КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра теорії та технології програмування

Кваліфікаційна робота

на здобуття ступеня бакалавра

за спеціальністю 122 Комп'ютерні науки

на тему:

Розробка програми для управління лабораторними роботами. Розробка аналізатора коду на унікальність

Виконав студент 4-го курсу
Кунцьо Дмитро Юрійович _____

Науковий керівник:
асистент
Федорус Олексій Мстиславович _____

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____

Роботу розглянуто й допущено до захисту
на засіданні кафедри теорії та
технології програмування
«21» травня 2021 р.,

протокол № ___
Завідувач кафедри
М. С. Нікітченко _____

Київ – 2021

РЕФЕРАТ

Обсяг роботи 47 сторінок, 31 ілюстрації, 3 таблиці, 9 джерел посилань.

ПЛАГІАТ, ВИХІДНИЙ КОД, ASP.NET CORE, ВЕБ РОЗРОБКА, «ЧИСТА» АРХІТЕКТУРА, AWS, DOCKER.

Об’єктом роботи є розробка серверної частини веб додатку для управління лабораторними роботами.

Предметом роботи є програмний продукт для дослідження програмного коду на наявність дублів.

Метою роботи є розробка веб додатку для управління лабораторними роботами з можливістю перевірки останніх на плагіат, що даватиме змогу правильно оцінювати роботи.

В роботі досліджено алгоритми на пошук плагіату в коді, а також проаналізовано «чисту» архітектуру програмних додатків.

Інструменти розробки: Visual Studio 2019 Community, Visual Studio Code, мова програмування C#, Docker, Amazon Web Services.

Були отримані наступні **результати**: виконано загальний огляд проектування архітектури програмних додатків. Було досліджено та застосовано кросплатформенну технологію ASP.NET Core. Розроблено додаток для перевірки програмного коду на плагіат.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

API - Application Programming Interface, прикладний програмний інтерфейс.

LINQ - Language Integrated Query, запити, інтегровані в мову.

IDE - Integrated Development Environment, інтегроване середовище розробки.

REST - Representational State Transfer.

EF – Entity Framework.

AWS – Amazon Web Services.

CRUD - create read update delete, створення, читання, оновлення і вилучення.

JSON - JavaScript Object Notation.

YAML - Another Markup Language.

HTTP - HyperText Transfer Protocol.

HTTPS – HTTP Secure.

SSL - Secure Sockets Layer.

EC2 - Amazon Elastic Compute Cloud.

IP - Internet Protocol.

ВСТУП

Оцінка сучасного стану об’єкта розробки. Сьогодні з кожним днем зростає з експоненціальною швидкістю кількість знань та даних в мережі Інтернет. Кожен має необмежений доступ до цих даних, тому значимість інтелектуальної власності сильно зростає. Адже дублювання всієї інформації відбувається кожної секунди.

Особливо важливою є сфера навчання, в якій потрібно правильно оцінювати результати роботи, тих хто навчається. Для правильної оцінки потрібне розуміння, чи людина виконала завдання сама, чи просто знайшла готовий розв'язок. На жаль, таких систем сьогодні вкрай мало.

Актуальність роботи та підстави для її виконання. Тема є надзвичайно актуальною, оскільки весь світ перейшов в віддалений режим роботи, а також в цей режим перейшло і навчання. Для викладачів потрібно знайти рішення, що буде зручним для проведення всіх курсів та для виставлення оцінок, для студентів – єдина платформа де вони будуть навчатись. Оцінювання повинно проводитись з врахуванням унікальності написаної студентом роботи, адже так у викладача буде розуміння, на скільки студент розібрався з темою.

Мета й завдання роботи. Метою кваліфікаційної роботи є створення програмного продукту для викладачів та студентів, щоб зручно оцінювати студентів, на основі перевірки роботи студента на плагіат. Для досягнення цієї мети поставленні такі завдання:

- Дослідити існуючі мобільні додатки для виявлення плагіату в програмному коді на ринку.
- Розробити технічне завдання до продукту.
- Розробити легко масштабований додаток, а саме його серверну частину.

- Розробити програмне забезпечення для перевірки програмного коду на плагіат.

Об’єктом даної роботи є створення веб додатку для навчальних курсів.

Предметом дослідження є створення засобу для перевірки програмного коду на плагіат.

В якості **інструменту створення** програмного засобу було обрано Visual Studio 2019 Community — інтегроване середовище розробки мовою програмування C#, яке є безкоштовним і вільно поширюваним від корпорації Microsoft.

C# — мова, яка є об’єктно-орієнтованою, є ідеальним вибором для розробки веб додатків.

Можливі сфери застосування. Програмний продукт даної роботи може бути застосованим насамперед різними учбовими закладами, такими як школи чи університети, а також може використовуватись як платформа для онлайн курсів.

РОЗДІЛ 1 ТЕОРЕТИЧНА ЧАСТИНА

1.1 Перевірка програмного коду на наявність плагіату

1.1.1 Типи плагіату

Дублювання програмного можна розділити на чотири типи. Розглянемо по черзі кожен з них [1].

Перший тип збігів можна уявити як повністю скопійовану ділянку коду і, по факту, є повною копією оригіналу. Тим не менше можуть бути деякі варіації в пробільних символах (прогалини, переноси рядків, символи табуляції тощо), коментарях та розмітці. Програми типу 1 широко відомі як точні клони. Розглянемо наступний фрагмент коду, представлений на рисунку 1.

```
if (a - b == 0) {
    c = d;
    a = b + d;
} else
    c = b; // Comment
```

Рисунок 1

І порівняємо його з копією, представленою на рисунку 2.

```
if (a-b==0) {
          c=d;
    a=b + d;
} else // new comment
c = b;
```

Рисунок 2

Ми бачимо, що ці два фрагмента є текстуально схожими після видалення пробілів і коментарів. Проте, навіть після видалення коментарів і прогалин, наступний фрагмент коду, зображений на рисунку 3, несхожий на два попередніх при

порядковому порівнянні починаючи з символу "{" і закінчуючи "}". Але при цьому, цей фрагмент також відноситься до першого типу і являється точною копією двох інших. Типові методи, що зрівнюють код по рядкам не виявлять в такому випадку дублювання коду.

```
if (a-b==0)
{
    c=d;
    a=b + d;
} else // new comment
c = b;
```

Рисунок 3

Другий тип дублювання являє собою фрагмент коду, який співпадає з оригінальним за винятком різних імен ідентифікаторів, що задаються користувачем (імен змінних, констант, класів, методів тощо), типів, розмітки, коментарів. Зарезервовані слова та синтаксична структура залишається такою ж, як в оригіналі. Порівняємо два фрагменти коду на рисунках 4 та 5, бачимо що синтаксична структура в них однакова, хоча деякі змінні перейменовані.

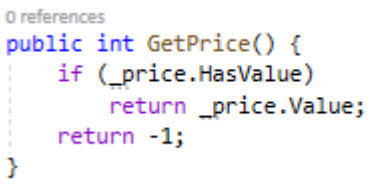
```
if (a - b == 0)
{
    c = d;
    a = b + 2;
} else // new comment
    c = b;
```

Рисунок 4

```
if (m - n == 0)
{
    x = y;
    m = n + 11;
} else // new comment
    x = n;
```

Рисунок 5

В третьому типі дублювання коду скопійований фрагмент додатково модифікований за допомогою зміни, додавання та видалення деякої ділянки коду. Розглянемо оригінальний фрагмент, на рисунку 6 та його копію на рисунку 7. Після чого в таблиці 1, розглянемо різницю між цими двома фрагментами коду.



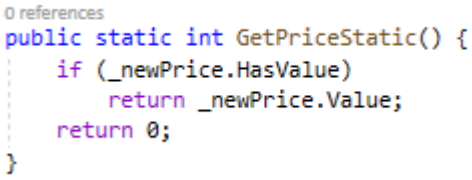


Рисунок 6 Рисунок 7

Таблиця 1

Фрагмент 1	Фрагмент 2	Статус
public	Public	Точна копія
	static	Вставка нового фрагменту
int	int	Точна копія
GetPrice	GetPriceStatic	Заміна
_price	_newPrice	Заміна
-1	0	Заміна

З таблиці видно, що деякі ділянки отримані з інших шляхом повного копіювання і заміни імен деяких ідентифікаторів та літералів, а також вставкою додаткового ключового слова. Тому ці зміни можна віднести до третього типу. Якщо не враховувати добавлення додаткового «шуму», то копіюється фрагмент може бути віднесений до дублювання другого типу.

Четвертий тип копій є результатом семантичного подібності між двома або більше фрагментами коду. У цьому типі плагіату, скопійований фрагмент не обов'язково скопійований з оригіналу. Два фрагмента коду можуть бути розроблені двома різними програмістами, реалізовувати при цьому однакову логіку і аналогічні за своєю функціональністю алгоритми.

На рисунках 8 та 9 представлені алгоритми пошуку факторіалу числа, ці фрагменти коду з семантичної точки зору є різними, проте однаковими за своїм функціональним показником.

```
0 references
public int FactorialFor(int n) {
    int res = 1;
    for (int i = 2; i <= n; i++)
        res *= i;
    return res;
}
```

Рисунок 8

```
1 reference
public int Factorial(int n) {
    if (n == 0)
        return 1;
    return n * Factorial(n - 1);
}
```

Рисунок 9

1.1.2 Види представлення програмного коду

Існує велика кількість способів подання і подальшого розбору вихідних кодів програм. В основному всі вони складаються із фази перетворення і фази порівняння. На першому етапі вихідний текст переводиться у внутрішній формат, який дозволяє використовувати більш ефективні алгоритм порівняння. У наступній фазі відбувається безпосередній пошук дублювання і обраховується певна метрика, що характеризує їх кількість. Тому логічно класифікувати методи виявлення плагіату відповідно до представлення тексту. Поширеними представленнями тексту вихідного коду є:

- String-based - є поширеним підходом, використовуваним в інформатиці. Представлення зберігається в такому ж виді як вихідний код.
- Token-based - код програми представляється набором.
- Tree-based - програмний код представлений у вигляді синтаксичного дерева.
- PDG-based (PDG - ProgramDependencyGraph) – програма представляється, як орієнтований граф, вершинами котрого будуть: точка входу, безліч точок

виходу, команди передачі управління. З кожною вершиною асоціюється структура, що зберігає адресу цього вузла, і адресу переходу з цього вузла (якщо вони існують).

1.1.3 Методи перевірки вихідного коду на плагіат

На основі представлення програмного коду ми можемо застосовувати різні методи знаходження дублів в коді. Розглянемо основні з них.

При використанні підходу представлення коду String-based, код програми не піддається ніяким змінам і являє собою символьний рядок. Метод не враховує структуру програми і її синтаксичні особливості, тому перейменування функцій і змінних або несуттєві зміни в коді можуть стати серйозною перешкодою для належного функціонування цього методу. Плюсом такого методу є те, що його можна досить просто реалізувати і можливість порівняння не тільки програмного коду, а й звичайного тексту - коментарів та інших незначних частин програми. Крім того, з текстом легко працювати і перед аналізом можна провести деяку нормалізацію і видалити несуттєві символи та слова. Так же варто відзначити що даний метод не має прив'язки до мов програмування.

Наступним методом, є метод оцінювання метрик [2]. Вихідний код програми теж не зазнає жодних змін. Але на відміну від попереднього методу просте перейменування функцій та змінних, а так само незначні маніпуляції з вихідним кодом не впливатимуть на роботу цього методу. У якості відправної точки для виявлення дублювання вибираються деякі кількісні характеристики програми. Це можуть бути кількість циклів, змінних, умов або їх об’єднання. Оскільки в досить значній кількості мов програмування використовуються однакові ключові слова для задання циклів та умов, то цей метод так само досить універсальний і може працювати

практично незалежно від мови програми. Мінусом даного метода є досить часті помилкові спрацьовування. Особливо помітно на невеликих програмах, так як кількість основних ключових конструкцій в них відрізняється незначно.

Одним із способів аналізу програми є дослідження частоти появлення і розміщення операторів в тілі програми. Частоту можна визначити як кількість поява конкретного оператора, поділене на кількість поява всіх операторів. Звичайно високий відсоток збігів для різних операторів двох різних вихідних кодів не завжди говорить про наявність плагіату, але служить стабільним індикатором для подальшої перевірки.

Можна модифікувати дану метрику, і досліджувати розміщення операторів в тілі програми, після чого знайдемо взаємну кореляцію для двох програм. Будемо циклічно здвигати послідовність операторів однієї програми відносно іншої, та записувати результат, в результаті отримуємо залежність зміщення послідовності операторів однієї програми відносно іншої до кількості однакових операторів при такому зміщенні.

Одним з найпростіших, але в той же час дієвих способів виявлення плагіату є аналіз коментарів до програм. Для отримання коментарів (як простих, так і багаторядкових) з програмного коду програми застосуємо функціонал регулярних виразів. Після, на отримані рядки коментарів можна застосувати метод Шинглів для аналізу їх схожості. Відсутність збігів не є гарантом унікальності програм, так як коментарі піддаються легкій модифікації або повному видаленню. Проте, досить високий відсоток їх схожості практично точно свідчить про наявність плагіату в програмах.

Ще один важливий метод – метод, що працює на токенізації. У цьому підході весь текст перетворюється в послідовність лексем, яка потім сканується для пошуку дублікатів. При такому підході зберігаються всі істотні і пропускаються всі поверхневі деталі коду. Всі долі вихідного тексту, які легко піддаються зміні - імена

змінних, функцій, класів, коментарі будуть ігноруватись. Візьмем для прикладу дві програми, зображені відповідно на рисунках 10 та 11.

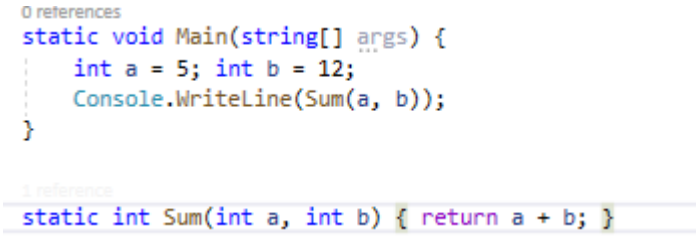


Рисунок 10

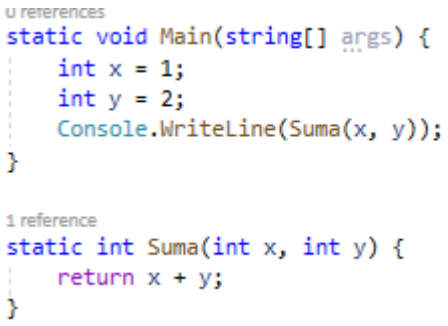


Рисунок 11

Очевидно, що обидві програми роблять одне і теж - виводять суму двох чисел. Незначною обробкою код однієї з них може бути отриманий з іншого, досить перейменувати змінні, функцію і підставити інші тестові значення змінних. Завдяки процедурі токенізації, вдасться не брати до уваги ці аспекти і виявити схожі участки. Розбивання програми на токени в загальному вигляді можна представити так:

Розбиваємо програму на набори операторів. Потім кожному оператору, що належить певному набору, приписуємо символьний або цифровий код, узгоджений спочатку.

З отриманих кодів, будуємо рядок, зберігаючи при цьому порядок як у вихідній програмі.

Застосовуємо один з алгоритмів порівняння для простих текстів.

Завдяки такому підходу, ми автоматично ігноруємо розділові символи, назви функцій і змінних (класів, об'єктів і так далі), запобігаємо впливу дрібних змін вихідного коду програми. Мінус такого підходу - це залежність процесу токенізації від конкретної мови програмування. Одне з рішень цієї проблеми - використання декількох різних лексерів для різних класів мов.

На основі утворених токенів можна використати різні методи для перевірки наявності дублікатів в коді. Одними з найбільш популярних алгоритмів є алгоритми W-шинглінг (W-shingling) та алгоритм Вагнера-Фішера. Оглянемо їх більш детально.

Шингл - це невеликий, що складається з декількох слів, фрагмент текста, оброблений за спеціальною методикою алгоритму шинглів (w-shingling), що являє собою обробку вхідних даних з використанням набору унікальних шинглів (N-грам, суміжних підпослідовностей токенів в документі), які можуть бути використані для оцінки подібності двох документів. Символ N означає кількість токенів в кожному наборі. Отже, ми маємо два тексти і нам потрібно припустити, чи є вони дублікатами. Реалізація алгоритму має на увазі кілька етапів:

- Нормалізація текстів;
- Розбиття тексту на шингли;
- Знаходження контрольних сум;
- Пошук однакових підпослідовностей.

Після цього результуюче значення, що виражає проценті подібності двох текстів можна буде знайти за формулою 1:

$$r\,(A, B) = \frac{S\,(A) \cap S\,(B)}{S\,(A) \cup S\,(B)} * 100 \qquad (1),$$

де чисельник відповідає за кількість однакових підпослідовностей, а знаменник – сумарну кількість підпослідовностей.

Таким чином, застосувавши даний алгоритм для послідовності токенів вихідних кодів програм, ми зможемо з'ясувати відсоток їх дублювання.

Алгоритм Вагнера-Фішера дозволяє обчислити найкоротшу відстань Левенштейна, яка відображає подібність між двома рядками [3]. Схожість вихідного і цільового тексту вимірюється як кількість замін, вставки і видалення, необхідних для зміни одного рядка в інший.

Відстань Левенштейна збільшується в залежності від кількості перетворень, необхідних для перетворення одного рядка в інший. Якщо вихідний рядок - «тип», а цільова - «топ», то відстань Левенштейна буде рівною одиниці. Це означає, що нам необхідно зробити одну заміну, для того щоб перетворити один рядок в інший. Реалізація алгоритму включає в себе кілька кроків:

1. Встановлюємо змінну n рівною довжині початкового рядка s.
2. Встановлюємо змінну m рівну довжині цільового рядка t.
3. Якщо n = 0, повертаємо m і завершуємо. Якщо m = 0, повертаємо n і завершуємо.
4. Створення матриці розміру n * m, ініціалізація першого рядок від 0 до n., ініціалізація першого стовпця від 0 до m.
5. У подвійному циклі по змінним n і m починаємо зрівнювати символи.
6. Якщо s [i] рівне t [j], вартість операції дорівнює 0. Якщо s [i] не дорівнює t [j], вартість операції дорівнює 1.
7. Встановити комірку матриці (відстань Левенштейна) d [i, j] в значення мінімальне з:
   a) Комірки зверху + 1: d [i-1, j] + 1
   b) Комірки зліва + 1: d [i, j-1] + 1
   c) Комірки зліва і зверху по діагоналі плюс вартість операції з кроку 6: d [i-1, j-1] + cost.

8. Після ітерацій кроків 3-6, відстань Левенштейна буде в останній комірці: d [m, n].

Ми можемо застосувати цей алгоритм для обчислення подібності вихідних кодів програм. Спочатку проводимо процес нормалізації вихідного коду, потім отримуємо рядок з лексем однієї та іншої програми в процесі токенізації. І викликаємо функцію знаходження відстані Левенштейна для цих рядків. Отримаємо чисельне значення, що виражає різницю між двома рядками.

Тоді значення відсотку схожості двох програм можна обчислити як: PlagiarizedValue = {1-DiffMax (SS, TS)} * 100, де SS = довжина першої програми, TS = довжина другої програми. Якщо значення PlagiarizedValue більше порогового значення, то ми вважаємо, що програми є плагіатом.

1.1.3 Існуючі додатки перевірки коду на наявність плагіату

Розглянемо основні додатки для перевірки коду на плагіат, що є на ринку [4].

Moss - це інструмент, розроблений для виявлення подібності вихідного коду з метою виявлення програмного плагіату. Він доступний як веб-служба: документи можна завантажувати за допомогою сценарію, а результати відображаються через веб-інтерфейс після обробки. Moss використовує n-грами рівня символів (суміжну підпослідовність довжиною n) як функції для порівняння документів. Замість порівняння всіх n-грамів, лише деякі функції порівнюються з міркувань ефективності. Зазвичай застосовувана техніка для вибору текстових функцій - це обчислення хеш-значення для кожної функції, але вибір лише підмножини цих функцій за допомогою 0 mod p для фіксованої p. Автори відзначають, що ця техніка часто залишає прогалини в документах, підвищуючи ймовірність пропуску збігів між документами. Щоб цього

не сталося, вони використовують алгоритм, який вони називають вікнуванням: замість випадкового вибору n-грамів з документа вони вибирають для кожного вікна принаймні одну функцію. Крім того, вони використовують велике значення n, щоб уникнути шумних результатів, і видаляють пробіли, щоб уникнути збігів на пробілах.

JPlag - це інструмент для упорядкування програм за схожістю з урахуванням набору програм. Автори стверджують, що порівняння програм, заснованих лише на векторному елементі, відкидає занадто багато структурної подібності. Натомість вони намагаються відповідати тим, що вони називають структурними особливостями. Замість безпосереднього використання тексту вони спочатку перетворюють вихідний код Java на список маркерів, таких як BEGINCLASS, ENDCLASS та BEGINMETHOD та ENDMETHOD. Потім вони використовують алгоритм для пошуку збігів між документами за допомогою списку маркерів, від найбільшого до найменшого. Існує певний параметр для мінімального розміру збігів, інакше малі збіги трапляються занадто часто. Вони застосовують кілька оптимізацій середовища виконання до базового алгоритму порівняння з найгіршим випадком O (n^3) складності (де n - розмір документів) з використанням алгоритму Карпа-Рабіна. Вони порівнюють різні критерії відсікання, використовуючи різні методи, щоб створити порогове значення, враховуючи значення подібності в наборі даних. Вони також порівнюють вплив мінімальної довжини відповідності та набору маркерів, які слід використовувати, виконуючи деякі вимірювання на наборах даних.

SIM - це програмний засіб виявлення плагіату та тексту, написаний на мові C. Він працює шляхом першого маркування файлів та пошуку найдовшої загальної послідовності в парах файлів.

Marble - це інструмент, який розробляється з урахуванням простоти. Інструмент складається з трьох фаз: нормалізації, сортування та виявлення. Етап нормалізації перетворює вихідний код із необробленого тексту в більш абстрактну програму.

Ключові слова, такі як клас, змінна, тип, імена перетворюються на X, а числові літерали - на N. Оператори та символи також залишаються на місці. Імпорт декларацій видаляється під час цього перетворення. Після нормалізації класи та члени класу сортуються лексикографічно, це робить інструмент нечутливим до впорядкування цих програмних конструкцій. Нарешті спрощена програма порівнюється різницевим інструментом на основі рядків Unix diff, використовуючи кількість змінених рядків, нормовану на загальну довжину двох файлів.

Plaggie - це механізм виявлення плагіату вихідного коду, призначений для вправ програмування на Java. За зовнішнім виглядом та функціональністю він схожий на JPlag, але існують також аспекти Plaggie, які сильно відрізняється від JPlag: Plaggie потрібно встановлювати локально, а його вихідний код відкритий. Це автономна програма Java з командного рядка. Основний алгоритм, який використовується для порівняння двох файлів вихідного коду, такий самий, як і для JPlag: токенізація з подальшим жадібним плитуванням рядків. Автори згадують, що вони не впровадили оптимізацію, яка була реалізована в JPlag.

В таблиці 2 представлена порівняльна характеристика додатків для виявлення плагіату.

Таблиця 2

Параметр	Moss	JPlag	SIM	Marble	Plaggie
Кількість мов	23	6	5	1	1
Відрикитий програмний код	Ні	Ні	Так	Ні	Так
Виключення шаблонних файлів	Так	Так	Ні	Ні	Так
Виключення малих файлів	Так	Так	Ні	Так	Ні
Історичне порівняння	Ні	Ні	Так	Так	Ні
Можливість розширення	Ні	Ні	Так	Ні	Ні

1.2 Архітектура Web додатку

1.2.1 Загальний огляд можливостей ASP.NET Core

ASP.NET Core є кроссплатформенним, високопродуктивним середовищем з відкритим вихідним кодом для створення сучасних хмарних додатків, підключених до Інтернету [5]. ASP.NET Core дозволяє виконувати наступні завдання:

- Створювати веб-додатки та служби, додатки Інтернету речей (IoT) і серверні частини для мобільних додатків.
- Використовувати вибрані засоби розробки в Windows, macOS і Linux.
- Виконувати розгортання в хмарі або локальному середовищі.
- Запускати в .NET Core.

Мільйони розробників використовували і продовжують використовувати ASP.NET 4.x для створення веб-додатків. ASP.NET Core - це модифікація ASP.NET 4.x з архітектурними змінами, що формують більш раціональну і більш модульну платформу. ASP.NET Core надає наступні переваги:

- Єдине рішення для створення призначеного для користувача веб інтерфейсу і веб-API.
- Тестованість.
- Razor Pages спрощує написання коду для сценаріїв сторінок і підвищує його ефективність.
- Blazor дозволяє використовувати в браузері мову C # разом з JavaScript. Спільне використання серверної і клієнтської логік додатків, написаних за допомогою .NET;
- Можливість розробки і запуску в ОС Windows, macOS і Linux.
- Відкритий вихідний код і орієнтація на співтовариство.

- Інтеграція сучасних клієнтських платформ і робочих процесів розробки.
- Підтримка розміщення служб віддаленого виклику процедур (RPC) за допомогою gRPC.
- Хмарна система конфігурації на основі середовища.
- Вбудоване впровадження залежностей.
- Спрощений високопродуктивний модульний конвеєр HTTP-запитів.
- Наступні можливості розміщення:
    - Kestrel
    - Служби IIS
    - HTTP.sys
    - Nginx
    - Apache
    - Docker
- Управління паралельними версіями.
- Інструментарій, що спрощує процес сучасної веб-розробки.

1.2.2 Принципи «чистої» архітектури

Розглянемо основні принципи «чистої» архітектури, якими керуватимемось, в процесі створення веб додатку.

Для розуміння чистої архітектури, потрібно спочатку розібратись, що таке архітектура програмного забезпечення в цілому.

Архітектура програмної системи - це форма, яка надається системі її творцями. Ця форма утворюється поділом системи на компоненти, їх організацією та визначенням способів взаємодій між ними. Мета форми - спростити розробку, розгортання і супровід програмної системи, що міститься в ній [6]. Головна стратегія

такого спрощення в тому, щоб якомога довше мати якомога більше варіантів. Головне призначення архітектури - підтримка життєвого циклу системи. Гарна архітектура робить систему легкою в освоєнні, простою в розробці, супроводі і розгортанні. Кінцева її мета - мінімізувати витрати протягом терміну служби системи і максимізувати продуктивність програміста.

За останні кілька десятиліть ми бачили цілий ряд ідей про організації архітектури. У тому числі:

- Гексагональна архітектура (Hexagonal Architecture, також відома як архітектура портів і адаптерів), розроблена Алістером Кокберн і описана Стівом Фріманом і Натом.
- DCI (дані, контекст, взаємодія), запропонована Джеймсом Копліеном і Трюгве Реенскаугом.
- BCE (границі, контроль, модель), запропонована Іваром Якобсоном.

Незважаючи на відмінності в деталях, всі ці архітектури дуже схожі. Вони всі переслідують одну мету - розподіл завдань. Вони всі досягають цієї мети шляхом ділення програмного забезпечення на рівні. Кожна архітектура має хоча б один рівень для бізнес-правил і ще один для користувацького і системного інтерфейсів.

Кожна з цих архітектур сприяє створенню систем, що володіють наступними характеристиками :

- Незалежність від фреймворків. Архітектура не залежить від наявності будь-якої бібліотеки. Це дозволяє розглядати фреймворки як інструменти, замість того щоб намагатися втиснути систему в їх рамки.
- Простота тестування. Бізнес-правила можна тестувати без користувацького інтерфейсу, бази даних, веб-сервера і будь-яких інших зовнішніх елементів.
- Незалежність від призначеного для користувача інтерфейсу. Призначений для користувача інтерфейс можна легко змінювати, не зачіпаючи решту системи.

Наприклад, веб-інтерфейс можна замінити консольним інтерфейсом, не міняючи бізнес-правил.

- Незалежність від бази даних. Ви можете поміняти Oracle або SQL Server на Mongo, BigTable, CouchDB або щось ще. Бізнес-правила не прив'язані до бази даних.
- Незалежність від будь-яких зовнішніх агентів. Ваші бізнес-правила нічого не знають про інтерфейси, що ведуть до зовнішнього світу.

Приклад ідеальної архітектури наведений на рисунку 12.

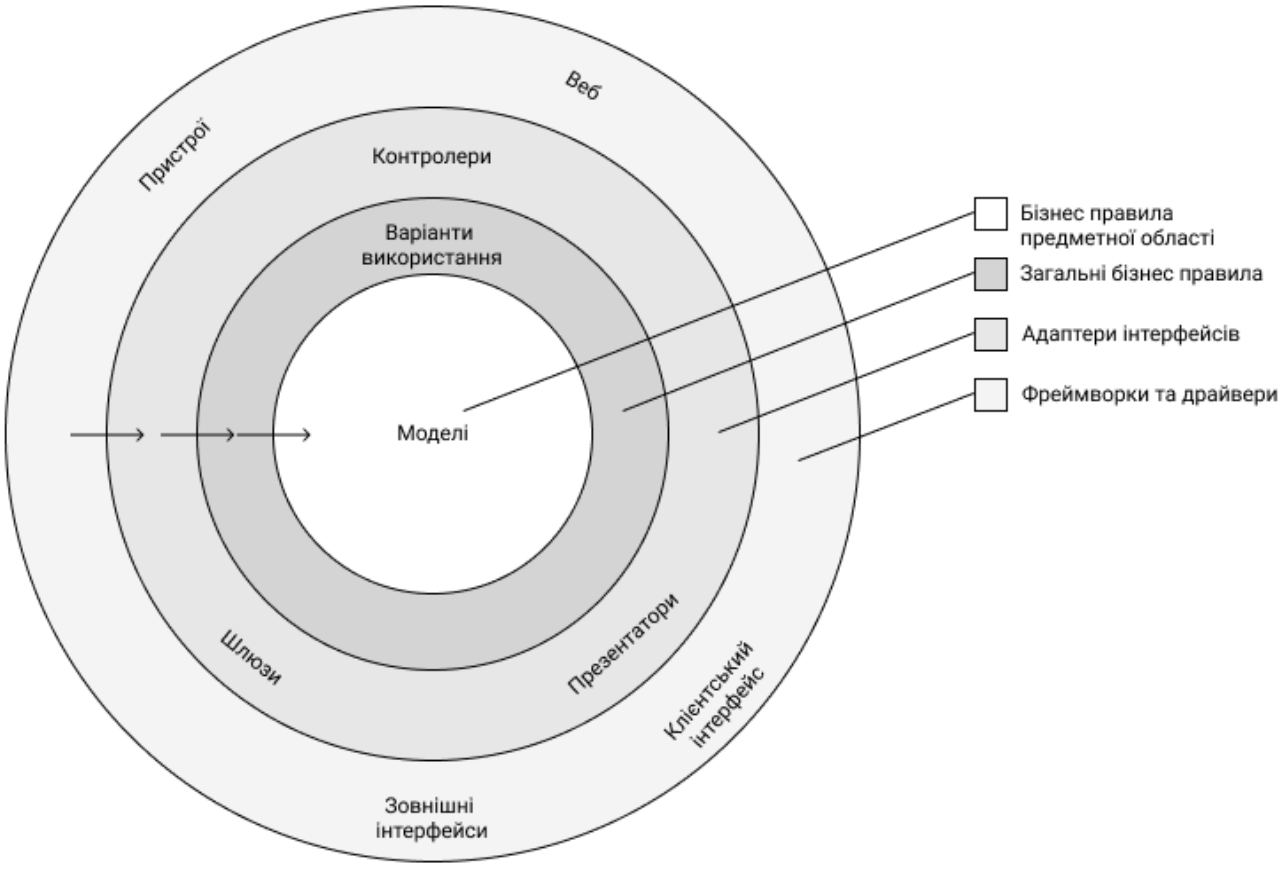


Рисунок 12

Концентричні кола на рисунку 12 представляють різні рівні программного забезпечення. Чим ближче до центру, тим вище рівень. Зовнішні кола - це механізми. Внутрішні - політики. Головним правилом, що призводить цю архітектуру в дію, є правило залежностей (Dependency Rule): залежності в вихідному коді повинні бути спрямовані всередину, в сторону високорівневих політик. Ніщо у внутрішньому колі нічого не знає про зовнішні колах. Наприклад, імена, оголошені в зовнішніх колах, не повинні згадуватися в коді, що знаходиться у внутрішніх колах. Сюди відносяться функції, класи, змінні і будь-які інші іменовані елементи програми. Точно так же формати даних, оголошені в зовнішніх колах, не повинні використовуватися у внутрішніх, особливо якщо ці формати генеруються фреймворком в зовнішньому колі. Ніщо в зовнішньому колі не повинно впливати на внутрішні кола.

Моделі містять в собі критичні бізнес-правила рівня підприємства. Сутність може бути об'єктом з методами або набором структур даних і функцій. Сама організація не важлива, якщо суті доступні для використання різними додатками на підприємстві.

Програмне забезпечення на рівні варіантів використання містить бізнес-правила, характерні для додатка. Воно інкапсулює і реалізує всі варіанти використання системи. Варіанти використання організовують потік даних до моделей і з них і вимагають від цих сутностей їх критичні бізнес-правила для досягнення своїх цілей.

Програмне забезпечення на рівні адаптерів інтерфейсів - це набір адаптерів, що перетворюють дані з формату, найбільш зручного для варіантів використання і сутностей, в формат, найбільш зручний для деяких зовнішніх агентів, таких як база даних або веб-інтерфейс.

Самий зовнішній рівень моделі на рисунку 12 зазвичай складається з фреймворків і інструментів, таких як база даних і веб-фреймворк. Як правило, для

цього рівня потрібно писати не дуже багато коду, і зазвичай цей код грає роль сполучної ланки з наступним внутрішнім світом.

Кола на рисунку 12 лише схематично зображують основну ідею: іноді вам може знадобитися більше чотирьох кіл. Фактично немає ніякого правила, який стверджує, що кіл має бути саме чотири. Але завжди діє правило залежностей. Залежності в вихідному коді завжди повинні бути спрямовані всередину.

1.3 Огляд використаних API

1.3.1 Amazon Web Serivces

1.3.1.1 AWS Cognito

Сервіс Amazon Cognito дозволяє швидко і просто додавати можливості реєстрації, авторизації і контролю доступу користувачів в мобільні та інтернет-додатки [7]. Amazon Cognito масштабується до мільйонів користувачів і підтримує авторизацію за допомогою соціальних постачальників посвідчень (Apple, Facebook, Google, Amazon), а також постачальників корпоративних посвідчень на основі SAML 2.0 і OpenID Connect. Схема роботи сервісу представлена на рисунку 13.

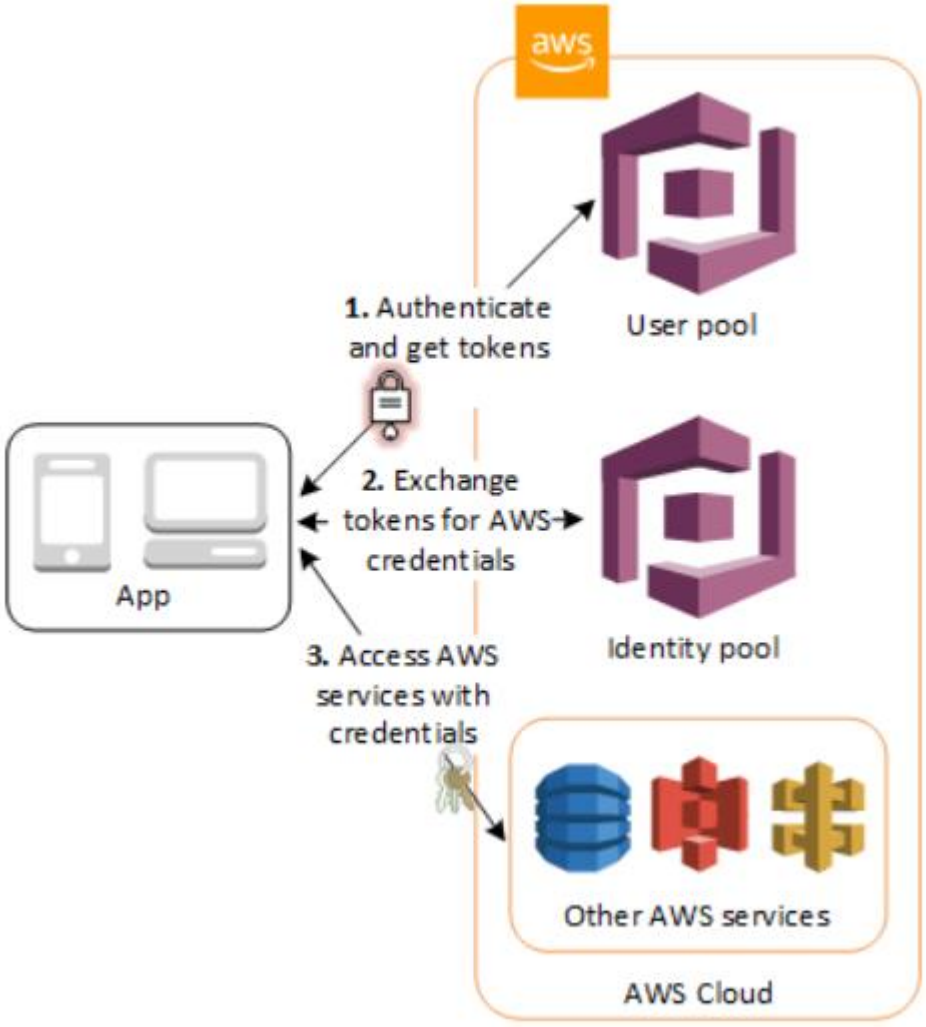


Рисунок 13

1.3.1.2 AWS Lambda

AWS Lambda - це безсерверний обчислювальний сервіс, який дозволяє виконувати код без необхідності виділяти сервери і управляти ними, створювати логіку масштабування кластера з урахуванням робочих навантажень, підтримувати

інтеграцію подій або керувати часом виконання [8]. Lambda дозволяє виконувати код практично будь-якої програми або серверного сервісу без адміністрування.

1.3.1.3 AWS EC2

Обчислювальний хмара Amazon Elastic Compute Cloud (Amazon EC2) - це веб-сервіс, що надає безпечні масштабовані обчислювальні ресурси в хмарі. Він допомагає розробникам, спрощуючи проведення обчислень в хмарі в масштабі всього Інтернету [9]. Простий веб-інтерфейс сервісу Amazon EC2 дозволяє отримати доступ до обчислювальних ресурсів і налаштувати їх з мінімальними зусиллями. Він надає користувачам повний контроль над обчислювальними ресурсами, а також перевірену обчислювальну середу Amazon для роботи.

1.3.2 GitHub

Github API (або Github ReST API) - це ті API, які ви можете використовувати для взаємодії з GitHub. Вони дозволяють створювати та керувати сховищами, гілками, проблемами, запитами на витягування та багатьма іншими. Для отримання загальнодоступної інформації (наприклад, загальнодоступних сховищ, профілів користувачів тощо) ви можете викликати API. Для інших дій потрібно надати автентифікований маркер.

1.3.3 Entity Framework Core

Entity Framework Core - це полегшена, розширювана, відкрита та кроссплатформенна версія популярної технології доступу до даних Entity Framework. EF Core може служити об’єктно-реляційним картографом (ORM), який:

- Дозволяє розробникам .NET працювати з базою даних із використанням об’єктів .NET.
- Усуває необхідність у більшості коду доступу до даних, який зазвичай потрібно
- EF Core підтримує багато механізмів баз даних

Entity Framework підтримує такі підходи до розробки моделі:

- Створення моделі із існуючої бази даних.
- Ручне створення модель, щоб вона відповідала базі даних.
- Після створення моделі , використання EF Migrations для створення бази даних із моделі. Міграції дозволяють розвивати базу даних у міру зміни моделі.

РОЗДІЛ 2 РЕАЛІЗАЦІЯ ВЕБ ДОДАТКУ

2.1 Уточнена постановка задачі

Метою даної розробки є задоволення потреби користувачів у легкому способі маніпулюванням лабораторними роботами студентів, а саме, для студентів, це єдина платформа, на якій вони опрацьовують всі лабораторні роботи курсу, і мають можливість спостерігати за своїми результатами. Для викладачів це система, яка дає можливість формувати групи, створювати курси, добавляти в них студентів та групи. Також в курс викладач може добавити лабораторну роботу, яка зберігатиметься в системі управління версіями вихідного коду GitHub, та автоматично буде провірятись на наявність плагіату. На основі результати роботи антиплагіатора та часу здачі роботи викладач матиме можливість виставити оцінку за дану роботу.

2.2 Реалізація додатку, що перевіряє код на наявність плагіату

Загальний опис роботи зображений на діаграмі послідовності на рисунку 14. Спочатку актор відправляє запит на сервіс перевірки програмного коду на плагіат. В свою чергу сервіс працює з декількома об’єктами, що відповідають за обробку файлів та обрахунок проценту плагіату. А саме існує система, що відповідає за перевірку валідності типу файлу, тобто перевіряє його розширення, на основі чого визначає мову програмування, якщо ця мова підтримується додатком, то повертається розширення сервісу, інакше помилка, в якій вказано про не підтримку даної мови програмування. Після отримання типу файлу і мови програмування сервіс створює «лексер» основною задачею, котрого є парсинг файлу, та перетворення його на масив токенів. Після чого сервіс отримує масив токенів, та передає його на обробку системі,

що відповідає за аналіз та підрахунок унікальності коду. Після чого отримуємо результат на скільки код унікальний.

А рамках всього додатку, коли студент відправляє лабораторну роботу на перевірку, тригериться бізнес логіка, яка обробляє GitHub сховище студента, та дістає всі файли з нього, після чого система проходить по всіх репозиторіях інших студентів і по черзі дістає файли кожного і відправляє запит на сервіс перевірки на унікальність коду. В свою чергу для поточної роботи вираховується процент унікальності з кожним іншим студентом, і серед цього переліку береться найменше значення унікальності, тобто там де процент плагіату найбільший.

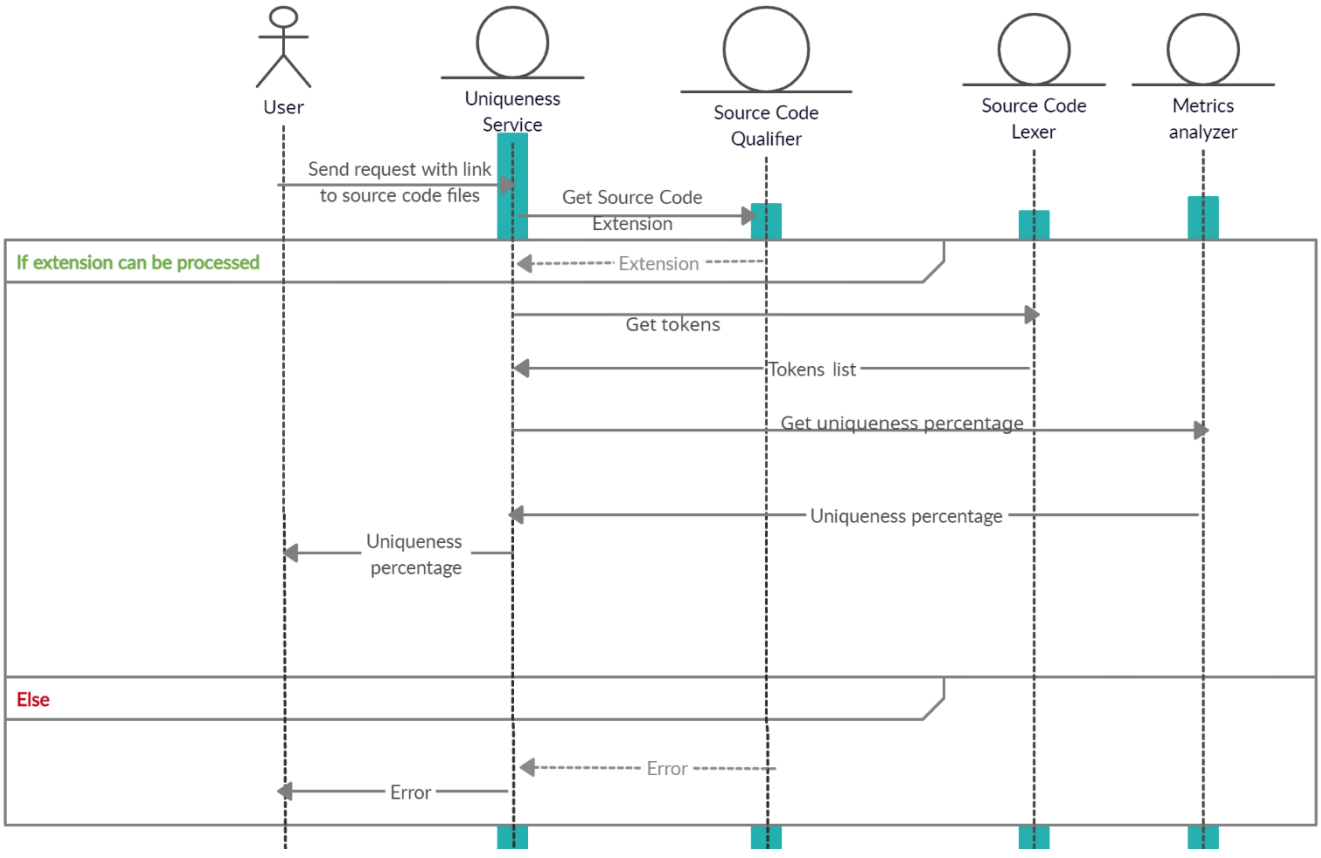


Рисунок 14

2.2.1 Токенізація програмного коду

Токенізація вихідного коду програми відбувається за допомогою класу BaseLexer, котрий на вхід приймає параметр розширення файлу, після чого він ініціалізує відповідну стратегію токенізації файлу, на даний момент підтримує мови C#, Java, C++. В класі лексера є два методи, один повертає масив чисел – токенів, для нормалізованого програмного коду, інший повертає також масив чисел – токенів, лише суто для коментарів в програмі. Таким чином на основі цих токенів відбувається підрахунок унікальності. На рисунку 15 представлено діаграму класів даного модуля додатку.

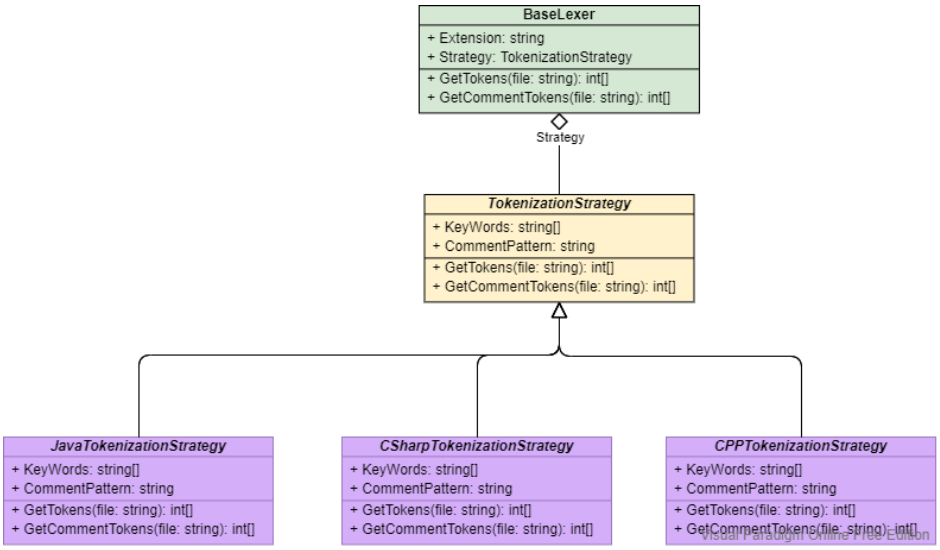


Рисунок 15

Сам процес токенізації відбувається у відповідній мові стратегії. Кожна стратегія реалізує базову стратегію, внаслідок чого наслідує такі параметри як масив ключових слів для мови, рядок який відповідає патерну пошуку коментарів в коді та методи аналогічні до методів лексера. На основі ключових слів відбувається лексичний та семантичний аналіз файлу, і кожній змінній, ідентифікатору імені

присвоюється нове автозгенерене ім’я. Приклад роботи такої заміни можна побачити на рисунку 16.

Після чого результуючий код по слову хешується, стандартною функцією хешування мови C#, котра в основі хешування бере просте число 31. В результаті отримуємо масив чисел що відповідають нашим токенам.

За допомогою патерну пошуку коментарів, ми знаходимо всі коментарі в коді і також по слову хешуємо їх, в результаті отримуємо набір чисел, що відповідають токенам коментарів.

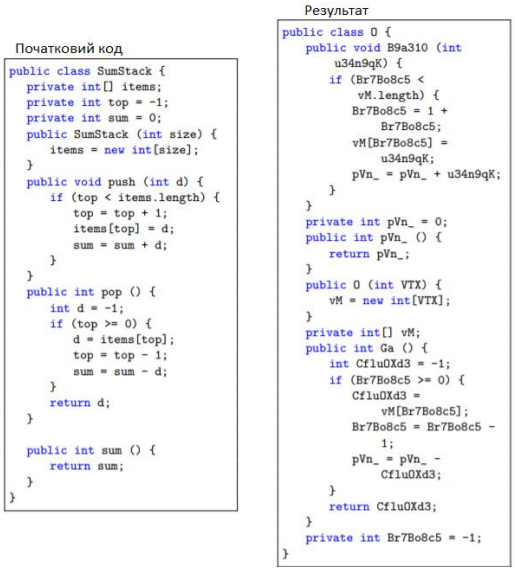


Рисунок 16

2.2.2 Підрахунок унікальності коду на основі токенів

Підрахунком унікальності коду на основі токенів є функціональною відповідальністю аналізатора метрик.

Його діаграма класів представлена на рисунку 17, з неї бачимо, що в складі нашого аналізатора метрик присутній не один клас.

В аналізаторі метрик є єдиний публічний метод на отримання унікальності, котрий в собі пробігається по всіх можливих алгоритмах системи, що реалізують базовий абстрактний алгоритм UniquenessAlgorithm, і викликає для них функцію отримання унікальності, таким чином ми отримуємо набір дробових чисел які лежать в діапазоні від 0 до 1, що відповідають на скільки код є унікальним, і ці числа співставляються з відповідними їм алгоритмами.

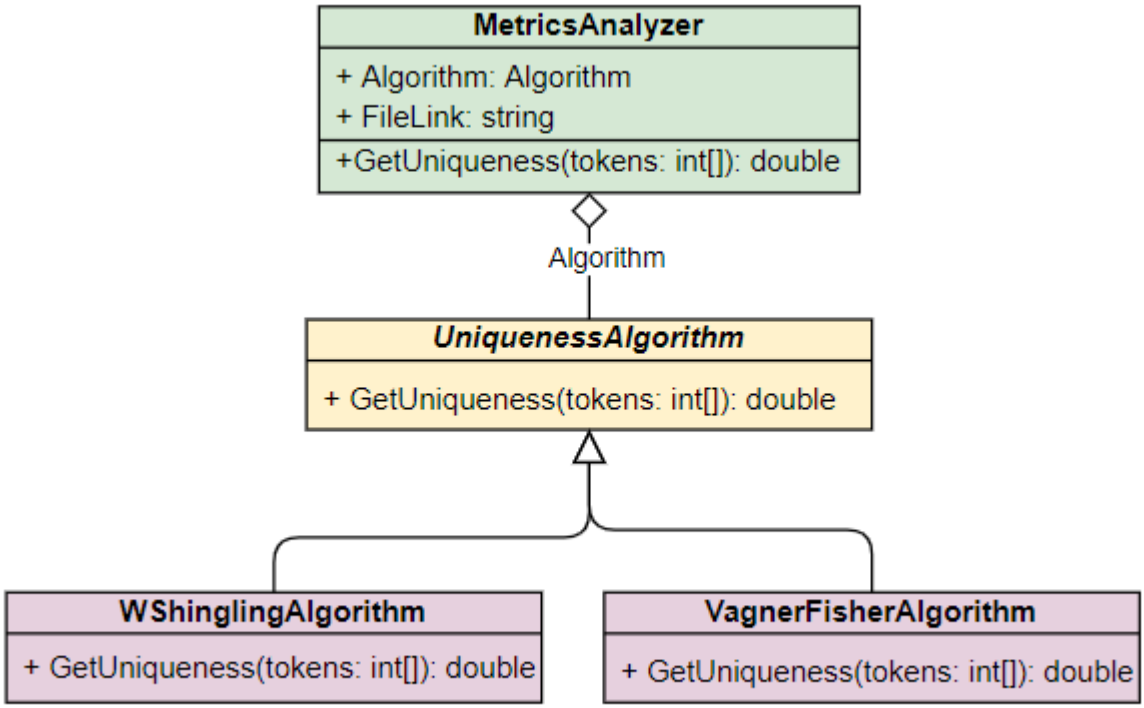


Рисунок 17

В системі реалізовано базові два алгоритми – В-шиглінг та Вагнера-Фішера. Також користувач може розширяти набір алгоритмів реалізуючий новий алгоритм, що імплементує клас UniquenessAlgorithm.

Так само ці алгоритми виконуються на токенах коментарів. Після чого рахується середнє зважене цих коефіцієнтів, вагові коефіцієнти вибираються користувачем, початковими значеннями є по 25% на кожний алгоритм, оскільки кожен алгоритм застосовується два рази, для коду та коментарів.

2.2.3 Аналіз результатів

В ході розробки додатку на перевірку плагіату, був проведений ряд тестувань, були порівняно результати додатка та програми JPlag. Результати наведені в таблиці 3.

Таблиця 3

Код першої програми	Код другої програми	Процент унікальності додатка	Процент унікальності JPlag
Зображено на рисунку 18	Зображено на рисунку 19	0%	0%
Зображено на рисунку 20	Зображено на рисунку 21	3%	0%
Зображено на рисунку 22	Зображено на рисунку 23	33%	20%
Зображено на рисунку 24	Зображено на рисунку 25	80%	54%

```
namespace ConsoleApp1 {
    0 references
    class Program {
        0 references
        static void Main(string[] args) {

        }
    }
}
```

Рисунок 18

```
namespace ConsoleApp2 {
    0 references
    class Program {
        0 references
        static void Main(string[] args) {

        }
    }
}
```

Рисунок 19

```
int a = 1;
int b = 100;
if (a > b) {
    a = a + b;
} else {
    b -= a;
}
```

Рисунок 20

```
var x = -10; var y = 33;
if (x > y) { // Comment
    x = x+y;
} else {
    y = y - x;
}
```

Рисунок 21

```
int a = 1;
int b = 100;
if (a > b) {
    a = a + b; // This is new a
} else {
    b -= a; // This is new b
}
```

Рисунок 22

```
int a = 1;
int b = 100;
if (a > b) {
    a = Sum(a, b); // This is new a
} else {
    Diff(out b, a); // this is new b
}
```

Рисунок 23

```
int n = 20, res = 1;
for (int i = 1; i <= n; i++)
    res *= i;
```

Рисунок 24

```
0 references
static void Main(string[] args) {
    var n = 25;
    int res = Factorial(n);
}

2 references
private static int Factorial(int n) {
    if (n == 0)
        return 1;
    return Factorial(n - 1) * n;
}
```

Рисунок 25

Результати роботи програм представлені для рисунка 18 та рисунка 19 очевидно зрозумілі – одна програма є копією іншої і процент унікальності рівний 0. Для наступної пари були зроблені заміни імен змінних, констант та дещо змінене оформлення, розроблений додаток дав, дещо вищу оцінку чим JPlag, це пояснюється не досконалістю алгоритмів. Для третьої пари отримуємо знову, що результат нашого додатку дещо вищий, чим JPlag, тут це пояснюється наявністю однакових коментарів, що на мою думку, є перевагою нашого алгоритму. Для останньої пари, ми маємо дублікат четвертого типу, коли синтаксичне дерево різне, тут наш алгоритм сильно програє JPlag, більший процент дублікату тут виявляється на стандартних конструкціях мови, але сама суть не виявляється.

Отже, при порівнянні двох програм для різних ситуацій, бачимо, що в нашого алгоритму є як сильні, так і слабкі сторони.

2.3 Реалізація веб додатку

2.3.1 Загальний опис архітектури

Архітектура додатку розроблялась таким чином, щоб створити максимально гнучкий додаток, який можливо легко розширювати, просто супроводжувати та можливо розмістити в хмарі не прикладаючи жодних зусиль. При проектуванні веб додатку використовувались всі принципи чистої архітектури, в результаті отримали архітектуру зображену на рисунку 26.

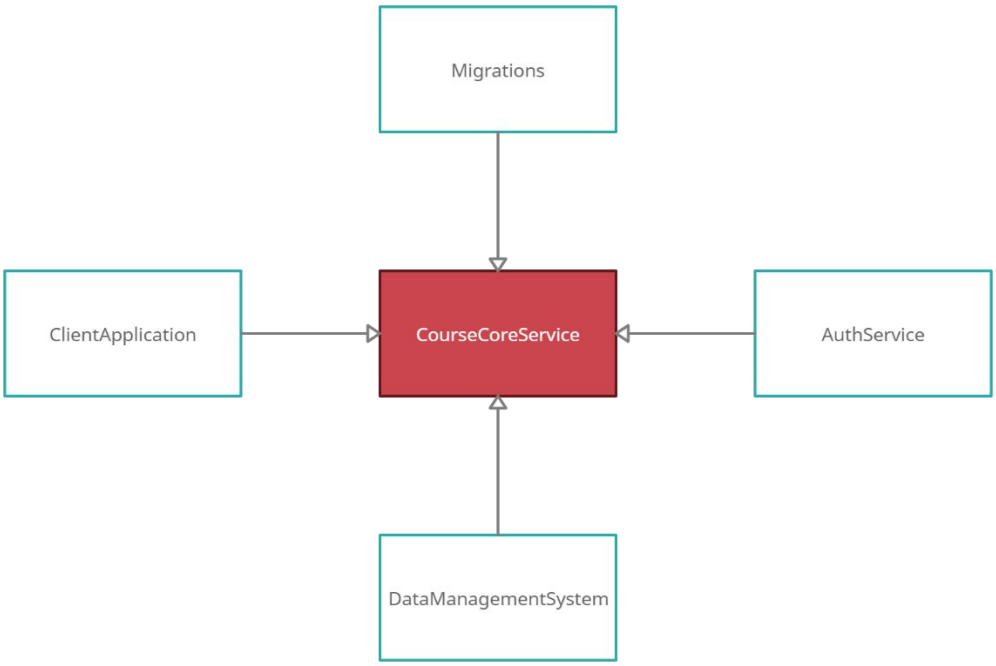


Рисунок 26

З діаграми компонент представлених на рисунку 26, бачимо що основною компонентою є CourseCoreService який відповідає за бізнес логіку проекту, всі направлення залежностей (стрілки) направлені в сторону до цієї компоненти – вона є ядром системи, все інше деталі, які легко можуть бути замінені на інші.

AuthService – це сервіс, що відповідає за авторизацію та автентифікацію користувачів, детальніше про цей сервіс йтиметься в наступних частинах.

DataManagementSystem – це компонент який відповідає за доступ до даних. Ядровий сервіс здатен сам вибирати з якою базою даних він хоче працювати, ця інформація передається до компоненти даних, яка в результаті розуміє з якою базою даних вона працюватиме. Таким чином ми не залежимо від якоїсь конкретної бази даних.

З причини незалежності від бази даних, був створений компонент міграцій – Migrations, який в залежності від обраної бази даних створює міграції до неї, і таким чином підтримує стан бази завжди актуальним.

І останній, але не менш важливий компонент – клієнтська частина, яка також залежить від нашого ядрового сервісу, тобто в будь який момент може бути замінена на іншу.

В якості хмари були використані сервіси компанії Amazon – AWS.

В якості бази даних була обрана реляційна база даних – PostgreSQL.

2.3.2 Сервіс авторизації та автентифікації

В якості системи зберігання та управління користувачами був обраний сервіс AWS Cognito, так як існує публічний API в .Net для роботи з цим сервісом, і він покриває всі можливі варіанти роботи системи для авторизації та автентифікації.

За допомогою консолі AWS був створений пул користувачів, в якому були налаштовані основні параметри, а саме:

- Користувачі можуть використовувати лише свої електронні пошти в якості логіна;
- Додаткові параметри при зберіганні користувача – його часовий пояс та мова;
- При реєстрації відправляється лист з підтвердженням електронної пошти;
- При реєстрації та авторизації була додана перевірка електронної пошти – тільки користувачі з доменом knu.ua здатні користуватись системою, даний функціонал був реалізований з використанням лямбда функцій AWS, код наведений на рисунку 27.

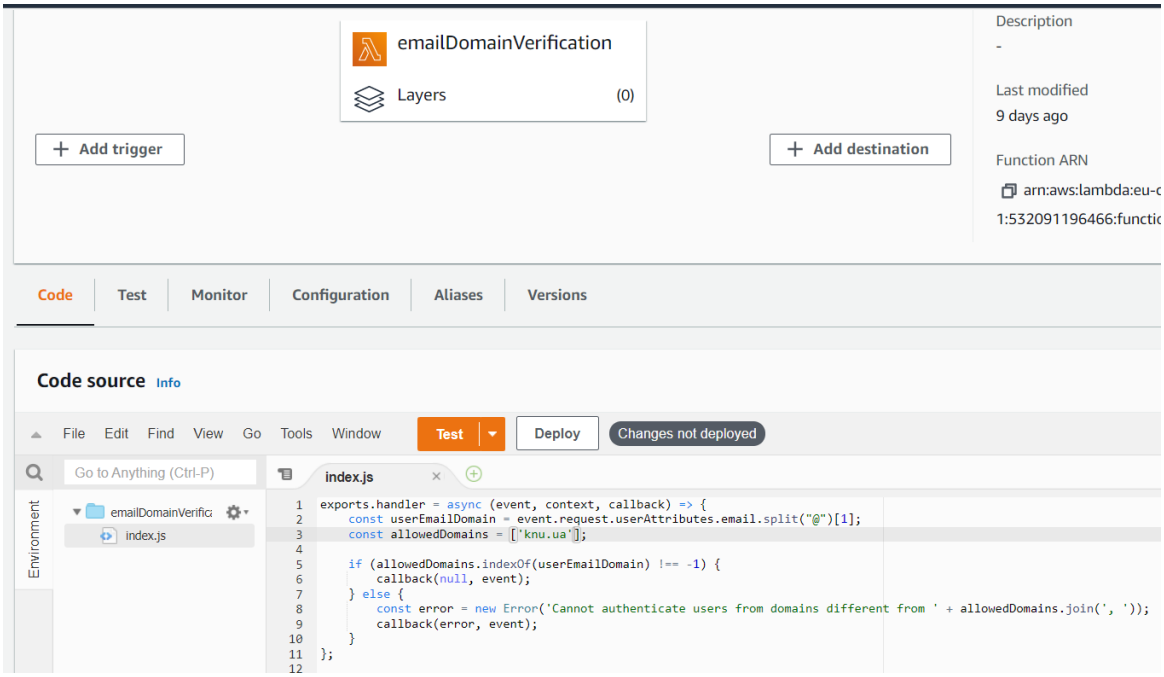


Рисунок 27

На рисунку 28 представлена діаграма прецедентів, на якій зображено можливості користувача. Користувач здатен увійти в систему, при цьому сервіс поверне йому куку, в яка буде використовуватись для автентифікації наступних запитів. Користувач може зареєструватись, при цьому підтвердити реєстрацію кодом, що буде відправлений на електронну пошту, і така ж процедура для дії відновлення пароля.

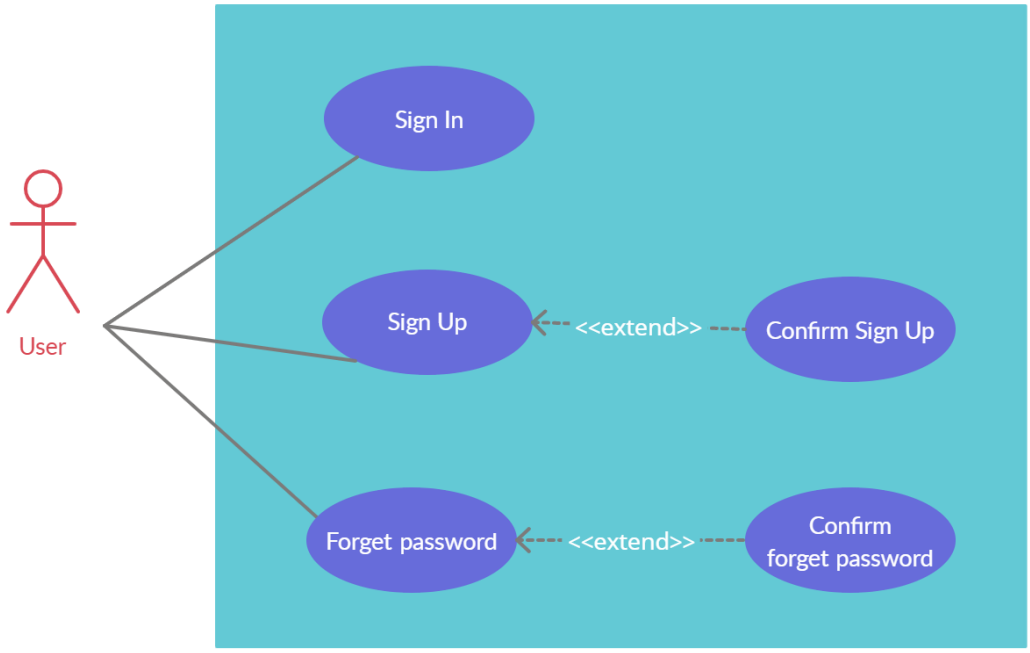


Рисунок 28

Взаємодія сервісу з сервісом AWS Cognito відбувається за допомогою функціоналу представленому в публічному API та бібліотеки Microsoft.AspNet.Identity, використовуючи класи UserManager<CognitoUser> та CognitoUserPool.

Також в рамках цього сервісу було реалізовано API для отримання інформації про користувача, а саме отримання інформації про ролі користувача, його мову, часовий пояс, а також можливість отримання Bearer токену, що дає можливість використовувати його в наступних авторизаціях.

2.3.3 Компонент доступу до даних

Компонент доступу до даних в своєму публічному API пропонує користувачеві п’ять запитів, чотири з яких відповідають за CRUD операції та один для агрегації даних. Публічний API відповідає OpenAPI специфікації.

Всі запити підтримують локалізацію, тобто враховується мова користувача, і результат повертається в правильній мові, а також для всіх запитів користувач повинен бути автентифікований в системі та рольова модель повинна дозволяти йому виконувати ту чи іншу операцію, наприклад читати дані з таблиці лабораторних робіт.

Варто згадати, що важливим аспектом побудови програми була її універсальність та легкий супровід в майбутньому, тому для доступу до даних використовується Entity Framework Core та LINQ запити, що формуються в запити до бази даних.

Архітектурним підходом до реалізації цього компоненту було використання патерну Repository. Був створений базовий типізований клас репозиторія, що реалізував всю логіку, таким чином для створення нового репозиторію нам потрібно тільки наслідувати базовий, також базовий репозиторій дає нам змогу заміщати такі методи, як до та після створення нового запису, до та після оновлення запису та до та після видалення запису. Ці методи дають простір для реалізації бізнес логіки. Підхід до реалізації цього компоненту зображений на рисунку 29.

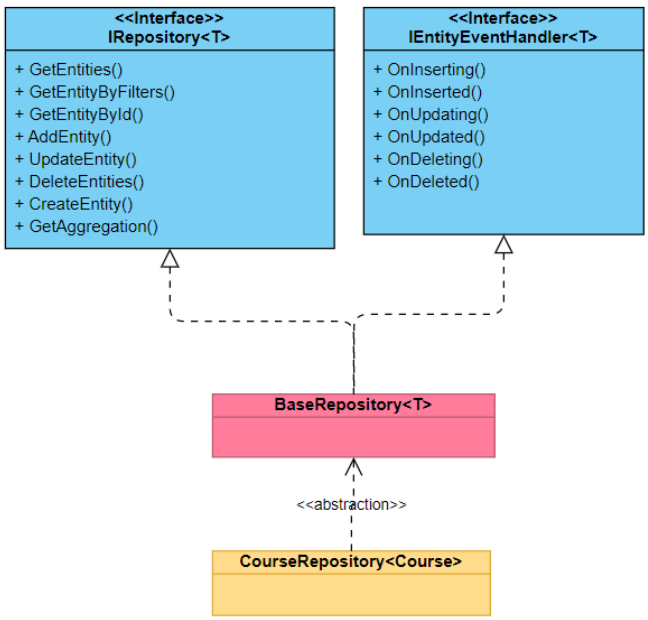


Рисунок 29

Особливо цікавими є запити на отримання даних та агрегацій. Саме в цих запитах використовуються фільтри, застосовуються колонки які б ми хотіли отримати чи агрегації даних. При запиті користувач відправляє серверу JSON файл, в якому перераховує фільтри, колонки чи агрегації. Важливо було, щоб система працювала швидко і чинила мінімальне навантаження на сервер, тому важливо було для таких запитів створити універсальний механізм, що буде перетворювати такі запити з початкового формату (рисунок 30) в LINQ запити. Для цього використовувався ExpressionTree. На льоту генерувались LINQ запити, таким чином ми не діставали всі записи з бази даних, навантажуючи оперативну пам’ять, а потім фільтрували їх, ми одразу генеруємо єдиний запит.


```
{
    "filters": {
        "expression": {
            "columnPath": "Id",
            "value": "5188500a-46fc-4c4f-aae1-1fd5c792ee49",
            "comparisonType": 0
        },
        "logicalOperator": 0,
        "filters": []
    },
    "columns": {
        "columnNames": [
            "Id",
            "Name",
            "Code"
        ],
        "relatedColumns": [
            {
                "columnNames": [
                    "Id",
                    "Name",
                    "Code"
                ],
                "columnName": "Parent",
                "relatedColumns": []
            }
        ]
    },
    "pageSize": 1,
    "pageIndex": 0
}
```


Рисунок 30

2.3.4 Бізнес логіка та об’єктна модель

Об’єктна модель проекту представлена на діаграмі бази даних – рисунок 31. Основними сутностями є контакти, тобто студенти та викладачі, курси, групи, лабораторні роботи. В деяких таблиць є зв’язані з ними таблиці локалізацій, що зберігають в собі інформацію на різних мовах. Для таблиць до яких потрібен доступ були реалізовані репозиторії, що імплементують базовий.

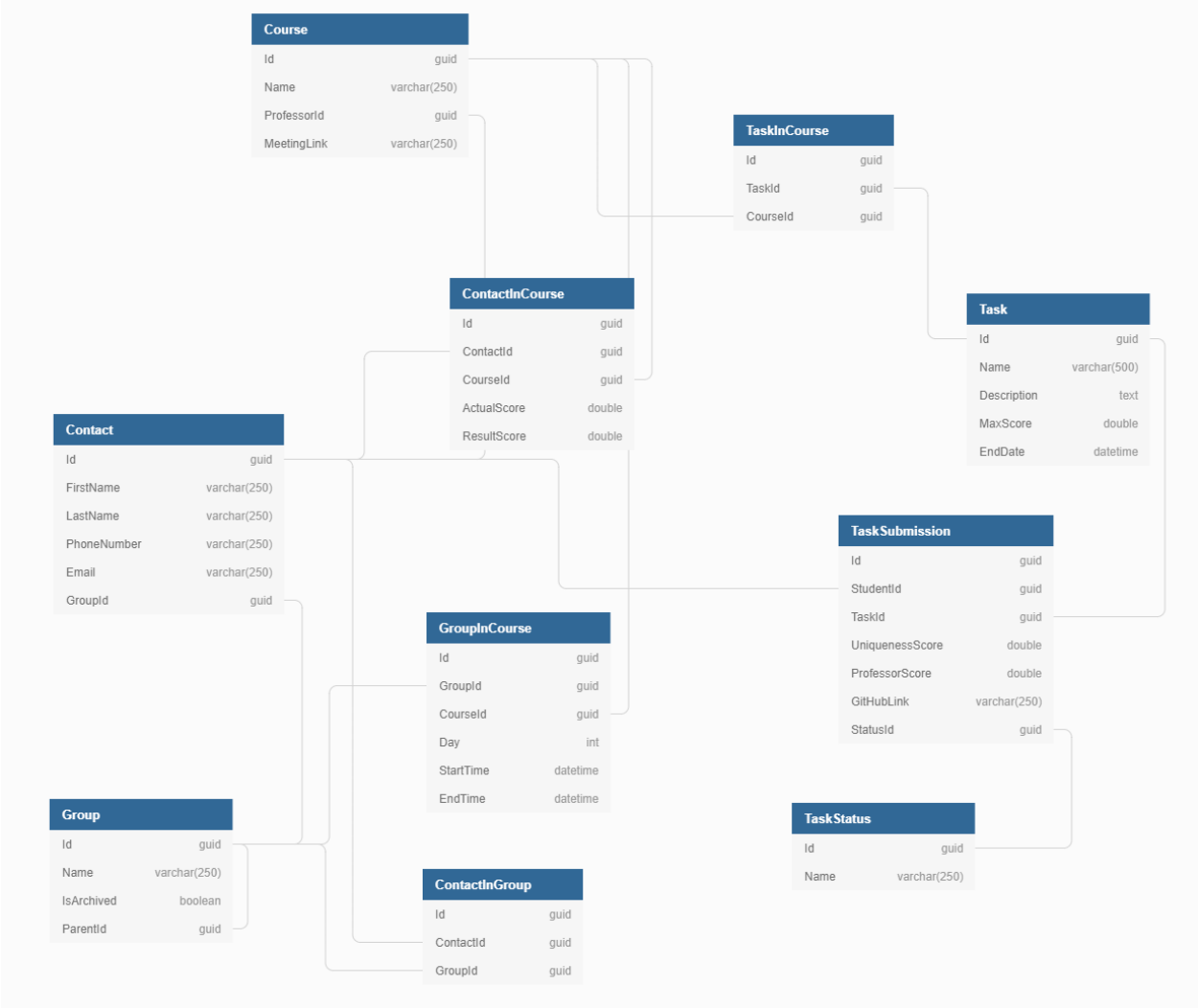


Рисунок 31

В рамках бізнес логіки була реалізована рольова модель, а саме в системі існує три ролі – адміністратор, студент та викладач. У адміністратора є права на будь-які операції, у викладача є права на створення, перегляд та оновлення таких таблиць, як групи, лабораторні роботи, контакти. Студент може тільки переглядати свої оцінки, відправляти лабораторні роботи на перевірку.

При відправленні лабораторної роботи студент вказує посилання на GitHub репозиторій, далі система створює новий запис в таблиці TaskSubmission, і запускається виконання методу OnInserted для створеного об’єкту TaskSubmission, де запускається підрахунок унікальності коду, який дістається з GitHub репозиторію за допомогою GitHub API.

2.3.5 Розгортання додатку та його супровід

Додаток розгортається в AWS EC2 сервісі на Linux машині та працює на Kestrel сервері. Зараз всі сервіси працюють на одній машині, на якій працює і база даних, але при зростанні навантаження, всі сервіси можна розмістити на окремих EC2 машинах, таким чином отримаємо мікросервісну архітектуру і горизонтальне масштабування.

Також як зворотній проксі-сервер використовується nginx, для перенаправлення запитів на правильні домени та порти. Вся система працює по протоколу HTTPS для зовнішнього світу, що вимагає генерацію SSL сертифікатів.

Процес розгортання додатку описується в yaml файлі – і розгортається за допомогою docker команди docker compose. Всі сервіси мають власні docker файли, які описують процес запуска цих сервісів, їх конфігурацію (змінні середовища), порти які відкриті для комунікації, зазвичай це 443 та 80 для роботи по протоколах HTTPS та HTTP відповідно.

В рамках конфігурації машин EC2, за допомогою груп захисту, ми можемо конфігурувати протоколи та порти по яких машина доступна для певного IP, тобто повністю конфігурувати захист середовища.

Таким чином, ми отримуємо додаток який розгортається та конфігурується за допомогою кількох docker команд, та опцій AWS.

ВИСНОВКИ

У процесі створення кваліфікаційної роботи для управлінням лабораторними роботами було зроблено наступне:

- Досліджено та порівняно існуючі на ринку додатки для перевірки програмного коду на плагіат.
- Досліджено принципи «чистої» архітектури програмних додатків.
- Спроектовано архітектуру додатка та реалізовано її в виді серверної частини веб додатку.
- Розроблено процес розгортання та супроводу додатка за допомогою Docker.
- Застосовано сервіси AWS для розміщення веб додатку.
- Розроблено додаток для перевірки вихідного коду на плагіат, та проведено його порівняння з уже існуючими.
- Виявлено, що розроблені алгоритми для перевірки коду на унікальність, в певних ситуаціях дають результати кращі аніж ринкові аналоги, проте в багатьох ситуаціях уступають конкурентам.

В подальшому планується покращення алгоритмів для пошуку дублів, за допомогою використання нейронних мереж та машинного навчання. Для веб додатку планується створення інтеграцій з існуючими системами, такими як GoogleClassroom, для можливості поєднання цих систем та простішого переходу з однієї системи на іншу. При зростанні навантаження на веб сервер додатку, можливий легкий перехід на мікросервісну архітектуру.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. A Machine Learning Based Tool for Source Code Plagiarism Detection. // International Journal of Machine Learning and Computing. – 2011. – №4. – С. 337–343.
2. Source Code Plagiarism Detection ‘SCPDet’: A Review. // International Journal of Computer Applications. – 2014. – №17. – С. 18–25.
3. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. – 2009. – С. 471–495.
4. Heres D. Source Code Plagiarism Detection using Machine Learning / Daniël Heres. – Utrecht: Utrecht University, 2017. – 35 с.
5. Introduction to ASP.NET Core [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-Us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-5.0.
6. Martin R. Clean Architecture / Robert Martin., 2018. – 422 с.
7. Amazon Cognito Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://docs.aws.amazon.com/cognito/?id=docs_gateway.
8. AWS Lambda Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://docs.aws.amazon.com/lambda/?id=docs_gateway.
9. Amazon Elastic Compute Cloud Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://docs.aws.amazon.com/ec2/?id=docs_gateway.