Вінницький національний технічний університет
(повне найменування вищого навчального закладу)
Факультет комп'ютерних систем і автоматики
(повне найменування інституту, назва факультету (відділення))
Кафедра автоматизації та інтелектуальних інформаційних технологій
(повна назва кафедри (предметної, циклової комісії))

Пояснювальна записка

до бакалаврської дипломної роботи
бакалавр
(освітньо-кваліфікаційний рівень)

на тему: «Мікросервісна система складання розкладу Вінницького Національного Технічного Університету»

Виконав: студент 4 курсу, групи 1СІ-16б
Напрям підготовки
151 – «Автоматизація та комп'ютерно-інтегровані технології»
(шифр і назва напряму підготовки, спеціальності)

Мамашвілі Л.О.
(прізвище та ініціали)
Керівник к.т.н. Довгалець С. М.
(прізвище та ініціали)

Рецензент к.т.н. Ковалюк О.А.
(прізвище та ініціали)

Вінниця 2020

Вінницький національний технічний університет

Факультет комп'ютерних систем і автоматики

Кафедра автоматизації та інтелектуальних інформаційних технологій

Освітньо-кваліфікаційний рівень бакалавр

Спеціальність 151 – «Автоматизація та комп'ютерно-інтегровані технології»

ЗАТВЕРДЖУЮ
Завідувач кафедрою АІІТ
д.т.н., проф. Квєтний Р. Н.
«___»_____ 202_ р.

ЗАВДАННЯ

НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Мамашвілі Леоніду Олеговичу

1. Тема роботи: Мікросервісна система складання розкладу Вінницького Національного Технічного Університету
   Керівник роботи: к.т.н., доцент каф. АІІТ Довгалець С.М.
   Затверджені наказом ВНТУ від «___»_____ 202_ року №___
2. Строк подання студентом роботи: «___» _____ 202_ р.
3. Вихідні дані до роботи: код програми, мови програмування *C# та TS*
4. Зміст розрахунково-пояснювальної записки: огляд предметної області; розробка архітектури програмної системи; розробка програмного забезпечення.
5. Перелік графічного матеріалу: Архітектура програмної системи; лістинг програмного забезпечення.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання отримав
1-3	Довгалець С.М., к.т.н., доцент каф. АІІТ		

7. Дата видачі завдання: «___»_____ 202_ р.

КАЛЕНДАРНИЙ ПЛАН

№ з	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Огляд предметної області		
2	Розробка структури бібліотеки		
3	Розробка програмного забезпечення		
4	Оформлення пояснювальної записки і графічного матеріалу		
5	Попередній захист роботи		
6	Остаточний захист роботи		

Студент _____ Мамашвілі Л. О.

(підпис) (прізвище та ініціали)

Керівник роботи _____ Довгалець С. М.

(підпис) (прізвище та ініціали)

АНОТАЦІЯ

В даній роботі розглянуто розробку програмного забезпечення для створення, редагування, перегляду та видалення розкладу ВНТУ на основі мікросервісній архітектурі. Розроблений програмний продукт для роботи з розкладом ВНТУ задовольняє вимогам поставленого завдання.

ABSTRACT

This paper considers the development of software for creating, editing, viewing and deleting the schedule of VNTU on the basis of microservice architecture. The developed software product for work with the schedule of VNTU satisfies the requirements of the set task.

ЗМІСТ

ВСТУП

С кожним днем розробка програмного забазпечення стає все складніше, дорожче, потребує більше часу, уваги та спеціалистів. Запити бізнесу потребують швидких змін у программному коді та швидкого оновлення програмного забезпечення у користувачів. А кількість користувачів, в свою чергу, лише зростає. Що приводить до зростання вимог до відмовостійкості програмних систем.

Існує велика кількість різних паттернів для проеєктування програмного забезпечення. Ці паттерни вірішують проблеми конкретного модуля системи, але не всієї системи. При великому навантажені системи виникає проблема як це навантаження розподілити. Саме тому на етапі проєктування програмної системи необхідно обрати архітектуру модулів яка підійде найкраще. На даний момент існує дві архітектури: монолітна та мікросервісна.

Довгий час популярною архітектурою додатка був моноліт. Він легкий в розуміні та використовує ресурси лише однієї машини. Але у монолітної архітектури легко знайти багато недоліків. Монолітні додатки можуть бути успішними, але все більше розробників відмовляються від них, особливий вплив має те, що все більше додатків розгортаються в хмарних сервісах. Будь-які зміни, навіть невеликі, вимагають компіляції коду і розгортання всього моноліту. З плином часу, стає важче зберігати гарну модульну структуру, зміни логіки одного модуля мають тенденцію впливати на код інших модулів. Масштабувати доводиться весь додаток, навіть якщо це потрібно тільки для одного модуля цього додатка.

На щастя, розвиток створення програмного забезпечення невпинно йде вперед. Альтернативою монолітної архітектури виступає мікросервісна архітектура.

Актуальність даної роботи полягає в тому, що все більше і більше додатків переходять на мікросервісну архітектуру або на етапі проектування обирають саме мікросервіси.

Мета і завдання дослідження - це розробка програмної системи на основі мікросервісів для створення, редагування, перегляду або видалення розкладу для ВНТУ. Ця система в свою чергу буде розбита на декілька мікросервісв.

Об'єкт дослідження – це процес розробки програмного забезпечення на основі мікросервісній архітектурі а також його реалізація.

Предмет дослідження – це переваги та недоліки мікросервісної архітектури.

Методи дослідження – При виконанні роботи використовувалися такі методи досліджень: методи аналізу та синтезу для розробки програмного забезпечення на основі мікросервіснох архітектури. Основним засобом для розробки програмного забезпечення є мови програмування C# та TypeScript

Задачі дослідження. Основні необхідні завдання налічують розв'язок таких задач, як:

– аналіз існуючих недоліків та переваг монолітної архітектури;

– аналіз існуючих недоліків та переваг мікросервісної архітектури;

– аналіз існуючих програмних систем для роботи з розкладом;

– розробка алгоритмічного забезпечення для розробки мікросервіса;

– розробка програмного забезпечення для створення, редагування, перегляду та видалення розкладу ВНТУ.

Практичним значенням одержаних результатів даної роботи є розроблене алгоритмічне та програмне забезпечення програмного забезпечення для створення, редагування, перегляду та видалення розкладу ВНТУ.

Апробація За результатами даної роботи опубліковано тези доповіді [6] на XLIX науково-технічній конференції Факультету комп'ютерних систем і автоматики (м. Вінниця, 2020).

1 СУЧАНИЙ СТАН ПИТАННЯ ТА ОСНОВНІ ЗАДАЧІ РОБОТИ

1.1 Загальні відомості про мікросервісну архітектуру

«Мікросервіси» - ще один новий термін в великій сфері розробки програмного забезпечення. За останні кілька років з'явилися безліч проектів, що використовують саме цей стиль, і результати досі були досить позитивними. Настільки, що для більшості розробників цей стиль стає основним стилем розробки програмного забезпечення. На жаль, існує не так багато інформації, яка описує, чому ж є мікросервіси і як застосовувати їх. Якщо коротко, то архітектурний стиль мікросервісов - це підхід, при якому єдине додаток будується як набір невеликих сервісів, кожен з яких працює у власному процесі і спілкується з іншими використовуючи легковажні механізми, як правило HTTP (рисунок 1.1). Ці сервіси побудовані навколо бізнес-потреб і розгортаються незалежно з використанням повністю автоматизованої середовища. Існує абсолютний мінімум централізованого управління цими сервісами. Самі по собі ці сервіси можуть бути написані на різних мовах і використовувати різні технології зберігання даних.

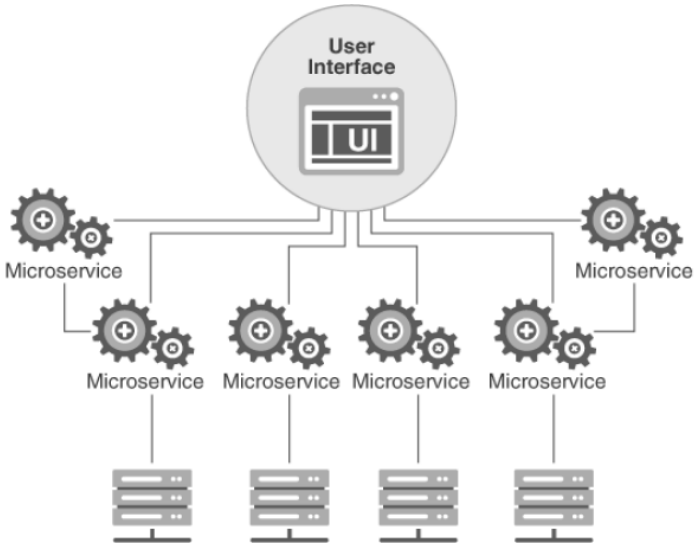

Рисунок 1.1 – Приклад діаграми додатку на мікросервісній архітектурі

1.2 Переваги мікросервісної архітектури

Як і будь-який підхід, ідея розвязку задачі або архітектура мають свої переваги та недоліки. С переваг можна виділити автономність і незалежність, можливості застосування різних мов програмування, легке масштабування, стабільність і керованість та децентралізація даних але в свою чергу при невірній реалізації це може стати серйозною проблемою для збереження інформації

1.2.1 Автономність і незалежність

Мікросервісная архітектура дозволяє створювати незалежні крос-функціональні команди, націлені на вирішення конкретного бізнес-завдання. Ці команди самостійні і максимально ефективні. Кожен елемент побудованої таким чином програмної системи виконує потрібну функцію, тому він може бути розгорнутий незалежно від інших сервісів.

1.2.2 Можливість застосування різних технологій

Через мікросервіси можна об'єднувати різні технології, вибираючи кращі з можливих рішень. А використання стандартних протоколів взаємодії (HTTP-виклики через API, брокери повідомлень) дозволяє створювати мікросервіси на різних мовах програмування і використовувати різні технології обробки та зберігання даних.

До того ж до вищесказаного, мікросервіси здатні функціонувати на будь-якому пристрої, в хмарних середовищах і на локальних серверах. Це дає можливість не залежити від операційної системи чи налаштувань середовища в якому працює мікросервіс.

1.2.3 Легка масштабованість

Безумовно, мова йде про нові можливості масштабованості, так як в разі потреби розгортання не буде потрібно масштабувати всю систему і розбирати її дощенту - досить буде внести зміни лише на конкретній ділянці програмної системи. Це робить мікросервіси дуже економічними, швидкими для оновлення, а кінцевий користувач взагалі не помітить зупинки програми. Для бізнеса масштабованість грає велику роль, це основна перевага мікросервісної архітектури над монолітною.

1.2.4 Стабільність і керованість

Так як мікросервіси є незалежними один від одного, стабільність системи підвищується. Збої і дефекти в одному мікросервісі не вплинуть на роботу інших, саме тому програмна система буде функціонувати з мінімальними простоями. Крім того, замість монолітного і надскладного масиву системи ми отримаємо окремі компоненти з більш керованою архітектурою, де кожен елемент буде відповідати за свою функцію.

Залишається додати, що мікросервіси можна перепрофілювати для інших завдань після початкового запуску, що забезпечить повторне використання готових рішень. Це дозволяє не витрачати додаткові кошти на розробку нового мікросервіса.

1.2.5 Децентралізація даних

Для збереження інформації для кожного мікросервіса, як правило, створюється окрема база даних. Це дозволяє ізолювати дані одного мікросервіса від іншого але це породжує неузгодженність даних. Наприклад, існує додаток в якому при реєстрації створюється 3 сутності: користувач, профіль і рахунок. Раптово на якомусь моменті із різних причин обробка

сутності рахунку зупиняється. Як повернути назад зміни, якщо дані розподіляються за різними мікросервісам? Про це слід пам'ятати на етапі проектування програмної системи.

1.3 Недоліки мікросервісної архітектури

Розробка розподілених програмних систем може бути важким завданням. Під цим мається на увазі, що всі компоненти тепер незалежні сервіси - потрібно дуже уважно обробляти запити які проходять між модулями. Можливий сценарій, коли один модуль не відповідає, змушуючи зупинити виконання іншого модулю. Доводитися використовувати додаткові рішення та писати додатковий код, для того щоб уникнути збою всієї програмної системи.

Також, може виникнути проблема зі розподілом та зберіганням інформації. Децентралізація відповідальності за дані серед мікросервісов впливає на те, як ці дані змінюються. Звичайний підхід до зміни даних полягає в використанні транзакцій для гарантування узгодженості при зміні даних, що знаходяться на декількох ресурсах. Такий підхід часто використовується в монолітних додатках.

Подібне використання транзакцій гарантує узгодженість даних, але призводить до суттєвої часової залежності, яка, в свою чергу, призводить до проблем при роботі з безліччю сервісів. Розподілені транзакції неймовірно складні в реалізації і, як наслідок, мікросервісна архітектура надає особливого значення координації між сервісами без використання транзакцій з явним позначенням того, що узгодженість даних може бути тільки підсумкової і виникають проблеми вирішуються операціями компенсації.

Управління неузгодженостями подібним чином - новий виклик для багатьох команд розробки, але це часто відповідає практикам бізнесу. Часто компанії прагнуть якомога швидше реагувати на дії користувача і мають процеси, позвояют скасувати дії користувачів в разі помилки. Компроміс

варто того доти, поки вартість виправлення помилки менше вартості втрат бізнесу при використанні сценаріїв, які гарантують узгодженість.

1.4 Переваги та недоліки монолітної архітектури

Великою перевагою моноліту є те, що його легше реалізувати (рисунок 1.2). У монолітної архітектури ви можете швидко почати реалізовувати свою бізнес-логіку, замість того щоб витрачати час на роздуми про взаємодії між процесами.

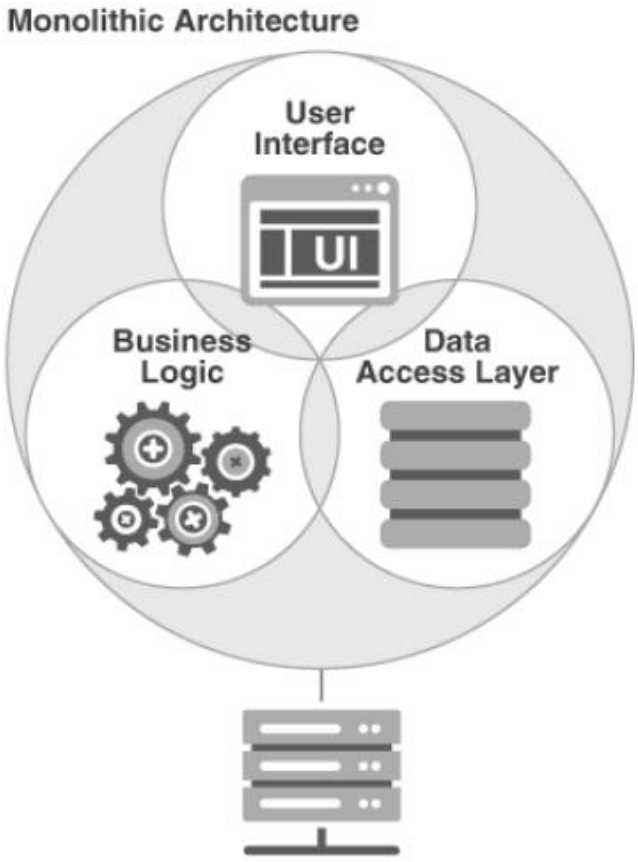

Рисунок 1.2 – Приклад діаграми додатку на монолітній архітектурі

Говорячи про операції, важливо згадати, що моноліт простий в розгортанні. Для розгортання ви можете використовувати скрипт, що завантажує ваш модуль і запускає додаток.

Також можна відмітити ще одну перевагу монолітної архітектури це те що багато програм зазвичай використовують одне й те саме - типовий функціонал: логування, права доступу. Коли всі процеси проходять через один додаток, дуже легко приєднати типовий функціонал до компонентів. Також є перевага в продуктивності, тому що загальний доступ до пам'яті швидше ніж комунікація між процесами.

Але не зважаючи на це, монолітна архітектура відстає в багаотьох моментах. Моноліти, як правило, в кінцевому стані дуже відрізняються від плановоного стану. Тому що архітектурні правила були порушені і з часом компоненти зрослися і утворили велику систему в якії модулі тісно повязані між собою, моделі даних лише збільшують кількість зв'язків а процессорний час використовується на очікування відповіді від стороніх сервісів.

Це переродження уповільнює процес розробки: кожну майбутню функцію буде складніше розвивати. Через те що компоненти ростуть разом, їх також необхідно міняти разом. Створення нової функції може означати дотик до 5 різних місць: 5 місць, в яких вам потрібно написати тести; 5 місць, які можуть мати небажані побічні ефекти для існуючих функцій.

Масштабування може бути проблематичним, коли тільки однієї частини системи потрібні додаткові ресурси, адже в монолітної архітектури ви не можете масштабувати окремі частини вашої системи. А використання додаткових серверів для декількох модулів може коштувати надто дорого для бізнесу.

У моноліті практично немає ізоляції. Проблема або помилка в модулі може уповільнити або зруйнувати всю роботу додатоку. Наприклад, ящко виникає проблема лише с авторизацією, зупинитись може весь додаток незалежно від природи помилки.

Будівництво моноліту часто протікає за допомогою вибору основи. Відключення або оновлення вашого початкового вибору може бути складним, тому що це має бути зроблено відразу і для всіх частин вашої системи.

1.5 Розгортання систем с точки зору DevOps методологій

DevOps - це поєднання культурних принципів, підходів і засобів, яке покращує здатність компаній створювати додатки і сервіси на високій швидкості. З DevOps розробка і оптимізація продуктів відбувається швидше, ніж при використанні традиційних процесів роботи над програмним забезпеченням і управління інфраструктурою. Завдяки такій швидкості компанії можуть підвищити рівень обслуговування клієнтів і більш ефективно конкурувати на ринку. Принцип методолгії DevOps зображено на Рисунку 1.3

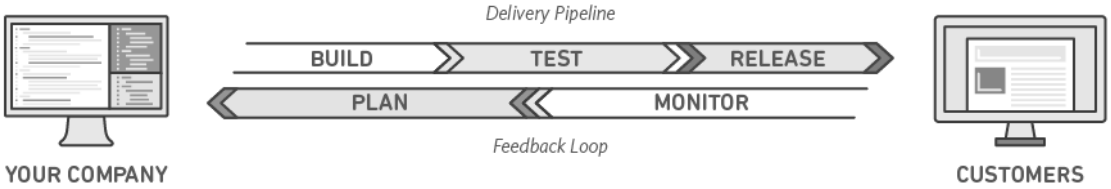

Рисунок 1.3 – Принципи метедолгії DevOps

У моделі DevOps кордони між групами розробки та експлуатації зникають. Іноді ці дві групи об'єднуються в одну загальну, де інженери працюють над усім життєвим циклом додатка - від розробки і тестування до розгортання і експлуатації - і розвивають цілий ряд навичок, не обмежуючись вузькою спеціалізацією.

У деяких моделях DevOps групи контролю якості та безпеки теж більш активно взаємодіють з групами розробки та експлуатації на протязі всього життєвого циклу програми. Якщо безпека є пріоритетом для всіх учасників групи DevOps, такий підхід іноді називають DevSecOps.

Ці групи використовують спеціальні методи для автоматизації процесів, які традиційно виконувалися повільно і вручну. Вони застосовують набір технологій і інструменти, які допомагають працювати з додатками і продовжувати розвивати їх зі збереженням високої швидкості і надійності. За допомогою цих інструментів технічні фахівці можуть самостійно вирішувати

такі завдання, для яких зазвичай потрібна допомога інших груп (наприклад, розгортання коду або ініціалізацію інфраструктури), і це ще більше підвищує швидкість спільної роботи.

Один з основних методів полягає у випуску невеликих оновлень через дуже короткі проміжки часу. Саме так компанії швидше пропонують нові можливості своїм клієнтам. Оновлення, як правило, носять інкрементальний характер, а не випадковий, як це відбувається при традиційному підході до релізів. Часті, але невеликі оновлення роблять кожне розгортання менш ризикованим. Вони допомагають швидше вносити виправлення, оскільки групи можуть ідентифікувати останнім розгортання, яке викликало помилку. Незважаючи на різну частоту і розмір оновлень, компанії, що використовують модель DevOps, розгортають їх набагато частіше, ніж ті, хто застосовують традиційні методи розробки програмного забезпечення.

Крім того, щоб зробити програми більш гнучкими і швидше впроваджувати інновації, можна використовувати архітектуру мікросервісов. Така архітектура розділяє великі і складні системи на прості самостійні проекти. Додатки розбиваються на безліч окремих компонентів (сервісів), кожен з яких має одну мету або можливість і працює незалежно від інших сервісів, а також додатки в цілому. Така архітектура скорочує витрати на координацію оновлень додатків, до того ж, коли кожному сервісу відповідає відповідальна за нього невелика крос-функціональна група, компанії можуть працювати швидше.

Однак поєднання мікросервісов і підвищеної частоти релізів значно збільшує число розгортання, в зв'язку з чим можуть виникати проблеми на рівні експлуатації. Тому необхідні такі методи DevOps, як безперервна інтеграція і безперервна доставка, які вирішують ці проблеми і дозволяють компаніям доставляти поновлення швидко, безпечно і надійно. Методики автоматизації інфраструктури - інфраструктура як код і управління конфігураціями - допомагають підтримувати гнучкість обчислювальних ресурсів і адаптуватися до частих змін. А моніторинг і ведення журналів

дозволяють інженерам відстежувати ефективність додатків та інфраструктури, щоб швидко реагувати на проблеми.

Концепція DevOps (рисунок 1.4) пропонує вирішувати цю проблему за допомогою додатка принципів Agile не тільки до розробки та тестування, а й до процесів експлуатації ПО, тобто до розгортання і підтримки. Таким чином, популярність DevOps виникла, в тому числі завдяки поширенню Agile-практик, орієнтованих на прискорення процесів поставки готового продукту і збільшення кількості випущених версій. Крім того, додатковим драйвером розвитку девопс стала мікросервісная архітектура, коли система складається з набору окремих слабосвязанних модулів, реалізація кожного з яких знаходиться в зоні відповідальності однієї людини, який розробляє, тестує і розгортає ПО. Завдяки невеликому розміру кожного модуля (сервісу), його архітектура може створюватися шляхом безперервного рефакторінга, що зменшує трудомісткість попереднього проектування і дозволяє постійно випускати нові релізи програмного продукту.

Рисунок 1.4 - Концепція DevOps як перетин розробки, експлуатації і тестування

З точку зору методології DevOps розгортання монолітних програмних є більш простим завданням ніж розгортання мікросервісних систем. Адже моноліт це лише один додаток і він не потребує великої концентрації. Але на відміну від мікросервісв цей стиль не дає легкої масштатобаності, це одна із причин чему DevOps методологія та мікросервісна архітектура дуже пов'язані.

1.6 Огляд та аналіз існуючих технічних рішень

Так як запропоноване роботою програмне забезпечення націлене для вирішень задач державного навчального закладу, огляд та аналіз буде виконуватись в порівнянні с іншим програмним забезпеченням для державних навчальних закладів. На даний момент можливо порівняти з додатками наступних державних унвіерситетів: Вінницький Національний Технічний Університет, Київський національний університет імені Тараса Шевченка та Київський політехнічний інститут імені Ігоря Сікорського. За допомогою аналізу запитів та відповідей до додатків що надають розклад навчального закладу, можно зробити висновок що Київський національний університет імені Тараса Шевченка (рисунок 1.5) та Київський політехнічний інститут імені Ігоря Сікорського (рисунок 1.6) використовують монолітну архітектуру. Для аналізу запитів да відповідей від програмних систем використовувався інструментарій розробника веб-браузера Chrome. Цей інструментарій дозволяє переглядати статичні файли, консоль веб-сторінки, працювати з JavaScript кодом веб-сторінки, досліджувати використану пам'ять, продуктивність додатку, переглядати HTML та CSS код веб-сторінки, cookies значення та досліджувати типи запиту, тіло запиту. Також, існує можливість досліджувати тип відповіді та тіло відповіді. Завдяки цьому інструментарію і був проведений аналіз.

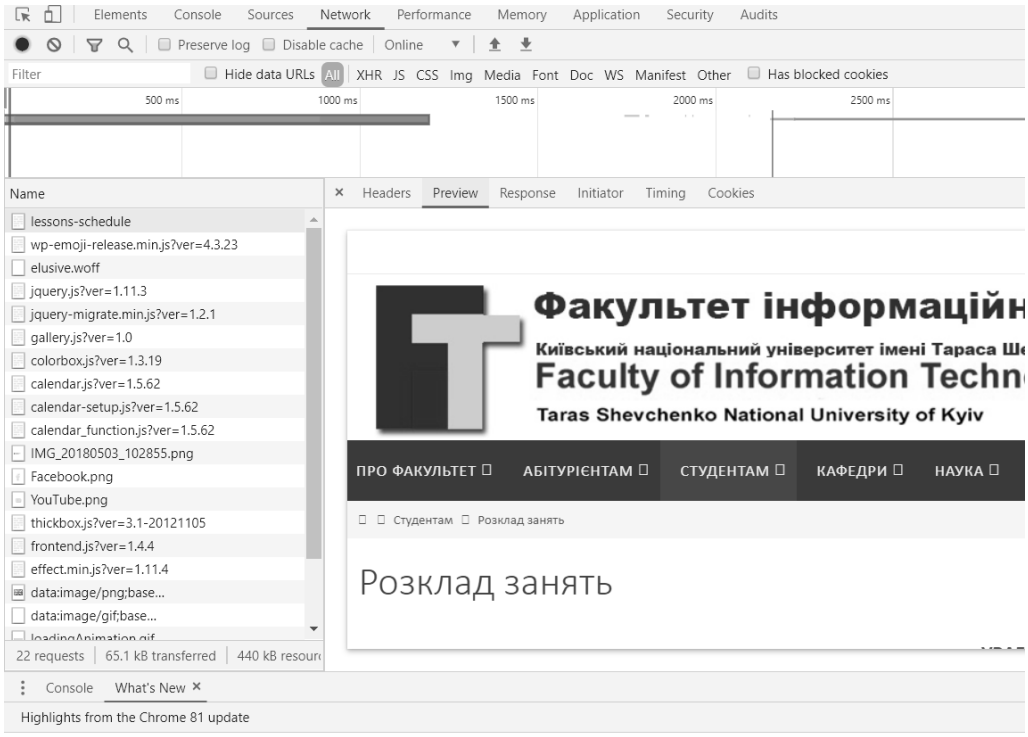

Рисунок 1.5 – Аналіз відповіді додатку Київського Національного університету імені Тараса Шевченка

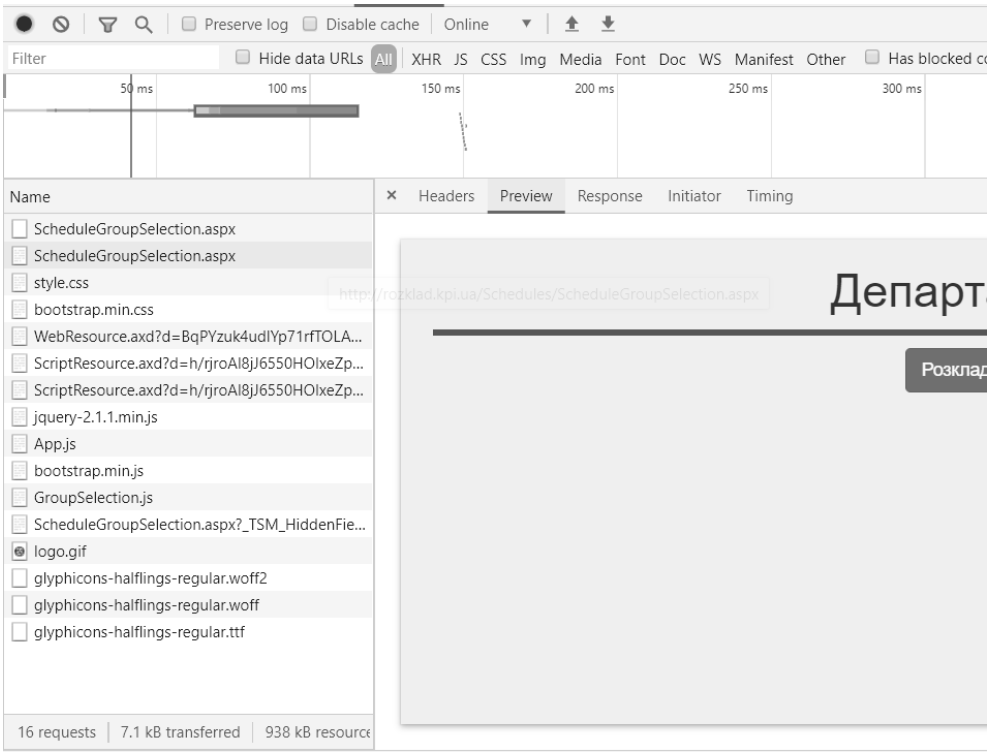

Рисунок 1.6 – Аналіз відповіді додатку Київського політехнічного інституту імені Ігоря Сікорського

Це означає що головна сторінка і сторінка розкладу працює на одному сервісі. В свою чергу це означає, цим додаткам властиві всі недоліки монолітної архітектури. Схоже, що додаток Вінницького Національного Технічного Університету працює на основі мікросервісній архітектурі. Але ці мікросервіси тісно пов'язані між собою, що характерно моноліту. Це тягне за собою проблеми відмовостійкості, адже якщо один модуль відмовляє в роботі не працює весь додаток.

Програмне забезпечення запропоноване дипломною роботою програмна система на основі мікросервісної архітектурі, що буде працювати незалежно від інших сервісів та систем. Дана програмна система може дати початок перехід на повністю мікросервісну архітектуру додатку JetIq. Так як для даної системи не існує бази даних груп, викладачів та інших даних необхідні для створення розкладу ця програмна система буде взаємодіяти з JetIq API. Завдяки невеликому розміру серверної частини перетворити цю систему на повністю ізольований сервіс у майбутньому не викличе серйозних проблем.

1.7 Висновки до розділу

У даному розділі було розглянуто основні переваги і недоліки мікросервісної та монолітної архітектури. А саме труднощі розгортання мікросервісв та моноліту, ізольованості окремих модулів, відмовостійкість, централізація чи децентралізація даних. На етапі проектування програмної систем слід звертати увагу на релевантність мов програмування, адже при виборі монолітної архітектури перехід на іншу мову програмування може зайняти надто довгий час та коштувати великих витрат. На основі ж мікросервісної архітектури можна комбінувати мови програмування для отримання найшвидших відповідей від системи. Адже, використання мови програмування, наприклад, Java підійде для розробки складної бізнес-логіки в великому мікросервісі, але в той же час мова Python найкраще підійде для створення нейроних мереж або для машиного навчання та штучного інтелекту.

Але не варто забувати і про переваги монолітної архітектури – а саме легкість розгортання, запуск лише на одному процесі, налаштування конфігурацій додатку, такі як паролі, права доступу до баз даних та інше.

Було досліджено та наведено основні поняття DevOps – методології. Адже поняття архітектури мікросервісов - це підхід до розробки програми у вигляді набору невеликих сервісів. Кожен сервіс працює в своєму власному процесі і обмінюється даними з іншими сервісами через детально продуманий інтерфейс, використовуючи простий механізм: як правило, програмний інтерфейс на базі HTTP. І для розгортання мікросервісної архітектури найкраще підійде DevOps – методологія.

Також в цьому розділі було проаналізовано додатки інших вищих навчальних національних закладів такі як КПІ та КНУ та виявлено недоліки характерні монолітним системам. На основі порівнянних архітектур можна зробити висновок, що мікросервісна архітектура більше підходить великим програмним системам ніж монолітна.

2 РОЗРОБКА СТРУКТУРИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Обґрунтування інструментів розробки

Розроблена програмна система складається із декількох додатків. А саме додаток користувацького інтерфейсу та додатку серверної частини (рисунок 2.1).

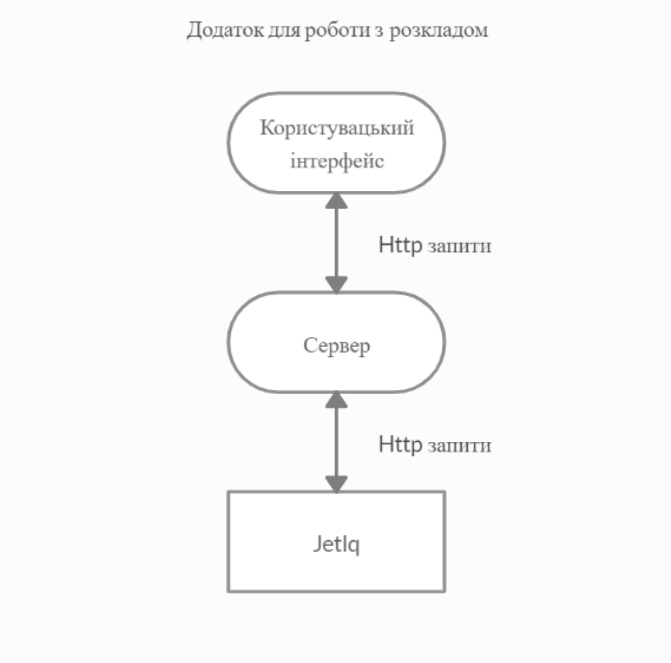

Рисунок 2.1 – Схема програмної системи

Для створення додатку користувацького інтерфейсу використовувалась мова програмування TypeScript та програмна платформа Angular. Для створеня додатку серверної частини була обрана мова програмування C# та програмна платформа ASP.NET core. Обрані програмні платоформи та мови програмування надають інструменти для роботи із REST API та HTTP запитами. Серверна частина буде обробляти запити користувацького інтерфейсу, створюватиме відповіди та буде тісно взаємодіяти з сервісами розкладу Вінницького Національного Тенхнічного Університету.

2.2 TypeScript та Angular як основа користувацького інтерфейсу

TypeScript представляє мову програмування на основі JavaScript. Розвиток TypeScript почався в кінці 2012 року. Хоча він зародився в компанії Microsoft, і його фактичним творцем є програміст Андерс Хейлсберг, так само відомий як творець таких мов як Delphi, C #, але даний проект відразу став розвиватися як OpenSource. І вже з самого початку нову мову став швидко поширюватися в силу своєї гнучкості і продуктивності. Чимало проектів, які були написані на JavaScript, стали переноситися на TypeScript. Популярність і актуальність ідей нової мови привела до того, що ряд з цих ідей в подальшому стануть частиною нового стандарту JavaScript. А нова версія одного з найпопулярніших фреймворків для веб - Angular повністю написана на TypeScript спільно компаніями Microsoft і Google. Однак, здавалося б, навіщо потрібен ще одну мову програмування для клієнтської сторони в середовищі веб, якщо з усією тією ж самою роботою прекрасно справляється і традиційний JavaScript, який використовується практично на кожному сайті, яким володіє безліч розробників і підтримка якого в співтоваристві програмістів досить висока. Але TypeScript це не просто новий JavaScript.

По-перше, слід зазначити, що TypeScript - це строго типізований і компільована мова, ніж, можливо, буде ближче до програмістам Java, C# та інших строго типізованих мов. Хоча на виході компілятор створює все той же JavaScript, який потім виповнюється браузером. Однак сувора типізація зменшує кількість потенційних помилок, які могли б виникнути при розробці на JavaScript.

По-друге, TypeScript реалізує багато концепції, які властиві об'єктно-орієнтованим мовам, як, наприклад, спадкування, поліморфізм, інкапсуляція і модифікатори доступу і так далі.

По-третє, потенціал TypeScript дозволяє швидше і простіше писати великі складні комплексні програми, відповідно їх легше підтримувати, розвивати, масштабувати і тестувати, ніж на стандартному JavaScript.

Angular являє собою фреймворк від компанії Google для створення клієнтських додатків. Перш за все він націлений на розробку SPA-рішень (Single Page Application), тобто односторінкових додатків. В цьому плані Angular є спадкоємцем іншого фреймворка AngularJS. У той же час Angular це не нова версія AngularJS, а принципово новий фреймворк. Angular надає таку функціональність, як двостороннє зв'язування, що дозволяє динамічно змінювати дані в одному місці інтерфейсу при зміні даних моделі в іншому, шаблони, маршрутизація і так далі. Однією з ключових особливостей Angular є те, що він використовує в якості мови програмування TypeScript. Але ми не обмежені мовою TypeScript. При бажанні є можливість писати програми на Angular за допомогою таких мов як Dart або JavaScript. Однак TypeScript все таки є основною мовою для Angular. Логотип цієї програмної платформи зображено на Рисунку 2.2.

Рисунок 2.2 – Логотип програмної платформи Angular

2.3 C# і ASP.NET перевірена часом платформа

На сьогоднішній момент мова програмування C# одна із найпотужніших, що швидко розвиваються і популярних мов в галузі створення прогармного забезпечення. На даний момент на цій мові пишуться найрізноманітніші програми: від невеликих комп'ютерних програм до великих веб-порталів і веб-сервісів, які обслуговують щодня мільйони користувачів.

У порівнянні з іншими мовами C# досить молода, але в той же час вона вже пройшла великий шлях. Перша версія мови вийшла разом з релізом Microsoft Visual Studio .NET в лютому 2002 року. Поточною версією мови є версія C # 8.0, яка вийшла у вересні 2019 року разом з релізом .NET Core 3.

C# є мовою з Сі-подібним синтаксисом і близький в цьому відношенні до C ++ і Java. Тому, якщо ви знайомі з одним з цих мов, то опанувати C # буде легше.

C# є об'єктно-орієнтованим і в цьому плані багато перейняв у Java і C++. Наприклад, C# підтримує поліморфізм, успадкування, перевантаження операторів, статичну типізацію. Об'єктно-орієнтований підхід дозволяє вирішити завдання з побудови великих, але в той же час гнучких, масштабованих і розширюваних додатків. І C# продовжує активно розвиватися, і з кожною новою версією з'являється все більше цікавих функцій, як, наприклад, лямбда, динамічне зв'язування, асинхронні методи.

Платформа ASP.NET Core представляє технологію від компанії Microsoft, призначену для створення різного роду веб-додатків: від невеликих веб-сайтів до великих веб-порталів і веб-сервісів. З одного боку, ASP.NET Core є продовженням розвитку платформи ASP.NET. Але з іншого боку, це не просто черговий реліз. Вихід ASP.NET Core фактично означає революцію всієї платформи, її якісна зміна.

Розробка над платформою почалася ще в 2014 році. Тоді платформа умовно називалася ASP.NET vNext. У червні 2016 року вийшов перший реліз платформи. А в грудні 2019 року побачила версія ASP.NET Core 3.1, яка власне

і буде охоплена в поточному керівництві. ASP.NET Core тепер повністю є opensource-фреймворком. Логотип цієї програмної платформи зображено на Рисунку 2.3

Рисунок 2.3 – Логотип програмної платформи ASP.NET Core

ASP.NET Core може працювати поверх крос-платформної середовища .NET Core, яка може бути розгорнута на основних популярних операційних системах: Windows, Mac OS, Linux. І таким чином, за допомогою ASP.NET Core ми можемо створювати крос-платформні додатки. І хоча Windows як середовище для розробки і розгортання програми досі превалює, але тепер вже ми не обмежені тільки цією операційною системою. Тобто ми можемо запускати веб-додатки не тільки на ОС Windows, але і на Linux і Mac OS. А для розгортання веб-додатки можна використовувати традиційний IIS, або крос-платформний веб-сервер Kestrel.

Завдяки модульності фреймворка всі необхідні компоненти веб-додатки можуть завантажуватися як окремі модулі через пакетний менеджер Nuget. Крім того, на відміну від попередніх версій платформи немає необхідності

використовувати бібліотеку System.Web.dll. ASP.NET Core включає в себе фреймворк MVC, який об'єднує функціональність MVC, Web API і Web Pages. У попередніх версії платформи дані технології реалізувалися окремо і тому містили багато дублюючої функціональності. Зараз же вони об'єднані в одну програмну модель ASP.NET Core MVC.

2.4 Архітектура серверної частини додатка

2.4.1 Загальні поняття архітектури REST API

REST (Representational state transfer) - це стиль архітектури програмного забезпечення для розподілених систем, таких як World Wide Web, який, як правило, використовується для побудови веб-служб. Термін REST був введений у 2000 році Роєм Філдінгом, одним з авторів HTTP-протоколу. Системи, що підтримують REST, називаються RESTful-системами.

У загальному випадку REST є дуже простим інтерфейсом управління інформацією без використання якихось додаткових внутрішніх прошарків. Кожна одиниця інформації однозначно визначається глобальним ідентифікатором, таким як URL. Кожна URL в свою чергу має строго заданий формат.

Незважаючи на те, що REST не є стандартом сам по собі, більшість RESTful-реалізацій використовують такі стандарти, як HTTP, URL, JSON і XML. Дані повінні передаватіся у виде невелікої кількості стандартних форматів (например, HTML, XML, JSON). Будь-який REST протокол (HTTP в тому чіслі) повинен підтрімуваті кешування, що не винен залежаться від мережевих прошарку, що не винен зберігаті информации про стан между парами «запит-відповідь». Стверджується, что такий підхід Забезпечує масштабовність системи и дозволяє їй еволюціонуваті з новімі Вимогами. Хоча ресурс може бути яким завгодно, дії, які можна виконувати над ресурсом,

визначаються повідомленнями, які визначено стандартним протоколом. REST API широко використовується в додатках у всьому світу.

2.4.2 Використання REST для побудови Web-сервісів.

Як відомо, web-сервіс - це додаток працює в World Wide Web і доступ до якого надається по HTTP-протоколу, а обмін інформацією йде за допомогою формату JSON. Отже, формат даних переданих в тілі запиту буде завжди JSON.

Для кожної одиниці інформації визначається 5 дій. А саме:

GET / info / (Index) - отримує список всіх об'єктів. Як правило, це спрощений список, тобто містить тільки поля ідентифікатора і назви об'єкта, без інших даних.

GET / info / {id} (View) - отримує повну інформацію про об'єкті.

PUT / info / або POST / info / (Create) - створює новий об'єкт. Дані передаються в тілі запиту без застосування кодування, навіть urlencode.

POST / info / {id} або PUT / info / {id} (Edit) - змінює дані з ідентифікатором {id}, можливо замінює їх. Дані так само передаються в тілі запиту, але на відміну від PUT тут є певний нюанс. Справа в тому, що POST-запит має на увазі наявність urldecoded-post-data. Тобто якщо не застосовувати кодування - це порушення стандарту.

DELETE / info / {id} (Delete) - видаляє дані з ідентифікатором {id}.

2.4.3 Огляд основних API серверної частини

Для обміну данними між серверною частиною та користувацьким інтерфейсом використовується технологія REST API. Основні API адреси, які має серверна частина додатку для збереження роботи з рокзладом навчального закладу, у вигляді груп описані у таблиці 2.1.

Таблиця 2.1 – Опис основних API серверної частини для групи

Приклад API URI	API опис	Опис параметрів	Приклад відповіді

GET /	Отримати розклад всього навчального закладу		{
			"schedule": [
			{
			"groupId": 1,
			"groupName": "1CI-16Б",
			"schedule": [
			{
			"0": [
			{
			"subject": "Програмування",
			"type": "ПЗ",
			"teacher": "Довгалець Сергій Михайлович",
			"classroom": "5213"
			}
			]
			}
			]
			},
			{
			"groupId": 2,
			"groupName": "2CI-16Б",
			"schedule": [
			{
			"0": [
			{
			"subject": "Програмування",
			"type": "ПЗ",
			"teacher": "Довгалець Сергій Михайлович",
			"classroom": "5213"
			}
			]
			}
			]
			}
			]
			}

Продовження таблиці 2.1 – Опис основних API серверної частини для групи

GET / 25	Отримати розклад групи з id = 25	id – унікальний ідентифікатор групи	{ "groupId": 1, "groupName": "1CI-16Б", "schedule": [{ "0": [{ "subject": "Програмування", "type": "ПЗ", "teacher": "Довгалець Сергій Михайлович", "classroom": "5213" }] }] }
POST / 25	Створити або редагувати розклад групи з id = 25	{ "groupId": 25, "groupName": "1CI-16Б", "schedule": [{ "0": [{ "subject": "Програмування", "type": "ПЗ", "teacher": "Довгалець Сергій Михайлович", "classroom": "5213" }] }] }	OK (code:200)
DELETE/25	Видалити розклад групи з id = 25	id – унікальний ідентифікатор групи	OK (code:200)

Схематиче представлення відповідей серверу представлено на риснку 2.1.

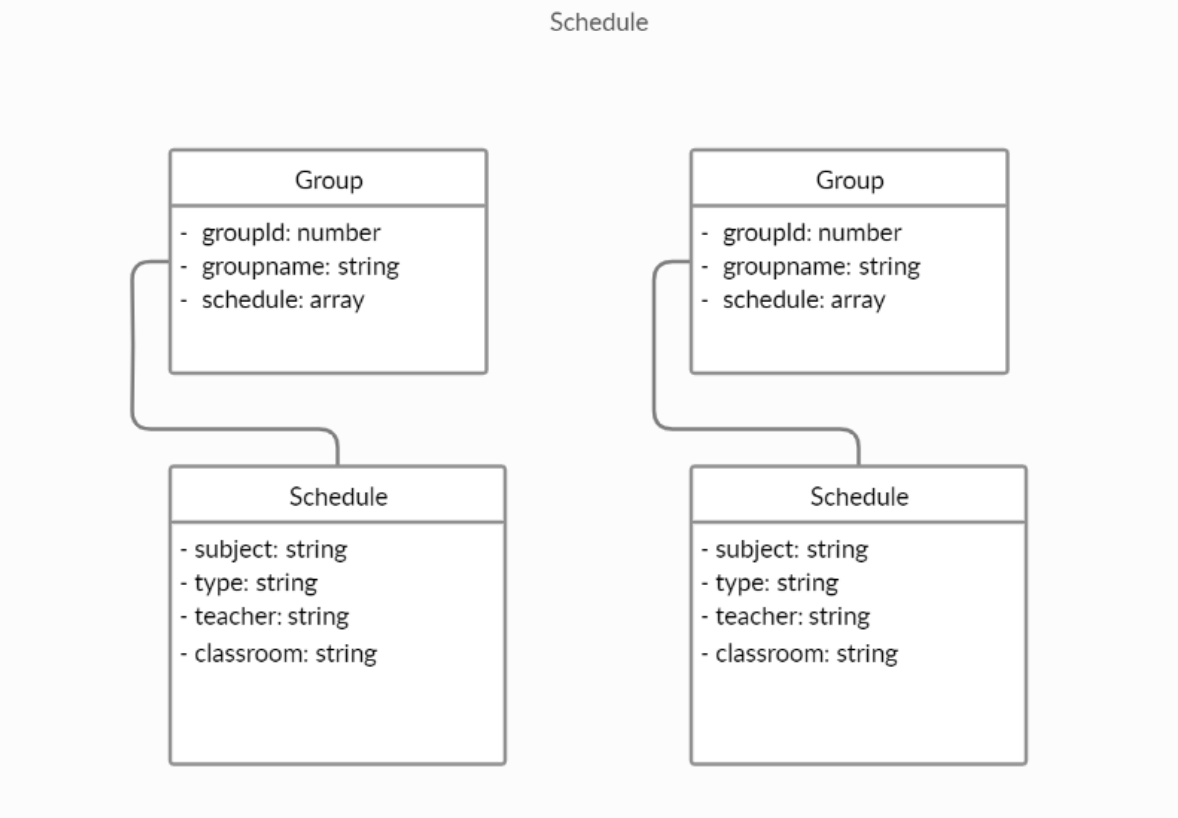

Рисунок 2.1 – Схематичне представлення відповіді сервера

Також серверна частина додатку реалізує роботу з розкладом для викладачів, тому існують API адреси, які дають можливість працювати з рокзладом навчального закладу у вигляді викладачів які описані у таблиці 2.2. Основний протокол для передачі даних викроистовується HTTP, а формат даних JSON. Існує можливість передачі параметрів через адресну стрічку. Надання можливості роботи з розкладом з точки зору викладача вимагає написання більше стрічок коду та зміни логіку контролера. Усі запити починаються з /schedule. Наприклад, /schedule/teachers/25 або /schedule/group/34. Але такі рішення створює можливість розробляти користувацький інтерфейс не тільки на основі однієї програмної платформи.

Таблиця 2.2 – Опис основних API серверної частини для викладача

Приклад API URI	API опис	Опис параметрів	Приклад відповіді

GET /	Отримати розклад всього навчального закладу		{
			"schedule": [
			{
			"groupId": 1,
			"groupName": "1CI-16Б",
			"schedule": [
			{
			"0": [
			{
			"subject": "Програмування",
			"type": "ПЗ",
			"teacher": "Довгалець Сергій Михайлович",
			"classroom": "5213"
			}
			]
			}
			]
			},
			{
			"groupId": 2,
			"groupName": "2CI-16Б",
			"schedule": [
			{
			"0": [
			{
			"subject": "Програмування",
			"type": "ПЗ",
			"teacher": "Довгалець Сергій Михайлович",
			"classroom": "5213"
			}
			]
			}
			]
			}
			]
			}

Продовження таблиці 2.2 – Опис основних API серверної частини для викладача

GET /teachers/25	Отримати розклад викладача з id = 25	id – унікальний ідентифікатор викладача	{ "groupId": 1, "groupName": "1CI-16Б", "schedule": [{ "0": [{ "subject": "Програмування", "type": "ПЗ", "teacher": "Довгалець Сергій Михайлович", "classroom": "5213" }] }] }
POST teachers/ 25	Створити або редагувати розклад викладача з id = 25	{ "groupId": 25, "groupName": "1CI-16Б", "schedule": [{ "0": [{ "subject": "Програмування", "type": "ПЗ", "teacher": "Довгалець Сергій Михайлович", "classroom": "5213" }] }] }	OK (code:200)
DELETE/teachers/25	Видалити розклад викладача з id = 25	id – унікальний ідентифікатор групи	OK (code:200)

2.5 Висновки до розділу

У даному розділі було розглянуто основні програмні платформи. Обґрунтоване іх використання. А саме ASP.NET Core та Angular. Мови програмування C# та TypeScript та їх історія. Для створення подібної програмної системи не обов'язково використовувати саме ці інструменти, завдяки цьому і існує перевага мікросервісної архітектури – всі технології додатку легко замінити на інші технології.

Також було розглянуто основні поняття REST API архітектури, адже саме ця архітектура дозволяє взаємодіяти мікросервісам.

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Для створення даної програмної системи було обрано дви програмні платформи та дві мови програмування. Кожен елемент цієї системи можна замінити завдяки ізольованості. Розроблена архітектура програмної системи зображена на Рисунку 3.1.

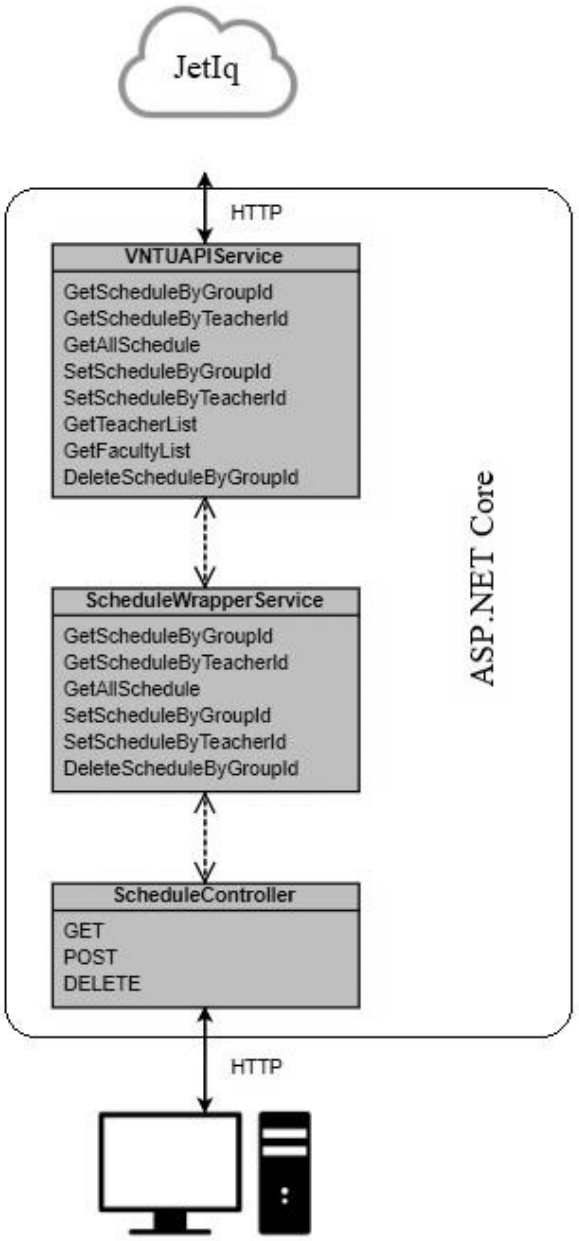

Рисунок 3.1 – Розроблена архітектура програмної системи

3.1 Розробка та структура користувацького інтерфейсу

Для створення користувацького інтерфейсу було обрано мова програмування TypeScript та програмна платформа Angular. За допомогою цих інструментів створення інтерфейсів займає меньше часу а логіку роботи додатка стає більш зрозумілим та структурованим. Архітектура додатку користувацького інтерфейсу зображено на Рисунку 3.2. Додаток на основі програмній платформі Angular складається с компонентів, модулів та сервісів.

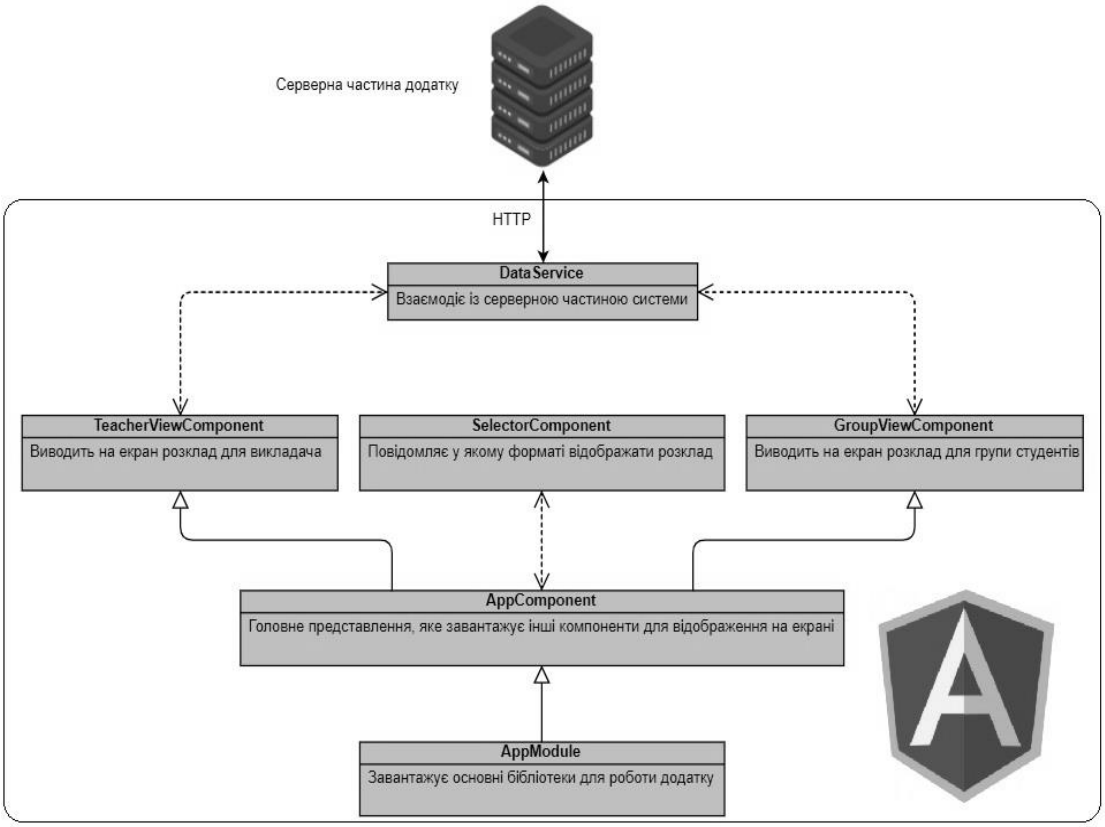

Рисунок 3.2 – Архітектура додатку користувацького інтерфейсу

3.1.1 Основні компоненти

Одним з ключових елементів додатку користувацького інтерфейсу є компоненти. Компонент управляє відображенням уявлення на екрані та показ інформації користувачу.

Основні компоненти додатку користувацького інтерфейсу є:

– app.component.ts;
– group-view.component.ts;
– selector.component.ts;
– teacher-view.component.ts;

Компонент app.component.ts є найголовнішим, адже саме він викликає в разі необхідності всі інші компоненти передаючи ті чи інши параметри для показу данних на екран.

Компонент selector.component.ts є простим, цей компонент повідомлює компоненту app.component.ts для кого і по якому формату показувати розклад: для группи студентів чи для викладача.

У разі вибору формату для студентів завантажується компонент group-view.component.ts. Цей компонент використовує сервіс DataService для отримання об'єкту розкладу та відворює цей об'єкт на екран у формі зручної таблиці для студентів.

Якщо після вибору формату студентів обрати формат у викладача, то компонент app.component.ts звільне пам'ять від компонента group-view.component.ts та завантаже компонент teacher-view.component.ts. Цей компонент в свою чергу використвує сервіс DataService для отримання об'єкту розкладу та відворює цей об'єкт на екран у формі зручної таблиці для викладача.

3.1.2 Основні сервіси

Сервіси в Angular представляють досить широкий спектр класів, які виконують деякі специфічні завдання, наприклад, логіювання, роботу з даними, роботу з запитами і так далі. На відміну від компонентів сервіси не працюють з уявленнями, тобто з розміткою html, не роблять на неї прямого

впливу. Вони виконують певне і досить вузьке завдання. Стандартні завдання сервісів:

– Надання даних з додатком. Сервіс може сам зберігати дані в пам'яті, або для отримання даних може звертатися до будь-якого джерела даних, наприклад, до сервера;

– Сервіс може представляти канал взаємодії між окремими компонентами програми;

– Сервіс може реалізовувати бізнес-логіку, різні обчислювальні завдання, завдання по логгіруванію, які краще виносити з компонентів. Тим самим код компонентів буде зосереджений безпосередньо на роботі з поданням. Крім того, тим самим ми також можемо вирішити проблему повторення коду, якщо нам буде потрібно виконати одну і ту ж задачу в різних компонентах і класах;

Основними сервісами додатку користувацького інтерфейсу це DataService та LogService.

Для роботи з відправки запитів та обробки відповідей використовується сервіс DataService. Основнми методами цього сервіса є:

– GetScheduleByGroupId;

– GetScheduleByTeacherId;

– GetAllScheudle;

– UpdateScheduleByGroupId;

– UpdateScheduleByTeacherId;

– DeleteScheduleByGroupId;

Сервіс для логування LogService має лише один метод “Log”. Він використовується для збереження інформації про роботу чи помилки компонентів, сервісів чи модулів під час виконання. Сервіс LogService використовується усіма компонентами та іншими сервісами для досліджень природи помилок.

3.1.3 Основні модулі

Модулі використовуються для поділу додатку на логічні розділи. Замість того, щоб писати код всього програми в одному місці, є можливість створювати окремі модулі для поділу функціоналу програми. Кожний додаток на програмній платформі Angular як мінімум має один кореневий модуль, який, згідно з умовностей, називається "AppModule". Для роботи модуля йому необхідні ряд бібліотек, тому на початку файлу йде їх підключення. Ім'я кожної бібліотеки Angular починається з префікса "@angular". Бібліотеки встановлюються через пакетний менеджер npm і імпортуються за допомогою директиви "import".

В додатку користувацького інтерфейсу використовується тільки один модуль, так як цей додаток не є великим і не складаєтсья із десятків компонентів чи сервісів.

3.2 Розробка та структура серверної частини

Для створення серверної частини було обрано мова програмування C# та програмна платформа ASP.NET Core. За допомогою цих інструментів можливо створити серверний додаток який буде працювати на будь-якій операційній системі та оброблювати запити максимально швидко. Програмна платфомра ASP.NET Core вважається однією із найшвидших платформ для створення роботи із запитами. Так як серверний додаток буде працювати с JetIq Api база даних для цього додатку не потрібна.

3.2.1 Основні класи серверної частини додатку

Задача серверної частини додатку це відправлення запитів, отримання відповіді та оброка відповіді та інша взаємодія з JetIq API та робота із додатком

користувацького інтерфейсу. Запити протоколу Http на основі REST API. Основними класами серверної частини дододаку є:

– VNTUAPIService;
– ScheduleWrapperService
– ScheduleController
– VNTUFormat
– Startup

Саме VNTUAPIService взаємодіє з JetIq API. Цей клас відповідає за відправлення запитів, обробку відповідей та передачу даних у вигляді об'єкту іншим класам. Завдяки VNTUAPIService існує можливість отримати данні про розклад, список груп, список викладачів та інші дані які необхідні для відображення розкладу. Також, цей клас дозволяє оновлювати дані саме на JetIq але цієї можливості поки немає, адже невідомо які вимоги є у JetIq API для оновлення даних. Існує така можливість в переспективі. JetIq API відправляє відповіді на запити своїми моделями, які не підходять для нового додатку. Виникає проблема невідповідність параметрів та змінних. Цю проблему вирішує клас ScheduleWrapperService головна задача якого трансформувати модель даних яку повертає JetIq API у нову модель, з якою і буде взаємодія інших класів та сервісів. Цей клас являє собою посередником між класом якому необхідно отримати дані та між VNTUAPIService. ScheduleWrapperService створений на основі паттерна проєктування схожий на декоратор. Відміність лише в тому, что цей посередник не додає більше функціоналу класу. Суть паттерна декоратор полягає в тому, щоб сворити певну обгортку для певного класу і надати такий ж або розширений інтерфейс. Іноді декоратор називають "розумним заступником". Тобто декоратор може прикидатися оригінальним класом і при цьому розширювати його функціональність. На приклад, у вас є заступник, який ховає виклики до стороннього сервісу. Можна створити декоратор, який буде обертати і кешувати результати викликів. Або інший приклад: потрібно розширити функціональність оригінального класу, але він закритий для наслідування.

Створюється декоратор, який розширює інтерфейс оригінального класу. моделі які повертає трансоформує об'єкт для подальшої обробки та створення відповіді на запит.

Клас ScheduleController створює інтерфейс для взаємодії з користувацьким інтерфейсом. Саме він отримує запити у вигляді моделей користувацького додатку та передає ці моделі на обробку ScheduleWrapperService. Наприклад такі як отримання розкладу від JetIq API.

Клас Startup є вхідною точкою в додаток ASP.NET Core. Цей клас створює конфігурацію додатка, налаштовує сервіси, які додаток буде використовувати, встановлює компоненти для обробки запиту.

3.2.2 Основні моделі сервеної частини

Так як база даних не використовується для цієї програмної системи, а основна робота це тісна взаємодія с JetIq API, то для роботи серверної частини слід використати паттерн Data Transfer Object. DTO - це клас, який містить дані без будь-якої логіки для роботи з ними. DTO зазвичай використовуються для передачі даних між різними додатками, або між шарами всередині однієї програми. Їх можна розглядати як сховище інформації, єдина мета якого - передати цю інформацію одержувачу.

Саме роботою з Data Transfer Object моделями і займається клас ScheduleWrapperService. Основні моделі які використовує серверна частина:

- Faculty
- Group
- Schedule
- Teacher

Ці моделі можуть змінюватись взалежності від потреб взаємодії з JetIq API. Адже, існує можливість змінити відповіді які надає віддалений сервіс і вдосконалити моделі. Основні поля зображені на Рисунку 3.3

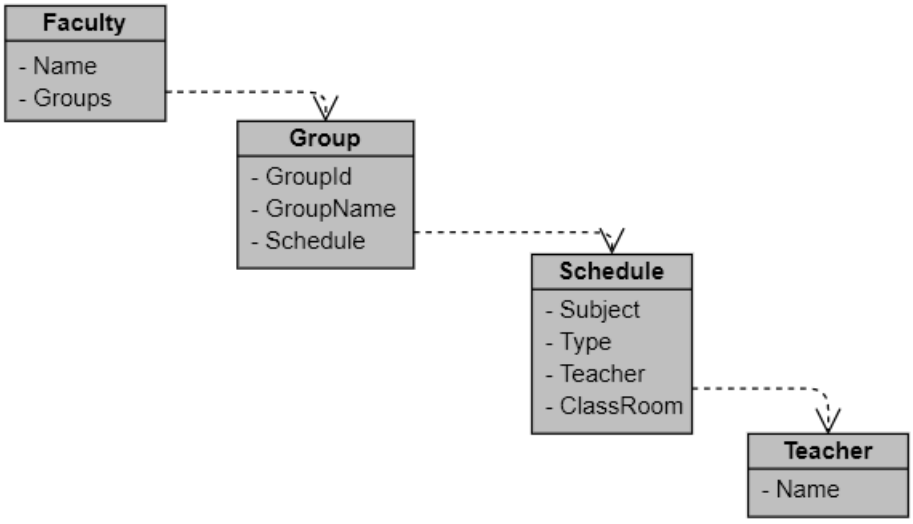

Рисунок 3.3 – Основні поля та залежності моделей

Дані моделі є проміжними та використовуються для передачі інформації до користувацького інтерфейсу. Для запитів до серверної частини слід використовувати саме такі моделі у форматі JSON.

3.3 Висновки до розділу

У даному розділу була розглянута архітектура всієї програмної системи. Представлені головні інструменти взаємодії між програмними компонентами легко заміняються та модифікуються. Серверну частину можливо легко замінити з програмної платформи ASP.NET Core на будь-яку іншу. Наприклад, на програмну платформу Node.js. Як і частину користувацького інтерфейсу можна замінити або створити ще один на основі, наприклад, React. Це демонструє високу гнучкість мікросервісної архітектури.

Також у даному розділі, розглянуто архітектуру додатку користувацького інтерфесу на основі програмної платформи Angular та

пояснено чому було обрано саме цю платформу. Представлено та описано основні класи цього додатку та спосіб взаємодії між ними.

У розділі було представлено архітектуру додатку серверної частини. Показано основні переваги програмної платформи ASP.NET Core. Наведенно поясненя використання паттернів проєктування які використовувались у цьому додатку. Також описані основні моделі для додаку серверної частини.

ВИСНОВКИ

У даній роботі розглянуто переваги та недоліки монолітної та мікросервісних архітектур. А саме переваги мікросервісної архітектури це автономність і незалежність, можливості застосування різних мов програмування, легке масшатабування, стабільність і керованість та децентралізація даних. Навпроти переваг моноліта порівняно легше почати розробку та легше розгортання. З цього випливає висновок що для великих систем, які проектуються для великого навантаження, кращим рішенням буде проектування програмної системи на основі мікросервісної архітектури. Для невеликх проектів кращим рішенням будет обрати саме монолітну архітектуру.

Також у першому розділі розглянуто методології розробки які найкраще підходить для розробки мікросервісної прогрманої системи, а саме методологія DevOps. Це набір практик для підвищення ефективності процесів розробки і експлуатації програмного забезпечення за рахунок їх безперервної інтеграції та активної взаємодії профільних фахівців за допомогою інструментів автоматизації. Девопс позиціонується як Agile-підхід для усунення організаційних і тимчасових бар'єрів між командами розробників та інших учасників життєвого циклу програмного забезпечення, а саме тестувальниками, адміністраторами, техпідтримка, щоб вони могли швидше і надійніше збирати, тестувати і випускати релізи програмних продуктів. Ця методологія найкраще підходить для мікросервісної архітектури.

У другому розділі розглянуто програмні платформи які були використані для створення прогрманої системи та їх історія. Додаток був розділений на додаток серверної частини на основі ASP.NET Core та на додаток користувацького інтерфейсу на основі Angular. Для створення цих додатків використовувались мови програмування C# та TypeScript.

Пропонується перехід від моноліта на мікросервіси на прикладі створеної програмної системи. Для цієї задачі використано інструменти

програмних платформ ASP.NET Core для серверної частини додатку та Angular для додатку користувацького інтерфейсу, на мовах програмування C# та TypeScript. Завдяки гнучкості мікросервісної архітектури для цих додатків існує можливість легко замінити програмні платформи на будь-які інші.

Визначено, що найголовнішими перевагами мікросервісної архітектури це – відмовостійкість та можливість легкої масштабованості. Наведені приклади додатків інших державних навчальних закладів для аналізу їх архітектури і визначення недоліків цієї архітектури для подальшого уникнення подібних проблем при проектування програмної системи.

Дана робота в перспективі може стати повноцінним додатком для роботи з розкладом навчальних закладів. Так як створені мікросервіси із-за своїх розмірів нескладні в модернізації. Що демонструє велику гнучкість мікросервісної архітектури.

ЛІТЕРАТУРА

1. Мамашвілі Л. О. Мікросервісна архітектура / Л. О. Мамашвілі // Матеріали доповідей XLIX науково-технічної конференції підрозділів Вінницького національного технічного університету, 18–29 травня. – Вінниця : ВНТУ, 2020.

2. Введение в TypeScript [Електронний ресурс]. Режим доступу: https://metanit.com/web/typescript/1.1.php. – Назва з екрану.

3. Просто о микросервисах [Електронний ресурс]. Режим доступу: https://habr.com/ru/company/raiffeisenbank/blog/346380/. – Назва з екрану.

4. Что такое Angular. Первый проект [Електронний ресурс]. Режим доступу https://metanit.com/web/angular2/1.1.php. – Назва з екрану.

5. Монолитная vs Микросервисная архитектура [Електронний ресурс]. https://proglib.io/p/monolitnaya-vs-mikroservisnaya-arhitektura-2019-09-16. – Назва з екрану.

6. Микросервисная архитектура: плюсы и минусы [Електронний ресурс]. https://javarush.ru/groups/posts/2015-mikroservisnaja-arhhitektura-pljusih-i-minusih. – Назва з екрану.

7. Рихтер Д. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#. 4-е изд./ Рихтер Д. – М.: Издательство — Питер, 2016, 896 с. ISBN: 978-5-4461-1102-2.

8. По стопам лучших: микросервисная архитектура в разрезе [Електронний ресурс]. https://techrocks.ru/2019/11/13/micro-service-architecture/. – Назва з екрану.

9. Архитектура REST [Електронний ресурс]. https://habr.com/ru/post/38730/. – Назва з екрану.

10. REST Representational State Transfer [Електронний ресурс]. http://dashvlas.com/blog/rest. – Назва з екрану.

11. Mark Masse. REST API Design Rulebook. – 1-ше видавництво./ Mark Masse. – Sebastopol: O'Reilly, 2011, 116 с. ISBN: 978-1449310509.

12. КАК ПРАВИЛЬНО РАБОТАТЬ С REST API [Електронний ресурс]. https://itvdn.com/ru/blog/article/rest-api-18/. – Назва з екрану.

13. ASP.NET Core - новая эпоха в развитии ASP.NET [Електронний ресурс]. https://metanit.com/sharp/aspnet5/1.1.php. – Назва з екрану.

14. Ричардсон К. Микросервисы. Паттерны разработки и рефакторинга. / Ричардсон К. – Питер, 2019, 544 с. ISBN: 978-5-4461-0996-8.

15. Что такое микросервисная архитектура и когда ее применять [Електронний ресурс]. https://proglib.io/p/microservices/. – Назва з екрану.

16. TypeScript [Електронний ресурс]. https://www.typescriptlang.org/. – Назва з екрану.

17. ASP.NET documentation [Електронний ресурс]. https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-3.1. – Назва з екрану.

18. Angular [Електронний ресурс]. https://angular.io/. – Назва з екрану.

19. Angular CLI [Електронний ресурс]. https://cli.angular.io/. – Назва з екрану.

20. C# documentation [Електронний ресурс]. https://docs.microsoft.com/en-us/dotnet/csharp/. – Назва з екрану.

21. What is DevOps? [Електронний ресурс.] https://aws.amazon.com/devops/what-is-devops/?nc1=h_ls. – Назва з екрану.

22. DevOps [Електронний ресурс.] https://www.bigdataschool.ru/wiki/devops. – Назва з екрану.

23. What is DevOps? [Електронний ресурс.] https://theagileadmin.com/what-is-devops/. – Назва з екрану.

24. Дэвис Д. Философия DevOps. Искусство управления IT. / Дэвис Д. – Питер, 2019, 416 с. ISBN: 978-5-4461-1141-1

25. Data transfer Object (объект передачи данных) [Електронний ресурс.] http://design-pattern.ru/patterns/data-transfer-object.html. – Назва з екрану.

26. Декоратор [Електронний ресурс.] https://refactoring.guru/ru/design-patterns/decorator. – Назва з екрану.

27. Паттерн Декоратор (Decorator; Wrapper, Обертка) [Електронний ресурс.] https://studref.com/329499/informatika/pattern_dekorator_decorator_wrapper_obertka. – Назва з екрану.

ДОДАТКИ

Додаток А

(обов’язковий)

Архітектура програмної системи

					08-02.БДР.005.00.000.ПД		
					Архітектура програмної системи	Літ.	Маса	Масштаб
Змн	Арк.	№ докум.	Підпис	Дата			1 : 1
Розроб.		Мамашвілі Л.О.				Арк.	Аркушів 1
Перевір.		Довгалець С. М.					
Т. Контр.		Довгалець С. М.					
Реценз.							
Н. Контр.		Довгалець С. М.				ВНТУ, 1СІ-16б	
Затверд.		Квєтний Р. Н.					

Додаток А (обов’язковий). Архітектура програмної системи.

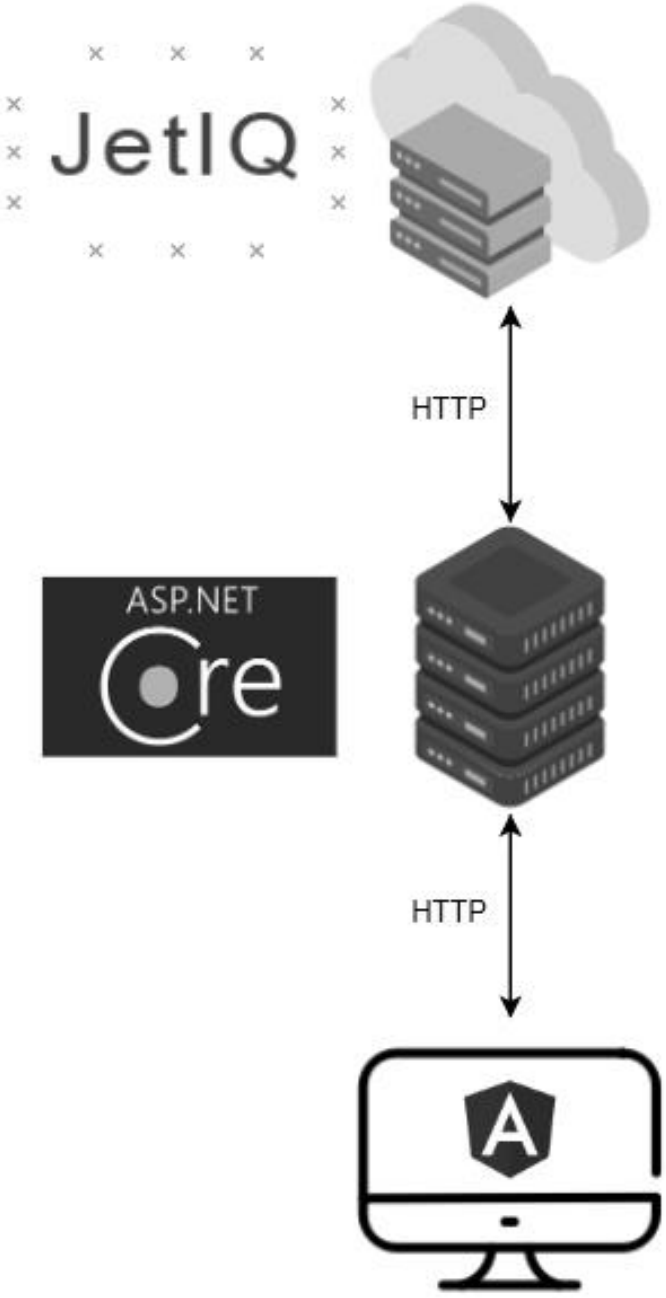

Рисунок А.1 – Архітектура програмної системи

Додаток Б

(обов’язковий)

Архітектура серверної частини

					08-02.БДР.005.00.000.ПД			
					Архітектура серверної частини	Літ.	Маса	Масштаб
Змн	Арк.	№ докум.	Підпис	Дата				1 : 1
Розроб.		Мамашвілі Л.О.				Арк.	Аркушів 1	
Перевір.		Довгалець С. М.						
Т. Контр.		Довгалець С. М.						
Реценз.						ВНТУ, 1СІ-16б		
Н. Контр.		Довгалець С. М.						
Затверд.		Квєтний Р. Н.						

Додаток Б (обов’язковий). Архітектура серверної частини.

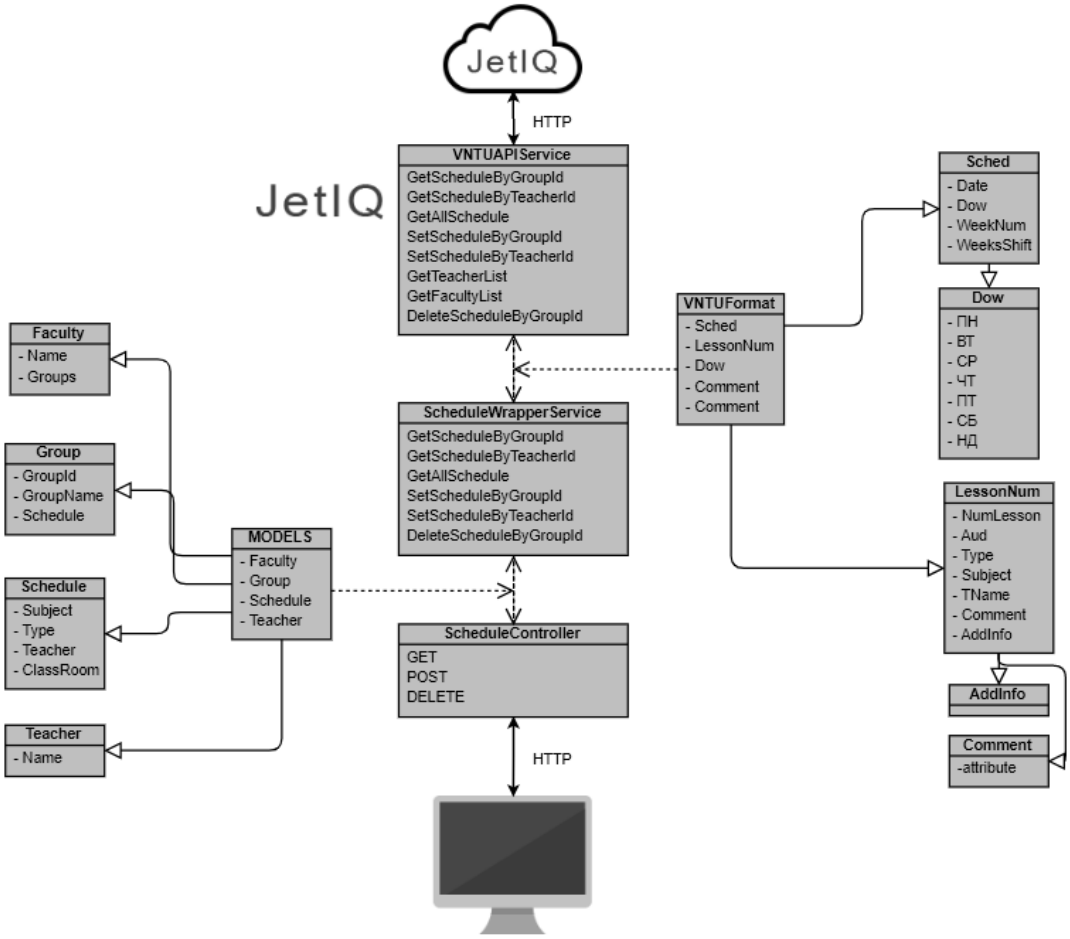

Рисунок Б.1 – Архітектура серверної частини

Додаток В

(обов’язковий)

Архітектура дотаку користувацького інтерфейсу

					08-02.БДР.005.00.000.ПД		
					Архітектура додатку користувацького інтерфейсу	Літ.	Маса	Масштаб
Змн	Арк.	№ докум.	Підпис	Дата				1 : 1
Розроб.		Мамашвілі Л.О.				Арк.	Аркушів 1	
Перевір.		Довгалець С. М.						
Т. Контр.		Довгалець С. М.						
Реценз.						ВНТУ, 1СІ-16б		
Н. Контр.		Довгалець С. М.						
Затверд.		Квєтний Р. Н.						

Додаток В (обов’язковий). Архітектура користувацького інтерфейсу

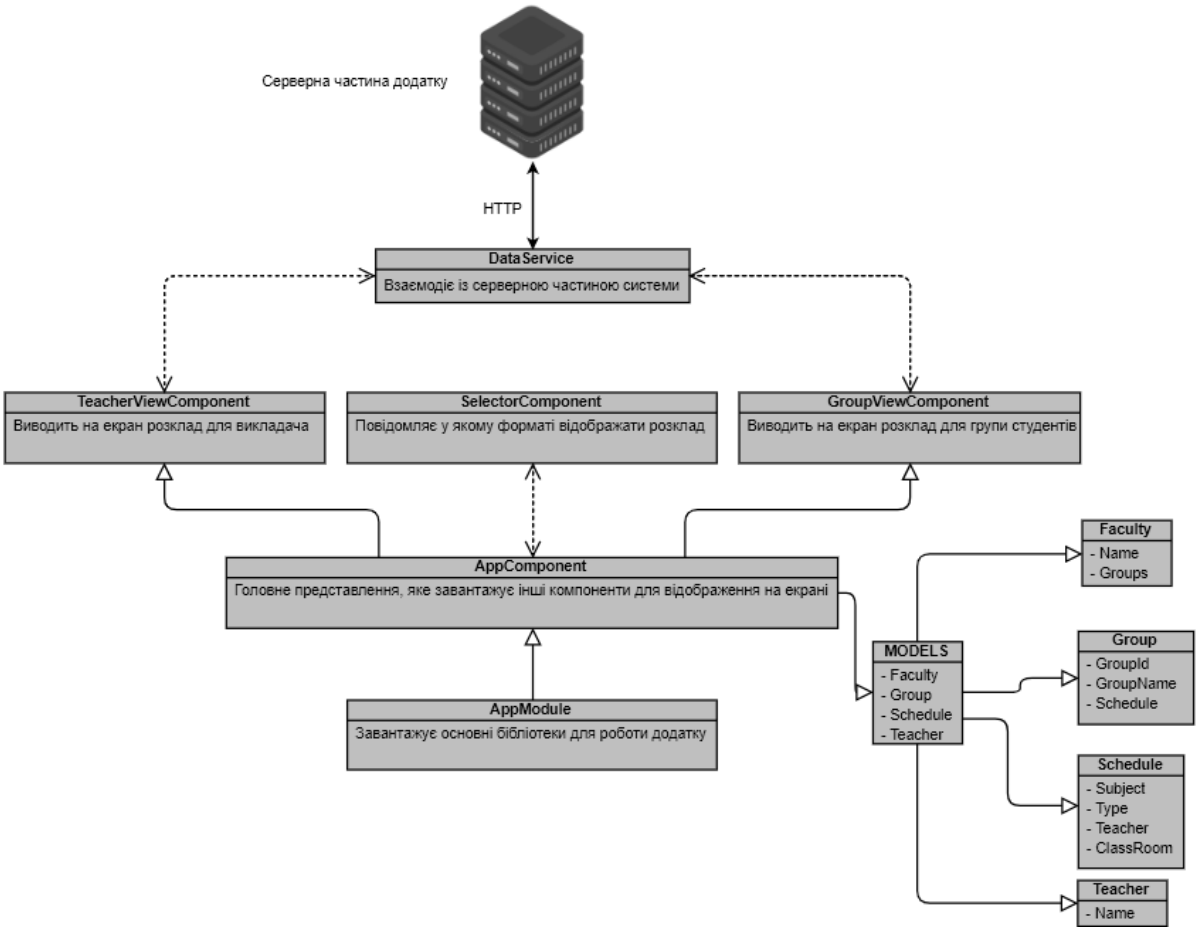

Рисунок В.1 – Архітектура додатку користувацького інтерфейсу

Додаток Г

(обов'язковий)

Основний код серверної частини

ScheduleController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Newtonsoft.Json;
using VNTU_DIPLOMA.Models;
using VNTU_DIPLOMA.Services;

namespace VNTU_DIPLOMA.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ScheduleController : ControllerBase
    {
        IScheduleWrapperService _scheduleService;

        ScheduleController(IScheduleWrapperService scheduleWrapperService)
        {
            _scheduleService = scheduleWrapperService;
        }

        [HttpGet]
        [Route("schedule")]
        public ActionResult<IEnumerable<Schedule>> Get()
        {
            return _scheduleService.GetAllSchedule();
        }

        // GET api/values/5
        [HttpGet("{id}")]
        [Route("schedule/group/{id}")]
        public ActionResult<string> GetByGroup(int id)
        {
            var result = JsonConvert.SerializeObject(_scheduleService.GetScheduleByGroupId(id));
            return result;
        }

        [HttpGet("{id}")]
        [Route("schedule/teacher/{id}")]
        public ActionResult<string> GetByTeacher(int id)
        {
            var result = JsonConvert.SerializeObject(_scheduleService.GetScheduleByTeacherId(id));
            return result;
        }


        // POST api/values
        [HttpPost]
        [Route("schedule/group/{id}")]
        public IActionResult UpdateScheduleByGroupId(int id, [FromBody] List<Schedule> value)
        {
            _scheduleService.SetScheduleForGroup(id, value);
            return Ok();
        }

        // POST api/values
        [HttpPost]
        [Route("schedule/teacher/{id}")]
        public IActionResult UpdateScheduleByTeacherId(int id, [FromBody] List<Schedule> value)
        {
            _scheduleService.SetScheduleForTeacher(id, value);
            return Ok();
        }

        [HttpDelete]
        [Route("schedule/group/{id}")]
```

```
        public IActionResult DeleteScheduleByGroupId(int id)
        {
            _scheduleService.DeleteScheduleByGroupId(id);
            return Ok();
        }
    }
}
```

ScheduleWrapperService.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using VNTU_DIPLOMA.DTO;
using VNTU_DIPLOMA.Models;

namespace VNTU_DIPLOMA.Services
{
    public class ScheduleWrapperService : IScheduleWrapperService
    {
        IVNTUAPIService _vntuApiSerivce;

        public ScheduleWrapperService(IVNTUAPIService VNTUAPIService)
        {
            _vntuApiSerivce = VNTUAPIService;
        }

        public List<Schedule> GetScheduleByGroupId(int id)
        {
            var oldScheds = _vntuApiSerivce.GetScheduleByGroupId(id);

            List<Schedule> newSchedule = new List<Schedule>();
            foreach (var oldSched in oldScheds)
            {
                Schedule newDay = new Schedule
                {
                    classroom = oldSched.Lesson.LessonNum.Aud,
                    subject = oldSched.Lesson.LessonNum.Subject,
                    teacher = $"{oldSched.Lesson.LessonNum.TName} {oldSched.Lesson.LessonNum.Comment}",
                    type = oldSched.Lesson.LessonNum.Type
                };
                newSchedule.Add(newDay);
            }

            return newSchedule;
        }

        public List<Schedule> GetAllSchedule()
        {
            var oldScheds = _vntuApiSerivce.GetAllSchedule();

            List<Schedule> newSchedule = new List<Schedule>();
            foreach (var oldSched in oldScheds)
            {
                Schedule newDay = new Schedule
                {
                    classroom = oldSched.Lesson.LessonNum.Aud,
                    subject = oldSched.Lesson.LessonNum.Subject,
                    teacher = $"{oldSched.Lesson.LessonNum.TName} {oldSched.Lesson.LessonNum.Comment}",
                    type = oldSched.Lesson.LessonNum.Type
                };
                newSchedule.Add(newDay);
            }

            return newSchedule;
        }

        public void SetScheduleForGroup(int GroupId, List<Schedule> schedule)
        {
            var transformedSchedule = new List<Sched>();

            long weekNum = 0;
            foreach (var schd in schedule)
            {
```

```
                if (weekNum == 6) weekNum = 0;
                Sched oldFormat = new Sched
                {
                    Lesson = new Lesson { LessonNum = {
                            Aud = schd.classroom,
                            Subject = schd.subject,
                            TName = schd.teacher,
                            Type = schd.type
                        } },
                    WeekNum = weekNum
                };
                transformedSchedule.Add(oldFormat);
            }

            _vntuApiSerivce.SetScheduleByGroupId(GroupId, transformedSchedule);
        }

        public List<Schedule> GetScheduleByTeacherId(int id)
        {
            var oldScheds = _vntuApiSerivce.GetScheduleByTeacherId(id);

            List<Schedule> newSchedule = new List<Schedule>();
            foreach (var oldSched in oldScheds)
            {
                Schedule newDay = new Schedule
                {
                    classroom = oldSched.Lesson.LessonNum.Aud,
                    subject = oldSched.Lesson.LessonNum.Subject,
                    teacher = $"{oldSched.Lesson.LessonNum.TName} {oldSched.Lesson.LessonNum.Comment}",
                    type = oldSched.Lesson.LessonNum.Type
                };
                newSchedule.Add(newDay);
            }

            return newSchedule;
        }

        public void SetScheduleForTeacher(int Id, List<Schedule> schedule)
        {

            var transformedSchedule = new List<Sched>();

            long weekNum = 0;
            foreach (var schd in schedule)
            {
                if (weekNum == 6) weekNum = 0;
                Sched oldFormat = new Sched
                {
                    Lesson = new Lesson
                    {
                        LessonNum = {
                            Aud = schd.classroom,
                            Subject = schd.subject,
                            TName = schd.teacher,
                            Type = schd.type
                        }
                    },
                    WeekNum = weekNum
                };
                transformedSchedule.Add(oldFormat);
            }

            _vntuApiSerivce.SetScheduleByTeacherId(Id, transformedSchedule);
        }

        public void DeleteScheduleByGroupId(int Id)
        {
            _vntuApiSerivce.DeleteScheduleByGroupId(Id);
        }
    }
}
```

VNTUAPIService.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Net.Http;
using System.Threading.Tasks;
using VNTU_DIPLOMA.DTO;
using VNTU_DIPLOMA.Models;
using Newtonsoft.Json;

namespace VNTU_DIPLOMA.Services
{
    public class VNTUAPIService : IVNTUAPIService
    {
        private string _scheduleByGroupURL= "https://iq.vntu.edu.ua/b04213/curriculum/api.php?view=g";
        private string _scheduleByTeacherURL = "https://iq.vntu.edu.ua/b04213/curriculum/api.php?view=t";
        private string _facultyListURL = "https://iq.vntu.edu.ua/b04213/curriculum/api.php?";
        private string unicJetIqURL = "https://iq.vntu.edu.ua/b04213/curriculum/api.php?";

        static readonly HttpClient _httpClient = new HttpClient();

        public List<Sched> GetScheduleByGroupId(int id)
        {
            string groupUrl = string.Concat(_scheduleByGroupURL, "&group_id=", id.ToString());
            string result = _httpClient.GetStringAsync(groupUrl).Result;
            List<Sched> sched = JsonConvert.DeserializeObject<List<Sched>>(result);

            return sched;
        }

        public List<Faculty> GetFacultyList()
        {
            string result = _httpClient.GetStringAsync(_facultyListURL).Result;
            List<Faculty> sched = JsonConvert.DeserializeObject<List<Faculty>>(result);

            return sched;
        }

        public List<Teacher> GetTeacherList()
        {
            string result = _httpClient.GetStringAsync(unicJetIqURL).Result;
            List<Teacher> teachers = JsonConvert.DeserializeObject<List<Teacher>>(result);

            return teachers;
        }

        public void SetScheduleByGroupId(int id, List<Sched> sched)
        {
            var result = _httpClient.PostAsJsonAsync(unicJetIqURL + id, value: sched).Result;
        }

        public List<Sched> GetScheduleByTeacherId(int id)
        {
            string teacherURL = string.Concat(_scheduleByTeacherURL, "&teacher_id=", id.ToString());
            string result = _httpClient.GetStringAsync(teacherURL).Result;
            List<Sched> sched = JsonConvert.DeserializeObject<List<Sched>>(result);

            return sched;
        }

        public List<Sched> GetAllSchedule()
        {
            string result = _httpClient.GetStringAsync(unicJetIqURL).Result;
            List<Sched> sched = JsonConvert.DeserializeObject<List<Sched>>(result);

            return sched;
        }

        public void SetScheduleByTeacherId(int TeacherId, List<Sched> OldFormatSched)
        {
            var result = _httpClient.PostAsJsonAsync(unicJetIqURL + TeacherId, value:
OldFormatSched).Result;
        }

        public void DeleteScheduleByGroupId(int GroupId)
        {
            _httpClient.DeleteAsync(unicJetIqURL + GroupId);
        }
    }
}
```

Додаток Ґ

(обов’язковий)

Основний код користувацького інтерфейсу

Angular.json

```
{
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
  "version": 1,
  "newProjectRoot": "projects",
  "projects": {
    "diplomaFE": {
      "projectType": "application",
      "schematics": {},
      "root": "",
      "sourceRoot": "src",
      "prefix": "app",
      "architect": {
        "build": {
          "builder": "@angular-devkit/build-angular:browser",
          "options": {
            "outputPath": "dist/diplomaFE",
            "index": "src/index.html",
            "main": "src/main.ts",
            "polyfills": "src/polyfills.ts",
            "tsConfig": "tsconfig.app.json",
            "aot": true,
            "assets": [
              "src/favicon.ico",
              "src/assets"
            ],
            "styles": [
              "./node_modules/@angular/material/prebuilt-themes/indigo-pink.css",
              "src/styles.css"
            ],
            "scripts": []
          },
          "configurations": {
            "production": {
              "fileReplacements": [
                {
                  "replace": "src/environments/environment.ts",
                  "with": "src/environments/environment.prod.ts"
                }
              ],
              "optimization": true,
              "outputHashing": "all",
              "sourceMap": false,
              "extractCss": true,
              "namedChunks": false,
              "extractLicenses": true,
```

```
      "vendorChunk": false,
      "buildOptimizer": true,
      "budgets": [
       {
        "type": "initial",
        "maximumWarning": "2mb",
        "maximumError": "5mb"
       },
       {
        "type": "anyComponentStyle",
        "maximumWarning": "6kb",
        "maximumError": "10kb"
       }
      ]
     }
    }
   },
   "serve": {
    "builder": "@angular-devkit/build-angular:dev-server",
    "options": {
     "browserTarget": "diplomaFE:build"
    },
    "configurations": {
     "production": {
      "browserTarget": "diplomaFE:build:production"
     }
    }
   },
   "extract-i18n": {
    "builder": "@angular-devkit/build-angular:extract-i18n",
    "options": {
     "browserTarget": "diplomaFE:build"
    }
   },
   "test": {
    "builder": "@angular-devkit/build-angular:karma",
    "options": {
     "main": "src/test.ts",
     "polyfills": "src/polyfills.ts",
     "tsConfig": "tsconfig.spec.json",
     "karmaConfig": "karma.conf.js",
     "assets": [
      "src/favicon.ico",
      "src/assets"
     ],
     "styles": [
      "./node_modules/@angular/material/prebuilt-themes/indigo-pink.css",
      "src/styles.css"
     ],
     "scripts": []
    }
   },
```

```
      "lint": {
        "builder": "@angular-devkit/build-angular:tslint",
        "options": {
          "tsConfig": [
            "tsconfig.app.json",
            "tsconfig.spec.json",
            "e2e/tsconfig.json"
          ],
          "exclude": [
            "**/node_modules/**"
          ]
        }
      },
      "e2e": {
        "builder": "@angular-devkit/build-angular:protractor",
        "options": {
          "protractorConfig": "e2e/protractor.conf.js",
          "devServerTarget": "diplomaFE:serve"
        },
        "configurations": {
          "production": {
            "devServerTarget": "diplomaFE:serve:production"
          }
        }
      }
    }
  },
  "defaultProject": "diplomaFE",
  "cli": {
    "analytics": "51a6cf1e-d75a-4ec0-9431-87ceff6857ca"
  }
}
```

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {A11yModule} from '@angular/cdk/a11y';
import {ClipboardModule} from '@angular/cdk/clipboard';
import {DragDropModule} from '@angular/cdk/drag-drop';
import {PortalModule} from '@angular/cdk/portal';
import {ScrollingModule} from '@angular/cdk/scrolling';
import {CdkStepperModule} from '@angular/cdk/stepper';
import {CdkTableModule} from '@angular/cdk/table';
import {CdkTreeModule} from '@angular/cdk/tree';
import {MatAutocompleteModule} from '@angular/material/autocomplete';
import {MatBadgeModule} from '@angular/material/badge';
import {MatBottomSheetModule} from '@angular/material/bottom-sheet';
import {MatButtonModule} from '@angular/material/button';
import {MatButtonToggleModule} from '@angular/material/button-toggle';
import {MatCardModule} from '@angular/material/card';
import {MatCheckboxModule} from '@angular/material/checkbox';
import {MatChipsModule} from '@angular/material/chips';
```

```
import {MatStepperModule} from '@angular/material/stepper';
import {MatDatepickerModule} from '@angular/material/datepicker';
import {MatDialogModule} from '@angular/material/dialog';
import {MatDividerModule} from '@angular/material/divider';
import {MatExpansionModule} from '@angular/material/expansion';
import {MatGridListModule} from '@angular/material/grid-list';
import {MatIconModule} from '@angular/material/icon';
import {MatInputModule} from '@angular/material/input';
import {MatListModule} from '@angular/material/list';
import {MatMenuModule} from '@angular/material/menu';
import {MatNativeDateModule, MatRippleModule} from '@angular/material/core';
import {MatPaginatorModule} from '@angular/material/paginator';
import {MatProgressBarModule} from '@angular/material/progress-bar';
import {MatProgressSpinnerModule} from '@angular/material/progress-spinner';
import {MatRadioModule} from '@angular/material/radio';
import {MatSelectModule} from '@angular/material/select';
import {MatSidenavModule} from '@angular/material/sidenav';
import {MatSliderModule} from '@angular/material/slider';
import {MatSlideToggleModule} from '@angular/material/slide-toggle';
import {MatSnackBarModule} from '@angular/material/snack-bar';
import {MatSortModule} from '@angular/material/sort';
import {MatTableModule} from '@angular/material/table';
import {MatTabsModule} from '@angular/material/tabs';
import {MatToolbarModule} from '@angular/material/toolbar';
import {MatTooltipModule} from '@angular/material/tooltip';
import {MatTreeModule} from '@angular/material/tree';
import {HttpClientModule} from '@angular/common/http';
import {FormsModule, ReactiveFormsModule} from '@angular/forms';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { SelectorComponent } from './selector/selector.component';
import { TeacherViewComponent } from './teacher-view/teacher-view.component';
import { GroupViewComponent } from './group-view/group-view.component';

@NgModule({
  declarations: [
    AppComponent,
    SelectorComponent,
    TeacherViewComponent,
    GroupViewComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserAnimationsModule,
    MatNativeDateModule,
    FormsModule,
    HttpClientModule,
    ReactiveFormsModule,
```

```
    MatButtonModule,
    MatInputModule,
    MatRippleModule
  ],
  exports: [
    A11yModule,
    ClipboardModule,
    CdkStepperModule,
    CdkTableModule,
    CdkTreeModule,
    DragDropModule,
    MatAutocompleteModule,
    MatBadgeModule,
    MatBottomSheetModule,
    MatButtonModule,
    MatButtonToggleModule,
    MatCardModule,
    MatCheckboxModule,
    MatChipsModule,
    MatStepperModule,
    MatDatepickerModule,
    MatDialogModule,
    MatDividerModule,
    MatExpansionModule,
    MatGridListModule,
    MatIconModule,
    MatInputModule,
    MatListModule,
    MatMenuModule,
    MatNativeDateModule,
    MatPaginatorModule,
    MatProgressBarModule,
    MatProgressSpinnerModule,
    MatRadioModule,
    MatRippleModule,
    MatSelectModule,
    MatSidenavModule,
    MatSliderModule,
    MatSlideToggleModule,
    MatSnackBarModule,
    MatSortModule,
    MatTableModule,
    MatTabsModule,
    MatToolbarModule,
    MatTooltipModule,
    MatTreeModule,
    PortalModule,
    ScrollingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

```
export class AppModule { }
```

App.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'diplomaFE';
  typeForView = "group"

  typeForViewWather() : void {

    switch (this.typeForView) {
      case "group":
         this.loadGroupView();
        break;
      case "teacher":
         this.loadTeacherView();
        break;
    }
  }
}
```

App.Component.html

```
<app-selector [typeForView]="typeForView"></app-selector>
`<div [ngSwitch]="typeForView">
    <napp-group-view ngSwitchCase="group">{{count * 10}}</app-group-view>
    <app-teacher-view ngSwitchCase="teacher"></app-teacher-view>
</div>`
```

Data-service.serivce.ts

```
import { Injectable } from '@angular/core';
import { HttpClientModule, HttpClient }    from '@angular/common/http';
import { schedule } from './Models/schedule';

@Injectable({
  providedIn: 'root'
})
export class DataService{

    constructor(private http: HttpClient){}

    getAllSchedule(): string[] {
      let result = null;
      this.http.get('schedule/').subscribe((data:any) => result=data);
      return result;
    }

    getScheduleByGroupId(id : number): string[] {
```

```
    let result = null;
    this.http.get('schedule/groups/' + id).subscribe((data:any) => result=data);
    return result;
  }

  getScheduleByTeacherId(id : number): string[] {
    let result = null;
    this.http.get('schedule/teacher/' + id).subscribe((data:any) => result=data);
    return result;
  }

  setScheduleByGroupId(id: number, sched: schedule) : void {
    this.http.post('schedule/groups/' + id, sched);
  }

  setScheduleByTeacherId(id: number, sched: schedule) : void {
    this.http.post('schedule/teachers/' + id, sched);
  }

  deleteScheduleByGroupId(id: number) : void {
    this.http.delete('schedule/groups/' + id);
  }
}
```