ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Пояснювальна записка

до бакалаврської роботи
на ступінь вищої освіти бакалавр

на тему: «**РОЗРОБКА BACK-END ЧАСТИНИ ДЛЯ LEARNING-ДОДАТКУ «CAMPUS» З ВИКОРИСТАННЯМ ФРЕЙМВОРКУ ASP.NET CORE 3.1 ТА ШАБЛОНУ WEB API НА МОВІ C#»**

Виконав: студент 4 курсу, групи ПД–42
спеціальності
121 Інженерія програмного забезпечення
(шифр і назва спеціальності)
Даценко.М.А.
(прізвище та ініціали)
Керівник Коба.А.Б.
(прізвище та ініціали)
Рецензент
(прізвище та ініціали)

Київ – 2021

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

Навчально-науковий інститут інформаційнихї технологій

Кафедра Інженерії програмного забезпечення
Ступінь вищої освіти - «Бакалавр»
Спеціальність -121 Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ

Завідувач кафедри
Інженерії програмного забезпечення

Негоденко.О.В

—_____‖__2021 року

З А В Д А Н Н Я

НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТУ

Даценко Михайло Андрійович

(прізвище, ім'я, по батькові)

1.Тема роботи: «Розробка Back-end частини для learning-додатку «Campus» з використанням фреймворку ASP.NET Core 3.1 та шаблону Web API на мові C#»

Керівник роботи Коба.А.Б
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом вищого навчального закладу від «12» березня 2021 року № 65.
2. Строк подання студентом роботи 1.06.2021

3. Вхідні дані до роботи:
3.1.Програми для розробки: Visual Studio 2019,WebStorm,DataGrip,Postman,Jira
3.2.Вхідні поняття: Continuous integration/deployment, Version control system
3.3.Інформаційні ресурси: Angular docs, Microsoft docs, xUnit docs, Stackoverflow.com

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити).
4.1 Збір та аналіз вимог до майбутньої системи дистанційного навчання.
4.2 Налагодження хостингу для неперервної інтеграції змін.
4.3 Проектування та реалізація доменної моделі за допомогою ORM фреймворку.
4.4 Побудова веб сервісу з механізмами аутентифікації та логування помилок.

5. Перелік графічного матеріалу

1. Мета, об’єкт та предмет дослідження
2. Огляд аналогів: Classroom
3. Огляд аналогів:Canvas
4. Огляд аналогів:Moddle
5. Розбір технічного завдання
6. Програмні засоби реалізації: локальна розробка
7. Програмні засоби реалізації: хостинг
8. Схема бази даних в Azure db
9. Загальна архітектура Web API
10. Клас контекст
11. Метод редагування події
12. Читання подій з віртуальної кімнати
13. Висновки

6. Дата видачі завдання 19.04.2021

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури	19.04-20.04	Виконано
2	Інтерв’ю з користувачами систем дист. навчання	21.04-23.04	Виконано
3	Налагодження процесу неперервної інтеграції в GitHub	23.04-25.04	Виконано
4	Адміністрування Azure App Service для хостингу	26.04-28.04	Виконано
5	Розробка сервісів веб застосунку	1.05-10.05	Виконано
6	Висновки	10.05	Виконано
7	Розробка демонстраційних матеріалів	10.05	Виконано
8	Попередній захист роботи	17.05	Виконано
9	Здача роботи	1.06	Виконано

Студент_____ _____
(підпис) (прізвище та ініціали)

Керівник роботи_____
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Текстова частина бакалаврської роботи: 53 с., 2 табл., 11 рис., 3 дод., 16 джерела.

Ключові слова: .NET, ASP.NET CORE, AZURE, C#, CI/CD, ENTITY FRAMEWORK CORE, GITHUB, MSSQL SERVER, PYTHON, REST, XUNIT

Об'єкт дослідження: дистанційне навчання в рамках централізованої системи організації учбового процесу учнів шкіл та університетів

Предмет дослідження: серверна частина веб застосунку для обміну та аналізу інформації щодо графіку та завдань учнів

Мета роботи: створити веб сервіс та інфраструктуру для зв'язку з БД та іншими сервісами на основі шаблону Web API фреймворку ASP.NET Core

Методи дослідження: user story, мозковий штурм, проектування з використання шаблонів GOFF, побудова діаграми відношення сутностей бази даних , модульне та інтеграційне тестування

Визначено набір вимог до системи, які допоможуть спростити процес навчання студентів вишів.

Здійснено розробку високопродуктивного веб сервісу, який надає доступ до сховища з набором дисциплін, та їх завдань. А також створено інтелектуальну перевірку фото користувача, на не справжність, за допомогою нейронної мережі.

На основі результатів виконаних робіт розроблено стратегію розвитку додатка, та розширення його функціональності. Це дозволить автоматизувати більшу частину учбового процесу учнів інженерних спеціальностей

Упровадження розробленого веб сервісу дозволяє скоротити кількість сторонніх додатків, для обміну інформацією стосовно навчального процесу. Що дасть змогу зекономити час як викладачів так і учнів.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ....9
ВСТУП....10
1. АНАЛІЗ ВИМОГ ДО ЗАСТОСУНКУ....12
1.1 Бізнес вимоги до сервісу....12
1.2 Вимоги до хостингу та інфраструктури Web API....14
1.3 Опис існуючих систем управління навчанням....15
1.3.1 Google Classroom....15
1.3.2 Canvas Learning Management System....16
1.3.3 Оновленна система дистанційного навчання Moodle(СДН ДУТ)....18
2. БЕЗПЕРЕРВНА ІНТЕГРАЦІЯ ТА БЕЗПЕРЕРВНЕ РОЗГОРТАННЯ ВЕБ ДОДАТКА....20
2.1 Створення потоків робіт в GitHub Actions....20
2.1.1 Визначення робіт для автоматизації та їх структура....20
2.1.2 Автоматизація побудови .NET Core додатка....22
2.1.3 Автоматична збірка Angular проекту....24
2.1.4 Побудова та розгортання ASP.NET Core додатка у Azure App Service....25
2.2 Налагодження ресурсу Azure App Service для безперервного розгортання з GitHub Actions....26
3. ВПРОВАДЖЕННЯ ENTITY FRAMEWORK CORE ORM ДЛЯ ДОСТУПУ ДО ДАНИХ....28
3.1 Підхід Code First та клас контексту....28
3.2 Конфігурація моделей за допомогою Fluent API....30
3.3 Попередня ініціалізація даними....33
3.4 Міграції та оновлення бази даних....34
3.5 Висновок та порівняння з Dapper micro ORM....37
4 РОЗРОБКА ВЕБ СЕРВІСУ ЗА ШАБЛОНОМ WEB API....40
4.1 Архітектура веб сервісу....40

4.1.1 Використання REST принципів у побудові веб сервісів 40
4.1.2 Аналіз суміжних архітектур 41
4.1.3 Реалізація архітектури 42
4.2 Розробка контролерів як інтерфейсу для користувачів WEB API 49
4.2.1 Розробка контролера для роботи з подіями 49
4.2.2 Діаграма станів та інтерфейс взаємодії з Web API 52
4.2.3 Фільтрація запитів та логування 54
4.2.4 Обробка винятків 58
ВИСНОВКИ 61
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 63
ДОДАТОК А 65
ДОДАТОК Б 66
ДОДАТОК В 67

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API (англ. Application programming interfaces) – створюється як верхній шар для обміну інформацією з іншими програмами.

REST (англ. REpresentational State Transfer) – це стиль архітектури для стандартизації інтерфейсів взаємодії між програмами в мережі інтернет.

CI (англ. Continuous integration) – практика частого об’єднування нових змін з централізованим сховищем коду, де запускаються автоматизовані тести та збірка проекту.

CD (англ. Continuous delivery) – це продовження CI, оскільки CD автоматично розміщує всі зміни коду в середовищі тестування або на виробничому сервері.

БД – скорочення від: “база даних”.

GitHub – платформа для збереження коду на віддалених серверах.

ORM (англ. Object–relational mapping) – це техніка для відображення даних об’єктів мов програмування, на таблиці реляційний баз даних.

EF Core – скорочення від Entity Framework Core, що є крос платформним ORM фреймворком від корпорації Microsoft.

User story – представлення потреби користувача у функціональності. Може використовуватися людьми з бізнесу та розробниками.

LMS (англ. Learning management system) – платформи для дистанційного навчання.

CLI (англ. Command-line interface) – інтерфейс для керування програмою за допомогою текстових команд та параметрів.

CRUD (англ. Create, read, update, delete) – базові операції на сховищем даних.

UI (англ. User interface) – інтерфейс програми видимий користувачу.

Nuget – пакетний менеджер який допомагає ділитися кодом зі сторонніми розробниками.

LINQ (англ. Language-Integrated Query) – компонент C# для запитів даних.

ВСТУП

На сьогоднішній день є дуже актуальним поняття платформи дистанційного навчання, яка буде автоматизувати комунікаційні та освітні процеси. Наразі учні залучають три або більше застосунків для обміну новинами та учбовими матеріалами, які рознесені по різних місцях. *Проблема* в тому що це займає час, а саме пошук потрібних матеріалів або завдань в різних додатках. Також іноді передане сповіщення про перенесене заняття втрачається в купі інших повідомлень. Ціль цієї роботи, вивчити потреби та створити високопродуктивну серверну частину освітнього додатка “Campus”.

Значущість цієї проблеми в довгостроковому перенесенні навчання на дистанційну основу у зв’язку з пандемією. Наявність єдиної універсальної платформи зараз, заощадить багато часу, та зробить процес навчання більш ефективним як для викладача так и для учнів.

Об‘єкт дослідження: дистанційне навчання в рамках централізованої системи організації учбового процесу учнів шкіл та університетів

Предмет дослідження: серверна частина веб застосунку для обміну та аналізу інформації щодо графіку та завдань учнів

Мета роботи: створити веб сервіс та інфраструктуру для зв’язку з БД та іншими сервісами на основі шаблону Web API фреймворку ASP.NET Core

*Завдання розробки:*в процесі розробки вирішувалися наступні завдання:

1. Циклічній аналіз недоліків існуючих додатків та вимог зі сторони майбутніх користувачів за методом user story.
2. Налагодження хостингу в хмарному середовищі та його зв’язку з БД.
3. Впровадження конвеєру CI/CD безпосередньо з платформи GitHub.
4. Розробка моделі БД та її реалізація через ORM фреймворк EF Core.
5. Розробка API головного сервісу та каналів зв’язку з допоміжними сервісами.

Методи дослідження: user story, мозковий штурм, проектування з використання шаблонів GOFF, побудова діаграми відношення сутностей БД, модульне та інтеграційне тестування

Наукова новизна цього дослідження полягає в наступному:

1. Залучення мікро-сервісної архітектури та docker контейнерів.
2. Використання штучного інтелекту в навчальних додатках з відкритим вихідним кодом.
3. Хостинг додатка та його даних на хмарній платформі Azure.
4. Використання крос-платформного фреймворку ASP.NET Core з оптимізацією під хмарне середовище.

Практична значущість результатів полягає в можливості впровадження цього веб сервісу в повсякденну роботу учнів, мінімізувавши таким чином кількість додатків для комунікації. Універсальність цього сервісу, дозволяє використовувати його як учням та викладачам шкіл, так і університетів.

1. АНАЛІЗ ВИМОГ ДО ЗАСТОСУНКУ

1.1 Бізнес вимоги до сервісу

Backend частина додатку потребує встановлення чітких вимог до бізнес процесів ще до початку проектування, це дасть розуміння які базові механізми потрібно закласти спочатку та як масштабувати додаток в майбутньому. Провівши інтерв'ю з чотирма активними користувачами, систем управління навчанням(СУМ) ДУТ та КПИ, опишемо задачі які слід вирішити нашому сервісу на користувацькому рівні.

Задля автоматизації сповіщень про планування занять та дедлайнів до задач досліджується ***система push[1] повідомлень*** яка матиме розмежування по ролям користувачів та вибір способу отримання цих повідомлень. Цей функціонал мінімізує необхідність мануально сповіщавати учнів про перенесення занять або появу нових. Система буде розпізнавати коли користувач прочитав повідомлення та скривати його с загального списку. У такому ж ключі розглядаються повідомлення про задачі(лабораторні, практичні, тести тощо), а саме здвиг дедлайну, заміна змісту роботи або додавання нової роботи до загального плану дисципліни. Ці зміни будуть відображатися по мірі їх надходження, та з деталями про джерело цих змін.

Наступна задача це ***запуск програмного коду*** (напр. лабораторної роботи) ***інтерактивно у браузері***, для перевірки та надання коментарів учню. Це допоможе перейти від паперового процесу перевірки, до наочного та значно мінімізувати затримку у внесенні змін учнем. Роботи не будуть губитися, та до них можна отримати доступ з будь якого девайсу у якого є доступ до інтернету, у будь який час. На початку планується додати виконання C# коду, та у майбутньому можна буде додати й інші мови програмування, але це залежить від наявності середовища виконання.

Ще одна задача автоматизації це ***авто присвоювання оцінки***, коли учень прострочив термін здачі роботи, або код його лабораторної роботи запускається з

помилками. Це дасть змогу автоматизувати дії викладача та вчасно сповістити учня. Наступним важливим бізнес правилом буде ***розподіл дисциплін по віртуальним кімнатам***, де викладач зможе:

- публікувати події та завдання;
- управляти списком учнів;
- продивлятися оцінки на потік або групу.

Тут же учень зможе:

- підписатися та відписатися від сповіщень цієї кімнати;
- завантажити собі теоретичні матеріали;
- додати та відслідковувати статус перевірки роботи.

Ще однією вимогою являється ***виявлення фальшивого зображення профілю*** при завантажені, це значить що користувач повинен завантажити фото з лицем людини. Це потрібно щоб точно ідентифікувати юзера, та не допустити завантаження фотографії, яка порушує такі правила:

- на фотографії повинно бути виявлено лице людини;
- не допускається завантаження фото з більше ніж одним лицем, щоб людину можна було ідентифікувати однозначно;
- не приймати псевдовипадкові фото людей які побудовані на основі генеративно-змагальної нейромережі та інших сторонніх сервісів.

Якщо якесь із цих правил порушено, повинно бути виведено відповідне повідомлення на екран, та запропоновано завантажити іншу фотографію.

Та заключна вимога це ***наявність календаря***, який відображатиме весь спектр подій та дедлайнів до завдань. Він повинен підтримувати місячний, тижневий та денний вид. А також бути адаптованим до відображення на мобільних пристроях.

1.2 Вимоги до хостингу та інфраструктури Web API

Застосунок повинен бути готовий до розміщення в хмарі, для цього було обрано оптимальний план для студентів від Microsoft Azure на 12 місяців. Для початку розробки достатньо мати доступ до таких ресурсів:

- azure app service – Повністю автономна платформа з вбудованим обслуговуванням інфраструктури, постійними оновленнями безпеки та масштабуванням. А також вона має інтеграцію з CI / CD сервісами для розгортання без простою, наприклад з GitHub Actions. Для старту буде достатньо одного сайту для API з 10 можливих, які йдуть безкоштовно;
- azure sql database – масштабована, реляційна служба баз даних, створена для хмар. А її безсерверні обчислювання та гіпер масштабоване сховище автоматично масштабують ресурси відштовхуючись від поточних потреб. Запуск проекту локально планується саме з підключенням до цієї бази, замість встановлення настільної версії, наприклад MS SQL Server 2017 Express. Що до об’єму бази, ми маємо можливість варіювати від 100 MB до 250 GB.

Усі ці, та подальші ресурси, повинні бути об’єднані в одну ресурсну групу. Ця група, це просто контейнер, що містить відповідні ресурси для рішення Azure.

Надалі планується розширяти наше Web Api, ще одним сервісом який покриває вимогу в виявленні фальшивого фото користувача. Це вже другий сервіс в нашому рішенні, який повинен бути видимий для запитів з нашого головного Web Api. Так як це нейронна мережа, то вона потребує додаткових ресурсів для зберігання своєї навченої моделі, для цього можна скористуватися:

- azure blob storage – допомагає створювати озера даних для потреб аналітики та забезпечує сховище для створення потужних хмарних додатків. З ним можна оптимізувати витрати за допомогою багаторівневого сховища для довгострокових даних та гнучко масштабуватися для високопродуктивних обчислень.

1.3 Опис існуючих систем управління навчанням

1.3.1 Google Classroom

Google Classroom – це веб сервіс розроблений компанією Google LLC, який використовується в закладах середньої та вищої освіти. Його мета спростити та прискорити процес обміну завданнями між учнем та викладачем. Кожен предмет може бути представлений окремою кімнатою, зі своїми учасниками, класною роботою та списком оцінок. Вид кімнат зі списком наступаючих подій має вигляд як на рис. 1.1

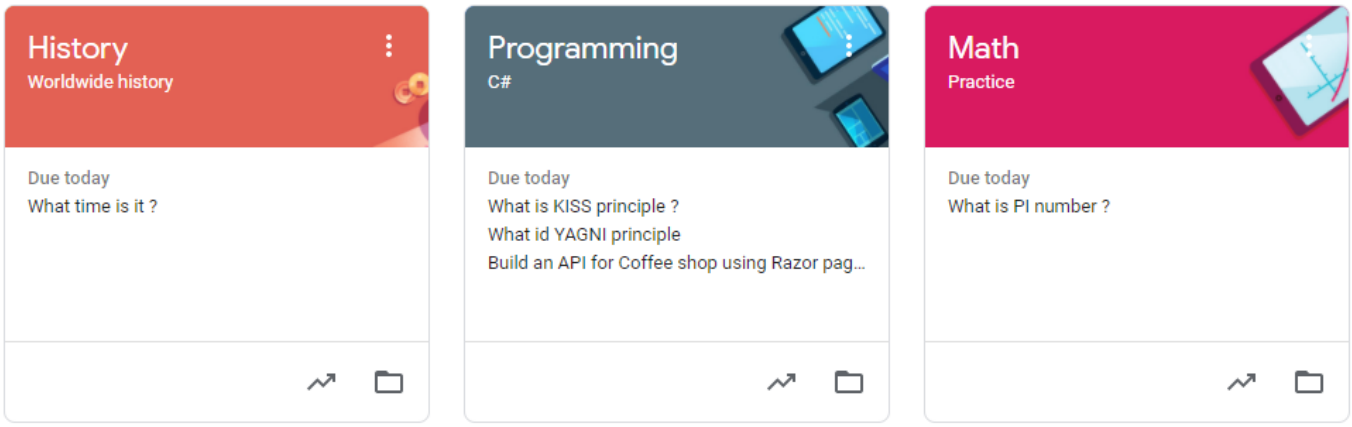


Рисунок 1.1 – Огляд доступних дисциплін

Цей додаток підтримує інтеграцію з такими спорідненими сервісами:

- google calendar – для відслідковування наступаючих подій та дедлайнів до завдань. При створенні кімнати користувач отримує новий пункт у списку своїх календарів, з назвою цієї кімнати, це інтеграція дозволяє розмежовувати події та фокусуватися на одній дисципліні;
- google docs, slides та sheets – цей офісній пакет додатків від Google інтегрується з кімнатами та дозволяє прикріплювати матеріали цих типів прямо з Google Диску, та з подальшою можливістю редагувати їх в онлайн режимі. Також ці додатки мають сумісність з форматами файлів Microsoft Office;
- gmail – для створення листів до яких входить:

- загальний список робіт;
- очікувана дата виконання;
- категорія;
- статус по кожній роботі.

При створенні, лист має автоматично згенеровану тему, яка включає: ім’я студента та назву дисципліни.

Також, Google Classroom має свій API з можливістю доступу з різних фреймворків, в тому числі і .NET. Цей API пропонує інтерфейс для керування кімнатами, перегляду списку викладачів та учнів, та доступ до метаданих у вигляді: ім’я, опису та розташування кімнати. Його можуть використовувати адміністратори домену G Suite for Education[2], для програмного забезпечення курсів від імені викладачів, синхронізації інформації про учнів та отримання базового опису занять, що викладаються у їхньому домені.

1.3.2 Canvas Learning Management System

Canvas - це веб-сервіс управління навчанням. Він використовується навчальними закладами, викладачами та студентами для доступу та управління навчальними матеріалами, а також для спілкування про розвиток навичок та досягнення в навчанні.

Canvas включає в себе різноманітні засоби створення та управління курсами, аналітику та статистику курсів та користувачів, а також інструменти внутрішнього спілкування.

Установи можуть надавати користувачам обліковий запис Canvas, або окремі користувачі можуть спробувати безкоштовну версію, зареєструвавшись у власному обліковому записі, що робить цей сервіс універсальним та доступним для всіх.

Що до основних функцій, то в більшості вони вбудовані, та мають можливість кастомізації. Наприклад інструменти побудови та управління курсами, які можна налаштувати під свої робочі потреби, щоб створити унікальний досвід навчання.

Викладачі можуть створювати та ділитися вмістом курсу, використовуючи такі сервіси як:

- завдання;
- дискусії;
- модулі;
- вікторини та сторінки.

Вони також можуть сприяти спільному навчанню, використовуючи Співпраці, Конференції та Групи. Залежно від налаштувань курсу, студенти можуть отримати доступ до цих областей у Canvas, щоб знайти навчальні матеріали та взаємодіяти з іншими користувачами курсу. Ось так наприклад виглядає сторінка студента, який має доступ до курсу англійської мови (рис. 1.2).

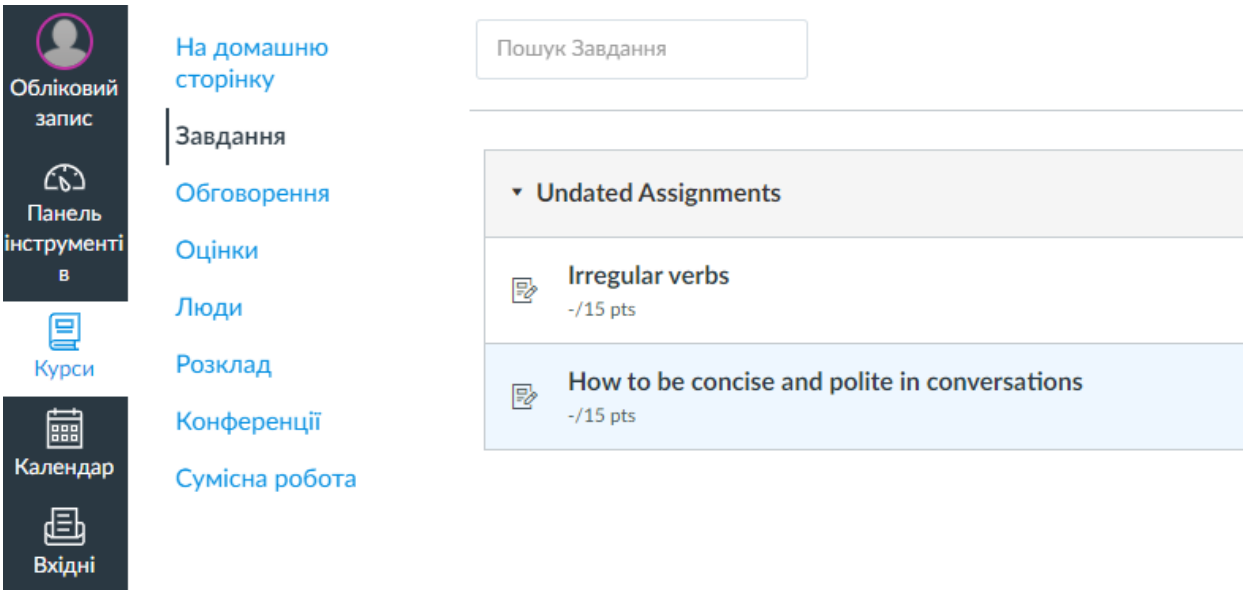


Рисунок 1.2 – Огляд сторінки курсу очима студента

Також Canvas дозволяє установам та викладачам додавати державні та інституційні результати навчання до рубрик для вимірювання та відстеження розвитку навичок та навчальних досягнень учнів. Крім того, творці курсів можуть використовувати інструмент імпорту курсів для масового завантаження попередньо існуючих пакетів курсів LMS та / або матеріалів курсу.

Викладачі можуть надати студентам вичерпні відгуки щодо завдань та тестів, використовуючи SpeedGrader, а також керувати звітуванням про оцінки в Canvas Gradebook. Вони також можуть сприяти взаємодії у режимі реального часу за допомогою чату та обмінюватися новинами та оновленнями курсів зі студентами за допомогою оголошень, а також календаря та навчальної програми.

1.3.3 Оновленна система дистанційного навчання Moodle(СДН ДУТ)

Moodle - це навчальна платформа, призначена для надання викладачам, адміністраторам та учням єдиної надійної, безпечної та інтегрованої системи для створення персоніфікованих навчальних середовищ.

Moodle надається безкоштовно як програмне забезпечення з відкритим кодом під загальною публічною ліцензією GNU. Будь-хто може адаптувати, розширити або модифікувати Moodle як для комерційних, так і для некомерційних проектів без будь-якої плати за ліцензування. Moodle є веб-додатком, тому доступ до нього можна отримати з будь-якої точки світу. Завдяки інтерфейсу, сумісному з мобільними пристроями, та сумісності між браузерами, вміст на платформі Moodle є легко доступним та послідовним у різних веб-браузерах та пристроях.

Проект Moodle добре підтримується активною міжнародною спільнотою, командою відданих штатних розробників та мережею сертифікованих партнерів Moodle. Керуючись відкритою співпрацею та великою підтримкою спільноти, проект продовжує швидко виправляти та вдосконалювати помилки, випускаючи основні нові випуски кожні шість місяців.

В основному Moodle використовують організації таких типів:

- університети;
- початкові школи;
- середні школи;
- державні відомства;
- організації охорони здоров’я.

Основна структура Moodle організована навколо курсів. Це в основному сторінки де викладачі можуть презентувати свої навчальні ресурси та завдання студентам. Вони можуть мати різну структуру, але зазвичай вони включають ряд центральних розділів, де відображаються матеріали, і бічні блоки, що пропонують додаткові функції або інформацію як на рис. 1.3.

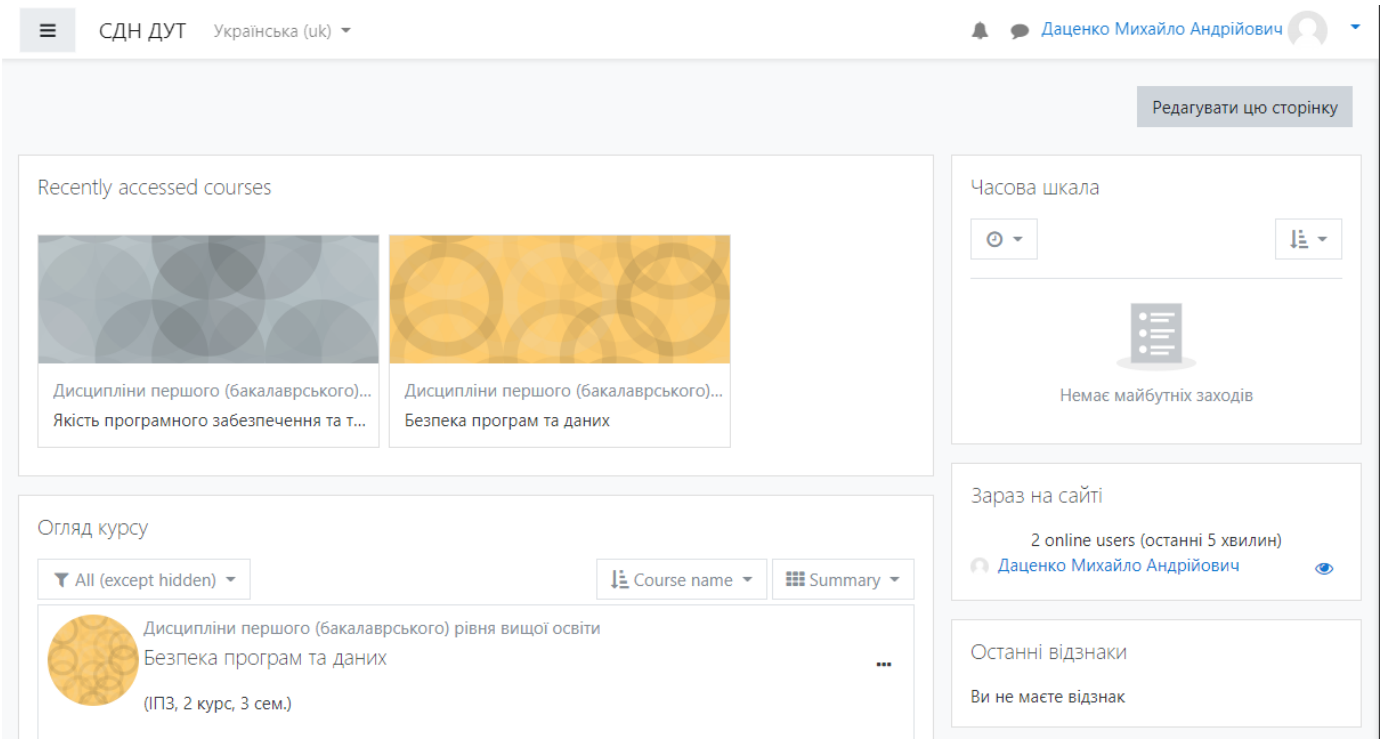


Рисунок 1.3 – Інформаційна панель студента

Курси можуть мати об’єм матеріалів на цілий рік навчання, окрему сесію або будь-які інші варіанти, залежно від викладача та навчального закладу. Їх може використовувати один вчитель або спільно використовувати група викладачів.

Зареєстрований користувач може отримати доступ до областей Moodle, таких як курси або профіль, за допомогою блоку навігації та блоку адміністрування. Те, що користувач бачить у цих блоках, залежить від його ролі та будь-яких привілеїв, наданих їм адміністратором. Адміністратором може бути власник сервера або спеціальний тип акаунта створений цим же власником.

2. БЕЗПЕРЕРВНА ІНТЕГРАЦІЯ ТА БЕЗПЕРЕРВНЕ РОЗГОРТАННЯ ВЕБ ДОДАТКА

2.1 Створення потоків робіт в GitHub Actions

2.1.1 Визначення робіт для автоматизації та їх структура

Для того щоб процес розробки та розгортання зробити ефективним з точки зору часу, потрібна автоматизація рутинних завдань по збірці проекту, запуску тестів та безпосередньо завантаження на головний або production сервер. Платформа GitHub вже має в своєму арсеналі вбудовані інструменти для побудови таких робіт, які доступні на безоплатній основі для використання в репозиторіях з відкритим вихідним кодом.

Кожен так званий action э керованим подіями, це означає, що можна запускати серію команд після того, як відбулася вказана подія. Наприклад, кожного разу, коли хтось створює запит на злиття коду в основну гілку сховища, ви можете автоматично запускати команду, яка виконує сценарій тестування. На даний момент маємо такі три задачі на автоматизацію, беручи до уваги те що ми використовуємо feature branch стратегію для організації гілок проекту:

1. *Збірка ASP.NET Core проекту* після створення запиту на вливання в гілку master. Це один з етапів збірки та тестування нещодавно завершеної одиниці роботи, який має два варіанти завершення:пройшов або ні, і відштовхуючись від цього ми приймаємо рішення. Або вносити нові зміни, або підтверджувати вливання. Також треба підмітити що якщо цей етап виявився провальним треба заборонити можливість вливати код до основної гілки.

2. *Збірка Angular проекту* після проходження попереднього етапу, повинна включати в себе як мінімум:

2.1. Встановлення Angular залежностей.

2.2. Оновлення Angular залежностей.

2.3. Перевірка файлів на слідування встановленим правилам синтаксису та оформленню коду в цілому(не використовувати console.log або неявний тип any).

2.4. Запуск тестів.

3. *Збірка та розгортання ASP.Net Core проекту у веб-порталі Azure*, а саме в одному з його ресурсів Azure App Service. Ця задача включає все з першого етапу плюс команду dotnet publish та безпосередньо команду по розгортанню всього застосунку у Azure веб додатках, після чого він стане доступний глобально в мережі інтернет.

Реалізувавши ці завдання, ми отримаємо повністю автоматизований потік робіт в нашій системі контролю версій. Запуск кожної з трьох робіт буде супроводжуватись логуванням дій з боку сервера, що дасть можливість віднайти конкретну стадію в рамках якої відбувся збій та відразу ж внести зміну в код, без необхідності робити збірку локально.

Історія по кожному запуску робіт зберігається в профілі репозиторія, так же як і текст файлу, який описує команди для виконання в рамках кожної роботи. Також з історії можна побачити час, який зайняли кожна з робіт, це дуже важлива метріка, тому що чим швидше пройдуть роботи по перевірці, тим менше буде затримка в випуску функціоналу.

Що до моніторингу статусів цих робіт, то ми маємо можливість вбудовувати так звані бейджики в файл README.md який тримає так звану markdown нотацію. Наприклад ось так виглядає статус перших двох робіт зараз (рис. 2.1), це означає що обидва додатка збираються без помилок та усі тести для них пройшли успішно.

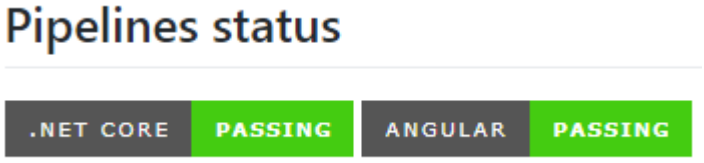


Рисунок 2.1 – статус перевірки .NET Core та Angular додатків

2.1.2 Автоматизація побудови .NET Core додатка

Задача стоїть розробити набір команд, які будуть спрацьовувати при створенні запиту на вливання коду до гілки master. Ці команди треба помістити до файлу dotnetcore.yml, назву ми вибираємо самі.

Отже, перша секція цього файлу майже однакова для всіх робіт, та містить ім’я роботи, та на які події треба спрацьовувати.

```
name: .NET Core

on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]
```

Це означає що робота спрацьовує при відправці коду до вітки master, та при створенні запиту на вливання коду до цієї ж вітки.

Далі йде секція: “Jobs” – роботи, це набір кроків, які виконуються на тому й самому сервері. За замовчуванням робочий процес із кількома завданнями буде виконувати ці завдання паралельно. Також можна налаштувати робочий процес для послідовного запуску завдань. У всіх файлах ми запускаємо команди на сервері з останньою версією операційної системи Linux, як видно з коду нижче:

```
jobs:
  build:

    runs-on: ubuntu-latest
```

Тепер перейдемо до секції: “Steps” яка є набором індивідуальних команд, які можуть виконуватись в роботі. Ось повний список команд для побудови ASP .NET Core проекту.

```
steps:
```

- uses: actions/checkout@v2

- name: Setup .NET Core

uses: actions/setup-dotnet@v1

with:

dotnet-version: 3.1.101

- name: Install dependencies

run: dotnet restore Campus.Master.API.sln

- name: Build

run: dotnet build Campus.Master.API.sln --configuration Release --no-restore

- name: Test

run: dotnet test Campus.Master.API.sln --configuration Release --no-restore.

Ці кроки можуть бути кастомізовані як завгодно, але ми поки працюємо з цим порядком.

Тут нам важливо що в *Setup .NET Core* кроці використовується *setup-dotnet* action з фіксованою версією .NET Core яка повинна співпасти з версією нашого API.Також на цьому кроці встановлюється середовище dotnet core cli, для вказання надалі команд для збірки та тестування нашого .NET Core проекту.

Наступним кроком маємо *Install dependencies*, тут ми запускаємо команду з вже встановленого CLI, для того щоб відновити залежності, а також конкретні інструменти проекту, які вказані у файлі проекту.

Надалі ми вказуємо що треба зібрати проект, при цьому прописуючи шлях до .sln файлу, який містить інформацію про усі проекти та їх розміщення в рамках нашого рішення. Тут вказуємо що ми збираємось у *Release* конфігурації, та унеможливлюємо *відновлення залежностей*, так як це було зроблено раніше.

Та останнім кроком ми запускаємо *тести* з такими ж параметрами як у попередньому кроці. Ця команда віднайде усі проекти по шаблону: “Test Project” та запустить колекцію тестів, якщо всі тести успішні, тестовий менеджер повертає 0, в іншому випадку він повертає 1.

2.1.3 Автоматична збірка Angular проекту

Назвемо наш наступний файл nodejs.yml, в ньому перша частина ідентична з .NET Core автоматизацією, тому одразу ж перейдемо до створення команд для перевірки Angular проекту. Ось повний опис кроків:

```
steps:
- uses: actions/checkout@v2
- name: Setup Angular via Node 12
  uses: actions/setup-node@v1
  with:
    node-version: '12'
- name: Install Packages
  run: npm install --prefix Src/Campus.Master.API/UI
- name: Lint UI
  run: npm run lint:ci --prefix Src/Campus.Master.API/UI
- name: Build UI
  run: npm run build:ci --prefix Src/Campus.Master.API/UI
- name: Test UI
  run: npm run test:ci --prefix Src/Campus.Master.API/UI
```

Так же як і з .NET Core проектом ми вказуємо версію середовища, в якому буде запущений наш проект на час перевірки, тут ми вибрали nodejs версії 12

Надалі ми виконуємо *Install Packages* команду, для Angular проекту, та вказуємо шлях до вихідних файлів користувацького інтерфейсу, де буде знайдений файл package.json. з описом використовуваних пакетів які треба інсталювати.

Наступним кроком ми запускаємо *Lint UI,* для аналізу вихідного коду, позначення помилок програмування, стилістичних помилок та підозрілих конструкцій, які можуть викликати помилки при виконанні. Додатково було додано недопустимість вказання конструкції console.log(‘’), тому що такий спосіб відлагодження є повільним, та іноді не дає повної картини значень змінних.

Далі ми збираємо та тестуємо проект вказуючи один і той самий префікс. Також треба підмітити що, так як це етап неперервної інтеграції, то і тип команд не включає префіксів для виробничого режиму, який ми розглянемо у наступному типі робіт.

2.1.4 Побудова та розгортання ASP.NET Core додатка у Azure App Service

В цій роботі ми вже не викликаємо набір команд при створенні запиту на вливання коду, а тільки спрацьовуємо при прямому вливанні до вітки master. Ця робота буде виконуватись після попередніх, тому ми принаймні можемо бути впевненими що тести зі збіркою проходять успішно. Тепер майже останнім кроком ми виконуємо такий набір команд по відношенню до .NET Core проекту:

```
- name: Set up .NET Core
  uses: actions/setup-dotnet@v1
  with:
    dotnet-version: '3.1.102'

- name: Build with dotnet
  run: dotnet build --configuration Release

- name: dotnet publish
  run: dotnet publish -c Release -o ${{env.DOTNET_ROOT}}/myapp
```

Тут ми бачимо відмінність у префіксі побудови проекту *--configuration Release*, це означає що ми створюємо версію програми, яка готова до розгортання, в нашому випадку у хмарі. Та надалі ми викликаємо команду публікації з таким же префіксом, яка компілює додаток, читає залежності, зазначені у файлі проекту, та публікує отриманий набір файлів у каталозі.

Вихідні файли команди dotnet publish готові до розгортання в хостинговій системі: на сервері, ПК, або Mac. Це єдиний офіційно підтримуваний спосіб підготовки програми до розгортання.

2.2 Налагодження ресурсу Azure App Service для безперервного розгортання з GitHub Actions

Коли GitHub Actions вибраний як провайдер для збірки, ми можемо згенерувати потрібний файл з описом команд для розгортання, вказавши платформу на якій виконується додаток та її версію. Azure відправляє цей файл у вибране сховище GitHub Actions, більш детально це працює таким чином:

- він додає профіль для публікації програми як ключ у GitHub. Саме останній файл із опису потоків робіт в GitHub Actions, використовує цей ключ для автентифікації;
- захоплює інформацію з журналів запуску робіт та відображає її на вкладці: “Журнали” в центрі розгортання програми як видно на рис. 2.2.

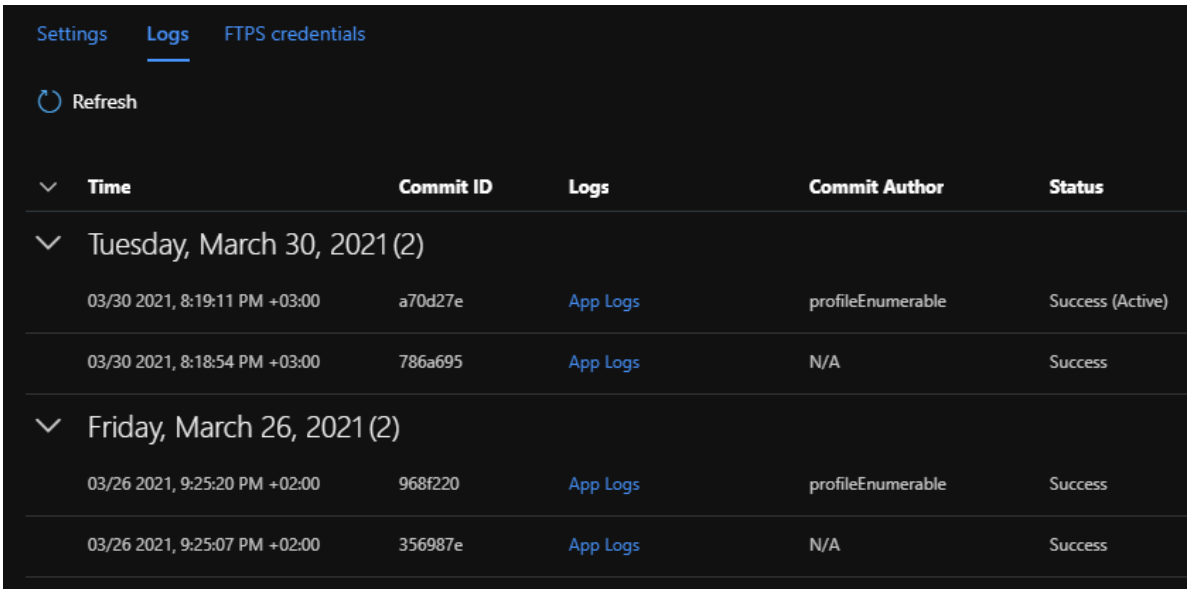


Рисунок 2.2 – Історія розгортань нових версій застосунка

Усі офіційно підтримувані методи розгортання вносять зміни до файлів у папці /home/site/wwwroot додатка. Ці файли використовуються для запуску додатка. Тому розгортання може не вдатися через заблоковані файли. Треба звернутися до офіційної

документації платформи розгортання, щоб урегулювати правила роботи з файлами.

Додаток також може поводитися непередбачувано під час розгортання, оскільки не всі файли оновлюються одночасно. Внесення змін до основної вітки Github, має запустити наш робочий процес і розгорнути веб-додаток в Azure. Після успішного завершення розгортання ми можемо перейти до порталу Azure та перевірити свої зміни у Deployment центрі.

Також гарною практикою є встановлення якось сервісного зображення, на період оновлення виробничого сервера. При переході на сторінку, користувач буде знати, що з додатком все гаразд, просто треба почекати. В деяких веб додатках реалізована спеціальна кнопка оновлення, яка стає активною коли нова версія застосунку готова до використання. З часом до цього можна прийти, але це потребує додаткових знань налагодження хмарної інфраструктури.

Ще для покращення користувацького досвіду після оновлення, додають інтерактивний огляд нових функцій, він представляє собою набір модальних вікон з описом нововведень, які з’являються при досягненні області додатка з новинкою. А задля більш детального документування, створюють так званні примітки до випуску. Це текстове представлення оновлення, яке може бути опубліковано на статичній(тільки текст і посилання по тексту) сторінці з іншою документацією додатка.

Ну і найпродвинутіший спосіб покращити досвід оновлення, це збирання відгуків та побажань користувачів про новий функціонал. Який може бути взятий в розробку, якщо значна(60%) кількість користувачів потребує якогось конкретного нововведення. А щоб тримати в курсі юзерів, про глобальні плани проекту, створюються дорожні карти, де з прив’язкою до версії оновлення, описуються майбутні та пройдешні нововведення. Але цей масштаб підходу може бути прийнятним, коли додаток став користуватися популярністю, та приносити користь хоча б кільком сотням людей.

3. ВПРОВАДЖЕННЯ ENTITY FRAMEWORK CORE ORM ДЛЯ ДОСТУПУ ДО ДАНИХ

3.1 Підхід Code First та клас контексту

Entity Framework Core - це модифікована версія існуючої бібліотеки Entity Framework, яка має розширювану, та крос-платформну підтримку. EF Core підтримує реляційні та нереляційні бази даних. На даний момент Microsoft надає ряд вбудованих провайдерів для роботи з: MS SQL Server, SQLite та PostgreSQL. Також є провайдери від сторонніх постачальників, наприклад для MySQL.

Основним класом, який керує функціональністю EF для даної моделі даних, є клас контекст бази даних: ProjectNameDbContext. Цей клас створюється на основі класу Microsoft.EntityFrameworkCore.DbContext. Похідний клас від DbContext визначає, які сутності включені в модель даних, а які просто виконують роль скелету моделі. EF Core надає підтримку підходів Code First або Database First, як модель програмування.

Code First підхід заснований на проектуванні моделей бази даних, перед появою самого екземпляру бази даних, це означає що ми можемо створювати класи які представляють нашу доменну область та створювати зв’язки між ними. Це робиться з ціллю сгенерувати базу даних на основі коду. Ось як виглядає звичайна модель:

```
public class Event {
    public int Id { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }
    public string ActualLocation { get; set; }
    public int ClassroomId { get; set; }
    public Classroom Classroom { get; set; }
}
```

Тут ми бачимо набір полів які розділені на умовні групи, поле Id зазвичай має кожна модель, воно відображається на базу даних як первинний ключ і може мати назву, або Id або ім’я сутності з постфіксом Id, така конвенція дозволяє Entity Framework Core автоматично виявляти ключі.

Існує декілька видів взаємозв’язків між сутностями. За замовчуванням зв'язок створюється, коли для типу виявлено властивість навігації. Властивість вважається навігаційною, якщо тип, на який вона вказує, не може бути зіставлений як скалярний тип поточним постачальником даних.

Відношення один до одного мають властивість навігації з обох сторін. Вони дотримуються тих самих домовленостей, що і відносини "один-до-багатьох", але для властивості зовнішнього ключа вводиться унікальний індекс. EF вибере одну із сутностей, яка буде залежною, залежно від очевидності назви зовнішнього ключа. Якщо було вибрано неправильну залежну сутность, можна скористатися Fluent API. Залежна модель за замовчуванням вважається необов’язковою, але її можна налаштувати за потреби. Однак EF не перевірятиме, чи було надано залежну сутність чи ні, тому ця конфігурація матиме значення лише тоді, коли відображення бази даних дозволяє її застосувати. Поширеним сценарієм цього випадку є типи посилань, які за замовчуванням використовують розділення таблиць.

Багато-до-багатьох відносини вимагають властивості навігації з обох сторін. Вони будуть виявлені за конвенцією, як і інші типи відносин. Під капотом EF створює тип сутності для представлення об’єднаної таблиці. У моделі може існувати кілька зв’язків багато-до-багатьох, тому типу сутності об'єднання потрібно дати унікальне ім'я

Клас контекст або в нашому випадку CampusContext, це найважливіший клас під час роботи з EF 6 або EF Core. Він представляє сеанс з базою даних, за допомогою якого ви можете виконувати операції CRUD (створення, читання, оновлення, видалення). Цей клас, походить від System.Data.Entity.DbContextDbContext в EF 6 та EF Core. Екземпляр цього контекстного класу представляє шаблони Unit Of Work і

Repository, де ми можемо поєднувати кілька змін в рамках однієї транзакції бази даних.

Колекції сутностей доменної області представленні у вигляді властивостей класу контексту. Кожна властивість має тип DbSet<TEntity> де TEntity це тип даних сутності як видно із коду нижче:

```
public DbSet<Classroom> Classrooms { get; set; }
public DbSet<Event> Events { get; set; }
public DbSet<Label> Labels { get; set; }
public DbSet<Participant> Participants { get; set; }
public DbSet<Privilege> Privileges { get; set; }
```

Через ці колекції ми можемо робити вибірку або оновлення даних, за допомогою спеціального набору методів розширення. Ці методи є частиною LINQ(Language Integrated Query) компоненту, що додає можливості робити запити даних до мов .NET.

3.2 Конфігурація моделей за допомогою Fluent API

Для того щоб налаштувати відображення сутностей вручну, а не приймати конфігурацію EF за замовченням, ми використовуємо набір методів з класу EntityTypeBuilder. Як правило, функціонал Fluent API задіюється при перевизначенні методу OnModelCreating як видно з коду нижче:

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    base.OnModelCreating(modelBuilder);

    modelBuilder.ApplyConfiguration(new ClassroomMap());
    modelBuilder.ApplyConfiguration(new EventLabelMap());
    modelBuilder.ApplyConfiguration(new EventMap());
    modelBuilder.ApplyConfiguration(new LabelMap());
    modelBuilder.ApplyConfiguration(new ParticipantMap());
```

```
        modelBuilder.ApplyConfiguration(new PrivilegeMap());
        modelBuilder.ApplyConfiguration(new RoleMap());
        modelBuilder.ApplyConfiguration(new RolePrivilegeMap());
        modelBuilder.ApplyConfiguration(new UserMap());
    }
```

Тут ми за замовченням визиваємо метод OnModelCreating з базового класу IdentityDbContext. Як правило, цей метод викликається лише один раз, коли створюється перший екземпляр похідного контексту. Потім модель для цього контексту кешована і призначена для всіх подальших екземплярів контексту в домені програми. Це кешування можна вимкнути, встановивши властивість ModelCaching на даному ModelBuilder-і, але це може серйозно погіршити продуктивність в цілому. Більший контроль над кешуванням можна забезпечити за допомогою DbModelBuilder та DbContextFactory.

У коді вище ми вказуємо набір конфігураційних класів, кожен з який має постфікс Map, що є скороченням від mapping(від анг. відображення). В них ми якраз і відображаємо налаштування для полів наших сутностей. В середині кожен з класів реалізує інтерфейс IEntityTypeConfiguration<TEntity> де TEntity це тип моделі для конфігурації, цей інтерфейс має метод Configure, приклад якого можна розглянути нижче:

```
public void Configure(EntityTypeBuilder<Classroom> builder)
{
    builder.Property(p => p.Id)
        .ValueGeneratedOnAdd();

    builder.Property(p => p.Name)
        .IsRequired()
        .HasMaxLength(100)
```

```
        builder.Property(p => p.Institution)
            .HasMaxLength(250);

        builder.Property(p => p.DefaultLocation)
            .IsRequired()
            .HasMaxLength(2048);

        builder.Property(p => p.IsRemote)
            .IsRequired();
    }
```

Майже у кожній конфігурації ми викликаємо метод ValueGeneratedOnAdd для конфігурації ідентифікатора, це задає значення для обраної властивості яке буде генеруватися базою даних щоразу, коли до неї додається нова сутність. Таким чином, властивість повинна бути проігнорована EF при побудові конструкції INSERT. Також часто для полів використовується метод *IsRequired* який налаштовує, чи має ця властивість мати якесь значення, або *null* є дійсним значенням. Властивість можна налаштувати як необов'язкову, лише якщо вона базується на CLR типі, якому можна присвоїти значення *null*.

Метод *HasMaxLength* застосовується до властивості, щоб вказати максимальну кількість символів або байтів для стовпця, на який має відображатись властивість.

Також в рамках конфігурації таблиць з відношенням багато-до-багатьох використовується такий підхід як у відображенні подій на ярлики:

```
        builder.HasKey(el => new {el.EventId, el.LabelId});

        builder.HasOne(el => el.Event)
            .WithMany(e => e.Labels)
            .HasForeignKey(el => el.EventId)
            .OnDelete(DeleteBehavior.Cascade);
```

```
builder.HasOne(el => el.Label)
    .WithMany(l => l.Events)
    .HasForeignKey(el => el.LabelId)
    .OnDelete(DeleteBehavior.Cascade);
```

Вище створюється складний первинний ключ та описується послідовність відображення полів з кожної таблиці, що в результаті дасть третю таблицю яка буде мати два зовнішніх ключа які будуть посилатися на записи у зв’язних таблицях. Доречі версія EF Core 5.0 підтримує відносини багато-до-багатьох без явного відображення таблиці об'єднання, це автоматично створює об’єднуючу таблицю у базі даних. Дані можна читати та оновлювати без явного посилання на таблицю об’єднання, що значно спрощує код. Але таблицю об’єднання все ще можна налаштувати явно, за необхідністю.

3.3 Попередня ініціалізація даними

Ініціалізація дуже подібна до конфігурації моделей, тому що використовується той же інтерфейс IEntityTypeConfiguration та його метод Configure, але замість набору різних методів для вказання конфігурації, використовується лише HasData метод, де ми вказуємо набір сутностей, які будуть завантажені автоматично перед першим використанням бази даних, наприклад ми маємо таку ініціалізацію:

```
public void Configure(EntityTypeBuilder<Privilege> builder)
{
    builder.HasData(
        new Privilege {
            Id = 1,
            Name = "Create Update Delete on events",
            Alias = "cud-on-events"
        },              new Privilege{
```

```
            Id = 2,
            Name = "Read events",
            Alias = "read-events"
        }
    }
```

В цьому випадку ми ініціалізуємо сутність Privilege і якщо зробити запит після застосування міграцій(див. наступний підрозділ), то ми маємо побачити наступну вибірку як на рис. 3.1.

	Id	Name	Alias
1	1	Create Update Delete on events	cud-on-events
2	2	Read events	read-events

Рисунок 3.1 – Вибірка за допомогою select * from Privileges запиту

3.4 Міграції та оновлення бази даних

Використання міграцій є стандартним підходом у створенні та оновленні бази даних за допомогою Entity Framework Core. Процес міграції складається з двох етапів: створення міграції та застосування міграції. Наша схема(таблиці) бази даних повинна бути узгоджена з доменною моделлю(C# класи), і кожна зміна у цій моделі повинна бути перенесена в саму базу даних. За час розробки мені доводилось розширювати моделі новими полями або додавати нові сутності, після кожної такої зміни слідувала нова міграція та оновлення БД на основі цієї міграції.

З ASP.NET Core версії 3.0 інструменти EF Core, які необхідні для міграцій, не встановлюються заздалегідь. Тому нам потрібно встановити бібліотеку Microsoft.EntityFrameworkCore.Tools. за допомогою пакетного менеджера Nuget.

Для створення міграцій ми можемо використовувати консоль Windows з такою командою та параметрами: dotnet ef migrations add InitialCreate --

Тут ми використовуємо donet cli[4] для EntityFramework Core, та вказуємо що хочемо створити нову міграцію з ім’ям InitialCreate, а також додаємо параметр як шлях до нашої бібліотеки з налаштуваннями доменної моделі. Це створить для нас папку з таким контентом як на рис. 3.2.

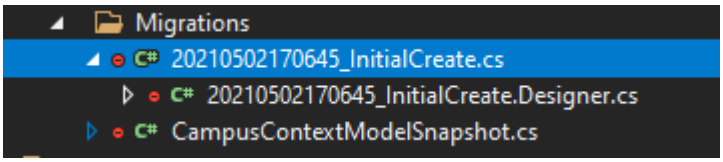


Рисунок 3.2 – Результат виконання команди по створенню міграцій

Коли ми створюємо міграцію, фреймворк порівнює поточний стан моделі з попередньою міграцією якщо вона існує, і генерує файл, що містить клас, який успадковується від Microsoft.EntityFrameworkCore.Migrations.Migration із методом Up та Down. Класу присвоєно те саме ім'я, що і було вказано для міграції. Саме ім'я файлу - це ім'я міграції з префіксом відмітки часу.

Метод Up містить код C#, який застосовує будь-які зміни внесені до моделі, до схеми бази даних з часу останньої міграції. Метод Down скасовує ці зміни, відновлюючи базу даних до стану попередньої міграції. Також створюється або оновлюється файл CampusContexModelSnapshot залежно від того, чи існував він раніше.

Після додавання міграцій їх потрібно застосувати до бази даних. Для цього існують різні стратегії, причому одні з них більше підходять для виробничого середовища, а інші – для цілей локальної розробки. Але перед оновленням бази даних навіть на локальному комп’ютері, ми повинні завжди перевіряти згенеровані міграції та тестувати їх перед застосуванням до виробничої БД. Міграція може видалити стовпець, коли намір був його перейменувати, або можуть виникнути конфлікти які не дадуть міграції успішно завершитися, наприклад несумісність у способі оновлення записів у зв’язних таблицях.

Отже для оновлення локальної БД з останньою міграцією, ми повинні очистити таблиці в існуючій БД, та виконати наступну команду:

dotnet ef database update – project=../Campus.Infrastructure.Data.EntityFrameworkCore

Також існує сценарій де нам потрібно виконати операції міграції/оновлення на віддаленій БД, наприклад у нашій Azure SQL database. Рекомендований спосіб розгортання міграцій у виробничій базі даних - це створення сценаріїв SQL. До переваг цієї стратегії можна віднести наступні[5]:

- сценарії SQL можна перевірити на точність; це важливо, оскільки застосування змін схеми до виробничих баз даних є потенційно небезпечною операцією, яка може спричинити втрату даних клієнтів;
- у деяких випадках сценарії можна налаштувати відповідно до конкретних потреб виробничої бази даних;
- сценарії SQL можна використовувати разом із технологією розгортання і навіть впроваджувати як частину процесу CI.

Тож у цьому випадку, щоб створити сценарій з SQL скриптами для оновлення БД, ми виконаємо таку команду: dotnet ef migrations script --project=../Campus.Infrastructure.Data.EntityFrameworkCore --configuration Release --no-build --idempotent --output migrations.sql. Це дасть нам можливість отримати файл migrations.sql з набором SQL операторів для застосування створених міграцій де іде.

До цієї ж команди був віднесений параметр –idempotent, який перевіряє таблицю з історією версій міграції, щоб знати які міграції вже застосовані , і застосовує лише відсутні. Це корисно, якщо ми точно не знаємо, якою була остання міграція застосована до бази даних, або якщо ми розгортаємо міграції до декількох баз даних, кожна з яких може мати різні версії міграцій.

Але так як ми використовуємо Azure SQL database, описаний вище підхід не доводиться використовувати дуже часто, це більше для експериментальних міграцій або для дослідів над тим як саме наші конфігурації(C# код) відображаються.

3.5 Висновок та порівняння з Dapper micro ORM

EntityFramework Core був обраний тому що нам були потрібні дві речі, це можливість генерувати базу даних на основі доменної моделі, та один загальний синтаксис для запитів даних, мова йде про LINQ. Остання вимога існує тому що ми працюємо в команді де використовується також операційна система Linux, та щоб продовжити розробку на іншій платформі, нам не потрібно переписувати набір запитів на аналогічний, який використовує PostgreSQL. Необхідний провайдер БД буде обраний автоматично на основі конфігураційного файлу, після запуску проекту.

Також протягом змін схеми доменної моделі, було зекономлено багато часу на оновленні БД за допомогою міграцій. Та якщо порівнювати EF Core з Dapper який є простим .NET об'єктом, та дуже вдалим рішенням з точки зору швидкості, ми маємо проаналізувати функціональні можливості обох ORM фреймворків.

Dapper та EF Core є двома основними ORM, що використовуються у світі .NET, незважаючи на те, що Dapper насправді взагалі не є ORM, це більше бібліотека для відображення, призначена для зіставлення результатів запитів із об’єктами C#. Таким чином, Dapper є дуже тонким шаром між нашим додатком та базою даних.

З іншого боку EF Core, є повноцінною ORM з безліччю цікавих функцій, включаючи відстеження змін при запитах. Ці функції роблять цю бібліотеку трохи громіздкою і, можливо, набагато менш ефективною, ніж Dapper.

Dapper дозволяє задіяти наші навички в SQL для конструювання запитів і команд, в тому вигляді, в якому вони повинні бути завжди, на відміну від EF, де процес генерації результуючого SQL коду закритий від користувача. Dapper ближче до рівня «заліза», ніж стандартний EF. Тому що EF Core конвертує написані нами запити у звичайний SQL, роблячи при цьому набір маніпуляцій по оптимізації. І в результаті, один і той же запит буде відрізнятися на обох ORM фреймворках.

У Dapper є вражаючі можливості перетворення, наприклад розбивання списку, який передається блоку WHERE IN. Здебільшого SQL,

що передається нами в Dapper, готовий до роботи, і запити потрапляють в БД набагато швидше. Якщо добре знати SQL, то, безумовно, можна написати настільки продуктивні команди, наскільки це можливо. Для виконання запитів ми повинні створити тип який унаслідується від IDbConnection, наприклад SqlConnection з рядком підключення. Потім Dapper за допомогою свого API може виконувати запити (за умови, що схему результатів запиту можна співвіднести з властивостями цільової моделі) автоматично і заповнювати об'єкти результатами запиту.

Що до переваг та недоліків Entity Framework у площині розробки, то маємо наступне:

Переваги:

1. Entity Framework дозволяє створити доменну модель написавши звичайний C# код, або візуальні інструменти в EF Designer.
2. Можливість писати код запитів з використанням LINQ, де система автоматично створюватиме для нас об'єкти, та відстежуватиме зміни в цих об'єктах, що спрощує процес оновлення даних.
3. Один загальний синтаксис(LINQ), для всіх типів запитів. Що вирішує проблему використання декількох провайдерів баз даних в рамках одного додатка.
4. EF може замінити великий шматок SQL коду, який нам інакше довелося б писати та підтримувати самостійно.

Недоліки:

1. Якщо в базі даних відбудеться будь-яка зміна схеми, EF не буде працювати, і нам доведеться оновлювати схему і у доменній моделі також.
2. Запити EF генеруються механізмом, який ми не можемо контролювати.
3. Складно підтримувати величезну домену модель.

Переваги та недоліки Dapper можуть розглядатися у порівнянні з EF, тому що все таки це різні інструменти які покривають свою область задач. Тому Dapper має:

Переваги:

1. Dapper спрощує параметризацію запитів.
2. Він може легко виконувати різнотипові запити (скалярні, багаторядкові, без результатні).
3. Спрощує перетворення результатів в об'єкти.
4. Дає більше контролю над формуванням запитів.

Недоліки:

1. Dapper не може автоматично створити модель класу.
2. Він не може генерувати запити.
3. Він не може відстежувати об'єкти та їх зміни.
4. Стандартна бібліотека Dapper не надає функції CRUD, але їх можна впровадити через додаткові пакети.

При подальшому розвитку додатка та переходу до мікросервісної архітектури, розглядається використання обох фреймворків для різних сервісів. Це може бути зручно коли ми маємо сервіс з джерелом даних, зі складною внутрішньою структурою, яка не піддається автоматичному відображенню. Також треба не забувати про покриття Dapper запитів модульними тестами, не менше ніж EF запити, тому що ми не можемо бути впевненими що Dapper запит виконається успішно, бо SQL код вказуються у звичайних рядках і навіть збірка проекту не дає ніяких гарантій. Та у заключенні маємо невелику таблицю з тестами продуктивності обох розглянутих ORM фреймворків, для БД де сумарно 100 подій та 15 віртуальних кімнат.

Таблиця 3.1 – Порівняння показників продуктивності Dapper та EF Core

Середній час у ms	GetEventById	GetClassroomEvents	GetAllClassrooms
Dapper	0.16	0.27	1.5
EF Core з AsNoTracking()	1.1	1.7	2.5
EF без AsNoTracking()	1.5	2.0	3.0

4 РОЗРОБКА ВЕБ СЕРВІСУ ЗА ШАБЛОНОМ WEB API

4.1 Архітектура веб сервісу

4.1.1 Використання REST принципів у побудові веб сервісів

З огляду на те що додаток буде використовуватись у багатокористувацький сфері та мати високе навантаження, було обрано саме WEB API шаблон. Тому що він масштабований і надійний, оскільки підтримує всі функції MVC, такі як маршрутизація, контролери та контейнер IOC. Також API планується будувати з використанням REST[3] принципів або обмежень. До них входять:

Уніфікований інтерфейс – тут сама назва обмеження говорить за себе, ми повинні створити інтерфейс для ресурсів всередині системи, який буде відкритий споживачам API. Ресурс у системі повинен мати лише один логічний URI, який повинен забезпечувати спосіб отримання пов'язаних або розрізнених даних. Також завжди краще синонімізувати ресурс із веб-сторінкою.

Клієнт-серверна архітектура – це обмеження по суті означає, що клієнтська та серверна програми повинні мати можливість розвиватися окремо, без будь-якої залежності один від одного. Клієнт повинен знати лише ресурсні URI, і це все. Сьогодні це стандартна практика веб-розробки.

Сервер без збереження стану - тут нам потрібно зробити усі взаємодії клієнт-сервер без збереження стану. Сервер не буде зберігати нічого про останні HTTP-запити зроблені клієнтом. Кожне звернення буде розглядатися як нове. Ні сесії, ні історії. Якщо клієнтський додаток повинен бути із збереженням стану для кінцевого користувача, де користувач входить один раз і виконує інші необхідні операції, тоді кожен запит від клієнта повинен містити всю інформацію, необхідну для обслуговування запиту, включаючи деталі про автентифікацію та авторизацію.

Можливість кешувати данні - кешування приносить покращення продуктивності на стороні клієнта та кращі можливості для масштабованості сервера.

Кешування може бути реалізоване на сервері або на стороні клієнта.*Система яка збирається зі слоїв* – це підхід, де ви розгортаєте API на сервері X і зберігаєте дані на сервері Y та аутентифікуєте запити на сервері Z.Архітектура веб сервісу.

4.1.2 Аналіз суміжних архітектур

Для побудови веб сервісу була обрана цибулина архітектура. Існує кілька традиційних архітектур, таких як трирівнева архітектура та архітектура n-рівня, кожна з яких має свої плюси та мінуси. У всіх цих традиційних архітектур є деякі фундаментальні проблеми, такі як тісній взаємозв'язок між проектами та погана масштабованість. Model-View-Controller - це найчастіше використовувана архітектура веб-додатків на сьогодні. Вона вирішує проблему розподілу відповідальності, оскільки існує поділ між інтерфейсом користувача, бізнес-логікою та логікою доступу до даних. View використовується для проектування інтерфейсу користувача. Модель використовується для передачі даних між View та Controller, над якими бізнес-логіка виконує будь-які операції. Контролер використовується для обробки веб-запитів і відповідно повертає View. З усіх цих фактів зрозуміло що це вирішує проблему відокремлення рівнів. По суті, MVC вирішує проблему відокремлення, але проблема жорсткої взаємодії все ще залишається.

В нашому випадку цибульна архітектура вирішує як проблему розділення, так і тісні взаємозв'язки. Загальна філософія цибульної архітектури полягає в тому, щоб сконцентрувати бізнес-логіку, логіку доступу до даних та модель в середині програми, та розсунути залежності якомога далі, назовні, а це означає, що все зчеплення відбувається до центру. При створенні архітектури веб сервісу треба розуміти, що реальна кількість рівнів тут вельми умовна. Залежно від масштабу завдань рівнів може бути і більше, і менше. Однак важливо розуміти сам принцип, що в центрі у нас моделі домену, а все інше залежить від них. Кожен зовнішній рівень може залежати від внутрішнього, але не навпаки. Також треба розуміти що цей шаблон цибулиної архітектури, не є срібною кулею для кожної проблеми.

Як і у випадку з усіма проблемами проектування, нам потрібно оцінити, чи потрібна нам ця додаткова абстракція, оскільки вона більше підходить для великих додатків, на яких працює багато інженерів. Як інженерам нам потрібно застосовувати критичне мислення, щоб визначити, чи буде цей підхід корисним для поставленого завдання. Крім того, додаткова складність визначення контрактів сервісів вимагає глибокого розуміння шаблону. Якщо проектування буде виконано якісно, то ми отримаємо значну гнучкість програм, що розробляються.

4.1.3 Реалізація архітектури

Цибулина архітектура значною мірою спирається на принцип інверсії залежностей. Користувацький додаток взаємодіє з бізнес-логікою через інтерфейси, концептуально це видно на рис. 4.1.

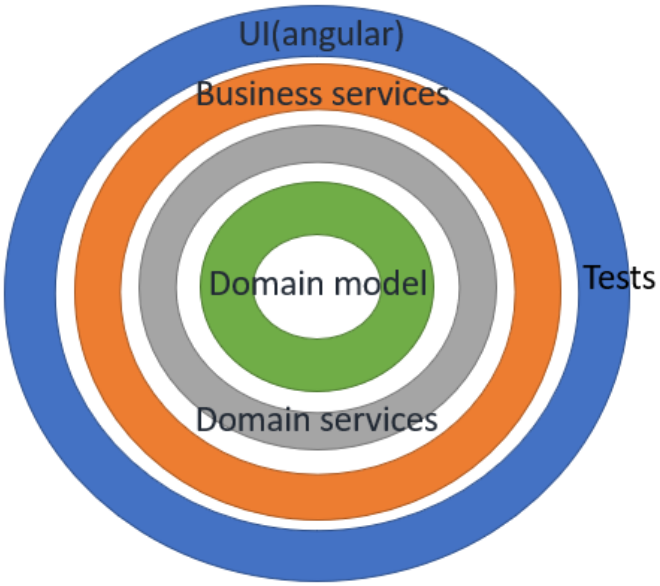


Рисунок 4.1 – Модель архітектури веб сервісу

Центральною частиною є сутності або моделі домену, що представляють об'єкти бізнесу. Ці шари можуть відрізнятися, але шар сутностей домену завжди є частиною центру. Верхні рівні визначають більш поведінкові моменти об’єкта. Ця архітектура

не підходить для невеликих веб-сайтів. Вона підходить для довготривалих(5+ років) проектів, а також для програм зі складною поведінкою. Якщо розібратися шар за шаром більш детально то:

Шар доменної моделі - частина архітектури яка вміщує всі об'єкти домену програми. Якщо додаток розробляється за структурою сутностей ORM[4], тоді цей рівень містить класи POCO у Code First підході. Ці сутності домену не мають прямих залежностей.

Шар доменних сервісів сховища - призначений для створення рівня абстракції між рівнем сутностей домену та рівнем бізнес-логіки програми. Це шаблон доступу до даних, який задає більш простий підхід з точки зору маніпуляції даними.

Рівень бізнес-сервісів - цей шар містить інтерфейси, які використовуються для зв'язку між рівнем користувацького інтерфейсу та рівнем сховища. Він містить бізнес-логіку для сутностей, тому він також називається рівнем бізнес-логіки.

Шар користувацького інтерфейсу - це самий зовнішній шар. Тут може бути розташований веб-додаток, веб-API або проект з модульними тестами. Цей рівень реалізує принцип інверсії залежностей, це є чудовий спосіб зменшити тісний зв'язок між програмними компонентами. Замість строго залежних модулів, таких як драйвери баз даних, ви встановлюєте ці зв’язки через третю сторону. Інверсія залежностей, що має центральне значення для таких програм як ASP.NET Core, дозволяє краще керувати змінами вимог та іншою складністю програмного забезпечення.

Визначивши шари архітектури, перейдемо до структури проекту з точки зору файлів і папок. По перше ми маємо справу з .NET рішенням, де слово рішення або з англ. solution представляє тип файлу який містить інформацію на основі тексту, яку середовище (в нашому випадку Visual Stidio) використовує для пошуку та завантаження зв’язаних проектів та інших сервісних даних.

Коли користувач відкриває рішення, середовище циклічно переглядає цю інформацію у файлі .sln та представляє ієрархію проектів та файлів. Так як більшість наших проектів має розширення .dll то посилання виглядає таким чином:

Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}")=
"Campus.Master.API","Src\Campus.Master.API\Campus.Master.API.csproj",
"{B7A332F4-936D-4572-82D1-088B40B3CC2C}"
Цей текст містить унікальний GUID код проекту та GUID код типу проекту.

У нашому рішенні є шість проектів, з яких чотири- це класичні бібліотечні проекти з розширенням .dll, один - проект з інтеграційними тестами, та один проект по шаблону ASP.NET Core WebAPI, який має в своєму наборі .dll, .exe, .pdb вихідні файли. Visual Studio показує набір проектів відштовхуючись від дерева залежностей, вона відображає кожен проект, як самостійну одиницю, а не просто файл. Ось як це виглядає у огляді структури рішення на рис. 4.2.

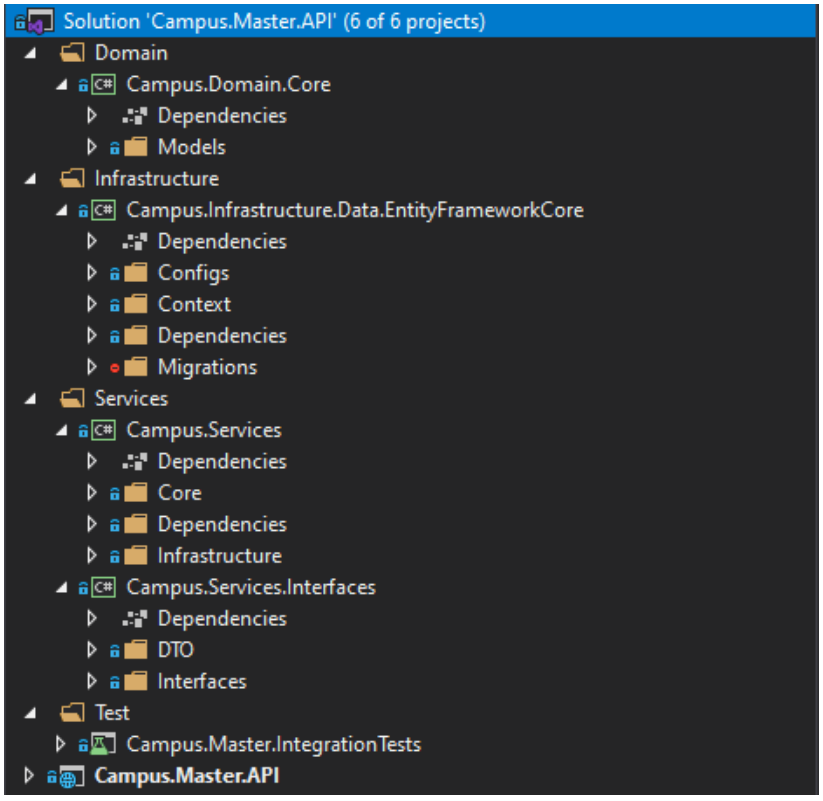


Рисунок 4.2 – Структура рішення проекту

Так як рівень Domain та Infrastructure будуть більш детально розглянуті в контексті розділу про Entity Framework, тож ми розглянемо два взаємо пов’язаних

проекта у папці Services, тому що вони можуть розвиватися незалежно. Шар сервісів у нашому проекті відповідальний за роботу з даними, а саме виконання операцій: зчитування, запису, оновлення та видалення записів у базі даних, через EntityFramework ORM систему. А також виконання інших операцій що потрібні з точки зору бізнес процесів, наприклад форматування даних перед видачею, в залежності від ролі користувача.

Для початку визначимо першу ключову сутність нашої доменної моделі, це сам користувач. Йому як і іншим користувачам веб сервісів, потрібен кабінет, де буде зберігатися притаманна користувачу власна інформація та інші налаштування профілю. Для цього ми маємо забезпечити базові операції для роботи з даними профілю, їх ми опишемо в інтерфейсі або контракті, який в свої чергу буде реалізований конкретним класом. Ось приклад готового контракту:

```
public interface IProfileService {
    Task CreateUserAsync(UserRegistrationDto registrationDto);
    Task<UserViewDto> GetUserByIdAsync(string id);
    Task<UserClaimsDto> VerifyUserAsync(UserAuthenticationDto model);
    Task DeleteUserAsync(string id);
    Task EditUserAsync(string id, UserEditDto editingDto);
}
```

Тут ми бачимо базові CRUD операції над сутністю юзера. Також важливо що ми передаємо сутність, яка має тип з приставкою:DTO, що з анг. означає data transfer object або об'єкт передачі даних, в ньому ми приховуємо частину полів які не обов’язково передавати або отримувати користувачеві.

Наприклад при створені профілю, користувач ще не знає свого унікального ідентифікатора, тому він вказує тільки такі поля:

```
public string Email { get; set; }
public string UserName { get; set; }
```

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

З цього виходить ще одна відповідальність сервісів, це так зване відображення або з англ. mapping полів з DTO сутностей на сутності бази даних, або навпаки. Ця операція може бути виконана за допомогою простого співставлення полів або з залученням стороннього пакету, який автоматично співставить поля сутностей в залежності від заданої конфігурації.

Реалізація цього контракту у класі, потребує вказання залежностей у конструкторі, а саме: менеджера користувачів та клас контекст, для роботи з колекціями даних, це самі розповсюджені залежності у всіх реалізаціях сервісів. Так як ці залежності часто використовуються, ми можемо зареєструвати їх за допомогою вбудованого контейнера залежностей, замість того щоб кожен раз створювати об’єкти через оператор new у конструкторі. Для цього створимо клас EntityFrameworkProvider з таким кодом:

```
public static class EntityFrameworkProvider
  {
    public static void AddSqlServerStorage (
      this IServiceCollection services, string connectionString)
    {
services.AddDbContext<CampusContext>(options=>options.UseSqlServer(connectonString));
        services.AddIdentityCore<User>()
          .AddRoles<Role>()
          .AddEntityFrameworkStores<CampusContext>();       }
  }
```

Метод AddSqlServerStorage є розширенням для інтерфейсу IServiceCollection, цей тип використовується як параметр у методі початкової конфігурації сервісів у

класі Startup нашого Web API. Але зазвичай ми маємо багато допоміжних сервісів, тому це зроблено з метою винесення набору суміжних залежностей в окрему сутність, щоб швидше було редагувати та шукати.

Метод AddDbContext є вбудованою реалізацією реєстрації нашого класу контексту, тут ми вказуємо що плануємо використовувати саме SQL Server як провайдер баз даних, та передаємо для нього параметр підключення, який містить інформацію про назву бази даних, та інші конфігураційні властивості.

Метод AddIdentityCore додає сервіси, які необхідні для керування користувачами, наприклад створення користувачів, хешування паролів тощо. По суті, ця реєстрація зводиться до екземпляра UserManager <User>, але спочатку ми реєструємо всі його залежності, а потім ми можемо отримати екземпляр UserManager <User> з DI контейнера та змінювати електронні адреси, отримувати екземпляри користувача з бази даних і т. ін.

Тепер, налаштувавши всі залежності, перейдемо до реалізації методу CreateUserAsync класу ProfileService. Для початку нам потрібно зрівняти два поля Password та ConfirmPassword, щоб впевнитись що вони однакові, бо навіть не дивлячись на те, що на стороні клієнтського інтерфейсу, може бути встановлена валідація, ми все одно представляємо API, яке може бути доступно в обхід UI частини. Тому ми вказуємо таку умову:

```
if (registrationDto.Password != registrationDto.ConfirmPassword)
        throw new ApplicationException("Passwords don't match");
```

В якій буде видано виключення ApplicationException яке попаде в фільтр виключень, та за одно залогується в консоль, про фільтри більш детально буде описано у підрозділі: *Фільтрація запитів та логування*. Далі нам потрібно впевнитись що не існує користувача з однойменною поштою або користувацьким нікнейом, тут можна скористатися вже доданим у залежності менеджером користувачів, в якому ми викликаємо відповідні методи для перевірки, як видно з коду нижче:

```
var userWithSameEmail = await
```

_userManager.FindByEmailAsync(registrationDto.Email); var
userWithSameUserName=await
_userManager.FindByEmailAsync(registrationDto.UserName);

if (userWithSameEmail != null || userWithSameUserName != null)
throw new ApplicationException("User already exists");

Якщо ж хоча б щось вже існує, ми видаємо вже згадане виключення.

Надалі нам залишається лише викликати метод CreateAsync на менеджері користувачів, де ми відобразимо поля які прийшли нам з DTO об’єктом у екземплярі сутності User. Також при відображенні ми створюємо пусту кімнату за замовченням, яка буде виконувати роль віртуальної кімнати, та покривати початкові вимоги з підрозділу *Бізнес вимоги до сервісу*.

Та перед виходом з методу ми повинні перевірити статус реєстрації користувача, за допомогою поля Succeeded об’єкта IdentityResult, яке скаже нам, чи вдалася операція, чи ні. У цьому випадку можуть виникнути помилки про порушення правил конфігурації сутностей, наприклад поле Email прийшло зі значенням null, і якщо ми не перевіримо реєстраційний статус, то користувач може навіть і не підозрювати що реєстрація була неуспішною.

Усі повідомлення які ми вказали у конструкторі класу ApplicationException, передаються до обробки у компонент реєстрації у користувацькому інтерфейсі, які у подальшому будуть представлені у вигляді сповіщень з червоним рівнем важливості.

Наступним кроком буде реалізація методу GetUserByIdAsync, який по своїй структурі схожий з рештою методів контракту, та щоб не повторюватися буде описаний тільки він. У ньому ми використовуємо вже знайомий менеджер користувачів, який вертає нам юзера по унікальному ідентифікатору. Та у кінці ми робимо обернену операцію відображення полів, як видно у коді нижче:

var user = await _userManager.FindByIdAsync(id);
if (user == null)

```
    throw new ApplicationException("User with this ID doesn't exist");

return new UserViewDto
{
    Email = user.Email,
    UserName = user.UserName,
    CreatedOn = user.CreatedOn,
    FullName = user.FullName
};
```

4.2 Розробка контролерів як інтерфейсу для користувачів WEB API

4.2.1 Розробка контролера для роботи з подіями

Для того щоб зробити наш сервіс доступним для користувачів, ми повинні створити контролер. Контролер використовується для визначення та групування набору дій. Дія або також метод дії - це метод в контролері, який обробляє запити. Контролери групують подібні дії разом, щоб окреслити доступ до якогось ресурсу(в нашому випадку подій користувача). Сукупність дій дозволяє спільно застосовувати загальні набори правил у REST сервісах, такі як маршрутизація, кешування та авторизація.

Також контролери повинні дотримуватися принципу явних залежностей. Існує кілька підходів до реалізації цього принципу. Якщо для кількох дій контролера потрібен один і той же сервіс, то треба використовувати інжектор конструктора для запиту цих залежностей. Якщо сервіс потрібен лише одній дії, то можна використати Action Injection, для отримання необхідних залежностей.

Взагалі контролер вважається абстракцією на рівні інтерфейсу. Його обов'язки полягають у забезпеченні достовірності даних, та виборі того, який UI компонент (або результат для API) слід повернути. Натомість контролер делегує усі ці,

та інші завдання до відповідальних сервісів.

Отже для того щоб дати можливість обмінюватись інформацією про події користувача, ми додамо до конструктора два сервіси як у коді нижче:

```
private readonly IEventService _eventService;
private readonly IClaimExtractionService _claimExtractionService;

public EventController(IEventService eventService,
   IClaimExtractionService claimExtractionService)
{
   _eventService = eventService;
   _claimExtractionService = claimExtractionService;
}
```

Сервіс EventService відповідає за операції обміну даними з БД, набір його методів дуже схожий з іншими сервісами, бо він надає звичайні CRUD операції над подіями. Всередині кожен метод використовує розширяючи методи LINQ, та клас контекст.

ClaimExtractionService має лише один метод GetUserIdFromClaims, що повертає Id користувача який звернувся до контролера. Це робить дії контролера дуже універсальними, та запобігає виникненню колізій. Ось як виглядає використання цього методу у дії додавання нової події:

```
public async Task<int> AddEvent(EventAddDto eventDto) =>
    await
_eventService.AddEvent(_claimExtractionService.GetUserIdFromClaims(), eventDto);
```

Кожен раз коли будь-який користувач буде запитувати цю дію, ми матимемо унікальний виклик GetUserIdFromClaims(), який не пересічеться з іншими юзерами. Аргумент eventDto це модель яка буде надіслана в рамках тіла запиту, вона повинна мати формат JSON, та включати усі необхідні поля з класу EventAddDto.

Але щоб мати можливість виконувати запити по відношенню до

EventController(назва класу контролера) нам потрібно бути авторизованими.

[ApiController]

[Authorize]

[ApplicationExceptionHandler]

[Produces("application/json")]

[Route("api/[controller]")]

public class EventController : ControllerBase

Про це свідчить другий атрибут зверху. Перед відправкою запиту клієнтська сторона повинна додати до заголовку токен, який і є підтвердженням того, що користувач авторизований. Та якщо токен достовірний, та його строк дії не закінчився, ми отримаєм статус-код, що свідчить успішний(код 200) запит, в іншому випадку нам повернеться статус-код 401 або Unauthorized. Це зроблено з ціллю того, щоб тільки зареєстровані та авторизовані користувачі могли відсилати запити до цього(події) ресурсу, це один з найважливіших аспектів безпеки.

Також у нашому наборі атрибутів(четвертий зверху) вказано, що даний контролер буде повертати результати у форматі JSON, також можливо отримувати XML формат.

Що до останнього атрибута, то він визначає шаблон, за яким можна звернутися до цього контролера. Це означає що шлях бути виглядати як api/Event, слово Controller просто опускається при запиті, це зроблено з ціллю спрощення іменування шляхів, а в коді ми залишаємо повне ім’я для ідентифікації класу.

У наборі доступних дій контролера, ми будемо бачити один і той же шлях до нього, у різних типів запитів є свої так звані HTTP дієслова, тип цих дієслів вказується над діями, ось як це виглядає для дії видалення події:

[HttpDelete]

public async Task DeleteEvent(string eventId)=> await _eventService.DeleteEventById(_claimExtractionService.GetUserIdFromClaims(),eventId);.Тобто щоб звернутися до цієї дії ми маємо вказати HTTP метод Delete, та шлях api/Event, а також вказати

у параметрах запиту ідентифікатор існуючої події.

4.2.2 Діаграма станів та інтерфейс взаємодії з Web API

Детальний процес обробки станів WebAPI під час запиту, буде розглядатися на прикладі авторизації користувача, та вже описаного контролера подій.

Першим кроком у майже кожному запиті є отримання авторизаційного токена, як видно з діаграми Додатка А, користувач ініціює перший стан вказуючи тіло запиту, його можна вказати за допомогою вбудованого графічного інтерфейсу Swagger[7](див. Додаток В для повного списку дій усіх контролерів), як видно з рис. 4.3.

Рисунок 4.3 – Поле для формування тіла запиту на авторизацію

Після виконання запиту ми маємо отримати результат як на табл. 4.1.

Таблиця 4.1 – Результат запиту POST /api/profile/auth

Code	Response body	Response headers
200	“токен у форматі base 64”	content-type: application/json; charset=utf-8 date: Tue, 04 May 2021 15:24:02 GMT server: Kestrel isActive: true transfer-encoding: chunked

Щоб продовжити використовувати інші дії які помічені знаком замка(див. Додаток В), ми повинні скопіювати токен та занести його в поле авторизації, яке з’явиться після натискання на кнопку Authorize(див. Додаток В), та для того щоб сервіс зрозумів що це саме токен, необхідно додати слово Bearer[6] перед текстом токену. Після цього ми можемо користуватися усім спектром дій, але ця можливість залишається ненадовго, тому що у майбутньому планується додати шар аутентифікації, де кожен користувач буде мати роль, яка визначає набір доступних дій. Наприклад студент повинен бачити інтерфейс Swagger по іншому, там не повинно бути можливості редагувати та видаляти події.

Що до роботи з подіями, то як видно з Додатка Б переходи між станами доволі подібні до процесу авторизації користувача. Тільки ми маємо більше вузлів в області “База даних”, тому що дані мають ієрархічну структуру, і потребують поглибленого проходження по властивостям у випадку завантаження через EF. Також ми маємо додатковий стан, який відповідає за відображення даних, де ми в більшості випадків форматуємо дати подій.

Реалізація Swagger інтерфейсу достатньо проста, ми повинні завантажити пакет Swashbuckle.AspNetCore до Web API проекту. Потім додати генератор Swagger до колекції служб у методі Startup.ConfigureServices, та увімкнути проміжне програмне забезпечення для обробки сформованого документа JSON з описом дій, та інтерфейсу Swagger як видно з коду нижче:

```
public void Configure(IApplicationBuilder app)
{
    app.UseSwagger();
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    });
}
```

Самі ж дії можуть мати XML коментарі зверху, для того щоб описати своє призначення, аргументи та значення яке повертається.

Ось наприклад повний опис дії видалення профілю, з додатковим тегом remarks для метаданих контролера:

```
/// <summary>
/// Delete profile.
/// </summary>
/// <remarks>
/// Request
///
///     DELETE /api/profile
///     Authentication: Bearer {token}
///     Content-Type: application/json
///
/// </remarks>
/// <returns>State transfer model.</returns>
/// <response code="200">Profile is deleted.</response>
/// <response code="401">User is unauthorized.</response>
```

Розробники, які використовують веб-API, найбільше стурбовані тим, що повертається, зокрема типами відповідей та кодами помилок (якщо вони не є стандартними). Типи відповідей та коди помилок позначаються у тегах response та анотаціях даних[8]. Цей опис дозволяє зробити інтерфейс Swagger більш інформативним з точки зору документації, користувачу не обов’язково зв’язуватись з розробником Web API щоб уточнити деталі дій.

4.2.3 Фільтрація запитів та логування

Фільтри в нашому Web API дозволяють виконувати ряд дій до, та після будь-якого запиту, це може бути корисно коли ми хочемо відслідковувати якісь конкретні

данні або загальну активність користувача по відношенню к тим чи іншим діям API. Існують такі види фільтрів:

Фільтри авторизації виконуються в першу чергу. Цей фільтр допомагає нам визначити, чи авторизований користувач для поточного запиту чи ні. Якщо користувач не має права на цей запит, ASP.NET Core припиняє подальшу його обробку і у більшості випадків далі спрацьовує фільтр виключень.

Ресурсні фільтри обробляють запит після авторизації. Він може запускати код до і після виконання решти фільтрів. Він виконується до того, як відбувається прив'язка моделі. Також з його допомогою можна здійснити кешування часто використовуваних даних.

Фільтри дій запускають код безпосередньо перед і після виклику методу дії контролера. Також ми можемо маніпулювати аргументами перед тим, як вони будуть передані в дію.

І самий поширений вид це *фільтри виключень*, їх можна використовувати для обробки винятків, які генеруються у веб-API. Цей фільтр виконується, коли метод дії викидає необроблений виняток. Також треба враховувати, що фільтр винятків не вловлює виняток HttpResponseException[9], оскільки він спеціально розроблений для повернення HTTP відповіді.

Фільтрація може бути особливо корисна коли ми працюємо з контекстом запиту. Наприклад нам потрібно реалізувати відслідковування кількості запитуваних подій користувача, за один запит, бо запит 1000 і більше елементів може створити проблему для сервера та мережі, якщо вони не мають достатню масштабованість та пропускну здатність відповідно.

Тому ми можемо реалізувати обмеження на рівні запиту, щоб потік управління програмою навіть не дійшов до рівня роботи з даними. Для цього ми використаємо асинхронний фільтр дій, він доволі простий у реалізації, та потребує лише число обмеження у конструкторі. А також реалізацію інтерфейсу IAsyncActionFilter[10] й його методу OnActionExecutionAsync. Сама реалізація виглядає таким чином:

```
public async Task OnActionExecutionAsync(ActionExecutingContext context, ActionExecutionDelegate next)
        {
            var requestedItemsValue = Convert.ToInt32(context.HttpContext.Request.Query[TargetQueryParameter]);
            if (requestedItemsValue > _limiterValue)
            {
                context.Result = new ObjectResult(FailureResponseMessage) { StatusCode = 403 };
            }
            else
            {
                await next();
            }
        }
```

Так як цей фільтр використовується прицільно на дію отримання подій користувача, то ми повинні відслідковувати конкретний параметр під назвою “items”, ім’я якого збережено у константі TargetQueryParameter. Значення цього параметру представляється числом, яке після вилучення з запиту зрівнюється з числом обмеження, яке було передане до конструктора цього фільтра. І якщо користувач API, перевищив допустимий ліміт хоча б на одну одиницю, то ми передаємо в якості результату 403 статус код з конкретним повідомленням. У іншому випадку ми просто передаємо виконання наступному фільтру дії, або безпосередньо самій дії.

Логування в нашому Web API виконує дві ролі, це документування викликів дії, та логування виданих помилок, текст цих джерел інформації про активність додатка, відображається у консолі, коли додаток запущений локально як на рис. 4.4.

Також є можливість логувати ці данні у сторонні сервіси, наприклад в splunk[11]. Він допоможе візуалізувати логи які згенеровані API, у графіки активності,

а також відслідковувати аномальні активності, які можуть нести загрозу безпеці додатка та його даним.

Рисунок 4.4 – Список логів після авторизації та запиту подій

У коді таке логування потребує реалізацію IActionFilter інтерфейсу, та двох методів OnActionExecuting ,OnActionExecuted, які викликаються до, та після виконання дії контролера. Код цих двох методів майже ідентичний, та виглядає таким чином:

```
_logger.LogInformation(JsonSerializer.Serialize(new
EntryPointLoggingMessage
{
    Date = DateTime.Now,
    Header = LoggingHeader.Info.ToString(),
    Origin = SenderName,
    ActionName = ActionName,
    Mode = LoggingMode.Entry.ToString()
}));
```

Де _logger це вбудована реалізація механізму логування, яка може бути додана до конструктору будь-якого класу. Та метод LogInformation, який приймає серіалізований варіант об’єкту з інформацією про конкретний виклик дії. У самому об’єкті, ми вказуємо дату коли ця подія виникла, та ім’я відправника з назвою дії, це

дасть можливість визначити, з якого контролера та дії було відправлено це повідомлення. Єдина різниця між реалізацією цих двох методів з інтефейсу IActionFilter, це значення поля Mode, у коді вище маємо LoggingMode.Entry, це значить що ми у методі OnActionExecuting, у OnActionExecuted будемо мати LoggingMode.Leave, так як ми вказуємо у повідомленні, що конкретна дія завершилась.

4.2.4 Обробка винятків

Можливості обробки винятків допомагають нам боротися з непередбаченими помилками, які можуть з'явитися в нашому коді. Для обробки винятків ми можемо використовувати блок try-catch[12], а ключове слово finally для очищення ресурсів після цього.

Незважаючи на те, що немає нічого поганого в блоках try-catch, ми можемо витягти всю логіку обробки винятків в єдине централізоване місце. Роблячи це, наші дії контролерів стають більш читабельними, а процес обробки помилок більш підтримуваним та гнучким. Якщо нам знадобиться модифікувати обробку якогось специфічного виключення, ми можемо це зробити в рамках одного класу, замість того щоб шукати та переписувати набір місць з цим конкретним виключенням.

В більшості випадків ми видаємо виключення, або вказуємо що виникла помилка в рамках сервісів, в яких при запиті даних ми отримаємо null значення, ось приклад такого випадку:

```
var events = classroom?.Events;

    if (events == null)
        throw new ApplicationException("There are no events for given user");
```

Тут ми видаємо виключення типу ApplicationException[13] з повідомленням що не існує подій для конкретного користувача. Після цього це виключення передається до фільтру виключень, який реалізує інтерфейс IExceptionFilter з одним методом

OnException який приймає контекст виключання. Код цього методу виглядає таким чином:

```
if (context.Exception.GetType() == typeof(ApplicationException)){
    _logger.LogWarning(JsonSerializer.Serialize(new
EntryPointLoggingMessage{
        Date = DateTime.Now,
        Header = LoggingHeader.Info.ToString(),
        Origin = "ApplicationExceptionFilter",
        ActionName = context.Exception.Message,
        Mode = LoggingMode.Leave.ToString() }));
    context.Result = new ContentResult
    {
        StatusCode = (int)HttpStatusCode.BadRequest,
        Content = context.Exception.Message
    };
    context.ExceptionHandled = true;
}
```

Процес логування виключення дуже схожий з звичайним логуванням при звертанні до дій контролера, але в назві дії ми вказуємо повідомлення виключення. Та модифікуємо результат вказуючи статус код 400, а в кінці обробки ми задаємо властивості ExceptionHandled значення true, як знак того що виключення оброблено.

Хоча фільтри винятків корисні для виловлення різнотипних помилок, які виникають у рамках дій Web API, але вони не настільки гнучкі, як вбудований механізм обробки винятків. Він може бути проміжним програмним забезпеченням у конвеєрі запитів, або технічно це має назву middleware[14]. Для обробки помилок існує UseExceptionHandler middleware. Корпорація Майкрософт рекомендує використовувати UseExceptionHandler[15], якщо нам не потрібно виконувати обробку помилок зі складною логікою логування, залежно від того, яку дію MVC або веб-API

вибрано, вміст відповіді може бути змінений зовні контролера. У Web API ASP.NET 4.0, одним із способів це зробити, було використання типу HttpResponseException[16].

Також іноді у об’єкт логування виключення додають трасування стека, у ньому знаходиться рядкове представлення безпосередніх кадрів у стеку викликів. Це дає змогу переглянути шлях викликів методів, який потік управління програмою пройшов до виникнення виключення, або до точки зупинки.

Якщо ми отримали трасування стека зовні (наприклад, із звіту про помилку), ми можемо відкрити його у вікні: “провідник стеку” у Visual Studio, та перейти до коду, звідки виник відповідний виняток. У трасуванні стека файли, типи та методи відображаються як гіперпосилання. Ми можемо натиснути на них, щоб відобразити відповідні елементи в редакторі.

Ці деталі відлагодження додатка дуже важливі, так як у більшості випадків розробник працює з кодом, який написаний іншою людиною, в рамках великого проекту. А знання та вміння пошуку помилок, може значно зменшити час на їх виправлення. А продвинуті розширення для середовищ розробки, допоможуть наочно ознайомитися з контекстом помилки.

ВИСНОВКИ

Проаналізувавши відгуки користувачів різних платформ дистанційного навчання, було зроблено висновок, що питання централізації каналів обміну учбової інформації залишається актуальним. Бо найчастіша потреба полягає у зменшені кількості сторонніх додатків для спілкування стосовно навчання, та обміну учбовими матеріалами.

1. Виконавши інтерв'ю з кількома користувачами систем дистанційного навчання різних університетів, було описано 4 головних особливості застосунку, яких не вистачало їм. У цей список входить:

    1.1 Наявність повноцінного календаря з трьома режимами перегляду(місяць, неділя, день).

    1.2 Можливість отримувати автоматичні сповіщення про перенесення або відміну занять.

    1.3 Розділення дисциплін по віртуальним кімнатам, та можливість самостійно створювати кімнати із запрошенням інших користувачів платформи.

    1.4 Можливість виконувати лабораторні по програмуванню прямо в додатку.

2. За допомогою запиту студентського пакету з програмним забезпеченням та учбовими акаунтами на хостингових платформах, було активовано обліковий запис у хмарній платформі Azure. Після цього було створено екземпляр Azure SQL server, та пустий контейнер для веб додатка. Та при появі доменної моделі, був згенерований рядок підключення, щоб запитувати данні вже з бази даних Azure, а не локально. Також протягом розробки було сконфігуровано набір правил брандмауера, щоб інші члени команди могли запускати додаток локально, без встановлення громіздкого екземпляру SQL Server 2017.

3. Щоб зробити розробку вихідного продукту більш швидшою та якіснішою,

були введені практики неперервної збірки та неперервного розгортання. Це дало можливість автоматизувати запуск тестів та виявляти помилки ще на стадії запиту на вливання коду до основної гілки. Також були додані практики перевірки коду на слідування конвенціям мови програмування. Що наприклад для Angular проекту, не давало можливість використовувати неявний тип any. На початкових стадіях розробки значна кількість перевірок завершались з помилками, бо не всі члени команди звикли до конвенцій, та не у всіх були встановленні розширення для перевірки коду в реальному часі. Після усіх перевірок код ставав готовим до розгортання на виробничому сервері, щоб не виконувати оновлення кожен раз в ручну, цей процес був автоматизований за допомогою подібного набору перевірок. Але останнім кроком у цьому наборі, був доданий авто згенерований скрипт, який відправляв результуючий набір бібліотек, до вже існуючого екземпляру Azure App Service.

4. Після визначення головних сутностей доменної моделі, таких як: користувач, роль користувача, подія, віртуальні кімната. Було спроектовано модель відношення сутностей, яка в свою чергу відобразилась у вигляді набору класів доменної області. Потім на основі цього набору класів було згенеровано базу даних, за допомогою підходу “Code First”. Модель(C# класи) в процесі розробки незначно змінювалась, а база даних синхронізувалась з нею через створення міграцій в EF Core.

5. Для реалізації веб сервісу було обрано цибулину архітектуру, що в результаті дало набір слабко зв'язних бібліотек, які спілкуються через інтерфейси. В ролі інтерфейсу для користувачів, в особливості для клієнтської частини додатка, було створено три контролера для управління користувачами, віртуальними кімнатами та подіями. За обробку та логування помилок відповідальні фільтри, які звітують детальні данні про виключення в консолі Вихідна версія додатка повністю базується в хмарі. Що дає можливість тестово користуватись застосунком на різних платформах та пристроях.

Здобутий додаток є чудовою платформою для практики різних навичок розробки, у тому числі: тестування, проектування та адміністрування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What are Push Notifications? [Електронний ресурс]. – Режим доступа: https://developers.google.com/web/ilt/pwa/introduction-to-push-notifications#what_are_push_notifications.
2. What is Google Workspace for Education? [Електронний ресурс]. – Режим доступа: https://support.google.com/a/answer/139019?hl=en#zippy=%2Cwhat-is-google-workspace-for-education.
3. Introduction to REST [Електронний ресурс]. – Режим доступа: https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design#introduction-to-rest.
4. Entity Framework Core tools reference [Електронний ресурс] . – Режим доступа: https://docs.microsoft.com/en-us/ef/core/cli/dotnet#installing-the-tools.
5. SQL scripts [Електронний ресурс] . – Режим доступа: https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/applying?tabs=dotnet-core-cli#sql-scripts.
6. Bearer Authentication [Електронний ресурс] . – Режим доступа: https://swagger.io/docs/specification/authentication/bearer-authentication/.
7. What Is Swagger? [Електронний ресурс] . – Режим доступа: https://swagger.io/docs/specification/2-0/what-is-swagger/.
8. Data annotations [Електронний ресурс] . – Режим доступа: https://docs.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle?view=aspnetcore-5.0&tabs=visual-studio#data-annotations.
9. HttpResponseException [Електронний ресурс] . – Режим доступа: https://docs.microsoft.com/en-us/aspnet/web-api/overview/error-handling/exception-handling#httpresponseexception.
10. IAsyncActionFilter Interface [Електронний ресурс] . – Режим доступа: https://docs.microsoft.com/en-

us/dotnet/api/microsoft.aspnetcore.mvc.filters.iasyncactionfilter?f1url=%3FappId%3DDev16IDEF1%26l%3DEN-US%26k%3Dk(Microsoft.AspNetCore.Mvc.Filters.IAsyncActionFilter);k(DevLang-csharp)%26rd%3Dtrue&view=aspnetcore-5.0.

11. What is Splunk? [Електронний ресурс] . – Режим доступа: https://www.guru99.com/splunk-tutorial.html#:~:text=Splunk%20is%20a%20software%20platform,%2C%20alerts%2C%20dashboards%20and%20visualizations.

12. Robert C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship 1st Edition // Pearson. - August 1, 2008. - 1st edition. – С. 105-106.

13. What is ApplicationException for in .NET? [Електронний ресурс] . – Режим доступа: https://stackoverflow.com/questions/5685923/what-is-applicationexception-for-in-net.

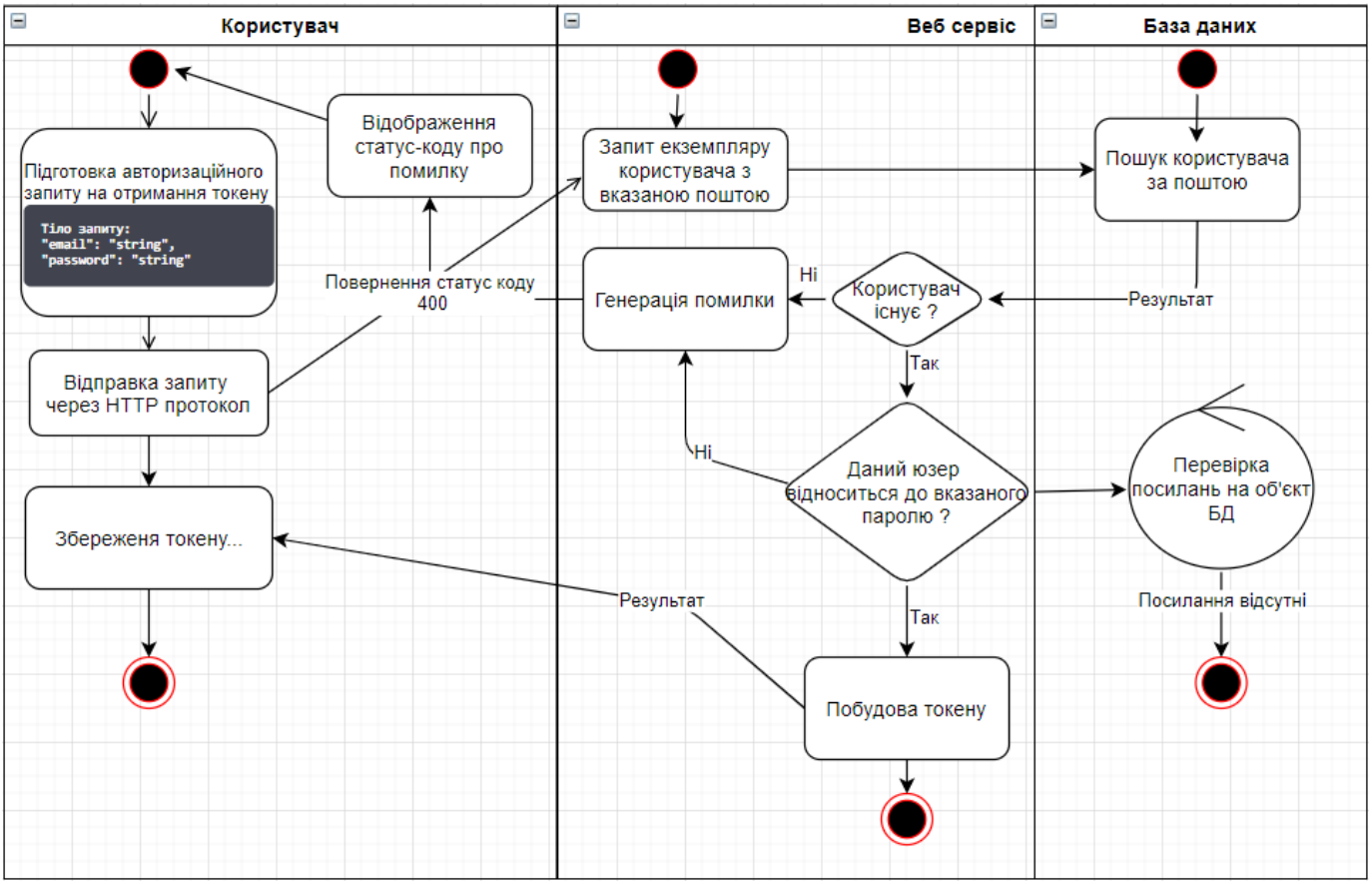
14. ASP.NET Core Middleware [Електронний ресурс] . – Режим доступа: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-5.0.

15. Exception handler page [Електронний ресурс] . – Режим доступа: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/error-handling?view=aspnetcore-5.0#exception-handler-page.

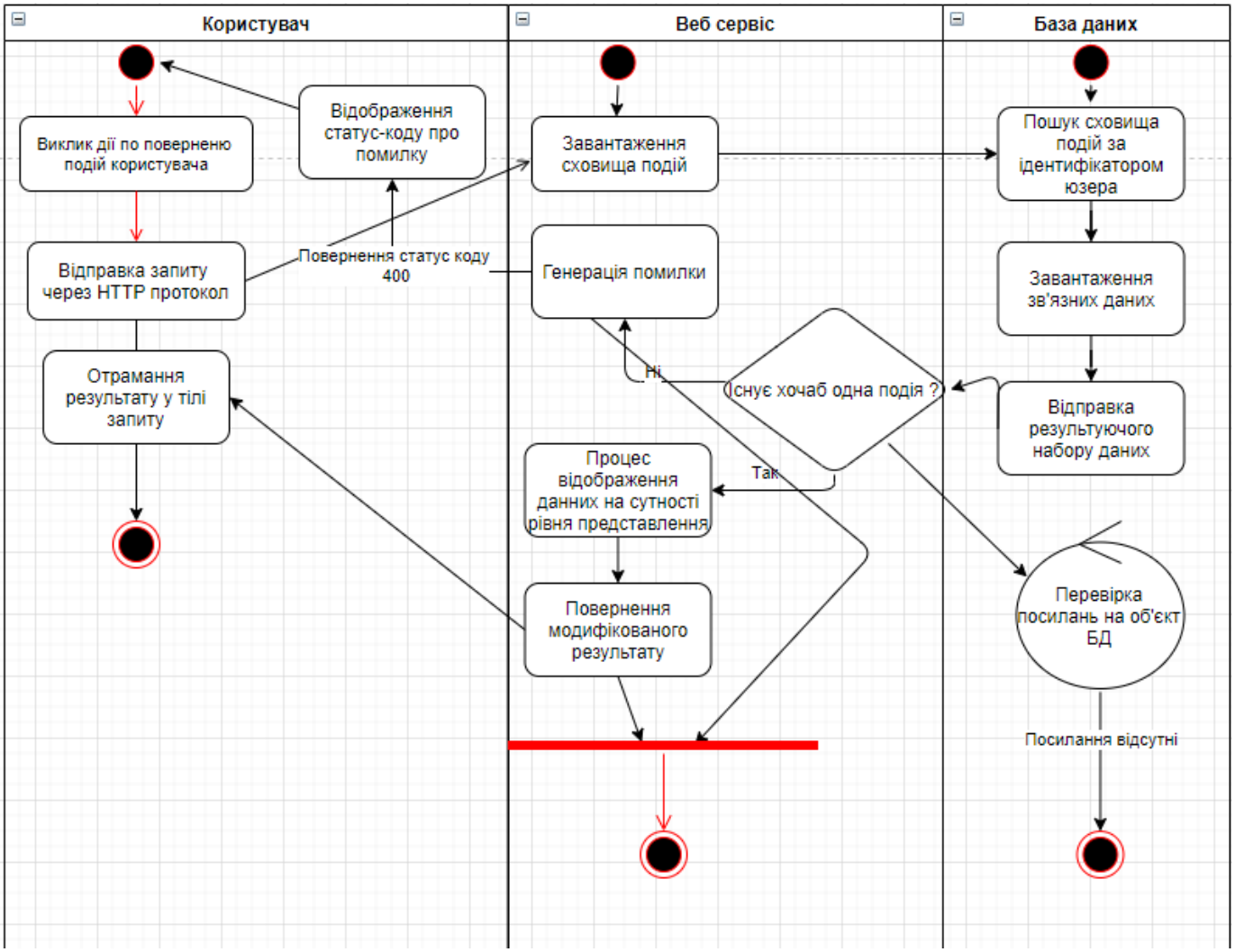
16. Throw HttpResponseException or return Request.CreateErrorResponse? [Електронний ресурс] . – Режим доступа: https://stackoverflow.com/questions/12519561/throw-httpresponseexception-or-return-request-createerrorresponse.

1. Проставити посилання по тексту

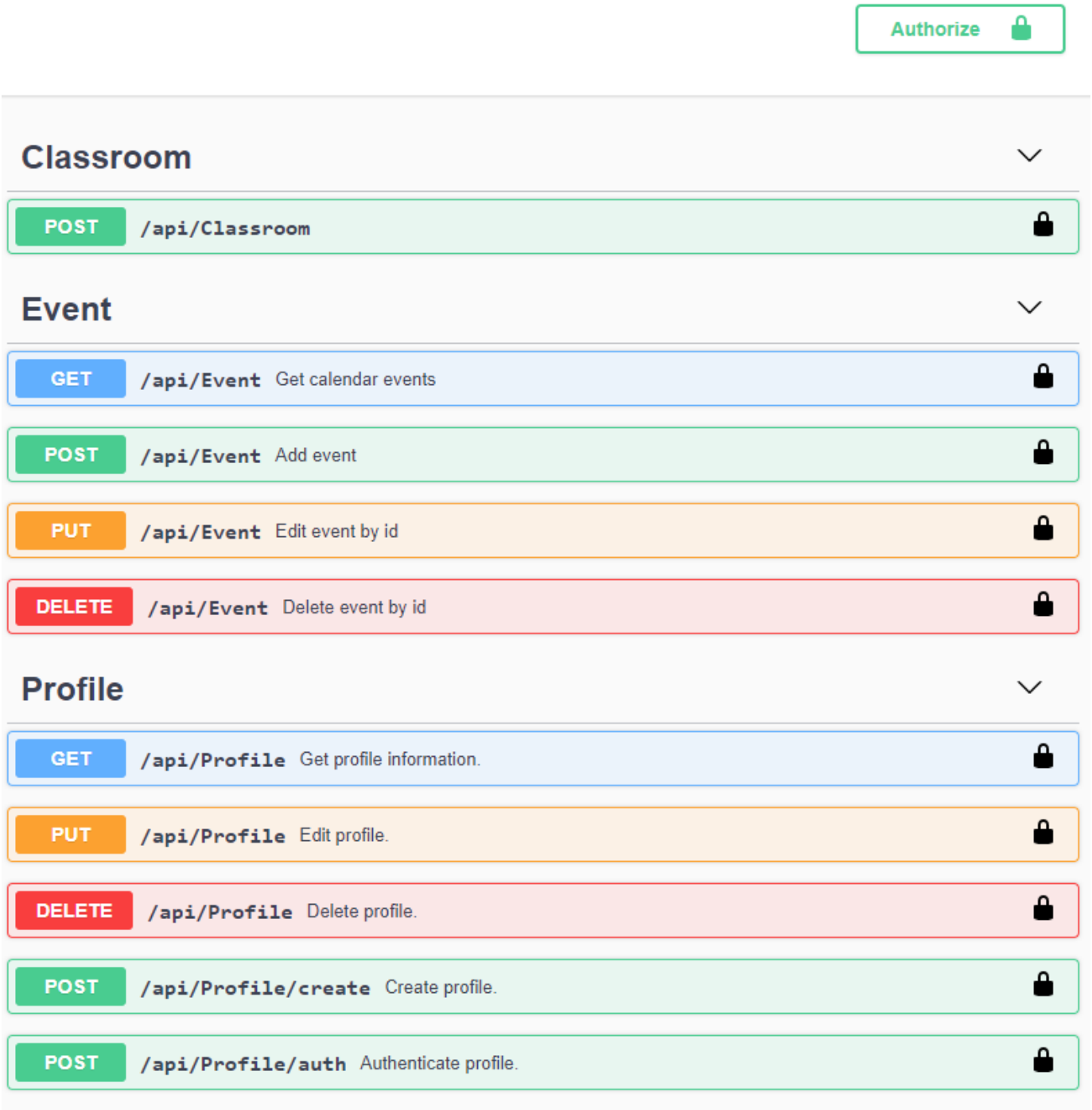
ДОДАТОК А


Користувач
Веб сервіс
База даних
Підготовка авторизаційного запиту на отримання токену
Тіло запиту:
"email": "string",
"password": "string"
Відображення статус-коду про помилку
Відправка запиту через HTTP протокол
Повернення статус коду 400
Збереженя токену...
Запит екземпляру користувача з вказаною поштою
Генерація помилки
Ні
Користувач існує ?
Так
Ні
Даний юзер відноситься до вказаного паролю ?
Так
Результат
Побудова токену
Пошук користувача за поштою
Результат
Перевірка посилань на об'єкт БД
Посилання відсутні

ДОДАТОК Б


Користувач
Веб сервіс
База даних
Виклик дії по поверненю подій користувача
Відображення статус-коду про помилку
Відправка запиту через HTTP протокол
Повернення статус коду 400
Отрамання результату у тілі запиту
Завантаження сховища подій
Генерація помилки
Ні
Існує хочаб одна подія ?
Так
Процес відображення данних на сутності рівня представлення
Повернення модифікованого результату
Пошук сховища подій за ідентифікатором юзера
Завантаження зв'язних даних
Відправка результуючого набору даних
Перевірка посилань на об'єкт БД
Посилання відсутні

ДОДАТОК В


Authorize
Classroom
POST /api/Classroom
Event
GET /api/Event Get calendar events
POST /api/Event Add event
PUT /api/Event Edit event by id
DELETE /api/Event Delete event by id
Profile
GET /api/Profile Get profile information.
PUT /api/Profile Edit profile.
DELETE /api/Profile Delete profile.
POST /api/Profile/create Create profile.
POST /api/Profile/auth Authenticate profile.

ДОДАТОК Г

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«РОЗРОБКА BACK-END ЧАСТИНИ ДЛЯ LEARNING-ДОДАТКУ «CAMPUS» З ВИКОРИСТАННЯМ ФРЕЙМВОРКУ ASP.NET CORE 3.1 ТА ШАБЛОНУ WEB API НА МОВІ C#»

Виконав студент 4 курсу
Групи ПД-42 Даценко.М.А

Київ – 2021

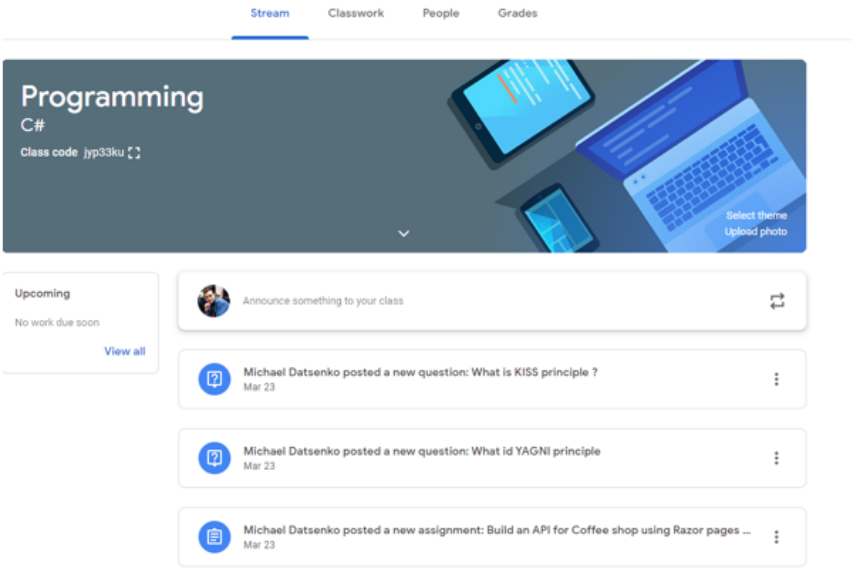
МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

- **Мета роботи:** створити веб сервіс та інфраструктуру для зв'язку з БД та іншими сервісами на основі шаблону Web API фреймворку ASP.NET Core
- **Об'єкт дослідження:** дистанційне навчання в рамках централізованої системи організації учбового процесу учнів шкіл та університетів
- **Предмет дослідження:** серверна частина веб застосунку для обміну та аналізу інформації щодо графіку та завдань учнів

АНАЛОГИ

Google classroom - це веб сервіс розроблений компанією Google LLC, який використовується в закладах середньої та вищої освіти.

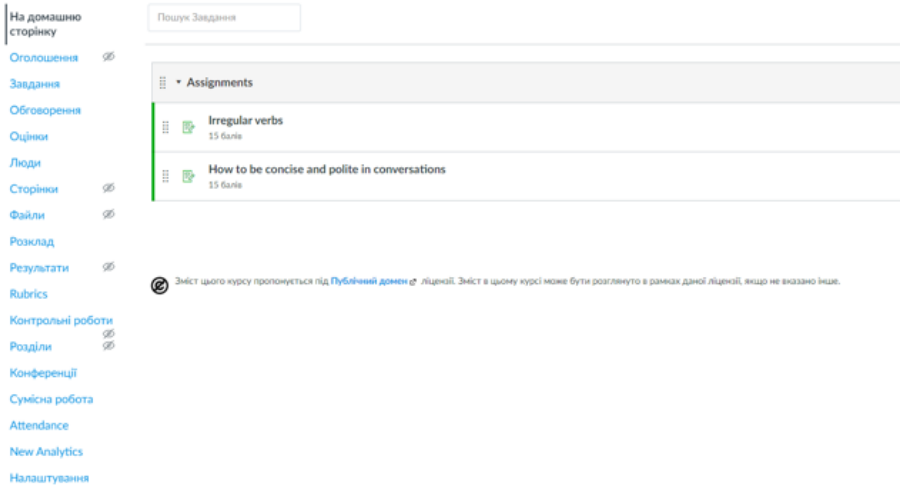
- Його мета спростити та прискорити процес обміну завданнями між учнем та викладачем.
- Кожен предмет може бути представлений окремою кімнатою, зі своїми учасниками, класною роботою та списком оцінок.



АНАЛОГИ

Canvas - це веб-сервіс управління навчанням. Він використовується навчальними закладами, викладачами та студентами для доступу та управління навчальними матеріалами.

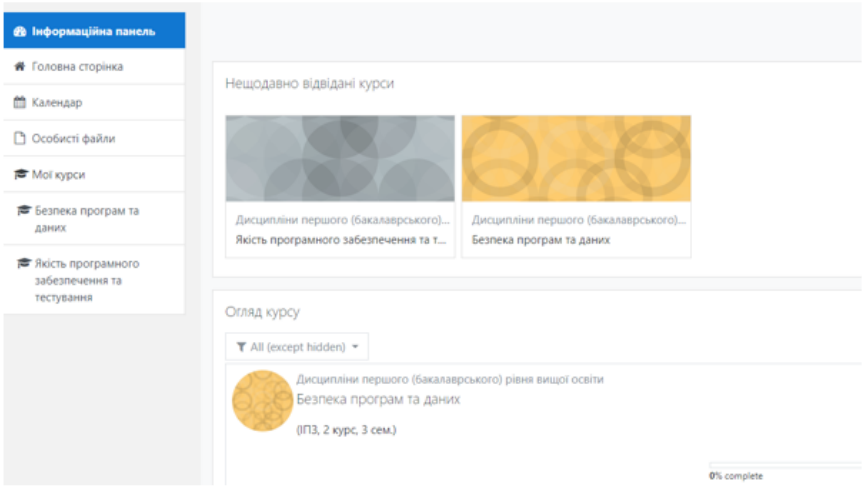
- Включає в себе різноманітні засоби створення та управління курсами, аналітику та статистику курсів та користувачів
- Викладачі можуть надати студентам вичерпні відгуки щодо завдань та тестів, використовуючи SpeedGrader



АНАЛОГИ

Moodle - це навчальна платформа, призначена для надання викладачам, адміністраторам та учням єдиної надійної, безпечної та інтегрованої системи для створення персоніфікованих навчальних середовищ

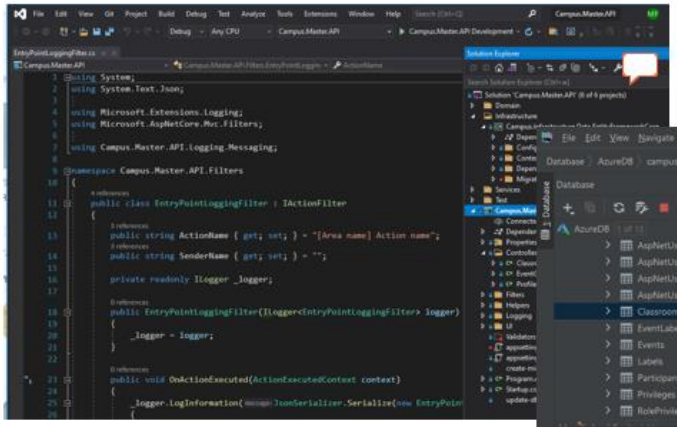
- Moodle надається безкоштовно як програмне забезпечення з відкритим кодом під загальною публічною ліцензією GNU
- Завдяки інтерфейсу, сумісному з мобільними пристроями, вміст на платформі Moodle є адаптивним на різних веб-браузерах.



ТЕХНІЧНІ ЗАВДАННЯ

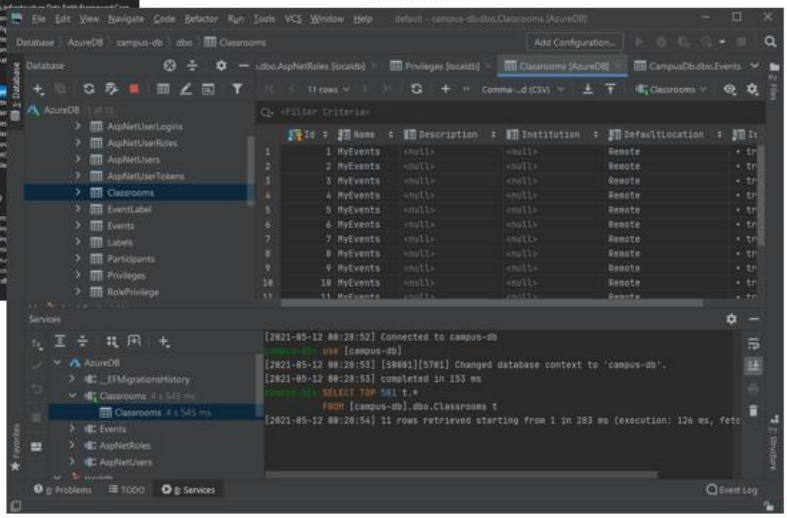
1. Налагодження хостингу в хмарному середовищі та його зв’язку з БД.
2. Впровадження конвеєру CI/CD безпосередньо з платформи GitHub.
3. Розробка моделі БД та її реалізація через ORM фреймворк EF Core.
4. Розробка API головного сервісу та каналів зв’язку з допоміжними сервісами.

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ



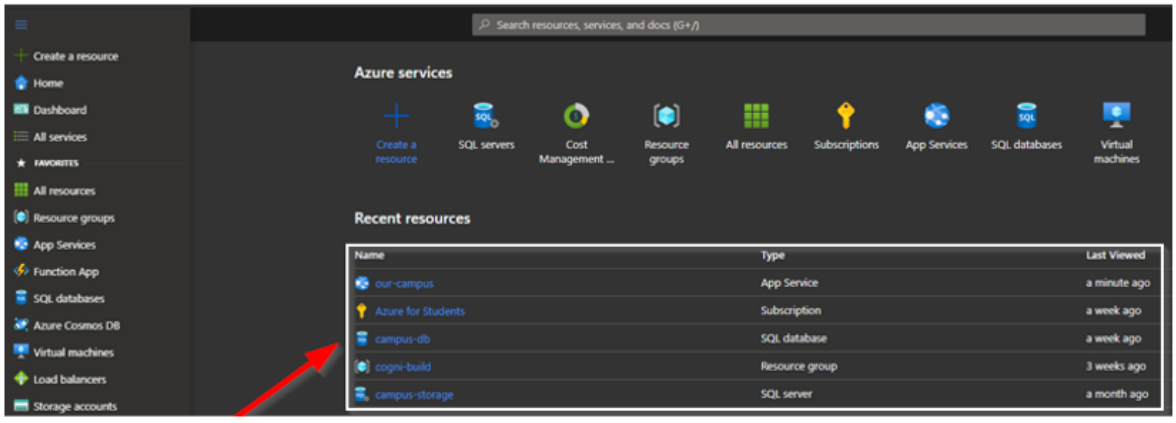
Visual Studio + Resharper

DataGrip



ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

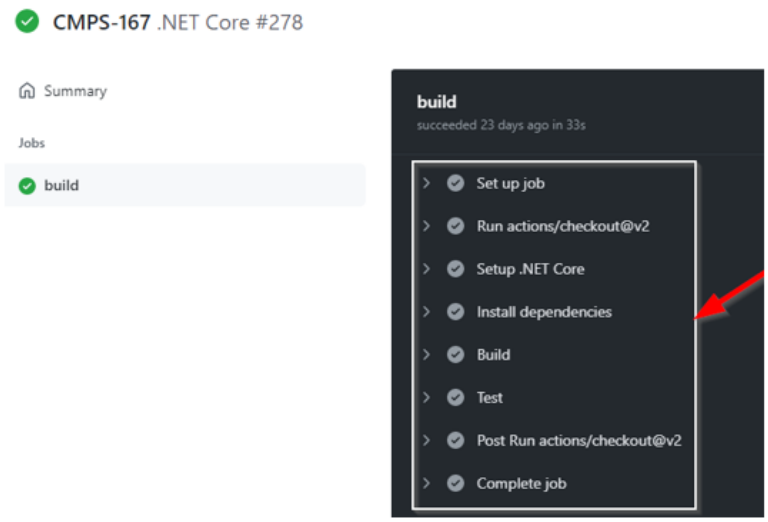
Azure Portal



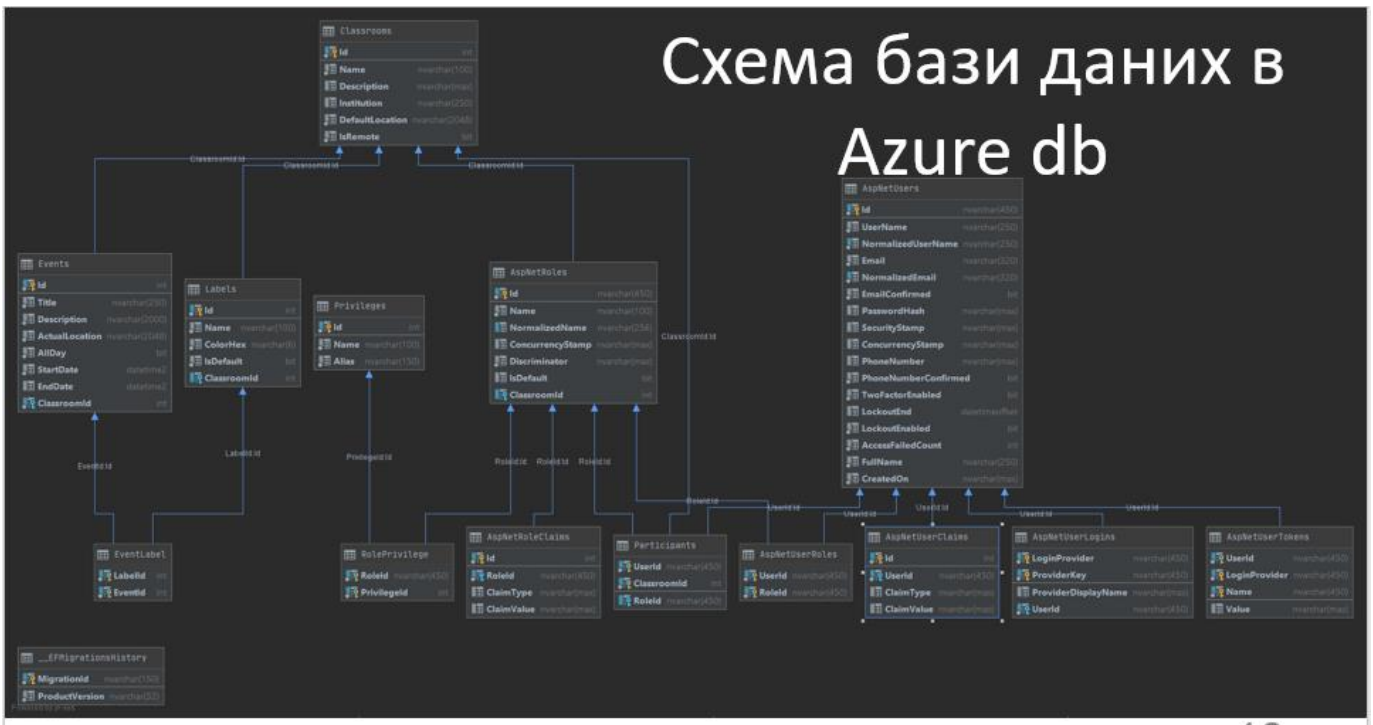
Набір ресурсів які подтримують роботу “Campus”

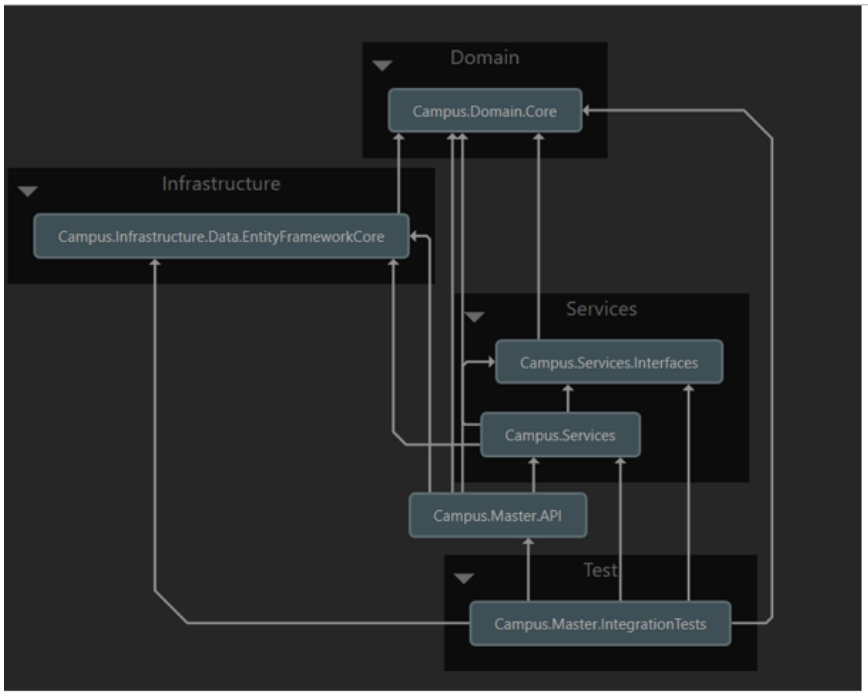
ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

GitHub Actions



Автоматизований набір завдань, які виконуються перед зливанням коду з основною гілкою





Загальна архітектура Web API з розподілом по директоріям


```
public class CampusContext : IdentityDbContext<User>
{
    7 references
    public DbSet<Classroom> Classrooms { get; set; }
    1 reference
    public DbSet<Event> Events { get; set; }
    0 references
    public DbSet<Label> Labels { get; set; }
    0 references
    public DbSet<Participant> Participants { get; set; }
    0 references
    public DbSet<Privilege> Privileges { get; set; }

    0 references
    public CampusContext(DbContextOptions<CampusContext> options)
        : base(options) {}

    0 references
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.ApplyConfiguration(new ClassroomMap());
        modelBuilder.ApplyConfiguration(new EventLabelMap());
        modelBuilder.ApplyConfiguration(new EventMap());
        modelBuilder.ApplyConfiguration(new LabelMap());
        modelBuilder.ApplyConfiguration(new ParticipantMap());
        modelBuilder.ApplyConfiguration(new PrivilegeMap());
        modelBuilder.ApplyConfiguration(new RoleMap());
        modelBuilder.ApplyConfiguration(new RolePrivilegeMap());
        modelBuilder.ApplyConfiguration(new UserMap());

        modelBuilder.ApplyConfiguration(new PrivilegePopulation());
    }
}
```

Клас контекст, який виконує роль шаблону Unit of Work

Метод редагування події користувача

```
public async Task EditEventById(string userId, EventEditDto eventDto)
{
    var defaultClassroom = await GetDefaultClassroomByUserId(userId);

    var eventToEdit = defaultClassroom.Events // ICollection<Event>
        .FirstOrDefault(e :Event => e.Id.ToString() == eventDto.Id);

    if (eventToEdit == null)
        throw new ApplicationException(message:$"Event with {eventDto.Id} id not found for given user");

    eventToEdit.Title = eventDto.Title;
    eventToEdit.Description = eventDto.Description;
    eventToEdit.StartDate = eventDto.Start;
    eventToEdit.EndDate = eventDto.End;
    eventToEdit.ActualLocation = eventDto.Location;
    eventToEdit.AllDay = eventDto.AllDay;

    await _context.SaveChangesAsync();
}
```


Дія отримання усіх подій з віртуальної кімнати

```
/// <summary>
/// Get calendar events
/// </summary>
/// <returns>Collection of events for default classroom</returns>
[HttpGet]
[EntryPointLogging(ActionName = "[Event] Get calendar events", SenderName = "EventController")]
0 references
public async Task<IEnumerable<EventViewDto>> GetEvents()
{
    return await _eventService.GetClassroomEventsByUserId(_claimExtractionService.GetUserIdFromClaims());
}
```

ВИСНОВКИ

1.Було автоматизовано повний цикл перевірки коду та його інтеграції до виробничого сервера
2. Реалізовано архітектуру як доменної моделі так і самого веб сервісу
3. Оптимізовано виконання запитів за допомогою асинхронних контролерів
4. Інтегровано веб сервіс з клієнтською частиною

Здобутий додаток є чудовою платформою для практики різних навичок розробки, у тому числі: тестування, проектування та адміністрування.