Київський національний університет імені Тараса Шевченка
Факультет інформаційних технологій
Кафедра програмних систем і технологій

УДК 004.9 *На правах рукопису*

ВИПУСКНА КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

Тема**:** “Проектування та розробка серверної частини веб застосування для ведення бази даних публікацій”

Спеціальність – 121 “Інженерія програмного забезпечення”

ПОЯСНЮВАЛЬНА ЗАПИСКА

Студент
ІПЗ-41__________/Олександр МУСТАФАЄВ/
Науковий керівник
к.ф.м.н.,доц.___________/ Сергій ПОЛЯКОВ/
Консультант
з питань нормоконтролю
фахівець___________/ Тамара ЧАПОВСЬКА/

Допускається до захисту
Завідувач кафедри
д.т.н.,доц.______________ /Олексій БИЧКОВ/

Київ – 2021

Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій

Кафедра програмних систем і технологій

Освітньо-кваліфікаційний рівень бакалавр

Спеціальність 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖЕНО

Зав. кафедри програмних систем і технологій_

_______________________(Олексій БИЧКОВ)

(підпис) **(прізвище та ініціали)**

ЗАВДАННЯ

НА ВИПУСКНУ КВАЛІФІКАЦІЙНУ БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТУ

Мустафаєву Олександру Серверовичу

---

1. Тема бакалаврської роботи “ Проектування та розробка серверної частини веб застосування для ведення бази даних публікацій ”, керівник проекту (роботи) Поляков Сергій Анатолійович, к.ф.-м.н., доцент___________________________ затверджені наказом вищого навчального закладу від “ ___ ” ________ 2021 р. № ______________

2. Строк подання студентом роботи _________ 2021 р.

3. Вихідні дані до проекту (роботи): Базові концепції та розуміння проектування та розробки програмних технологій

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Вибір середи розробки та її основні переваги

2. Проектування бази даних та допоміжні засоби для цього

3. Використання контролерів

5. Перелік графічного матеріалу

1. Абстрактна схема між базою даних та моделями (рис. 1.1, ст. 15)

2. Візуальна модель MVC (рис. 1.2, ст. 16)

3. Моделювання бази даних (рис.1. 3, ст. 17)

4. Додавання можливості міграцій (рис. 1.4, ст. 18)

5. Додавання міграції “Create” (рис. 1.5, ст. 23)

6. Таблиця користувачів у базі даних (рис. 2.1, ст. 33).

7. Таблиця ролей у базі даних (рис. 2.2, ст. 34)

8. Таблиця зав’язків користувачів та ролей (рис. 2.3, ст. 35)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Проектування бази даних	Поляков Сергій Анатолійович		

7. Дата видачі завдання _____________________ 2021 р.

Керівник ___________________________ (Сергій ПОЛЯКОВ)

Завдання прийняв до виконання _________________ (Олександр МУСТАФАЄВ)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	**Назва етапів бакалаврської роботи**	**Термін виконання етапів роботи**	**Примітка**
1	Підбір та вивчення літератури	10.01.2021	Виконано
2	Аналіз можливих алгоритмів	15.01.2021	Виконано
3	Вивчення додаткових бібліотек	13.02.2021	Виконано
4	Розробка алгоритмічної моделі	17.02.2021	Виконано
5	Опис розробленого алгоритму	25.03.2021	Виконано
6	Затвердження пояснювальної записки роботи завідуючого кафедри	17.04.2021	Виконано

Студент – бакалавр ___________________(Олександр МУСТАФАЄВ)

Керівник роботи ___________________(Сергій ПОЛЯКОВ)

АНОТАЦІЯ

Випускна кваліфікаційна бакалаврська робота: 70 с., 8 рис.,2 додат., 4 джерела.

Тема: Проектування та розробка серверної частини веб застосування для ведення бази даних публікацій

Об'єкт дослідження: публікації різних авторів та додаткові закріплені дані до публікації.

Мета роботи: розробка серверної частини для публікацій для різних авторів.

Предмет дослідження: технологія серверної частини та обробки різних видів запитів на сервер з подальшою відповіддю.

Результати дослідження: Досліджено можливості застосування серверної частини веб застосування для подальшої повноцінної роботи клієнтської частини. Запропонований логічний підхід для вирішення цієї задачі.

Висновок

В результаті досліджень була розроблена та протестована повністю функціонуюча програма, яка здатна оброблювати вхідні та вихідні дані та працювати з базою даних та клієнтом (клієнтською частиною).

БАЗА ДАНИХ, ПРОГРАМА, ПРОЕКТУВАННЯ, РОЗРОБКА, СЕРВЕР, ЗАПИТ, ПРОГРАМУВАННЯ, ПРОТОКОЛ.

SUMMARY

Final qualifying bachelor's thesis: 70 pages, 8 pictures, 2 appendices, 4 sources.

Topic: Design and development of the server part of a web application for maintaining a database of publications

Object of research: publications of various authors and additional attached data to the publication.

Purpose: development of a server part for publications for different authors.

Subject of research: technology of the server part and processing of various types of requests to the server with the subsequent answer.

Research results: The possibilities of using the server part of the web application for further full-fledged work of the client part are investigated. A logical approach to this problem is proposed.

Conclusion

Because of research, a fully functional program was developed and tested, which is able to process input and output data and work with the database and the client (client part).

DATABASE, PROGRAM, DESIGN, DEVELOPMENT, SERVER, REQUEST, PROGRAMMING, PROTOCOL.

АННОТАЦИЯ

Выпускная квалификационная бакалаврская работа: 70 с., 8 рис., 2 доп., 4 источника.

Тема: Проектирование и разработка серверной части веб приложение для ведения базы данных публикаций

Объект исследования: публикации различных авторов и дополнительные закреплены данные к публикации.

Цель работы: разработка серверной части для публикаций для различных авторов.

Предмет исследования: технология серверной части и обработки различных видов запросов на сервер с последующим ответом.

Результаты исследования: Исследованы возможности применения серверной части веб приложение для дальнейшей полноценной работы клиентской части. Предложенный логический подход для решения этой задачи.

Вывод

В результате исследований была разработана и протестирована полностью функционирующая программа, которая способна обрабатывать входящие и исходящие данные и работать с базой данных и клиентом (клиентской частью).

БАЗА ДАННЫХ, ПРОГРАММА, ПРОЕКТИРОВАНИЯ, РАЗРАБОТКИ, СЕРВЕР, ЗАПРОС, ПРОГРАММИРОВАНИЕ, ПРОТОКОЛ.

ЗМІСТ

Стр.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ……………………...……………………………9

ВСТУП……………………………………………………………………………..10

РОЗДІЛ 1

РОЗРОБКА РЕЛЯЦІЙНОЇ МОДЕЛІ ДАНИХ

1.1 Бібліотеки та технології для розробки бази даних…………………………….13

1.2 Проектування бази даних………………………………………………………16

1.3 Міграції…………………………………………………………………………..18

РОЗДІЛ 2

РОЗРОБЛЕННЯ КОНТРОЛЕРІВ ТА ЛОГІЧНОЇ ЧАСТИНИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Створення контролера для реєстрації та авторизації користувача…………...25

2.2 Створення контролера авторів та публікацій……………………………………38

ВИСНОВОК…………………………………………………….………………51

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ…….…………….…………………...52

ДОДАТКИ……………………………………………………………………………53

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

БД - база даних

ООП - об’єктивно орієнтоване програмування

MVC - model view controller

EF - entity framework

ПЗ - програмне забезпечення

VS - visual studio

ВСТУП

Актуальність роботи

Дане програмне забезпечення на сьогоднішні часи є актуальним, починаючи від стеку технологій, які будуть входити в розробку програмного забезпечення та закінчуючи самою ідеєю та темою роботи.

Багато сучасних компаній використовують вибрані фреймворки та стек технологій, які використовуються у розробці ПЗ.

Список прикладних областей, де вона застосовується дуже широкий.

Порівняння роботи з відомими розв’язаннями проблеми

Саме такого рішення під дану проблему розробки програмного забезпечення не існує, тому наше рішення є оригінальним.

Хоча існує мільйони програм, написані іншими користувачами, які розгортають ASP.NET додатки або тільки серверну частину WEB API , але кожний додаток є особливим та оригінальним, бо навіть невеликі зміни у базі даних зачіпають логіку всієї програми.

У даній роботі були використані основні переваги та рекомендації інших розробників з різних сайтів та відео та лекцій. Використання наступних алгоритмів показує високу ефективність та працездатність ПЗ.

Мета і задачі дослідження

Метою бакалаврської роботи є проектування та розробка серверної частини веб застосування для ведення бази даних публікацій для подальшої взаємодії з ПЗ збоку клієнтської частини або мобільної версії.

Досліджувана модель повинна мати можливість взаємодіяти декількома способами та можливість виконання таких **задач**:

- Отримання даних
- Отримання даних, а саме одного об’єкта з усього списку
- Додавання даних
- Редагування даних
- Видалення даних

Особливу увагу потрібно приділити останнім двом пунктам, оскільки вони мають деякі обмеження щодо їх подальшого використання та контролю робото спроможності серверної частини.

Об’єктом дослідження є публікації різних авторів, представлені у вигляді реляційних структур даних.

Предметом дослідження є технологія розробки серверної частини WEB API, яка має змогу приймати та передавати дані багатьма способами.

Методи дослідження

Для роботи з програмним забезпеченням були використані методи моделювання реляційних баз даних та використання нормальних форм проектування реляційних структур даних.

Практичне значення одержаних результатів

Отримана реалізація розробленої моделі реляційних структур даних та основна логіка робото спроможності програми дає можливість для подальшого розвитку програми та перехід її на Full Stack розробку.

Тобто це дає можливість спростити подальшу обробку даних та взаємодію з ними. Також дає можливість коректного представлення їх у спеціальному вигляді.

Особистий внесок студента

Основним результатом є:

1. Авторський та оригінальний підхід до розробки реляційних структур даних (бази даних).
2. Розроблення особистих алгоритмів між базою даних та результатом роботи ПЗ.

Структура та обсяг роботи даних

Робота викладена на 68 сторінках друкованого тексту. Робота містить 12 таблиць, 32 рисунки та 3 додатки, обсягом 12 стор.

РОЗДІЛ 1

РОЗРОБКА РЕЛЯЦІЙНОЇ МОДЕЛІ ДАНИХ

1.1 Бібліотеки та технології для розробки бази даних

Даний проект буде написаний на платформі Visual Studio.

Visual Studio - серія продуктів фірми Майкрософт, які містять інтегроване середовище розробки програмного забезпечення та низку інших інструментальних засобів. Ці продукти дають змогу розробляти як консольні програми, так і програми з графічним інтерфейсом, включно з підтримкою технології Windows Forms, а також веб-сайти, веб-застосунки, веб-служби як в рідному, так і в керованому кодах для всіх платформ, що підтримуються Microsoft Windows, Windows Mobile, Windows Phone, Windows CE, .NET Framework, .NET Compact Framework та Microsoft Silverlight, WEB API, ASP.NET.

Для проектування бази даних ми будемо використовувати додатковий NuGet пакет для роботи із вбудованою базою даних для простоти роботи оглядач об'єктів SQL Server на Localhost (щоб не користуватися сторонньою базою даних (MS SQL, MY SQL і т.д.)).

Для полегшення взаємодії між WEB API та базою даних я в подальшому буду використовувати ORM технологію.

Об'єктно-реляційне відображення (ORM, O / RM та O / R mapping tool) в інформатиці - це техніка програмування для перетворення даних між системами несумісного типу за допомогою об'єктно-орієнтованих мов програмування. Це по суті створює "базу даних віртуального об'єкта", яка може використовуватися в мові програмування. Доступні як безкоштовні, так і комерційні пакети, які виконують об'єктно-реляційне відображення, хоча деякі програмісти вирішують створювати власні інструменти ORM.

В об'єктно-орієнтованому програмуванні завдання управління даними діють на об'єкти, які майже завжди є нескалярними значеннями. Наприклад, розглянемо запис адресної книги, який представляє одну особу разом із нулем або більше номерів телефонів та нулем або більше адрес. Це може бути змодельовано в об'єктно-орієнтованій реалізації "Об'єктом особи" з атрибутом / полем для зберігання кожного елемента даних, що містить запис: ім'я людини, список телефонних номерів та список адрес. Сам список телефонних номерів міститиме "PhoneNumber objects" тощо. Кожен такий запис адресної книги мовою програмування трактується як єдиний об’єкт (наприклад, на нього може посилатися одна змінна, що містить вказівник на об’єкт). З об’єктом можуть бути пов’язані різні методи, наприклад методи повернення бажаного номера телефону, домашньої адреси тощо.

На відміну від цього, багато популярних продуктів баз даних, таких як системи управління базами даних SQL (СУБД), не є об'єктно-орієнтованими і можуть зберігати та маніпулювати скалярними значеннями, такими як цілі числа та рядки, організовані в таблицях. Програміст повинен або перетворити значення об'єкта на групи простіших значень для зберігання в базі даних (і перетворити їх назад при отриманні), або використовувати лише прості скалярні значення всередині програми. Об'єктно-реляційне відображення реалізує перший підхід.

Суть проблеми полягає у перекладі логічного подання об’єктів у атомізовану форму, яка може зберігатися в базі даних, зберігаючи властивості об’єктів та їх взаємозв’язки, щоб їх можна було перезавантажити як об’єкти, коли це потрібно. Якщо ця функція зберігання та пошуку реалізована, об’єкти вважаються стійкими.

Одним з популярних варіантів є Entity Framework.

Entity Framework - це набір технологій в ADO.NET, які підтримують розробку програмно-орієнтованих на дані програмних додатків. Архітектори та розробники програм, орієнтованих на дані, зазвичай борються з необхідністю досягнення двох дуже різних цілей. Вони повинні моделювати сутності,

взаємозв'язки та логіку бізнес-проблем, які вони вирішують, а також повинні працювати з движками даних, що використовуються для зберігання та отримання даних.

Дані можуть охоплювати кілька систем зберігання, кожна з яких має власні протоколи; навіть програми, які працюють з єдиною системою зберігання, повинні збалансувати вимоги системи зберігання та вимоги щодо написання ефективного та ремонтопридатного коду програми.

EF є прошарком між базою даних та об’єктами у самій системі C# (Рис. 1.1).

Рис. 1.1 абстрактна схема між базою даних та моделями з використанням ORM

1.2 Проектування бази даних

У розробці проекту я буду використовувати патерн MVC (Model-View-Controller).

Моде́ль–вигляд–контро́лер (Рис. 1.2) (MVC, Модель–представлення–контролер, англ. Model-view-controller) — архітектурний шаблон, який використовується під час проектування та розробки програмного забезпечення.

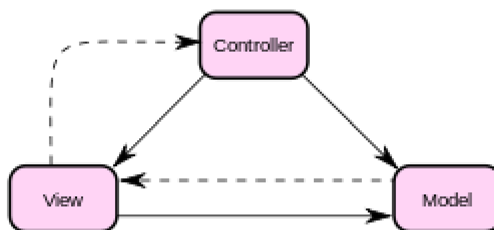


Рис. 1.2 візуальна модель MVC

Цей шаблон передбачає поділ системи на три взаємопов'язані частини: модель даних, вигляд (інтерфейс користувача) та модуль керування. Застосовується для відокремлення даних (моделі) від інтерфейсу користувача (вигляду) так, щоб зміни інтерфейсу користувача мінімально впливали на роботу з даними, а зміни в моделі даних могли здійснюватися без змін інтерфейсу користувача.

Мета шаблону — гнучкий дизайн програмного забезпечення, який повинен полегшувати подальші зміни чи розширення програм, а також надавати можливість повторного використання окремих компонентів програми. Крім того використання цього шаблону у великих системах сприяє впорядкованості їхньої структури і робить їх більш зрозумілими за рахунок зменшення складності.

Для проектування нашої моделі бази даних будемо використовувати такі основні сутності:

- Користувач
- Права користувача
- Автор
- Публікація
- Група
- Видавництво
- Ключові слова
- Конференція
- Журнал
- Тип публікації
- Опис публікації

Перенесемо їх на модель бази даних для кращого розуміння (Рис. 1.3).

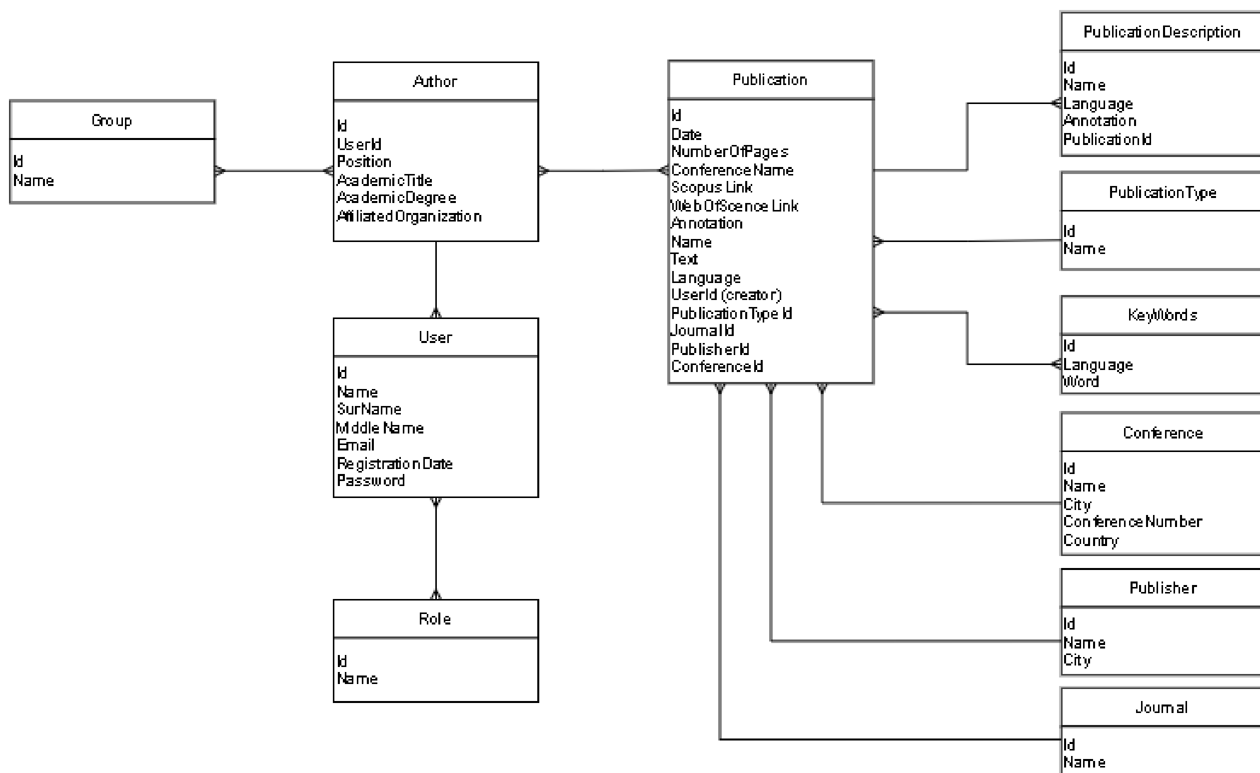


Рис. 1.3 моделювання бази даних

1.3 Міграції

Тепер створимо ці моделі у Visual Studio.

В цьому нам допоможе саме EF, бо він допомагає перенести нашу модель з коду на C# у SQL таблиці. Для початку потрібно встановити в NuGet пакетах Microsoft Entity Framework.

У розробці я буду використовувати метод розробки Code First.

По-перше нам потрібно підключити Migration (міграції) у наш проект. Вони потрібні для того щоб ми могли в подальшому змінювати об'єкти у Entity, а за допомогою міграції ми можемо переносити ці зміни на моделі у базі даних.

А сама папка з міграціями допомагає нам відслідковувати зміни, які відбулися у процесі розробки ПЗ.

Підключення потрібно робити у *консолі диспетчера пакетів* (Рис. 1.4).

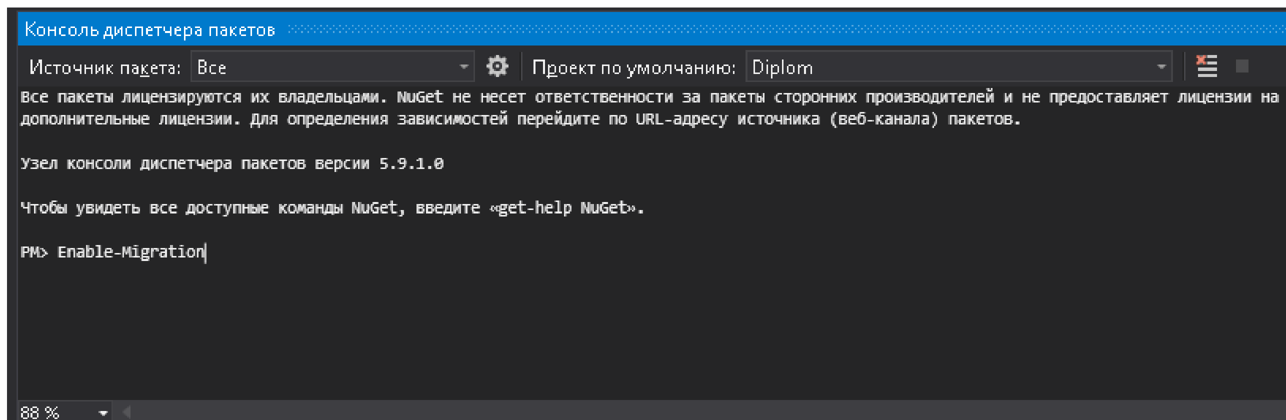


Рис. 1.4 додавання можливості міграцій

Далі створимо модель, наприклад, публікацій.

```
public class PublicationModels
    {
```

```
public int Id { get; set; }
[Display(Name = "Дата")]
public DateTime Date { get; set; }
[Display(Name = "Количество страниц")]
public int NumberOfPages { get; set; }
[Display(Name = "Название конференции или журнала или издательства")]
public string ConferenceName { get; set; }
[Display(Name = "Ссылка на Sqopus")]
public string SqopusLink { get; set; }
[Display(Name = "Ссылка на Web Of Science")]
public string WebOfScienceLink { get; set; }
[Display(Name = "Аннотации")]
public string Annotation { get; set; }
[Display(Name = "Название")]
public string Name { get; set; }
[Display(Name = "Текст")]
public string Text { get; set; }
[Display(Name = "Язык")]
public string Language { get; set; }
[Required]
[Display(Name = "Id создателя публикации")]
public string UserId { get; set; }


[Display(Name = "Id типа публикации")]
public int PublicationTypeId { get; set; }
public PublicationTypeModels PublicationType { get; set; }


[Display(Name = "Id журнала")]
public int? JournalId { get; set; }
public JournalModels Journal { get; set; }

[Display(Name = "Id издательства")]
public int? PublisherId { get; set; }
public PublisherModels Publisher { get; set; }

[Display(Name = "Id конференции")]
public int? ConferenceId { get; set; }
public ConferenceModels Conference { get; set; }

public virtual ICollection<PublicationDescriptionModels> Descriptions { get;
set; }
public virtual ICollection<AuthorModels> Authors { get; set; }
```

```
public virtual ICollection<KeyWordModels> KeyWords { get; set; }
public PublicationModels()
{
  KeyWords = new List<KeyWordModels>();
  Authors = new List<AuthorModels>();
  Descriptions = new List<PublicationDescriptionModels>();
}

}
```

На цьому прикладі ми бачимо створення моделі з деякими атрибутами над кожним полем. Для цього потрібно підключити бібліотеку System.ComponentModel.DataAnnotations.

Вона надає класи атрибутів, які використовуються для визначення метаданих для ASP.NET MVC і елементів управління даними ASP.NET (а також для використання у WEB API).

Тепер детальніше розберемо та зробимо пояснення, що робить кожен з атрибутів у даному коді (або у інших кодах при створені різних моделей).

- [Display(Name = "")]

  Цей елемент надає атрибут загального призначення, який дозволяє вказати локальні рядки для типів і членів поділюваних класів сутностей.
- [Required]

  Цей атрибут вказує на те, що дане поле має бути обов’язково заповненим, тобто не повинно бути null.
- [StringLength]

  Такий атрибут можна використовувати у тому випадку, коли ми хочемо конкретно задати мінімальний розмір поля (MinimumLength), максимальний розмір (вказується першим), повідомлення у разі помилки (ErrorMessage).
- [Datatype]

  Задає ім'я додаткового типу, який необхідно пов'язати з полем даних (наприклад паролю).

Також у Entity Framework обов'язково потрібно вказувати Id (ідентифікатор), бо із-за цього виникне помилка при міграції нашої моделі і ми не зможемо створити базу даних, але можливо робити і без Id тільки у таблиця із зіставними ідентифікаторами, наприклад при зв'язку багато до багатьох.

Модифікатор Id позначається так:

```
public int Id { get; set; }
```

Тут ми вказуємо модифікатор доступу, тип змінної моделі, назву змінною, а також можливі варіанти взаємодії з змінною.

Get set-це модифікатори доступу до властивості. Get зчитує поле властивості. Set встановлює значення властивості. Get-це як доступ тільки для читання. Set-це як доступ тільки для запису. Щоб використовувати властивість, як писати читати як Get і set повинні бути використані.

Також потрібно після створення моделей створити клас, який буде унаслідуваний від класу "IdentityDbContext".

Для взаємодії з базою даних в просторі імен Microsoft.AspNet.Identity.EntityFramework визначено клас IdentityDbContext. Це звичайний контекст даних, похідний від DbContext, який вже містить властивості, необхідні для управління користувачами і ролями: властивості Users і Roles.

Хоча в реальному додатку ми будемо мати справу з класами, похідними від IdentityDbContext.

Назвемо наш клас ApplicationDbContext. Тепер ми напишемо всі наші таблиці у цьому класі для того щоб EF зміг перенести ці таблиці у базу даних, для подальшою взаємодії з цими тамблицями.

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
    {
        public DbSet<AuthorModels> Author { get; set; }
```

```
public DbSet<ConferenceModels> Conference { get; set; }

public DbSet<GroupModels> Group { get; set; }



public DbSet<JournalModels> Journal { get; set; }

public DbSet<KeyWordModels> KeyWord { get; set; }

public DbSet<PublicationDescriptionModels> PublicationDescriptions { get; set;
}


public DbSet<PublicationModels> Publications { get; set; }

public DbSet<PublicationTypeModels> PublicationType { get; set; }

public DbSet<PublisherModels> Publisher { get; set; }

public ApplicationDbContext()

   : base("DefaultConnection", throwIfV1Schema: false)

{

}

public static ApplicationDbContext Create()

{

   return new ApplicationDbContext();

}

}
```

Метод “DbSet” являється указом для EF що потрібно створили таблицю, далі у “<>” ми вказуємо назві моделі.

Потім вказуємо назву майбутньої таблиці в базі даних. Поля “{get;set;}” просто вказують на те, що дані про створення таблиці є властивостями.

Після додавання таблиць із моделей в IdentityDbContext, відповідно до кожної з таблиць у діаграмі бази даних, створюю міграцію нашої моделі.

Міграції допомагають змінювати базу даних при зміні моделі у самому коді програми.

Для цього у консолі диспетчера пакетів (PowerShell Window) додаємо нову міграцію за допомогою команди add-migration (Рис. 1.5).

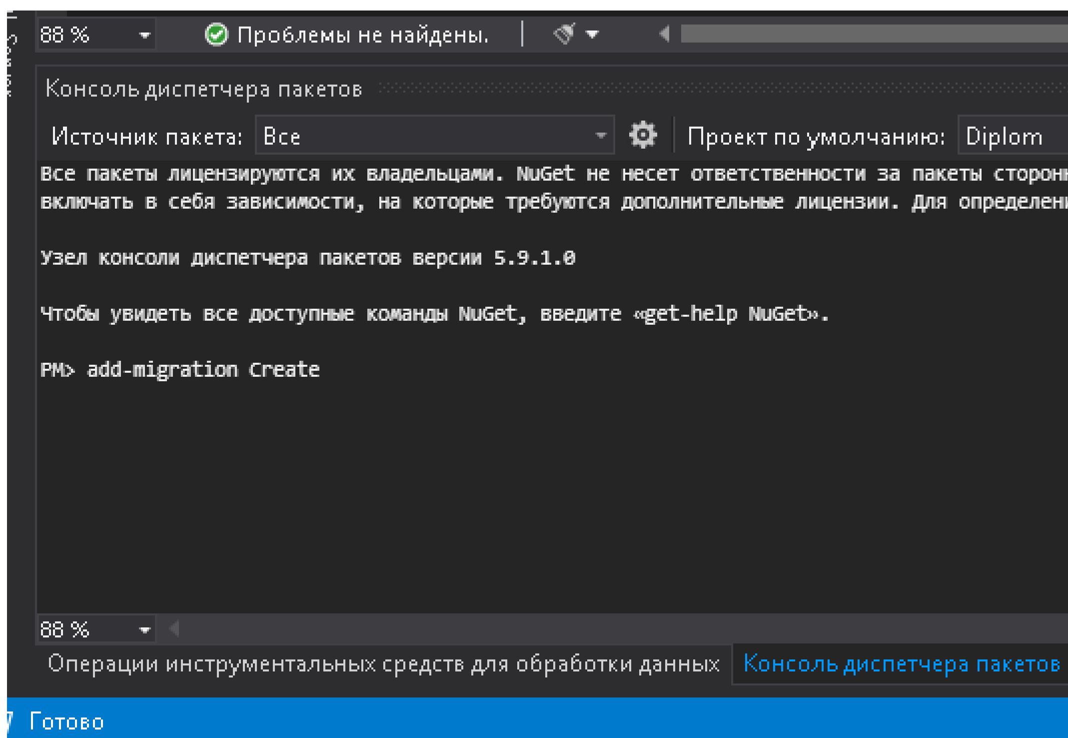


Рис. 1.5 додавання міграції “Create”

Після запуску даного коду Entity Framework перевірить спочатку, чи була раніше задіяна Enable-Migrations частина, а вже потім перейде до папки Models і перевірить зміни у кожній з моделей.

У нашому випадку ми робимо це вперше, тому EF просто зчитає дані з різних моделей та конвертує їх у скрипт до бази даних.

Всю цю конвертацію ми зможемо побачити у папці Migrations, а далі у класі з назвою нашої міграції (Create).

РОЗДІЛ 2

РОЗРОБЛЕННЯ КОНТРОЛЕРІВ ТА ЛОГІЧНОЇ ЧАСТИНИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Створення контролера для реєстрації та авторизації користувача

Завдяки тому, що при створенні проекту ми з самого початку вказали використання авторизації, яка вмонтована у WEB API, то тепер не потрібно прописувати все вручну. Тому у папці Controllers вже існує клас AccountController.

Перед класом AccountController ми додаємо атрибути:

- [Authorize]

  Знаходиться в просторі імен System.Web.Http. Він одночасно перевіряє, аутентифікований користувач в системі, і також перевіряє права користувача на доступ до цього ресурсу (якщо даний ресурс має на увазі доступ тільки для певних ролей иили користувачів). Якщо користувач нерозпізнаних, то система посилає у відповідь клієнту статусний код 401 (Unauthorized).
- [RoutePrefix("api/Account")]

  Маршрутизація - це те, як веб-API узгоджує URI з дією. Web API 2 підтримує новий тип маршрутизації, який називається атрибутною маршрутизацією. Як випливає з назви, маршрутизація атрибутів використовує атрибути для визначення маршрутів. Маршрутизація атрибутів дає вам більше контролю над URI у вашому веб-API. Наприклад, ви можете легко створити URI, що описують ієрархію ресурсів.

  Раніше стиль маршрутизації, званий маршрутизацією на основі

конвенції, все ще повністю підтримується. Насправді ви можете поєднати обидві техніки в одному проекті.

- [EnableCors] - це стандарт консорціуму W3C, що дозволяє сервера пом'якшити ту ж політику. За допомогою CORS сервер може явно дозволити деякі запити незалежно від джерела, а інші - відхилити. CORS безпечніше і більш гнучкий, ніж більш ранні методики, такі як JSONP.

Перейдемо до першого пункту, тобто спочатку користувач має зареєструватися для подальшої роботи.

Для цього в нас створений метод public async Task<IHttpActionResult> Register(RegisterBindingModel model).

Також перед цим ми додаємо атрибути:

- [AllowAnonymous]

  Атрибут AllowAnonymous дозволяє вирішити доступ до ресурсів для анонімних, що не авторизованих користувачів. Наприклад, якщо до контролера застосовується атрибут Authorize, то фреймворк спочатку буде дивитися, чи є у користувача необхідні права для доступу до методу контролера.

  Однак в цьому випадку ми потрапляємо у замкнене коло - щоб авторизуватися, треба звернутися до методу Login, але щоб звернутися до цього методу, вже треба бути авторизованим. Тому застосовується атрибут AllowAnonymous, який відкриває публічний доступ до методу контролера.

- [Route("Register")] За зіставлення запитів з певними маршрутами, як і в MVC, відповідає система маршрутизації. Ключовим класом для системи маршрутизації є клас HttpRoutingDispatcher, який обробляє запит для отримання даних про маршрут і додає ці дані в колекцію HttpRequestContext.RouteData. Тобто Route зчитую строку URI та

перенаправляє до відповідного класу, а потім методу у відповідному контролері.

Доступ до цього методу буде встановлюватися методом POST по URL api/Account/Register. У цьому URL api/Account є префіксом, який ми вказували раніше, а /Register це вже назва метода, який ми визиваємо з допомогою метода POST.

Покажемо метод реєстрації:

```
public async Task<IHttpActionResult> Register(RegisterBindingModel model)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }
        //данные для ввода информации о пользователе при регистрации
        var user = new ApplicationUser() { UserName = model.UserName,
            Email = model.Email,
            Name = model.Name,
            Surname = model.Surname,
            MiddleName = model.MiddleName,
            RegistrationDate = DateTime.Now }; //ставим текущее время
        IdentityResult result = await UserManager.CreateAsync(user, model.Password);
        if (!result.Succeeded)
        {
```

```
        return GetErrorResult(result);
      }
      return Ok();
    }
```

Даний метод є асинхронним, це ми можемо побачити за допомогою слова “async” при створенні. Також при використанні даного слова нам потрібно ще вказати “await”.

Асинхронність дозволяє винести окремі завдання з основного потоку в спеціальні асинхронні методи або блоки коду. Особливо це актуально в графічних програмах, де тривалі завдання можу блокувати інтерфейс користувача. І щоб цього не сталося, потрібно задіяти асинхронність. Також асинхронність несе вигоди в веб-додатках при обробці запитів від користувачів, при зверненні до баз даних або мережевих ресурсів. При великих запитах до бази даних асинхронний метод просто засне на час, поки не отримає дані від БД, а основний потік зможе продовжити свою роботу. У синхронному ж додатку, якби код отримання даних знаходився в основному потоці, цей потік просто б блокувався на час отримання даних.

Ключовими для роботи з асинхронними викликами в C # є два ключових слова: async і await, мета яких - спростити написання асинхронного коду. Вони використовуються разом для створення асинхронного методу.

Асинхронний метод володіє наступними ознаками:

У заголовку методу використовується модифікатор async

Метод містить одне або кілька виразів await

Як повертається типу використовується один з наступних:

- void
- Task

- Task <T>
- ValueTask <T>

Асинхронний метод, як і звичайний, може використовувати будь-яку кількість параметрів або не використовувати їх взагалі. Однак асинхронний метод не може визначати параметри з модифікаторами out і ref.

Також варто відзначити, що слово async, яке зазначається в ухвалі методу, що не робить автоматично метод асинхронним. Воно лише вказує, що даний метод може містити одне або кілька виразів await.

IHttpActionResult, який вказаний у якості параметра повернення – це новий тип об'єктів, які використовуються в якості результатів методів. Цей тип являє інтерфейс IHttpActionResult. Класи, що реалізують даний інтерфейс, багато в чому аналогічні класам, похідним від ActionResult в MVC.

- OkResult повертає відповідь HttpStatusCode.OK, тобто статусний код HTTP 200. Створюється за допомогою методу Ok без параметрів
- OkNegotiatedContentResult <T> повертає статусний код HTTP 200, а також результат обробки у вигляді об'єкта типу T. Створюється за допомогою методу Ok, який в якості параметра приймає об'єкт, що повертається клієнту
- NotFoundResult повертає у відповідь статусний код 404. Створюється за допомогою методу NotFound
- ExceptionResult повертає у відповідь статусний код 500, що свідчить про помилку обробки запиту. Генерується методом InternalServerError (Exception), в який передається об'єкт з описом помилки
- UnauthorizedResult повертає у відповідь статусний код 401, який говорить про те, що запит повинен бути аутентифікований. Генерується методом Unauthorized
- BadRequestResult повертає у відповідь статусний код 400. Генерується методом BadRequest () без параметрів

- JsonResult повертає об'єкти в форматі json. Генерується методом Json (T content), який в якості параметра приймає деякий об'єкт типу T. Даний об'єкт серіалізуются в формат json і відправляє в такому вигляді клієнту.
- RedirectResult дозволяє переадресувати виконання запиту. Генерується методом Redirect (string path)

Вхідними параметрами ми візьмемо RegisterBindingModel. RegisterBindingModel є компонентом ViewModel частиною, яку створюють разом з моделями у подальшій розробці програмного забезпечення, коли ми хочемо створити свою оригінальну модель.

Далі показаний код ViewModel частини на RegisterBindingModel.

```
public class RegisterBindingModel
    {
        [Required]
        [Display(Name = "Адрес электронной почты")]
        public string Email { get; set; }
        [Required]
        [Display(Name = "Логин")]
        public string UserName { get; set; }

        [Required]
        [StringLength(100, ErrorMessage = "Значение {0} должно содержать не менее {2} символов.", MinimumLength = 6)]
        [DataType(DataType.Password)]
        [Display(Name = "Пароль")]
```

```
public string Password { get; set; }


[DataType(DataType.Password)]
[Display(Name = "Подтверждение пароля")]
[Compare("Password", ErrorMessage = "Пароль и его подтверждение не совпадают.")]
public string ConfirmPassword { get; set; }
[Required]
[Display(Name = "Фамилия")]
public string Surname { get; set; }
[Required]
[Display(Name = "Имя")]
public string Name { get; set; }
[Required]
[Display(Name = "Отчество")]
public string MiddleName { get; set; }
}
```

Тут ми бачимо тільки основні поля, які присутні у моделі реєстрації. Також у моделі користувача є дата реєстрації, яку ми встановимо автоматично в коді програми.

Далі в нас йде

```
if (!ModelState.IsValid)
    {
```

```
return BadRequest(ModelState);
}
```

Цей блок коду, а саме ModelState отримує стан моделі після прив'язування моделі, та повертає стан моделі після прив'язування моделі.

Далі йде саме створення нової моделі (запису нового рядка у базі даних нашого програмного забезпечення).

```
var user = new ApplicationUser() { UserName = model.UserName,
        Email = model.Email,
        Name = model.Name,
        Surname = model.Surname,
        MiddleName = model.MiddleName,
        RegistrationDate = DateTime.Now };
```

Тут ми на модель ApplicationUser перенесемо дані з моделі, яку ми створювали раніше у ViewModel.

А у полі RegistrationDate = DateTime.Now, ми кажемо, що час реєстрації дорівнює поточному часу.

У наступному полі

```
IdentityResult result = await UserManager.CreateAsync(user, model.Password);
```

Ми створюємо користувача та записуємо результат нашої роботи у змінну result.

Тепер ми перевіряємо результат додавання нового користувача.

```
if (!result.Succeeded)
    {
        return GetErrorResult(result);
```

}

У разі якоїсь помилки, ми повертаємо результат Error.

У разі успішного виконання ми повертаємо результат, зо все добре (return Ok();).

Після реєстрації нового користувача, до нашої бази даних додається новий користувач (Рис. 2.1).

dbo.AspNetUsers [Данные] AccountController.cs

Максимальное количество строк: 1000

	Id	Name	Surname	MiddleName	RegistrationDate	Email	EmailConfirmed	Passwo
▷	e69128ea-c5f5-...	Aleksandr	Mustafaev	Sergeevich	29.04.2021 14:2...	amustik2000@...	False	ABpPge
	e88b2c39-f806-...	ThisIsTestName	ThisIsTestSurna...	ThisIsTestMiddl...	28.04.2021 16:1...	TestUserEmail...	False	ANmLq
	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Рис. 2.1 таблиця користувачів у базі даних

На цьому рисунку ми можемо побачити що на даний момент у базі існує два користувача. Також у кожного користувача є роль (Рис. 2.2), яку ми можемо використовувати для того щоб надати користувачам різні ролі, наприклад роль адміністратора для розширених можливостей у напрямку взаємодії з нашим програмним забезпеченням.

Id	Name
1	Администратор
2	Тестировщик
3	TEST
4	Пользователь
5	Гость
6	Проходящий
7	User
8	Стандартный
9	Користувач
NULL	NULL

Рис. 2.2 таблиця ролей у базі даних

Для призначення ролі для користувача ми можемо додати нову роль, в нашому випадку роль адміністратора, а потів встановити залежність у зв’язку багато до багатьох між користувачем та ролями (Рис. 2.3).

dbo.AspNetUserRoles [Данные]   AuthorController.cs

Максимальное количество строк: 1000

	UserId	RoleId
	e69128ea-c5f5-...	1
	e69128ea-c5f5-...	2
	e69128ea-c5f5-...	3
	e69128ea-c5f5-...	4
	e69128ea-c5f5-...	5
	e69128ea-c5f5-...	6
	e69128ea-c5f5-...	7
	e69128ea-c5f5-...	8
	e69128ea-c5f5-...	9
	e88b2c39-f806-...	3
	NULL	NULL

Рис. 2.3 таблиця зав'язків користувачів та ролей

Далі роздивимось наступний крок, який йде у користувача, а саме авторизація на сайті.

Для цього створимо модель токену:

```
public class AddExternalLoginBindingModel
  {
     [Required]
     [Display(Name = "Внешний маркер доступа")]
     public string ExternalAccessToken { get; set; }
  }
```

У WEB API існує багато вмонтованих засобів авторизації, такі як Google auth, Facebook… , але ми будемо використовувати локальну авторизацію та реєстрацію на програмному забезпеченні.

У класі “Startup.Auth.cs” у методі “ConfigureAuth” знайдемо код

```
OAuthOptions = new OAuthAuthorizationServerOptions
    {
        TokenEndpointPath = new PathString("/Token"),
        Provider = new ApplicationOAuthProvider(PublicClientId),
        AuthorizeEndpointPath = new PathString("/api/Account/ExternalLogin"),
        AccessTokenExpireTimeSpan = TimeSpan.FromDays(14),
        // В рабочем режиме задайте AllowInsecureHttp = false
        AllowInsecureHttp = true
    };
```

Тут рядок “TokenEndpointPath = new PathString("/Token")” виконує роль кінцевої точки, тому коли користувач робить запит на “https://localhost:44306/token” то його автоматично перекидає сюди і вже після цього він отримує свій тимчасовий токен, який буде активним при роботі з серверною частиною лише два тижні, як це ми задали у коді раніше.

Після виклику даного маршруту, користувачу буде наданий тимчасовий ключ, для того щоб він зміг користуватися розділами серверної частини за в яких є обмеження щодо того, авторизована людина на сайті чи ні. Тобто вся інформація, яка є, або функціонал (додавання нового запису, видалення даних, зміна будь-якої інформації в базі даних за допомогою серверної частини стане недоступною, та при виклику цих частин коду, ми будемо отримувати помилку, що потрібно виконати авторизацію).

Цей ключ має зберігатися та буде валідним (дійсним) на протязі двох тижнів. Цей параметр часу можливо встановлювати вручну, але ми вибираємо саме такий час, оскільки це є безпечним, та користувачу не потрібно буде кожен раз виконували авторизацію при підключенні до серверу.

2.2 Створення контролера авторів та публікацій

Тож тепер роздивимось нашу схему далі та побачимо, що далі в нас йде таблиця авторів.
Ця таблиця має зберігати основні дані просто авторів та мати посилання на користувачів.

Тобто кожний автор має бути користувачем, але не кожний користувач має бути автором, наприклад може читати публікації.

Кожний контролер має в собі такі основні методи:

- Get

  Робить запит до бази даних, повертає список об'єктів певного типу у форматі JSON.
- Get/id

  Робить запит до бази даних, повертає лише єдиний елемент зі всього списку, який задовольняє вимові this.Id = Id.
- Post

  Робить додавання нової моделі до бази даних.
- Put

  Робить зміну у рядку бази даних по певному Id, тобто задовольняє вимові this.Id = Id.
- Delete

  Видаляє рядок з бази даних, який задовольняє вимові this.Id = Id.

Ми можемо додавати і свої контролери в подальшій розробці.

Тож перейдемо до створення контролеру авторів.

Перед початком класу вказуємо префікс пошуку: [RoutePrefix("api/Author")] та вказуємо [Authorize].

Почнемо з методу GET. Вказуємо маршрутизатор для даного методу [Route("Get")].

Перейдемо до коду даного методу у нашій програмі: public async Task<IEnumerable<AuthorModels>> Get() => await new ApplicationDbContext().Author

.Include(a => a.Publication)

.Include(a=>a.Groups)

.ToListAsync();

Почнемо розбирати по пунктам, що саме ми написали у даному методі. Ключовими для роботи з асинхронними викликами в C # є два ключових слова: **async** і **await**, мета яких - спростити написання асинхронного коду. Вони використовуються разом для створення асинхронного методу.

Task - це є поясненням, що ми будемо використовувати асинхронну задачу. У середині < > ми вказуємо елемент типу, який ми повертаємо.

IEnumerable - надає нумератори, який підтримує простий перебір елементів у зазначеній колекції.

AuthorModels – модель, яку ми повертаємо.

Get – назва методу.

‘=>’**-** У лямбда-виразах оператор лямбда (=>) відокремлює вхідні параметри зліва від тіла лямбда з правого боку. Тобто він зменшує код та прибирає ключове слово ‘return’.

new ApplicationDbContext() – це елемент для роботи з даними, він звертається до бази даних та перетворює її у модель C#, щоб ми взаємодіяли не з самою базою, а через використання Linq у програмному забезпеченні.

Author – це вказівник, що ми будемо звертатися до таблички авторів у нашій базі даних.

Include – він вказує на те, що при завантажені цієї таблиці ми підключаємо ще іншу таблицю за допомогою Foreign Key References у базі даних, наприклад ‘a => a.Publication’ або ‘a=>a.Groups’.

ToListAsync - завдання, яке представляє асинхронну операцію. Результат завдання містить List <T>, що містить елементи з вхідної послідовності (тобто асинхронний ToList<T>). Також використання його можливе лише, якщо до цього ми вказали await.

Далі ми роздивимось метод Get/id.

На початку ми, як і завжди вказуємо допоміжний маршрутизатор, але в даному випадку ми вказуємо Id, як змінну, тому він буде виглядати так ‘[Route("Get/{id}")]’.

В даному випадку ми будемо передавати наш Id у URL.

Перейдемо до коду нашої програми.

```
public async Task<AuthorModels> Get(int id) => await new ApplicationDbContext().Author
        .Include(a => a.Publication)
        .Include(a => a.Groups)
        .Where(a=>a.Id==id)
        .FirstOrDefaultAsync();
```

Цей метод не сильно відрізняється від минулого, бо він має все теж саме, окрім одного метода розширення LINQ.

Where – використовується для фільтрації даних у процесі їх виборки (працює так як і у SQL).

У середині ми вказуємо умову, по якій ми будемо порівнювати дані при фільтрації ‘(a=>a.Id==id)’.

Далі метод додавання нової моделі у базу даних.

Маршрутизатор ми вказуємо таким чином ‘[Route("Post")]’.

```
public async Task<IHttpActionResult> Post([FromBody] AuthorModels author)
        {
            if(author.UserId!=User.Identity.GetUserId() ||
User.IsInRole("Администратор"))
            {
                return Ok("Вы не можете сделать другого человека автором");
            }
            using (ApplicationDbContext db = new ApplicationDbContext())
            {
                foreach (var item in db.Author.ToList())
                {
                    if (item.UserId == author.UserId)
                    {
                        return BadRequest("Автор с таким UserId уже существует");
                    }
                }
                db.Author.Add(new AuthorModels
                {
                    Position = author.Position,
                    AcademicTitle = author.AcademicTitle,
                    AcademicDegree = author.AcademicDegree,
```

```
            AffiliatedOrganization = author.AffiliatedOrganization,
            UserId = author.UserId
          });
          await db.SaveChangesAsync();
        }
        return Ok();
      }
```

Атрибут **FromBody** вказує, що параметр методу контролера повинен бути витягнутий з даних тіла http-запиту і потім десеріалізован за допомогою форматування вхідних даних (input formatter). За замовчуванням є тільки форматтер JSON. У цього атрибута є одна особливість: він може бути застосований лише до одного параметру методу.

Спочатку при створенні нового рядка ми перевіряємо на те чи є ви адміністратором серверу, або це створюєте саме ви (це зроблено для того щоб ви не змогли зробити іншого користувача автором):

‘if(author.UserId!=User.Identity.GetUserId() || User.IsInRole("Администратор"))’.

Про це нам вказує наступна помилка, якщо ж ми не є даним автором або адміном.

‘return Ok("Вы не можете сделать другого человека автором")'.

Взаємодія з базою даних буде проходити через використання її, як змінною:

‘using (ApplicationDbContext db = new ApplicationDbContext())’

Далі за допомогою використання **foreach** ми перевіряємо на те, чи є в базі вже такий автор.

if (item.UserId == author.UserId)

```
{
    return BadRequest("Автор с таким UserId уже существует");
}
```

А вже тільки після цих дій, у разі проходження всіх умов, ми додаємо нову модель у базу ‘db.Author.Add'.

Після цього ми маємо сказати нашій базі, що вона була змінена та зберегти нові дані за допомогою наступної команди: ‘await db.SaveChangesAsync();’. Вона також буде виконуватися асинхронно (використанням ‘SaveChangesAsync()’ замість ‘SaveChanges()’) для того щоб користувач не чекав виконання зберігання та економив свій час.

Тепер розповімо про метод редагування у авторах.

```
[Route("Put/{id}")]
    [Authorize]
    public async Task<IHttpActionResult> Put(int id, [FromBody] AuthorModels newAuthor)
    {
        using (ApplicationDbContext db = new ApplicationDbContext())
        {
            if (id != newAuthor.Id)
            {
                return BadRequest();
            }
            var UserId = User.Identity.GetUserId();
            if (newAuthor.UserId == UserId || User.IsInRole("Администратор"))
```

```
            {
                db.Entry(newAuthor).State = EntityState.Modified;
                await db.SaveChangesAsync();
            }
            else
            {
                return Ok("У вас нет доступа к изменению автора");
            }
        }
        return Ok();
    }
```

У рядку db.Entry(newAuthor).State = EntityState.Modified; ми змінюємо стару модель на нову за допомогою влаштованої функції ‘Entry’ та нової моделі ‘EntityState.Modified’.

Видалення авторів не передбачене, тому ми не будемо його використовувати, але додаємо ще один контролер, для того щоб отримати повні дані про автора та його елементі в табличці користувача (Users).

Маршрутизатором даного елементу буде ‘[Route("GetInfo")]’.

Сам метод буде таким:

```
public async Task<AuthorModels> GetInfo()
    {
        using (ApplicationDbContext db = new ApplicationDbContext())
        {
```

```
var UserId = User.Identity.GetUserId();

return await db.Author.Where(a => a.UserId == UserId).Include(a => a.Publication).Include(a => a.Groups).FirstOrDefaultAsync();

}

}
```

Єдиний рядок, який ми на пояснювали раніше є ‘var UserId = User.Identity.GetUserId()’.

Тут ви у змінну, під назвою ‘UserId’, записуємо ідентифікатор користувача, який на даний момент користується програмою.

Далі перейдемо до створення контролеру публікацій, саме це є найважливішим контролером у нашій програмі.

У контролері ми вже маємо стандартні методи, які створюються автоматично, про які ми розповідали раніше.

Одним з важливих змін тут є використання нової моделі, яку ми впишемо у папку ViewModels.

```
public class PublicationWithUser
   {
       public PublicationModels Publication { get; set; }
       public ApplicationUser User { get; set; }
   }
```

Тут нам потрібно буде передавати публікацію та користувача, який саме створим цю публікацію.

Для початку створимо новий лист цієї моделі.

```
List<PublicationWithUser> publicationWithUsers = new List<PublicationWithUser>();
```

Далі створимо об’єкт та впишемо в нього нашу таблицю публікацій, з підключенням зв’язних таблиць між ними.

```
var publications = await new ApplicationDbContext().Publications
            .Include(p => p.Authors)
            .Include(p => p.Conference)
            .Include(p => p.KeyWords)
            .Include(p => p.Descriptions)
            .Include(p => p.PublicationType)
            .Include(p => p.Publisher)
            .Include(p => p.Journal)
            .ToListAsync();
```

А далі вже записуємо користувачів через цикл.

```
foreach (var item in publications)
        {
            publicationWithUsers.Add(new PublicationWithUser {
                Publication = item,
                User = await new ApplicationDbContext().Users.Where(u => u.Id == item.UserId).FirstOrDefaultAsync()
            });
        }
```

FirstOrDefaultAsync – отримує перший елемент з усього списку, а у разі якщо елементів у списку немає, то повертає NULL (розширення є асинхронним та можливо тільки з ‘await’).

У методі ‘Get/Id’ ми робимо все теж саме, але видираємо при умові ‘Where(p=>p.Id == id)’.

Далі роздивимось метод додавання нової моделі в базу даних (метод POST).

Спочатку витягаємо саме публікацію з вхідної змінної.

```
PublicationModels publication = publicationWithAuthors.publication;
```

Потім підключаємо базу даних для подальшої взаємодії з нею у нашому коді.

```
using (ApplicationDbContext db = new ApplicationDbContext())
```

Створюємо список авторів та додаємо в цей список нові дані за допомогою LINQ.

```
var authors = new List<AuthorModels>();
            foreach (var item in publicationWithAuthors.addAuthors)
            {
                authors.Add(db.Author.Where(a => a.Id == item).FirstOrDefault());
            }
```

Також створюємо нові об’єкти конференції, журналу та видавництва.

```
            var conference = db.Conference.Where(c => c.Id == publication.ConferenceId).FirstOrDefault();
            var journal = db.Journal.Where(j => j.Id == publication.JournalId).FirstOrDefault();
            var publisher = db.Publisher.Where(p => p.Id == publication.PublisherId).FirstOrDefault();
```

Далі отримуємо Id User та додаємо новий об’єкт до бази даних.

```
var UserId = User.Identity.GetUserId();
            db.Publications.Add(new PublicationModels {
               Annotation = publication.Annotation,
               PublicationType = publication.PublicationType,
               Authors = authors,
               KeyWords = publication.KeyWords,
               Language = publication.Language,
               SqopusLink = publication.SqopusLink,
               WebOfScienceLink = publication.WebOfScienceLink,
               ConferenceName = publication.ConferenceName,
               Date = DateTime.Now,
               Descriptions = publication.Descriptions,
               Text = publication.Text,
               Name = publication.Name,
               NumberOfPages = publication.NumberOfPages,
               PublicationTypeId = publication.PublicationTypeId,
               UserId = UserId,
               Journal = journal,
               Conference = conference,
               Publisher = publisher,
               PublisherId = publication.PublisherId,
               ConferenceId = publication.ConferenceId,
```

```
        JournalId = publication.JournalId
    });
```

У властивості часу ми змінюємо дату та час на дату та час у даний момент

Date = DateTime.Now.

Також змінюємо інші властивості на змінні, які ми розраховували раніше.

```
Authors = authors,
UserId = UserId,
Journal = journal,
Conference = conference,
Publisher = publisher
```

У методі Put логіка така ж сама як у авторів, але головна зміна буде в нас у рядку db.Entry(newPublication).State = EntityState.Modified;

А також потрібно перевіряти, чи є даний користувач тих хто створив дану публікацію. Це зроблено для того щоб неможливо було змінити публікацію іншого користувача.

І на останок роздивимось метод видалення публікації.

```
public async Task<IHttpActionResult> Delete(int id)
    {
        using (ApplicationDbContext db = new ApplicationDbContext())
        {
            var UserId = User.Identity.GetUserId();
            var publication = db.Publications.Where(p => p.Id == id).FirstOrDefault();
            if (publication.UserId == UserId || User.IsInRole("Администратор"))
```

```
            {
                db.Publications.Remove(publication);
                await db.SaveChangesAsync();
            }
            else
            {
                return Ok("Вы не являетесь создателем данной записи");
            }
        }
        return Ok("Публикация удалена");
    }
```

Для початку нам потрібно також виявити, чи є дана публікація створена саме цим користувачем, а не іншою людиною. Це зроблено для того, щоб не можливо було видаляти публікації інших користувачів.

На цьому логіка нашої програми завершена. Можливо ще дуже багато робити розширень до неї, оскільки вона написана з дотриманням усіх правил ООП.

ВИСНОВОК

Застосування новітніх технологій у розробці даного проекту дає великі можливості починаючи з проектування та початку роботи. За допомогою них було розроблену нову програму, як була написана за всіма основними та новими принципами програмування.

Виконана робота показує перспективність використання алгоритмів програмування, які були задіяні в роботі. Була розроблена серверна частина онлайн застосунку для ведення бази даних публікацій.

Спроектована модель бази даних дозволяє й надалі розширювати функціонал нашої програми, оскільки при проектуванні були витримані усі правила проектування баз даних.

Дана робота допоможе полегшити роботу з публікаціями, якщо б це робилося вручну, чи за допомогою Excel. Також вона дає можливість використання її у вигляді CRM системи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Марк Дж. Прайс., C# 7 и .NET Core. Кросс-платформенная разработка для профессионаловя. – 2009. – С. 46 – 51.
2. Троелсен и Джепикс., Язык программирования C# 7 и платформы .NET и .NET Core. – 2018. – С. 33 – 41.
3. Скит. C# для профессионалов. Тонкости программирования. – 2016. – С. 93 – 97.
4. Джозеф Албахари. C# 7.0. Справочник. Полное описание языка. – 2016. – С. 125 – 131.

Додаток А

ВИХІДНИЙ КОД ПРОГРАМИ

```
using Diplom.Models;
using Diplom.ViewModels;
using Microsoft.AspNet.Identity;
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using System.Web.Http;
using System.Web.Http.Cors;
//TODO: сделать еще контроллеры к !группе авторов!, !типу публикаций!,
//TODO: (!конференции!, !издательства!, журналы) в последних трех доступ только администратору дать
namespace Diplom.Controllers
{
    [Authorize]
    [RoutePrefix("api/Publication")]
```

```
//[EnableCors(origins: "*", headers: "*", methods: "*")]
public class PublicationController : ApiController
{
    // GET api/<controller>
    [Route("Get")]
    [HttpGet]
    public async Task<List<PublicationWithUser>> Get()
    {
        List<PublicationWithUser> publicationWithUsers = new
List<PublicationWithUser>();

        var publications = await new ApplicationDbContext().Publications
            .Include(p => p.Authors)
            .Include(p => p.Conference)
            .Include(p => p.KeyWords)
            .Include(p => p.Descriptions)
            .Include(p => p.PublicationType)
            .Include(p => p.Publisher)
            .Include(p => p.Journal)
            .ToListAsync();
        foreach (var item in publications)
        {
            publicationWithUsers.Add(new PublicationWithUser {
```

```
                Publication = item,
                User = await new ApplicationDbContext().Users.Where(u => u.Id ==
item.UserId).FirstOrDefaultAsync()
            });
        }

        return publicationWithUsers;
    }

    // GET api/<controller>/5
    [Route("Get/{id}")]
    [HttpGet]
    public async Task<PublicationWithUser> Get(int id)
    {
        var publication = await new ApplicationDbContext().Publications
            .Include(p => p.Authors)
            .Include(p => p.Conference)
            .Include(p => p.KeyWords)
            .Include(p => p.Descriptions)
            .Include(p => p.PublicationType)
            .Include(p => p.Publisher)
            .Include(p => p.Journal)
            .Where(p=>p.Id == id)
```

```
                .FirstOrDefaultAsync();
            return new PublicationWithUser
            {
                Publication = publication,
                User = await new ApplicationDbContext().Users.Where(u => u.Id ==
publication.UserId).FirstOrDefaultAsync()
            };


        }


        // POST api/<controller>
        [Route("Post")]
        [HttpPost]
        public async Task<IHttpActionResult> Post([FromBody]
PublicationWithAuthorsViewModel publicationWithAuthors)
        {
            PublicationModels publication = publicationWithAuthors.publication;


            using (ApplicationDbContext db = new ApplicationDbContext())
            {
                var authors = new List<AuthorModels>();
                foreach (var item in publicationWithAuthors.addAuthors)
                {
```

```
            authors.Add(db.Author.Where(a => a.Id == item).FirstOrDefault());
        }


        var conference = db.Conference.Where(c => c.Id ==
publication.ConferenceId).FirstOrDefault();
        var journal = db.Journal.Where(j => j.Id ==
publication.JournalId).FirstOrDefault();
        var publisher = db.Publisher.Where(p => p.Id ==
publication.PublisherId).FirstOrDefault();


        var UserId = User.Identity.GetUserId();
        db.Publications.Add(new PublicationModels {
            Annotation = publication.Annotation,
            PublicationType = publication.PublicationType,
            Authors = authors,
            KeyWords = publication.KeyWords,
            Language = publication.Language,
            SqopusLink = publication.SqopusLink,
            WebOfScienceLink = publication.WebOfScienceLink,
            ConferenceName = publication.ConferenceName,
            Date = DateTime.Now,
            Descriptions = publication.Descriptions,
            Text = publication.Text,
```

```
                Name = publication.Name,
                NumberOfPages = publication.NumberOfPages,
                PublicationTypeId = publication.PublicationTypeId,
                UserId = UserId,
                Journal = journal,
                Conference = conference,
                Publisher = publisher,
                PublisherId = publication.PublisherId,
                ConferenceId = publication.ConferenceId,
                JournalId = publication.JournalId
            });
            await db.SaveChangesAsync();
        }
        return Ok("Публикация добавлена");
    }

    // PUT api/<controller>/5
    [Route("Put/{id}")]
    public async Task<IHttpActionResult> Put(int id, [FromBody]
PublicationModels newPublication)
    {
        using (ApplicationDbContext db = new ApplicationDbContext())
        {
```

```
        var UserId = User.Identity.GetUserId();
        var publication = db.Publications.Where(p => p.Id == id).FirstOrDefault();
        if (publication.UserId == UserId || User.IsInRole("Администратор"))
        {
            if (id != newPublication.Id)
            {
                return BadRequest();
            }
            db.Entry(newPublication).State = EntityState.Modified;
            await db.SaveChangesAsync();
        }
        else
        {
            return Ok("Вы не являетесь создателем данной записи");
        }
    }
    return Ok("Публикация обновлена");
}

// DELETE api/<controller>/5
[Route("Delete/{id}")]
public async Task<IHttpActionResult> Delete(int id)
```

```
        {
            using (ApplicationDbContext db = new ApplicationDbContext())
            {
                var UserId = User.Identity.GetUserId();
                var publication = db.Publications.Where(p => p.Id == id).FirstOrDefault();
                if (publication.UserId == UserId || User.IsInRole("Администратор"))
                {
                    db.Publications.Remove(publication);
                    await db.SaveChangesAsync();
                }
                else
                {
                    return Ok("Вы не являетесь создателем данной записи");
                }
            }
            return Ok("Публикация удалена");
        }
    }
}
using Diplom.Models;
using Microsoft.AspNet.Identity;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Threading.Tasks;
using System.Web.Http;
using System.Data.Entity;
using System.Web.Http.Cors;

namespace Diplom.Controllers
{
    [Authorize]
    [RoutePrefix("api/Author")]
    //[EnableCors(origins: "*", headers: "*", methods: "*")]
    public class AuthorController : ApiController
    {
        [Route("Get")]
        public async Task<IEnumerable<AuthorModels>> Get() => await new
ApplicationDbContext().Author
            .Include(a => a.Publication)
            .Include(a=>a.Groups)
            .ToListAsync();

        // GET api/Author/5
        [Route("Get/{id}")]
```

```
        public async Task<AuthorModels> Get(int id) => await new
ApplicationDbContext().Author
            .Include(a => a.Publication)
            .Include(a => a.Groups)
            .Where(a=>a.Id==id)
            .FirstOrDefaultAsync();



        // POST api/<controller>
        [Route("Post")]
        public async Task<IHttpActionResult> Post([FromBody] AuthorModels author)
        {
            if(author.UserId!=User.Identity.GetUserId() ||
User.IsInRole("Администратор"))
            {
                return Ok("Вы не можете сделать другого человека автором");
            }
            using (ApplicationDbContext db = new ApplicationDbContext())
            {
                foreach (var item in db.Author.ToList())
                {
                    if (item.UserId == author.UserId)
                    {
```

```
                return BadRequest("Автор с таким UserId уже существует");
            }
        }
        db.Author.Add(new AuthorModels
        {
            Position = author.Position,
            AcademicTitle = author.AcademicTitle,
            AcademicDegree = author.AcademicDegree,
            AffiliatedOrganization = author.AffiliatedOrganization,
            UserId = author.UserId
        });
        await db.SaveChangesAsync();
    }
    return Ok();
}

// Edit api/<controller>/5
[Route("Put/{id}")]
[Authorize]
public async Task<IHttpActionResult> Put(int id, [FromBody] AuthorModels
newAuthor)
{
    using (ApplicationDbContext db = new ApplicationDbContext())
```

```
    {
        if (id != newAuthor.Id)
        {
            return BadRequest();
        }
        var UserId = User.Identity.GetUserId();
        if (newAuthor.UserId == UserId || User.IsInRole("Администратор"))
        {
            db.Entry(newAuthor).State = EntityState.Modified;
            await db.SaveChangesAsync();
        }
        else
        {
            return Ok("У вас нет доступа к изменению автора");
        }
    }
    return Ok();
}
[Route("GetInfo")]
public async Task<AuthorModels> GetInfo()
{
    using (ApplicationDbContext db = new ApplicationDbContext())
```

```
            {
                var UserId = User.Identity.GetUserId();
                return await db.Author.Where(a => a.UserId == UserId).Include(a =>
a.Publication).Include(a => a.Groups).FirstOrDefaultAsync();
            }
        }

        //// DELETE api/<controller>/5
        //public void Delete(int id)
        //{
        //}
    }
}
```

Додаток Л

Taras Shevchenko National University of Kyiv

Design and development of the server part of the web application for maintaining a database of publications

Software Architecture Document (SAD)

Content Owner: Oleksandr Mustafaiev

DOCUMENT NUMBER: 1.0

RELEASE: 1.0

RELEASE DATE: 01.06.2021

TABLE OF CONTENTS

1. Introduction

    1.1. Purpose

    1.2. Scope

    1.3. Definitions

    1.4. References

2. Architecture Representation

3. Architectural Goals and Constraints

4. Sequence diagram

5. Deployment View

6. Size and Performance

1. Introduction

1.1. Purpose

This document provides a comprehensive architectural overview of the system, using a number of different architectural views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions which have been made on the system.

1.2. Scope

This Software Architecture Document provides an architectural overview of the WEB API for backend. WEB API has own multifunctional tasks list.

1.3. Definitions

WEB API is the server side of the site, which serves as the main logic between the interaction of the client side and the database.

1.4. References

Applicable references are:

- WEB API
- MS SQL
- Entity Framework
- MVC

2. **Architecture Representation**

The model-view-controller pattern taken as a basis is taken. It allows you to divide all the logic of the program into three parts and simplify the work of the program and its structure. Also, for modeling the database, we took the Entity Framework, which allows you to transfer model entities to tables and dependencies in the database.

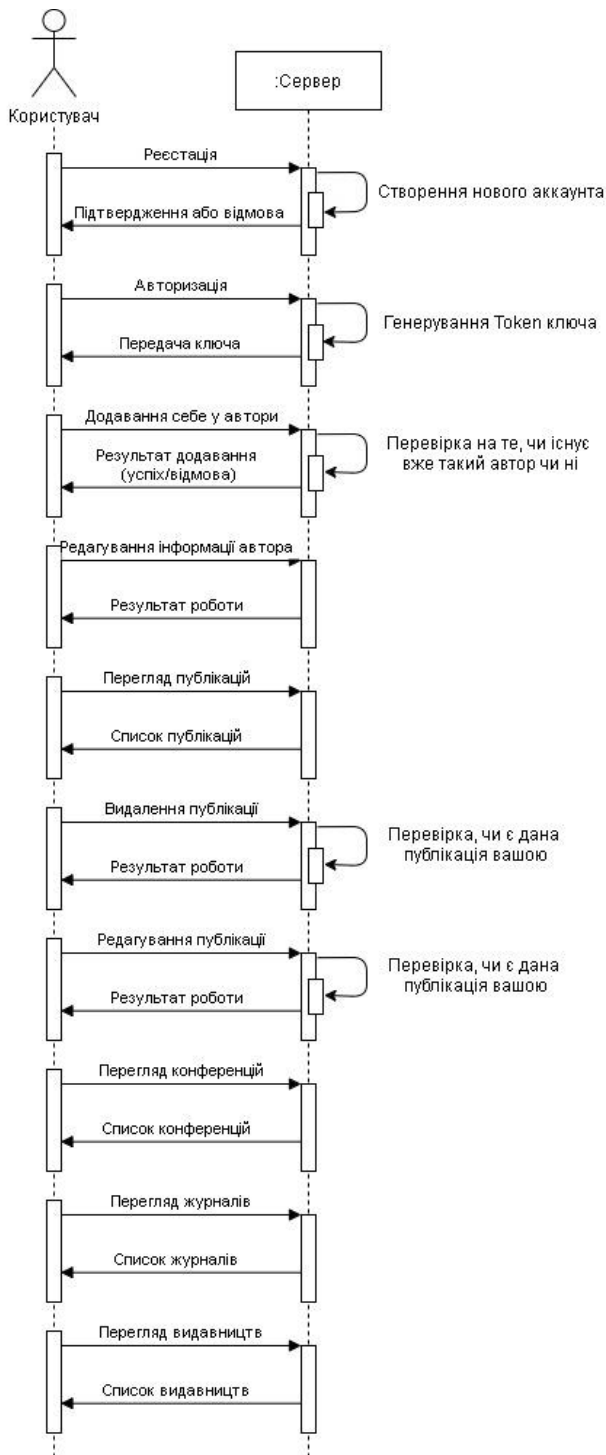
3. **Architectural Goals and Constraints**

- The database has storage for only 2 gigabytes of data.

- The server is located in the southern part of the United States, so there may be delays of a couple of milliseconds.

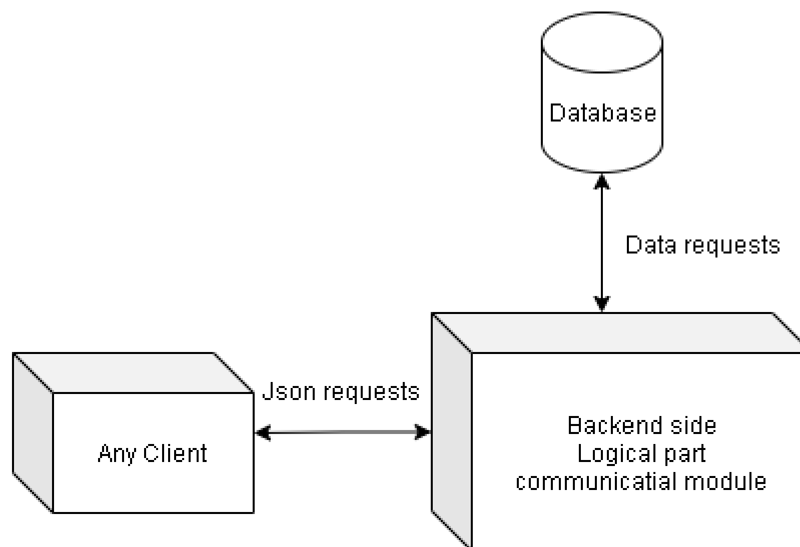
4. **Sequence diagram**

This diagram, which for a certain set of objects on a single time axis shows the life cycle of an object (creation-activity-destruction of a certain entity) and the interaction of actors (actors) of the information system within the framework of the precedent.



5. **Deployment View**

A description of the deployment view of the architecture Describes the various physical nodes for the most typical platform configurations. Also describes the allocation of tasks to the physical nodes. This section is organized by physical network configuration; each such configuration is illustrated by a deployment diagram, followed by a mapping of processes to each processor



6. **Size and Performance**

The chosen software architecture supports multithreading and is well optimized for many users. The program is also located on a remote server, therefore all restrictions rest only on the limitations of Microsoft Azure.

Requirements:

- The system must support up to 10,000 concurrent users.
- The system provides access to the information of the dataset in a few seconds.
- The system must be able to fulfill any design requirements.
- The server application requires 1000 MB of disk space and 2 GB of RAM if you want to use it locally.