Державний вищий навчальний заклад
“Прикарпатський національний університет імені Василя Стефаника”
Кафедра інформаційних технологій

УДК 004

ДИПЛОМНИЙ ПРОЕКТ

Тема: Розроблення програмного забезпечення визначення ефективності енергоносіїв

Спеціальність: 121 Інженерія програмного забезпечення

ПОЯСНЮВАЛЬНА ЗАПИСКА

ДП.ІПЗ-04.ПЗ
(позначення)

Рецензент
доц. Лазарович І.М.
(посада) (підпис) (дата) (розшифровка підпису)

Студент
ІПЗ-41 Грицків Б.Л.
(шифр групи) (підпис) (дата) (розшифровка підпису)

Нормоконтролер
доц. Лазарович І.М.
(посада) (підпис) (дата) (розшифровка підпису)

Керівник дипломного проекту
проф. Кузь М.В.
(посада) (підпис) (дата) (розшифровка підпису)

Допускається до захисту

Завідувач кафедри

доц. Козленко М.І.
(посада) (підпис) (дата) (розшифровка підпису)

2020
(рік)

Державний вищий навчальний заклад
«Прикарпатський національний університет імені Василя Стефаника»
Факультет математики та інформатики Кафедра інформаційних технологій
Спеціальність Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Завідувач кафедри Козленко М.І.
_____
„_____”_____ 20___р.

ЗАВДАННЯ
НА ВИКОНАННЯ ДИПЛОМНОГО ПРОЕКТУ

Студенту Грицківу Богдану Любомировичу
(прізвище, ім’я, по батькові студента)

1. Тема проекту Розроблення програмного забезпечення визначення ефективності енергоносіїв
затверджена розпорядженням по факультету математики та інформатики від „11” вересня 2019 р.№7
2. Термін здачі студентом закінченого проекту 22 травня 2020 р.
3. Вихідні дані до дипломного проекту вимоги до обчислення енергії енергоносія, технологія програмування серверної частини – .NET Core, технології програмування клієнтської частини – HTML, TypeScript, Angular, Технології розгортання програмного забезпечення – Docker, Kubernetes
4. Зміст пояснювальної записки (перелік питань, що їх належить опрацювати)
   1. Огляд сфери енергоефективності та постановка задачі до дипломного проекту
   2. Опис вимог програмного забезпечення визначення ефективності енергоносіїв та моделювання
   3. Практична реалізація програмного забезпечення визначення ефективності енергоносіїв
   4. Бізнес-план для програмного забезпечення визначення ефективності енергоносіїв
5. Перелік графічного матеріалу (з точним забезпеченням обов’язкових креслень)_____
_____
_____
_____
_____
_____
_____
6. Дата видачі завдання 11.09.2020

Керівник _____ Кузь М.В
(підпис) (розшифровка підпису)

Завдання прийняв до виконання _____ Грицків Б.Л.
(підпис) (розшифровка підпису)

КАЛЕНДАРНИЙ ПЛАН

Номер і назва етапів дипломного проекту	Термін виконання етапів проекту	Примітка
1. Огляд сфери енергоефективності та постановка задачі до дипломного проекту	02.12.2019	Виконав
2. Опис вимог програмного забезпечення визначення ефективності енергоносіїв та їх моделювання	15.02.2020	Виконав
3. Практична реалізація програмного забезпечення визначення ефективності енергоносіїв	17.04.2020	Виконав
4. Бізнес-план для програмного забезпечення визначення ефективності енергоносіїв	11.05.2020	Виконав
5. Оформлення пояснювальної записки	18.05.2020	Виконав

Студент _____ Грицків Б.Л.
(підпис) (розшифровка підпису)

Керівник проекту _____ Кузь М.В.
(підпис) (розшифровка підпису)

РЕФЕРАТ

Пояснювальна записка: 73 сторінки (без додатків), 24 рисунки, 1 таблиця, 42 джерела, 5 додатків на 46 сторінках.

Ключові слова: ЕНЕРГОЕФЕКТИВНІСТЬ, ЕНЕРГОНОСІЙ, МІКРОСЕРВІСИ, .NET CORE, ANGULAR, PWA, KUBERNETES.

Об’єктом дослідження є: програмне забезпечення по визначенні ефективності енергоносіїв

Мета роботи: спроектувати та розробити програмне забезпечення, яке б могло визначати ефективність усіх доступних енергоносіїв.

Стислий опис тексту пояснювальної записки:

В даному дипломному проекті розглядається поняття ефективності енергоносіїв, можливості її визначення і вже існуючі аналоги, які б можна було для цього застосувати. Також було описано проектування і реалізацію такого програмного забезпечення, його цінність і економічну обгрунтованість.

ABSTRACT

Explanatory note: 73 pages (without appendix), 24 figures, 1 table, 42 references, 5 appendices on 46 pages.

Key words: ENERGY EFFICIENCY, ENERGY RESOURCE, MICROSERVICES, .NET CORE, ANGULAR, PWA, KUBERNETES.

Object of study: energy efficiency determination software

Designing and implementing of software, which can determine efficiency of all available energy resources is the main purpose of the work.

Brief description of the explanatory note:

In this diploma project there are considered term of efficiency of energy resources, ability to determine it and available analogues, which can be used for determination purposes. Also there are described designing and implementing of such software, its value and economic feasibility.

ЗМІСТ

ВСТУП

Ні для кого не секрет, що останнім трендом як для України так і для цілого світу питання енергоефективності та екології стали досить гострообговорюваними.

В ЄС все більше грантів і коштів спрямовані на підтримку енергоефективності і її розвитку як пріоритетного напрямку політики. Так, Польща 20 років підтримувала оновлення усіх хрущівок та панельних будинків, створивши для цього фонд з термомодернізації та впровадивши різноманітні стимули та програми підтримки населенню [1].

В Україні на розвиток енергоефективності виділяється невелика сума й задача утеплення і економії коштів на витрати на енергоносії є не пріоритетною. Зазначається, що 1,6 мільярда гривень уряд планує витратити на діяльність Фонду енергоефективності на 2020 рік. Ще 400 мільйонів гривень передбачено на фінансування «теплих кредитів» [2]. Разом виходить близько 2 мільярдів гривень, що складає 0,17% з усього бюджету.

Не зважаючи на це попит на енергоефективні заходи в країні зростає, переважно через бажання мешканців будинків зекономити на платіжках та жити у теплі і комфорті [1]. Також з’являються багато грантів, які підтримують розвиток екології і доцільного використання енергоресурсів.

Але є ряд трудностей, які потрібно вирішити перш, ніж обирати ту чи іншу стратегію по утепленні, налагодженні деяких процесів та інше. Першою проблемою є визначення найвигіднішого енергоносія для обраної задачі. Так як їх може бути велика кількість для обраної області, то часто обирається не найбільш економний або ефективний серед наявних. Окрім цього ситуація з енергоносіями може кардинально змінюватись з часом. Енергоефективність одних зростає за рахунок знаходження кращих технологій видобуття з них енергії, а інші стають дешевшими через зменшення витрат на їх видобування, або навпаки.

Тому метою дипломної роботи є розробка програмного забезпечення для визначення ефективності представлених на ринку енергоносіїв.

Важливість вирішення даної проблеми, через великий попит на рішення у сфері енергоефективності, робить дану тему актуальною для розгляду і розробки відповідного рішення, яке б могло задовольнити усі поставлені перед нею вимоги.

Так як зараз існує багато людей, які б хотіли заощадити на енергоресурсах, які використовують у побуті, господарстві, інших цілях, то попит на таку продукцію все ж таки існує. Тому знайдене рішення може бути досить популярним.

1 ОГЛЯД СФЕРИ ЕНЕРГОЕФЕКТИВНОСТІ ТА ПОСТАНОВКА ЗАДАЧІ ДО ДИПЛОМНОГО ПРОЕКТУ

1.1 Поняття і стан енергоефективності

Енергоефективність – ефективне (розсудливе, доцільне) використання енергетичних запасів. Це застосування меншої кількості енергії для підтримання того ж рівня енергетичного забезпечення будівель або технологічних процесів на виробництві [3].

На відміну від енергозбереження (заощадження, збереження енергії), головним чином спрямованого на зменшення енергоспоживання, енергоефективність (корисність енергоспоживання) – доцільне (ефективне) витрачання енергії. Наприклад, менше користуватись авто – заощадження енергії, а пересісти на авто з меншою витратою палива, або на електромобіль – енергоефективність. Але і енергозбереження, і енергоефективність є техніками зменшення використання енергії [4].

Для оцінки енергоефективності: випуску продукції або технологічного процесу, використовується «показник енергетичної ефективності», який оцінює споживання або втрати енергетичних запасів.

З початком 1970-х років, багато країн впроваджували політику і програми з підвищення енергоефективності. Сьогодні на промисловий сектор припадає майже 40% річного світового споживання первинних енергоресурсів і приблизно така ж частка світових викидів вуглекислого газу. Прийнято міжнародний стандарт ISO 50001, який регулює в тому числі енергоефективність.

Енергоефективність та поновлювані джерела енергії, як стверджується, є двома боками стійкої енергетичної політики і є високими пріоритетами в розподілі сталого енергетичного сектору. У багатьох країнах енергоефективність також, має вигоду для національної безпеки, оскільки її може бути використано для зниження рівня імпорту енергії з іноземних країн, а ще – уповільнити темпи споживання енергії, за яких внутрішні енергетичні ресурси виснажуються [3].

Перелік міжнародних програм по енергоефективності:

- 80 Plus;
- Міжнародне партнерство зі співробітництва в галузі енергоефективності (IPEEC) – http://www.ipeec.org;
- Міжнародне енергетичне агентство (IEA) – Energy Efficiency: http://www.iea.org/efficiency/;
- Україна – Державне агентство енергоефективності та енергозбереження: http://saee.gov.ua/;
- Європейський Союз – European Commission – Energy efficiency: http://ec.europa.eu/energy/efficiency/index_en.htm, European Council for an Energy Efficient Economy: http://www.eceee.org/.

На даний момент на українському ринку діє більше 30 компаній, які постачають послуги з енергоефективності (ППЕ). Більшість цих компаній є зареєстрованими протягом 5 років, що веде до того, що їхні послуги досить популярні і користуються попитом. В основному, ці компанії займаються консалтингом в різних напрямах.

Згідно дослідження про «Оцінку ринку постачальників послуг з енергоефективності», опублікованого на сайті Держенергоефективності, можна зробити висновок, що найменше компаній ППЕ, які приймали участь у дослідженні, надають послуги з розробки програмних продуктів або це для них не є пріоритетною задачею (рис. 1.1). Хоча потенціал розвитку даної послуги є високим, оскільки попит буде збільшуватись разом із розвитком ринку ЕЕ послуг [5, 6].

Рисунок 1.1 – Загальний відсоток послуг, що надають ППЕ, з розробки програмних продуктів та інше

1.2 Існуючі аналоги по визначенні енергоефективності

1.2.1 Check24

Першим серед розглянутих аналогів по визначенні ефективності енергоносіїв є Check24. Check24 – це німецький сайт, який має можливості до знаходженні більш вигідних і ефективних сервісів або провайдерів, які надають послуги не тільки по купівлі і використанні енергоносіїв, а ще й для страхування автомобіля, кредитування, туристичних агенств та інших [7]. Серед представлених енергоносіїв є електрична енергія і природний газ. Для того, щоб визначити більш вигідного постачальника енергоресурсів, необхідно вибрати регіон, в якому ви проживаєте (ввести поштовий індекс), обрати ваше місто, розмір домогосподарства або ввести розмір споживання природного газу чи електроенергії в кВт * год. Після чого буде сформовано список з найдоступнішими постачальниками енергоресурсу. Також є можливість

відфільтрувати список, вказавши більш специфічні характеристики певного провайдера або тариф, за яким він надає послугу. Сам сайт має підтримку тільки німецької мови і є загалом орієнтований на німецький ринок. Також слід врахувати, що так як сайт приймає тільки покази у кВт * год, то при визначенні більш вигідного провайдера природного газу необхідно наперед використовувати такі покази, а не метри кубічні, що не є сумісним з українськими реаліями.

Плюси:

- відносно простий у користуванні з можливістю конфігурування під потреби користувача;
- швидкий і інтуїтивно зрозумілий;
- прив’язання до конкретного регіону.

Мінуси:

- доступний тільки німецькою мовою;
- серед доступних енергоносіїв є тільки природний газ і електрика;
- не відповідає українським реаліям.

1.2.2 Verivox

Наступним серед аналогів є інший німецький сайт Verivox. Verivox спеціалізується на зміні постачальника природного газ або електроенергії по більш вигідному тарифі [8]. Також наявні послуги по пошуку більш вигідніших мобільних операторів і страхуванні авто та інших. Орієнтований на пересічного користувача, хоча також містить пропозиції і для бізнесу. Для того, щоб визначити постачальника з більш вигіднішими тарифами і послугами, потрібно також ввести поштовий індекс, обрати місто або регіон, розмір домогосподарства, у випадку природного газу, кількості людей, у випадку електроенергії, на яких розраховане споживання або кількість кВт * год. Після цього буде відображений список з найвигіднішими тарифами за обраними потребами. Має менше можливостей по фільтруванні і пошуку більш

спеціалізованішого постачальника, але надає більше інформації про нього. Сайт також орієнтований на німецький ринок і німецькі реалії, які не відповідають українським.

Плюси:

- відносно простий у користуванні;
- прив’язання до конкретного регіону;
- надає більше статистичної і загальної інформації про певно постачальника;
- має більше додаткових можливостей, окрім визначення провайдера енергоносіїв.

Мінуси:

- менше можливостей по фільтрації;
- доступний тільки німецькою мовою;
- серед доступних енергоносіїв є тільки природний газ і електрика;
- не відповідає українським реаліям.

1.2.3 Wechselpilot

Ще один із аналогів Wechselpilot – це сайт для підбору більш економічного постачальника електроенергії і природного газу [9]. Він є зосередженим тільки на послуги по енергоносіям. Допомагає вести розрахунковий рахунок і рахунок по показникам. Також має можливості по визначенні більш економнішого постачальника енергоресурсів не тільки для будинків, але й, наприклад, визначенні вигоди електромобілів, теплових насосів або торгівлі енергією. Також має можливість перетворення метрів кубічних у кВт * год, що є більш приближеним до даних, якими володіють пересічні користувачі. Сайт відображає не список усіх доступних постачальників, а суму грошей у євро, яку можна зекономити і сплатити, при зміні постачальника енергоресурсу. Для того, щоб переглянути усіх постачальників, потрібно попередньо зареєструватись. Щоб визначити ці дані, необхідно також вказати поштовий індекс і регіон,

кількість людей, розмір домогосподарства або кількість енергії природного газу або електрики у кВт * год. Орієнтований на німецький ринок і німецькі умови.

Плюси:

- зосереджується тільки на постачальниках енергоресурсів;
- дозволяє слідкувати і швидко оновлювати провайдера на більш економного;
- простота користування;
- швидкий і інтуїтивно зрозумілий;
- прив’язання до конкретного регіону.

Мінуси:

- мало можливостей по фільтрації постачальників;
- вимагає попередньої реєстрації;
- доступний тільки німецькою мовою;
- серед доступних енергоносіїв є тільки природний газ і електрика;
- не відповідає українським реаліям;

1.2.4 Minfin

Minfin – це сайт Міністерства фінансів України, у якому знаходиться вся інформація по комунальним тарифам загалом по Україні і Києві [10]. Є дані по електроенергії, природному газу, водопостачанні і, для Києва, гарячій воді та опаленні. Але, варто зауважити, що всі ці дані, окрім електроенергії, представлені у вигляді табличок з різними показниками ресурсу і його ціни. Ціна електроенергії визначається за допомогою внесення відповідних показників кВт * год за попередній і поточний період. Також, за умови наявності відповідного лічильника, є можливість обчислення ціни за денний, нічний або піковий період, що у деяких випадках зменшує вартість спожитої електроенергії. Так як на сайті немає даних про інші типи енергоносіїв у кВт * год, як це зроблено у калькуляторі споживання електроенергії, то вони не можуть порівнюватись між собою, за рахунок чого сайт не несе великої інформативності по ситуації ефективності наданих енергоносіїв.

Плюси:

- уся інформація подана згідно українського ринку і українських реалій;
- доступні дані по багатьом енергоносіям;
- детальний калькулятор по визначенні вартості електроенергії;
- прив’язання даних до конкретного регіону.

Мінуси:

- наявність даних по деяким енергоносіям тільки для Києва;
- необхідність проведення обчислень по наданим даним енергоносіїв вручну;
- низька інформативність по порівнянні різних енергоносіїв по ефективності.

1.2.5 Висновок по аналогам

Як можна помітити, жоден із представлених сайтів по визначенні ефективності енергоносіїв не до кінця підходить під ці цілі, так як не підпадає під ряд критеріїв. Більшість сайтів по визначенні енергоефективності є закордонного походження, що не дозволить їх використовувати в межах України. Доступний же для України сайт не має достатньої інформативності для того, щоб обрати найвигідніший варіант енергоносія, і автоматизованого механізму підбору даних і обчислення. Крім того, жоден сайт не може порівнювати енергоносії між собою, що не дає повної картини по енергоефективності кожного ресурсу.

1.3 Постановка задачі до дипломного проекту

Основною задачею дипломної роботи є створення програмного забезпечення для визначення ефективності представлених на ринку енергоносіїв.

Вхідними даними мають бути регіон, для енергоносіїв якого здійснюються обчислення, та окремі дані ресурсу, які б використовували значення по замовчуванню з можливістю їхнього коректування.

Застосувавши різні способи визначення енергії, яку виділяє енергоносій, та математичні обчислення для визначення енергоефективності певного ресурсу, усі доступні дані про енергоносій будуть перетворенні в числові значення – коефіцієнти «вартості енергї».

Вихідними даними будуть ціна енергоносія, енергія, яка виділяється з певної його одиниці та коефіцієнти «вартості енергії», які будуть відображатися в таблиці на сторінці у веб-браузері. Також на окремій сторінці можна буде дізнатись усі дані конкретного ресурсу.

Для створення програмного забезпечення для визначення ефективності енергоносіїв необхідно:

- розробити дизайн сторінок на яких буде відображатись уся описана інформація;
- реалізувати обчислення вартості енергії основі даних енергоносія;
- здійснити автоматичну обробку і отримання даних, які будуть застосовуватись для обрахунків;
- налаштувати середовище для належної роботи і функціонування програмного забезпечення;
- розробити усі веб-сторінки, які будуть задіяні у роботі веб-сайту.

Плюсами даного програмного забезпечення у порівнянні з аналогами буде:

- простота використання;
- порівняння енергоефективності різних ресурсів;
- прив’язання до конкретного регіону;
- автоматизація усіх процесів по визначенні енергії ресурсу згідно його загальних даних;
- підтримка більшості енергоносіїв представлених на ринку;
- постійний моніторинг і оновлення даних.

Мінусами даного програмного забезпечення буде:

- прив’язаність до конкретних джерел інформації;
- орієнтованість тільки на український ринок і його регіони;
- значно менша додаткова функціональність.

2 ОПИС ВИМОГ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ВИЗНАЧЕННЯ ЕФЕКТИВНОСТІ ЕНЕРГОНОСІЇВ ТА ЇХ МОДЕЛЮВАННЯ

2.1 Функціональні вимоги до програмного забезпечення визначення ефективності енергоносіїв

Функціональні вимоги – це один із видів вимог за функціональним характером, які застосовуються до проектування і розробки програмного забезпечення, які передбачають в собі опис того, що повинен робити програмний продукт, поведінку програми. Загалом цей вид вимог передбачає якусь завершену функціональність, яку може використовувати кінцевий користувач, бізнес і яку можна протестувати [11].

Для задання і опису функціональних вимог зручно використовувати різні діаграми і графічні представлення, так як вони несуть у собі набагато більше інформації про роботу системи і мають більшу цінність з точки зору розробників, власників, замовників і кінцевих користувачів, так як є узагальненими і більш зрозумілими. Також при текстовому описі системи може виникнути двозначність або непорозуміння зі сторони розробника і замовника, бачення і розуміння речей яких може кардинально відрізнятись, що веде до більших витрат часу на аналіз і добування вимог, а не на розробку.

Описана вище проблема неоднозначності, двозначності і непорозуміння може бути притаманна і графічним представлення роботи продукту, тому було створено уніфіковану мову моделювання задач, яка б вирішувала дану проблему. Такою мовою стала UML.

UML (скорочення Unified Modeling Language) – це уніфікована мова моделювання, яка використовується для графічного представлення роботи системи програмного забезпечення. Вона була створена для того, щоб вирішити проблеми проектування, візуалізації, документування, проектування і опису

програмних систем. Незважаючи на специфіку, UML може використовуватись для опису будь-якої із систем з реального світу [12].

Перш ніж описувати функціональні вимоги, необхідно описати вимоги з точки зору бізнесу і кінцевого користувача.

З точки зору бізнесу, програмний продукт, який розробляться в рамках дипломного проекту, має єдине призначення – це застосування його у сфері, пов'язаною з енергоефективністю; вирішення проблем з визначенням найбільш оптимальних і ефективних енергоносіїв для тієї чи іншої задачі.

З точки зору кінцевого користувача, він має мати змогу визначити найбільш ефективний енергоносій базуючись на енергії, яку він отримує з нього і його ціні.

Для опису функціональних вимог системи програмного продукту дипломного проекту було вирішено використовувати use case діаграму, так як вона найкраще підходить для опису можливостей і поведінки взаємодії кінцевого користувача і програмного забезпечення. Окрім діаграми, представлено список усіх можливостей системи і короткий опис й розгляд випадків зображених на діаграмі.

На рис. 2.1 зображено загальну use case діаграму з усіма випадками використання програмного забезпечення. Як можна побачити кінцевий користувач може визначати ефективність того чи іншого енергоносія використовуючи значення по замовчуванню або внести свої конкретні дані, якщо такі не співпадають з тими, які використовуються при обчисленні енергоефективності ресурсу самим програмним забезпеченням.

Також, незважаючи на те, що визначення ефективності енергоносіїв з використанням інформації про певний ресурс отриманої із зовнішніх джерел або від кінцевого користувача, є пріоритетною і первинною задачею програмного продукту, існує ще декілька вторинних функціональних можливостей системи, які потрібно реалізувати, так як без них система не функціонувала б в повній мірі, а деякі із головних задач не працювали б коректно. Усі функціональні вимоги перераховані і детально описані в таблиці 2.1.

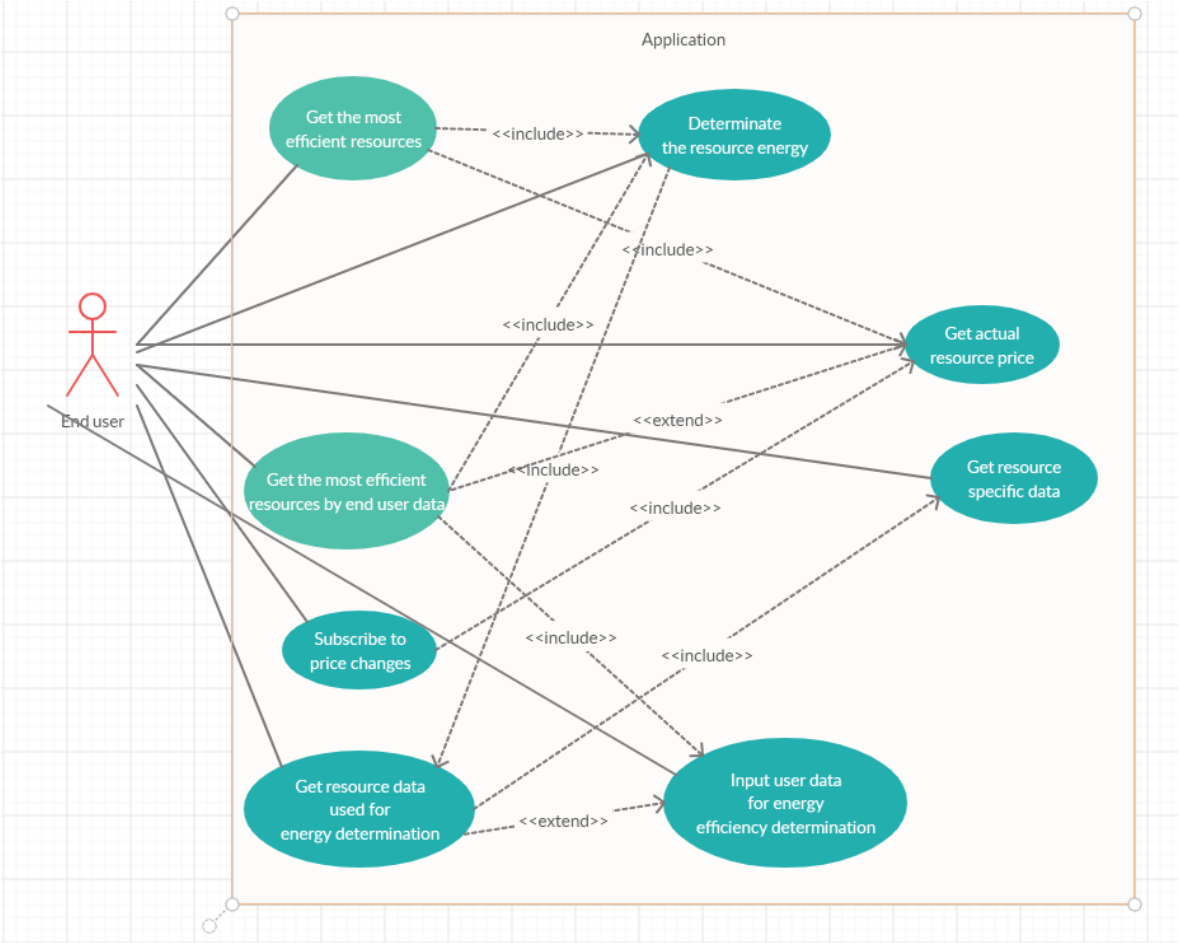

Рисунок 2.1 – Use case діаграма програмного забезпечення

Таблиця 2.1 – Функціональні вимоги програмного забезпечення

Функціональна можливість	**Опис**
Отримання найбільш вигідного і ефективного енергоносія базуючись на даних отриманих від зовнішніх джерел	Обчислення, виведення списку усіх значень по енергоефективності і визначення найбільш вигідного і ефективного енергоносія використовуючи інформацію про кількість енергії, яка даним ресурсом виділяється і його ціні. Для обчислень використовуються тільки дані, які беруться із системи.

Продовження таблиці 2.1

Отримання найбільш вигідного і ефективного енергоносія базуючись на даних отриманих від зовнішніх джерел	Обчислення, виведення списку усіх значень по енергоефективності і визначення найбільш вигідного і ефективного енергоносія використовуючи інформацію про кількість енергії, яка даним ресурсом виділяється і його ціні. Для обчислень можуть використовуватись дані, які беруться із системи, і дані, які внесені кінцевим користувачем. При виявленні відмінності користувацьких значень і значень, отриманих з системи, список й обчислені дані повинні бути відкореговані і виведені разом із усіма раніше представленими.
Визначення кількості енергії, яка виділяється певним ресурсом за отриманими даними	Обчислення і виведення кількості енергії, яка виділяється і використовується за отриманими даними. Використання конкретних даних, значення і кількість яких можуть міняти відповідно до ресурсу.
Отримання актуальної ціни ресурсу	Отримання і збереження актуальної ринкової ціни на певний енергоносій з офіційних джерел інформації. З плином часу ціна на ресурс може змінюватись, тому її потрібно постійно верифікувати і коригувати.
Отримання специфічних і актуальних даних певного ресурсу	Отримання і збереження специфічних даних відповідно до ресурсу з офіційних джерел інформації. З плином часу деякі з даних можуть змінюватись, тому їх потрібно постійно верифікувати і коригувати. Також їхня кількість залежить від специфіки обчислення виділеної енергії, тобто їх може бути необмежена кількість або не бути зовсім.

Кінець таблиці 2.1

Підписка на сповіщення про зміну ціни	Отримання сповіщень при зміні ціни на певний ресурс. Кінцевий користувач сам вирішує чи потрібно підписуватись чи ні. Передбачити легку і швидку відписку від раніше підписаних сповіщень.
Введення користувацьких даних для обчислення ефективності енергоносія	Можливість введення або коригування даних користувачем, які використовуються для визначення кількості енергії, яка виділяється, від певного ресурсу, або його ціни. Передбачити різну кількість даних, необхідних для визначення енергії певного ресурсу.
Отримання даних, які використовуються для визначення енергії певного ресурсу	Отримання і виведення кількості й значень даних, які використовуються для визначення енергії, що виділяється, певного енергоносія. Передбачити можливість коригування деяких значень за допомогою введення користувацьких даних.

2.2 Архітектура проекту програмного забезпечення визначення ефективності енергоносіїв

2.2.1 Загальна архітектура програмного забезпечення визначення ефективності енергоносіїв

Архітектура програмного забезпечення – це спосіб структурування програмного забезпечення на абстракції, кожна з яких має своє призначення, функції, фази роботи і місце використання [13].

Абстракція – це підхід до подання методів і значень змінних, як деяку окрему сутність, яка приховує деталі реалізації і прицип роботи, і з якою взаємодіють тільки через точки входу (endpoints) або публічні методи. Спосіб

відділення від низькорівневої взаємодії між програмою і апаратним забезпеченням або взаємодії однієї абстракції від іншої, також є абстракцією[14].

В залежності від потреб, програмне забезпечення, що проектується, може мати кілька шарів абстракції. Вони надають програмі набагато більшу гнучкість і здатність до змін або випралень. Потрібно також враховувати те, що кожен новий рівень повинен бути виправданим, так як занадто велика кількість абстракції може значно сповільнити її роботу у випадках, коли дану задачу можна було б виконати з використанням меншого абстрагування [13].

Для початку при проектуванні цілої системи потрібно визначити яка загальна архітектура буде використовуватись, зсилаючись на вимоги, які були вище описані. Для того, щоб визначитись із архітектурою, потрібно перерахувати усі відомі і популярні приклади архітектур й паттерни, що зараз використовують при проектуванні програмного забезпечення та їхні плюси й мінуси.

Одними з найпопулярніших видів архітектур на даний час є:

- Монолітна;
- Клієнт-серверна (front-end and back-end);
- Шарова;
- Мікросервісна.

Монолітна архітектура передбачає, що вся функціональність, від генерування вигляду програми (view) до бізнес-логіки, розміщена в межах одного проекту. Монолітна архітектура є однією з найстаріших і, зате, найвідоміших методів проектування системи. Вона є інтуїтивно простою в проектуванні й розумінні, не має надмірної абстракції, через що показує найкращі результати по продуктивності, є найбільш вивченою, завдяки чому має велику кількість паттернів і прикладів до проектування, так як усі компоненти тісно зв’язані, є однією з найбільш безпечних [15].

Також слід зауважити, що саме поняття «монолітна архітектура» є досить обширним, так що неможливо дати остаточно правильну і точну класифікацію певної системи як монолітну чи не монолітну.

В сучасному світі все більше компаній відходять від монолітів через деякі критичні мінуси, які випливають при роботі з нею. Те, що просто і зрозуміло для одного, викликає труднощі для групи розробників, які працюють над одним проектом. Це пов’язана із залежністю один від одного, менша можливість внесення нових змін і модифікації вже існуючого коду, конфлікти між версіями й постійні блокування один одного при спробі змін через тісну взаємодію компонентів програми. Програмне забезпечення написане з допомогою монолітної арітектури не відзначається високою надійністю і масштабованістю. При переході програми в режим високого навантаження вона не в змозі обробити кожен запит через специфіку побудови, а збільшення пропускної здатності за допомогою збільшення обчислювальних потужностей, зазвичай, є неможливим або неефективним.

Підсумовуючи усі плюси і мінуси, можна відмітити такі

Плюси:

- простота у використанні та розробці однією людиною;
- висока швидкодія і безпека;
- велика кількість відомих шаблонів.

Мінуси:

- складна робота у групі людей;
- низька надійність і масштабованість;
- погана гнучкість і здатність до внесення змін.

Клієнт-серверна архітектура – це спроба відділити рівень представлення (view) від бізнес-логіки. Є однією з найвдаліших і найпопулярніших реалізацій монолітної архітектури. Так як є, по суті, однією із реінкарнацій монолітної архітектури, то їй притаманні більшість плюсів попередньої, плюс вигода від розділення представлення і бізнес-логіки [16].

Серед власних плюсів слід зазначити краща структурованість, завдяки розділеності, з якої випливає менша залежність одних компонентів від інших, краща робота над проектом у невеликих групах, можливість використання інтерактивного представлення і відділення тільки від одного виду клієнта, якщо

вони використовують ті ж дані. З монолітної архітектури вона зберігає відносну простоту, високу продуктивність і вивченість серед розробників.

Не зважаючи на те, що клієнт-серверна архітектура нівелює проблеми по внесенні змін і роботі в невеликій групі, все одно, при роботі над клієнтською частиною або серверною частиною більше, ніж однією людиною, часто виникають конфлікти між версіями і проблеми із внесенням змін або додаванням нової функціональності. Також зберігаються проблеми із надійністю і маштабованістю, так як бізнес-логіка і представлення залишаються одним цілим. Крім цього, додаються власні мінуси, такі як безпека. Так як представлення і бізнес-логіка розділені, то тепер існують дві сторони взаємодії і тепер, окрім обмеження доступу до даних на клієнтській частині, потрібно ще й обмежувати дані на серверній частині.

Підсумовуючи усі плюси і мінуси, можна відмітити такі

Плюси:

- покращена зручність у роботі невеликими групами;
- збереження швидкодії і кількості шаблонів;
- здатність до використання більше, ніж одним клієнтом з деякими умовностями.

Мінуси:

- все ще велика складність при роботі у великій команді;
- низька надійність і масштабованість;
- недостатня гнучність кожної із частин;
- підвищені вимоги до безпеки.

Шарова архітектура – спроба розділити бізнес-логіку програмного забезпечення на декілька шарів роботи із даними. Найчастіше це рівень доступу до даних (Data access), рівень бізнес-логіки або сервісів обробки даних (Business layer), рівень зовнішніх точок входу (API layer). Також можливими шарами можуть бути рівень, який відповідає за авторизацію й аутентифікацію і рівень представлення, який, найчастіше, є окремою програмою, як у випадку клієнт-серверної архітектури [17].

Таке розбиття на шари збільшує зручність при роботі у середніх і великих командах, так як компонент кожного рівня розділений, то і перетинань і блокувань з іншими компонентами і розробниками є меншою. Зберігає плюси клієнт-серверної архітектури зі здатністю використання одним і більше клієнтами і нівелює умовностями про те, що всі клієнти повинні отримувати однакові дані, так як для кожного клієнта можна розробити свій шар з точками входу.

Серед мінусів можна відзначити зростаючу складність і рівень абстракції, який може, в деяких випадках, вплинути на продуктивність в гіршу сторону. Також при наявності більше, ніж одного клієнта і, відповідно, більше, ніж одного рівня входу, може призвести до розбухання коду. Проблема полягає в тому, що на кожному рівні сутності хоч і логічно розділені, вони все одно прив’язані до роботи з конкретним шаром, тому може виникнути ситуація, що при додаванні підтримки нового клієнта, окрім шару точки входу, потрібно буде додати шари бізнес-логіки і доступу до даних. Якщо цього не зробити, то програмне забезпечення може втратити структурованість. Також, так як усі шари розбиті тільки логічно, зберігаються проблеми з надійністю і масштабованістю.

Підсумовуючи можна виділити такі плюси і мінуси

Плюси:

- краща гнучкість і здатність до внесення змін;
- зручніша робота у середніх та великих групах;
- краща взаємодія між декількома клієнтами.

Мінуси:

- зростаюча складність і набухання коду;
- можливе зменшення продуктивності;
- все ще підвищені вимоги до безпеки.

Мікросервісна архітектура – це спосіб проектування програмного забезпечення, який полягає у повному розділенні програми на підпрограми, які працюють незалежно одна від одної в автономному режимі. Мікросервісна архітектура є досить молодим, але дуже популярним зараз способом представлення програмного забезпечення. Раніше мала значні проблеми з

продуктивністю, так як повинна була запускатись на віртуальних машинах. З відкриттям контейнеризації більшість проблем було усунуто і вона стала однією з основних практик при розробці програмних продуктів.

Серед плюсів слід відзначити високу масштабованість, надійність, гнучкість і зручність при розробці. Так як кожен компонент є окремою невеликою програмою, то зміни, які у них здійснюються, не зачіпають усіх решту. З цього також полягає і більша масштабованість, тепер, якщо одна функціональність потребує більшої пропускної здатності, а інші – ні, ми задіюємо збільшені обчислювальні потужності на додану функціональність, а не на цілий проект. З цього ж випливає і надійність, так як кілька запущених сервісів є взаємозамінними при неполадках одного із них [18].

Серед мінусів слід відзначити просідання продуктивності, так як при збільшенні сервісів зростає і затримка передачі даних між ними, що може призвести до повільнішої обробки запитів для користувача. Також слід відзначити ще те, що вимоги до безпеки значно підвищили, як у випадку переходу з монолітної до клієнт-серверної архітектур. Тепер точок входу є рівно стільки, скільки й сервісів, тому потрібно передбачити складну систему авторизації і аутентифікації.

Підсумовуючи усі плюси і мінуси можна виділити такі

Плюси:

- розділеність апаратних ресурсів на кожен окремо запущений сервіс;
- незалежність виконання кожного сервісу;
- велика масштабованість і надійність системи даної архітектури;
- зручність використання і внесення змін;
- підтримка будь-якого виду клієнту при додаванні відповідного сервісу.

Мінуси:

- складність у проектуванні;
- просідання продуктивності і затрики передачі даних;
- великі вимоги до безпеки.

Проаналізувавши усі вимоги і приклади можливих архітектур можна виділити наступні пункти:

- робота над проектуванням або розробкою проекту буде здійснюватись одним розробником;
- необхідно, щоб деякі частини проекту працювали в автономному режимі, щоб підтримувати постійний моніторинг даних;
- в системі буде тільки один кінцевий користувач, тому вимоги до безпеки не будуть великими;
- потрібна відбуватись взаємодія між різними компонентами;
- уся взаємодія повинна проходити в автоматичному режимі.

Взявши до уваги вище описані пункти, можна зробити висновок, що кращим вибором для загальної архітектури проекту є мікросервісна архітектура з ряду причин:

- існує чітка розділеність компонентів по ролям і призначенні;
- для зменшення навантаження на систему є сенс винести сервіс для обчислень і моніторингу даних, так як перший буде буде обробляти вхідні запити, а другий – працювати постійно в автономному режимі;
- різноплановість задач не підпадає під шарову архітектуру;
- найвищий показник надійності і масштабованості, що може бути актуальним при збільшенні обчислень або кількості значень ресурсів для моніторингу;
- незалежна модифікація кожного сервісу значно пришвидшує розробку і зменшує розбухання коду;
- через невелику кількість сервісів вплив затримки передачі даних майже не буде відчуватись;
- вимоги до безпеки також не будуть високими, тому ця проблема також не буде відчутною.

Щоб зобразити структуру і взаємодію між сервісами було використано діаграму мікросервісів (Microservice diagram) від IBM Cloud.

На рис. 2.2 зображено приблизний вигляд мікросервісної архітектури, яка буде використовуватись у проекті. Як можна побачити функціональність розбита на 3 сервіси: сервіс обчислень або Calculation service, сервіс моніторингу або Monitor service, сервіс сповіщень або Notification service. Також окремою

сутністю у цій схемі буде клієнтська частина або UI. Слід зауважити, що усі 3 сервіси будуть використовувати одну базу даних, тому, щоб не дублювати програмний код у кожному сервісі, було вирішено додати бібліотеку DataAccess, яка б якраз містила усю функціональність доступу до бази даних.

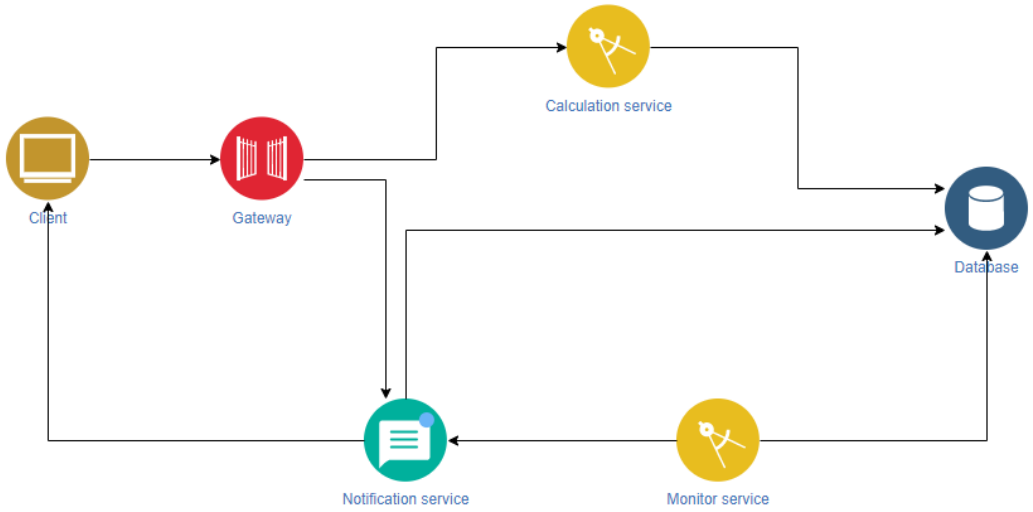

Рисунок 2.2 – Діаграма мікросервісної архітектури проекту

2.2.2 Архітектура Calculation Service

Calculation service або сервіс обчислень – це сервіс, який буде містити у собі бізнес-логіку обчислень енергії, яка виділяється, певного ресурсу, маніпулювати даними, що використовуються для обчислень, отримання користувацьких даних і формування списку всіх ресурсів з обчисленими показниками ефективності і вигідності енергоносія.

Розглядаючи вимоги до Calculation service, можна прийти до висновку, що у нього слід застосувати стандартну шарову архітектуру з сутностями, які оперують забором даних, їх обробкою і видачею результату. Однак присутні деякі деталі, які при аналізі вимог є не помітними, але при проектуванні випливають їхні недоліки.

На рис. 2.3 зображено початкову архітектуру за допомогою діаграми класів. Як можна тут помітити, сутність GatherService відповідає за витягування даних певного енергоносія, а AnalyzeService – за обчислення і виведення

результатів в контроллер. Перша проблема полягає в тому, що ми не маємо тих даних, які забирає GatherService, тобто ми не зможемо відправити їх для користувача для коригування. Наступна проблема – це прив'язування сервісів і методу контроллера до одного ресурсу. Для кожного ресурсу потрібно окремо створити GatherService, AnalyzeService і метод контроллера, що призводить до змін на усіх шарах архітектури, хоча при додаванні підтримки нового ресурсу потрібно тільки вказати алгоритм аналізу і забору даних цього ресурсу. Остання проблема – при великій кількості методів контроллера унеможливлюється забір усіх результатів обчислень і формування списку.

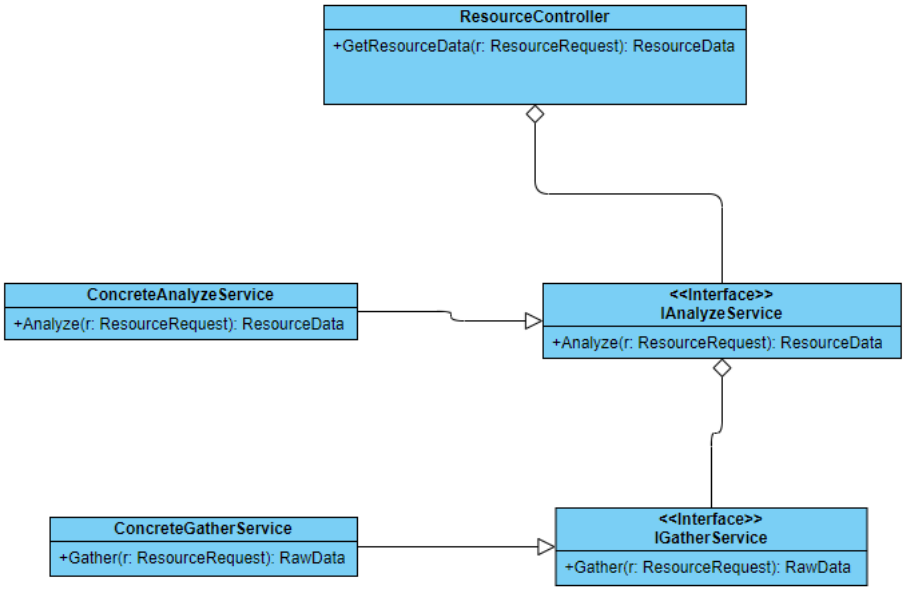

Рисунок 2.3 – Початкова діаграма класів Calculation service

Тож враховуючи недоліки такої структури класів слід додати сутність, яка б зберігала дані забору GatherService, яка б не була прив'язана до конкретного сервісу і відділяла б GatherService і AnalyzeService від контроллера, а також була для певного ресурсу, за яким би її можна було ідентифікувати, щоб її можна було викликати як для формування списку, так і для отримання результатів обчислень певного енергоносія.

Слідуючи цим вимогам на рис. 2.4 можна побачити нову сутність ResourceFactory, яка містить в собі GatherService і AnalyzeService і зберігає дані

забору для GatherService. Також вона містить тип ресурсу, для якого здійснюються обчислення і результати забору даних для обчислення за допомогою GatherService.

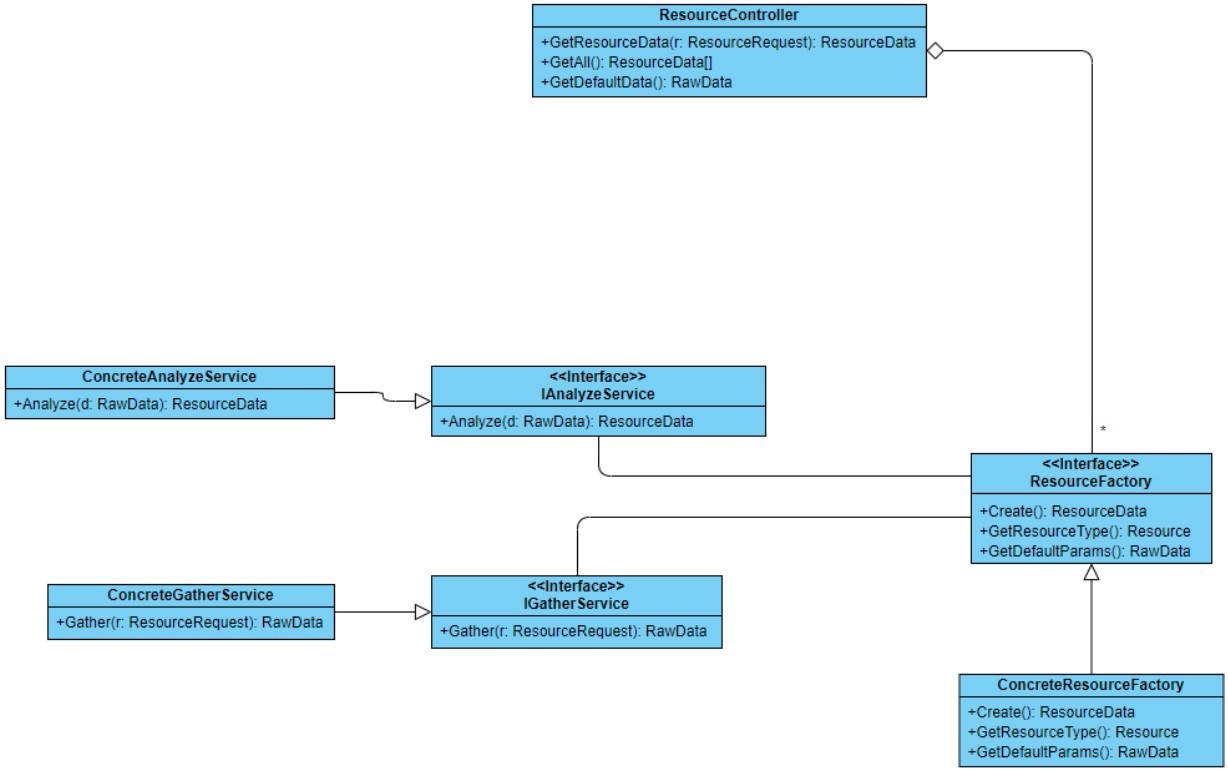

Рисунок 2.4 – Кінцева архітектура Calculation Service

2.2.3 Архітектура Monitor Service

Призначення Monitor Service полягає в тому, щоб постійно працювати без припинення роботи. Серед його можливостей слід відмітити витягування даних з різних джерел інформації і оновлення вже існуючих. На основі цього можна зробити висновок про необхідність розділення сутностей на тих, які б дані отримували, і тих, які б ці дані фіксували і записували.

Як можна побачити, на рис. 2.5 зображено діаграму класів архітектури Monitor Service. Сутності тут поділені на DataSourceService і DataCollector. DataSourceService займається добуванням даних з певного джерела інформації по вказаному шляху, по якому вона знаходиться. DataCollector, в свою чергу,

приймає цю інформацію, збирає її і зберігає. CollectingRunner у цій ієрархії займається запуском усіх DataCollector'ів.

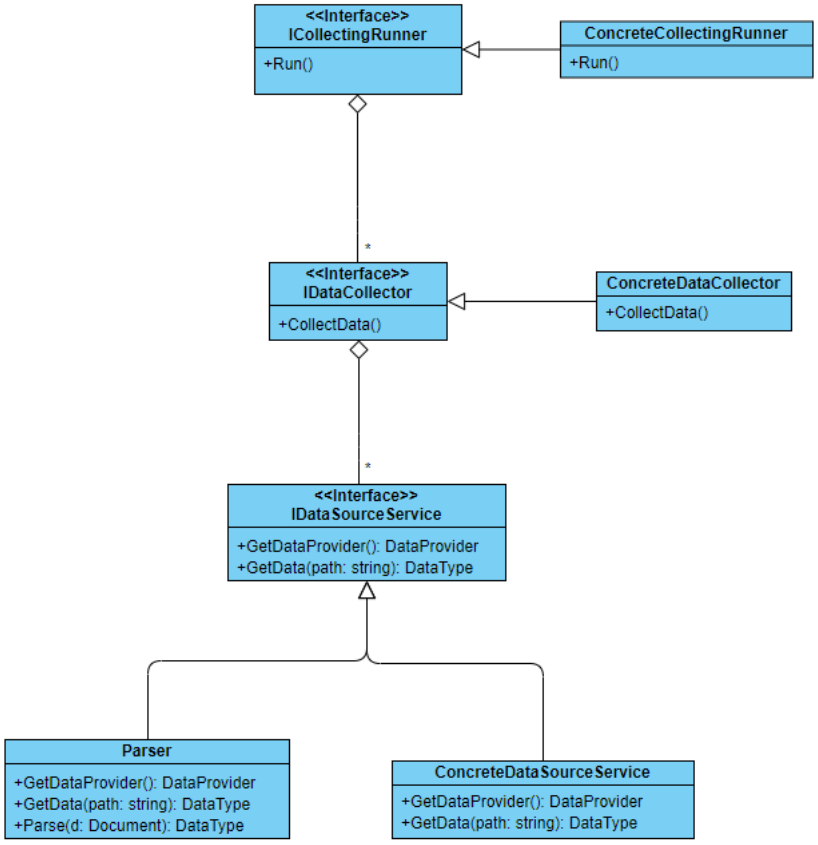

Рисунок 2.5 – Архітектура Monitor Service

2.2.4 Архітектура Notification Service і UI

Notification Service використовується у системі для генерації сповіщень, які передаються на кінцевого користувача. UI – це графічний інтерфейс, з яким взаємодіє кінцевий користувач.

Notification Service і UI потрібно розглядати разом, так як вони дуже тісно пов'язані. Всі операції, які здійснюються в Notification Service, а також функціональність, використовуються в UI.

На рис. 2.6 зображено взаємодію користувача з UI при підписуванні і відписуванні на сповіщення про зміну ціни енергоносіїв. Як видно, при

виконанні дій користувача по підпискам, UI використовує для цього методи Notification Service, в якому реалізована ця функціональність.

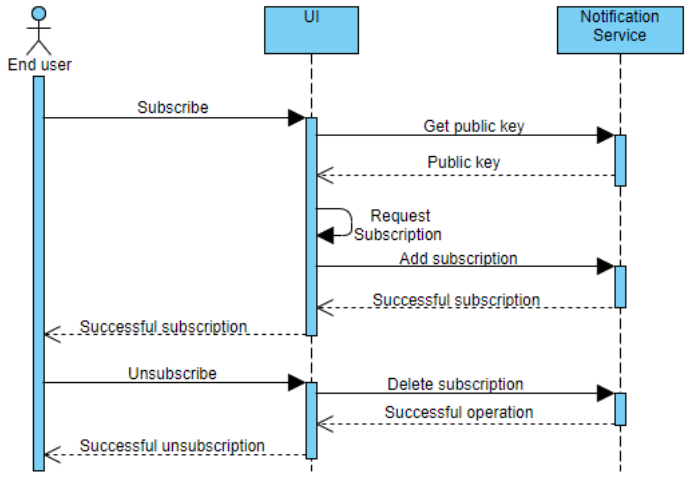

Рисунок 2.6 – Діаграма взаємодії Notification Service і UI

На рис. 2.7 відображено сценарій взаємодії Monitor Service, Notification Service і UI. Як можна побачити, при перевірці актуальних цін і тих, що є збереженими в системі, Monitor Service виконує метод Notifcation Service, коли ціни не співпадають і потрібно здійснити корекцію. Notification Service, при цьому, відправляє запит на UI для того, щоб відправити сповіщення про зміну цін тих людям, які на неї підписані.

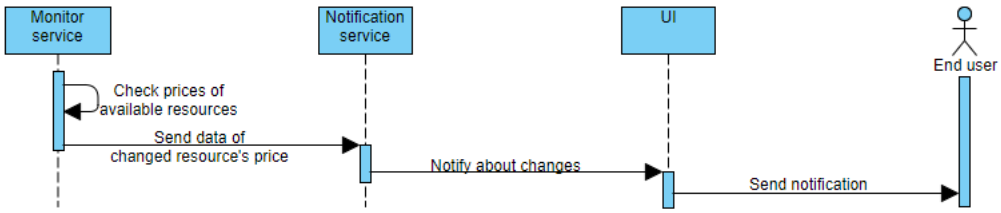

Рисунок 2.7 – Діаграма взаємодії Monitor Service, Notification Service і UI

2.2.5 Архітектура DataAccess

Як вже говорилося вище, DataAccess – це бібліотека, яка містить у собі засоби для роботи з базою даних. Вона потрібна для того, щоб відокремити

логіку роботи з базою даних від існуючої сторонньої реалізації для більшої гнучкості.

На рис. 2.8 зображено діаграму класів архітектури бібліотеки DataAccess. Вона реалізовує паттерн UnitOfWork і Repository. Вони є простими в користуванні і надають всі CRUD операції над сутностями бази даних.

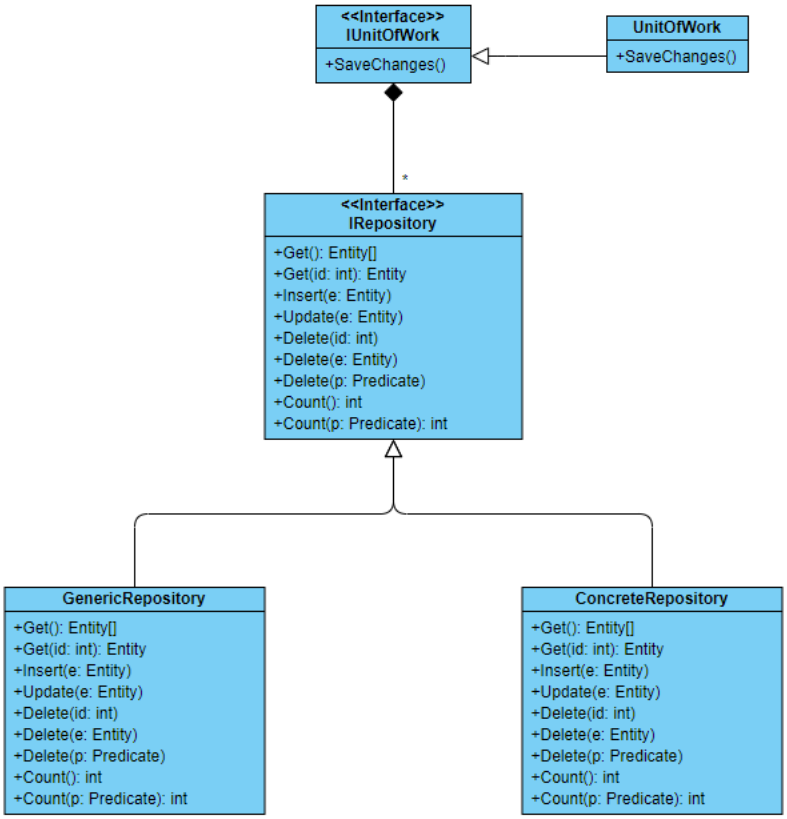

Рисунок 2.8 – Архітектура DataAccess

2.3 Структура бази даних програмного забезпечення визначення ефективності енергоносіїв

Щоб спроектувати структуру бази даних, потрібно виходити із того, яка функціональність повинна використовуватись і реалізовуватись. Для роботи з енергоносіями потрібно створити структуру, в якій можна було б легко записати різні специфічні дані до кожного ресурсу, їхню ціну, джерела звідки ціна й специфічні дані беруться і підписки на сповіщення про зміну ціни енергоносія кожного кінцевого користувача.

На рис. 2.9 зображено структуру бази даних, яка б могла відповідати описаним вимогам. Для кожної підписки була створена таблиця Subscriptions, яка містить дані про підписаних користувачів, які необхідні для відправки сповіщень. Головною сутністю можна вважати ResourcePrice, яка містить дані про енергоносій з таблиці Resources, його ціну, посилання на джерело інформації і шлях, по якому її можна знайти, а також регіон, в якому діє вказана інформація. Для додавання специфічних даних кожного ресурсу, потрібно створити таблицю, яка б містила такі дані і посилання на джерело інформації, з якого ці дані беруться.

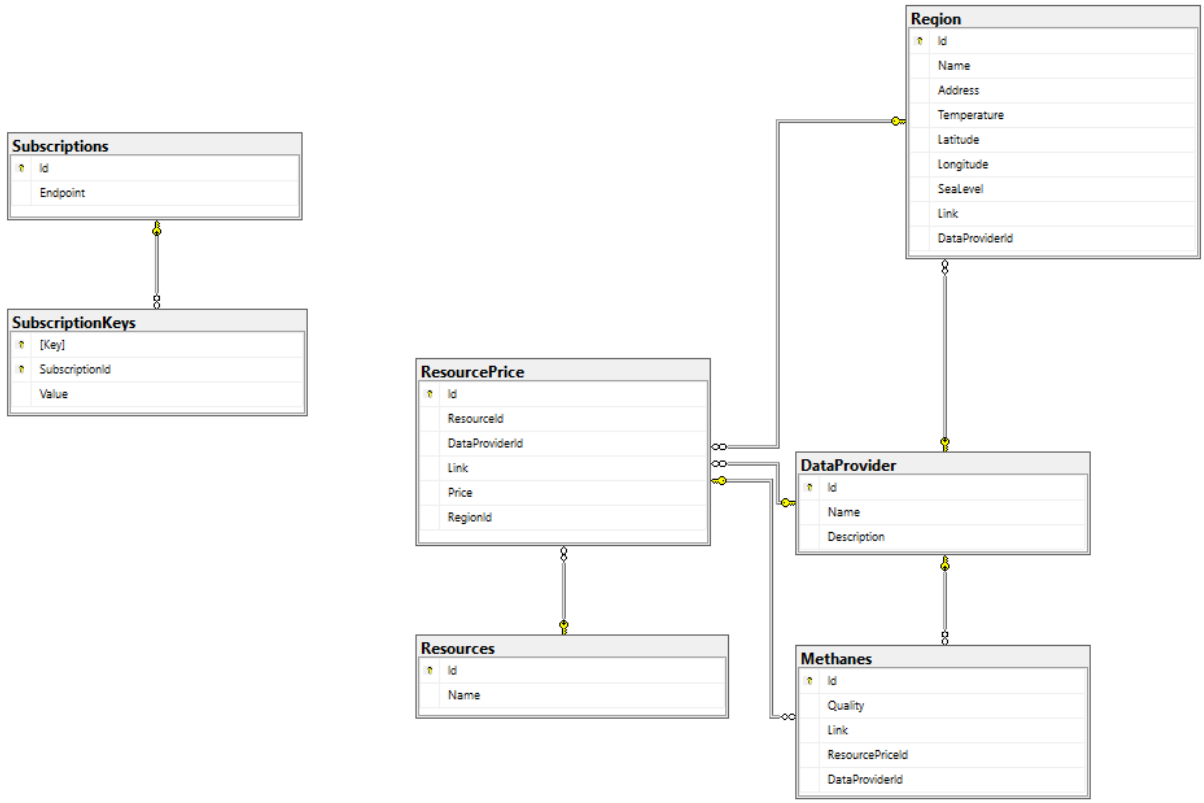

Рисунок 2.9 – Структура бази даних, що використовується проектом

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ВИЗНАЧЕННЯ ЕФЕКТИВНОСТІ ЕНЕРГОНОСІЇВ

3.1 Розробка серверної частини програмного забезпечення визначення ефективності енергоносіїв

3.1.1 Опис інструментів та технологій для розробки програмного забезпечення визначення ефективності енергоносіїв

Під час роботи над реалізацією дипломного проекту і моделей, описаних раніше, було вирішено використовувати технології і інструменти, які надаються, згідно роботи із технологією .NET Core.

.NET Core – це вільний, донедавна альтернативний в середовищі .NET, крос-платформний фреймворк. Містить в собі крос-платформну реалізацію віртуальної машини CLR – CoreCLR, яка керує виконанням програм написаних з допомогою .NET Core. Так як зникла залежність від Microsoft Windows, програми можуть запускатись на будь-якому середовищі під управлінням будь-якої з операційних систем (ОС Windows, Linux, Mac OSX). Підтримує створення усіх застосунків із звичайного .NET Framework. Є модульним, компоненти якого є пакетами NuGet. Окрім традиційних можливостей, які були доступні з використанням Visual Studio, має власний CLI [19, 20].

Також варто згадати й те, що у майбутньому релізі, .NET Core стане .NET 5 – новою ланкою у екосистемі .NET, яка стане уніфікованим рішення як для створення класичних десктопних застосунків, так і для веб-застосунків, мобільних застосунків, застосунків інтернету речей та інших. В результаті цього .NET Framework переводиться в статус legacy, тобто такого, підтримка і виправлення критичних помилок якого буде надалі здійснюватись, але з кожною новою версією в нього не буде привноситись нова функціональність, якої немає у .NET 5 [21].

Окрім вибору платформи для розробки програмного продукту, потрібно також визначитись із вибором мови програмування, на якій програма буде

написана. Платформа .NET підтримує близько 20 мов програмування, але Microsoft рекомендує використовувати мови, які є вбудованими і йдуть разом із .NET Core. Тому було вирішено використовувати мову програмування C#.

Так як здійснюється розробка серверної частини веб-застосунка, то для таких цілей існує ASP.NET. Active Server Pages for .NET або ASP.NET – це технологія, веб-фреймворк, який містить у собі засоби і компоненти для створення динамічних веб-застосунків і веб-сервісів. Раніше використовувалась для створення веб-сторінок із динамічним контентом. Зараз використовує розділеність бізнес-логіки і представлення. Потрібно згадати той факт, що для .NET Core використовується ASP.NET Core, яка, на відміну від ASP.NET під управлінням .NET Framework, не обмежена тільки Microsoft Windows і веб-сервером IIS, є крос-платформною і використовує власний веб-сервер Kestrel [22].

Незважаючи на те, що, в основному, усі застосунки, які розробляються під платформою .NET, для розробки використовують середовище Visual Studio, для розробки дипломного проекту буде використовуватись IDE від компанії JetBrains – Rider. Окрім того, що він має усі можливості повноцінної Visual Studio, також у нього вбудований статичний аналізатор коду ReSharper, який не тільки допомагає дотримуватись чистоти і ефективності програмного коду, а й пришвидшує розробку через великі можливості по генерації коду, який потрібно було б вводити вручну [23].

3.1.2 Розробка загального шаблону проекту програмного забезпечення визначення ефективності енергоносіїв

Перед роботою над серверною частиною і сервісами для неї потрібно визначитись із кістяком або каркасом того, як має виглядати загальний проект. Для організації роботи і кращої структурованості проекту потрібно створити папку під кожну сутність, яка використовується в ході створення і розробки програмного забезпечення, а також чітко визначити межі їхнього призначення,

роль, мету застосування і те, чи можуть вони бути перевикористані для вирішення схожих задач.

На даний момент єдиними сутностями, якими можна оперувати – це сервіси загальної мікросервісної архітектури, які вище описані. Для зручності загальний проект буде носити робочу назву “Sulfur”. Папки, для кожного із сервісів, будуть носити назву відповідно: “CalculationService”, “MonitorService”, “NotificationService”. Для їхнього іменування використовується стиль під назвою регістр Паскаля (Pascal case), який передбачає написання кількох слів разом, але кожне слово повинно починатися з великої букви.

Так як проект розробляється на платформі .NET, то використовується регістр Паскаля як умовно прийнятий спільнотою і регулюється конвенцією написання коду (code style convention). Разом до цього потрібно додати те, що кожен проект або сервіс такоє має спеціальний спосіб іменування. Кожен новий рівень простору імен (namespace) повинен містити назву попереднього розділеного крапками, а головна папка, в свою чергу, повинна співпадати назвою з назвою простору імен, в якому знаходиться.

У кожній із папок проекту буде розміщуватись програмний код, що використовується для функціонування і виконання програми. Але для того, щоб прискорити розробку і мати більший контроль над написаним функціоналом, потрібно створити проекти для тестування функціоналу кожного із сервісів.

Як можна замітити на рис. 3.1 самі папки сервісів містять папки “src”, в якій буде розміщуватись функціональність сервісу, і “test”, яка буде містити в собі тести для проектів розміщених в папці “src”. Слід зауважити, що папки такого виду будуть повторюватись для кожного із сервісів і не можуть винестись на нижчий рівень. Це зроблено тому, що кожен сервіс є незалежним і не може прямо посилатись один на одний, а якби папки були рівнем нижче, то тести не мали б доступу до функціональності, яку тестують. Таке розділення допоможе тримати кілька проектів в одному сервісі і тестів для них в одному місці без втрати структурованості і лаконічності.

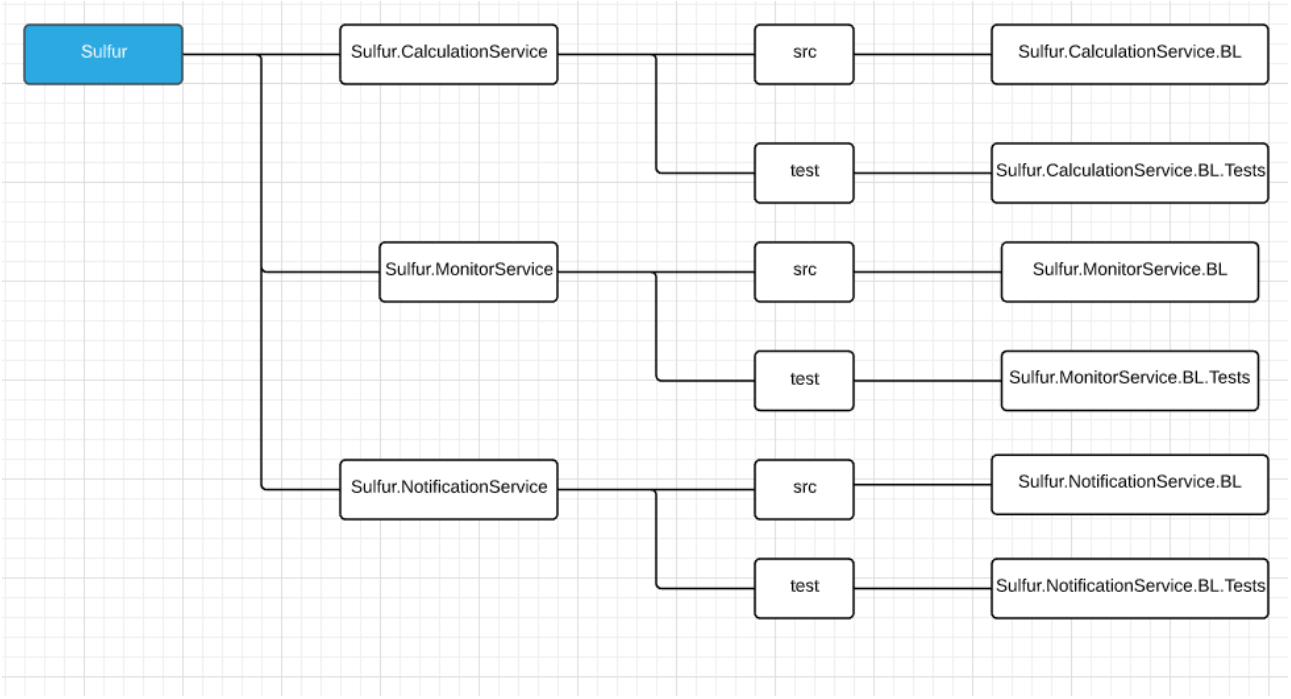

Рисунок 3.1 – Файлова структура проекту

Говорячи про проекти, потрібно згадати і те, що у платформі .NET уже є схоже поняття, яке фігурує із поняттям рішення (solution). Суть проектів у .NET полягає в тому, що це є повністю функціонуюча одиниця, яка описує його структуру, залежності (dependencies) та різні метадані. Якщо проект містить точку запуску, то його можна виконати, інакше можна залежати від нього і використовувати функціональність, яку він надає. Суть рішень – це контроль усіх проектів, наявних в ньому. Він являється найвищим рівень в програмного забезпеченні, яке розробляється. Саме він вказує середовищу розробки які файли, папки і проекти відносяться до нього, які можна використовувати або на які посилатись.

З першої точки зору може здатись, що для створення окремого сервісу потрібно використовувати саме проекти, тому що:

- вони є окремими одиницями програмного забезпечення;
- ними легше буде управляти при розробці, тому що вони усі будуть видимі в межах середовища розробки;
- для окремих проектів набагато легше використовувати контейнеризацію, мова про яку піде потім.

Але в цього підходу є декілька значних мінусів, які суперечать обраній мікросервісній архітектурі і структурі самого проекту:

- можливість посилатися не тільки на проекти, функціональність яких використовується, а й на інші незалежні сервіси;
- можливе розбухання коду, викликане тим, що весь код проекту буде розміщуватись в ньому;
- структура такого проекту буде повторювати попередню, яку було відхилено через надмірність і недостатню структурованість.

Тому, зважаючи на це, можна зробити висновок, що для даного проекту більше підходить принцип створення сервісу на рішення (service per solution). В такому випадку, ідеологічно проект розбивається на підпрограми, які взаємодіють між собою.

Говорячи про проекти і рішення, потрібно також зачепити структуру бібліотеки DataAccess, яка буде розміщувати не тільки у окремому рішенні, а й у окремому репозиторії. Це зв’язано з тим, що сервіси хоч і будуть на нього посилатись, його призначення зовсім інше. Репозиторій, створений для DataAccess буде містити у собі всі можливі бібліотеки і залежності, які будуть використовуватись у репозиторії “Sulfur”. Такий репозиторій буде містити назву “Sulfur-dependencies”, яка якраз вказує на те, що в ній розміщені залежності головного репозиторію. А так як DataAccess буде відноситись до загального проекту “Sulfur”, то його рішення буде містити назву “Sulfur.DataAccess”.

На рис. 3.2 зображено структуру репозиторію залежностей, в якому знаходиться бібліотека “Sulfur.DataAccess”. Також, щодо бібліотек, можливий випадок, коли крім проекту з функціональністю, немає жодного. Тоді, для простоти і зручності, відкидається необхідність у додаванні до назви проекту нового закінчення і назви проекту і рішення можуть співпадати.

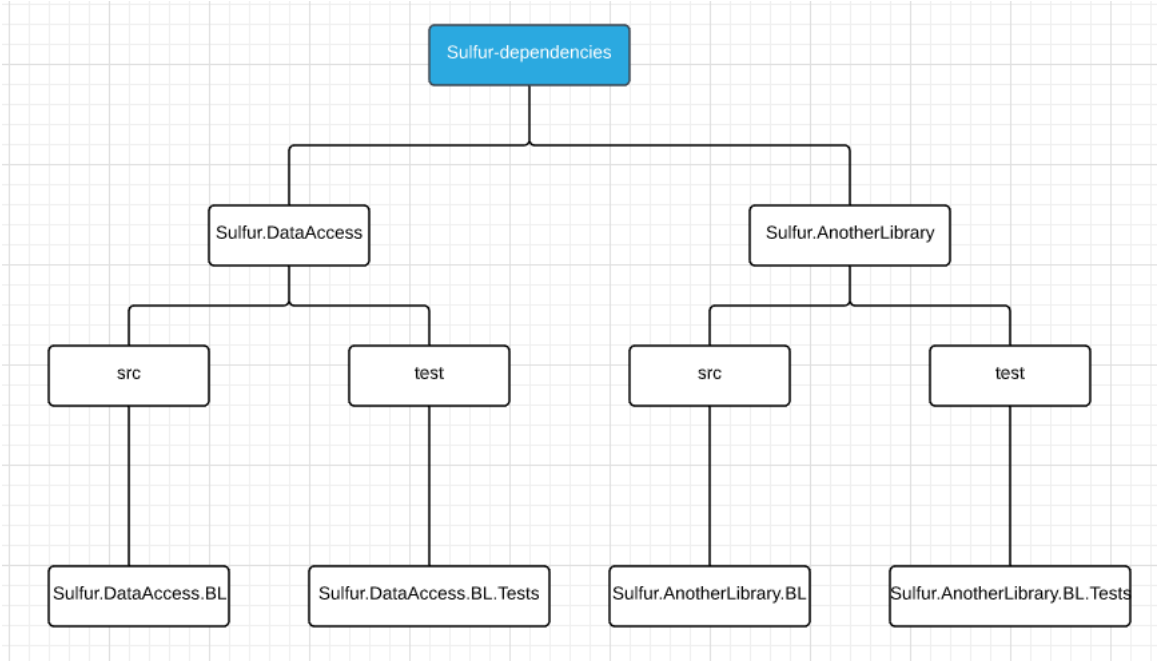

Рисунок 3.2 – Файлова структура репозиторію залежностей

3.1.3 Розробка Calculation Service

Для розробки Calculation Service, архітектура якого була описана раніше, потрібно створити проект, який буде мати змогу приймати веб-запити і відповідати на них. Для цього можна скористатися одним із готових шаблонів середовища розробки Rider або додати усі залежності в проект вручну. Так як додавання веб-сервісу – задача досить тривіальна, краще створити його через наявний шаблон, тим більше він не містить великої кількості надмірних речей і може бути легко перекофігурованим.

При створенні Web API проекту, під капотом створюється контроллер, файл Program, в якому знаходиться одноіменний клас, який, в свою чергу, являється точкою запуску програми, і Startup, в якому знаходиться конфігурація веб-сервісу.

Якщо коротко говорити про клас Startup, то він реалізовує специфікацію OWIN (Open Web Interface for .NET), яка дозволяє зменшити залежність від веб-серверу на якому запускається застосунок, а також дозволяє зробити її таким, яке б саме себе забезпечувало (self-hosted). Для цього використовується IoC-контейнер, який підтримує ін’єкцію залежностей (Dependency injection) і конвеєр

серединної обробки (middleware). Ін’єкція залежностей – це механізм, який реалізує принцип інверсування залежностей (Dependency inversion principle) з принципів SOLID. Його призначення полягає в тому, щоб автоматично створювати об’єкти і підставляти їх у місця, які цього потребують. В ASP.NET Core, найчастіше, це контроллери. Middleware – це ланка у конвеєрі обробки запиту, яка вказує як той чи інший запит має оброблятись [24].

Окрім головного API проекту, потрібно створити ще два інші проекти: BL і Common. Проект Common буде містити сутності і абстракцію, які вибудовують архітектуру сервісу, а проект BL – їхню реалізацію. Таке розділення зумовлено тим, щоб архітектура могла розширюватись незалежно і мати більше, ніж одну реалізацію, щоб потім використовувати ту, яка необхідна. Так як сама бізнес-логіка буде розміщуватись у проекті BL, то і юніт-тести потрібно створювати тільки для неї.

В даному випадку, для юніт-тестування використовується фреймворк xUnit. Для створення проекту, який би містив функціональність, необхідну для головного проекту, потрібно вибрати шаблон Class Library. Важливим моментом є вибір фреймворку проекту, так як при виборі .NET Core потрібно впевнитись у сумісності версії, тому що проект з нижчою версією фреймворку не може використовувати проект з версією нижчою. Перед додавання самих проектів потрібно додатково додати ще й папку рішення (solution folder), щоб слідувати обраній структурі проекту, а також, при додаванні проектів, вибрати їм фізичний шлях, який би співпадав папкам рішення, так як вони є всього лиш абстрактними шляхами, які зберігаються в описі рішення.

На рис. 3.3 зображено структуру Calculation Service з усіма створеними для неї проектами. Проект Sulfur.CalculationService.Api міститиме засоби ASP.NET Core по передачі даних, Sulfur.CalculationService.Common – архітектуру сервісу, Sulfur.CalculationService.BL – реалізовані сутності з проекту Sulfur.CalculationService.Common.

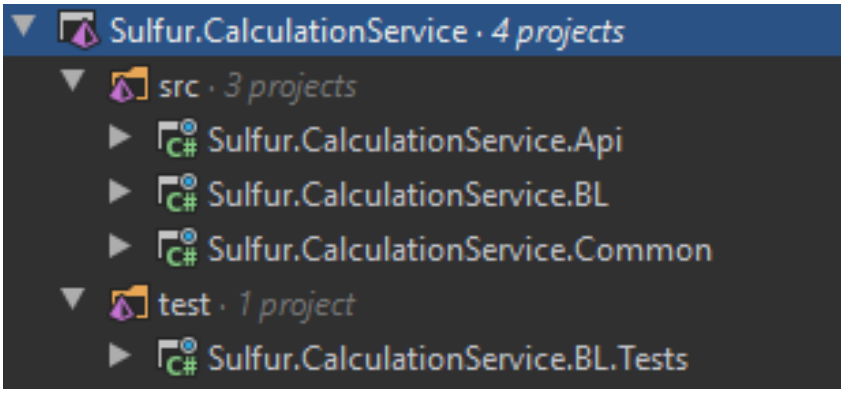

Рисунок 3.3 – Структура рішення Calculation Service

Щоб реалізувати архітектуру Calculation Service, потрібно, передусім, створити контроллер, який би обробляв запити стосовно енергоносії. Запити будуть відбувати по шляхах: *GET /resources, GET /resources/{resourceType}, GET /default-data/{resourceType}*. Так як запит на отримання даних по енергоефективності конкретного ресурсу (*/resources/{resourceType}*) може містити як числове, так і символьне значення, через використання перечислень (enum), потрібно створити два контроллера окремо, через те, що друга частина шляху на отримання даних про вхідні параметри для обчислень (*/default-data/{resourceType}*) конфліктувала б з першим шляхом, тому що система маршрутизації не змогла б знайти енергоносія з таким іменем. Тому було створено ResourceController і DefaultDataController.

Як можна побачити, на рис. 3.4 створюється контроллер ResourceController, з двома методами, кожен з яких обробляє конкретний шлях, які є асинхронними. Асинхронність – це процес виконання коду, коли під час очікування результатів обробки зовнішньої операції може виконуватись інша частина програмного коду. Здійснюється за допомогою неблокуючого виконання операцій вводу-виводу (IO-bound) [25]. В .NET і, конкретно, в C# використовується TAP (Task-based Asynchronous Pattern).

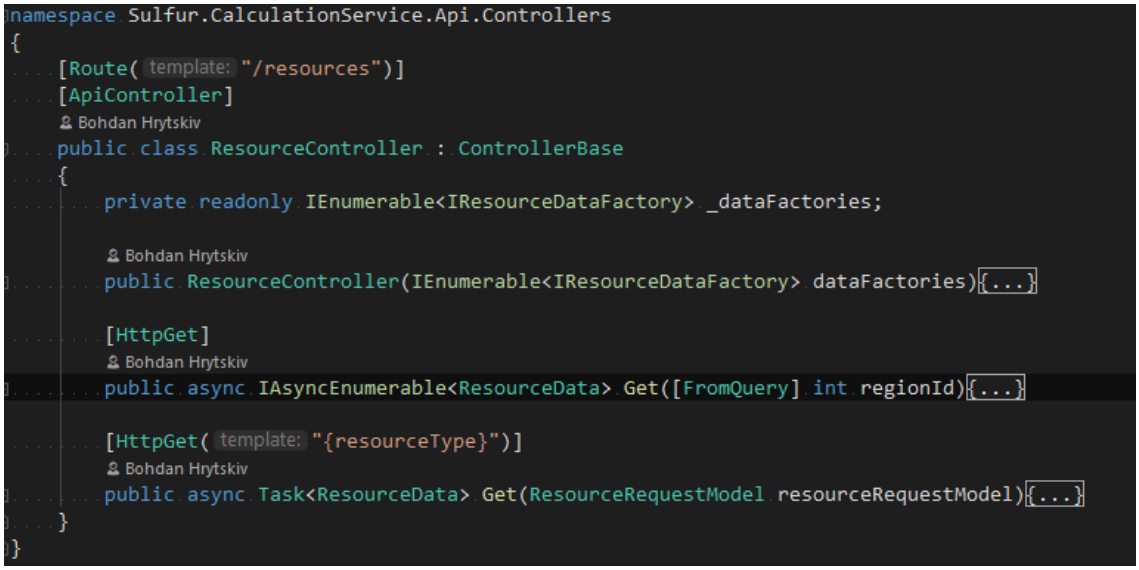

Рисунок 3.4 – Код ResourceController

Далі, для того, щоб контроллер міг приймати ResourceDataFactory, потрібно створити його інтерфейс в проекті Common. Він, як і усі наступні сутності буде містити асинхронні методи, для створення повністю неблокуючого виконання програми. Він буде містити методи Create, який буде використовуватись для приймання вхідних значень, необхідних для обчислень, і повертатись самі результати обчислень, і GetDefaultData, який буде повертати значення, що використовуються для обчислень по замовчуванні. Після його створення, можна додати в ResourceController і DefaultDataController приймання колекції IResourceDataFactory, яка буде передаватись за допомогою ін’єкції залежностей, коли він буде для цього сконфігурований.

Далі потрібно створити сутності GatherService і AnalyzeService, які б в собі зберігав ResourceDataFactory. Але на етапі додавання їх до архітектури, з’являється перша проблема. Через те, що C# є статично-типізованим, все впирається в те, який тип має приймати AnalyzeService і GatherService, а також повертати GatherService. При чому GatherService має завжди повертати тип, який має приймати AnalyzeService. В цьому випадку можна використати узагальнення (generic) і вказати якийсь базовий тип, від якого потрібно далі наслідуватись і узагальнювати реалізації сервісів, але це суттєво впливає на архітектуру таким чином, що кожна сутність повинна зберігати хоча б одне узагальнення про тип, який має приймати або повертати. Також це ставить обмеження на ін’єкцію

залежностей, так як потрібно явно вказати ResourceDataFactory з конкретною моделькою, яку приймає GatherService, AnalyzeService і повертає AnalyzeService, тому тоді не вийде отримати колекцію ResourceDataFactory через ін’єкцію залежностей і використати її в контроллерах. Іншим варіантом є використання динамічного типу даних (dynamic) або ExpandoObject, але даний варіант ще більше погіршує ситуацію, так як він може вплинути на продуктивність і стабільну роботу сервісу, так і внести деякі трудності при розробці, а саме не визначеність типу і наявності даних, які приймаються сервісом, що призводить до частих помилок.

Рішення поставленої проблеми полягає в створенні посереднього абстрактного класу для кожної із сутностей, який би зберігав типи модельок, що використовуються конкретним сервісом, і реалізовував метод кожного й додавав перегружений абстрактний метод, який би і використовувався при прийманні і поверненні даних. Реалізація методу полягала б у перетворенні вищого типу до нижчого або низхідне перетворення (downcast).

На рис. 3.5 і 3.6 представлено вигляд інтерфейсу і абстрактного класу AnalyzeService, які будуть використовуватись для створення сутностей для аналізування відібраних даних. Інтерфейс приймає тип object, найвищий тип в ієрархії типів платформи .NET, а повертає ResourceData, який містить результат аналізу енергоносія і який не буде мінятись. Абстрактний клас, використовуючи паттерн шаблонного метода (template method), реалізовує метод інтерфейсу, але додає свій перевантажений метод, тільки з узагальненим параметром. Метод з узагальненим параметром якраз і буде використовуватись при створенні сутності для аналізування даних. Інтерфейс і абстрактний клас створюються аналогічно, тільки в ньому будуть узагальнюватись не тільки вхідний параметр, але й вихідний. Такий підхід щодо сервісів дозволяє маніпулювати різною кількістю даних, які б приходили у вигляді запита або які потрібно було б обробити.

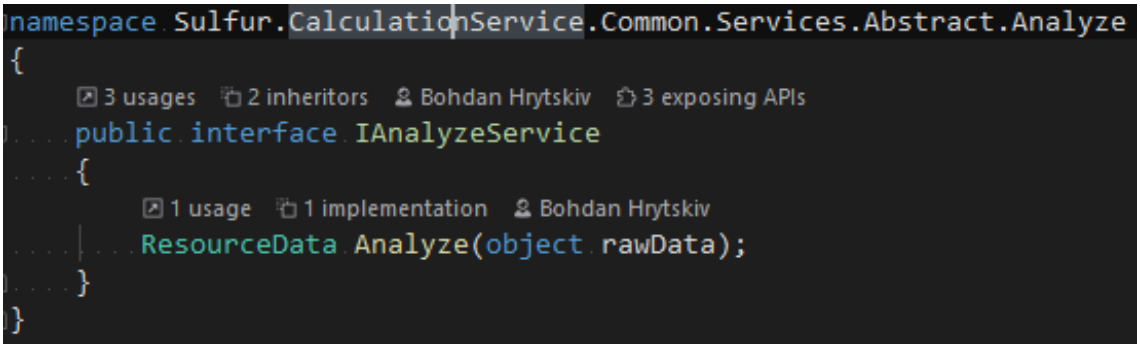

Рисунок 3.5 – Інтерфейс сутності AnalyzeService

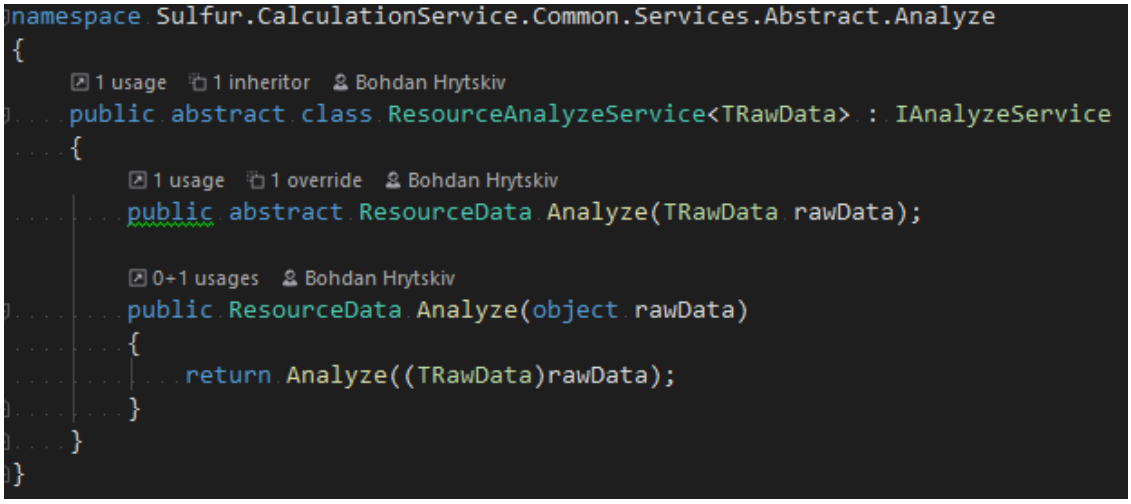

Рисунок 3.6 – Абстрактний клас сутності AnalyzeService

Продовжуючи тему про вхідні параметри, потрібно також поговорити про те, як будуть прийматись і зберігатись усі вхідні параметри. Для такої задачі у ASP.NET Core існує сутність ModelBinder, який дозволяє перетворити певні вхідні параметри у модельку і працювати з набором даних, як з об’єктом. Також для того, щоб кожного разу, в залежності від типу ресурсу, отримувати певний набір даних і певну модельку, потрібно окремо створити інтерфейс IResourceRequestModelBinder, який би вказував, яку модельку потрібно повертати контроллеру і які дані для моделі відбирати з вхідних. Конкретна моделька конкретного ресурсу буде прийматися GatherService’ом для подальшого відбору даних.

Тепер коли дані можеть прийматися через контроллер, по цим даним може проходити відбір і вони можуть бути проаналізовані, час зв’язати їх разом, щоб можна було здійснити повну обробку прийнятих через сервіс даних. Для цього

потрібно створити абстрактний клас ResourceDataFactory, який би реалізував IResourceDataFactory і зберігав у собі GatherService, AnalyzeService, ResourceRequest, який би використовувався при запитах по замовчуванні, та тип ресурсу, до якого належить, що його можна було ідентифікувати з колекції у контроллері.

Після цього, коли архітектура Calculation Service написана, а усі її компоненти налаштовані, потрібно реалізувати для певного ресурсу ResourceRequestModelBinder, ResourceDataFactory, GatherService, AnalyzeService, ResourceRequest, додати моделі для передачі даних між сервісами і зареєструвати ResourceDataFactory до уже наявних. Як можна побачити, при додаванні підтримки обчислень нового ресурсу не потрібно додавати нічого нового у наявну архітектуру, а тільки реалізувати її компоненти, хоча можливість розширення наявного функціоналу також присутня.

На рис. 3.7 показано результат роботи обчислень для визначення енергоефективності природного газу, функціональність для якого була описана у проекті BL в CalculationService.

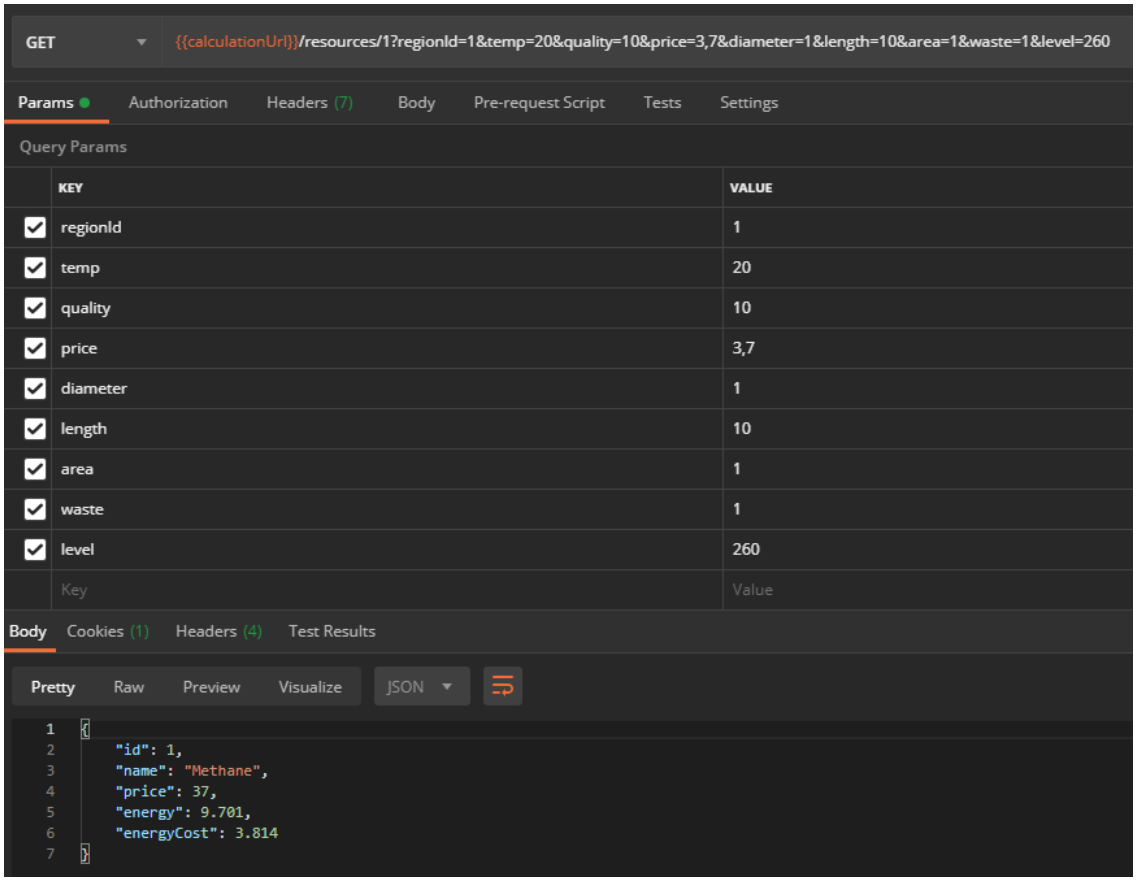

Рисунок 3.7 – Результат роботи обчислень енергоефективності природного газу

3.1.4 Розробка Monitor Service

Створення і налаштування проекту для сервісу Monitor Service за допомогою середовища розробки Rider практично нічим не відрізняється від Calculation Service, за винятком того, що цей сервіс використовує інший тип проекту – Worker Service. Також потрібно створити усі проекти і структуру папок відповідно до того, як це проектувалося для даного сервісу.

Слідуючи усі раніше описаним діаграмам, структура Monitor Service повинна виглядати так, як це зображено на рис. 3.8. Тут можна помітити, що немає проекту, в якому б розміщувалась архітектурна логіка. Це зроблено по тій причині, що сама архітектура складається з кількох сутностей і створення окремого проекту не є великою необхідністю. Натомість, вся архітектурна абстракція винесена у папку *Services/Abstract* в проекті BL.

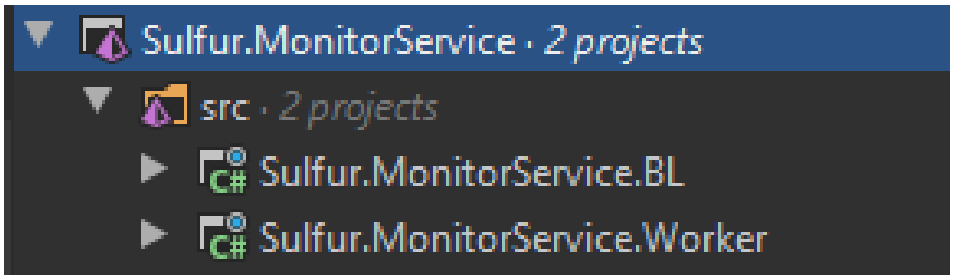

Рисунок 3.8 – Структура рішення Monitor Service

Головною точкою входу є клас Worker, який зареєстрований як постійно приймаючий і працюючий сервіс (HostedService). У ньому відбувається виконання коду, який мав би постійно виконуватись, що чудово підходить для задачі неперервного моніторинг і забору даних із зовнішніх джерел. Так як він тісно працює з операціями вводу-виводу, то якого головний метод Execute є асинхронним. Зважаючи на це, архітектура також повинна підтримувати асинхронність [26].

Для того, що здійснювати вибірку даних із зовнішніх джерел інформації, потрібно створити інтерфейс для сутності DataSourceService, який буде в собі містити асинхронний метод по витягуванні даних, який потрібно реалізувати у

відповідності до бажаного джерела інформації і тип провайдера інформації, по якому можна було б ідентифікувати DataSourceService.

Оскільки велику кількість інформації потрібно буде діставати із джерел, які представляють собою різні веб-сторінки із даними, які, щоб дістати для збереження, потрібно розпарсити, тобто за певним відкинути всю непотрібну інформацію із веб-сторінки і перетворити потрібну у формат, з яким би можна було далі працювати, то було вирішено створити окрему сутність, яка реалізовує паттерн шаблонного метода і обрамлює метод отримання даних з DataSourceService своїм абстрактним методом, який потрібно реалізувати для того, щоб парсити веб-сторінку певного джерела інформації.

На рис. 3.9 показано реалізацію класу Parser, який для реалізовує інтерфейс IDataSourceService і використовує бібліотеку AngleSharp для того, щоб парсити веб-сторінки. Після створення і реалізації сутності, її потрібно зареєструвати для ін’єкції залежностей, так як її має використовувати DataCollector.

```
namespace Sulfur.MonitorService.BL.Services.Abstract
{
    3 usages  3 inheritors  Bohdan Hrytskiv
    public abstract class Parser<T> : IDataSourceService<T>
    {
        0+3 usages  3 overrides  Bohdan Hrytskiv
        public abstract DataProvider DataProvider { get; }

        0+3 usages  Bohdan Hrytskiv
        public async Task<T> GetDataAsync(string path)
        {
            var config = Configuration.Default.WithDefaultLoader();
            var context = BrowsingContext.New(config);
            var addressUrl = new Url(path);
            var document = await context.OpenAsync(addressUrl);
            var result = Parse(document);

            return result;
        }

        1 usage  3 overrides  Bohdan Hrytskiv
        public abstract T Parse(IDocument document);
    }
}
```

Рисунок 3.9 – Абстрактний клас Parser

Для створення сутності DataCollector, необхідно створити асинхронний метод, який би виконував збір і збереження інформації в необхідну секцію. Для прикладу можна взяти ResourcePriceDataCollector, який здійснює відбір по

колекції доступних DataSourceService'ів і при знаходженні відмінностей від тих даних, які вже збережені в системі.

Також слід зауважити, що окрім фіксування і коригування змін, при їхньому виявленні, він також відправляє дані про зміну ціни певного енергоносія. Це показано на рис. 3.10.

```
foreach (var resource in resources)
{
    var resourcePrices = resource.ResourcePrices;

    if (!resourcePrices.Any())
    {
        continue;
    }

    var previousPrice = resourcePrices.Min(x => x.Price);
    var prices = new List<decimal>(resourcePrices.Count);

    foreach (var resourcePrice in resourcePrices)
    {
        var service = _dataSourceServices.Single(x => (int)x.DataProvider == resourcePrice.DataProviderId);

        var price = await service.GetDataAsync(resourcePrice.Link);

        if (price != resourcePrice.Price)
        {
            resourcePrice.Price = price;
        }

        prices.Add(price);
    }

    _unitOfWork.ResourceRepository.Update(resource);

    await SendNotification(resource, previousPrice, currentPrice: prices.Min());
}

await _unitOfWork.SaveChangesAsync(cancellationToken);
```

Рисунок 3.10 – Частина методу оновлення цін енергоресурсів ResourcePriceDataCollector

Усі сутності DataCollector використовуються у методі Run сутності CollectingRunner, яка була створена для того, щоб відділити логіку запуску збору даних від тої, яка могла бути ще наявна у класі Worker. Варто зауважити, що постійні операції входу-виходу суттєво навантажують апаратне забезпечення, на якому вони запускаються, а так як дані не будуть часто змінюватись у часі, то є сенс робити затримки у роботі сервісу на деякий час, щоб зменшити нагрузку на апаратне забезпечення і базу даних, в яку будуть здійснюватись безперервні запити. Дану затримку, краще за все, додати у CollectingRunner для того, щоб частини даних оновлювались поступово, що, також, може зменшити проблеми, викликані з паралельним виконанням запитів у базі даних.

3.1.5 Розробка Notification Service

Для розробки Notification Service використовується використовується шаблон проекту Web API, як це було показано у Calculation Service, для того, щоб він мав змогу приймати запити, серед яких є запити на отримання публічного ключа, підписки для отримання сповіщень і оповіщення підписаних користувачів з отриманими даними. Сама структура проекту подібна до структури Monitor Service, так як не містить великої кількості архітектурних сутностей.

Головною задачею Notification Service є відправка сповіщень усім користувачам, які на неї підписались раніше. Для цього застосовується технологія відправки повідомлень, яка називається Web Push. Уся передача даних здійснюється через повідомлення, які є зашифрованими і передаються потоком байтів. Для того, щоб забезпечити безпеку даних і бути впевненим, що сповіщення на клієнтську частину були відправлені від надійного серверу, використовується аутентифікація заснована на ідентифікації добровільного сервера (Voluntary Application Server або VAPID).

VAPID аутентифікація здійснюється на основі передачі і зберігання клієнтами публічного ключа, який використовується для розшифрування отриманого повідомлення з серверу. Приватний ключ, який зберігається на сервері, використовується для шифрування повідомлення, яке буде відправлятися на клієнтську частину. Такий підхід забезпечує те, що джерелом сповіщень буде надійне джерело [27].

Для застосування технологій Web Push і VAPID, в сервісі Notification Service використовується бібліотека Lib.Net.Http.WebPush. Вона дозволяє легко працювати з відправкою повідомлень для підписаних користувачів і приховує деталі реалізації шифрування і відправки повідомлень. Все що потрібно – це використати наявну підписку (Subscription), яка має зберігати кінцеву точку (endpoint), на яку потрібно відправляти повідомлення, публічний і приватний ключі.

Для того щоб почати керувати підписками і сповіщеннями, потрібно створити три контроллери: PublicKeyController, SubscriptionController і NotifyController. PublicKeyController буде відповідати за відправку на клієнтську сторону публічного ключа, необхідного для розшифрування отриманих повідомлень. SubscriptionController буде здійснювати збереження або видалення усіх наявних підписок. NotifyController буде приймати запити від Monitor Service про зміну цін енергоносіїв і відправляти про це сповіщення.

Для того, щоб зменшити кількість коду в контроллерах і зробити їх більш легкими, потрібно також створити сутності NotifyService і SubscriptionService. Вони, якраз, і будуть зберігати в собі всю бізнес-логіку взаємодії з підписками.

NotifyService буде працювати зі спеціальною сутністю бібліотеки Lib.Net.Http.WebPush – PushServiceClient. Цей клас є розширенням звичайного HttpClient, який є стандартним для відправки запитів по протоколу HTTP платформи .NET, але, в свою чергу, він також містить логіку шифрування повідомлення, яке необхідно відправити на певну кінцеву точку з авторизаційними ключами. Відправка повідомлень по HTTP є досить довгою і важкою операцією, тому метод відправки сповіщень PushServiceClient і NotifyService є асинхронними.

SubscriptionService буде працювати із базою даних проекту, а конкретніше сутністю UnitOfWork, яку потрібно пізніше реалізувати. Головним моментом являється де, що окрім операцій створення-видалення нових підписок, він ще містить метод, який повертає колекцію усіх підписаних клієнтів. Так як дана колекція не буде змінюватись допоки нова підписка не буде створена або існуюча видалена, то немає необхідності постійно, при кожному запиті на відправку сповіщень, витягувати з бази даних усі підписки. Для такого випадку логічно застосувати кешування, яке можна здійснити використавши стандартний клас платформи .NET MemoryCache, який зберігає результат виконання певного запиту по деякому ключу, у випадку витягування усіх сутностей з бази даних, такий ключ буде один. Після кожного створення нової підписки або видалення, збережений кеш у пам’яті буде очищений, а при наступному запиті на витягуванні даних – знову заповнений.

3.1.6 Розробка DataAccess

Для створення бібліотеки DataAccess, потрібно створити окремий репозиторій, в якому буде розміщено рішення і проект DataAccess. Так як окремим компонентом або загальною бібліотекою є не рішення, а проект, то рішення міститиме тільки один проект, в якому повинна знаходитись уся необхідна функціональність. За потреби, може бути створений окремий проект як окрема бібліотека, який би містив архітектурну логіку проекту, який би використовувався на практиці.

Як говорилось раніше, щоб створити проект типу Class library, треба вибрати версію, яка б не конфліктувала з тими, які її будуть використовувати. Таке правило було допустимим, якби мова йшла про створення проекту всередині одного рішення, але бібліотека DataAccess може використовуватись окрім проектів з різними версіями .NET Core, так ще й із проектами написаними з допомогою .NET Framework, Mono, які є не сумісними з .NET Core. Для такого було створено .NET Standart.

.NET Standart – це специфікація, яка описує, які функціональні можливості повинна мати технологія платформи .NET, яка її реалізовує. Він був створений з метою узагальнення і сумісності різних версій платформ. Так як містить тільки мінімально необхідний функціонал, то збірка, написана з його використанням, може бути застосована не тільки для різних версій одного фреймворку, але й між окремими, що прекрасно підходить для створення бібліотек спільного доступу.[28]

Тому бібліотека DataAccess буде створена за допомогою .NET Standart 2.0. До цього також необхідно додати деяку складність і невизначеність при розробці бібліотеки. Так як проект типу Class Library не може бути запущений, тому повністю до кінця перевірити правильність роботи написаного функціоналу буде досить проблематично. Щоб вирішити дану проблему у папці, призначеній для тестування, можна створити проект будь-якого типу, який би можна було запустити і, так як вони будуть знаходитись в одному рішенні, додати посилання

на локальну збірку бібліотеки DataAccess. Тоді набагато легше буде провести тестування з точки зору розробника (dev-testing) написаного функціоналу.

Для роботи з базою даних буде використовуватись технологія Entity Framework Core. Entity Framework Core або EF Core – це ORM-технологія, яка використовується для отримування і маніпулювання даними з бази даних, як з сутностями ООП. EF Core дозволяє легко взаємодіяти з базою даних, писати код, який слідує принципам ООП, і писати набагато гнучкіший код, який в будь-який момент можна модифікувати [29, 30].

Згідно з архітектурою проекту DataAccess, він буде містити сутності Repository, які реалізовані у вигляді інтерфейсу з набором усіх CRUD операцій (create, read, update, delete) і узагальнений репозиторій, який буде містити реалізацію методів інтерфейсу для більш загальних випадків. Узагальнений репозиторій буде містити, в даному випадку, об’єкт EF Core DbContext, який буде отримувати з UnitOfWork. UnitOfWork, в свою чергу, буде містити тільки метод для збереження змін, здійснених через Repository, і властивості (property), в яких будуть знаходитись конкретні реалізації Repository.

Мета реалізації паттернів Repository і UnitOfWork – це абстракція і відв’язання від конкретної технології роботи з даними. Щоб не переписувати весь проект, при переході на іншу базу даних або технологію, легше переписати методи сутностей DataAccess, які він використовує. Ще слід додати, що сам DbContext, окрім наслідування конкретним контекстом, потрібно додати ще й інтерфейс контексту бази даних для того, щоб полегшити юніт-тестування кожного із рівнів абстракції.

Для додавання нової таблиці в базу даних, потрібно створити її модельку і додати її до реально контексту бази і його інтерфейсу. Кожна із таблиць має бути типізована класом DbSet, але, так як його неможливо заміти при тестуванні, у інтерфейсі потрібно використовувати IQueryable. DbSet якраз є розширення IQueryable, тому це не має викликати проблем. Також потрібно враховувати те, що база даних, яка використовується, накладає певні обмеження на те, як мають виглядати самі моделі EF Core і які типи повинні мати їхні властивості.

Так як для проекту використовується база даних MS SQL Server, то це наводить на одну проблему. MS SQL Server – це реляційна база даних, тому її дані зберігаються в окремих таблицях, які можуть зберігати примітивні типи даних. Але для того, щоб зберігати ключі, які використовуються в парі з ендпоїнтом підписки, потрібно використовувати словник (Dictionary) рядків, ключі якого також є рядками. При привносить деякі трудності, так як реляційні бази, на відміну від документо-орієнтованих баз даних, не можуть зберігати дані такого типу. Перший, який спадає на думку, підхід – це перетворення словника у формат JSON і зберігання його як тексту, а при читанні, перетворення назад у словник. Дане рішення зберігає нормалізацію бази даних і не потребує створення додаткових стовбців, але в ньому є деякі недостатки. Так як довжина і кількість самих ключів є невідомою, а їхнє значення може мати довжину більше 200 символів, то довжину строки в базі даних потрібно обрати максимальну, що є нераціональним використанням пам’яті і швидкодія такої бази може значно просісти.

Тому необхідно застосувати інший підхід. Таким підходом є створення таблиці, яка б імітувала функціональність словника, і при перетворенні даних з бази даних у модельку зберігалась як словник, а не колекція модельок іншої сутності. Для того, щоб це зробити, потрібно створити модельку зі значеннями ключа, значення ключа і зовнішнього ключа (foreign key) модельки, яка його буде використовувати. Для того, щоб не було повторень серед ключів, ключ словника і зовнішній ключ потрібно зробити композитними ключами таблиці. Тоді досягнеться імітація можливостей словника.

На рис 3.11 показано вигляд такої модельки, де властивості Key і SubscriptionId – є композитним ключем, тому в межах однієї підписки може використовуватись тільки один ключ, а Value – це значення, що зберігається по ключу.

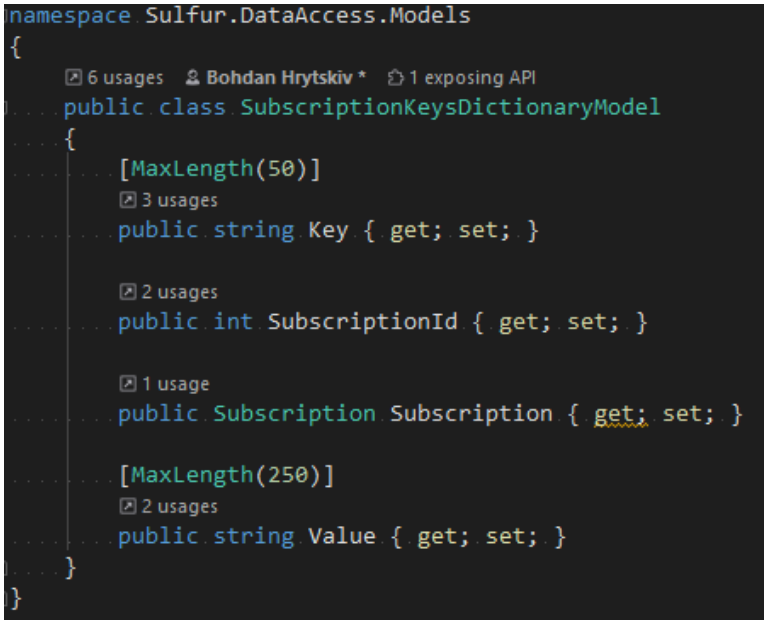

Рисунок 3.11 – Вигляд моделі, яка імітує словник

Коли усі моделі, згідно структури бази даних, створяться, то далі потрібно запустити міграцію бази даних і оновити базу даних створеною міграцією. Потрібно це робити у проектах, в яких є точки запуску. Але, також, так як DbContext знаходиться в збірці DataAccess, але використовуєтсья у іншій, потрібно також явно вказати проект, для якого створюється міграція у методі конфігурування DbContext'у.

3.2 Написання клієнтської частини програмного забезпечення визначення ефективності енергоносіїв

Для написання клієнтської частини даного проекту використовується фреймворк Angular. Angular – це веб-фреймворк від Google, для побудови і написання інтерактивних веб-сайтів, які працюють за допомогою JavaScript. Angular не є новим фреймворком, а продовженням Angular.js, який був написаний на мові програмування JavaScript. Angular, у свою чергу, з версії 2, був переписаний під TypeScript, статично-типізовану мову програмування, яка компілюється у JavaScript, який вже виконується у браузері. Серед плюсів даного фреймворку можна назвати структурованість, чітко вибудовану архітектуру,

статичну типізацію TypeScript і велику кількість можливостей, доступних “з коробки” [31, 32].

Також хотілося б згадати про ще одну технологію, яка тісно пов’язана з Angular, а також підтримується компанією Google – прогресивні веб-застосунки (Progressive Web Application). Progressive Web Application або PWA – це загальна назва підходу створення веб-застосунків придумана Google, яка використовує кілька технологій, що дозволяють створювати гібридний веб-застосунок, який функціонує як повноцінний веб-сайт і мобільний застосунок. В його основі лежить кешування і Service Worker, який є бар’єром між браузером і веб-сервером. Це дозволяє, у випадку, коли веб-сервер недоступний або веб-сайт відкривається без доступу до Інтернету, використовувати сторінку і дані, які були попередньо закешовані. Через те, що PWA є, по суті, застосунком, написаним на JavaScript, його можна зберегти будь-де, де є браузер, який підтримує прогресивні веб-застосунки [33].

Веб-застосунок використовує модульну архітектуру, кожен модуль якого означає завершену функціональність. Модуль ділиться на компоненти, з яких складається сторінка і які являють собою візуальну частину представлення. Усі модулі будуть зберігатися у папці feature, яка буде зберігати папку того чи іншого модуля. Для створення веб-застосунків використовується стиль написання camel case для коду на TypeScript і kebab case для HTML і CSS.

Застосунок буде мати головну сторінку, на які буде знаходитись уся інформація по ефективності енергоносіїв, та сторінку, на якій можна було б підписатись на сповіщення про зміну цін енергоресурсів. Для переходу між ними потрібно додати маршрутизацію, яка буде задіюватись на рівні всього застосунку. Усі посилання, по яким можна буде переходити між сторінками, будуть знаходитись на висувному меню, яке, разом із шапкою сайту, являється каркасом веб-застосунку.

Так як усі запити будуть здійснюватись на кілька сервісів, потрібно додати сутність архітектури Angular, яка виконується перед кожним запитом до веб-серверу – Interceptor. Він дозволяє переписати шлях, по якому мав би здійснюватись запит на коректний. Тому потім, при написанні сервісів, які б

діставали дані з веб-серверу, не потрібно буде вказувати повний шлях маршруту, по якому знаходиться веб-сервер.

Також для створення компонентів сайту використовується стандартна бібліотека Angular Material. В ній знаходяться всі елементи управління і засоби для створення застосунків використовуючи Material Design. Для того, щоб зменшити кількість однакового коду і збільшити можливість перевикористання, компоненти, з яких буде складатися веб-сайт, будуть винесені у загальну папку shared, звідки можуть бути отримані при підключенні відповідного модуля.

Для підтримки сповіщень, які будуть отримуватись з Notification Service, можна використати одну із можливостей PWA – SwPush. Це сервіс, який керує оброблення отриманих повідомлень у вигляді сповіщень. Щоб його використовувати, потрібно впевнитись, що режим PWA працює.

На рис. 3.12 і 3.13 можна побачити те, як виглядає веб-сайт із браузера і у встановленому виді. Можна помітити, що суттєвих відмінностей між ними немає, якщо не враховувати те, встановлений PWA запускається у окремому вікні і функціонує як окремий застосунок на пристрої, а тому з ним працювати набагато зручніше, аніж як з окремою вкладкою в браузері. На рис. 3.14 показано результат роботи сповіщень. Після того, як ціну одного із енергоносіїв було змінено, з’являється сповіщення з детальною інформацією про зміну ціни конкретного ресурсу.

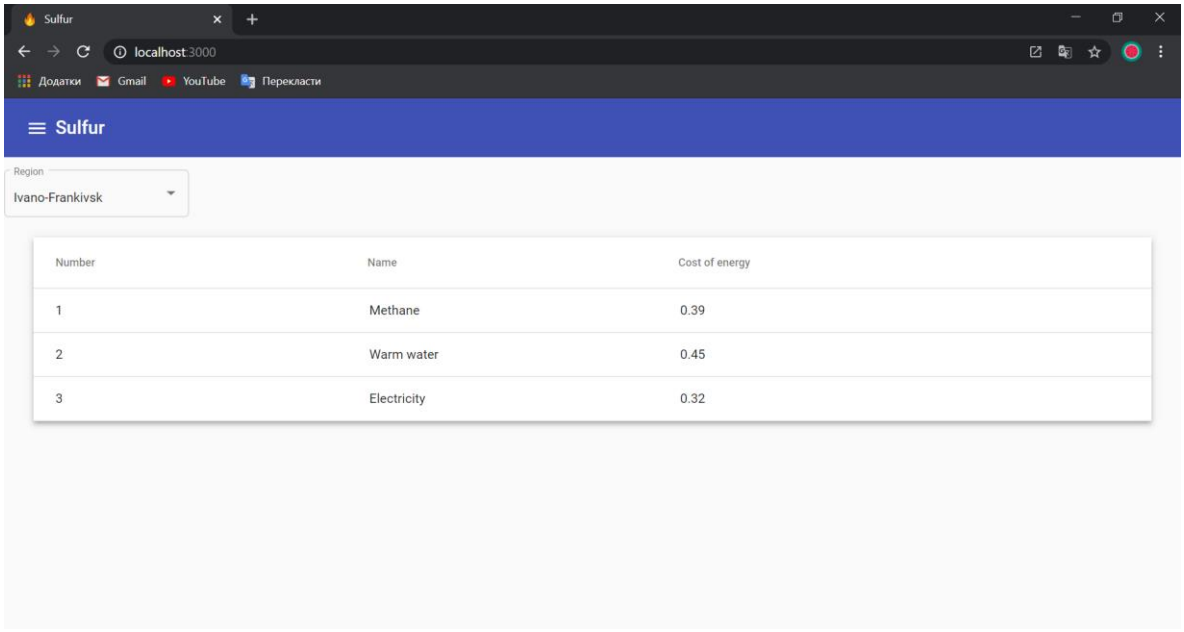

Рисунок 3.12 – Вигляд веб-сайту проекту з браузера

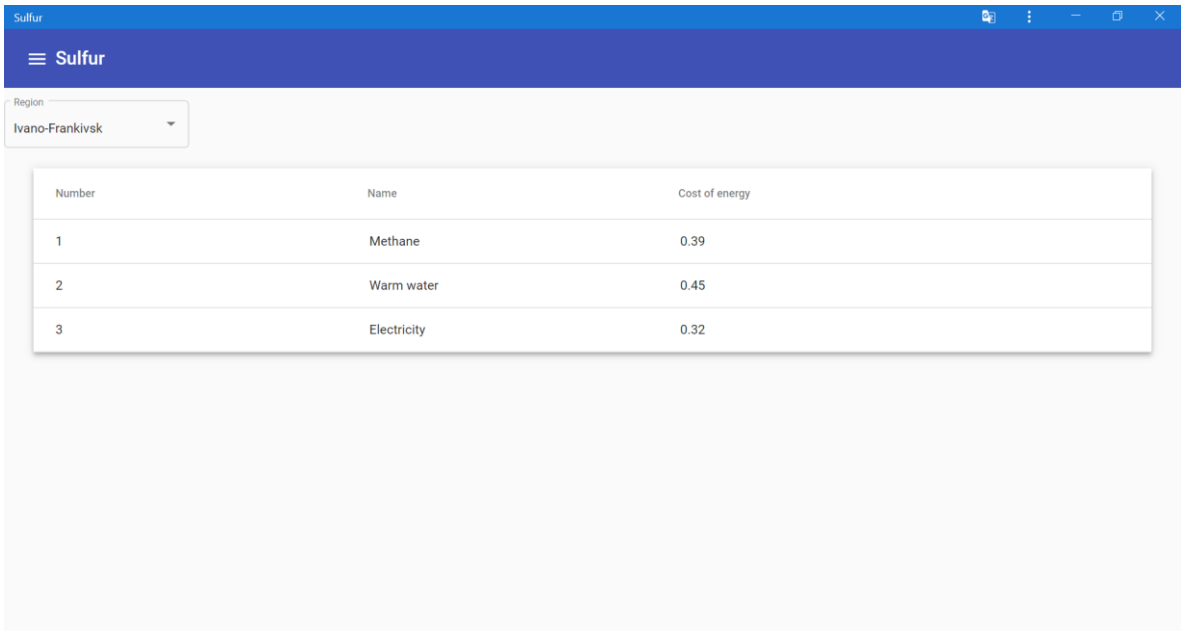

Рисунок 3.13 – Вигляд встановленого веб-сайту проекту

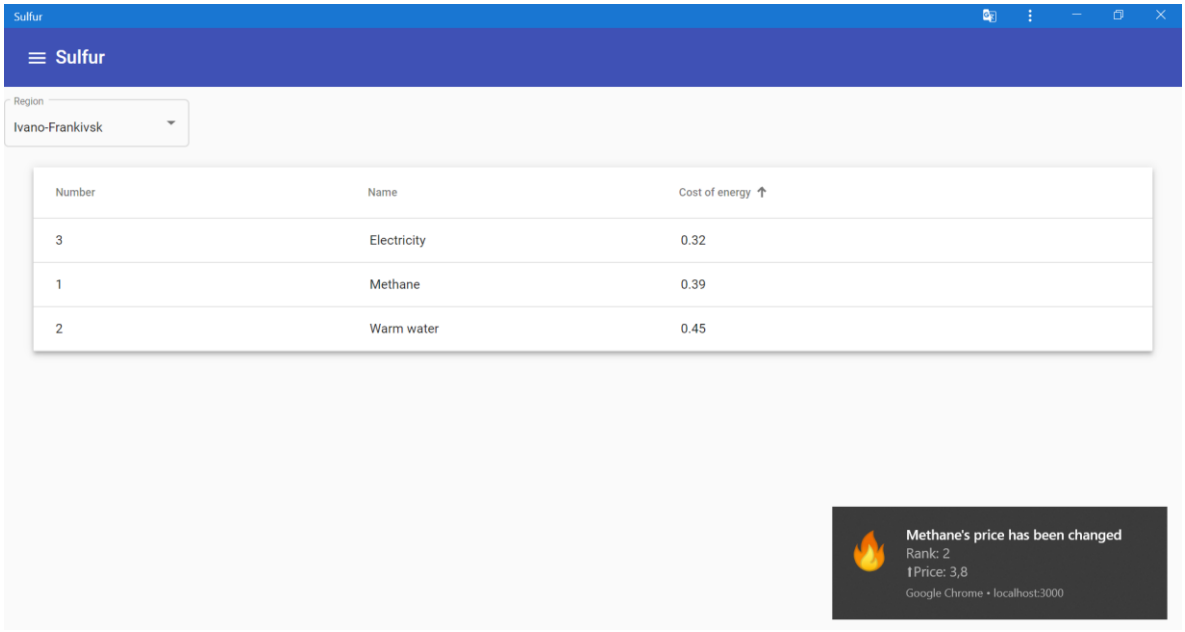

Рисунок 3.14 – Вигляд сповіщення про зміну ціни

3.3 Розгортання програмного забезпечення визначення ефективності енергоносіїв

Для розгортання проекту побудованого на мікросервісах найчастіше використовують контейнеризацію, яка забезпечується за допомогою інструменту під назвою Docker.

Docker – це інструмент для створення і управління ізольованими Linux-контейнерами. Контейнери, створені за допомогою Docker, не лишають ніякого сліду у системі, на якій запускаються, і можуть бути легко зупиненими і видаленими. Контейнеризація зменшує залежність від системи коду, який виконується, до мінімуму, залишаючи тільки артефакт, для запуску якого потрібен сам движок Docker. Стан кожного запущеного контейнеру можна зберегти і в майбутньому запустити, зробивши його знімок (image). Для того, щоб автоматизувати створення знімка з іншого знімка або самої Linux-системи, потрібно створити спеціальну інструкцію, яка вказує, які кроки потрібно відтворити, щоб потім запустити застосунок – Dockerfile [34].

Так як кожен сервіс – це окрема автономна підпрограма, яка визначається її рішенням (solution) у .NET застосунку, то, слідуючи структурі проекту, Dockerfile повинен розміщуватись в корені сирцевого коду програми. Dockerfile має в собі мати інструкції по будуванні проекту і його запущенні, тому для побудови потрібно використовувати знімок SDK для .NET Core 3.0 і середовища для запуску ASP.NET Core застосунку. В усіх сервісах Dockerfile буде схожий, так як для їхньої побудови потрібно виконувати ті самі кроки. Для створення знімку клієнтської частини потрібно використати Angular CLI, який перекомпілює весь застосунок у JavaScript, після чого, створений каталог зі скриптами, потрібно перенести на веб-сервер, наприклад Nginx, який буде приймати запити на отримання JavaScript коду.

Але самих знімків замало для того, щоб запустити проект, і щоб сервіси почали між собою взаємодіяти. На допомогу приходить інструмент від Google під назвою Kubernetes. Kubernetes або k8s – це оркестратор контейнерів, який здійснює управління ними в автоматичному режимі. Він контролює працездатність кластеру і її навантаження, рівномірно розприділяючи навантаження на кожен із сервісів і, при потребі, створенні копій сервісів для їх надійності. При виведенні одного із сервісів з ладу, Kubernetes робить спробу його перестворення з вказаного знімка. У термінології Kubernetes є багато власних понять, таких як поди, деплойменти, сервіси, секрети та інші. Усі вони називаються видами ресурсу Kubernetes. З Kubernetes можна працювати за

допомогою консолі або власного графічного інтерфейсу. Рекомендованим є створення файлів конфігурації кластера, так як у випадку, коли програмне забезпечення буде хоститись в іншому місці, всі попередні налаштування будуть задіяні за допомогою файлів конфігурації YAML формату [35].

Оскільки Kubernetes використовує знімки ззовні, потрібно створити Docker Registry, де будуть зберігатись створені знімки кожного сервісу. Для цього буде створена приватна Docker Registry на Azure Portal, хоча знімки можуть зберігатись де завгодно, наприклад на Docker Hub. Кожен знімок, який відноситься до певної Docker Registry, повинен містити в назві на початку назву конкретного Registry, для того, щоб його можна було ідентифікувати.

Після створення усіх знімків і їхнього злиття на обрану Docker Registry, потрібно створити тип ресурсу Kubernetes під назвою деплоймент (deployment). Його призначення – створення подів, які є, по суті, місцем, де запускаються контейнери, і їхнім управлінням. Для створення деплойменту потрібно вказати назву знімку, для якого створюється даний деплоймент. Також, так як Azure Docker Registry є приватним, необхідно створити секрет (secret) – тип ресурсу, який містить певну чутливу інформацію, потрібну для інших типів ресурсів для того чи іншого випадку. Збереження файлів секретів є не рекомендованим, так як вони всередині містять чутливу інформацію, яка може бути легко добутою з самого файлу.

Для того, щоб поди, створені деплойментами, могли функціонувати, потрібно створити ще один тип ресурсу – сервіс (service). Сервіси відповідають за видимість подів всередині кластеру або за його межами. Щоб встановити видимість у межах кластеру потрібно вибрати тип NodePort, якщо за його межами – LoadBalancer.

Коли усі сервіси налаштовані і готові до використання, вони будуть не готові для того, щоб їх використовували ззовні, тому що кластер немає точок входу для цього. Можна створити сервіси з типом LoadBalancer, але це в майбутньому може призвести до проблем із виконанням і роботою кластеру. Тому для такого використовується інгрес (Ingress). Ingress – це набір правил, за якими має здійснюватись маршрутизація по кластеру, для інгрес-контроллера,

яким може виступати веб-сервер, в залежності від обраного провайдера. Інгрес якраз і відповідає за балансування нагрузки між сервісами розміщеними в кластері [36]. Він також може вказувати як обробляти певний запит, наприклад один із маршрутів можна перенаправляти на певний сервіс, а інший заборонити. Серед сервісів у архітектурі проекту є маршрут, який потрібно обмежити ззовні – це *POST /notify* від Notification Service. Це необхідно для того, щоб ніхто, окрім сервісів всередині кластеру, не зміг би відправити на нього повідомлення, яке б було далі передано усіх підписникам.

Для того, щоб встановити інгрес-контроллер потрібно використати Helm – пакетний менеджер в екосистемі Kubernetes, який дозволяє автоматизувати процеси, пов’язані із конфігуруванням кластеру [37]. Щоб встановити контроллер, в нашому випадку це Nginx-контроллер, потрібно створити репозиторій, в якому він знаходиться за допомогою команди:

helm repo add stable https://kubernetes-charts.storage.googleapis.com/

Далі необхідно створити інгрес-контроллер з доданого репозиторію. Це можна зробити за допомогою команди:

helm install nginx-ingress stable/nginx-ingress

Також потрібно врахувати те, що кожного разу при створенні інгрес-контроллера, для нього буде виділятись нова IP-адреса, тому, якщо збереження однієї IP-адреси є важливим, то потрібно використовувати статичну адресу.

Після того, коли всі файли конфігурації були створені і застосовані для існуючого кластеру, а інгрес-контроллер був встановлений, то знімки з Azure Docker Registry мають підтягнутись і функціонувати в повній мірі. Тому при переході на корінь (root) відповідного адресу, має відкритись клієнтська частина програмного забезпечення, а всі запити на сервіси, які вона буде здійснювати, повинні працювати належним чином.

4 БІЗНЕС-ПЛАН ДЛЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ВИЗНАЧЕННЯ ЕФЕКТИВНОСТІ ЕНЕРГОНОСІЇВ

4.1 Резюме

Програмне забезпечення визначення ефективності енергоносіїв буде займатися наданням послуг у сфері надання інформації по енергоефективності ресурсів, які представлені на ринку. Воно вирішує проблему пошуку більш вигідного й доступного енергоносія для різних потреб населення, так як має всі можливості для автоматичної обробки і актуалізації даних. Також через відсутність прямих аналогів у межах поширення є унікальною за своїм призначенням.

Створене ПЗ можна використовувати для знаходження найдоступнішого енергоносія для власних потреб, замість того, щоб проводити аналіз і обчислення вручну. Слід зазначити, що він може бути використаний тільки на українському ринку з даними від публічних і офіційних джерел інформації.

Програмний продукт буде поширюватись вільно, так як буде розміщений в мережі Інтернет у вільному доступі, при наявності з'єднання до Інтернету.

Цільова аудиторія – люди від 23 до 60 років, які вміють користуватися будь-яким девайсом з доступом до мережі Інтернет і яким необхідні дані по наявних енергоносіях в цілях утеплення, ведення домогосподарства, налагодження різних технічних процесі, які вимагають енергії, та інших. Через великий попит на такого роду послуги серед населення, веб-сайт може мати досить велику популярність.

Серед виробничих потужностей слід зазначити, що для розробки і функціонування веб-сайту приміщення не вимагається, так як реалізація є одноразовою. Передбачається кооперація з компаніями, які б надавали послуги по розробці дизайну інтерфейсу веб-сайту. Накладними витратами є витрати на послуги хостинг-провайдера, інтернет, оплату кооперації та електроенергію.

Фінансування буде здійснюватись з вложень інвесторів, державних грантів і підтримки державних органів. Серед грантів, які могли б підтримати проект,

можуть бути гранти з веб-ресурсу «Громадський простір» і «Гурт» [38, 39]. Так як гранти на підтримку енергоефективності та екології з'являються часто і не залежать від політичної і економічної ситуації у державі, то великий шанс, що проект може бути одобреним для фінансування.

4.2 Трудомісткість розробки програмного забезпечення

Трудомісткість розробки програмного забезпечення найзручніше зробити у людино-годинах. Цей спосіб простий для приблизно представлення загальної кількості роботи, але немає високої точності, так як не враховує великої кількості факторів.

Трудомісткість розробки програмного забезпечення можна визначити за формулою [40]:

$$t = t_o + t_u + t_a + t_n + t_{відл} + t_д, \quad (4.1)$$

де t_o – час на підготовку і опис задачі, t_u – час на дослідження алгоритму рішення задачі, t_a – час на розробку блок-схеми алгоритму, t_n – час на програмування готової блок-схеми, $t_{відл}$ – час на відлагодження написаної програми, $t_д$ – час на підготовку документації.

Для визначення витрати асу на розробку необхідно також визначити умовне число операторів.

Умовне число операторів [40]:

$$Q = q * C * (1 + p), \quad (4.2)$$

де q – передбачуване число операторів, C – коефіцієнт складності програми, p – коефіцієнт кореляції програми в ході її розробки.

Визначення t_u відбувається за формулою [40]:

$$t_u = \frac{Q * B}{(75..85) * k}, \quad (4.3)$$

де B – коефіцієнт збільшення часу на розробку внаслідок недостатнього опису, k – коефіцієнт кваліфікації програміста.

Визначення t_a відбувається за формулою [40]:

$$t_a = \frac{Q}{(20..25) * k}, \quad (4.4)$$

Визначення t_n відбувається за формулою [40]:

$$t_n = \frac{Q}{(20..25) * k}, \quad (4.5)$$

Визначення $t_{відл}$ відбувається за формулою [40]:

$$t_{відл} = \frac{Q}{(4..5) * k}, \quad (4.6)$$

Визначення $t_д$ відбувається за формулою [40]:

$$t_д = 1,75 * \frac{Q}{(15..20) * k}, \quad (4.7)$$

Обчислення зі знаходженням трудомісткості розробки даного програмного забезпечення:

$$Q = 900 * 1.3 * (1 + 0.8) = 2106;$$

$$t_u = (2106 * 1.3) / (80 * 0.8) = 42.78;$$

$$t_a = 2106 / 23 * 1.3 = 119.03;$$

$$t_n = 2106 / 22 * 1.3 = 124.45;$$

$$t_{відл} = 2106 / 5 * 0.8 = 526.5$$

$$t_д = 1.75 * (2106 / 17 * 0.8) = 173.44;$$

$$t = 42.78 + 119.03 + 124.45 + 526.5 + 173.44 = 986.2;$$

4.3 Витрати на розробку програмного забезпечення

Перш ніж обчислити загальні витрати на розробку програмного забезпечення, необхідно визначитись із оплатою роботи програміста. Це можна зробити, якщо відоме місячне грошове забезпечення програміста. Середнє грошове забезпечення програміста рівня Junior, по даним на 2020 рік, складає 10000,00 гривень. Оплата буде здійснюватись не по мірі виконаної роботи, а щомісяця, так як така схема оплати більш поширена у компаніях, які надають послуги з розробки програмного забезпечення.

Щоб визначити кількість місяців, необхідних для розробки, необхідно перед цим визначити кількість робочих годин у році. Це можна зробити за такою формулою [41]:

$$n(p) = (N - N(n) - N(в)) * 8, \quad (4.8)$$

де N – загальне число днів у році, $N(n)$ – число святкових і неробочих днів у році, $N(в)$ – число вихідних днів у році.

У році є 10 святкових днів і 104 – вихідних. Тому, кількість робочих годин у 2020 році дорівнює:

$$n(p) = (366 -10 - 104) * 8 = 2016$$

З цього випливає, що середня кількість робочих годин за місяць дорівнює[39]:

$$n_{міс}(p) = 2016 / 12 = 168$$

Тепер, щоб визначити кількість місяців, необхідних для розробки, треба поділити кількість людино-годин, необхідних для розробки програмного забезпечення і отримане число округлити до верхньої межі (отримати стелю числа). Виходячи з цього, можна обчислити кількість місяців для розробки:

$$n_{міс} = 986.2 / 168 = 5.87 \approx 6$$

Це означає, що для розробки програмного забезпечення одній людині потрібно потратити 6 місяців, що в грошовому еквіваленті дорівнює:

$$C_{розр} = 10000 * 6 = 60000, грн$$

Також, для того, щоб зменшити час розробки, можна найняти не 1 розробника, а 2 або 3, що дозволить краще розпаралелити роботу між розробниками і, також, мала кількість працівників не повинна вплинути на продуктивність розробки, а, можливо, і навпаки її збільшити при колаборації. Тому, при наймі 2 або 3 працівників для розробки програмного забезпечення, загальний час, який потрібно потратити на розробку, буде приблизно дорівнювати відповідно 3 або 2 місяці.

Номінальні щомісячні витрати витрати, до і після розробки програмного забезпечення, будуть приблизно становити:

Оренда бази даних	500 грн.
Оренда статичного IP-адресу	70 грн.
Оренда Docker Registry	200 грн.
Оренда хостингу	1500 грн.
Разом:	**2270 грн.**

Але потрібно врахувати, що вказані ціни були взяті з урахуванням ціни курсу долара, так як обраний хостинг (Azure) публікує ціни у даній валюті. Тому наведені ціни є такими згідно з 2020 роком. Вони можуть помінятись як в більшу, так і в меншу сторону. В основному вони залежать від провайдера даних послуг,

регіону, під яким хоститься програмне забезпечення, курсу долара і економічної ситуації в Україні та світі.

ВИСНОВКИ

В ході даної роботи було проведено розгляд стану енергоефективності у світі та Україні, проведено аналіз та порівняння вже існуючих рішень, які можуть бути застосованими у сфері енергоефективності та їх основні недоліки.

Було спроектовано програмне забезпечення по визначенню ефективності енергоносіїв, загальний вигляд, функціональні можливості та внутрішню архітектуру компонентів.

Також проаналізовано використані технології для розробки даного програмного продукту, їхню доцільність, деякі важливі моменти при застосуванні та принцип створення спроектнованого програмного забезпечення з їхнім використанням.

Розроблене програмне забезпечення має більше можливостей по знаходженню найбільш доступного та ефективного енергоносія, так як він порівнює енергію різних видів ресурсів, яка обчислюється з врахування особливостей та нюансів кожного із них.

Створена програма є простою у використанні, орієнтована на український ринок, дозволяє працювати з нею з будь-якого пристрою з доступом до мережі Інтернет і, за наявності відповідного браузера, бути встановленою на будь-які операційній системі й працювати як окремий застосунок.

Програмне забезпечення може використовуватись будь-ким, хто б хотів знайти найекономніший енергоносій для власних потреб.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

REFERENCES

1. Дайте людям можливість заощаджувати: стаття. URL: https://www.epravda.com.ua/columns/2017/11/16/631267/ (дата звернення: 26.01.2020).
2. Уряд планує витратити 2 млрд. гривень на енергоефективність у 2020 році: стаття. URL: https://hromadske.ua/posts/uryad-planuye-vitratiti-2-mlrd-grn-na-energoefektivnist-u-2020-roci (дата звернення: 26.01.2020).
3. Поняття енергоефективності. URL: https://uk.wikipedia.org/wiki/Енергоефективність (дата звернення: 26.01.2020).
4. Поняття енергозбереження. URL: https://uk.wikipedia.org/wiki/Енергозбереження (дата звернення: 26.01.2020).
5. M. Kozlenko and M. Kuz, "Joint capturing of readouts of household power supply meters," 2016 13th International Conference on Modern Problems of Radio Engineering, Telecommunications and Computer Science (TCSET), Lviv, 2016, pp. 755-757, doi: 10.1109/TCSET.2016.7452172.
6. Оцінка ринку постачальників послуг з енергоефективності: брошура. URL: https://saee.gov.ua/sites/default/files/EE_broshure_out_2018.pdf (дата звернення: 26.01.2020).
7. Check24: веб-сайт. URL: https://www.check24.net/gasanbieter-wechseln/ (дата звернення: 26.01.2020)
8. Verivox: веб-сайт. URL: https://www.verivox.de/landingpage-affiliates/?source_id=153&utm_medium=affiliate&utm_source=affilinet&utm_campaign=gas&utm_content=text&ref=478440&affmt=2&affmn=2 (дата звернення: 26.01.2020)
9. Wechselpilot: веб-сайт. URL: https://www.wechselpilot.com/ (дата звернення: 26.01.2020)

10. Minfin: веб-сайт. URL: https://index.minfin.com.ua/ua/tariff/electric/ (дата звернення: 26.01.2020)
11. Поняття функціональних вимог. URL: https://uk.wikipedia.org/wiki/Вимоги_до_програмного_забезпечення (дата звернення: 10.05.2020).
12. Поняття UML. URL: https://uk.wikipedia.org/wiki/Unified_Modeling_Language (дата звернення: 10.05.2020).
13. Поняття архітектури програмного забезпечення. URL: https://uk.wikipedia.org/wiki/Архітектура_програмного_забезпечення (дата звернення: 10.05.2020).
14. Поняття абстракції. URL: https://uk.wikipedia.org/wiki/Абстракція (дата звернення: 10.05.2020)
15. Монолітна система. URL: https://uk.wikipedia.org/wiki/Монолітна_система (дата звернення: 10.05.2020)
16. C. Saternos, Client-Server Web Apps with JavaScript and Java: Rich, Scalable, and RESTful. United States: O'Reilly Media, 2014, pp. 260.
17. T. Carl, Deep Skin Architecture: Design Potentials of Multi-Layered Architectural Boundaries. Germany: Springer Vieweg, 2019, pp. 196.
18. S. Newman, Building Microservices: Designing Fine-Grained Systems. United States: O'Reilly Media, 2015, pp. 280.
19. Поняття .NET Framework. URL: https://uk.wikipedia.org/wiki/.NET_Framework (дата звернення: 11.05.2020).
20. J. Richter, CLR via C# 4th Edition. Redmond, United States: Microsoft Posts, 2013, pp. 896.
21. Поняття .NET Core. URL: https://ru.wikipedia.org/wiki/.NET_Core (дата звернення: 11.05.2020)
22. A. Lock, ASP.NET Core in Action. United States: Manning Publications, 2018, pp. 712.
23. JetBrains Rider. URL: https://www.jetbrains.com/ru-ru/rider/ (дата звернення: 11.05.2020)

24. Про специфікацію OWIN: стаття. URL: https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/owin?view=aspnetcore-3.1 (дата звернення: 11.05.2020)
25. S. Clearly, Concurrency in C# Cookbook Asynchronous, Parallel, and Multithreaded Programming. United States: O'Reilly Media, 2014, pp. 208.
26. Про Worker Service: стаття. URL: https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/host/hosted-services?view=aspnetcore-3.1&tabs=visual-studio (дата звернення: 11.05.2020)
27. Про VAPID аутентифікацію: стаття. URL: https://developers.google.com/web/fundamentals/push-notifications/web-push-protocol (дата звернення: 11.05.2020)
28. Про .NET Standart. URL: https://docs.microsoft.com/ru-ru/dotnet/standard/net-standard (дата звернення: 11. 05.2020)
29. Поняття ORM. URL: https://uk.wikipedia.org/wiki/Об%27єктно-реляційне_відображення (дата звернення: 11.05.2020)
30. J. P Smith, Entity Framework Core in Action. United States: Manning Publications, 2018, pp. 520.
31. A. Freeman, Pro Angular 6 Third Edition. New York, United States: Apress, 2018, pp. 804.
32. B. Cherny, Programming TypeScript: Making Your JavaScript Applications Scale. United States: O'Reilly Media, 2019, pp. 324.
33. Поняття PWA. URL: https://uk.wikipedia.org/wiki/Поступовий_вебзастосунок (дата звернення: 11.05.2020)
34. Поняття Docker. URL: https://uk.wikipedia.org/wiki/Docker (дата звернення: 11.05.2020)
35. J. D. Moore, Kubernetes: The Complete Guide To Master Kubernetes. United States: Amazon Digital Services LLC - KDP Print US, 2019, pp. 240.
36. Поняття Ingress. URL: https://kubernetes.io/docs/concepts/services-networking/ingress/ (дата звернення: 11.05.2020)
37. Поняття Helm. URL: https://helm.sh/ (дата звернення: 11.05.2020)

38. Громадський простір: веб-ресурс. URL: https://www.prostir.ua/category/grants/ (дата звернення: 16.05.2020)
39. Гурт: веб-ресурс. URL: https://gurt.org.ua/news/grants/ (дата звернення: 16.05.2020)
40. Визначення трудомісткості розробки програмного забезпечення: стаття. URL: https://lektsii.org/16-50350.html (дата звернення: 16.05.2020)
41. Визначення робочих годин: стаття. URL: https://i.factor.ua/ukr/journals/ot/2019/august/issue-16/article-70313.html (дата звернення: 16.05.2020)
42. СІТ 2:2018. Стандарт кафедри інформаційних технологій. Дипломний проект. Вимоги до змісту та оформлення [Чинний від 2018-09-03]. Вид. офіц. Івано-Франківськ, 2018. 43 с.

ДОДАТОК А

Сирцевий код Calculation Service

DefaultDataController.cs

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Sulfur.CalculationService.Common.Infrastructure;
using Sulfur.DataAccess.Enums;

namespace Sulfur.CalculationService.Api.Controllers
{
    [Route("/default-data")]
    [ApiController]
    public class DefaultDataController : ControllerBase
    {
        private readonly IEnumerable<IResourceDataFactory> _dataFactories;

        public DefaultDataController(IEnumerable<IResourceDataFactory>
dataFactories)
        {
            _dataFactories = dataFactories;
        }

        [HttpGet("{resourceType}")]
        public async Task<object> Get([FromRoute] Resource resourceType,
[FromQuery] int regionId)
        {
            var resourceData = await _dataFactories
                .First(x => x.IsProduce(resourceType))
                .GetDefaultData(regionId);

            return resourceData;
        }
    }
}
```

ResourceController.cs

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Sulfur.CalculationService.Api.Models;
using Sulfur.CalculationService.Common.Infrastructure;
using Sulfur.CalculationService.Common.Models;

namespace Sulfur.CalculationService.Api.Controllers
{
    [Route("/resources")]
    [ApiController]
    public class ResourceController : ControllerBase
    {
        private readonly IEnumerable<IResourceDataFactory> _dataFactories;

        public ResourceController(IEnumerable<IResourceDataFactory>
dataFactories)
        {
            _dataFactories = dataFactories;
```

```
        }

        [HttpGet]
        public async IAsyncEnumerable<ResourceData> Get([FromQuery] int
regionId)
        {
            foreach (var dataFactory in _dataFactories)
            {
                var resourceData = await dataFactory.Create(regionId);

                yield return resourceData;
            }
        }

        [HttpGet("{resourceType}")]
        public async Task<ResourceData> Get(ResourceRequestModel
resourceRequestModel)
        {
            //TODO: Add exception handler
            var resourceData = await _dataFactories
                .First(x => x.IsProduce(resourceRequestModel.Resource))
                .Create(resourceRequestModel.ResourceRequest);

            return resourceData;
        }
    }
}
```

ModelBinders.cs

```
using System.Collections.Generic;
using Sulfur.CalculationService.BL.ResourceRequestModelBinders;
using Sulfur.CalculationService.Common.Infrastructure;
using Sulfur.DataAccess.Enums;

namespace Sulfur.CalculationService.Api.ModelBinders
{
    public static class ModelBinders
    {
        public static Dictionary<Resource, IResourceRequestModelBinder>
ResourceRequestModelBinders =
            new Dictionary<Resource, IResourceRequestModelBinder>
            {
                { Resource.Methane, new MethaneRequestModelBinder() }
            };
    }
}
```

ResourceModelBinders.cs

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Sulfur.CalculationService.Api.Models;
using Sulfur.CalculationService.Common.Models;
using Sulfur.DataAccess.Enums;

namespace Sulfur.CalculationService.Api.ModelBinders
{
    public class ResourceModelBinder : IModelBinder
    {
        private const int DefaultRegionId = 1;
```

```
        public Task BindModelAsync(ModelBindingContext bindingContext)
        {
            var resourceRequestModel = new ResourceRequestModel();
            var valueProvider = bindingContext.ValueProvider;

            var resourceType =
valueProvider.GetValue("resourceType").FirstValue;

            Enum.TryParse(resourceType, out Resource resource);

            resourceRequestModel.Resource = resource;
            resourceRequestModel.ResourceRequest = GetResourceRequest(resource,
bindingContext);
            resourceRequestModel.ResourceRequest.RegionId =
                int.TryParse(valueProvider.GetValue("regionId").FirstValue, out
var regionId)
                    ? regionId
                    : DefaultRegionId;

            bindingContext.Result =
ModelBindingResult.Success(resourceRequestModel);

            return Task.CompletedTask;
        }

        private ResourceRequest GetResourceRequest(Resource resource,
ModelBindingContext bindingContext)
        {
            return ModelBinders.ResourceRequestModelBinders.Any(x => x.Key ==
resource)
                ?
ModelBinders.ResourceRequestModelBinders[resource].Bind(bindingContext)
                : new ResourceRequest();
        }
    }
}
```

ResourceModelBinderProvider.cs

```
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Sulfur.CalculationService.Api.Models;

namespace Sulfur.CalculationService.Api.ModelBinders
{
    public class ResourceModelBinderProvider : IModelBinderProvider
    {
        public IModelBinder GetBinder(ModelBinderProviderContext context)
        {
            return context.Metadata.ModelType == typeof(ResourceRequestModel)
                ? new ResourceModelBinder()
                : null;
        }
    }
}
```

ResourceRequestModel.cs

```
using Sulfur.CalculationService.Common.Models;
using Sulfur.DataAccess.Enums;

namespace Sulfur.CalculationService.Api.Models
{
    public class ResourceRequestModel
```

```
    {
        public Resource Resource { get; set; }

        public ResourceRequest ResourceRequest { get; set; }
    }
}
```

Startup.cs

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Sulfur.CalculationService.Api.ModelBinders;
using Sulfur.CalculationService.BL.Extensions.DependencyInjection;
using Sulfur.DataAccess.Extensions.DependencyInjection;

namespace Sulfur.CalculationService.Api
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add
services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllers(options =>
            {
                options.ModelBinderProviders.Insert(0, new
ResourceModelBinderProvider());
            });

            services.AddCors(options => options.AddDefaultPolicy(
                builder => builder
                    .AllowAnyOrigin()
                    .AllowAnyHeader()
                    .AllowAnyMethod()
            ));

            services.AddResourceDataFactories();

            services.AddUnitOfWork(Configuration,
"Sulfur.CalculationService.Api");
        }

        // This method gets called by the runtime. Use this method to configure
the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
```

```
                app.UseHttpsRedirection();
            }

            app.UseCors();

            app.UseRouting();

            app.UseAuthorization();

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

IResourceDataFactory.cs

```
using System.Threading.Tasks;
using Sulfur.CalculationService.Common.Models;
using Sulfur.DataAccess.Enums;

namespace Sulfur.CalculationService.Common.Infrastructure
{
    public interface IResourceDataFactory
    {
        bool IsProduce(Resource resource);

        Task<ResourceData> Create(ResourceRequest resourceRequest = null);

        Task<ResourceData> Create(int regionId);

        Task<object> GetDefaultData(int regionId);
    }
}
```

IResourceRequestModelBinder.cs

```
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Sulfur.CalculationService.Common.Models;

namespace Sulfur.CalculationService.Common.Infrastructure
{
    public interface IResourceRequestModelBinder
    {
        ResourceRequest Bind(ModelBindingContext bindingContext);
    }
}
```

ResourceDataFactory.cs

```
using System.Threading.Tasks;
using Sulfur.CalculationService.Common.Models;
using Sulfur.CalculationService.Common.Services.Abstract.Analyze;
using Sulfur.CalculationService.Common.Services.Abstract.Gather;
using Sulfur.DataAccess.Enums;

namespace Sulfur.CalculationService.Common.Infrastructure
{
    public abstract class ResourceDataFactory : IResourceDataFactory
    {
        public Task<ResourceData> Create(ResourceRequest resourceRequest = null)
        {
```

```
            var request = resourceRequest ?? DefaultRequest;
            var isDefault = resourceRequest is null;
            return Create(request, isDefault);
        }

        public Task<ResourceData> Create(int regionId)
        {
            var request = DefaultRequest;
            request.RegionId = regionId;

            return Create(request, true);
        }

        public async Task<object> GetDefaultData(int regionId)
        {
            var request = DefaultRequest;
            request.RegionId = regionId;

            return await GetRawData(request, true);
        }

        public virtual bool IsProduce(Resource resource)
        {
            return resource == Resource;
        }

        protected abstract Resource Resource { get; }

        protected abstract ResourceRequest DefaultRequest { get; }

        protected abstract IGatherService GatherService { get; }

        protected abstract IAnalyzeService AnalyzeService { get; }

        private async Task<ResourceData> Create(ResourceRequest resourceRequest,
bool isDefault)
        {
            var analyzeService = AnalyzeService;
            var rawData = await GetRawData(resourceRequest, isDefault);
            var result = analyzeService.Analyze(rawData);

            return result;
        }

        private Task<object> GetRawData(ResourceRequest resourceRequest, bool
isDefault)
        {
            var request = resourceRequest;
            var gatherService = GatherService;
            gatherService.IsDefaultRequest = isDefault;

            return gatherService.Gather(request);
        }
    }
}
```

ResourceData.cs

```
namespace Sulfur.CalculationService.Common.Models
{
    public class ResourceData
    {
```

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

        public string Name { get; set; }

        public decimal Price { get; set; }

        public double Energy { get; set; }

        public double EnergyCost { get; set; }
    }
}
```

ResourceRequest.cs

```
namespace Sulfur.CalculationService.Common.Models
{
    public class ResourceRequest
    {
        public int RegionId { get; set; }
    }
}
```

IAnalyzeService.cs

```
using Sulfur.CalculationService.Common.Models;

namespace Sulfur.CalculationService.Common.Services.Abstract.Analyze
{
    public interface IAnalyzeService
    {
        ResourceData Analyze(object rawData);
    }
}
```

ResourceAnalyzeService.cs

```
using Sulfur.CalculationService.Common.Models;

namespace Sulfur.CalculationService.Common.Services.Abstract.Analyze
{
    public abstract class ResourceAnalyzeService<TRawData> : IAnalyzeService
    {
        public abstract ResourceData Analyze(TRawData rawData);

        public ResourceData Analyze(object rawData)
        {
            return Analyze((TRawData)rawData);
        }
    }
}
```

IGatherService.cs

```
using System.Threading.Tasks;
using Sulfur.CalculationService.Common.Models;

namespace Sulfur.CalculationService.Common.Services.Abstract.Gather
{
    public interface IGatherService
    {
        bool IsDefaultRequest { set; }

        Task<object> Gather(ResourceRequest resourceRequest);
    }
}
```

Продовження додатку А

ResourceGatherService.cs

```
using System;
using System.Threading.Tasks;
using Sulfur.CalculationService.Common.Models;

namespace Sulfur.CalculationService.Common.Services.Abstract.Gather
{
    public abstract class ResourceGatherService<TResourceRequest, TRawData> :
IGatherService
        where TResourceRequest : ResourceRequest
    {
        private const string AlreadyAssignedMessage = "IsDefault have been
already assigned";
        private bool? _isDefaultRequest;
        private TResourceRequest _defaultRequest;

        public abstract Task<TRawData> Gather(TResourceRequest resourceRequest);

        public async Task<object> Gather(ResourceRequest resourceRequest)
        {
            _defaultRequest = (TResourceRequest)resourceRequest;

            return await Gather((TResourceRequest)resourceRequest);
        }

        public bool IsDefaultRequest
        {
            set
            {
                if (_isDefaultRequest.HasValue)
                {
                    throw new InvalidOperationException(AlreadyAssignedMessage);
                }

                _isDefaultRequest = value;
            }
        }

        /// <summary>
        /// Replace value of <see cref="T:ResourceRequest"/> to your given
value, if it's default request
        /// </summary>
        /// <param name="resourceRequestValue">Property of <see
cref="T:ResourceRequest"/></param>
        /// <param name="value">Your default value</param>
        /// <typeparam name="TValue">Type of <see cref="T:ResourceRequest"/>
property or default value</typeparam>
        /// <returns>Default value if it's default request, property of <see
cref="T:ResourceRequest"/> otherwise</returns>
        protected TValue Replace<TValue>(Func<TResourceRequest, TValue>
resourceRequestValue, TValue value)
        {
            var result = _isDefaultRequest.Value ? value :
resourceRequestValue(_defaultRequest);

            return result;
        }
    }
}
```

MethaneDataFactory.cs

```
using Sulfur.CalculationService.BL.Models.ResourceRequest;
using Sulfur.CalculationService.BL.Services.Analyze;
using Sulfur.CalculationService.BL.Services.Gather;
using Sulfur.CalculationService.Common.Infrastructure;
using Sulfur.CalculationService.Common.Models;
using Sulfur.CalculationService.Common.Services.Abstract.Analyze;
using Sulfur.CalculationService.Common.Services.Abstract.Gather;
using Sulfur.DataAccess.Abstraction;
using Sulfur.DataAccess.Enums;

namespace Sulfur.CalculationService.BL.Factories
{
    public class MethaneDataFactory : ResourceDataFactory
    {
        private readonly IUnitOfWork _unitOfWork;

        public MethaneDataFactory(IUnitOfWork unitOfWork)
        {
            _unitOfWork = unitOfWork;
        }

        protected override Resource Resource => Resource.Methane;

        protected override ResourceRequest DefaultRequest => new MethaneRequest
        {
            PipeDiameter = 300,
            PipeLength = 10,
            Area = 1,
            WasteAmount = 1,
        };

        protected override IGatherService GatherService => new
MethaneGatherService(_unitOfWork);

        protected override IAnalyzeService AnalyzeService => new
MethaneAnalyzeService();
    }
}
```

MethaneRawData.cs

```
namespace Sulfur.CalculationService.BL.Models.RawData
{
    public class MethaneRawData
    {
        public string Name { get; set; }

        public double EnvironmentTemperature { get; set; }

        public double MethaneQuality { get; set; }

        public decimal MethanePrice { get; set; }

        public double PipeDiameter { get; set; }

        public double PipeLength { get; set; }

        public double Area { get; set; }

        public double WasteAmount { get; set; }

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

```
    }
}
```

MethaneRequest.cs

```
namespace Sulfur.CalculationService.BL.Models.ResourceRequest
{
    public class MethaneRequest : Common.Models.ResourceRequest
    {
        public double EnvironmentTemperature { get; set; }

        public double MethaneQuality { get; set; }

        public decimal MethanePrice { get; set; }

        public double PipeDiameter { get; set; }

        public double PipeLength { get; set; }

        public double Area { get; set; }

        public double WasteAmount { get; set; }

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

MethaneRequestModelBinder.cs

```
using System.Globalization;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Sulfur.CalculationService.BL.Models.ResourceRequest;
using Sulfur.CalculationService.Common.Infrastructure;
using Sulfur.CalculationService.Common.Models;

namespace Sulfur.CalculationService.BL.ResourceRequestModelBinders
{
    public class MethaneRequestModelBinder : IResourceRequestModelBinder
    {
        public ResourceRequest Bind(ModelBindingContext bindingContext)
        {
            var valueProvider = bindingContext.ValueProvider;
            var environmentTemperature =
                double.Parse(valueProvider.GetValue("temp").FirstValue,
CultureInfo.InvariantCulture);
            var methaneQuality =
                double.Parse(valueProvider.GetValue("quality").FirstValue,
CultureInfo.InvariantCulture);
            var methanePrice =
                decimal.Parse(valueProvider.GetValue("price").FirstValue,
CultureInfo.InvariantCulture);
            var pipeDiameter =
                double.Parse(valueProvider.GetValue("diameter").FirstValue,
CultureInfo.InvariantCulture);
            var pipeLength =
                double.Parse(valueProvider.GetValue("length").FirstValue,
CultureInfo.InvariantCulture);
            var area =
                double.Parse(valueProvider.GetValue("area").FirstValue,
CultureInfo.InvariantCulture);
            var wasteAmount =
                double.Parse(valueProvider.GetValue("waste").FirstValue,
CultureInfo.InvariantCulture);
```

```
            var seaLevel =
                int.Parse(valueProvider.GetValue("level").FirstValue,
CultureInfo.InvariantCulture);

            return new MethaneRequest
            {
                EnvironmentTemperature = environmentTemperature,
                MethaneQuality = methaneQuality,
                MethanePrice = methanePrice,
                PipeDiameter = pipeDiameter,
                PipeLength = pipeLength,
                Area = area,
                WasteAmount = wasteAmount,
                SeaLevel = seaLevel
            };
        }
    }
}
```

MethaneAnalyzeService.cs

```
using System;
using Sulfur.CalculationService.BL.Models.RawData;
using Sulfur.CalculationService.Common.Models;
using Sulfur.CalculationService.Common.Services.Abstract.Analyze;
using Sulfur.DataAccess.Enums;

namespace Sulfur.CalculationService.BL.Services.Analyze
{
    public class MethaneAnalyzeService : ResourceAnalyzeService<MethaneRawData>
    {
        public override ResourceData Analyze(MethaneRawData rawData)
        {
            var denominator = 1 + (rawData.EnvironmentTemperature / 293.15 - 1)
*
                              Math.Exp(CalculateR(
                                  rawData.PipeDiameter,
                                  rawData.PipeLength,
                                  rawData.Area,
                                  rawData.WasteAmount));

            var volume = CalculateP(rawData.SeaLevel) / denominator;
            var energy = volume * rawData.MethaneQuality;
            var energyCost = (double)rawData.MethanePrice / energy;

            return new ResourceData
            {
                Id = (int)Resource.Methane,
                Name = rawData.Name,
                Price = rawData.MethanePrice,
                Energy = Math.Round(energy, 4),
                EnergyCost = Math.Round(energyCost, 4)
            };
        }

        private double CalculateP(int seaLevel)
        {
            var result = 1 - 1.15e-4 * seaLevel;

            return result;
        }
```

```
        private double CalculateR(
            double pipeDiameter,
            double pipeLength,
            double area,
            double wasteAmount)
        {
            var numerator = 1.96 * pipeDiameter * pipeLength + 16.17 * area;
            var result = -21.77 * pipeDiameter * pipeLength - (numerator /
wasteAmount);

            return result;
        }
    }
}
```

MethaneGatherService.cs

```
using System.Linq;
using System.Threading.Tasks;
using Sulfur.CalculationService.BL.Models.RawData;
using Sulfur.CalculationService.BL.Models.ResourceRequest;
using Sulfur.CalculationService.Common.Services.Abstract.Gather;
using Sulfur.DataAccess.Abstraction;
using Sulfur.DataAccess.Models;

namespace Sulfur.CalculationService.BL.Services.Gather
{
    public class MethaneGatherService : ResourceGatherService<MethaneRequest,
MethaneRawData>
    {
        private readonly IUnitOfWork _unitOfWork;

        public MethaneGatherService(IUnitOfWork unitOfWork)
        {
            _unitOfWork = unitOfWork;
        }

        public override async Task<MethaneRawData> Gather(MethaneRequest
resourceRequest)
        {
            var region = await
_unitOfWork.RegionRepository.GetAsync(resourceRequest.RegionId);
            var resourcePrices = await _unitOfWork.ResourcePriceRepository.
                GetAsync(
                    filter: x => x.ResourceId ==
(int)DataAccess.Enums.Resource.Methane &&
                                 x.RegionId == region.Id,
                    includeProperties: nameof(ResourcePrice.Resource));

            var resourcePrice = resourcePrices.First();
            var methaneCollection = await
_unitOfWork.MethaneRepository.GetAsync(
                filter: x => x.ResourcePriceId == resourcePrice.Id);
            var methane = methaneCollection.First();

            return new MethaneRawData
            {
                Name = resourcePrice.Resource.Name,
                EnvironmentTemperature =
                    Replace(request => request.EnvironmentTemperature,
region.Temperature),
                MethaneQuality =
```

Продовження додатку А

```
                    Replace(request => request.MethaneQuality, methane.Quality),
                MethanePrice =
                    Replace(request => request.MethanePrice,
resourcePrice.Price),
                PipeDiameter = resourceRequest.PipeDiameter,
                PipeLength = resourceRequest.PipeLength,
                Area = resourceRequest.Area,
                WasteAmount = resourceRequest.WasteAmount,
                SeaLevel = Replace(request => request.SeaLevel, region.SeaLevel)
            };
        }
    }
}
```

ДОДАТОК Б

Сирцевий код Monitor Service

Worker.cs

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using Sulfur.MonitorService.BL.Services.Abstract;

namespace Sulfur.MonitorService.Worker
{
    public class Worker : BackgroundService
    {
        private readonly ILogger<Worker> _logger;
        private readonly ICollectingRunner _collectingRunner;

        public Worker(ILogger<Worker> logger, ICollectingRunner
collectingRunner)
        {
            _logger = logger;
            _collectingRunner = collectingRunner;
        }

        protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
        {
            while (!stoppingToken.IsCancellationRequested)
            {
                _logger.LogInformation("Worker running at: {time}",
DateTimeOffset.Now);

                await _collectingRunner.RunAsync(stoppingToken);

                await Task.Delay(TimeSpan.FromMinutes(1), stoppingToken);
            }
        }
    }
}
```

ICollectingRunner.cs

```
using System.Threading;
using System.Threading.Tasks;

namespace Sulfur.MonitorService.BL.Services.Abstract
{
    public interface ICollectingRunner
    {
        Task RunAsync(CancellationToken cancellationToken = default);
    }
}
```

IDataCollector.cs

```
using System.Threading;
using System.Threading.Tasks;

namespace Sulfur.MonitorService.BL.Services.Abstract
{
```

```
    public interface IDataCollector
    {
        Task CollectDataAsync(CancellationToken cancellationToken = default);
    }
}
```

IDataSourceService.cs

```
using System.Threading.Tasks;
using Sulfur.DataAccess.Enums;

namespace Sulfur.MonitorService.BL.Services.Abstract
{
    public interface IDataSourceService<T>
    {
        DataProvider DataProvider { get; }

        Task<T> GetDataAsync(string path);
    }
}
```

Parser.cs

```
using System.Threading.Tasks;
using AngleSharp;
using AngleSharp.Dom;
using Sulfur.DataAccess.Enums;

namespace Sulfur.MonitorService.BL.Services.Abstract
{
    public abstract class Parser<T> : IDataSourceService<T>
    {
        public abstract DataProvider DataProvider { get; }

        public async Task<T> GetDataAsync(string path)
        {
            var config = Configuration.Default.WithDefaultLoader();
            var context = BrowsingContext.New(config);
            var addressUrl = new Url(path);
            var document = await context.OpenAsync(addressUrl);
            var result = Parse(document);

            return result;
        }

        public abstract T Parse(IDocument document);
    }
}
```

ResourcePriceDataCollector.cs

```
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Text.Json;
using System.Threading;
using System.Threading.Tasks;
using Sulfur.DataAccess.Abstraction;
using Sulfur.DataAccess.Models;
using Sulfur.MonitorService.BL.Services.Abstract;

namespace Sulfur.MonitorService.BL.Services.Collectors
{
```

Продовження додатку Б

```
    public class ResourcePriceDataCollector : IDataCollector
    {
        private readonly IEnumerable<IDataSourceService<decimal>>
_dataSourceServices;
        private readonly IUnitOfWork _unitOfWork;
        private readonly HttpClient _httpClient;

        public ResourcePriceDataCollector(
            IEnumerable<IDataSourceService<decimal>> dataSourceServices,
            IUnitOfWork unitOfWork)
        {
            _dataSourceServices = dataSourceServices;
            _unitOfWork = unitOfWork;
            _httpClient = new HttpClient();
        }

        public async Task CollectDataAsync(CancellationToken cancellationToken =
default)
        {
            var resources = await _unitOfWork.ResourceRepository.GetAsync(
                includeProperties: nameof(Resource.ResourcePrices),
                cancellationToken: cancellationToken);

            foreach (var resource in resources)
            {
                var resourcePrices = resource.ResourcePrices;

                if (!resourcePrices.Any())
                {
                    continue;
                }

                var previousPrice = resourcePrices.Min(x => x.Price);
                var prices = new List<decimal>(resourcePrices.Count);

                foreach (var resourcePrice in resourcePrices)
                {
                    var service = _dataSourceServices.Single(x =>
(int)x.DataProvider == resourcePrice.DataProviderId);

                    var price = await service.GetDataAsync(resourcePrice.Link);

                    if (price != resourcePrice.Price)
                    {
                        resourcePrice.Price = price;
                    }

                    prices.Add(price);
                }

                _unitOfWork.ResourceRepository.Update(resource);

                await SendNotification(resource, previousPrice, prices.Min());
            }

            await _unitOfWork.SaveChangesAsync(cancellationToken);
        }

        private async Task SendNotification(Resource resource, decimal
previousPrice, decimal currentPrice)
        {
```

```
            if (previousPrice == currentPrice)
            {
                return;
            }

            var message = new
            {
                ResourceName = resource.Name,
                Rank = 0,
                Price = currentPrice,
                IsCheaper = currentPrice < previousPrice
            };

            var options = new JsonSerializerOptions
            {
                PropertyNamingPolicy = JsonNamingPolicy.CamelCase
            };

            var json = JsonSerializer.Serialize(message, options);
            await _httpClient.PostAsync(
                "http://localhost:8081/notify",
                new StringContent(json, Encoding.UTF8, "application/json"));
        }
    }
}
```

Ua104PriceParser.cs

```
using AngleSharp.Dom;
using Sulfur.DataAccess.Enums;
using Sulfur.MonitorService.BL.Services.Abstract;

namespace Sulfur.MonitorService.BL.Services.DataSource
{
    public class Ua104PriceParser : Parser<decimal>
    {
        private const string MethaneUnits = "грн/м3";

        public override DataProvider DataProvider => DataProvider.Ua104Price;
        public override decimal Parse(IDocument document)
        {
            var cellSelector = "div.b-webt-digit div.digit";
            var cell = document.QuerySelector(cellSelector);
            var price = cell?.TextContent.Replace(MethaneUnits, string.Empty);

            return decimal.TryParse(price, out var result)
                ? result
                : new decimal();
        }
    }
}
```

CollectingRunner.cs

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
using Sulfur.MonitorService.BL.Services.Abstract;

namespace Sulfur.MonitorService.BL.Services
{
    public class CollectingRunner : ICollectingRunner
```

Продовження додатку Б

```
    {
        private readonly IEnumerable<IDataCollector> _dataCollectors;

        public CollectingRunner(IEnumerable<IDataCollector> dataCollectors)
        {
            _dataCollectors = dataCollectors;
        }

        public async Task RunAsync(CancellationToken cancellationToken =
default)
        {
            foreach (var dataCollector in _dataCollectors)
            {
                await dataCollector.CollectDataAsync(cancellationToken);

                await Task.Delay(TimeSpan.FromDays(1), cancellationToken);
            }
        }
    }
}
```

ДОДАТОК В

Сирцевий код Notification Service

Startup.cs

```
using Lib.Net.Http.WebPush;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Sulfur.DataAccess.Extensions.DependencyInjection;
using Sulfur.NotificationService.BL.Extensions.DependencyInjection;
using Sulfur.NotificationService.BL.Models;

namespace Sulfur.NotificationService.Api
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add
services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllers();

            services.AddCors(options => options.AddDefaultPolicy(
                builder => builder
                    .AllowAnyOrigin()
                    .AllowAnyHeader()
                    .AllowAnyMethod()
            ));


services.Configure<PushNotificationOptions>(Configuration.GetSection("PushNotifi
cation"));

            services.AddHttpClient<PushServiceClient>();

            services.AddUnitOfWork(Configuration,
"Sulfur.NotificationService.Api");

            services.AddCommonServices();

        }

        // This method gets called by the runtime. Use this method to configure
the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
```

```
            else
            {
                app.UseHttpsRedirection();
            }

            app.UseCors();

            app.UseRouting();

            app.UseAuthorization();

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

PublicKeyController.cs

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Options;
using Sulfur.NotificationService.BL.Models;

namespace Sulfur.NotificationService.Api.Controllers
{
    [Route("public-key")]
    [ApiController]
    public class PublicKeyController : ControllerBase
    {
        private readonly PushNotificationOptions _options;

        public PublicKeyController(IOptions<PushNotificationOptions> options)
        {
            _options = options.Value;
        }

        [HttpGet]
        public string Get()
        {
            return _options.PublicKey;
        }
    }
}
```

NotifyController.cs

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Sulfur.NotificationService.BL.Models;
using Sulfur.NotificationService.BL.Services.Abstract;

namespace Sulfur.NotificationService.Api.Controllers
{
    [Route("notify")]
    [ApiController]
    public class NotifyController : ControllerBase
    {
        private readonly INotifyService _notifyService;

        public NotifyController(INotifyService notifyService)
        {
```

```
            _notifyService = notifyService;
        }

        [HttpPost]
        public async Task Post([FromBody] ResourceChangeInfo changeInfo)
        {
            await _notifyService.NotifyAsync(changeInfo);
        }
    }
}
```

SubscriptionController.cs

```
using System.Threading.Tasks;
using Lib.Net.Http.WebPush;
using Microsoft.AspNetCore.Mvc;
using Sulfur.NotificationService.BL.Services.Abstract;

namespace Sulfur.NotificationService.Api.Controllers
{
    [Route("subscription")]
    [ApiController]
    public class SubscriptionController : ControllerBase
    {
        private readonly ISubscriptionService _subscriptionService;

        public SubscriptionController(ISubscriptionService subscriptionService)
        {
            _subscriptionService = subscriptionService;
        }

        //TODO: Consider returning bool
        [HttpPost]
        public async Task Post([FromBody] PushSubscription subscription)
        {
            await _subscriptionService.SubscribeAsync(subscription);
        }

        //TODO: Consider returning bool
        [HttpDelete("{endpoint}")]
        public async Task Delete(string endpoint)
        {
            await _subscriptionService.UnsubscribeAsync(endpoint);
        }
    }
}
```

PushNotification.cs

```
namespace Sulfur.NotificationService.BL.Models
{
    public class PushNotification
    {
        public string Title { get; set; }

        public string Body { get; set; }

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

PushNotificationOptions.cs

```
namespace Sulfur.NotificationService.BL.Models
```

```
{
    public class PushNotificationOptions
    {
        public string PublicKey { get; set; }

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

ResourceChangeInfo.cs

```
namespace Sulfur.NotificationService.BL.Models
{
    public class ResourceChangeInfo
    {
        public string ResourceName { get; set; }

        public int Rank { get; set; }

        public decimal Price { get; set; }

        public bool IsCheaper { get; set; }
    }
}
```

INotifyService.cs

```
using System.Threading;
using System.Threading.Tasks;
using Sulfur.NotificationService.BL.Models;

namespace Sulfur.NotificationService.BL.Services.Abstract
{
    public interface INotifyService
    {
        Task NotifyAsync(ResourceChangeInfo changeInfo, CancellationToken
cancellationToken = default);
    }
}
```

ISubscriptionService.cs

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Lib.Net.Http.WebPush;

namespace Sulfur.NotificationService.BL.Services.Abstract
{
    public interface ISubscriptionService
    {
        Task<IEnumerable<PushSubscription>> Subscriptions();

        Task SubscribeAsync(PushSubscription pushSubscription);

        Task UnsubscribeAsync(string endpoint);
    }
}
```

NotifyService.cs

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Lib.Net.Http.WebPush;
using Lib.Net.Http.WebPush.Authentication;
```

Продовження додатку В

```
using Microsoft.Extensions.Options;
using Sulfur.NotificationService.BL.Extensions;
using Sulfur.NotificationService.BL.Models;
using Sulfur.NotificationService.BL.Services.Abstract;
using Sulfur.NotificationService.BL.Static;

namespace Sulfur.NotificationService.BL.Services
{
    public class NotifyService : INotifyService
    {
        private const string UnsubscribeMessage = "Subscription has been
removed";
        private readonly ISubscriptionService _subscriptionService;
        private readonly PushServiceClient _pushServiceClient;

        public NotifyService(
            ISubscriptionService subscriptionService,
            PushServiceClient pushServiceClient,
            IOptions<PushNotificationOptions> options)
        {
            _subscriptionService = subscriptionService;
            _pushServiceClient = pushServiceClient;
            _pushServiceClient.DefaultAuthentication =
                new VapidAuthentication(options.Value.PublicKey,
options.Value.PrivateKey);
        }

        public async Task NotifyAsync(
            ResourceChangeInfo changeInfo,
            CancellationToken cancellationToken = default)
        {
            var arrow = !changeInfo.IsCheaper ? "⬆" : "⬇";
            var notification = new AngularPushNotification
            {
                Title = $"{changeInfo.ResourceName}'s price has been changed",
                Body = $"Rank: {changeInfo.Rank}\n{arrow}Price:
{changeInfo.Price}",
                Icon = Images.Icon
            };
            var pushMessage = notification.ToJson().ToPushMessage();
            var subscriptions = await _subscriptionService.Subscriptions();

            foreach (var subscription in subscriptions)
            {
                try
                {
                    await _pushServiceClient.RequestPushMessageDeliveryAsync(
                        subscription,
                        pushMessage,
                        cancellationToken);
                }
                catch (PushServiceClientException exception)
                {
                    Console.WriteLine(exception);
                    await
_subscriptionService.UnsubscribeAsync(subscription.Endpoint);
                    Console.WriteLine(UnsubscribeMessage);
                }
            }
        }
    }
```

```
}
```

SubscriptionService.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Threading.Tasks;
using Lib.Net.Http.WebPush;
using Microsoft.Extensions.Caching.Memory;
using Sulfur.DataAccess.Abstraction;
using Sulfur.DataAccess.Models;
using Sulfur.NotificationService.BL.Services.Abstract;

namespace Sulfur.NotificationService.BL.Services
{
    public class SubscriptionService : ISubscriptionService, IDisposable
    {
        private readonly IUnitOfWork _unitOfWork;
        private readonly IMemoryCache _cache;
        private readonly object _cacheKey;

        public SubscriptionService(IUnitOfWork unitOfWork)
        {
            _unitOfWork = unitOfWork;
            var options = new MemoryCacheOptions();
            _cache = new MemoryCache(options);
            _cacheKey = new object();
        }

        public async Task<IEnumerable<PushSubscription>> Subscriptions()
        {
            var subscriptions = await _cache.GetOrCreateAsync(_cacheKey, async
entry =>
                {
                    var collection = await
_unitOfWork.SubscriptionRepository.GetAsync(
                        includeProperties:
nameof(Subscription.KeysDictionaryModels));

                    return collection;
                });

            var pushSubscriptions = subscriptions.Select(x => new
PushSubscription
            {
                Endpoint = x.Endpoint,
                Keys = x.Keys
            });

            return pushSubscriptions;
        }

        public async Task SubscribeAsync(PushSubscription pushSubscription)
        {
            var subscription = new Subscription
            {
                Endpoint = pushSubscription.Endpoint,
                Keys = pushSubscription.Keys
            };
```

Продовження додатку В

```
            await _unitOfWork.SubscriptionRepository.InsertAsync(subscription);
            await _unitOfWork.SaveChangesAsync();

            _cache.Remove(_cacheKey);
        }

        public async Task UnsubscribeAsync(string endpoint)
        {
            endpoint = WebUtility.UrlDecode(endpoint);
            await _unitOfWork.SubscriptionRepository.DeleteAsync(x =>
x.Endpoint.Equals(endpoint));
            await _unitOfWork.SaveChangesAsync();

            _cache.Remove(_cacheKey);
        }

        public void Dispose()
        {
            _cache.Dispose();
        }
    }
}
```

ДОДАТОК Г

Сирцевий код DataAccess

IModel.cs

```
namespace Sulfur.DataAccess.Abstraction
{
    public interface IModel
    {
        int Id { get; set; }
    }
}
```

IRepository.cs

```
using System;
using System.Collections.Generic;
using System.Linq.Expressions;
using System.Threading;
using System.Threading.Tasks;

namespace Sulfur.DataAccess.Abstraction
{
    public interface IRepository<TEntity>
        where TEntity : class
    {
        Task<IEnumerable<TEntity>> GetAsync(
            Expression<Func<TEntity, bool>> filter = null,
            Expression<Func<TEntity, object>> orderBy = null,
            string includeProperties = "",
            int? page = null,
            int? amount = null,
            CancellationToken cancellationToken = default);

        Task<TEntity> GetAsync(object id, CancellationToken cancellationToken =
default);

        Task InsertAsync(TEntity entity, CancellationToken cancellationToken =
default);

        void Update(TEntity entity);

        Task DeleteAsync(object id);

        Task DeleteAsync(TEntity entity);

        Task DeleteAsync(Expression<Func<TEntity, bool>> predicate);

        Task<int> CountAsync(CancellationToken cancellationToken = default);

        Task<int> CountAsync(
            Expression<Func<TEntity, bool>> predicate,
            CancellationToken cancellationToken = default);
    }
}
```

ISulfurDbContext.cs

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

```
using System.Threading.Tasks;
using Sulfur.DataAccess.Models;

namespace Sulfur.DataAccess.Abstraction
{
    public interface ISulfurDbContext : IDisposable, IAsyncDisposable
    {
        IQueryable<TEntity> Query<TEntity>()
            where TEntity : class;

        IQueryable<Resource> Resources { get; }

        IQueryable<Subscription> Subscriptions { get; }

        IQueryable<SubscriptionKeysDictionaryModel> SubscriptionKeys { get; }

        IQueryable<Methane> Methanes { get; }

        Task AddAsync<TEntity>(TEntity entity, CancellationToken
cancellationToken = default)
            where TEntity : class;

        void Update<TEntity>(TEntity entity)
            where TEntity : class;

        void Remove<TEntity>(TEntity entity)
            where TEntity : class;

        void RemoveRange<TEntity>(IEnumerable<TEntity> entities)
            where TEntity : class;

        Task<int> SaveChangesAsync(CancellationToken cancellationToken =
default);
    }
}
```

IUnitOfWork.cs

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Sulfur.DataAccess.Models;

namespace Sulfur.DataAccess.Abstraction
{
    public interface IUnitOfWork : IDisposable, IAsyncDisposable
    {
        IRepository<Resource> ResourceRepository { get; }

        IRepository<ResourcePrice> ResourcePriceRepository { get; }

        IRepository<Methane> MethaneRepository { get; }

        IRepository<Region> RegionRepository { get; }

        IRepository<Subscription> SubscriptionRepository { get; }

        Task SaveChangesAsync(CancellationToken cancellationToken = default);
    }
}
```

SulfurDbContext.cs

Продовження додатку Г

```
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using Sulfur.DataAccess.Abstraction;
using Sulfur.DataAccess.Models;

namespace Sulfur.DataAccess.Data
{
    public class SulfurDbContext : DbContext, ISulfurDbContext
    {
        public SulfurDbContext(DbContextOptions<SulfurDbContext> options)
            : base(options)
        {
        }

        public DbSet<Resource> Resources { get; set; }

        public DbSet<Subscription> Subscriptions { get; set; }

        public DbSet<SubscriptionKeysDictionaryModel> SubscriptionKeys { get;
set; }

        public DbSet<Methane> Methanes { get; set; }

        IQueryable<TEntity> ISulfurDbContext.Query<TEntity>()
            => Set<TEntity>();

        IQueryable<Resource> ISulfurDbContext.Resources =>
Resources.AsQueryable();

        IQueryable<Subscription> ISulfurDbContext.Subscriptions =>
Subscriptions.AsQueryable();

        IQueryable<SubscriptionKeysDictionaryModel>
ISulfurDbContext.SubscriptionKeys => SubscriptionKeys.AsQueryable();

        IQueryable<Methane> ISulfurDbContext.Methanes => Methanes.AsQueryable();

        async Task ISulfurDbContext.AddAsync<TEntity>(
            TEntity entity,
            CancellationToken cancellationToken)
        {
            await base.AddAsync(entity, cancellationToken);
        }

        void ISulfurDbContext.Update<TEntity>(TEntity entity)
        {
            base.Update(entity);
        }

        void ISulfurDbContext.Remove<TEntity>(TEntity entity)
        {
            base.Remove(entity);
        }

        void ISulfurDbContext.RemoveRange<TEntity>(IEnumerable<TEntity>
entities)
        {
            base.RemoveRange(entities);
```

```
        }

        protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
        {
            if (!optionsBuilder.IsConfigured)
            {

optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=EFProviders
.InMemory;Trusted_Connection=True;ConnectRetryCount=0");
            }
        }

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

            modelBuilder
                .Entity<SubscriptionKeysDictionaryModel>()
                .HasKey(x => new { x.SubscriptionId, x.Key });

            modelBuilder
                .Entity<Resource>()
                .HasKey(x => x.Id);

            modelBuilder
                .Entity<Methane>()
                .HasOne(x => x.ResourcePrice)
                .WithMany()
                .OnDelete(DeleteBehavior.Restrict);

            modelBuilder
                .Entity<ResourcePrice>()
                .HasOne(x => x.Region)
                .WithMany(x => x.ResourcePrices)
                .OnDelete(DeleteBehavior.Restrict);

            SulfurData.SeedData(modelBuilder);
        }
    }
}
```

Repository.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Linq.Expressions;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using Sulfur.DataAccess.Abstraction;

namespace Sulfur.DataAccess.Repositories
{
    public class Repository<TEntity> : IRepository<TEntity>
        where TEntity : class, IModel
    {
        private const string RecordsNotFoundMessage = "There is no records with
such id";
        private readonly ISulfurDbContext _dbContext;
        private readonly IQueryable<TEntity> _query;
```

Продовження додатку Г

```
        public Repository(ISulfurDbContext dbContext)
        {
            _dbContext = dbContext ?? throw new
ArgumentNullException(nameof(dbContext));
            _query = dbContext.Query<TEntity>();
        }

        public async Task<IEnumerable<TEntity>> GetAsync(
            Expression<Func<TEntity, bool>> filter = null,
            Expression<Func<TEntity, object>> orderBy = null,
            string includeProperties = "",
            int? page = null,
            int? amount = null,
            CancellationToken cancellationToken = default)
        {
            var query = _query;

            if (filter != null)
            {
                query = query.Where(filter);
            }

            query = includeProperties
                .Split(new[] {',', ' '}, StringSplitOptions.RemoveEmptyEntries)
                .Aggregate(query, (current, includeProperty) =>
current.Include(includeProperty));

            if (orderBy != null)
            {
                query = query.OrderBy(orderBy);
            }

            if (page.HasValue && amount.HasValue)
            {
                query = query
                    .Skip((page.Value - 1) * amount.Value)
                    .Take(amount.Value);
            }

            var result = await query.ToListAsync(cancellationToken);

            return result;
        }

        public async Task<TEntity> GetAsync(object id, CancellationToken
cancellationToken = default)
        {
            var result = await _query.FirstOrDefaultAsync(x => x.Id == (int)id,
cancellationToken);

            return result;
        }

        public async Task InsertAsync(TEntity entity, CancellationToken
cancellationToken = default)
        {
            await _dbContext.AddAsync(entity, cancellationToken);
        }

        public void Update(TEntity entity)
```

```
        {
            _dbContext.Update(entity);
        }

        public async Task DeleteAsync(object id)
        {
            var entity = await GetAsync(id)
                         ?? throw new
InvalidOperationException(RecordsNotFoundMessage);

            _dbContext.Remove(entity);
        }

        public Task DeleteAsync(TEntity entity)
        {
            _dbContext.Remove(entity);

            return Task.CompletedTask;
        }

        public Task DeleteAsync(Expression<Func<TEntity, bool>> predicate)
        {
            _dbContext.RemoveRange(predicate is null ? _query :
_query.Where(predicate));

            return Task.CompletedTask;
        }

        public async Task<int> CountAsync(CancellationToken cancellationToken =
default)
        {
            var result = await _query.CountAsync(cancellationToken);

            return result;
        }

        public async Task<int> CountAsync(
            Expression<Func<TEntity, bool>> predicate,
            CancellationToken cancellationToken = default)
        {
            var result = await _query.CountAsync(predicate, cancellationToken);

            return result;
        }
    }
}
```

UnitOfWork.cs

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Sulfur.DataAccess.Abstraction;
using Sulfur.DataAccess.Models;
using Sulfur.DataAccess.Repositories;

namespace Sulfur.DataAccess
{
    public class UnitOfWork : IUnitOfWork
    {
        private readonly ISulfurDbContext _dbContext;
```

```
        public UnitOfWork(ISulfurDbContext dbContext)
        {
            _dbContext = dbContext ?? throw new
ArgumentNullException(nameof(dbContext));

            ResourceRepository = new Repository<Resource>(_dbContext);
            ResourcePriceRepository = new Repository<ResourcePrice>(_dbContext);
            MethaneRepository = new Repository<Methane>(_dbContext);
            RegionRepository = new Repository<Region>(_dbContext);
            SubscriptionRepository = new Repository<Subscription>(_dbContext);
        }

        public IRepository<Resource> ResourceRepository { get; }

        public IRepository<ResourcePrice> ResourcePriceRepository { get; }

        public IRepository<Methane> MethaneRepository { get; }

        public IRepository<Region> RegionRepository { get; }

        public IRepository<Subscription> SubscriptionRepository { get; }

        public async Task SaveChangesAsync(CancellationToken cancellationToken =
default)
        {
            await _dbContext.SaveChangesAsync(cancellationToken);
        }

        public void Dispose()
        {
            _dbContext.Dispose();
        }

        public async ValueTask DisposeAsync()
        {
            await _dbContext.DisposeAsync();
        }
    }
}
```

DataProvider.cs

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace Sulfur.DataAccess.Models
{
    public class DataProvider
    {
        public int Id { get; set; }

        [NotMapped]
        public Enums.DataProvider EnumId
        {
            get => (Enums.DataProvider)Id;
            set => Id = (int)value;
        }

        [MaxLength(50)]
        public string Name { get; set; }

        [MaxLength(50)]
```

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

Methane.cs

```
using Sulfur.DataAccess.Abstraction;

namespace Sulfur.DataAccess.Models
{
    public class Methane : IModel
    {
        public int Id { get; set; }

        public double Quality { get; set; }

        public string Link { get; set; }

        public int ResourcePriceId { get; set; }

        public ResourcePrice ResourcePrice { get; set; }

        public int DataProviderId { get; set; }

        public DataProvider DataProvider { get; set; }
    }
}
```

Region.cs

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using Sulfur.DataAccess.Abstraction;

namespace Sulfur.DataAccess.Models
{
    public class Region : IModel
    {
        public int Id { get; set; }

        [MaxLength(20)]
        public string Name { get; set; }

        [MaxLength(50)]
        public string Address { get; set; }

        public double Temperature { get; set; }

        public double Latitude { get; set; }

        public double Longitude { get; set; }

        public int SeaLevel { get; set; }

        [MaxLength(250)]
        public string Link { get; set; }

        public int DataProviderId { get; set; }

        public DataProvider DataProvider { get; set; }

        public List<ResourcePrice> ResourcePrices { get; set; }
    }
```

```
}
```

Resource.cs

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using Sulfur.DataAccess.Abstraction;

namespace Sulfur.DataAccess.Models
{
    public class Resource : IModel
    {
        public int Id { get; set; }

        [NotMapped]
        public Enums.Resource EnumId
        {
            get => (Enums.Resource)Id;
            set => Id = (int)value;
        }

        [MaxLength(20)]
        public string Name { get; set; }

        public List<ResourcePrice> ResourcePrices { get; set; }
    }
}
```

ResourcePrice.cs

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using Sulfur.DataAccess.Abstraction;

namespace Sulfur.DataAccess.Models
{
    public class ResourcePrice : IModel
    {
        public int Id { get; set; }

        public int ResourceId { get; set; }

        public Resource Resource { get; set; }

        public int DataProviderId { get; set; }

        public DataProvider DataProvider { get; set; }

        [MaxLength(250)]
        public string Link { get; set; }

        [Column(TypeName = "decimal(16, 2)")]
        public decimal Price { get; set; }

        public int RegionId { get; set; }

        public Region Region { get; set; }
    }
}
```

Subscription.cs

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

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Linq;
using Sulfur.DataAccess.Abstraction;

namespace Sulfur.DataAccess.Models
{
    public class Subscription : IModel
    {
        public int Id { get; set; }

        [MaxLength(250)]
        public string Endpoint { get; set; }

        /// <summary>
        /// Property for storing <see cref="IDictionary{TKey, TValue}" /> value
in database
        /// </summary>
        /// <remarks>
        /// Try to avoid using this property directly.
        /// Better choice is to use <see cref="Keys"/>
        /// </remarks>
        public List<SubscriptionKeysDictionaryModel> KeysDictionaryModels {
private get; set; }

        [NotMapped]
        public IDictionary<string, string> Keys {
            get => KeysDictionaryModels.ToDictionary(x => x.Key, x => x.Value);

            set => KeysDictionaryModels = value
                .Select(x => new SubscriptionKeysDictionaryModel
                {
                    Key = x.Key,
                    Value = x.Value,
                    SubscriptionId = Id,
                    Subscription = this
                })
                .ToList();

        }
    }
}
```

SubscriptionKeysDictionaryModel.cs

```
using System.ComponentModel.DataAnnotations;

namespace Sulfur.DataAccess.Models
{
    public class SubscriptionKeysDictionaryModel
    {
        [MaxLength(50)]
        public string Key { get; set; }

        public int SubscriptionId { get; set; }

        public Subscription Subscription { get; set; }

        [MaxLength(250)]
        public string Value { get; set; }
    }
}
```

ДОДАТОК Д

Сирцевий код UI

app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule, PreloadAllModules } from '@angular/router';

const routes: Routes = [
  {
    path: '',
    loadChildren: () => import('./features/main/main.module').then(m =>
m.MainModule)
  },
  { path: '**', redirectTo: '' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes, {preloadingStrategy:
PreloadAllModules})],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

app.component.html

```
<app-spinner></app-spinner>
<router-outlet></router-outlet>
```

main-routing.module.ts

```
import { Routes, RouterModule } from '@angular/router';
import { NgModule } from '@angular/core';
import { MainComponent } from './containers/main/main.component';

const routes: Routes = [
  {
    path: '',
    component: MainComponent,
    children: [
      {
        path: '',
        loadChildren: () => import('./features/home/home.module').then(x =>
x.HomeModule)
      },
      {
        path: 'subscription',
        loadChildren: () =>
import('./features/subscription/subscription.module').then(x =>
x.SubscriptionModule)
      }
    ]
  },
];

@NgModule({
  imports: [ RouterModule.forChild(routes) ],
  exports: [ RouterModule ]
})
export class MainRoutingModule { }
```

Продовження додатку Д

main.component.html

```
<app-header [isBlurred]="isBlurred" [routes]="mainRoutes">
  <router-outlet></router-outlet>
</app-header>
```

main.component.ts

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

import { SpinnerService } from '@core/services/spinner/spinner.service';
import { MainRoutes } from '../../constants/routes';

@Component({
  selector: 'app-main',
  templateUrl: './main.component.html',
  styleUrls: ['./main.component.scss']
})
export class MainComponent implements OnInit {
  isBlurred: boolean;
  mainRoutes = MainRoutes;

  constructor(private spinnerService: SpinnerService) { }

  ngOnInit() {
    this.spinnerService.spinnerStatus.subscribe(value => {
      this.isBlurred = value;
    });
  }
}
```

services.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import {MatSnackBarModule} from '@angular/material/snack-bar';

import { SpinnerService } from './spinner/spinner.service';
import { CommonSnackBarService } from './snack-bar/common-snack-bar.service';

@NgModule({
  imports: [
    CommonModule,
    MatSnackBarModule
  ],
  providers: [
    SpinnerService,
    CommonSnackBarService
  ]
})
export class ServicesModule { }
```

spinner.service.ts

```
import { Injectable } from '@angular/core';
import { Subject } from 'rxjs';

@Injectable()
export class SpinnerService {
  spinnerStatus: Subject<boolean> = new Subject<boolean>();

  startSpinner(): void {
    this.spinnerStatus.next(true);
  }
```

```
  stopSpinner(): void {
    this.spinnerStatus.next(false);
  }
}
```

default.interceptor.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { HttpClientModule, HTTP_INTERCEPTORS } from '@angular/common/http';
import { DefaultInterceptor } from './default.interceptor';

@NgModule({
  imports: [
    CommonModule,
    HttpClientModule
  ],
  exports: [
    HttpClientModule
  ],
  providers: [
    {
      provide: HTTP_INTERCEPTORS,
      useClass: DefaultInterceptor,
      multi: true
    }
  ]
})
export class InterceptorsModule { }
```

home-routing.module.ts

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

import { HomeComponent } from './containers/home.component';

const routes: Routes = [
  {
    path: '',
    component: HomeComponent
  }
];

@NgModule({
  imports: [
    RouterModule.forChild(routes)
  ],
  exports: [ RouterModule ]
})
export class HomeRoutingModule { }
```

home.component.html

```
<div class="spacer"></div>
<div fxLayout="column" fxLayoutAlign="space-around none">
  <div fxLayout="row" fxLayoutAlign="start center">
    <app-drop-down
      [options]="options"
      [(selectedValue)]="selectedRegion"
      (selectedValueChange)="selectRegion($event)"
      label="Region">
    </app-drop-down>
  </div>
```

```
  <div fxLayout="row" fxLayoutAlign="space-around center">
    <app-table
      fxFlexFill
      fxFlex="95%"
      [columns]="columns"
      [elements]="elements"
      (rowSelected)="select($event)">
    </app-table>
  </div>
</div>
```

home.component.ts

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

import { DropDownOption } from '@share/models/drop-down-option';
import { TableColumn } from '@share/models/table-column';
import { ResourceService } from '../services/resource/resource.service';
import { Resource } from '../models/resource';
import { ModalProvider } from '../services/modal-provider/modal-
provider.service';
import { ResourceModalData } from '../models/resource-modal-info';

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.scss']
})
export class HomeComponent implements OnInit {

  constructor(
    private homeService: ResourceService,
    private modalProvider: ModalProvider) { }

  options: DropDownOption[] = [
    { id: 1, name: 'Ivano-Frankivsk' }, // TODO: Get from backend
    { id: 1, name: 'Kyiv' }
  ];
  selectedRegion = this.options[0].id;
  columns: TableColumn[] = [
    { propertyName: 'id', name: 'Number' },
    { propertyName: 'name', name: 'Name' },
    { propertyName: 'price', name: 'Price' },
    { propertyName: 'energy', name: 'Energy' },
    { propertyName: 'energyCost', name: 'Cost of energy' },
  ];
  elements: Resource[] = [];

  ngOnInit() {
    this.updateElements(this.selectedRegion);
  }

  select(row: Resource): void {
    const data: ResourceModalData = {
      regionId: this.selectedRegion,
      resource: row
    };

    this.modalProvider.openMethaneDialog(data);
  }

  selectRegion(regionId: number): void {
```

```
    this.updateElements(regionId);
  }
  private updateElements(regionId: number): void {
    this.homeService
      .getResources(regionId)
      .subscribe(
        value => {
          this.elements = value;
        },
        console.error);
  }
}
```

methane-modal.component.html

```
<div mat-dialog-title>
  <p>{{data.resource.name}}'s input parameters</p>
</div>
<div mat-dialog-content>
  <form [formGroup]="methaneForm">
    <mat-form-field>
      <input matInput required matTooltip="Info about the action" type="number"
placeholder="Temperature" formControlName="temp">
    </mat-form-field>
    <mat-form-field>
      <input matInput required type="number" placeholder="Methane quality"
formControlName="quality">
    </mat-form-field>
    <mat-form-field>
      <input matInput required type="number" placeholder="Price"
formControlName="price">
    </mat-form-field>
    <mat-form-field>
      <input matInput required type="number" placeholder="Pipe diameter"
formControlName="diameter">
    </mat-form-field>
    <mat-form-field>
      <input matInput required type="number" placeholder="Pipe length"
formControlName="length">
    </mat-form-field>
    <mat-form-field>
      <input matInput required type="number" placeholder="Area"
formControlName="area">
    </mat-form-field>
    <mat-form-field>
      <input matInput required type="number" placeholder="Counter wastes"
formControlName="waste">
    </mat-form-field>
    <mat-form-field>
      <input matInput required type="number" placeholder="Sea level"
formControlName="level">
    </mat-form-field>
  </form>
</div>
<div mat-dialog-actions>
  <button mat-button (click)="cancel()">Cancel</button>
  <button
    mat-raised-button
    color="primary"
    [disabled]="!methaneForm.valid"
    (click)="save()">
    Save
```

```
  </button>
</div>
```

methane-modal.component.ts

```
import { Component, OnInit, Inject } from '@angular/core';
import { MatDialogRef, MAT_DIALOG_DATA } from '@angular/material/dialog';
import { FormBuilder, Validators } from '@angular/forms';

import { ResourceService } from '../../services/resource/resource.service';
import { MethaneParams } from '../../models/methane-params';
import { ResourceModalData } from '../../models/resource-modal-info';
import { DefaultDataService } from '../../services/default-data/default-
data.service';

@Component({
  selector: 'app-methane-modal',
  templateUrl: './methane-modal.component.html',
  styleUrls: ['./methane-modal.component.scss']
})
export class MethaneModalComponent implements OnInit {
  methaneForm = this.fb.group({
    temp: ['', Validators.required],
    quality: ['', Validators.required],
    price: ['', Validators.required],
    diameter: ['', Validators.required],
    length: ['', Validators.required],
    area: ['', Validators.required],
    waste: ['', Validators.required],
    level: ['', Validators.required]
  });

  constructor(
    public dialogRef: MatDialogRef<MethaneModalComponent>,
    @Inject(MAT_DIALOG_DATA) public data: ResourceModalData,
    private fb: FormBuilder,
    private resourceService: ResourceService,
    private defaultDataService: DefaultDataService) { }

  ngOnInit() {
    this.initForms();
  }

  save(): void {
    const methaneParams = this.methaneForm.value as MethaneParams;
    methaneParams.regionId = this.data.regionId;
    this.resourceService
      .getResource(this.data.resource.id, methaneParams)
      .subscribe(value => {
        this.data.resource.name = value.name;
        this.data.resource.price = value.price;
        this.data.resource.energy = value.energy;
        this.data.resource.energyCost = value.energyCost;
      });

    this.dialogRef.close();
  }

  cancel(): void {
    this.dialogRef.close();
  }

  private initForms(): void {
```

```
    this.defaultDataService
      .getDefaultParams(this.data.resource.id, this.data.regionId)
      .subscribe(value => {
        this.methaneForm.setValue({
          temp: value.environmentTemperature,
          quality: value.methaneQuality,
          price: value.methanePrice,
          diameter: value.pipeDiameter,
          length: value.pipeLength,
          area: value.area,
          waste: value.wasteAmount,
          level: value.seaLevel
        });
      },
      console.error);
  }
}
```

methane-params.ts

```
export interface MethaneParams {
  regionId: number;
  temp: number;
  quality: number;
  price: number;
  diameter: number;
  length: number;
  area: number;
  waste: number;
  level: number;
}
```

resource-modal-info.ts

```
import { Resource } from './resource';

export interface ResourceModalData {
  regionId: number;
  resource: Resource;
}
```

resource.ts

```
export interface Resource {
  id: number;
  name: string;
  price: number;
  energy: number;
  energyCost: number;
}
```

default-data.service.ts

```
import { Injectable } from '@angular/core';
import { HttpParams, HttpClient } from '@angular/common/http';

import { Observable } from 'rxjs';

import { MethaneParams } from '../../models/methane-params';

@Injectable()
export class DefaultDataService {

  constructor(private http: HttpClient) { }
```

```
  getDefaultParams(resourceId: number, regionId: number): Observable<any> {
    const params = new HttpParams()
      .set('regionId', regionId.toString());

    const result = this.http
      .get<any>(`calculation/default-data/${resourceId}`, { params });

    return result;
  }
}
```

modal-provider.service.ts

```
import { Injectable } from '@angular/core';
import { MatDialog, MatDialogRef } from '@angular/material/dialog';

import { MethaneModalComponent } from '../../modals/methane-modal/methane-
modal.component';
import { ResourceModalData } from '../../models/resource-modal-info';

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

  constructor(private dialog: MatDialog) { }

  openMethaneDialog(data: ResourceModalData):
MatDialogRef<MethaneModalComponent> {
    const dialogRef = this.dialog.open(MethaneModalComponent, {
      width: '90%',
      data
    });

    return dialogRef;
  }
}
```

resource.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpParams } from '@angular/common/http';
import { Observable, forkJoin, from, of } from 'rxjs';
import { Resource } from '../../models/resource';
import { tap, finalize, concatAll, map, switchMap } from 'rxjs/operators';
import { SpinnerService } from '@core/services/spinner/spinner.service';
import { MethaneParams } from '../../models/methane-params';

@Injectable()
export class ResourceService {

  constructor(private http: HttpClient) { }

  getResources(regionId: number): Observable<Resource[]> {
    const params = new HttpParams()
      .set('regionId', regionId.toString());

    const result = this.http
      .get<Resource[]>('calculation/resources', { params });

    return result;
  }
```

Продовження додатку Д

```
  getResource(resourceId: number, methaneParams: MethaneParams):
Observable<Resource> {
    const params = new HttpParams()
      .set('regionId', methaneParams.regionId.toString())
      .set('temp', methaneParams.temp.toString())
      .set('quality', methaneParams.quality.toString())
      .set('price', methaneParams.price.toString())
      .set('diameter', methaneParams.diameter.toString())
      .set('length', methaneParams.length.toString())
      .set('area', methaneParams.area.toString())
      .set('waste', methaneParams.waste.toString())
      .set('level', methaneParams.level.toString());

    const result = this.http
      .get<Resource>(`calculation/resources/${resourceId}`, { params });

    return result;
  }
}
```

subscription-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

import { SubscriptionComponent } from './containers/subscription.component';

const routes: Routes = [
  {
    path: '',
    component: SubscriptionComponent
  }
];

@NgModule({
  imports: [
    RouterModule.forChild(routes)
  ],
  exports: [ RouterModule ]
})
export class SubscriptionRoutingModule { }
```

subscription.component.html

```
<button
  mat-raised-button
  [attr.disabled]="operationName == null"
  (click)="operation()">
  {{ operationName || 'Subscribe' }}
</button>
```

subscription.component.ts

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { SwPush } from '@angular/service-worker';

import { Subscription, from } from 'rxjs';
import { switchMap } from 'rxjs/operators';

import { CommonSnackBarService } from '@core/services/snack-bar/common-snack-
bar.service';
import { SubscriptionService } from '../services/subscription.service';
import { Messages } from '../constants/messages';
```

Продовження додатку Д

```
@Component({
  selector: 'app-subscription',
  templateUrl: './subscription.component.html',
  styleUrls: ['./subscription.component.scss']
})
export class SubscriptionComponent implements OnInit, OnDestroy {
  private swPush$: Subscription;
  private subscription: PushSubscription;

  operationName: string;

  constructor(private swPush: SwPush,
              private subscriptionService: SubscriptionService,
              private commonSnackBarService: CommonSnackBarService) { }

  ngOnInit() {
    this.swPush$ = this.swPush.subscription.subscribe(value => {
      this.subscription = value;
      this.operationName = (this.subscription === null) ? 'Subscribe' :
'Unsubscribe';
    });
  }

  operation(): void {
    (this.subscription === null) ? this.subscribe() :
this.unsubscribe(this.subscription.endpoint);
  }

  ngOnDestroy(): void {
    this.swPush$.unsubscribe();
  }

  private subscribe(): void {
    const requestSubscription = (publicKey: string) =>
this.swPush.requestSubscription({
      serverPublicKey: publicKey
    });

    this.subscriptionService
      .getPublicKey()
      .pipe(
        switchMap(value => from(requestSubscription(value))),
        switchMap(value => this.subscriptionService.addSubscription(value))
      )
      .subscribe(() =>
this.commonSnackBarService.openSnackBar(Messages.subscribe), console.error);
  }

  private unsubscribe(endpoint: string): void {
    from(this.swPush.unsubscribe())
      .pipe(
        switchMap(() =>
this.subscriptionService.deleteSubscription(encodeURIComponent(endpoint)))
      )
      .subscribe(() =>
this.commonSnackBarService.openSnackBar(Messages.unsubscribe), console.error);
  }
}
```

subscription.service.ts

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

Продовження додатку Д

```
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable()
export class SubscriptionService {

  constructor(private httpClient: HttpClient) { }

  getPublicKey(): Observable<string> {
    return this.httpClient.get('notification/public-key', { responseType: 'text'
});
  }

  addSubscription(subscription: PushSubscription): Observable<void> {
    return this.httpClient.post<void>('notification/subscription',
subscription);
  }

  deleteSubscription(endpoint: string): Observable<void> {
    return
this.httpClient.delete<void>(`notification/subscription/${endpoint}`);
  }
}
```