, C#, Loosely Coupled Monolith, SQL database.*

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

БД – база даних

ПЗ – програмне забезпечення

ACID − Atomicity, Consistency, Isolation, Durability

API − Application Programming Interface

DTO − Data Transfer Object

GUI − Graphical User Interface

HTTP − HyperText Transfer Protocol

IIS − Internet Information Services

JSON − JavaScript Object Notation

LCM − Loosely Coupled Monolith

NF – Normal Form

ORM − Object-Relational Mapping

REST − Representational State Transfer

UI − User Interface

URI − Uniform Resource Identifier

ВСТУП

Система для автоматизації тестування знань, що розроблюється, має клієнт-серверну архітектуру. Серверна частина відповідає за зберігання та обробку даних користувача. Клієнтська частина забезпечує взаємодію із користувачем через графічний інтерфейс та обмін даними із сервером.

Back-end розробка є перспективним видом діяльності зі значним попитом на сучасному ринку праці та високою заробітною платою. Отже, освоєння серверних технологій є актуальним.

Якісна реалізація back-end частини застосунку є вкрай важливою для успіху програмного продукту, адже саме на неї покладається відповідальність за імплементацію основної бізнес-логіки. Серверна частина складається із бази даних та застосунку на базі фреймворку ASP .NET Web API, який відповідає на запити клієнта.

Для створення back-end частини системи необхідно виконати наступні завдання:

- розглянути поняття клієнт-серверної архітектури, її особливості, переваги та недоліки;
- порівняти наявні технології створення back-end, здійснити та обґрунтувати вибір технології розробки;
- здійснити огляд фреймворку ASP .NET Web API, його структури та принципів роботи, пояснити основні поняття, пов’язані з технологією;
- здійснити проєктування бази даних;
- здійснити програмну реалізацію та тестування системи.

Результатом роботи має бути готова back-end частина застосунку для автоматизації тестування знань.

1 КЛІЄНТ-СЕРВЕРНІ ТЕХНОЛОГІЇ

Клієнт-сервер – це комп'ютерна модель, яка працює наступним чином: клієнт відправляє запит на отримання якихось даних, ці дані надходять до сервера, сервер оброблює ці дані та надає відповідь клієнту.

Зазвичай клієнт та сервер обмінюються повідомленнями між собою через мережу на різних пристроях. Таким чином, разом працюють різні програми, в зборі дають велику монолітну систему. Наприклад, програма, написана під мобільний пристрій, відправляє запити до програми-серверу. Також може бути написана програма під браузер, або десктоп-версія, і кожна з цих програм буде взаємодіяти з програмою, яка буде давати відповіді на їх запити.

Сервер надає функціонал, який використовують один або багато клієнтів, які роблять запити до сервера. Зазвичай, сервер працює зі сховищем даних, де зберігається інформація, яка стосується системи в цілому, а також дані клієнтів, які вже працювали з системою. Клієнт при вході в систему також зберігає деякі дані, але ці дані, зазвичай, стосуються того клієнта, який в даний момент працює з системою. При вході в систему клієнтом створюється нова сесія, яка активна до моменту виходи клієнта зі системи.

Основні властивості клієнта:

- відправляє запити серверу;
- отримує відповіді від серверу у форматі, який клієнт здатний обробити;
- зв'язаний з невеликою кількістю серверів, часто з одним;
- взаємодіє напряму з користувачем через GUI.

Основні властивості серверу:

- отримує запити від клієнта та відповідає на них;
- взаємодіє зі сховищем даних;
- взаємодіє з іншими серверами.

Переваги клієнт-серверної архітектури [6]:

- всі дані зберігаються в одному сховищі даних, що забезпечує безпеку збереження даних;

- так як дані зберігаються в одному місці, та мають централізоване управління з боку сервера, їх простіше змінювати, модифікувати, знаходити потрібну інформацію в сховищі даних, навіть якщо система зберігає в собі велику кількість даних;
- можна написати багато варіацій клієнтів під різні пристрої, які працюватимуть з одним і тим самим сервером без зміни програми самого серверу.

Недоліки клієнт-серверної архітектури:

- сервер може не витримати велику кількість запитів одночасно та перезавантажитися;
- клієнтський застосунок не буде працювати, якщо сервер не відповідає на запити.

Для розробки серверу було обрано ASP.NET Web API Framework, середовище програмування Microsoft Visual Studio 2022.

1.1 ASP.NET Web API Framework

Для написання серверної частини системи обрано ASP.NET Web API фреймворк.

1.1.1 Загальні відомості

ASP .NET створено компанією Microsoft для створення динамічних вебсторінок. Фреймворк є наступником класичних застосунків ASP (Active Server Pages). Технологія була значно вдосконалена порівняно з попередником. В таблиці 1 наведені відмінності між ними.

Таблиця 1.1 – Відмінності між технологіями ASP та ASP .NET

ASP	ASP .NET
Інтерпретована мова (VBScript)	Компільована мова (C#)
Використовує ADO (ActiveX Data Objects) для з’єднання та роботи з БД	Використовує ADO.NET для з’єднання та роботи з БД

Кінець таблиці 1.1

ASP	**ASP .NET**
Частково об’єктно-орієнтована	Повністю об’єктно-орієнтована
Погана обробка помилок	Якісна обробка помилок
Складне налагодження (бо використовується інтерпретована мова)	Значно легше налагодження
Немає можливості задання корфігурації	Конфігурація відбувається через файл Web.config
Має всього 4 вбудовані класи: Request, Response, Session, Application	Має більш ніж 2000 вбудованих класів

Отже, варто розрізняти ці дві технології. В порівнянні з ASP, фреймворк ASP .NET було значно покращено, завдяки чому він повністю відповідає сучасним вимогам веброзробки.

На сьогоднішній день ASP .NET є популярною технологією створення вебсторінок. На рисунку 1.1 представлено кількість сайтів різних категорій, розроблених на ASP .NET у порівнянні із його головним конкурентом – PHP-фреймворком Laravel. ASP .NET значно випереджає Laravel за часткою ринку та частотою використання.

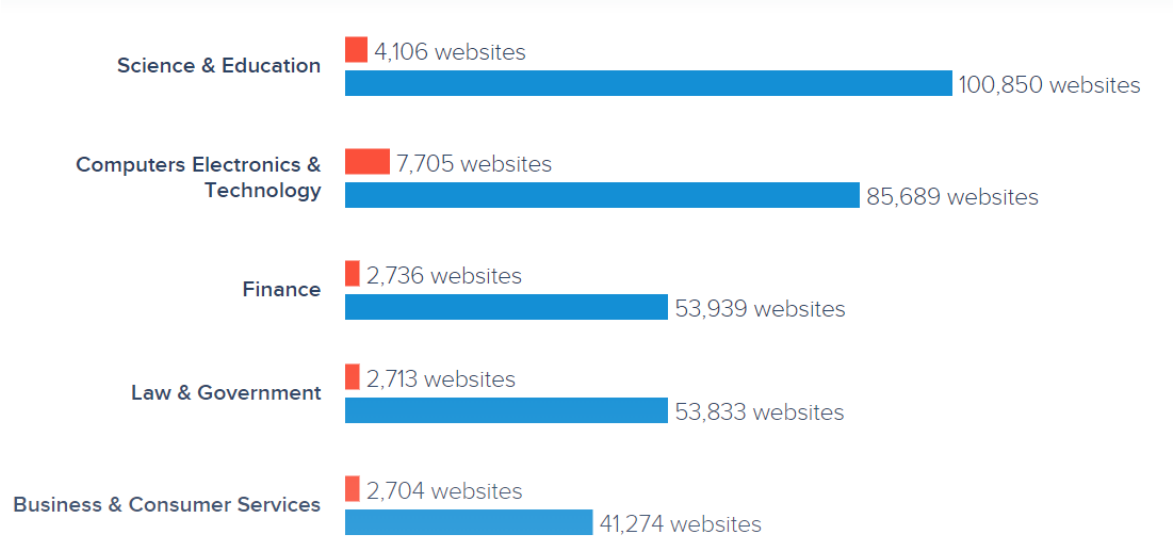

Рисунок 1.1 – Кількість вебсайтів на ASP .NET (блакитним) та Laravel (оранжевим)

Перевагами фреймворку ASP .NET є:

- чудова масштабованість;
- виявлення помилок підчас компіляції;
- велика кількість додаткових інструментів розробки;
- бездоганно працює на платформі Windows;
- технологія стабільно підтримується та вдосконалюється компанією Microsoft;
- велике ком'юніті розробників в зв'язку зі значною популярністю технології.

В таблиці 2 наведено порівняння ASP .NET з іншим популярним вебфреймворком Laravel.

Таблиця 1.2 – Порівняння технологій Laravel та ASP .NET

	Laravel	**ASP .NET**
Розмір проєкту	Підходить для малих та середніх проєктів	Підходить для середніх та великих проєктів більшої складності

Кінець таблиці 1.2

	Laravel	**ASP .NET**
Код	Інтерпретований (PHP)	Компільований (C#)
Масштабованість	Добра (горизонтальна)	Добра (вертикальна)
Безпека	Менш захищена	Дуже захищена
Особливості	Найбільша увага приділяється клієнтському та користувацькому інтерфейсам	Найбільша увага приділяється захищеності та функціональності
Швидкість	Дещо повільніша	Дещо швидша
Вартість	Низька (open-source)	Висока (власність Microsoft)
Легкість вивчення	Легше опанувати	Складніше опанувати

Основними недоліками ASP .NET в порівнянні з Laravel є висока ціна та більша складність вивчення. Проте це компенсується більшою швидкістю та захищеністю. Висока безпека досягається завдяки вбудованим функціям, що захищають програми від атак типу cross-site scripting (XSS) та cross-site request forgery (CSRF) [1]. Також фреймворк надає вбудовану БД користувачів з підтримкою багатофакторної аутентифікації. Вагомою перевагою ASP .NET є використання мови C# та можливість виявлення помилок на етапі компіляції. Отже, для розробки застосунку було зроблено вибір на користь ASP .NET.

ASP .NET є загальним терміном, що включає в себе декілька технологій. Розглянемо різницю між ASP .NET MVC та ASP .NET Web API.

ASP .NET MVC використовується для створення вебзастосунків, що повертають і представлення, і дані. З іншого боку, ASP .NET Web API використовується для створення повноцінних HTTP-сервісів, що повертають лише дані, але не представлення. Web API дозволяє створювати REST-ful сервіси

та підтримує механізм узгодження вмісту (content negotiation), завдяки якому сервер може «домовитися» із клієнтом про найкращий формат відповіді – JSON, XML, ATOM тощо [3]. Формат повернення даних можна задати у заголовку (header) в запиті клієнта. На відміну від Web API, ASP MVC може повертати дані лише у форматі JSON. Web API має легку архітектуру, окрім вебзастосунків технологію можна використовувати для мобільних застосунків.

Для створення застосунку, що розробляється, найкраще підходить саме ASP .NET Web API. Застосунок має клієнт-серверну архітектуру, тому представлення будуть реалізовані за допомогою окремої front-end технології. Завданням back-end є обробка запитів та надання даних клієнту через HTTP.

1.2 Принцип роботи

За допомогою даного фреймворку створюються сервери на основі HTTP, до яких можна звернутися з різних платформ: браузер, операційна система Windows, мобільні пристрої.

Властивості ASP.NET Web API:

- побудований на основі ASP.NET;
- підтримує конвеєр запитів/відповідей ASP.NET;
- платформа для створення RESTful служб;
- відповіді на запити у різних форматах (JSON, XML, BSON);
- зіставляє імена методів з назвами HHTP запитів;
- може бути розгорнутий на власному хості, на IIS, на іншому сервері [8].

На рисунку 1.2 показана взаємодія серверу, написаному на ASP.NET Web API, з клієнтом.

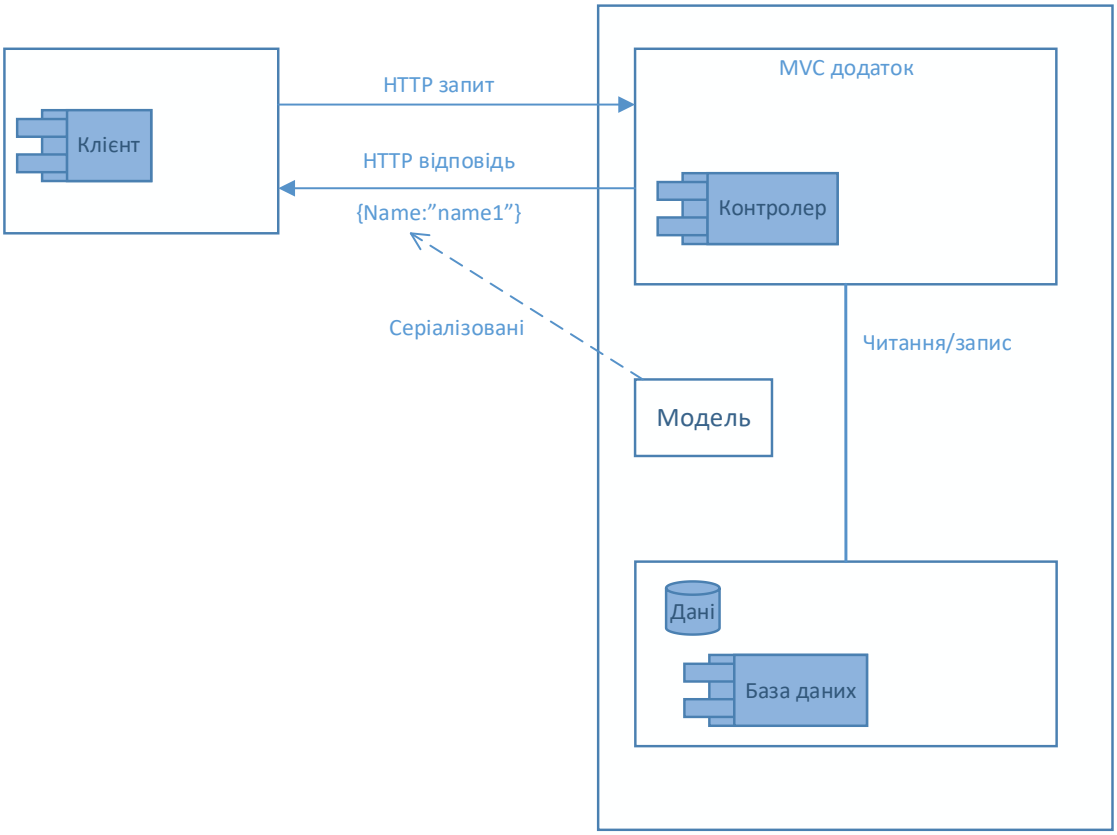

Рисунок 1.2 – Шаблон ASP.NET Web API

Клієнт надсилає запит серверу, контролер серверу виконує операції з читання та запису даних в сховищі даних, виконує маніпуляції з даними, та надає відповідь у вигляді серіалізованих даних.

Основними компонентами системи є:

1. Модель – набір класів, які представляють дані зі сховища даних та керуються контролерами.

2. Контекст бази даних – основний клас, який є похідним від класу Microsoft.EntityFrameworkCore.DbContext та координує функціональність Entity Framework для моделі даних.

3. Контролер – клас, який оброблює HTTP запити. Методи цього класу оброблюють GET, POST, PUT, DELETE та інші запити, та надають відповідь на той чи інший запит.

1.1.1 Атрибути та маршрутизація ASP.NET Web API

Маршрутизація – це зіставлення URI з певною дією. ASP.NET Web API 2 підтримує маршрутизацію через атрибути. Таким чином, через атрибути визначаються маршрути.

Інший варіант маршрутизації в ASP.NET Web API – на основі конвенцій. Даний вид маршрутизації використовує шаблони маршрутів, які додаються до таблиці маршрутів, щоб обробляти різні HTTP-запити. Перевагою даного методу маршрутизації є те, що шаблони маршрутів визначаються в одному місці. Але маршрутизація на основі конвенцій ускладнює підтримку певних шаблонів URI.

Тому при розробці системи маршрутизація описана через атрибути. Для цього додано атрибут Route перед визначення класу контролера та методів класу контролера. В параметрах зазначений відповідний шлях, який зіставляється з URI.

За замовчуванням Web API обирає тип запиту (GET, POST, PUT тощо) через відповідність без урахуванням регістру початку назви методу контролера. Але наочніше для читання коду контролера використовувати атрибути, які вказують на тип запиту. В такому випадку можна називати метод назвою без прив’язки до типу запиту. До таких атрибутів відносяться:

- [HttpDelete];
- [HttpGet];
- [HttpHead];
- [HttpOptions];
- [HttpPatch];
- [HttpPost];
- [HttpPut].

З назви атрибуту зрозуміло, який тип HTTP запиту виконуватиметься. У роботі було використано описаний вище вид атрибутів. Наприклад, метод CreateTest() позначений атрибутом HttpPost.

У ПЗ використано конвенції Web API. Конвенції надають змогу визначати найпоширеніші типи, які повертає запит, та коди стану, які вказують на успіх або невдачу виконання того чи іншого запиту. Також конвенції визначають дії, які відхиляються від визначеного стандарту.

Для застосування конвенції до методу використовується атрибут [ApiConventionMethod] – вказує тип конвенції та метод конвенції, який застосовується. Для того, щоб вказати тип конвенції для всіх методів контролера є атрибут [ApiConventionType], який вказується перед оголошенням класу контролера. Також даний атрибут можна застосувати до всіх контролерів поточної збірки.

Для того, щоб вказати типи відповідей на запит, методи мають бути анотовані атрибутами [ProducesResponseType] або [ProducesDefaultResponseType]. Атрибути мають параметри зі статусом відповіді, типом даних, який буде переданий при відповіді на запит, та додаткові параметри.

Коди статусу відповіді описані у класі Status Codes. Вони поділені на п’ять основних груп.

1. Інформаційні відповіді (100-199) – вказують на попередню відповідь, яка складається з рядка стану та необов’язкових заголовків.
2. Успішні відповіді (200-299) – запит виконано успішно.
3. Повідомлення про переспрямування (300-399) – вказує на те, що клієнту потрібно виконати запропоновані дії, щоб виконати запит.
4. Відповіді на помилки клієнта (400-499) – помилка при виконанні запиту від клієнта (неавторизований користувач, вийшов час очікування, неправильний запит тощо).
5. Відповіді на помилки сервера (500-599) – помилка сервера, сервер не може обробити запит.

Ще однією складовою запиту є параметри запиту. Якщо запит містить параметри, то при виклику методу контролера, Web API встановлює значення для параметрів. Даний процес називається прив'язкою даних.

За замовчуванням, якщо параметр є простим типом (int, double, long тощо), то WebAPI намагається отримати значення з URI. Для того, щоб вказати, що параметр буде зчитуватись з URI строки, необхідно додати анотацію параметру у вигляді атрибута [FromQuery]. Для параметрів складних типів значення зчитується з тіла повідомлення. Для позначення такого параметру необхідно додати анотацію атрибутом [FromBody].

На рисунку 1.3 наведений приклад оголошення методу контролера.

```
[HttpPut]
[ApiConventionMethod(typeof(DefaultApiConventions), nameof(DefaultApiConventions.Put))]
[Route("users/{userid}/tests/")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status500InternalServerError)]
Ссылок: 0
public async Task<ActionResult> PutTest(string userID, [FromBody] Test updatedTest)
```

Рисунок 1.3 – Оголошення методу PutTest

Даний метод містить наступні атрибути:

- HttpPut – атрибут вказує, що це запит на оновлення даних;
- Route – задає маршрутизацію, тобто шлях до запиту;
- ApiConventionMethod – вказує тип конвенції та метод конвенції;
- ProducesResponseType – можливий статус відповіді (успіх, помилка клієнта, помилка сервера);
- FromBody – вказує на те, що параметр updatedTest типу Test зчитуватиметься з тіла повідомлення.

Таким чином, атрибути є важливою частиною при оголошенні методів для обробки запитів. За допомогою атрибутів можна зчитати більшість інформації про метод: тип запиту, шлях, тип та спосіб передачі параметрів тощо.

1.1.2 Авторизація та аутентифікація у ASP.NET Web API

Більшість функціоналу системи доступне лише тільки тим клієнтам, які пройшли аутентифікацію. Виконання того чи іншого запиту буде успішним, якщо цей запит надійшов від авторизованого користувача.

Аутентифікація – ідентифікація особи користувача. Тобто при вході в систему користувач вводить своє ім’я та пароль, а система використовує ці дані для того щоб аутентифікувати користувача. Web API передбачає, що аутентифікація відбувається на хості. Для веб-хостингу хостом є IIS, який використовує HTTP модулі для аутентифікації.

В системі реалізована аутентифікація на основі токенів. При вході в систему за користувачем закріплюється згенерований токен. Таким чином користувачу не потрібно вводити свої дані кожного разу для виконання певної дії. На рисунку 1.4 представлений метод генерації токену:

```
JwtSecurityToken jwt = new JwtSecurityToken(
        issuer: issuer,
        audience: audience,
        notBefore: now,
        claims: identity.Claims,
        expires: now.Add(TimeSpan.FromMinutes(lifetime)),
        signingCredentials: new SigningCredentials(symmetricSecurityKey, SecurityAlgorithms.HmacSha256));

string encodedJwt = new JwtSecurityTokenHandler().WriteToken(jwt);

Token token = new Token(identity.Name, user.ExternalId, encodedJwt);
return Ok(token);
```

Рисунок 1.4 – Генерація токену

Аутентифікація на основі токенів проходить в п’ять етапів:

1. Запит – користувач вводить свої облікові дані, надсилається запит до серверу.
2. Перевірка – перевірка інформації, яку надав користувач.
3. Подання токену – сервер генерує підписаний токен.
4. Зберігання – токен закріплюється за користувачем на стороні клієнта для збереження доступу користувача до сайту.
5. Термін дії – токен активний до того моменту, поки користувач не вийду з системи.

Авторизація – визначає дозвіл користувача на виконання певної дії. Наприклад, користувач хочу редагувати раніше створений тест. Сервер спочатку перевіряє, чи має право користувач на редагування конкретного тесту, а вже потім виконує маніпуляції з тестом.

Авторизація відбувається перед дією контролера. Фільтри авторизації перевіряють доступ користувача щодо конкретного запиту та надають відповідь. Якщо запит не авторизований, то дія викликатися не буде.

На рисунку 1.5 показана послідовність авторизації та аутентифікації користувача:

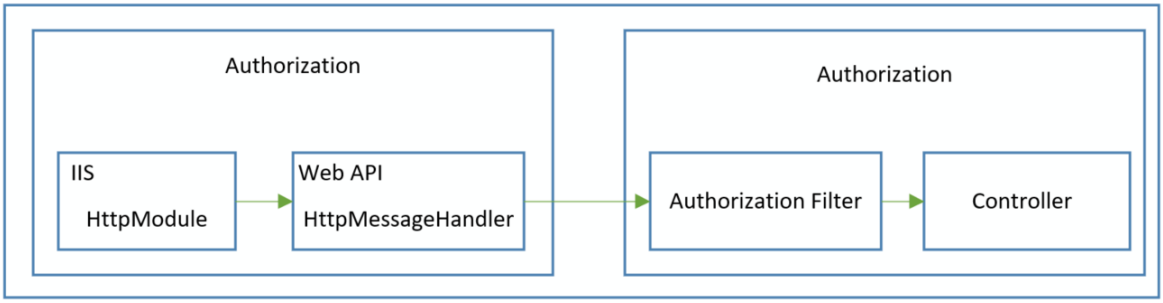

Рисунок 1.5 – Авторизація та аутентифікація

В Web API є вбудований фільтр авторизації, AuthorizeAttribute. Фільтр перевіряє, чи пройшов автентифікацію користувач, і у випадку невдачі, повертає код статусу 401 та не виконує дій, які запитує користувач. Даний фільтр визначається через анотацію контролера або методу контролера атрибутом [Authorize] (рисунок 1.6).

```
[ApiVersion("1.0")]
[Authorize]
[ApiController]
[Route("results-management")]
Ссылок: 2
public class ResultsController : ControllerBase
```

Рисунок 1.6 – Оголошення контролера

Таким чином, всі методи контролеру будуть проходити перевірку автентифікації перед тим як виконувати ту чи іншу дію.

У випадку, коли один із методів повинен бути доступний і для неавторизованих користувачів, необхідно додати анотацію методу атрибутом [AllowAnonymus]. В системі неавторизований користувач має можливість пройти публічний тест. Тому запит на отримання даний тесту анотовано відповідним атрибутом:

```
[HttpGet]
[AllowAnonymous]
[ApiConventionMethod(typeof(DefaultApiConventions), nameof(DefaultApiConventions.Get))]
[Route("tests/{testId}")]
[ProducesResponseType(StatusCodes.Status200OK, Type = typeof(IEnumerable<Test>))]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
Ссылок: 0
public async Task<ActionResult<IEnumerable<Test>>> GetTest(string testId)
```

Рисунок 1.7 – Оголошення методу

Таким чином, система розрізняє можливості неавторизованого та авторизованого користувача.

1.1.3 Впровадження залежності у ASP.NET Web API

Впровадження залежності допомагає створювати більш зручніший для підтримки код. Даний підхід вирішує проблему залежності, коли один компонент дуже тісно залежить від іншого. Така тісна залежність компонентів один з одним тягне за собою складність обслуговування коду, що впливає на вартість робіт [7].

Впровадження принципу інверсії залежності робить проблему залежності більш керованою. Цей принцип базується на тому, що:

- модулі як високого, так і низького рівня повинні залежати від абстракцій, а не один від одного;
- деталі повинні залежати від абстракцій, а не навпаки.

На рисунку 1.8 схематично показано принцип інверсії залежності:

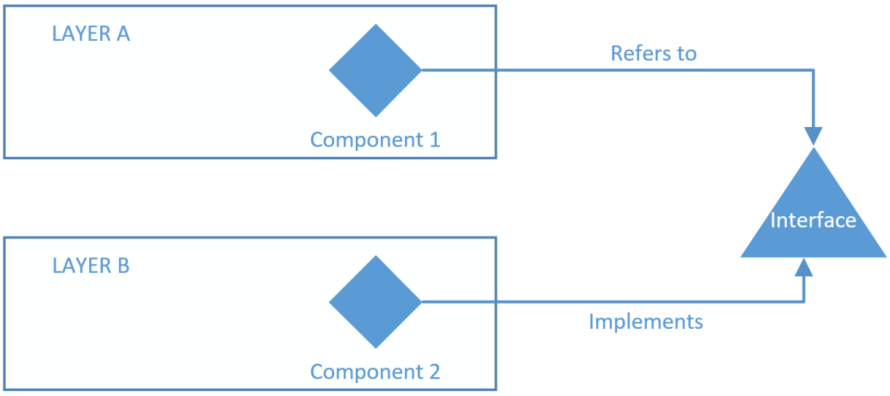

Рисунок 1.8 – Інверсія залежності

Зв'язок між компонентами відбувається через абстракцію у вигляді інтерфейсу. В системі кожна сутність реалізована з використанням інверсії залежності. Наприклад, сутність «Результати» має свою модель з даними, а також клас репозиторію, який маніпулює даними моделі, використовуючи Entity Framework. На рисунку _ показаний клас репозиторію:

```
Ссылок: 2
public class ResultRepository : IResultRepository
{
    private readonly ResultDBContext resultDBContext;
    Ссылок: 0
    public ResultRepository(ResultDBContext resultDBContext)
    {
        this.resultDBContext = resultDBContext;
    }
```

Рисунок 1.9 – Клас репозиторію

Клас репозиторію зв'язаний з класом контролера через інтерфейс IResultRepository (рисунок 1.10).

```
public class ResultsController : ControllerBase
{
    private readonly IConfiguration configuration;
    private readonly IResultRepository resultRepository;
```

Рисунок 1.10 – Клас контролера

Такий підхід надає багато переваг при подальшій підтримці коду:

- якщо потрібно буде замінити ResultRepository іншою реалізацією, не потрібно буде змінювати клас контролеру;
- не потрібно налаштовувати залежності класу репозиторію всередині контролера;
- дає змогу швидко і без змін класу контролера провести модульне тестування, так як для цього достатню змінити посилання на фіктивний клас репозиторію або репозиторій-заглушку.

Таким підхід до написання системи робить код зручним для його подальшої підтримки, тестування.

1.2 Опис API системи за допомогою Swagger

Для того, бачити список запитів, параметрів до них, в систему був інтегрований Swagger. Це зручний інструмент, який допомагає описувати API. За допомогою даного інструменту можна автоматично створювати зручну та інтерактивну документацію API. Також можливо створювати бібліотеки для API багатьма мовами, а також проводити автоматичне тестування.

На рисунку 1.11 показана інтеграція в систему Swagger UI.

```
app.UseSwagger();
app.UseSwaggerUI();
```

Рисунок 1.11 – Підключення Swagger UI

Swagger UI візуалізує ресурси OpenAPI і взаємодію з ними без відображення логіки реалізації. Таким чином, за допомогою даного інструменту є можливість виконувати різні запити для тестування API.

Переваги Swagger UI:

- інтерфейс доступний незалежно від середовища розробки;
- підтримується будь-яким браузером;
- зручність у використанні дозволяє показати клієнту всі операції, які надає API;
- є можливість налаштувати інтерфейс, багато параметрів налаштувань;

– простий у навігації, дані представлені у вигляді списку, який поділений на категорії.

На рисунку показано, як виглядає Swagger UI до системи, що розроблюється:

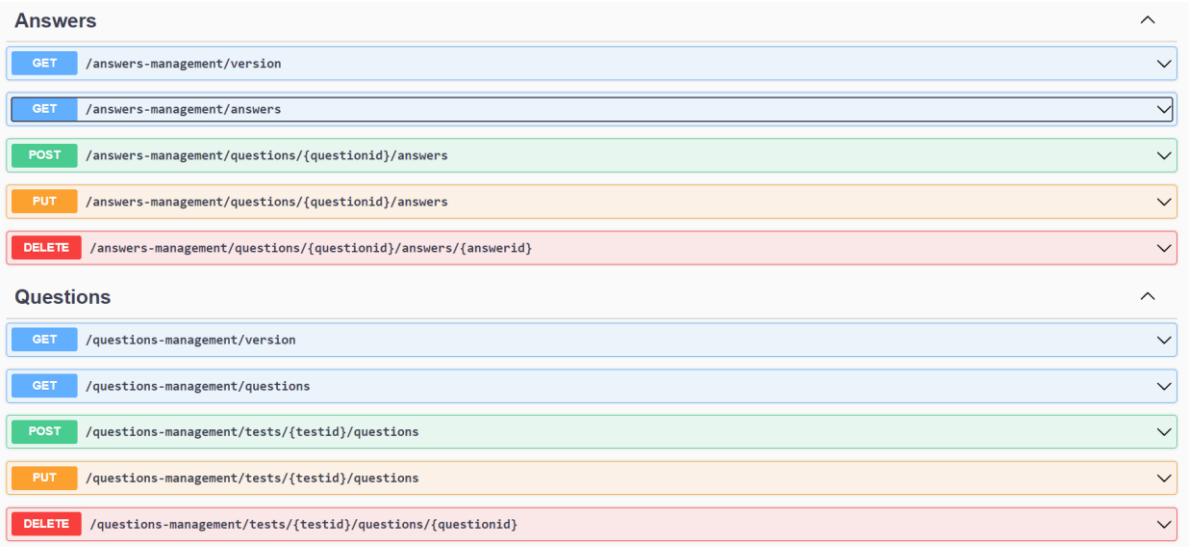

Рисунок 1.12 – Swagger UI

Показано всі запити, які надає API. Запити згруповані за групами, та описаний тип запиту. По кожному запиту можна дізнатися детальну інформацію. Таким чином, даний інструмент дозволяє економити час розробникам при розробці документації, тестуванні системи, та опису API іншим розробникам та клієнтам.

1.3 Проєктування баз даних

Правильно розроблена база даних надає доступ до актуальної, точної інформації. Правильно спроєктована база даних має велике значення для досягнення цілей при розробці системи, адже це напряму впливає на швидкість роботи системи, зручність при розробці.

Основні принципи при розробці бази даних [9]:

– уникнення дублювання даних, розподілення інформації таким чином, щоб зменшити зайві дані;
– забезпечення доступу до потрібної інформації;
– точність і цілісність інформації;

- повнота інформації: дані, присутні у таблицях, повинні задовольняти потреби при генерації звітів та при обробці даних.

Правильно спроєктована база даних є одним із гарантів хорошої системи. Тому при розробці бази даних необхідно ретельно проаналізувати дані, які будуть зберігатися в базі, та дотримуватись наступних кроків:

1. визначити призначення БД;
2. знайти і впорядкувати інформацію, яка призначена для зберігання у БД;
3. розподілити інформацію по таблицям;
4. перетворити елементи інформації на стовпці;
5. визначити первинні ключі;
6. налаштувати зв’язки між таблицями;
7. провести аналіз створеної БД;
8. провести нормалізацію.

Важливою складовою правильно побудованої БД є застосування правил нормалізації. Вона проводиться поступово по ходу проєктування БД. На кожному з етапів потрібно зважати на досягненні вимог однієї з «нормальних» форм. Загальноприйнятними є п’ять нормальних форм (NF) – від першої до п’ятої нормальної форми [4].

Перша нормальна форма вимагає, щоб на кожному перетині рядків і стовпців існувало одне значення. У системі є таблиця тест, та таблиця користувачів. Так як не можна помістити в одне поле значення всіх підписників на тест, то було створено додаткову таблицю, в якій кожен рядок таблиці описує конкретного підписника на тест.

Друга нормальна форма вимагає, щоб кожен неключовий стовпець залежав від усього первинного ключа, а не лише від частини ключа.

Третя нормальна форма вимагає, щоб неключові стовпці були незалежними один від одного.

Зважаючи на всі ці правила, спроєктовано БД до системи, яке відповідає всім вимогам нормальних форм.

1.3.1 Схема бази даних

Схема БД показує, як організовані дані в реляційній БД, а саме:

- імена таблиць;
- поля;
- типи даних;
- зв’язки між об’єктами.

Схема БД є дуже важливою як для розробників БД, так і для клієнта. Якщо система розвивається, то цілком природньо, що до такої системи будуть додаватися нові функції, та для цього потрібні будуть створені нові сутності. Маючи схему даних, набагато легше знайти спосіб інтегрувати нові сутності в систему, та зв’язати їх з існуючими.

Переваги схем бази даних наступні:

- доступ та безпека: розробка схеми даних допомагає організувати дані в окремі об’єкти, що полегшує спільний доступ до однієї схеми в іншій БД;
- організація та комунікація: документування схем БД дозволяє організовано спілкуватися як розробникам між собою, так і замовнику. Схема БД дає змогу розуміти обмеження, методи агрегації між таблицями;
- цілісність: за допомогою схеми БД можливо керувати процесом нормалізації, а також слідкувати за відповідністю обмеженням у дизайні БД, забезпечуючи дотримання властивостей ACID.

Всього система має 6 сутностей.

1. Users – містить повну інформацію про користувача.

2. Answers – містить інформацію про відповіді, до якого питання відноситься.

3. Questions – інформація щодо питань тесту, а також до якого тесту відноситься.

4. Tests – інформація щодо тесту, його власника, предмету тощо.

5. Subscribers – містить дані щодо того хто з користувачів підписаний на той чи інший тест.

6. Results – загальні дані щодо результатів конкретного користувача: скільки тестів він пройшов, з якою точністю тощо.

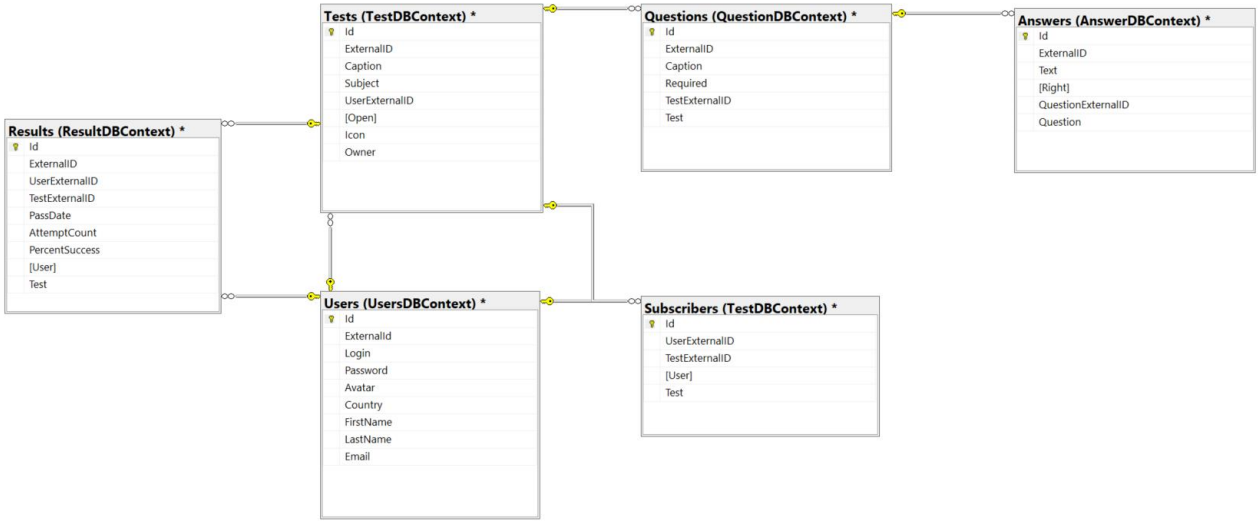

Рисунок 1.13 – Схема БД

Таким чином, система зберігає інформацію у вигляді, який зрозумілий у перегляді, швидкий у пошуку потрібної інформації, та буде орієнтиром при розширені системи для включення нових сутностей.

Висновки до розділу 1

В першому розділі кваліфікаційної роботи розглянуто технології, які були задіяні при розробці системи, визначено їх переваги та недоліки. Обґрунтовано причини вибору клієнт-серверної архітектури системи та фреймворку розробки ASP .NET Web API. Розглянути деякі важливі поняття фреймворку. Здійснено проєктування бази даних відповідно до вимог нормалізації залежностей, наведено її схему та опис.

Відомості, наведені у розділі, є основою для подальшої програмної реалізації серверної частини застосунку.

2 ПРОГРАМНА РЕАЛІЗАЦІЯ СЕРВЕРНОЇ ЧАСТИНИ ДОДАТКУ

2.1 Розробка структури системи

Важливим елементом для успішної подальшої підтримки та розширенню програмного додатку є грамотно написана структура програмного забезпечення. На етапі створення системи потрібно зважати на те, що в подальшому система може бути розширена, мати додатковий функціонал, а також може бути модифікована (зміна джерела даних тощо).

Система, яка передбачає легку модифікацію, підтримку коду, готова до масштабованості без значних змін у проєкті буде у набагато вигіднішій ситуації, ніж система, у якій об’єкти мають тісні зв’язки між собою. У першому випадку розробники будуть у виграшній ситуації з точки зору часу та ресурсів, які необхідно спрямувати на роботу з проєктом. У другому випадку розробники матимуть великі проблеми, адже зміна одного елементу системи буде тягнути за собою зміни цілого модуля, а можливо й системи в цілому.

Тому тісно зв’язані системи приречені на програш на ринку, адже чим швидше система оновлюється та підтримує нові технології, чим менший ресурс використовується на такі маніпуляції, тим більша ймовірність, що продукт буде користуватися попитом.

Тому при розробці системи продумані зв’язки між об’єктами, їх взаємодія.

2.1.1 Loosely Coupled Monolith патерн в основі системи

При написанні проєкту за основу було взято слабко зв’язаний моноліт. Суть патерну в тому, щоб розділити функціонал системи на окремі підпроєкти, що надає змогу розмежувати імплементацію модулів системи. Проєкти обмінюються між собою повідомленнями і таким чином підтримують зв’язок один з одним.

Кожен контекст поділений на три проєкти, які призначені для конкретних цілей (рисунок 2.1).

Рисунок 2.1 – Структура проєкту

1. Contracts – в даному проєкті знаходяться інтерфейси, делегати, об’єкти передачі даних (DTO).
2. Implementation – в даному проєкті міститься вся реалізація контексту.
3. Tests – даний проєкт містить в собі тести для перевірки правильності реалізації функціоналу контексту.

Одним із головних контекстів системи є контекст «Тест». Структура проєкту виглядає наступним чином:

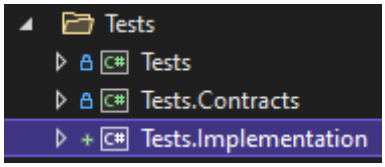

Рисунок 2.2 – Проєкт Tests

Всі інші контексти мають відповідні проєкти, в яких міститься функціонал навколо конкретного контексту. В системі є ще декілька контекстів (питання, відповідь, рейтинг, результати тощо). Реалізація кожного з контекстів відокремлена від реалізації інших обмежених контекстів. Для того, щоб контексти взаємодіяли між собою, проєкт з реалізацією контексту може мати посилання на проєкт з контрактами іншого проєкту, тому що в проєкті з контрактами прописані повідомлення, події, DTO.

Оскільки кожен контекст є власником власних даних, не обов’язково всім контекстам системи посилатись на одну базу даних. Кожен контекст має свою

конфігурацію, яка передається конкретному контексту при його створенні (рисунок 2.3).

```
Ссылок: 9
public class TestDBContext : DbContext
{
    Ссылок: 5
    public DbSet<Test> Tests { get; set; }
    Ссылок: 0
    public DbSet<Subscribers> Subscribers { get; set; }

    private readonly IConfiguration configuration;
    Ссылок: 0
    public TestDBContext(DbContextOptions<TestDBContext> options, IConfiguration configuration) : base(options)
    {
        this.configuration = configuration;
    }
```

Рисунок 2.3 – Передача параметрів та конфігурації

Отже, якщо для одного контексту буде змінена конфігурація, то це не вплине на роботу інших контекстів.

Загалом структура системи виглядає наступним чином (рисунок 2.4).

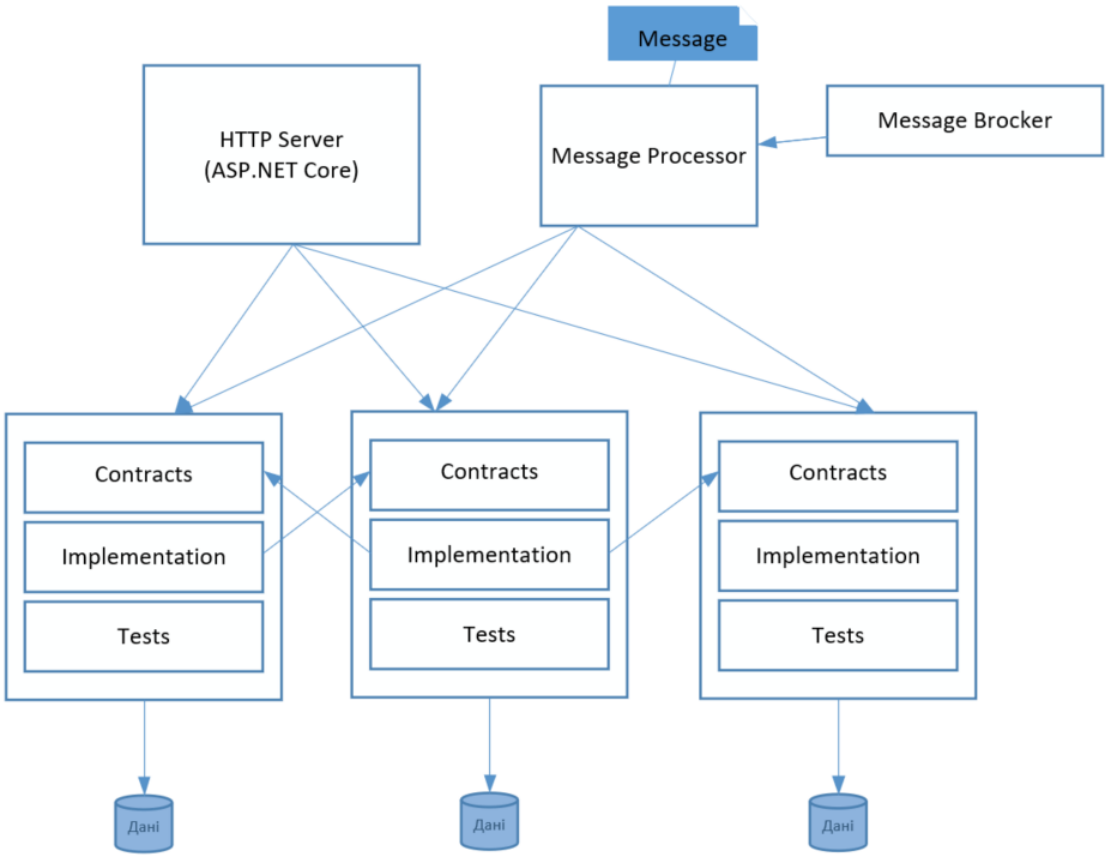

Рисунок 2.4 – Структура системи

Проєкт ASP.NET Core є хостом. Даний проєкт має посилання на всі класи обмеженого контексту. Цей проєкт містить точки входу верхнього рівня системи.

Проєкт складає всі контексти разом, надає їм засоби для розкриття їх маршрутів HTTP (рисунок 2.5).

```
// Add services to the container.
builder.Services.AddControllers();

builder.Services.AddUsersServices(configuration);
builder.Services.AddAnswerServices(configuration);
builder.Services.AddQuestionServices(configuration);
builder.Services.AddTestServices(configuration);
builder.Services.AddResultsServices(configuration);
```

Рисунок 2.5 – Налаштування сутностей системи

Для того, щоб контексти могли обмінюватися повідомленнями, реалізовано посередник повідомлень. Щоразу, коли в обмеженому контексті змінюється стан, що є похідним від поведінки, він опубліковує подію брокеру повідомлень.

Переваги Loosely Coupled Monolith:

- хоча кожен контекст знаходиться в окремому проєкті, вся система загалом є монолітною, весь вихідний код знаходиться в одному рішенні;
- якщо потрібно реорганізувати подію, легко знайти всі контексти, які використовують цю подію;
- спрощене розгортання та налагодження;
- дуже легко взяти окремий контекст та працювати з ним незалежно від стану інших контекстів.

Недоліки LCM:

- одноразове розгортання – так як система монолітна, то існує більше ризиків під час розгортання;
- потрібно не порушувати принципів архітектури системи: не посилатися на реалізацію іншого контексту тощо; важливо, щоб всі розробники розуміли всі правила і нюанси такого підходу;
- коли система розширяється, то час розгортання збільшується в рази.

Взявши до уваги всі переваги та недоліки, можна зробити висновок, що для системи, що розробляється, добре підходить дана структура проєкту.

2.1.2 Entity Framework – ORM та міграції бази даних

Для того, щоб позбавитись від помилок переносу при поводженні з даними, а також задати універсальний механізм, який буде працювати з даними незалежно від того, як ці дані пов’язані з джерелом, використано такі механізми як ORM та міграції до БД.

Міграції баз даних – це контрольований набір змін у структурі об’єктів реляційної БД. Міграції переводять схеми баз даних із поточного стану до нового, незалежно від характеру дії (додавання таблиць, полів, видалення тощо).

Основною перевагою такого підходу є запобігання втраті даних. При виконання кожної міграції програмне забезпечення створює артефакти, які описують точний набір операцій, який виконувався для переводу БД із поточного стану у новий. Тому міграція є контрольованим процесом, і має можливість слідкувати за змінами та обмінюватися ними між членами команди шляхом контролю версій.

В системі, що розробляється, був застосований Entity Framework для того, щоб виконувати міграції даних. Entity Framework – це платформа ORM з відкритим вихідним кодом для додатків .NET, яку підтримує Microsoft [2]. За допомогою даного фреймворку розробники працюють з даними на більш високому рівні абстракції, не вдаючись до подробиць та не зосереджуючи увагу на тому, де зберігаються дані.

Після виконання міграції генерується артефакт зі змінами, які були виконанні [5].

```
ссылка: 1
public partial class InitialCreate : Migration
{
    Ссылок: 0
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.EnsureSchema(
            name: "ResultDBContext");

        migrationBuilder.CreateTable(
            name: "Results",
            schema: "ResultDBContext",
            columns: table => new
            {
                Id = table.Column<int>(type: "int", nullable: false)
                    .Annotation("SqlServer:Identity", "1, 1"),
                ExternalID = table.Column<string>(type: "nvarchar(max)", nullable: false),
                UserExternalID = table.Column<string>(type: "nvarchar(max)", nullable: false),
                TestExternalID = table.Column<string>(type: "nvarchar(max)", nullable: false),
                PassDate = table.Column<DateTime>(type: "datetime2", nullable: false),
                AttemptCount = table.Column<int>(type: "int", nullable: false),
                PercentSuccess = table.Column<float>(type: "real", nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Results", x => x.Id);
            });
```

Рисунок 2.6 – Артефакт міграції

Таким чином, кожна міграція надає нову версію БД, зі збереженням стану минулої версії.

Ще один механізм, який надає змогу працювати з даними незалежно від джерела їх збереження, є механізм ORM. На основі абстракції ORM керує деталями зіставлення між набором об’єктів і базовими реляційними БД, приховуючи деталі пов’язаних інтерфейсів, від розробників.

При зміні джерела даних потрібно змінити лише ORM, а не програми, які використовують ORM. Така можливість забезпечує легше розширення, а також надає змогу користуватися новими класами по мірі їх створення.

На рисунку 2.7 показано використання механізму ORM для роботи з даними.

```
/// <summary>
/// Get all questions
/// </summary>
/// <returns>
/// A list of questions.
/// </returns>
Ссылок: 2
public async Task<IEnumerable<Question>> GetQuestions()
{
    return await questionDBContext.Questions.ToArrayAsync();
}
```

Рисунок 2.7 – Використання механізму ORM

В даному випадку використаний метод, який повертає дані у вигляді масиву. І у випадку зміни джерела даних БД код методу змінювати не потрібно. Всі механізми, описані вище, спрямовані на те, щоб система могла легко розширюватись, змінюватись, щоб код системи був легкий для підтримки та читання.

2.2 Програмна реалізація системи

Після написання структури системи, налаштувань головного проєкту, маршрутів до запитів, атрибутів, точки входу в систему, реалізовано функціонал, який надає система.

В першу чергу створено проєкти контекстів, в яких прописані моделі, реалізовані методи маніпуляції з даними моделей, та контролери, які оброблюють запити.

Моделі містять в собі поля, в яких зберігається інформація сутності. Конструктори моделей призначені для створення об’єкту сутності з параметрами за замовчуванням (рисунок 2.8).

```
Ссылок: 32
public class Test
{
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    ссылка: 1
    public int Id { get; set; } = default;
    Ссылок: 6
    public string ExternalID { get; set; } = default;
    Ссылок: 3
    public string Caption { get; set; } = default;
    Ссылок: 3
    public string Subject { get; set; } = default;
    Ссылок: 5
    public string UserExternalID { get; set; } = default;
    Ссылок: 3
    public bool Open { get; set; } = default;
    Ссылок: 3
    public string Icon { get; set; } = default;
```

Рисунок 2.8 – Модель сутності Test

Реалізовано контексти БД до кожної моделі даних, параметри яких вказують на порядок виконання міграції даних моделі у БД (рисунок 2.9).

```
public class TestDBContext : DbContext
{
    Ссылок: 5
    public DbSet<Test> Tests { get; set; }
    Ссылок: 0
    public DbSet<Subscribers> Subscribers { get; set; }

    private readonly IConfiguration configuration;
    Ссылок: 0
    public TestDBContext(DbContextOptions<TestDBContext> options, IConfiguration configuration)
    {
        this.configuration = configuration;
    }

    Ссылок: 0
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.HasDefaultSchema(nameof(TestDBContext));
    }
}
```

Рисунок 2.9 – Контекст TestDBContext

Реалізовано класи репозиторії, які виконують маніпуляції з конкретним контекстом.

```
Ссылок: 2
public class TestRepository : ITestRepository
{
    private readonly TestDBContext testDBContext;
    Ссылок: 0
    public TestRepository(TestDBContext testDBContext)
    {
        this.testDBContext = testDBContext;
    }
    /// <summary> Delete an test by testId
    Ссылок: 5
    public void Delete(Test test)...
    /// <summary> Delete an test by testId
    Ссылок: 2
    public async Task Delete(int testId)...
    /// <summary> Delete an test by testExternalId
    Ссылок: 2
    public async Task Delete(string testExternalId)...
    /// <summary> Get an test by testId
    Ссылок: 2
    public async Task<Test> GetTestBy(int testId)...
    /// <summary> Get an test by testExternalId
    Ссылок: 4
    public async Task<Test> GetTestBy(string testExternalId)...
    /// <summary> Get all tests
    Ссылок: 2
```

Рисунок 2.10 – Клас TestRepository

Даний клас імплементує інтерфейс ITestRepository. Такий підхід потрібний для того, щоб зробити елементи системи менш зв’язаними один з одним.

Після реалізації моделі, контексту моделі, класу репозиторію, реалізовано контролер до кожного контексту, в якому визначено шляхи до кожного запиту, імплементовано логіку обробки кожного запиту.

```
public class TestsController : ControllerBase
{
    private readonly IConfiguration configuration;
    private readonly ITestRepository testsRepository;

    Ссылок: 0
    public TestsController(IConfiguration configuration, ITestRepository testRepository)
    {
        this.configuration = configuration;
        this.testsRepository = testRepository;
    }
```

Рисунок 2.11 – Клас Test Controller

Кожен контролер має свою конфігурацію, та посилання на клас репозиторію.

Кожен запит передбачає можливі помилки при обробці запиту, та опрацьований таким чином, щоб при виникненні помилки програми сервер не закінчив аварійно свою роботу.

```
[HttpGet]
[ApiConventionMethod(typeof(DefaultApiConventions), nameof(DefaultApiConventions.Get))]
[Route("tests")]
[ProducesResponseType(StatusCodes.Status200OK, Type = typeof(IEnumerable<Test>))]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
Ссылок: 0
public async Task<ActionResult<IEnumerable<Test>>> GetTests()
{
    IEnumerable<Test> tests;
    try
    {
        tests = await testsRepository.GetTests();
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status400BadRequest, ex.Message);
    }

    return Ok(tests);
}
```

Рисунок 2.12 – Запит на список всіх тестів

Контролери контекстів обробляють CRUD (create, read, update, delete) запити, а також додаткові запити, які необхідні клієнту. Це такі запити як читання окремих частин сутностей, запити з параметром тощо.

```
[HttpPost]
[ApiConventionMethod(typeof(DefaultApiConventions), nameof(DefaultApiConventions.Post))]
[Route("users/{userid}/tests/")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status500InternalServerError)]
Ссылок: 0
public async Task<ActionResult<Test>> CreateTest(string userid)
{
    Test newTest;
    try
    {
        newTest = new Test(userid);

        await testsRepository.Insert(newTest);
        await testsRepository.SaveChanges();
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }

    return Ok(newTest);
}
```

Рисунок 2.13 – Метод створення тесту

Запит на створення тесту – при створенні тесту через параметр передається ідентифікатор користувача, який створює тест, для того, щоб зберегти власника тесту, таким чином, закріпивши тест за ним. Інші поля моделі встановлюються за замовчуванням. Коли користувач оновить поля за замовчуванням і збереже зміни, на сервер надійде запит на оновлення даних PUT.

```
public async Task<ActionResult> PutTest(string userID, [FromBody] Test updatedTest)
{
    Test test;
    try
    {
        test = await testsRepository.GetTestBy(updatedTest.ExternalID);
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status400BadRequest, ex.Message);
    }

    if (!test.ExternalID.Equals(updatedTest.ExternalID) || !test.UserExternalID.Equals(updatedTest.UserExternalID))
        return StatusCode(StatusCodes.Status400BadRequest, "Updated ad external id or user id missmatch.");

    try
    {
        test.Caption = updatedTest.Caption;
        test.UserExternalID = updatedTest.UserExternalID;
        test.Subject = updatedTest.Subject;
        test.Icon = updatedTest.Icon;
        test.Open = updatedTest.Open;
        await testsRepository.SaveChanges();
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }

    return Ok();
}
```

Рисунок 2.14 – Запит на оновлення даних тесту

В тілі запиту передається об’єкт Test з оновленими значеннями. Після перевірки, чи існує тест з переданим ідентифікатором, виконується присвоєння нових значень даних тесту та збереження змін.

У випадку, якщо користувача не влаштовує тест, він може видалити тест зі списку своїх тестів.

```
[HttpDelete]
[ApiConventionMethod(typeof(DefaultApiConventions), nameof(DefaultApiConventions.Delete))]
[Route("users/{userid}/tests/{testid}")]
[ProducesResponseType(StatusCodes.Status200OK, Type = typeof(Test))]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
Ссылок: 0
public async Task<ActionResult<Test>> DeleteTest(string userid, string testid)
{
    if (!ExternalIdPassedGuidValidation(userid))
        return StatusCode(StatusCodes.Status400BadRequest, $"Invalid {nameof(userid)}.");

    if (!ExternalIdPassedGuidValidation(testid))
        return StatusCode(StatusCodes.Status400BadRequest, $"Invalid {nameof(testid)}.");

    Test test;
    try
    {
        test = await testsRepository.GetTestBy(testid);
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status400BadRequest, ex.Message);
    }

    try
    {
        testsRepository.Delete(test);
        await testsRepository.SaveChanges();
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }

    return Ok(test);
}
```

Рисунок 2.15 –Метод видалення тесту

Серверу надходить запит на видалення з ідентифікатором користувача, який видаляє тест, та ідентифікатором тесту. Якщо такий тест існує, і власником тесту, що видаляється, є користувач, то тест видаляється з бази тестів. В іншому випадку запит не виконається та повідомить клієнта про помилку.

Неавторизований користувач має доступ до обмеженого функціоналу системи. Він може тільки проходити тести, які знаходяться у публічному доступі, бачити рейтинги користувачів щодо результатів проходження публічних тестів. Тому для того, щоб отримати повний доступ до функцій системи, користувачу необхідно зареєструватися. При реєстрації користувач дає свій нікнейм та пароль для входу. Сервер опрацьовує запит на додає нового користувача в сховище даних.

```
[HttpPost]
[ApiConventionMethod(typeof(DefaultApiConventions), nameof(DefaultApiConventions.Post))]
[Route("users/signup")]
[AllowAnonymous]
[ProducesResponseType(StatusCodes.Status200OK, Type = typeof(User))]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status500InternalServerError)]
Ссылок: 0
public async Task<ActionResult<User>> SignUp(User user)
{
    user.ExternalId = Guid.NewGuid().ToString();
    user.Password = HashPassword(user.Password);
    try
    {
        await usersRepository.Insert(user);
        await usersRepository.SaveChanges();
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
    return Ok(user);
}
```

Рисунок 2.16 – Запит на реєстрацію

Перед тим як додавати дані користувача у сховище даних, пароль користувача зашифровується для безпеки на випадку несанкціонованого доступу до даних.

Висновки до розділу 2

В другому розділі описано архітектурний підхід, застосований при створенні системи, розкрито сутність паттерну проєктування Loosely Coupled Monolith та наголошено на перевагах слабко зв’язаної архітектури. Описано структуру проєкту. Розглянуто механізм роботи Entity Framework, переваги застосування міграцій для роботи з БД та технології ORM. Наведено деталі реалізації системи та приведено приклади застосування обраних технологій у коді застосунку. Результатом є розроблена back-end частина програмного забезпечення автоматизації тестування знань.

ВИСНОВКИ

Використання інформаційних технологій здатне значно підвищити ефективність проведення тестування для перевірки знань учнів та сприяти оптимізації навчального процесу в умовах дистанційної освіти.

Дана частина роботи присвячена розробці back-end частини програмного забезпечення автоматизації тестування знань.

В ході роботи були виконані наступні завдання:

- здійснено огляд клієнт-серверних технологій, їх принципів роботи та особливостей;
- проведено аналіз можливих технологій імплементації back-end частини застосунку, зроблено вибір на користь фреймворку ASP .NET Web API;
- розглянуто особливості технологій, використаних при розробці: ASP .NET Web API Framework, Entity Framework, Swagger та ін., описано основні поняття, пов’язані із ними;
- розкрито сутність паттерну Loosely Coupled Monolith, що лежить а основі системи;
- виконано проєктування бази даних відповідна до вимог нормалізації відношень;
- здійснено програмну реалізацію та тестування коректності роботи застосунку.

Отже, були успішно виконані усі поставлені завдання. Результатом роботи є готова back-end частина застосунку для автоматизації тестування знань.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Dresher T., Zuker A., Friedman S. Hands-On Full-Stack Web Development with ASP. NET Core: Learn end-to-end web development with leading frontend frameworks, such as Angular, React, and Vue. Packt Publishing Ltd, 2018.

2. Framework E. Entity framework. *Architecture*. Vol. 19, 2021. P. 20.

3. Kanjilal J. ASP. NET Web API: Build RESTful web applications and services on the. NET framework. Packt Publishing Ltd, 2013.

4. Kent W. A simple guide to five normal forms in relational database theory. *Communications of the ACM*. Vol. 26, Issue 2. P. 120–125.

5. Lerman J. Programming Entity Framework: Building Data Centric Apps with the ADO. NET Entity Framework. « O'Reilly Media, Inc.», 2010.

6. Oluwatosin H. S. Client-server model. *IOSR Journal of Computer Engineering*. Vol. 16, Issue 1. P. 67–71.

7. Seemann M. Dependency injection in. NET. Manning New York, 2012.

8. Uurlu A., Zeitler A., Kheyrollahi A. Pro ASP. NET Web API: HTTP Web Services in ASP. NET. Apress, 2013.

9. Watt A., Eng N. Database design. 2014.

ДОДАТОК А

Точка входу в програму. Налаштування проєктів рішення

Файл Program.cs (точка входу) головного проєкту WebAPI:

```
using MassTransit;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Mvc;
using Microsoft.IdentityModel.Tokens;
using System.Text;
using Users;
using Answers;
using Questions;
using Tests;
using Results;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

IConfiguration configuration = builder.Configuration;

string MyAllowSpecificOrigins = "MeOrigins";

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: MyAllowSpecificOrigins,
        builder =>
            builder
                .AllowAnyMethod()
                .AllowAnyHeader()
                .AllowAnyOrigin()
                .WithOrigins("http://localhost:4200")
        );
});

IConfigurationSection tokenOptions = configuration.GetSection("TokenOptions");
string validIssuer = tokenOptions.GetSection("Issuer").Value;
string validAudience = tokenOptions.GetSection("Audience").Value;
string key = tokenOptions.GetSection("Key").Value;

builder.Services
    .AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.RequireHttpsMetadata = false;
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidIssuer = validIssuer,
            ValidateAudience = true,
            ValidAudience = validAudience,
            ValidateLifetime = true,
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.ASCII.GetBytes(key)),
            ValidateIssuerSigningKey = true,
        };
    });

// Add services to the container.
builder.Services.AddControllers();

builder.Services.AddUsersServices(configuration);
builder.Services.AddAnswerServices(configuration);
```

```
builder.Services.AddQuestionServices(configuration);
builder.Services.AddTestServices(configuration);
builder.Services.AddResultsServices(configuration);

builder.Services.AddApiVersioning(o =>
{
   o.ReportApiVersions = true;
   o.AssumeDefaultVersionWhenUnspecified = true;
   o.DefaultApiVersion = new ApiVersion(1, 0);
}
);

builder.Services.AddMediator(config =>
{
   config.AddUsersMediatorConsumers();
   config.AddAnswerMediatorConsumers();
   config.AddQuestionMediatorConsumers();
   config.AddTestMediatorConsumers();
   config.AddResultMediatorConsumers();
});

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
//if (app.Environment.IsDevelopment())
//{
app.UseSwagger();
app.UseSwaggerUI();
//}

app.UseHttpsRedirection();

app.UseRouting();

app.UseCors(MyAllowSpecificOrigins);

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

app.MapControllers();

app.Run();
```

Налаштування проєкту. Налаштування підключень інших проєктів.

WebApi.csproj

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

        <ItemGroup>
                <PackageReference Include="MassTransit.AspNetCore" Version="7.3.0" />

                <PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Version="6.0.1" />
```

```
        <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="6.0.1">
          <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
          <PrivateAssets>all</PrivateAssets>
        </PackageReference>

        <PackageReference Include="Swashbuckle.AspNetCore" Version="6.2.3" />
    </ItemGroup>

    <ItemGroup>
        <ProjectReference Include="..\Users\Users.csproj" />
        <ProjectReference Include="..\Answers\Answers.csproj" />
        <ProjectReference Include="..\Questions\Questions.csproj" />
        <ProjectReference Include="..\Tests\Tests.csproj" />
        <ProjectReference Include="..\Results\Results.csproj" />
    </ItemGroup>

    <ItemGroup>
        <Content Update="wwwroot\default.mp4">
            <CopyToOutputDirectory>Always</CopyToOutputDirectory>
        </Content>
        <Content Update="wwwroot\default.png">
            <CopyToOutputDirectory>Always</CopyToOutputDirectory>
        </Content>
    </ItemGroup>

    <ProjectExtensions><VisualStudio><UserProperties appsettings_1json__JsonSchema=""
/></VisualStudio></ProjectExtensions>

</Project>
```

ДОДАТОК Б

Модулі системи. Проєкт Test

Кожен проєкт працює з окремою сутністю.

Модель проєкту Test.

Test.cs

```
using System;
using System.ComponentModel.DataAnnotations.Schema;

namespace Tests.Models
{
   public class Test
   {
      [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
      public int Id { get; set; } = default;
      public string ExternalID { get; set; } = default;
      public string Caption { get; set; } = default;
      public string Subject { get; set; } = default;
      public string UserExternalID { get; set; } = default;
      public bool Open { get; set; } = default;
      public string Icon { get; set; } = default;

      public Test(string userExternalID)
      {
         if (string.IsNullOrEmpty(userExternalID))
            throw new NullReferenceException($"Parameter {nameof(userExternalID)} passed into {nameof(Test)} constructor
is null or empty.");
```

```
            UserExternalID = userExternalID;
            Caption = "Test caption...";
            Subject = "Test subject...";
            Open = true;
            Icon = "base64 image";
            ExternalID = Guid.NewGuid().ToString();
        }
    }
}
```

TestDBContext.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Storage;
using Microsoft.Extensions.Configuration;

namespace Tests.Models
{
    public class TestDBContext : DbContext
    {
        public DbSet<Test> Tests { get; set; }
        public DbSet<Subscribers> Subscribers { get; set; }

        private readonly IConfiguration configuration;
        public TestDBContext(DbContextOptions<TestDBContext> options, IConfiguration configuration) : base(options)
        {
            this.configuration = configuration;
        }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.HasDefaultSchema(nameof(TestDBContext));
        }
    }
}
```

Взаємодія з класом сутності.

ITestRepoository.cs

```
using Tests.Models;

namespace Tests.Repository
{
    public interface ITestRepository
    {
        Task<IEnumerable<Test>> GetTests();
        Task<IEnumerable<Test>> GetUserTests(string userExternalId);
        Task<Test> GetTestBy(int testId);
        Task<Test> GetTestBy(string testExternalId);
        Task<bool> TestExists(string testExternalId);
        Task Insert(Test test);

        void Delete(Test test);
        Task Delete(int testId);
        Task Delete(string testExternalId);
        Task SaveChanges();
    }
            }
```

TestRepository.cs

```
using Tests.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace Tests.Repository
{
   public class TestRepository : ITestRepository
   {
      private readonly TestDBContext testDBContext;
      public TestRepository(TestDBContext testDBContext)
      {
         this.testDBContext = testDBContext;
      }
      /// <summary>
      /// Delete an test by <paramref name="testId"/>
      /// </summary>
      /// <param name="test">Test object.</param>
      /// <exception cref="TestNotExistException">Thrown when <paramref name="test"/> is null.</exception>
      public void Delete(Test test)
      {
         if (test == default)
            throw new NullReferenceException($"Parameter {nameof(test)} passed into {nameof(Delete)} is null");

         testDBContext.Tests.Remove(test);
      }
      /// <summary>
      /// Delete an test by <paramref name="testId"/>
      /// </summary>
      /// <param name="testId">Test id.</param>
      /// <exception cref="TestNotExistException">Thrown when the test with <paramref name="testId"/> not
exist.</exception>
      public async Task Delete(int testId)
      {
         Test test = await GetTestBy(testId);
         Delete(test);
      }
      /// <summary>
      /// Delete an test by <paramref name="testExternalId"/>
      /// </summary>
      /// <param name="testExternalId">Test external id.</param>
      /// <exception cref="TestNotExistException">Thrown when test with <paramref name="testExternalId"/> not
exist.</exception>
      public async Task Delete(string testExternalId)
      {
         Test test = await GetTestBy(testExternalId);
         Delete(test);
      }
      /// <summary>
      /// Get an test by <paramref name="testId"/>
      /// </summary>
      /// <returns>
      /// An test object.
      /// </returns>
      /// <param name="testId">Test id.</param>
      /// <exception cref="TestNotExistException">Thrown when test with <paramref name="testId"/> not exist.</exception>
      public async Task<Test> GetTestBy(int testId)
      {
         Test test = await testDBContext.Tests.FirstOrDefaultAsync(t => t.Id == testId);
```

```
            if (test == null)
                throw new NullReferenceException($"Test with {nameof(testId)} = {testId} not exist.");

            return test;
        }
        /// <summary>
        /// Get an test by <paramref name="testExternalId"/>
        /// </summary>
        /// <returns>
        /// An test object.
        /// </returns>
        /// <param name="testExternalId">Test external id.</param>
        /// <exception cref="TestNotExistException">Thrown when test with <paramref name="testExternalId"/> not
exist.</exception>
        public async Task<Test> GetTestBy(string testExternalId)
        {
            Test test = await testDBContext.Tests.FirstOrDefaultAsync(t => t.ExternalID == testExternalId);

            if (test == default)
                throw new NullReferenceException($"Test with {nameof(testExternalId)} = {testExternalId} not exist.");

            return test;
        }
        /// <summary>
        /// Get all tests
        /// </summary>
        /// <returns>
        /// A list of tests.
        /// </returns>
        public async Task<IEnumerable<Test>> GetTests()
        {
            return await testDBContext.Tests.ToArrayAsync();
        }
        /// <summary>
        /// Insert new test.
        /// </summary>
        /// <param name="test">Test object.</param>
        /// <exception cref="TestNotExistException">Thrown when <paramref name="test"/> is null.</exception>
        public async Task Insert(Test test)
        {
            if (test == default)
                throw new NullReferenceException($"Parameter {nameof(test)} passed into {nameof(Insert)} is null");

            await testDBContext.AddAsync(test);
        }
        public async Task<bool> TestExists(string testExternalId)
        {
            return await testDBContext.Tests.AnyAsync(t => t.ExternalID.Equals(testExternalId));
        }
        public async Task SaveChanges()
        {
            await testDBContext.SaveChangesAsync();
        }

        public async Task<IEnumerable<Test>> GetUserTests(string userExternalId)
        {
            return await testDBContext.Tests.Where(t => t.UserExternalID.Equals(userExternalId)).ToArrayAsync();
        }
    }
}
```

TestController.cs

```
using Microsoft.AspNetCore.Mvc;
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading.Tasks;
using Tests.Repository;
using Tests.Models;
using Microsoft.AspNetCore.Authorization;
using System.Security.Claims;
using System.IdentityModel.Tokens.Jwt;
using Microsoft.IdentityModel.Tokens;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Configuration;

namespace Tests.Controllers
{
   [ApiVersion("1.0")]
   [Authorize]
   [ApiController]
   [Route("tests-management")]
   public class TestsController : ControllerBase
   {
      private readonly IConfiguration configuration;
      private readonly ITestRepository testsRepository;

      public TestsController(IConfiguration configuration, ITestRepository testRepository)
      {
         this.configuration = configuration;
         this.testsRepository = testRepository;
      }

      [HttpGet]
      [ApiConventionMethod(typeof(DefaultApiConventions), nameof(DefaultApiConventions.Get))]
      [AllowAnonymous]
      [Route("version")]
      [ProducesResponseType(StatusCodes.Status200OK, Type = typeof(string))]
      public ActionResult<string> Version(ApiVersion apiVersion) => Ok($"Active {nameof(TestsController)} API ver is
{apiVersion}");

      [HttpGet]
      [ApiConventionMethod(typeof(DefaultApiConventions), nameof(DefaultApiConventions.Get))]
      [Route("tests")]
      [ProducesResponseType(StatusCodes.Status200OK, Type = typeof(IEnumerable<Test>))]
      [ProducesResponseType(StatusCodes.Status400BadRequest)]
      public async Task<ActionResult<IEnumerable<Test>>> GetTests()
      {
         IEnumerable<Test> tests;
         try
         {
            tests = await testsRepository.GetTests();
         }
         catch (Exception ex)
         {
            return StatusCode(StatusCodes.Status400BadRequest, ex.Message);
         }
         return Ok(tests);
      }

      [HttpGet]
      [ApiConventionMethod(typeof(DefaultApiConventions), nameof(DefaultApiConventions.Get))]
```

```
[Route("tests/{userid}")]
[ProducesResponseType(StatusCodes.Status200OK, Type = typeof(IEnumerable<Test>))]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<ActionResult<IEnumerable<Test>>> GetUserTests(string userid)
{
    IEnumerable<Test> tests;
    try
    {
        tests = await testsRepository.GetUserTests(userid);
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status400BadRequest, ex.Message);
    }
    return Ok(tests);
}

[HttpPost]
[ApiConventionMethod(typeof(DefaultApiConventions), nameof(DefaultApiConventions.Post))]
[Route("users/{userid}/tests/")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status500InternalServerError)]
public async Task<ActionResult<Test>> CreateTest(string userid)
{
    Test newTest;
    try
    {
        newTest = new Test(userid);

        await testsRepository.Insert(newTest);
        await testsRepository.SaveChanges();
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
    return Ok(newTest);
}

[HttpPut]
[ApiConventionMethod(typeof(DefaultApiConventions), nameof(DefaultApiConventions.Put))]
[Route("users/{userid}/tests/")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status500InternalServerError)]
public async Task<ActionResult> PutTest(string userID, [FromBody] Test updatedTest)
{
    Test test;
    try
    {
        test = await testsRepository.GetTestBy(updatedTest.ExternalID);
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status400BadRequest, ex.Message);
    }
    if (!test.ExternalID.Equals(updatedTest.ExternalID) || !test.UserExternalID.Equals(updatedTest.UserExternalID))
        return StatusCode(StatusCodes.Status400BadRequest, "Updated ad external id or user id missmatch.");
    try
    {
        test.Caption = updatedTest.Caption;
        test.UserExternalID = updatedTest.UserExternalID;
        test.Subject = updatedTest.Subject;
```

```
            test.Icon = updatedTest.Icon;
            test.Open = updatedTest.Open;
            await testsRepository.SaveChanges();
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
        return Ok();
    }

    [HttpDelete]
    [ApiConventionMethod(typeof(DefaultApiConventions), nameof(DefaultApiConventions.Delete))]
    [Route("users/{userid}/tests/{testid}")]
    [ProducesResponseType(StatusCodes.Status200OK, Type = typeof(Test))]
    [ProducesResponseType(StatusCodes.Status400BadRequest)]
    public async Task<ActionResult<Test>> DeleteTest(string userid, string testid)
    {
        if (!ExternalIdPassedGuidValidation(userid))
            return StatusCode(StatusCodes.Status400BadRequest, $"Invalid {nameof(userid)}.");

        if (!ExternalIdPassedGuidValidation(testid))
            return StatusCode(StatusCodes.Status400BadRequest, $"Invalid {nameof(testid)}.");

        Test test;
        try
        {
            test = await testsRepository.GetTestBy(testid);
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status400BadRequest, ex.Message);
        }
        try
        {
            testsRepository.Delete(test);
            await testsRepository.SaveChanges();
        }
        catch (Exception ex)
        {
            return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
        }
        return Ok(test);
    }

    private bool ExternalIdPassedGuidValidation(string externalId) => Guid.TryParse(externalId, out _);

  }
}
```

ConfigureServices.cs

```
using Microsoft.EntityFrameworkCore;
using Tests.Models;
using Tests.Repository;
using Tests.Consumer;
using MassTransit.ExtensionsDependencyInjectionIntegration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;

namespace Tests
{
  public static class ConfigureServices
  {
```

```
    public static void AddTestServices(this IServiceCollection services, IConfiguration configuration)
    {
        string connection = configuration.GetConnectionString("DBConnection");
        services.AddDbContext<TestDBContext>(options => options.UseSqlServer(connection));

        services.AddTransient<ITestRepository, TestRepository>();
    }

    public static void AddTestMediatorConsumers(this IServiceCollectionMediatorConfigurator configuration)
    {
        configuration.AddConsumer<TestStatusConsumer>();
    }
}
        }
```

Взаємодія з іншими проєктами через запити до них.

GetTestStatusRequest.cs проєкту Tests.Contracts.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace Tests.Contracts
{
    public class GetTestStatusRequest
    {
        public string TestExternalId { get; }
        public GetTestStatusRequest(string testExternalId)
        {
            if (string.IsNullOrEmpty(testExternalId))
                throw new ArgumentException($"Parameter {nameof(testExternalId)} passed into
{nameof(GetTestStatusRequest)} is null or empty.");
            TestExternalId = testExternalId;
        }
    }
}
```

GetTestStatusResponse проєкту Tests.Contracts.

```
namespace Tests.Contracts
{
    public class TestStatusResponse
    {
        public string TestExternalId { get; }
        public bool Exists { get; }
        public TestStatusResponse(string testExternalId, bool exists)
        {
            if (string.IsNullOrEmpty(testExternalId))
                throw new ArgumentException($"Parameter {nameof(testExternalId)} passed into {nameof(TestStatusResponse)}
is null or empty.");

            TestExternalId = testExternalId;
            Exists = exists;
        }}}
```

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інженерії програмного забезпечення

КОМПЛЕКСНА КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ ЗНАНЬ

СПЕЦІАЛЬНА ЧАСТИНА З ОХОРОНИ ПРАЦІ

ПИТАННЯ ОХОРОНИ ПРАЦІ В ТРУДОВІЙ ДІЯЛЬНОСТІ ПРОГРАМІСТА

Спеціальність «Інженерія програмного забезпечення»

121 – ККРБ.1 – 409.21920801

Студент _____М. С. Кіяшко

підпис

«___» _____ 2022 р.

Консультант канд. техн. наук, доцент кафедри екології

___А. О. Алексєєва

підпис

«___» _____ 2022 р.

Миколаїв – 2022

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

ДСанПіН – державні санітарні правила і норми

ДСН – державні санітарні норми

ЕМВ – електромагнітне випромінювання

ЕМП – електромагнітне поле

ІТ – інформаційні технології

ОП – охорона праці

ПЗ – програмне забезпечення

ПК – персональний комп’ютер

РКЕ – рідкокристалічний екран

ВСТУП

Безпечні умови праці є важливими для представників усіх професій, в тому числі й програмістів. Недотримання норм охорони праці призводить до розвитку професійних хвороб, які значно погіршують якість життя працівників та впливають на продуктивність праці.

Охорона праці для розробників програмного забезпечення має свої особливості в зв'язку з характером їх роботи. Через тривале перебування за комп'ютером можуть виникнути проблеми з зором та хребтом. Інтенсивність розумової діяльності може викликати нервове напруження та перевтому. Тому важливо знати та додержуватись вимог до організації робочого місця для зменшення негативних наслідків для здоров'я працівника.

Актуальним є також питання безпеки життєдіяльності працівника. Хоча робота програміста не належить до найбільш небезпечених, час від часу можуть виникати надзвичайні ситуації, що становлять загрозу життю та здоров'ю. Наприклад, в офісі може виникнути пожежа через несправність електромережі. Потенційну небезпеку становить також ураження струмом при роботі з електроприладами. Тому важливо знати правила першої допомоги при нещасних випадках та правила поводження у надзвичайних ситуаціях.

Метою розділу є аналіз вимог до охорони праці та безпеки життєдіяльності розробника ПЗ. Для досягнення мети розглядаються вимоги до організації робочого місця, аналізуються параметри приміщення, де проходить трудова діяльність, такі як освітленість, електромагнітне випромінювання, рівень шуму, мікроклімат. Крім того, наводяться правила дій під час небезпечних ситуацій, які становлять найбільший ризик для працівника в офісі – алгоритм дій при пожежі та перша допомога при ураженні струмом.

1 Умови праці на робочому місці розробника програмного забезпечення

1.1 Освітлення під час роботи з комп'ютером

Людина отримує завдяки зору близько 90% інформації про оточуюче середовище. Тому правильне освітлення є важливою складовою організації робочого процесу. Погане освітлення призводить до стомлюваності та підвищує травмобезпеку.

За джерелом освітлення поділяють на:

– природне (створюється сонцем);

– штучне (здійснюється за допомогою електричних ламп);

– змішане.

Існують різні види освітлення робочих приміщень. За функціональним призначенням освітлення поділяють на [1]:

– робоче;

– аварійне;

– евакуаційне;

– охоронне;

– чергове.

Робоче освітлення необхідне для забезпечення нормальної роботи.

Аварійне освітлення застосовується для можливості продовжувати роботу при раптовому вимкненні світла. При цьому норма освітленості має складати 5% від робочого освітлення, але повинна бути не меншою за 2 лк всередині приміщень та 1 лк на території підприємства.

Евакуаційне освітлення застосовуються у разі виникнення аварій та потребі евакуювати людей з приміщення. Воно може бути встановлене над сходами та в іншим місцях, де рухатися без освітлення при евакуації було б небезпечно. Світильники приєднують до окремої мережі, незалежної від робочого освітлення.

Охоронне освітлення призначене для використання охоронним підрозділом, якщо немає природного світла.

Чергове освітлення використовується вночі, у вихідні та святкові дні.

Для дотримання належного рівня освітлення у виробничих приміщеннях варто пам’ятати про правила експлуатації джерел світла. Так, при плануванні освітлення приміщення слід брати до уваги те, що при тривалій експлуатації світловий потік ламп зменшується: для ламп розжарювання – на 10-15%, для люмінесцентних – на 20-25%.

Очищення скла вікон має здійснюватися не менше, ніж 2 рази на рік у приміщеннях із незначним виділенням пилу та не менше ніж 4 рази на рік, якщо виділення пилу значне. Очищення світильників має проводитися 4-12 разів на рік.

Не менш, ніж раз на 6 місяців необхідно перевіряти стан проводів освітлювальної установки. Не менш ніж один раз на рік слід перевіряти рівень освітленості в контрольних точках приміщення.

Для визначення рівня освітленості використовуються люксометр.

Робота програміста напряму пов’язана із зоровим сприйняттям, тому дуже важливе дотримання норм освітлення робочих приміщень. Нестача світла призводить до втомлюваності очей, а в довготривалій перспективі веде до виникнення хвороб зору, таких як короткозорість чи спазм акомодації. Погане освітелення позначається також на зниженні показників продуктивності, призводить до збільшення кількості помилок підчас роботи.

В наступній таблиці наведені норми освітленості для офісу.

Таблиця 1 – Норми освітленості у приміщеннях

Тип приміщення в залежності від виконуваних робіт	Норма освітленості, лк
Великий офіс з вільним плануванням	400
Офіс з комп’ютерами	200-300
Конференц зал	200
Сходові прольоти, ескалатори, хол	75-100

Значення світлового потоку можна розрахувати за формулою:

Світловий потік = Норма освітленості * Площа * Коефіцієнт висоти стелі

Значення коефіцієнтів висоти стелі наведені в таблиці 2.

Таблиця 2 – Коефіцієнти висоти стелі для розрахунку світлового потоку

Висота стелі, м	Коефіцієнт
2,5-2,7	1
2,7-3	1,2
3-3,5	1,5
3,5-4,5	2

Розрахуємо значення для офісу площею 35 м^2 та з висотою стелі 2,7 м.

Світловий потік:

$$Ф = 300 \cdot 35 \cdot 1{,}2 = 12600(Лм)$$

Якщо відомий необхідний світловий потік, можна розрахувати кількість лампочок, необхідну для освітлення даного приміщення. Кількість необхідних лампочок залежить від їх типу та потужності. Приблизна кількість світильників для даного приміщення обрахована у наступній таблиці. Отримане число округлюють в більшу сторону.

Таблиця 3 – Розрахунок кількості лампочок для освітлення офісу

Тип лампочки	Потужність, Вт	Світловий потік, Лм	Кількість лампочок, шт.
Розжарювання	40	470	12 600/470 = 27
Енергоощадна	15	700	12 600/700 = 18
Світлодіодна	10	750	12 600/750 = 17

Яскравість освітлення залежить також від кольору інтер’єру. Для темного інтер’єру необхідне більш яскраве оствітлення, ніж для світлого.

1.2 Електромагнітне випромінювання при роботі з комп’ютером

На робочих місцях з комп’ютерною технікою за походженням розрізняють два види полів:

– поля, створені безпосередньо ПК;

– поля, створені іншими джерелами поблизу робочого місця (фонові поля).

Електронна техніка при роботі може створювати навколо себе різні види полів, такі як:

– електростатичне поле (присутнє лише для техніки з використанням електронно-променевих трубок, РКЕ такого поля не породжують);
– змінні низькочастотні електричні поля;
– змінні низькочастотні магнітні поля.

Також потенційну загрозу здоров'ю становлять електромагнітне випромінювання радіочастотного діапазону та електромагнітні поля, що створюються на робочому місці сторонніми джерелами.

Джерелами змінних електромагнітних полів у комп'ютері є вузли з високою змінною напругою або великими струмами. Норми напруженості елекромагнітних полів затверджені у ДСанПіН 3.3.2.007-98 [5] та загальноєвропейському стандарті MPR II (так званий «шведський стандарт»). Також вимоги до рівнів електромагнітного випомінювання задає універсальний рекомендаційний стандарт ТСО'99.

В таблиці 4 наведені допустимі рівні випромінювань при роботі з монітором ПК.

Таблиця 4 – Норми електромагнітних випромінювань моніторів

Види поля	ТСО (на відстані 0.3 м від центра екрана і 0,5 м навколо монітора)	MRP II (на відстані 0,5 м навколо монітора)
Змінне електричне поле		
5 Гц – 2 кГц	10 В/м	2,5 В/м
2 кГц – 400 кГц	1 В/м	2,5 В/м
Змінне магнітне поле		
5 Гц – 2 кГц	200 мА/м	200 мА/м
2 кГц – 400 кГц	20 мА/м	20 мА/м

Рідкокристалічні екрани, що використовуються в сучасних офісах, не створюють шкідливого випромінювання, що властиве моніторам на основі електронно-променевих трубок: вони не є джерелами рентгенівського випромінювання та електростатичного поля. Проте для них все одно існують змінні електричні та змінні магнітні поля. Джерелами ЕМВ можуть бути РКЕ, плазмені монітори та високочастотні перетворювачі живлення портативних комп'ютерів.

ЕМВ сучасних моніторів не перевищує встановлених норм, проте лише на відстані не менш, ніж 0,5 м від монітору. Якщо відстань менша, можливе перевищення норм випромінювання. Також важливе значення має заземлення. Якщо монтіори і системні блоки не заземлені, но рівень випромінювання на робочому місці майже завжди перевищує норми.

Отже, для забезпечення низького рівня електромагнітного випромінювання необхідно подбати про заземлення та збереження мінімальної відстані 0,5 м від екрану під час роботи за комп'ютером.

Для дотримання вимог щодо ЕМВ на робочому місці необхідно виконувати наступні правила:

- приміщення повинно знаходитися далеко від сторонніх джерел потужних електромагнітних полів (таких як трансформатори, електричні розподільні щитки, радіопередавальними пристроями тощо);
- металеві грати на вікнах приміщення, якщо такі є, мають бути заземлені;
- робочі місця із значним скупченням комп'ютерної та іншої техніки краще розміщати на нижніх поверхах. Таким чином вони матимуть мінімальний вплив на загальну електромагнітну обстановку в будинку. При цьому також знижується потужність ЕМП на самих робочих місцях, адже опір заземлення на нижніх поверхах будинків є найменшим;
- заземлення має підводитися до кожного робочого місця;
- проводи живлення рекомендується прокладати в екрануючих металевих оболонках;

– місця підключення декількох ПК повинні мати екрановані щитки з достатньою кількістю розеток. Розміщенні вони мають бути якомога далі від робочих місць користувачів ПК та інших прицівників офісу;
– до групового місця бажано підключати не більше, ніж 2-3 комп'ютери;
– екран та системний блок комп'ютера мають знаходитись на максимальній відстані від користувача;
– користувач має знаходитися якомога далі від мережних розеток та проводів електроживлення. Небажане використання різноманітних подовжувачів.

Важливим є розташування ПК та іншого обладання на робочому місці. Найбільш оптимальними з точки зору електромагнійтної безпеки є компонування, представлені на рисунку 1. Для цих компонувань характерне те, що повністю розділені зона, в якій перебуває користувач ПК та зона підведення електропостачання до технічного обладнання.

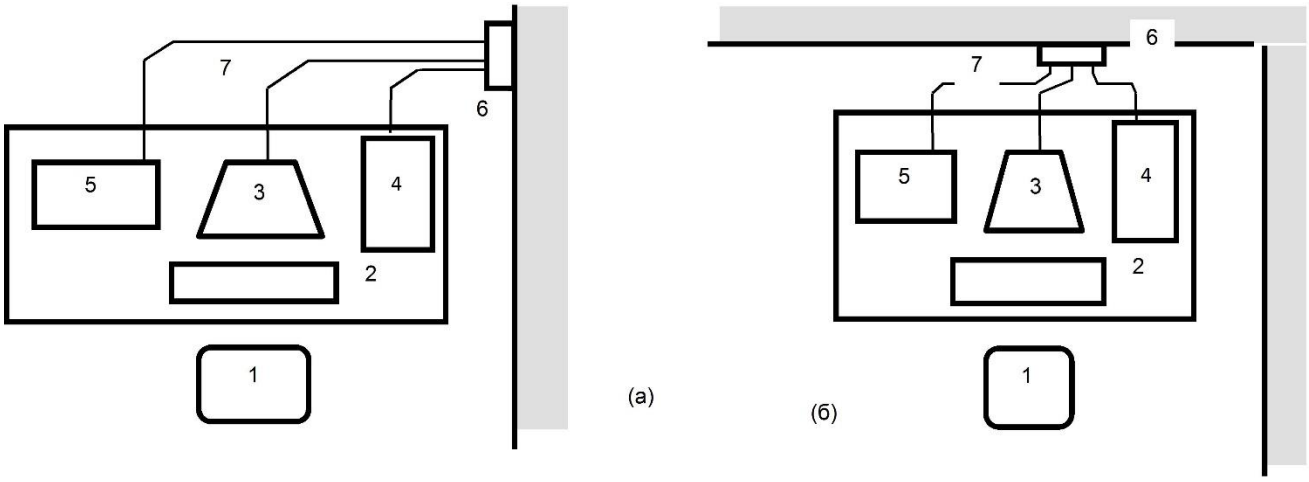

Рисунок 1 – Рекомендований спосіб облаштування робочого місця (1 – робоче місце оператора, 2 – клавіатура, 3 – дисплей, 4 – системний блок, 5 – принтер, 6 – розетки живлення, 7 – мережні кабелі живлення)

Для порівняння на рисунках 2 та 3 наведені небажані способи компонування робочих місць. На них кабелі електроживлення розташовані поряд із користувачем. Схема, представлена на рисунку 2 є незадовільною, якщо на робочому місці знаходиться велика кількість технічних засобів, що мають

значний рівень електроспоживання. Вкрай небажаним є спосіб організації робочого місця на рисунку 3.

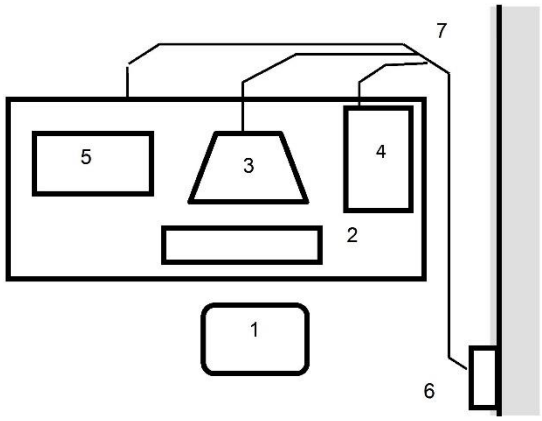

Рисунок 2 – Небажаний спосіб планування робочого місця

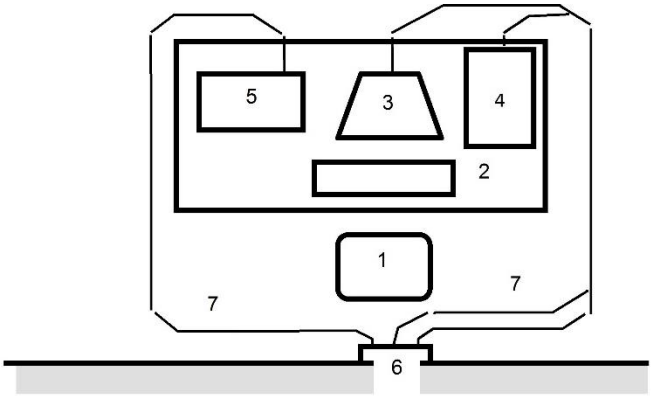

Рисунок 3 – Неприпустимий спосіб планування робочого місця

Зазвичай в офісному приміщенні розташовується одразу декілька робочих місць з ПК. В цьому випадку необхідно дотримуватись таких основних принципів:

– робочі місця повинні мати автономне електроживлення, завдяки якому їх легко відключити, коли вони не експлуатуються. Це дозволяє зменшити електромагнітний фон у приміщенні;
– мережні елементи мають бути максимально віддалені від кожного користувача, також користувачі мають знаходитися на максимальній відстані від апаратури сусідніх робочих місць.

1.3 Мікроклімат в приміщенні з використанням комп'ютерів

Мікроклімат виробничого приміщення – сукупність метеорологічних умов внутрішнього середовища приміщень. Основними параметрами мікроклімату є температура, відносна вологість, швидкість руху повітря, теплове випромінювання.

Самопочуття працівників та продуктивність їх праці значною мірою залежать від факторів довкілля. Тому працедавець зобов'язаний створити оптимальні, або принаймні допустимі мікрокліматичні умови на робочому місці.

Метеорологічні умови робочої зони відповідно до санітарних норм повинні вимірюватися на висоті 2 м над рівнем підлоги. Робоча зона – простір, в якому знаходяться робочі місця працівників.

Оптимальними умовами вважаються такі, при яких не відбувається напруження механізмів терморегуляції; такі умови є найбільш сприятливими та при тривалій дії на людину не викликають відчуття дискомфорту. Такі умови сприяють підтримці високого рівня працездатності.

Допустимими вважаються умови, при довготривалій дії яких у людини можуть виникати дискомфортні відчуття, проте вони швидко минають і стан людини нормалізується за рахунок напруження механізмів терморегуляції та пристосування організму до умов середовища.

Для створення оптимальних умов праці в офісі температура повітря в офісному приміщенні має виносити 21-25°С. Оптимальна вологість повітря становить 40-60%, при цьому вважається допустимою вологість не більше 75%. Швидкість руху повітря не повинна перевищувати 0,1 м/с [2].

Таблиця 5 – Норми температури в залежності від пори року та виду праці

Період року	**Категорія робіт**	**Температура, °С**
Холодний	Легка (І а)	22-24
	Легка (І б)	21-23
Теплий	Легка (І а)	23-25
	Легка (І б)	22-24

В таблиці 5 наведені оптимальні показники температури відповідно до періоду року для легкої за фізичним навантаженням категорії робіт, до яких відноситься праця за комп'ютером у офісі.

1.3 Рівень шуму на робочому місці з комп'ютером

Здатність працівника до концентрації та ефективної роботи сильно залежить від рівню шуму в приміщенні. Шум відволікає від роботи, призводить до збільшення кількості помилок під час праці. Рівень шуму нормується ДСН 3.3.6.037-99 «Санітарні норми виробничого шуму, ультразвуку та інфразвуку» [3].

Допустимий рівень шуму залежить від характеру трудової діяльності. Норми для різних професій наведені у таблиці 6.

Таблиця 6 – Вимоги до рівня шуму відповідно до виду діяльності

№	Вид трудової діяльності	Рівні звуку, дБА
1	Творча діяльність, керівна діяльність із підвищеними вимогами, наукова діяльність, конструювання й проектування, програмування, викладання	50
2	Висококваліфікована робота, що адміністративно-керівна діяльність, вимірювальні й аналітичні роботи в лабораторії	60
3	Робота із часто одержуваними вказівками й акустичними сигналами, робота, що вимагає постійного слухового контролю, операторська робота, диспетчерська робота	65
4	Робота, що вимагає зосередження, робота з підвищеними вимогами до процесів спостереження й дистанційного керування	75
5	Виконання всіх видів робіт (крім перерахованих у п. п. 1-4 та аналогічних їм) на постійних робочих місцях у виробничих приміщеннях і на території підприємств	80

В залежності від характеру шуму норми мають свої особливості:

– для широкосмугового постійного шуму варто приймати значення, наведені в таблиці 6;
– для тонального та імпульсного шуму припустимі значення на 5 дб нижче, ніж наведені;
– для шуму створюваного приладами опалення, вентиляції та кондиціонування повітря значення мають бути на 5 дб менше фактичного рівню шуму у приміщенні;
– для переривчастого шуму рівень звуку має бути не більшим 110 дбА;
– для імпільсного шуму рівень звуку повинен бути нижчим за 125 дБА.

Для розрахунку рівня звуку на робочому місці скористаємося даними з таблиці 7.

Таблиця 7 – Вхідні дані для розрахунку рівня шуму

Призначення приміщення	Джерела шуму		
	Шумові характеристики, дБА		
Офіс	Комп’ютер	Кондиціонер	Інше обладнання
	43	39	35

Загальний рівень шумового тиску визначається за наступною формулою:

$$L = 10lg \sum_{i=1}^{n} 10^{0,1*L_i}$$

де

n – кількість джерел шуму,

L_i – рівень шумового тиску і-го джерела звуку.

Після підстановки даних про звуковий тиск обладнання у формулу, отримуємо:

$$L = 10\lg(10^{0,1\cdot 43} + 10^{0,1\cdot 39} + 10^{0,1\cdot 35}) = 44,92 \text{ (дБА)}$$

Отримане значення порівнюємо із нормами, наведеними у таблиці 8.

Для роботи програміста рівень шуму не має перевищувати 50 дБА. А отже розраховане значення 44,92 дБА вкладається в норму та відповідає вимогам ОП.

2 Безпека життєдіяльності при роботі в офісі

2.1 Перша допомога при ураженні струмом

При щоденній праці з електроприладами існує ризик ураження електричним струмом, тому важливо пам’ятати алгоритм дій в такій ситуації.

Ураження струмом може бути наслідком порушення техніки безпеки при роботі з електроприладами або статися при несправності обладнання. Причиною ураження може бути дотик безпосередньо до провідника або джерела струму або непряме дотикання – через електричну дугу.

Електротравма – це травма, що виникає внаслідок дії на організм струму або електричної дуги.

При проходженні через організм людини струм спричинює термічну, біологічну та електролітичну дії.

Внаслідок термічної дії струму з'являються опіки, відбувається ураження судин, нервових клітин, головного мозку, серця. Електролітична дія призводить до розкладання органічної рідини та крові, порушення їх складу. Біологічна дія проявляється у подразненні тканин організму. [1]

Серйозність травми залежить від наступних чинників:

– сили струму;
– тривалості дії струму на організм;
– шляху проходження струму через тіло людини;
– площі контакту з струмопровідними частинами;
– індивідуальних особливостей організму.

В більшості випадків ураження струмом викликає в потерпілого мимовільне скорочення м’язів, через що людина не може сама звільнитися від дії електричного струму. Тому критично важливим кроком при допомозі ураженому

є відключення від живлення тієї частини електроустановки, до якої він дотикається. Якщо відключення неможливе, потрібно відокремити потерпілу людину від струмопровідного елементу, застосовуючи підручний ізоляційний матеріал. При цьому слід пам’ятати про власну безпеку, адже дотик до ураженої людини без належних запобіжних заходів може становити загрозу життю.

Після звільнення людини від контакту зі струмом, слід провести наступні дії відповідно до стану постраждалого:

– якщо людина в свідомості, але до цього була непритомною, або тривалий час знаходилася під струмом, треба забезпечити їй повний спокій до прибуття швидкої або терміново відвести у лікарню;

– якщо потерпілий не в свідомості, але зберігає дихання, його треба покласти рівно, розстібнути одяг, забезпечити надходження свіжого повітря, дати понюхати нашатирний спирт, збризкувати тіло водою, розтирати та зігрівати до прибуття лікаря;

– якщо постраждалий не дихає, або дихає дуже слабко (зрідка, судорожно), йому необхідно провести штучне дихання до появи медиків. У разі зупинення серцебиття потрібно провести зовнішній масаж серця.

Потерпілого обов’язково треба відправити до медичного закладу для огляду, навіть якщо ураження не здається значним. Наслідки електротравми можуть виникати не одразу, а навіть через 2-3 години після ураження. Своєчасна допомога дозволяє уникнути ускладнень після електротравми.

2.2 Заходи пожежної безпеки

Відповідно до щорічних аналізів, до найпоширеніших причин виникнення пожежі відносять наступні:

– порушення правил пожежної безпеки, а також норм і правил у технологічних процесів;
– невідповідне облаштування систем вентиляції, опалення, електропостачання;
– порушення правил зберігання легкозаймистих та несумісних матеріалів;

– використання відкритого вогню в непризначених для цього місцях;
– відсутність протипожежного водозабезпечення, систем пожежної сигналізації, первинних засобів пожежегасіння;
– погана обізнаність персоналу в питаннях пожежної безпеки;
– порушення інструктажу з пожежної безпеки підчас виконання робіт.

Для запобігання пожежам на підприємствах слід здійснити ряд організаційних та технічних заходів. До організаційних заходів відносять:

– розробка інструктажів з пожежної безпеки, правил поводження та інструкцій;
– проведення інструктажів, навчання персонала основ пожежної безпеки;
– контроль за дотриманням норм та правил;
– розробка плану евакуації, способу оповіщення працівників у разі небезпеки;
– організація належного пожежного нагляду за об’єктами;
– регулярна перевірка стану пожежного інвентарю.

Серед технічних заходів пожежної безпеки виділяють такі:

– облаштування приміщень підприємства відповідно до затверджених норм та правил;
– підтримання справного стану системи вентиляції, опалення, електромережі, обладнання;
– встановлення пожежної сигналізації, систем автоматичного пожежогасіння.

Виконання усіх наведених протипожежних заходів обов’язкове та може зберегти життя та здоров’я працівників.

ВИСНОВОК

Під час написання спеціальої частини з охорони праці були проаналізовані умови праці на робочому місці розробника програмного забезпечення.

Розглянуті основні параметри мікроклімату у робочому приміщенні, такі як температура, вологість та швидкість руху повітря, наведені їх норми для роботи в офісі. Для роботи програміста особливо важливе значення має якісне освітлення, оскільки праця супроводжується постійною зоровою напругою, тому приведено норми освітленості та спосіб розрахунку необхідної кількості світильників в залежності від параметрів офісу. Вагому роль для можливості концентрації під час розробки ПЗ відіграє також рівень шуму. Наведено спосіб визначення рівня звукового тиску та вимоги до рівня шуму різного характеру на робочому місці. Розглянуто вплив електромагнітних випромінювань на здоров’я людини та варіанти планування робочого місця в офісі для мінімізації шкідливої дії електромагнітних полів.

Серед небезпек, пов’язаних із роботою в офісі, слід виділити можливе ураження струмом та пожежу. Було розглянуто першу допомогу постраждалому від ураження струму, а також запобіжні заходи та дії при пожежі.

Отже, було розглянуто основні аспекти охорони праці та безпеки життєдіяльності, пов’язані з працею розробника програмного забезпечення.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Винокурова Л. Е., Основи охорони праці. 2001. 192 р.

2. Санітарні норми мікроклімату виробничих приміщень | від 01.12.1999 № 42. URL: https://zakon.rada.gov.ua/rada/show/va042282-99#Text (accessed 02/06/2022).

3. Санітарні норми виробничого шуму ультразвуку та інфразвуку | від 01.12.1999 № 37. URL: https://zakon.rada.gov.ua/rada/show/va037282-99#Text (accessed 02/06/2022).

4. Про затвердження пожежної безпеки в Україні | від 30.12.2014 № 1417. URL: https://zakon.rada.gov.ua/laws/show/z0252-15#Text (accessed 31/05/2022).

5. ДСаНПіН 3.3.2.007-98. ДСанПіН 3.3.2.007-98 Державні санітарні правила та норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин. (40939). URL: https://dnaop.com/html/40939/doc-ДСаНПіН_3.3.2.007-98 (accessed 02/06/2022).

6. Про охорону праці | від 14.10.1992 № 2694-XII. URL: https://zakon.rada.gov.ua/laws/show/2694-12#Text (accessed 02/06/2022).

7. Про затвердження Вимог щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями | від 14.02.2018 № 207. URL: https://zakon.rada.gov.ua/laws/show/z0508-18#Text (accessed 02/06/2022).