Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

Розробка веб-сайту на ASP.NET Core з використанням GraphQL

Текстова частина до курсової роботи

за спеціальністю «Комп'ютерні науки» - 122

Керівник курсової роботи
старший викладач
Борозенний С. О.
_____
(підпис)
"____" _____ 2021 р.
Виконала студентка КН-4:
Яськова Д. В.
"____" _____ 2021 р.

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мережних технологій факультету інформатики

ЗАТВЕРДЖУЮ

старший викладач
Борозенний С. О.
_____
(підпис)

„ ____”_____2020р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу
студентці Яськовій Дарині Вадимівні
факультету інформатики 4 курсу бакалаврської програми
ТЕМА: **Розробка веб-сайту на ASP.NET Core з використанням GraphQL**

Зміст ТЧ до курсової роботи:

Анотація
Вступ
Розділ 1. Дослідження та аналіз предметної області
Розділ 2. Опис застосунку
Розділ 3. Розробка застосунку
Висновок
Список використаної літератури

Дата видачі „ ___” _____ 2020 р.

Керівник _____
(підпис)

Завдання отримав _____
(підпис)

Календарний план виконання роботи

№	Назва етапу	Термін виконання	Примітка
1.	Отримання завдання на курсову роботу	06.10.2020	
2.	Огляд технічної літератури за темою роботи	10.11.2020	
3.	Вибір предметної області для застосунку	20.11.2020	
4.	Проведення дослідження	25.11.2020	
5.	Написання застосунку	30.11.2020	
7.	Перегляд застосунку із викладачем	04.01.2021	
8.	Коригування застосунку	10.01.2021	
9.	Написання текстової частини	10.02.2021	
10.	Створення презентації	01.04.2021	
11.	Захист роботи	17.04.2021	

Зміст

Анотація

У цій роботі детально розглядається розробка API за допомогою фреймворку ASP.NET Core з використанням GraphQL. Основною метою цієї роботи було:

- познайомитись із технологією GraphQL;
- переглянути на якому рівні наразі реалізована бібліотека для інтеграції із цим підходом на C#;
- продемонструвати все це на власному прикладі, в якості якого було обрано електронний журнал для школи.

Вступ

Актуальність теми

На сьогоднішній день, GraphQL стрімко розвивається та розповсюджується серед технологій побудови API. Це може бути спричинене тим, що REST підхід програє у гнучкості у порівнянні із GraphQL.

У 2015 для .NET була створена бібліотека з відкритим кодом, яка імплементує технологію GraphQL. Але наразі вона ще не набула стільки популярності, як на інших мовах типу Javascript або Python. На просторах інтернету можна знайти декілька прикладів із її використанням, але всі вони досить сирі.

Мета дослідження

Дослідження технології GraphQL при побудові вебсайту на основі фреймворку ASP.NET Core.

Постановка задачі

1. Ознайомлення із загальними концепціями GraphQL.
2. Ознайомлення із відповідною бібліотекою на ASP.NET Core.
3. Вибір теми та створення основного застосунку.

Розділ 1. Дослідження та аналіз предметної області

1.1 Загальна інформація

Спочатку оберемо предметну область, яку будемо використовувати для прикладів та побудови основного застосунку. Розглянемо приклад школи, оскільки технології дистанційного навчання стають все більш актуальними, а деталі предметної області усім знайомі.

Школа – це заклад освіти, який зазвичай надає початкову та середню освіту. Зазвичай освітній процес складається з 9 або 11 классів. Учні вивчають предмети. Щоб закінчити навчальний рік, необхідно закрити всі предмети на задовільний бал. Ведення обліку оцінок та відвідування відбувається у класному журналі. Семестрові та річні оцінки вираховуються вчителем на основі проміжних оцінок, виставляються у табелі та надаються учневі.

1.2 Дослідження

Було проведено дослідження щодо наявності схожих програм на ринку та у вільному доступі:

1) Використання Excel

Це програма, яка призначена для роботи із таблицями, створена корпорацією Microsoft. Однією із головних її переваг є можливість налаштування автоматичних розрахунків за допомогою формул та генерація різних видів діаграм. Так як класний журнал складається із таблиць, то його можна було б легко перенести до Excel.

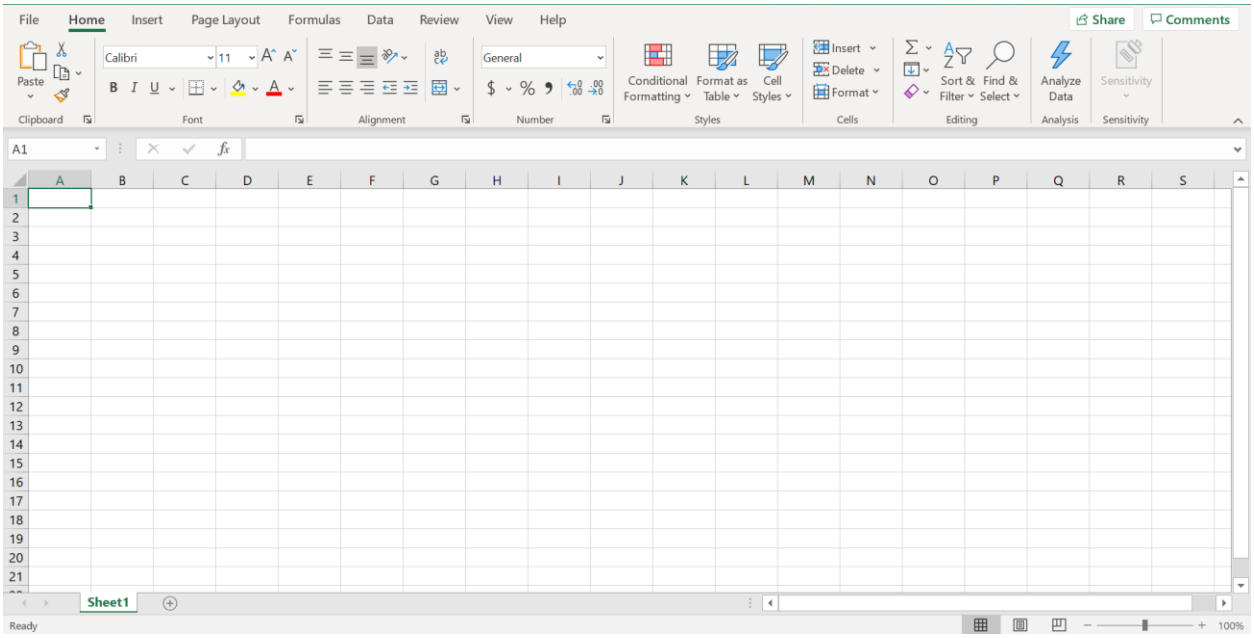


Рисунок 1.1 Інтерфейс програми Excel

Але варто зазначити, що програма Excel може бути не зовсім зручною для використання у якості електронного журналу через те що, вона не призначена для роботи в якості СКБД. Також потрібна людина, яка буде докупи збирати інформацію, тому що кожен вчитель має свій окремий файл.

2) SMLS

Це система, яка використовується дла управління освітніми закладами та навчальним процесом. Через те, що вона є платною, зазвичай її використовують приватні школи. Головною її перевагою є багатофункціональність та наявність усіх необхідних для цієї ПО типів користувачів.

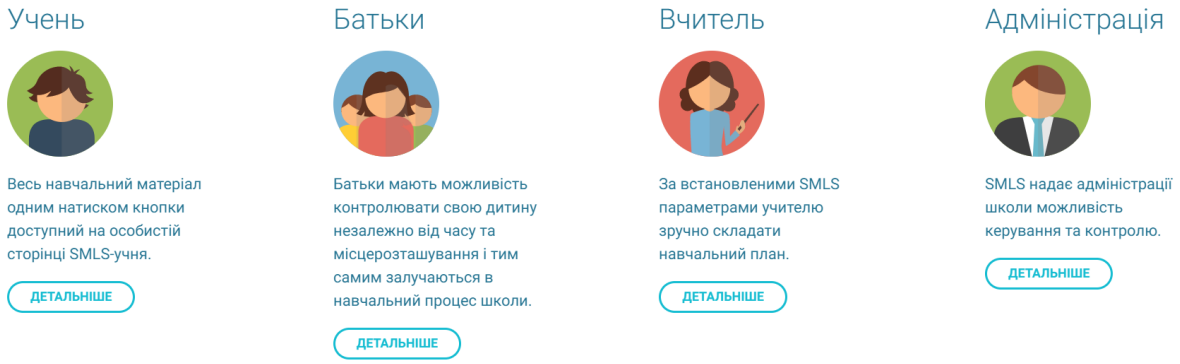


Рисунок 1.2 Типи користувачів системи SMLS[1]

Для вчителів система пропонує:

- інструменти для складання розкладу;
- доступ до онлайн бібліотеки;
- створення звітів тощо.

Для учнів система пропонує:

- доступ до навчальних ресурсів;
- перегляд оцінок та домашніх завдань тощо.

Окрім цього система надає можливість батькам слідкувати за навчальним процессом своїх дітей.

У підсумку можна сказати, що система SMLS пропонує багато й інших рішень, окрім ведення електронного журналу. За допомогою цієї системи

можна повністю автоматизувати роботу школи, починаючи із розкладу класів і закічуючи меню їдальні. Єдиним недоліком цієї системи є те, що вона не є безкоштовною.

Розділ 2. Опис застосунку

Головною метою застосунку є створення електронного журналу, за допомогою якого вчителі зможуть автоматизувати процес ведення оцінок, а адміністратори в будь-який момент будуть мати доступ до журналів усіх класів.

Даний застосунок вирішує проблему дублювання оцінок із журналу у табель. Вчителі зможуть нотувати всі оцінки на сайті, включаючи семестрові і річні, а застосунок в свою чергу буде надавати можливість в один клік переглядати автоматично заповнені табелі для кожного учня і вигружати їх у PDF форматі. Щоб ця функція стала доступною необхідно:

1) Додати учнів до загальної бази.
2) Створити клас X.
3) Додати до класу X учнів із загальної бази.
4) Створити предмети, які читаються у классі X.
5) Виставити оцінки.

2.1 Опис користувачів

У системі існує два типи користувачів – адміністратор та вчитель.

Адміністратор є розширеним типом користувача і має доступ до всіх можливостей, що надаються у системі. Ця роль надається директору та його заступникам. На початку, система надається із root адміністратором, за допомогою якого можна створити акаунти для інших адміністраторів та вчителів. Для цього необхідно ввести пошту майбутнього користувача та придумати для нього пароль, який надалі йому необхідно передати. Адміністратори мають доступ до всіх класів і відповідно до всіх предметів класу.

Вчитель має обмежені дії в порівнянні із адміністратором. Він має доступ лише до деяких класів та до деяких предметів. Наприклад, якщо вчитель А веде математику, то він має доступ до предмету “Математики” у класах X1, X2, …, Xn, а вчитель В, який є класним керівником класу X1, має доступ до всіх предметів класу X1. Таким чином, можна надавати доступ до одного класу багатьом вчителям, але обмежувати його по предметам. Вчитель може виставляти оцінки по предметам, до яких має доступ. Після виставлення всіх оцінок, вчитель може перейти до інтейферу із табелем та завантажити його у PDF форматі.

2.2 Функціональні вимоги

Адміністратор повинен мати можливість:

1. Додавати адміністраторів та вчителів за їх поштою.
2. Редагувати/видаляти інших користувачів.
3. Надавати вчителям доступ до їх класів та предметів.
4. Створювати/редагувати/видаляти класи.
5. Переглядати учнів класу.
6. Додавати/редагувати/видаляти учнів до загальної бази.
7. Додавати учнів до певного класу.
8. Додавати/редагувати/видаляти предмети для певного класу.
9. Ставити/редагувати/видаляти оцінки учневі.
10. Отримувати автоматично згенерований табель учня за рік.

Вчитель повинен мати можливість:

1. Переглядати учнів своїх класів.
2. Додавати нових учнів до загальної бази.
3. Редагувати учнів.
4. Додавати/видаляти учнів до своїх класів.
5. Додавати/редагувати/видаляти предмети для своїх класів.
6. Ставити оцінки за певний предмет за певну дату певному учневі.
7. Редагувати та видаляти оцінки.
8. Отримувати оцінки за семестр та рік.

Розділ 3. Розробка застосунку

3.1 Вибір інструментів

Web-застосунок – це застосунок, в якому в ролі клієнту виступає браузер, а в ролі серверу виступає вебсервер. Переваги полягають у тому, що клієнти не залежать від ПЗ користувача, тому такі застосунки є досить універсальними.

3.1.1 Огляд ASP.NET Core, Node.js

ASP.NET Core – це платформа, створена компанією Microsoft, яка призначена для створення вебзастосунків. Її особливостями є:

- відкритий код;
- кросплатформленість;
- підтримка консольних інструментів.

У проекті цей фреймворк використовується для реалізації GraphQL серверу та роботою з базою даних.

Node.js – це середовище виконання Javascript коду поза браузером. Воно має багато бібліотек з відкритим кодом, зокрема й для інтеграції з Firebase. Саме тому вона й була обрана для реалізації сервісу аутентифікації.

3.1.2 Огляд React, Next.js, Material-UI

React — це JavaScript-бібліотека для створення інтерфейсів користувача[2]. Бібліотека була створена компанією Facebook у 2013 році. Вона є відкритою, її код можна знайти на платформі Github. Ця бібліотека стала дуже популярною з декількох причин.

Вирішення проблеми часткового оновлення. Бібліотека використовує віртуальний DOM, щоб відслідковувати зміни та ефективно оновлювати DOM браузеру.

Компоненти. За допомогою цієї бібліотеки, можна виносити однакові частини HTML коду у окремі компоненти і перевикористовувати їх.

Next.js – це веб-фреймворк, який забезпечує функції рендерингу на стороні сервера та створення статичних вебсайтів на основі бібліотеки React. Фреймворк пропопує безліч можливостей, чере що він і набув стільки успіху.

Декілька способів відображення сторінки. Next.js має дві форми попереднього рендерингу: статичну генерацію та рендеринг на стороні сервера.

Для рендерингу на стороні сервера HTML формується для кожного запиту.

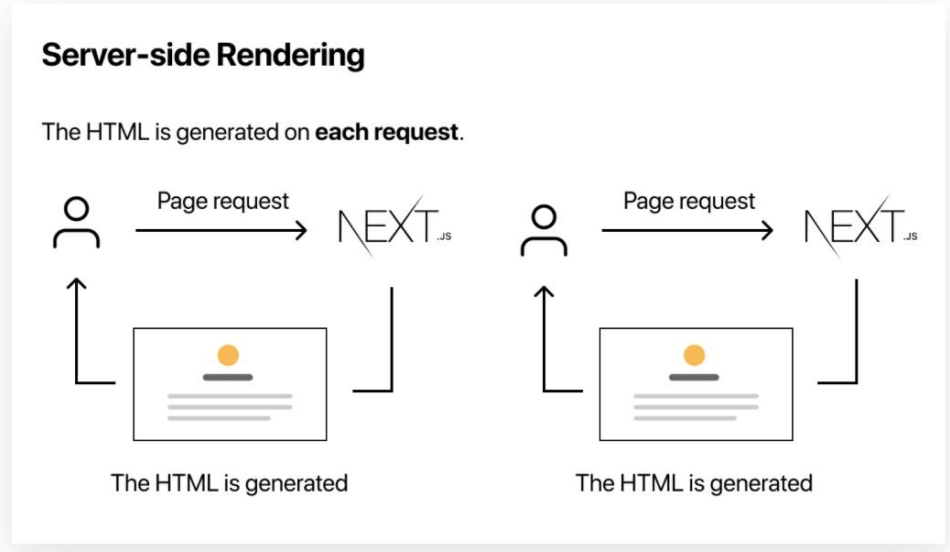


Рисунок 3.1 Рендеринг на стороні сервера у Next.js[3]

Для статичної генерації HTML створюється під час побудови проекту і буде повторно використаний при кожному запиті.

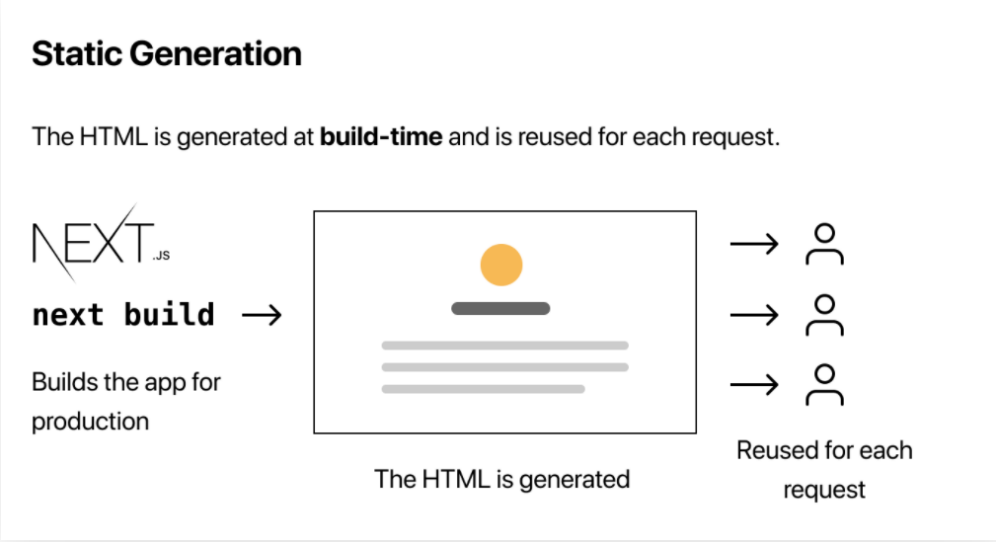


Рисунок 3.2 Статична генерація у Next.js[4]

Безкоштовний хостинг. Компанія Vercel надає послуги безкоштовного хостину для Next.js застосунків.

У проекті цей фреймворк використовується з причини покращення продуктивності сторінок та винесення більшості логіки зі сторони клієнта на сервер.

Material-UI – це бібліотека із графічними компонентами, створена для побудови швидких, гарних та доступних застосунків, що дотримуються рекомендацій material design від Google. Була використана у проекті, бо значно пришвидшує розробку багатофункціональних користувацьких інтерфейсів.

3.1.3 Огляд Firebase

Firebase – це платформа, розроблена Google для створення мобільних та веб застосунків.

Вона пропонує багато різних можливостей, таких як:

- Realtime Database;
- Cloud Functions;
- Authentication;
- і так далі.

У цьому проекті було використано їх рішення Authentication для менеджменту акаунтів користувачів.

3.1.4 Огляд Docker

Docker – це інструмент для управління ізольованими Linux-контейнерами. Використовується у проекті для налаштування мікросервісної архітектури, у якій кожний сервіс знаходиться у окремому контейнері.

3.2 Структура бази даних

3.2.1 Опис сутностей

`Class` – сутність, що описує модель класу в школі. Містить наступні властивості: ім’я, рік та список студентів.

`Grade` – сутність, що описує модель оцінки за предмет. Містить наступні властивості: оцінка, студент, якому вона була виставлена та колонка у журналі, у якій було виставлено оцінку.

`GradeSpace` – сутність, що представляє колонку у журналі за певний предмет. Містить наступні властивості: дата, назва колонки (наприклад “Домашнє завдання №1”), предмет, класс та тип оцінки (звичайна, семестрова або річна).

`Params` – сутнісь, що описує налаштування системи. Містить наступні властивості: поточний навчальний рік та список всіх навчальних років.

`Student` – сутність, що описує модель учня. Містить наступні властивості: ФІО, дата народження, номер телефону, пошта та номер особистої справи.

`Subject` – сутність, що описує модель шкільного предмету. Містить наступні властивості: назва предмету та клас, до якого він належить. Варто зазначити, що предмет “Історія” у 8 класі та “Історія” у 9 класі є різними об’єктами.

`User` – сутність, що описує модель користувача системи. Містить наступні властивості: ФІО, пошта, до якої прив’язан акаунт, параметр чи є

користувач адміністратором, список класів, до яких користувач має доступ та унікальний ідентифікатор у системі Firebase.

3.2.2 Використання MongoDB

MongoDB – це нереляційна база данних, яка зберігає дані у гнучких, JSON-подібних документах. Для C# існує бібліотека `MongoDB.Driver` для інтеграції з цією базою даних.

Бібліотеку можна додати за допомогою команди:

```
> dotnet add package MongoDB.Driver
```

Далі треба створити директорію `Models` та додати до неї конфігураційний файл `SchoolDatabaseSettings.cs`, де міститься клас с описами параметрів підключення (`ConnectionString`).

В цю ж директорію додаємо моделі сутностей, кожну - в свій файл. Кожній моделі відповідає клас у C# та колекція у MongoDB, яка створюється автоматично.

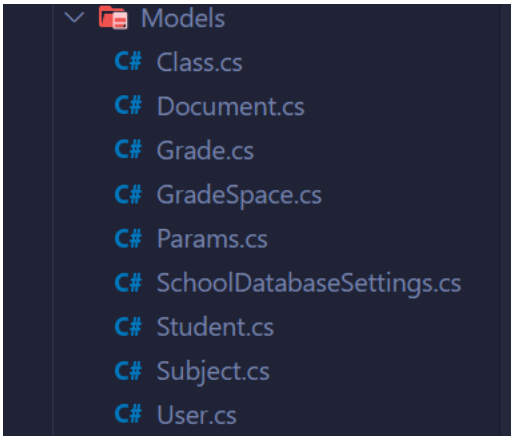


Рисунок 3.3 Структура директорії Models

Кожний клас сутності містить атрибут `BsonCollection` та наслідується від абстрактного класу `Document`, який містить властивість унікального ідентифікатора документа в коллекції. Атрибут `BsonCollection` вказує, як відповідна колекція буде називатись у базі даних. Наприклад, файл із сутністю учня має такий вигляд:

```
[BsonCollection("Students")]
public class Student : Document
{
    public string Name { get; set; }
    public string Surname { get; set; }
    public string Patronymic { get; set; }
    public DateTime Birthday { get; set; }
    public string Phone { get; set; }
    public string Email { get; set; }
    public string RegistryId { get; set; }
}
```

3.2.3 Використання патерну «Репозиторій»

Патерн «Репозиторій» є одним із найпопулярніших підходів до роботи із базою даних. По-перше, він надає можливість абстрагуватися від конкретної бази даних. По-друге, за допомогою цього патерну відбувається певне розділення обов’язків між класами, які працюють із даними, та всією іншою програмою. Це зменшує зв’язність компонентів застосунку.

Створюємо загальний клас Repository, який описує всі основновні дії із базою даних. Створений клас репозиторію має наступний вигляд:

```
public class BaseRepository<T> where T : Document
  {
    protected IMongoCollection<T> _collection;
    protected IMongoDatabase _database;

    public BaseRepository(ISchoolDatabaseSettings settings)
    {
      _database = new MongoClient(settings.ConnectionString).GetDatabase(settings.DatabaseName);
      _collection = _database.GetCollection<T>(GetCollectionName(typeof(T)));
    }

    public List<T> Get() =>
        _collection.Find(el => true).ToList();

    public void Add(T entity) =>
        _collection.InsertOne(entity);

    // і так далі
}
```

Більш високорівневу роботу із колекціями ми агрегуємо у сервіси (на приклад, «видалити предмет із всіма його оцінками» - метод сервісу `SubjectService`). Для кожної сутності створюємо сервіс, який буде агрегувати у собі клас репозиторію із відповідною моделлю. Таким чином класс `SubjectService` має доступ до коллекції з предметати через репозиторій `Repository<Subject>`. Також сервіси можуть агрегувати один одного, якщо під час виконання операцій із однією сутністю потрібен доступ до інших.

3.3 Використання технології GraphQL

GraphQL - це мова запитів для API та середовище для виконання цих запитів з наявними даними. GraphQL надає повний і зрозумілий опис даних у API, надає клієнтам можливість запитувати саме те, що їм потрібно, і нічого більше, полегшує розробку API з часом та забезпечує потужні інструменти для розробників. [5]

Основна задача розробника полягає у визначенні двох кореневих типів Query та Mutation. Для визначення типу необхідно вказати його поля та резолвери для цих полів. Резолвер - це функція, що повертає значення поля для певного об’єкту.

Запити, які отримують дані з серверу і не роблять ніяких side-ефектів, знаходяться у кореневому типі Query. Запити, які певним чином змінюють дані – додають, оновлюють, видаляють – знаходяться у кореневому типі Mutation.

3.3.1 Інтеграція з ASP.NET Core

Загалом на офіційному сайті GraphQL можна знайти декілька відкритих бібліотек для C#/.NET, а саме `graphql-dotnet`, `Hot Chocolate`, `graphql-net`, `Entity GraphQL` та `NGraphQL`. Для цієї роботи було обрано найпопулярнішу бібліотеку `graphql-dotnet` з вищенаведеного списку. Щоб встановити цю бібліотеку, необхідно виконати наступну команду:

```
> dotnet add package GraphQL
```

Далі необхідно обрати підхід для створення схеми. Загалом їх існує два: Schema First Approach та GraphType First Approach. При першому підході розробник спочатку визначає всю схему, а потім пише реалізацію до типів (резолвери). Другий підхід полягає у тому, що всі типи та резолвери до них визначаються у коді застосунку, а схема генерується динамічно. Бібліотека `graphql-dotnet` підтримує обидва підходи, але для проекту було використано підхід GraphType First.

Створюємо для кожної з сутностей свій GraphQL тип. Для цього необхідно наслідуватись від класу `ObjectGraphType`, який приймає в якості параметру модель з бази даних. Через те, що клас параметрозований, для кожного поля можна просто вказати його ім'я та лямбда функцію, що повертатиме відповідне Property з моделі, а резолвери створяться автоматично. Якщо тип поля не є стандартним в GraqhQL, то його треба явно вказати. Наприклад, клас, що описує тип студента має такий вигляд:

```
public class StudentType : ObjectGraphType<Student>
    {
        public StudentType(StudentService cs)
        {
            Field("Id", x => x.Id, type: typeof(NonNullGraphType<IdGraphType>));
            Field("Surname", x => x.Surname);
            Field("Name", x => x.Name);
            Field("Patronymic", x => x.Patronymic, nullable: true);
            Field("Birthday", x => x.Birthday, type: typeof(NonNullGraphType<DateGraphType>));
            Field("RegistryId", x => x.RegistryId);
            Field("Phone", x => x.Phone, nullable: true);
            Field("Email", x => x.Email, nullable: true);
        }
    }
```

Зробити поле обов'язковим можна двома способами. Для поля із стандартним типом треба передати у якості третього аргументу булеве значення nullable, для інших випадків необхідно вказувати тип як параметр классу `NonNullGraphType`.

Створивши всі типи сутностей, додаємо кореневий тип Query. В ньому створюємо поля, які відповідають майбутнім запитам у нашому застосунку. Прикладом може бути запит на отримання конкретного учня за його id:

```
public class Query : ObjectGraphType
{
  public Query(StudentService ss)
  {
    Name = "Query";

    Field<StudentType>(
    "student",
    arguments: new QueryArguments(
        new QueryArgument<NonNullGraphType<StringGraphType>> { Name = "id" }
    ),
    resolve: context =>
    {
      return ss.GetById(context.GetArgument<string>("id"));
    });
  }
}
```

Так як клас Query наслідується від `ObjectGraphType` без параметру, то резолвери треба створювати власноруч.

Для операцій зміни даних за аналогією додаємо тип Mutation та визначаємо всі необхідні поля. Коли класи Query та Mutation завершені, визначаємо головний тип Schema:

```
public class SchoolSchema : Schema
{
    public SchoolSchema(IDependencyResolver resolver): base(resolver)
    {
      Query = resolver.Resolve<Query>();
      Mutation = resolver.Resolve<Mutation>();
    }
}
```

Для експорту схеми до окремого файлу використовуємо утиліту `graphqurl`[6]. В результаті отримуємо файл с описами всіх типів, який можна знайти у додатку А.

Додаємо в проекті контролер із єдиним endpoint, який буде обробляти запити та передавати їх на виконання до схеми. Повний код із контролером можна переглянути в додатку Б.

Щоб код успішно скомпілювався, необхідно додати всі вручну визначені GraphQL типи (включаючи Schema, Query, Mutation) до методу `ConfigurateService` у файлі `Startup.cs`. Повний файл конфігурації `Startup.cs` можна переглянути в додатку В. Також в цьому ж файлі у методі `Configure` можна підключити сервіси GraphiQL та Graphql Playground для дебагу запитів і схеми.

Запускаємо проект, переходимо в браузері на http://localhost:5001/ui/graphiql і бачимо інтерфейс GraphiQL:

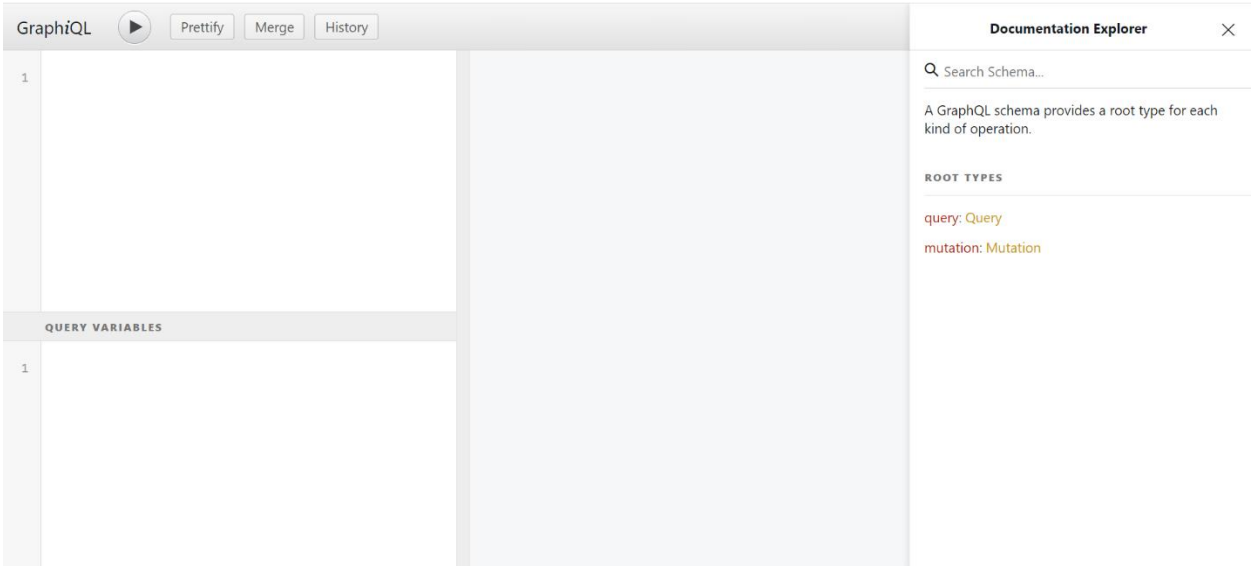


Рисунок 3.4 Інтерфейс сервісу GraphiQL

Цей інтерфейс зручний для дебагу, оскільки має підсвітку коду, перевірку запитів на правильність, автодоповнення, опис типів схеми та вікно для змінних. Спробуємо зробити тестовий запит:

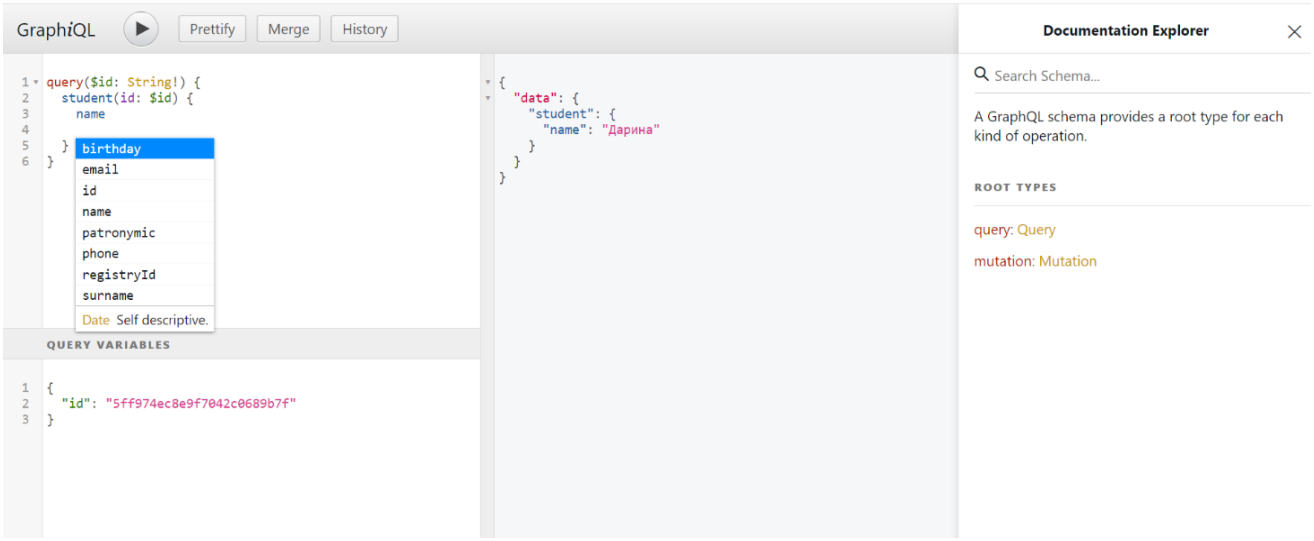


Рисунок 3.5 Можливості сервісу GraphiQL

Залишається проблема, що наразі сервер не перевіряє хто саме, викладач або адміністратор, робить запит. Тобто будь-який вчитель може зробити дії, на які не має прав. Вирішити цю проблему можна за допомогою пакету `GraphQL.Authorization`. Для цього необхідно завантажити його та зареєструвати права доступу для вчителя, адміністратора та спільні права доступу для обох ролей у методі `ConfigurateService` у файлі `Startup.cs` (додаток В):

```
services.AddSingleton<IAuthorizationEvaluator, AuthorizationEvaluator>();
services.AddTransient<IValidationRule, AuthorizationValidationRule>();

services.AddSingleton(s =>
{
    var authSettings = new AuthorizationSettings();

    authSettings.AddPolicy("AdminPolicy",
      _ => _.RequireClaim(ClaimTypes.Role, "Admin"));
    authSettings.AddPolicy("TeacherPolicy",
      _ => _.RequireClaim(ClaimTypes.Role, "Teacher"));
    authSettings.AddPolicy("AdminOrTeacherPolicy",
      _ => _.RequireClaim(ClaimTypes.Role, new[] { "Teacher", "Admin" }));

    return authSettings;
});
```

Далі створюємо класс `GraphQLContext`, який надає об'єкт користувача `ClaimsPrincipal` з правами доступу, які надалі GraphQL буде перевіряти:

```
public class GraphQLContext : Dictionary<string, object>, IProvideClaimsPrincipal
{
  public ClaimsPrincipal User { get; set; }
  public User UserDb { get; set; }
}
```

У сервісі Next.js додаємо до усіх запитів заголовок `X-Firebase-Uid`. У класі `GraphQLContoller` (додаток Б) під час обробки запитів зчитуємо цей заголовок та отримуємо з бази даних користувача за його ідентифікатором. Якщо такий користувач існує, тоді створюємо экземпляр класу `GraphQLContext` та визначаємо права доступу в ньому. Контекст передаємо до методу виконання схеми разом із Query та Variables.

Для обмеження доступу до полів необхідно викликати розширений метод `AuthorizeWith`, який у якості аргументу приймає роль користувача із доступом до певного поля. Наприклад, надамо доступ до отримання всіх користувачів системи тільки адміністраторам. Для цього треба для поля users у типі Query викликати метод `AuthorizeWith` наступним чином:

```
Field<NonNullGraphType<ListGraphType<NonNullGraphType<UserType>>>>(
 "users",
 resolve: context =>
 {
     return us.Get();
 }).AuthorizeWith("AdminPolicy");
```

3.3.2 Переваги та недоліки

Переваги:

1. *Отримання усіх необхідних даних за допомогою одного запиту*. В той час працюючи із REST, програмісту необхідно робити декілька запитів, щоб зібрати усю необхідну інформацію докупи.

2. *Чітка структура запитів.* При використанні GraphQL можна чітко задавати потрібну модель даних. При цьому немає потреби створювати нові запити у ситуаціях, коли у моделі старих запитів не вистачає певних полів, як це буває із REST.

3. *Автоматично згенерована документація.* Підключивши сервіс GraphiQL, програміст отримує зручний інтерфейс для перегляду схеми. Цей сервіс надає необхідну інформацію по типам у зручному вигляді та слугує гарним джерелом для документації.

4. *Можливість створення схеми двома способами.* Бібліотека `graphql-dotnet` пропонує підтримку як Schema First, так і Code First підходів.

Недоліки:

1. *Відсутність кешування*. Маючи багато endpoints із точною структурою моделей, REST API надає можливість кешування. У GraphQL кешування обмежене через те, що всі типи запитів робляться на один endpoint.

2. *Проблема N+1 запитів.* Через те, що у GraphQL для кожного поля існує свій резолвер, виникає проблема зайвих запитів до бази даних.

3. *Недопрацьована бібліотека у ASP.NET Core*. У бібліотеці `graphql-dotnet` треба додавати всі створені типи до файлу Startup.cs. Це виглядає неохайно і додає багато коду до файлу. Також важко знайти працюючі приклади на просторах Інтернет, хоча ця бібліотека є найпопулярнішою для C#.

3.4 Проектування архітектури

У проекті використовується архітектура із трьома сервісами.

Перший сервіс – це сервер, написанний на ASP.NET Core у поєднанні із GraphQL підходом та базою даних MongoDB. Його головною метою є робота із даними. Другий сервіс – це сервер, який відповідає за аутентифікацію. Він написаний на Node.js та Express. Третій сервіс – це сервер, написанний на Next.js, який генерує сторінки на стороні серверу та повертає їх браузеру.

Розглянемо декілька прикладів взаємодії сервісів:

1. Користувач намагається залогіниться.

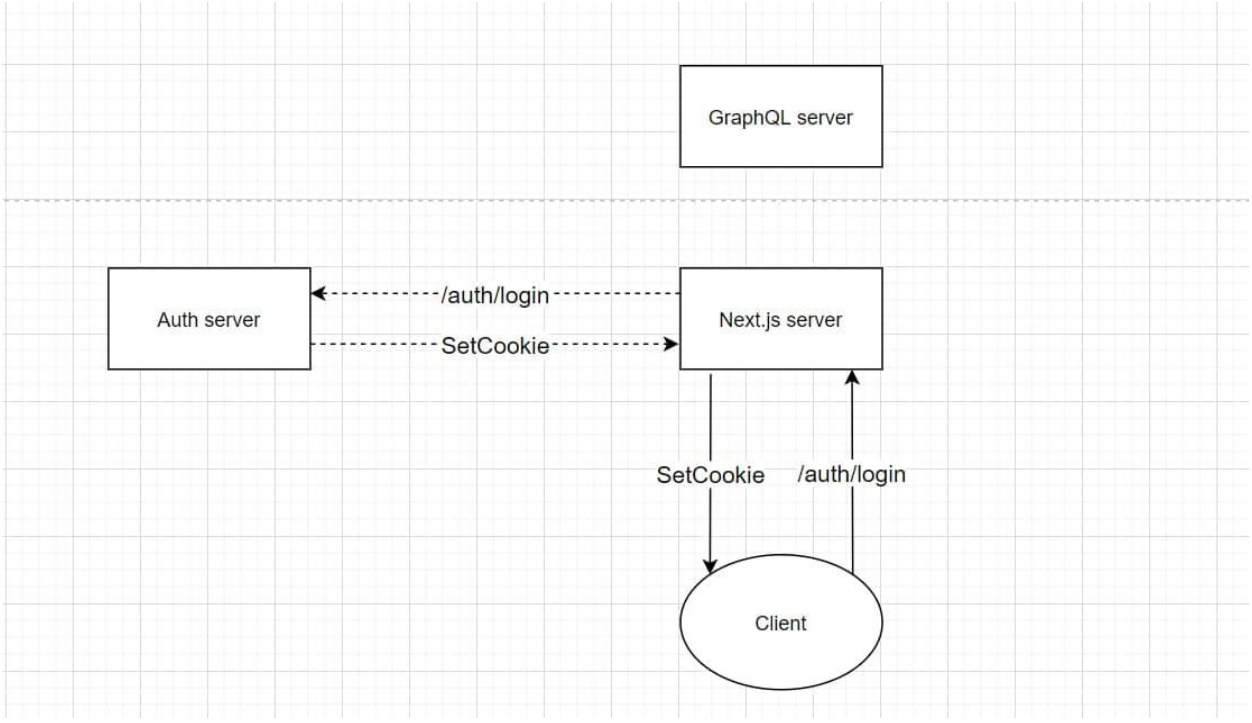


Рисунок 3.6 Взаємодія сервісів при логіні користувача

Спочатку клієнт з даними користувача робить запит на Firebase і перевіряє чи існує такий користувач. Якщо такий користувач існує, то клієнт

робить запит на Next.js сервер із id токеном користувача. Next.js сервер робить запит на Auth сервер, який перевіряє, чи id токен валідний, і створює сесію, яка надалі зберігається у cookie.

2. Адміністратор переходить на сторінку із користувачами.

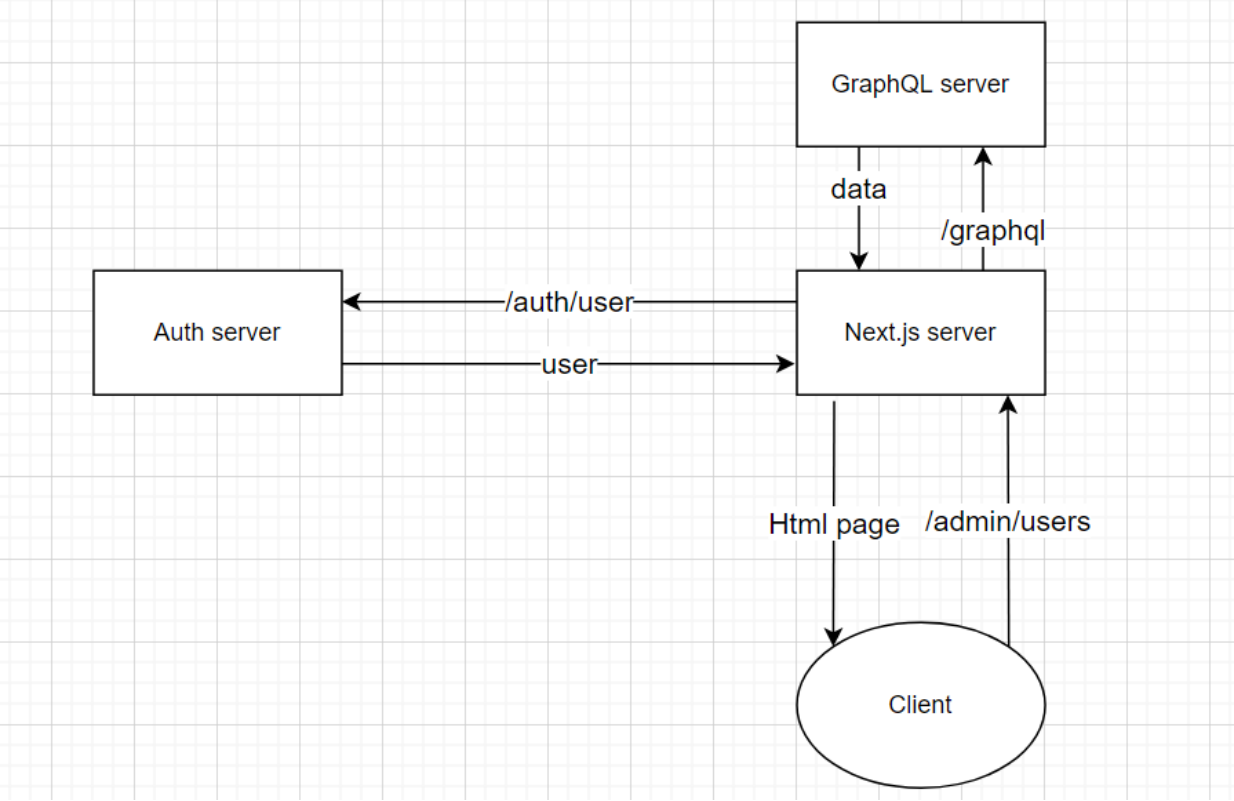


Рисунок 3.7 Взаємодія сервісів при переході на сторінку із користувачами

Перед тим як отримати дані, Next.js сервер робить запит на Auth сервер із cookie сесії, щоб перевірити її валідність. Якщо сесія валідна, то Next.js сервер робить запит на GrapqQL сервер, щоб отримати дані про користувачів. Далі Next.js на сервері формує сторінку та повертає її клієнту.

3. Адміністратор додає нових користувачів.

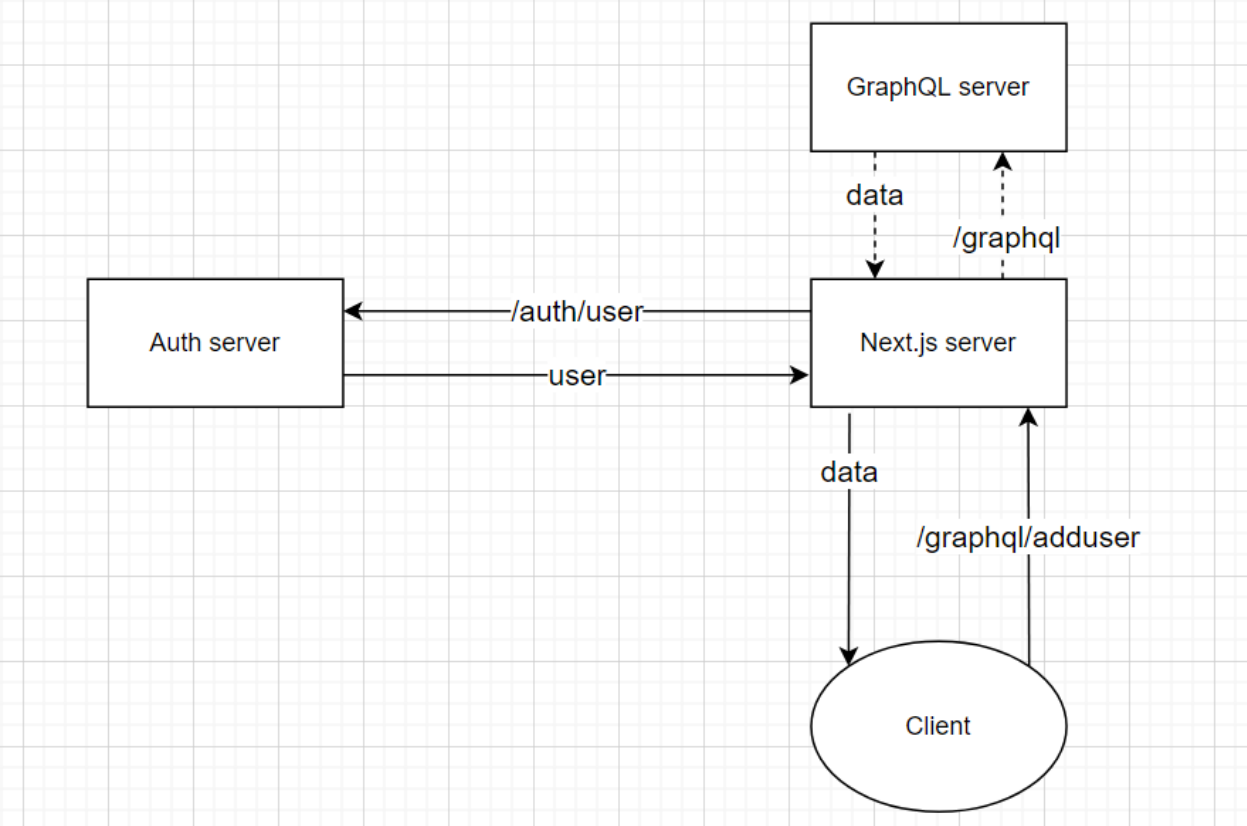


Рисунок 3.8 Взаємодія сервісів при додаванні нового користувача

Клієнт робить запит на додавання нового користувача на Next.js сервер. Next.js сервер робить запит на Auth сервер із cookie сесії, щоб перевірити її валідність. Якщо сесія валідна, Next.js сервер проксіює запит на GrapqQL сервер, щоб додати нового користувача до бази данних та повертає результат клієнту.

Висновки

У роботі було детально досліджено використання технології GraphQL у контексті фреймворку ASP.NET Core. В результаті дослідження можна зробити висновок, що реалізація цієї технології для фреймворку ще не є повноцінною та потребує доопрацювань.

Загалом було знайдено декілька бібліотек для використання GraphQL у C#. Для написання проекту було обрано найпопулярнішу з них - `graphql-dotnet`.

Перевагами обраної бібліотеки можна вважати наявність вбудованого інтерфейсу для дебагу GraphiQL та підтримку як Schema First, так і Code First підходів.

Із недоліків бібліотеки варто зазначити погану документацію та непрацюючі приклади в ній. Для проектів із великою кількістю сутностей треба виконувати довгу і монотонну роботу, створюючи для кожної сутності окремий GraphQL тип. Окрім цього, кожний створений власноруч тип необхідно додавати до файлу Startup.cs. Цей підхід дещо позбавляє гнучкості у розробці – однієї з основних переваг GraphQL.

Отже, зважаючи на всі недоліки бібліотеки `graphql-dotnet` та складнощі роботи з нею, можна зробити висновок, що наразі її використання для написання GraphQL серверу на основі фреймворку ASP.NET Core може бути не надто вдалим рішенням для розробки великих систем.

Список використаної літератури

1. Система Управління Школою та Навчанням - LMS – Адміністрація[Електронний ресурс] – Режим доступу до ресурсу: https://smls.com.ua/site/administration/

2. React – JavaScript-бібліотека для створення користувацьких інтерфейсів[Електронний ресурс] – Режим доступу до ресурсу: https://uk.reactjs.org/

3. Basic Features: Pages | Next.js[Електронний ресурс] – Режим доступу до ресурсу: https://nextjs.org/docs/tag/v9.2.2/basic-features/pages#server-side-rendering

4. Basic Features: Pages | Next.js[Електронний ресурс] – Режим доступу до ресурсу: https://nextjs.org/docs/tag/v9.2.2/basic-features/pages#static-generation

5. GraphQL | A query language for your API[Електронний ресурс] – Режим доступу до ресурсу: https://graphql.org/

6. Exporting the Hasura GraphQL schema | Hasura GraphQL Docs[Електронний ресурс] – Режим доступу до ресурсу: https://hasura.io/docs/latest/graphql/core/guides/export-graphql-schema.html#using-graphqurl

ДОДАТКИ

Додаток А

(Обов’язковий)

Файл schema.graphql зі схемою графу застосунку

```
input ClassInput {
  name: String!
  year: String!
}

input ClassSubjectsInput {
  classId: String!
  subjectAccess: [String!]!
}

type ClassSubjectsType {
  class: ClassType!
  subjectAccess: [SubjectType!]!
}

type ClassType {
  id: ID!
  name: String!
  students: [StudentType!]!
  subjects: [SubjectType!]!
  year: String!
}

input GradeInput {
  id: String = null
  mark: Int!
  student: String = null
  gradeSpace: String = null
}

input GradeSpaceInput {
  id: String = null
  name: String!
  date: Date!
  subject: String = null
  class: String = null
  type: String!
}

type GradeSpaceType {
  date: Date!
  id: ID!
  name: String!
  subject: SubjectType!
  type: String!
}
```

```
type Mutation {
  addStudentsToClass(classId: String!, students: [String!] = null): ClassType!
  createClass(data: ClassInput!): ClassType!
  createGrades(data: [GradeInput] = null): [GradeType!]!
  createGradeSpace(data: GradeSpaceInput!): GradeSpaceType!
  createStudent(data: StudentInput!): StudentType!
  createSubject(data: SubjectInput!): SubjectType!
  createUser(data: UserInput!): UserType!
  deleteClass(id: String!): Boolean!
  deleteGradeSpace(id: String!): Boolean!
  deleteStudent(id: String!): Boolean!
  deleteSubject(id: String!): Boolean!
  deleteUser(id: String!): Boolean!
  editClass(data: ClassInput!, id: String!): ClassType!
  editGrades(data: [GradeInput] = null): [GradeType!]!
  editGradeSpace(data: GradeSpaceInput!): GradeSpaceType!
  editStudent(data: StudentInput!, id: String!): StudentType!
  editSubject(data: SubjectInput!, id: String!): SubjectType!
  editUser(data: UserInput!, id: String!): UserType!
  removeGrades(data: [String] = null): Boolean!
  removeStudentFromClass(classId: String!, studentId: String!): Boolean!
}

type ParamsType {
  currentYear: String!
  id: ID!
  years: [String!]!
}

type Query {
  class(id: String!): ClassType
  classes: [ClassType!]!
  grades(id: String = null): [GradeType!]!
  params: ParamsType
  student(id: String!): StudentType
  studentGrades(studentId: String = null, classId: String = null): [GradeType!]!
  students(surname: String = null): [StudentType!]!
  subject(id: String!): SubjectType
  subjects(classId: String!): [SubjectType!]!
  teachers: [UserType!]!
  user(id: String!): UserType
  userByUid(uid: String!): UserType
  users: [UserType!]!
}


input StudentInput {
  name: String!
  surname: String!
  patronymic: String = null
  birthday: Date!
  email: String = null
  phone: String = null
  registryId: String!
}
```

```
type StudentType {
  birthday: Date!
  email: String
  id: ID!
  name: String!
  patronymic: String
  phone: String
  registryId: String!
  surname: String!
}

input SubjectInput {
  name: String!
  class: String!
}

type SubjectType {
  class: ClassType!
  gradeSpaces: [GradeSpaceType!]!
  id: ID!
  name: String!
}

input UserInput {
  name: String!
  isAdmin: Boolean!
  classAccess: [ClassSubjectsInput] = null
  email: String!
  uid: String!
}

type UserType {
  classAccess: [ClassSubjectsType!]!
  email: String!
  id: ID!
  isAdmin: Boolean!
  name: String!
  uid: String!
}
```

Додаток Б

(Обов’язковий)

Код файлу GraphqlController.cs

```
namespace School.Controllers
{
    public class GraphQLQuery
    {
        public string OperationName { get; set; }
        public string Query { get; set; }
        public JObject Variables { get; set; }
    }

    [ApiController]
    [Route("[controller]")]
    public class GraphQLController : ControllerBase
    {
        private SchoolSchema _schema;
        private UserService _us;

        public GraphQLController(SchoolSchema schema, UserService us)
        {
            _schema = schema;
            _us = us;
        }

        [HttpPost]
        public async Task<IActionResult> PostAsync([FromBody] GraphQLQuery query,
          [FromServices] IEnumerable<IValidationRule> validationRules)
        {
            var user = Request.Headers["X-Firebase-Uid"].Count != 0 ?
                                        _us.GetByUid(Request.Headers["X-Firebase-Uid"]) : null;

            var claims = new List<Claim>();

            if(user != null) {
                claims.Add(new Claim(ClaimTypes.Role, user.IsAdmin ? "Admin" : "Teacher"));
            }

            var json = await _schema.ExecuteAsync(_ =>
            {
                _.ValidationRules = validationRules;
                _.Query = query.Query;
                _.Inputs = query.Variables.ToInputs();
                _.UserContext = new GraphQLContext
                {
                  User = new ClaimsPrincipal(new ClaimsIdentity(claims)),
                  UserDb = user
                };
            });

            return new JsonResult(JsonConvert.DeserializeObject(json));
        }
    }
}
```

Додаток В

(Обов'язковий)

Код файлу Startup.cs

```
namespace School
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddSingleton<IDependencyResolver>(
                s => new FuncDependencyResolver(s.GetRequiredService));

            services.Configure<SchoolDatabaseSettings>(
                Configuration.GetSection(nameof(SchoolDatabaseSettings)));

            services.AddSingleton<ISchoolDatabaseSettings>(sp =>
                sp.GetRequiredService<IOptions<SchoolDatabaseSettings>>().Value);

            services.AddSingleton<IAuthorizationEvaluator, AuthorizationEvaluator>();
            services.AddTransient<IValidationRule, AuthorizationValidationRule>();

            services.AddSingleton(s =>
            {
                var authSettings = new AuthorizationSettings();

                authSettings.AddPolicy("AdminPolicy",
                  _ => _.RequireClaim(ClaimTypes.Role, "Admin"));
                authSettings.AddPolicy("TeacherPolicy",
                  _ => _.RequireClaim(ClaimTypes.Role, "Teacher"));
                authSettings.AddPolicy("AdminOrTeacherPolicy",
                  _ => _.RequireClaim(ClaimTypes.Role, new[] { "Teacher", "Admin" }));

                return authSettings;
            });

            services.AddSingleton<BaseRepository<User>>();
            services.AddSingleton<BaseRepository<Class>>();
            services.AddSingleton<BaseRepository<Student>>();
            services.AddSingleton<BaseRepository<Subject>>();
            services.AddSingleton<BaseRepository<Grade>>();
            services.AddSingleton<BaseRepository<GradeSpace>>();
            services.AddSingleton<BaseRepository<Params>>();
            services.AddSingleton<ParamsService>();
            services.AddSingleton<UserService>();
            services.AddSingleton<ClassService>();
            services.AddSingleton<StudentService>();
            services.AddSingleton<GradeSpaceService>();
            services.AddSingleton<GradeService>();
```

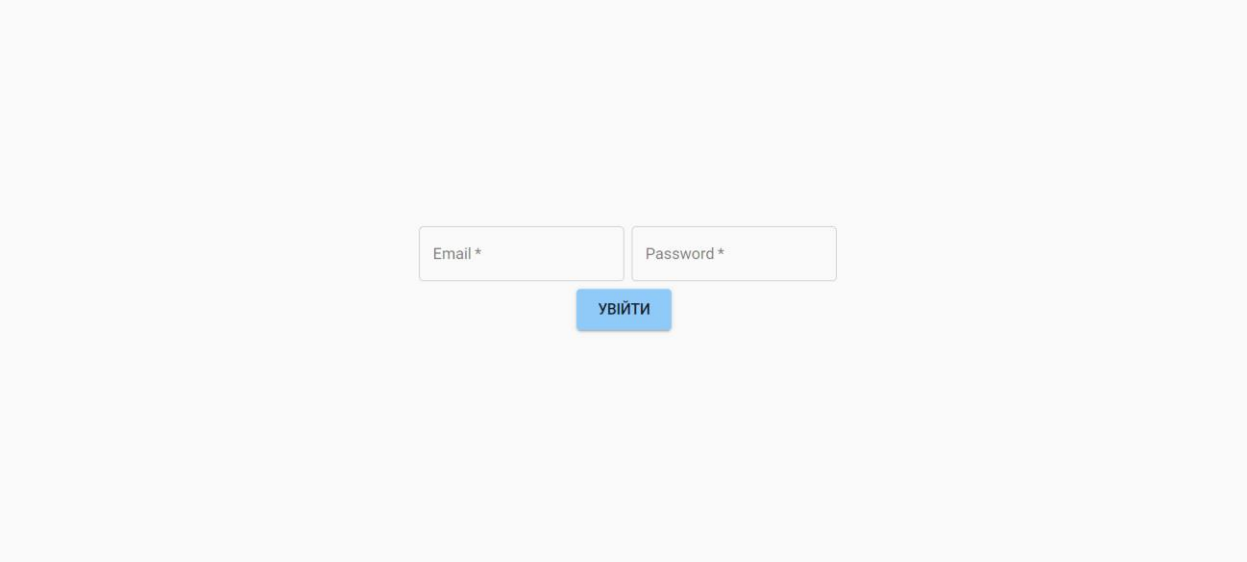
```
            services.AddSingleton<SubjectService>();
            services.AddSingleton<ParamsService>();
            services.AddSingleton<UserType>();
            services.AddSingleton<ClassType>();
            services.AddSingleton<ClassSubjectsType>();
            services.AddSingleton<StudentType>();
            services.AddSingleton<ParamsType>();
            services.AddSingleton<SubjectType>();
            services.AddSingleton<GradeSpaceType>();
            services.AddSingleton<GradeType>();
            services.AddSingleton<ClassSubjectsInputType>();
            services.AddSingleton<UserInputType>();
            services.AddSingleton<ClassInputType>();
            services.AddSingleton<StudentInputType>();
            services.AddSingleton<GradeSpaceInputType>();
            services.AddSingleton<GradeInputType>();
            services.AddSingleton<SubjectInputType>();
            services.AddSingleton<Query>();
            services.AddSingleton<Mutation>();
            services.AddSingleton<SchoolSchema>();
            services.AddControllers().AddNewtonsoftJson();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseHttpsRedirection();
            }

            app
            .UseCors("MyPolicy")
            .UseWebSockets()
            .UseGraphQLVoyager(new GraphQLVoyagerOptions() {})
            .UseGraphiQLServer()
            .UseGraphQLPlayground("/");

            app.UseRouting();
            app.UseAuthorization();

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

Додаток Г

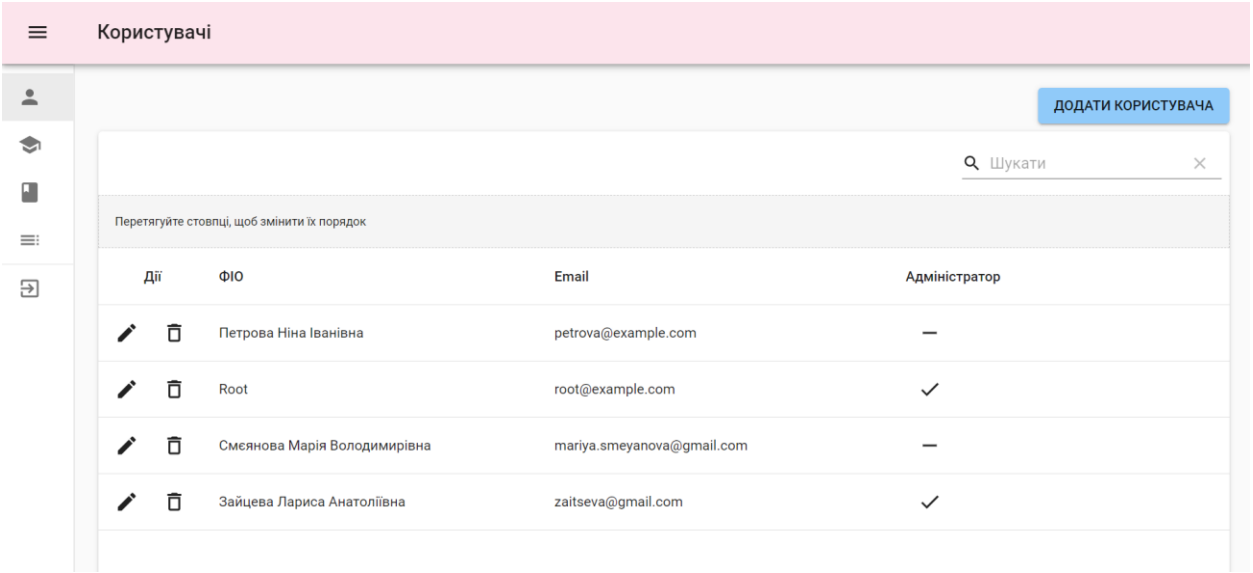
(Обов’язковий)

Скріншоти інтерфейсу застосунку для адміністратора

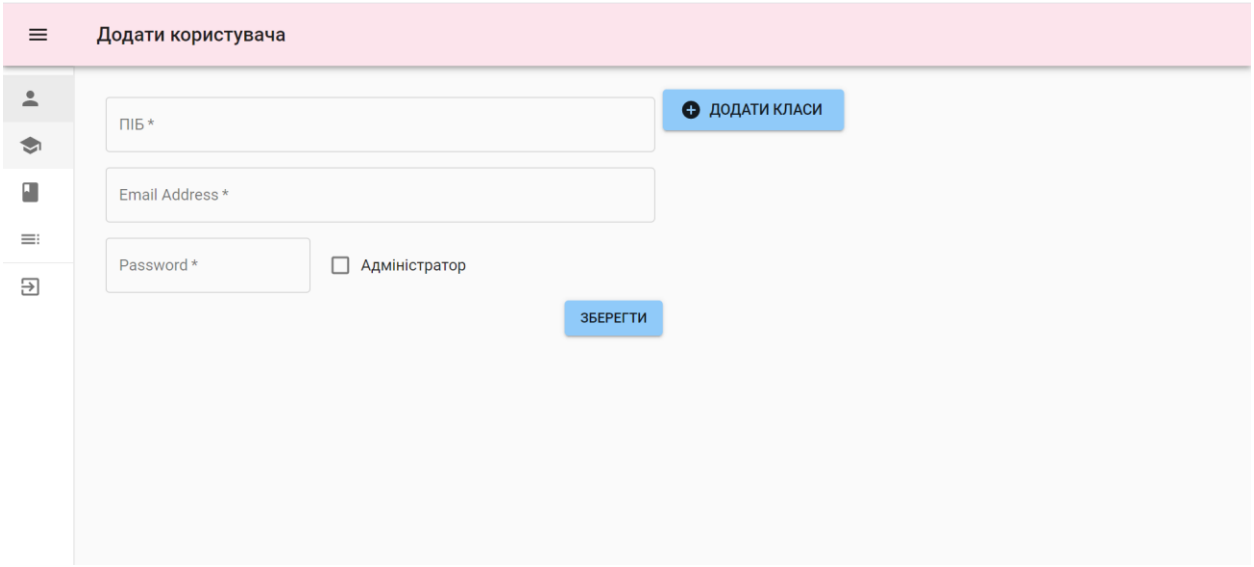
Сторінка входу



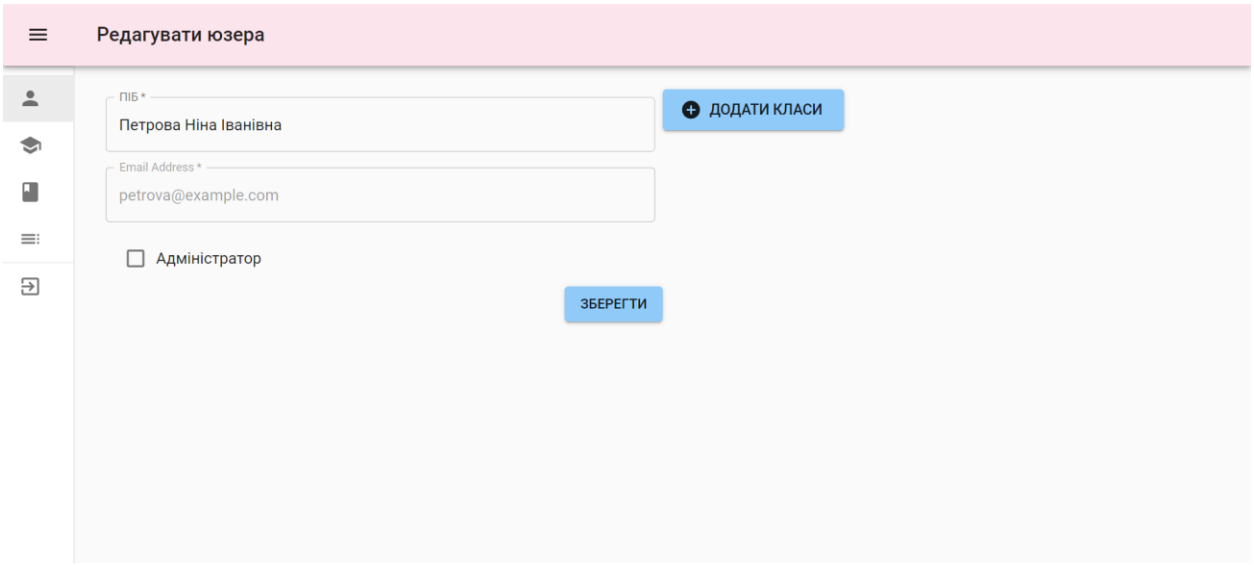
Сторінка з користувачами системи



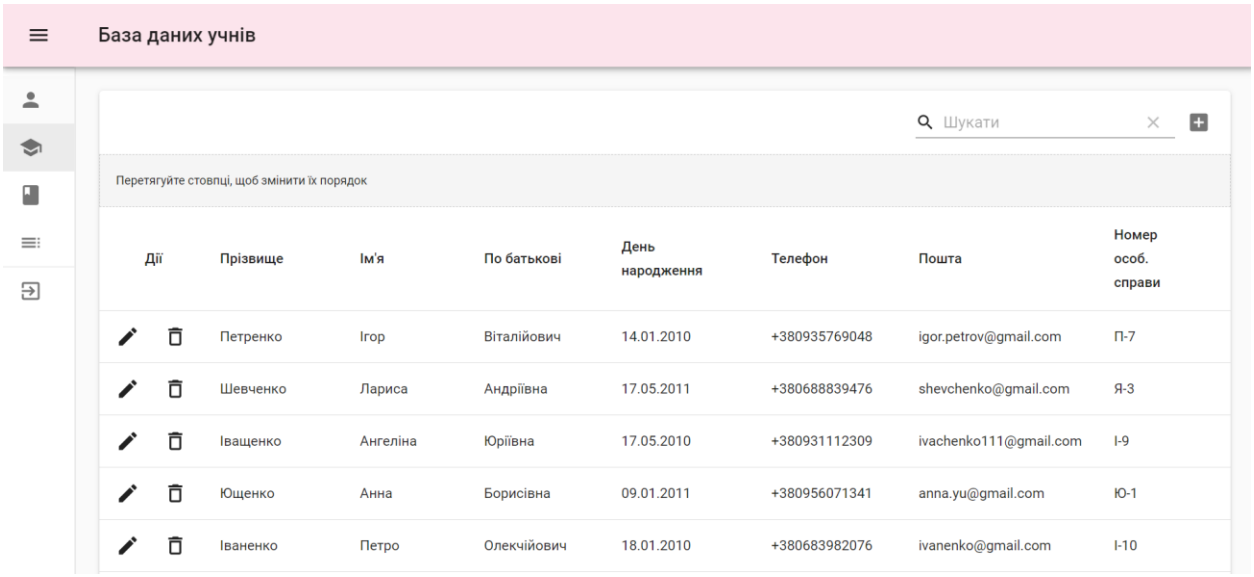
Сторінка додавання нового користувача



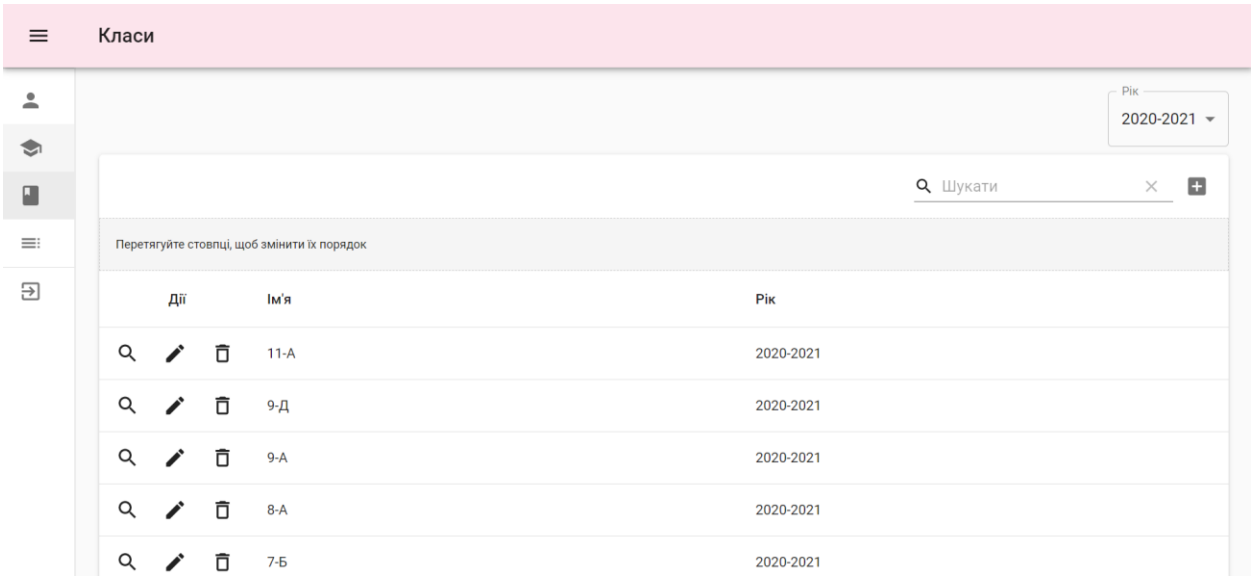
Сторінка редагування користувача



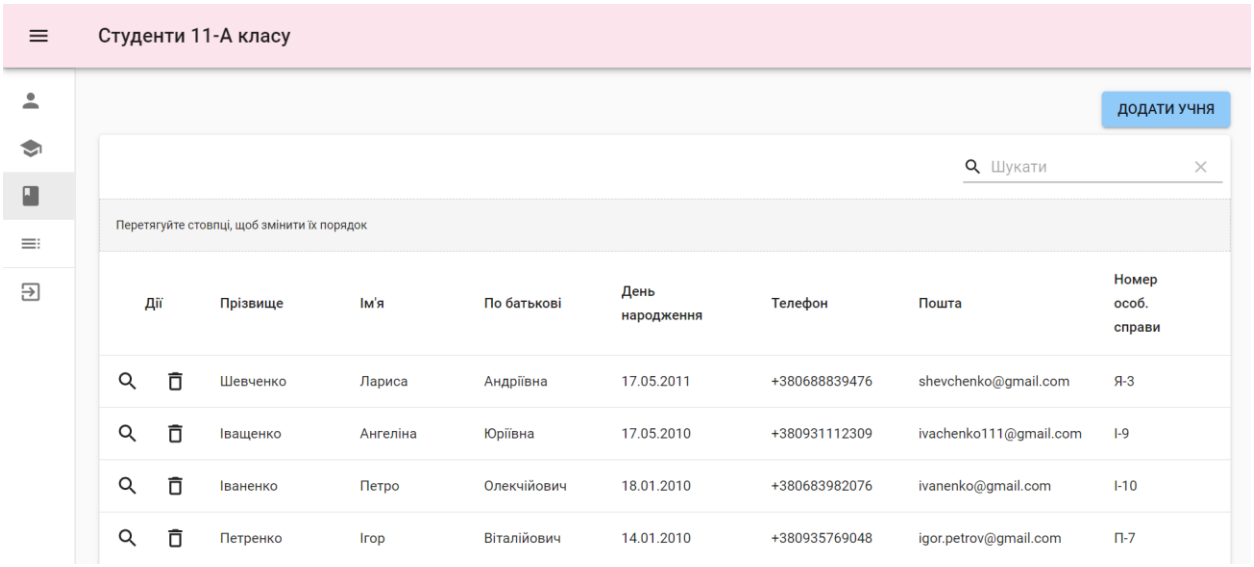
Сторінка із загальною базою учнів



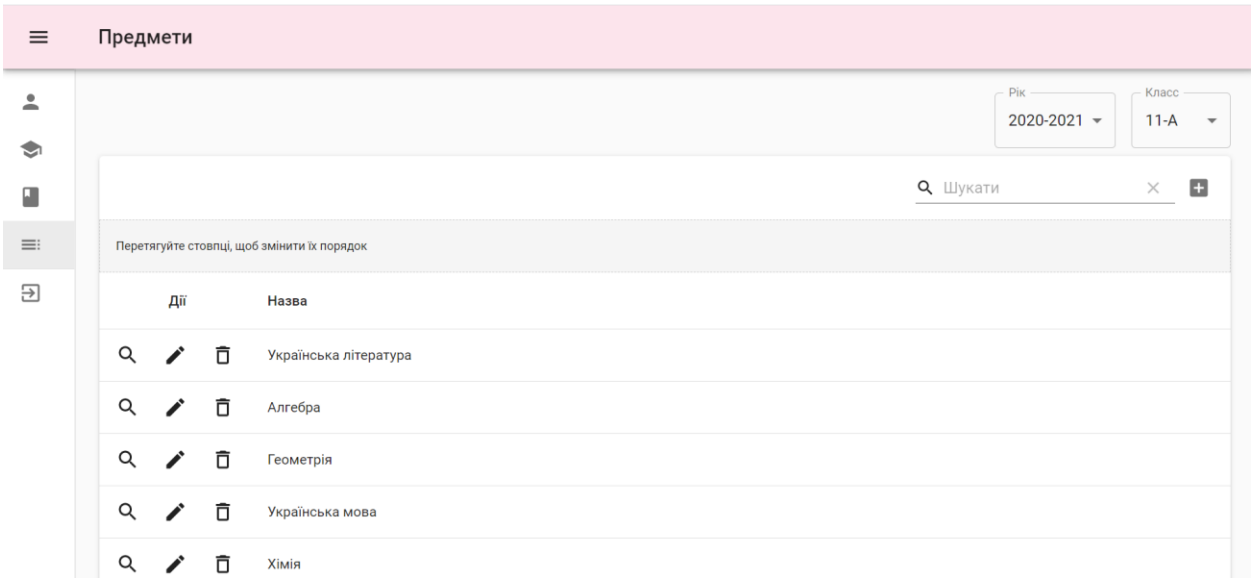
Сторінка з доступними класами



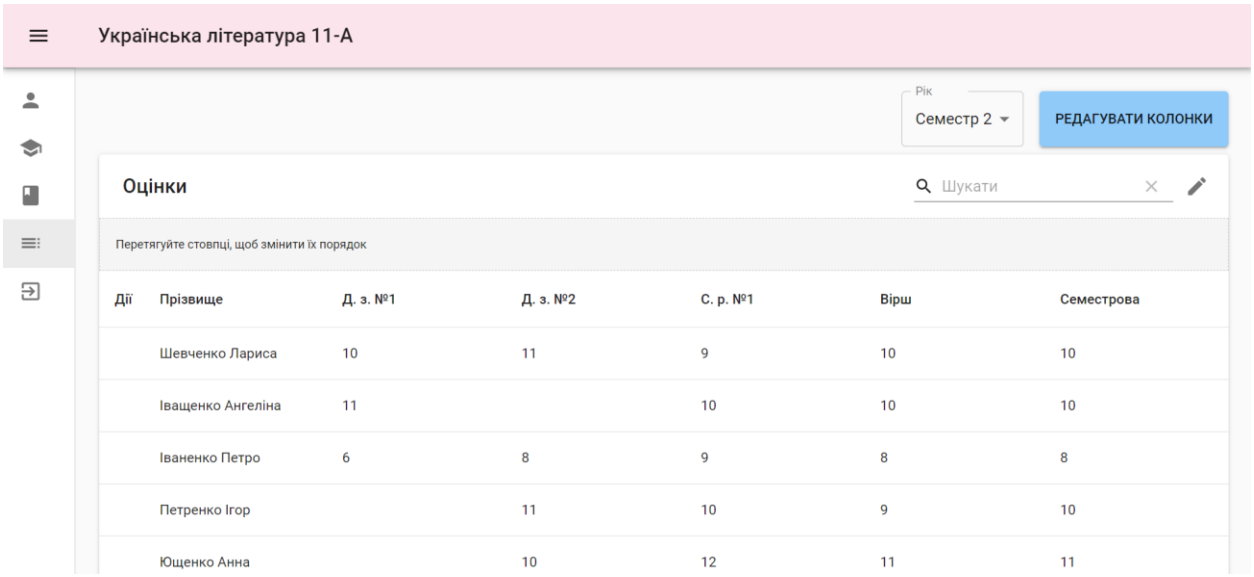
Сторінка із студентами певного класу

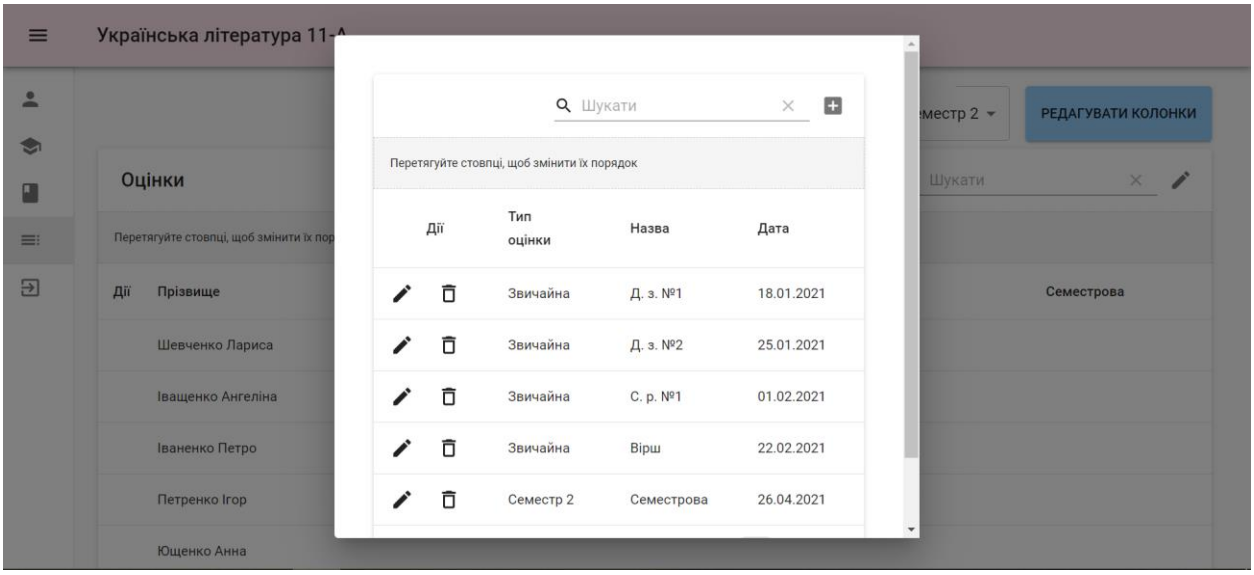


Сторінка із предметами певного класу

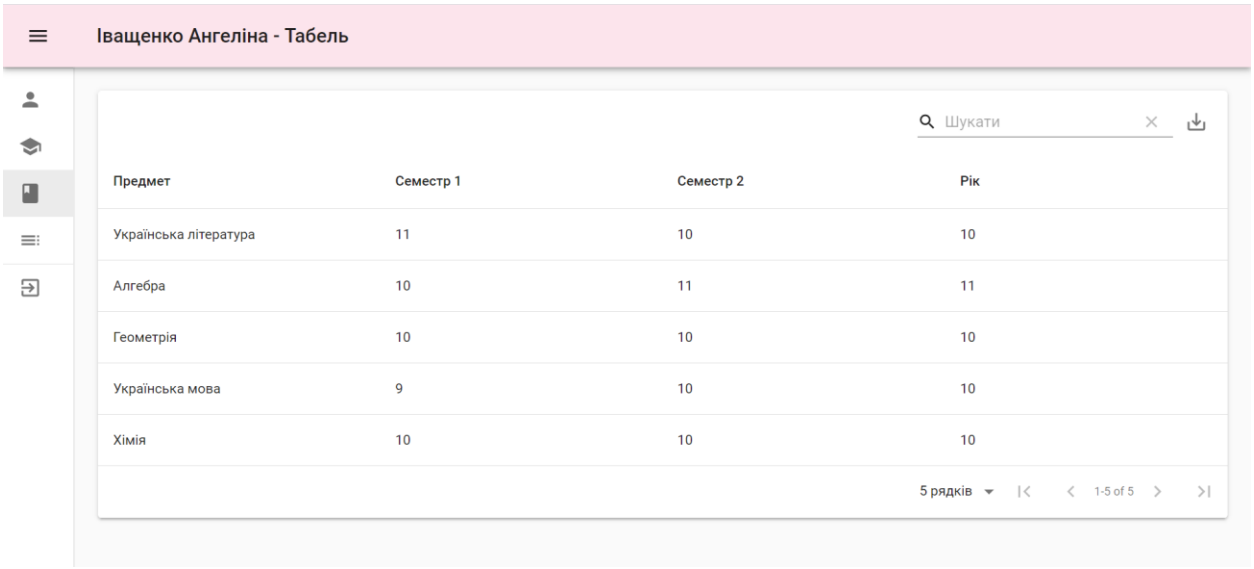


Сторінки з класним журналом





Сторінка з табелем учня



Додаток Д

(Обов’язковий)

Посилання на сайт

https://school.dsigma.me/

Доступ до акаунту вчителя

Email: teacher@example.com

Пароль: 000000