ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені І.І.МЕЧНИКОВА
(повне найменування вищого навчального закладу)

Факультет математики, фізики та інформаційних технологій
(повне найменування інституту, назва факультету (відділення))

Кафедра математичного забезпечення комп'ютерних систем
(повна назва кафедри (предметної, циклової комісії))

Кваліфікаційна робота

на здобуття ступеня вищої освіти «бакалавр»
(освітньо-кваліфікаційний рівень)

на тему Побудова універсальних інтерфейсів взаємодії інформаційних систем при застосуванні кросплатформних програмних компонентів.
Creation of universal interfaces for the interaction of information systems when using cross-platform software components.

Виконав: студент денної форми навчання
спеціальності 126 – Інформаційні системи та технології .
(шифр і назва напряму підготовки, спеціальності)
Освітня програма «Інформаційні системи та технології»
(назва освітньої програми)

Сергатий Євген Юрійович
(прізвище, ім'я, по-батькові)

Керівник Ст. викл. Максимов Олександр Семенович
(науковий ступінь, вчене звання, прізвище та ініціали)

Рецензент Доц. Антоненко О.С
(науковий ступінь, вчене звання, прізвище та ініціали)

Рекомендовано до захисту:
Протокол засідання кафедри
№___ від «___»_____ 2023 р.
Завідувач кафедри
_____ _Євгеній МАЛАХОВ_
(підпис) (ім'я, прізвище)

Захищено на засіданні ЕК № ___
протокол № __ від «_ _» ____ 2023 р.
Оцінка _____/_____/_____
(за національною шкалою, шкалою ECTS, бали)
Голова ЕК
_____ _Володимир ВИЧУЖАНІН_
(підпис) (ім'я, прізвище)

Одеса - 2023

АНОТАЦІЯ

У дипломній роботі розробляється тема «Побудова універсальних інтерфейсів взаємодії інформаційних систем при застосуванні кросплатформних програмних компонентів».

Дипломна робота бакалавра містить 78 сторінок, 13 рисунків, 3 таблиці, 22 посилань та 2 додатки.

Метою роботи є розробка моделей та інструментальних засобів комбінованого методу інтеграції різнорідних інформаційних систем та його застосування для підвищення ефективності реалізації проблемно-орієнтованих систем та прикладного програмного забезпечення.

Робота складається з 3 розділів.

У першому розглядається такі теми, як: кросплатформні програмні компоненти, інтерфейси взаємодії інформаційних систем та аналіз технологій інтеграції застосунків. У яких аналізуються проблеми та існуючі різновиди їх вирішення.

У другому розділі розглядається поясненно вибір програмних інструментів, архітектурних рішень та функціональності систем.

У третьому розділі розглядається програмна структура систем та їх детальна реалізація. Також розглянуті перспективи розвитку системи.

ABSTRACT

The name of the thesis is "Creation of universal interfaces for the interaction of information systems when using cross-platform software components".

The bachelor's thesis contains 78 pages, 13 figures, 3 tables, 22 references and 2 additions.

The purpose of the work is to develop models and tools for a combined method of integrating heterogeneous information systems and applying it to improve the efficiency of implementing problem-oriented systems and application software.

The paper consists of 3 chapters.

The first covers such topics as cross-platform software components, interfaces for information systems interaction, and analysis of application integration technologies. The latter analyzes the problems and the existing variety of ways to solve them.

The second section discusses the choice of software tools, architectural solutions and system functionality.

The third section discusses the software structure of the systems and their detailed implementation. The prospects for system development are also discussed.

ЗМІСТ

СПИСОК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ І ТЕРМІНІВ

ГОС - гетерогенні обчислювальні середовища;

EAI - Enterprise Application Integration;

ESB - Enterprise Service Bus;

IOT - Internet of Things;

JSON - JavaScript Object Notation;

MOM - Message-Oriented Middleware;

REST - Representational State Transfer;

SOAP - Simple Object Access Protocol;

HTTP - HyperText Transfer Protocol;

URL - Uniform Resource Locator;

XML - Extensible Markup Language;

ВСТУП

Актуальність роботи. В даний час розвиток інформаційних систем і програмних комплексів відбувається в бік ускладнення їх функціональності. Це пов'язано з тим, що за останнє десятиліття зростання продуктивності персональних комп'ютерів, робочих станцій і сучасних мобільних пристроїв, пропускної здатності мережевих концентраторів і каналів зв'язку якісно змінили ситуацію в обчислювальній техніці і сферах її застосування [1].

Відомо, що основні дослідження і розробки протягом перших трьох десятиліть комп'ютерної ери були спрямовані на розвиток апаратних комп'ютерних засобів. Це призвело до різкого збільшення продуктивності комп'ютерів при значному зниженні їх вартості. Однак основним завданням кінця минулого століття і початку XXI століття стало вдосконалення методів підвищення ефективності, надійності і якості розроблюваних інформаційних систем і комплексів, можливості яких цілком визначаються створюваним програмним забезпеченням.

Споживачі продукції інформаційної індустрії бажають мати програмне забезпечення, ще краще пристосоване для їх потреб, а це, в свою чергу, призводить до ускладнення програмних продуктів. Часто, бажання отримати більш потужні і складні програмні рішення не поєднується з тим, яким чином вони розробляються. Сучасні комплексні інформаційні системи відрізняються великою розгалуженістю технологічних підсистем, великим числом і різнотипністю обладнання, програмно-апаратних платформ, складністю алгоритмів управління.

У зв'язку модернізацією бізнес-застосунків, а також необхідністю збереження спадкоємності існуючих конструкторсько-технологічних розробок та інформаційних систем, що забезпечують їх працездатність, особливу увагу слід приділити вирішенню комплексу питань, пов'язаних зі збереженням раніше реалізованих програмних компонент (модулів) і оболонок при еволюційному розвитку інформаційної інфраструктури підприємства в цілому.

Виходячи з викладеного, важливість розробки ефективних методів універсальних інтерфейсів взаємодії як існуючих неоднорідних інформаційних середовищ і систем, так і знову розроблюваних набуває особливої актуальності.

Ступінь розробленості проблеми. До теперішнього часу вироблено безліч підходів до розробки інформаційних систем різного рівня і методів їх інтеграції. Так , авторами Коваленко О. Є. , Roco M. C., Bainbridge W. S. виконано аналіз технологій створення сучасних систем обробки інформації та показано, що на рівні програмних засобів міварний підхід значно розширює можливості сервісно-орієнтованої архітектури в частині інтеграції застосунків, інтелектуальної обробки даних та адаптивного розвитку інформаційних систем в цілому.

У роботі Коваленко О. Є. [2] розглядається забезпечення якості інтегрованих інформаційних систем як основне завдання, що вирішується в ході об'єднання локальних інформаційних ресурсів в рамках великомасштабних інтегрованих інформаційних систем.

Розроблені і застосовувані методи інших вищевказаних авторів знайшли своє втілення лише в окремо взятих інформаційних системах і завданнях і не є універсальними.

Існують підходи, які відрізняються методами і стандартами побудови, застосовуваними рішеннями і підтримують їх апаратними і програмними засобами. Інтеграція здійснюється на рівні баз даних за рахунок можливості обміну даними з єдиною БД.

В якості основної ланки інтеграції ІС пропонується використання електронного документа, інтерпретація і обробка якого прив'язана до ОС сімейства Windows. Крім того, пропоновані методи інтеграції інформаційних систем не враховують різнорідність програмно-апаратних платформ, засновані на єдиній шині взаємодії, а також XML-орієнтованої взаємодії (роботи Roco M. C., Bainbridge W. S..), і не мають уніфікованих інтерфейсів

Даючи, безумовно, позитивну оцінку результатам роботи всіх перерахованих вище дослідників, відзначимо наявність ще невирішених

проблем, серед яких-вибір ефективної моделі взаємодії інформаційних систем; розробка уніфікованого методу інтеграції неоднорідних інформаційних систем, що відрізняється від існуючих інваріантністю програмно-апаратних платформ і типів даних.

З викладеного можна зробити висновок, що **актуальним завданням** є розробка уніфікованого методу інтеграції неоднорідних інформаційних середовищ, що поєднує в собі переваги різних підходів і підвищує надійність і ефективність розробки нових інформаційних систем, а також знижує накладні витрати при експлуатації.

Об'єктом дослідження бакалаврської роботи є інформаційні системи і комплекси як сукупність прикладних програмних застосунків.

Предметом дослідження бакалаврської роботи є алгоритми, методи і програмні засоби інтеграції різнорідних інформаційних систем.

Метою роботи є розробка моделей та інструментальних засобів комбінованого методу інтеграції різнорідних інформаційних систем та його застосування для підвищення ефективності реалізації проблемно-орієнтованих систем та прикладного програмного забезпечення.

У роботі поставлені і вирішені наступні завдання:

1) виявити методи та дослідити підходи до інтеграції неоднорідних інформаційних середовищ;

2) розробити метод, що дозволяє уніфікувати інтерфейси взаємодії інформаційних систем при застосуванні кроссплатформенних програмних компонент;

3) розробити уніфікований метод інтеграції неоднорідних інформаційних систем на базі застосування кроссплатформенних програмних компонент;

4) розробити методику оцінки ефективності інтеграції та розробки інформаційних систем;

5) виконати практичну інтеграцію різнорідних інформаційних потоків у середовищі прикладної інформаційної системи на основі розробленого методу інтеграції з метою підтвердження теоретичних положень, висунутих у роботі.

1 КРОСПЛАТФОРМНІ КОМПОНЕНТИ ЯК ІНСТРУМЕНТ СИСТЕМНОЇ ІНТЕГРАЦІЇ

У цьому розділі розглядається такі теми, як: кросплатформні програмні компоненти, інтерфейси взаємодії інформаційних систем та аналіз технологій інтеграції застосунків. У яких аналізуються проблеми та існуючий різновиді їх вирішення.

1.1 Проблема взаємодії між різнорідними кросплатформеними системами

Проблема взаємодії між різнорідними кросплатформеними системами є актуальною в сучасному інформаційному середовищі. У зв'язку з розвитком технологій та поширенням різноманітних платформ і пристроїв, виникає потреба в ефективному об'єднанні цих систем для забезпечення гладкого взаємодії та обміну даними. Рознорідність систем може виникати з різних причин, таких як використання різних мов програмування, різних архітектур чи навіть різних операційних систем. Це створює перешкоди для безперешкодної комунікації та обміну даними між цими системами.

Конфігурування підключення для кожного окремого сервісу є дуже трудомістким завданням, особливо коли маємо справу з великою кількістю різнорідних систем. Це завдання вимагає знання специфічних деталей кожного сервісу, таких як його протоколи, адреси, порти та інші параметри підключення. При такому підході адміністрування та підтримка великої кількості конфігурацій може стати важким та часозатратним процесом.

Проте, нове рішення комунікації може значно спростити цей процес. Якщо використовувати універсальний інтерфейс, який уніфікує спілкування між різнорідними системами, користувачам більше не потрібно буде розглядати окремі конфігурації для кожного сервісу. Замість цього, можна буде налаштувати загальні параметри підключення для цього універсального

інтерфейсу. Таке рішення надасть можливість здійснювати зв'язок з будь-яким сервісом, який підтримує цей універсальний інтерфейс, використовуючи однаковий набір налаштувань. Такий підхід сприятиме ефективному об'єднанню різнорідних систем та спрощенню процесу комунікації. Він забезпечить зручну та надійну взаємодію між цими системами, зменшуючи залежність від специфічних конфігурацій.

1.2 Кросплатформні програмні компоненти

Кросплатформні програмні компоненти це фрагменти коду, які можна використовувати на різних операційних системах або платформах без значних змін. Ці компоненти розробляються з метою безперебійної роботи на різних платформах, як настільні комп'ютери, мобільні пристрої та веб-браузери.

Кросплатформні програмні компоненти широко використовуються для спрощення розробки програмного забезпечення, дозволяючи розробникам написати код один раз і розгорнути його на декількох платформах [6]. Це зменшує потребу в розробці для конкретної платформи і може заощадити час та ресурси. Крім того, кросплатформні компоненти можуть бути корисними в ситуаціях, коли програмне забезпечення потрібно швидко розгорнути на різних платформах, оскільки їх можна легко інтегрувати в існуючі системи.

Використання кросплатформних програмних компонентів має свої переваги та недоліки. Перш за все, вони дозволяють розробникам охопити ширшу аудиторію зі своїми додатками, роблячи їх доступними на різних платформах [7]. Друга перевага полягає в тому, що вони можуть допомогти скоротити витрати і час на розробку, дозволяючи розробникам написати код один раз і повторно використовувати його на різних платформах, замість того, щоб писати і підтримувати окремий код для кожної платформи.

Однак, з розробкою кросплатформних програмних компонентів пов'язані й певні проблеми. Однією з них є забезпечення коректної роботи компонентів на кожній платформі, оскільки різні платформи можуть мати

різні вимоги та обмеження. Інша проблема полягає в тому, щоб компоненти були ефективними та продуктивними на кожній платформі, оскільки продуктивність може суттєво відрізнятися на різних платформах.

Нарешті, при використанні кросплатформних програмних компонентів можуть виникнути юридичні та ліцензійні питання, особливо якщо компоненти є пропрієтарними або мають ліцензійні обмеження [6].

Історію розробки кросплатформних програмних компонентів можна простежити з кінця 1980-х - початку 1990-х років, коли концепція програмних компонентів та об'єктно-орієнтованого програмування почала набувати популярності [7]. У той час розробка програмного забезпечення часто була платформо-специфічною, коли додатки писалися для запуску на певній операційній системі або апаратній архітектурі.

Розробка кросплатформних програмних компонентів почалася як спосіб вирішення цієї проблеми, що дозволило розробникам створювати компоненти, які можна повторно використовувати на різних платформах і в різних додатках..

У наступні роки з'явилися інші крос-платформні технології створення програмних компонентів, зокрема JavaBeans, ActiveX та .NET Framework. Ці технології були розроблені для того, щоб полегшити розробникам створення та повторне використання програмних компонентів на різних платформах, а також для забезпечення сумісності між додатками та сервісами, що працюють на різних системах [8].

Отже, історія розробки кросплатформних програмних компонентів бере свій початок наприкінці 1980-х - на початку 1990-х років, коли розробники почали шукати шляхи створення багаторазових програмних компонентів, які можна використовувати на різних платформах і в різних додатках. З роками з'явився цілий ряд технологій для вирішення цього завдання, включаючи JavaBeans і .NET Framework. Сьогодні крос-платформні програмні компоненти залишаються невід'ємною частиною сучасної розробки програмного забезпечення, забезпечуючи інтероперабельність і портативність на широкому спектрі платформ і систем.

Існують різні типи кросплатформних програмних компонентів, такі як:

1) мови програмування: найпростіша форма крос-платформних програмних компонентів. Такі мови, як Java, Python і Ruby, розроблені як незалежні від платформи і можуть працювати на будь-якій платформі, на якій встановлено сумісне середовище виконання [9];

2) бібліотеки та фреймворки: набори попередньо написаного коду, які надають розробникам певні функціональні можливості. Крос-платформні бібліотеки та фреймворки дозволяють розробникам створювати додатки, які можуть працювати на різних платформах з мінімальними зусиллями [9];

3) веб-застосунки: програмні компоненти, які працюють у веб-браузерах і доступні через Інтернет. Ці додатки написані з використанням веб-технологій, таких як HTML, CSS і JavaScript, які не залежать від платформи. Оскільки вони працюють у веб-браузері, а не в операційній системі, веб-застосунки можна запускати на будь-якому пристрої з веб-браузером, незалежно від операційної системи [7];

4) віртуальні машини: програмний компонент, який створює віртуальне середовище в операційній системі, що дозволяє програмному забезпеченню працювати в цьому середовищі без змін [6]. Віртуальна машина може емулювати різні операційні системи, що дозволяє запускати програмне забезпечення, розроблене для однієї операційної системи, на іншій;

5) середовища виконання: програмний компонент, який надає стандартний набір бібліотек і сервісів, що дозволяють програмному забезпеченню працювати на різних операційних системах. Цей тип компонентів часто використовується для мов програмування, таких як .NET і Qt, які надають середовища виконання, що дозволяють додаткам, написаним на цих мовах, працювати на різних операційних системах [6];

6) крос-компіляція: процес компіляції програмного забезпечення на одній платформі для іншої платформи. Наприклад, розробник може скомпілювати застосунок на комп'ютері з Windows для платформи Linux. Цей тип крос-платформних програмних компонентів часто використовується для вбудованих систем, де програмне забезпечення компілюється для певної апаратної архітектури [9].

Нижче наведена зведена таблиця, яка включає переваги та недоліки кожного типу компонентів:

Таблиця 1.1 – Порівняння переваг та недоліків компонентів

Тип компонента	**Переваги**	**Недоліки**
Мови програмування	розробка на одній мові, запуск на різних платформах;	можлива нижча продуктивність порівняно з мовами для конкретної платформи.
Бібліотеки та фреймворки	кросплатформні бібліотеки та фреймворки надають набір готового коду, який дозволяє створювати кросплатформні додатки з мінімальними зусиллями; пропонують велику екосистему інструментів, бібліотек і фреймворків, які дозволяють розробникам створювати складні додатки; пропонують велику екосистему інструментів, бібліотек і фреймворків, які дозволяють розробникам створювати складні додатки.	вони можуть мати нижчу продуктивність порівняно з бібліотеками та фреймворками для конкретної платформи; може бракувати певних функцій, характерних для конкретної платформи; може бракувати певних функцій, характерних для конкретної платформи.

Продовження таблиці 1.1

Тип компонента	Переваги	Недоліки
Веб-застосунки	простота розгортання та оновлення; висока доступність і масштабованість; незалежність від платформи.	обмежений доступ до функцій пристрою; продуктивність може бути нижчою у порівнянні з нативними додатками; обмежені офлайн-можливості.
Віртуальні машини	не залежить від платформи; забезпечує ізоляцію та безпеку; легко розгортати та оновлювати.	накладні витрати на продуктивність через емуляцію; вимагає додаткових ресурсів для запуска віртуальної машини; обмежений доступ до апаратних функцій.
Середовища виконання	висока продуктивність; доступ до апаратних функцій; незалежність від платформи.	обмежений доступ до низькорівневих функцій операційної системи; може вимагати додаткових зусиль при розробці для забезпечення сумісності з різними платформами.
Крос-компіляція	висока продуктивність; доступ до апаратних можливостей; забезпечує високий рівень контролю над цільовою платформою.	вимагає значних зусиль при розробці для забезпечення сумісності з декількома платформами; обмежений доступ до низькорівневих функцій операційної системи; може вимагати додаткових ресурсів для запуска скомпільованого додатку.

Після аналізу таблиці можна зробити висновок, що кросплатформні програмні компоненти пропонують розробникам багато переваг.

Сьогодні кросплатформені програмні компоненти продовжують відігравати важливу роль у розробці програмного забезпечення, особливо в контексті веб та хмарних додатків. Такі технології, як RESTful веб-сервіси та API, дозволили розробникам створювати компоненти, до яких можуть мати доступ додатки, що працюють на будь-якій платформі або пристрої з підключенням до Інтернету.

1.3 Аналіз та вирішення проблем створення універсальних інтерфейсів для інформаційних систем

Інтерфейс - це зв'язок між двома або більше системами, який дозволяє їм спілкуватися та обмінюватися даними. У контексті інформаційних систем інтерфейси необхідні для обміну даними між різними компонентами системи, такими як бази даних, програми та користувачі [10].

Інтерфейси взаємодії інформаційних систем відносяться до методів і стандартів, які полегшують комунікацію між різними комп'ютерними системами і додатками. Вони уможливлюють обмін даними та функціональністю між різними системами, дозволяючи їм безперешкодно працювати разом.

Існують різні типи інтерфейсів, що використовуються в інформаційних системах, зокрема:

– інтерфейси користувача: візуальні та інтерактивні компоненти програми, які дозволяють користувачам взаємодіяти з системою. Інтерфейси користувача можуть мати різні форми, включаючи графічні інтерфейси користувача, інтерфейси командного рядка та веб-інтерфейси [10];

– інтерфейси прикладного програмування: набори протоколів, процедур та інструментів, які дозволяють різним програмним додаткам взаємодіяти один з одним. API визначають, як програмні компоненти повинні

спілкуватися та обмінюватися даними, що полегшує інтеграцію різних додатків і систем [11];

– веб-сервіси: тип API, який використовує веб-протоколи для забезпечення зв'язку між різними системами через Інтернет. Веб-сервіси зазвичай використовують такі стандарти, як SOAP (простий протокол доступу до об'єктів) і REST (передача представницького стану) для обміну даними між системами [11];

– проміжне програмне забезпечення, орієнтоване на повідомлення (MOM): тип проміжного програмного забезпечення, який дозволяє різним програмам обмінюватися повідомленнями один з одним. MOM забезпечує систему обміну повідомленнями, яка управляє маршрутизацією і доставкою повідомлень між додатками, дозволяючи їм спілкуватися асинхронно [12];

– корпоративна сервісна шина (ESB): тип проміжного програмного забезпечення, який забезпечує центральний вузол для інтеграції різних додатків і сервісів в межах підприємства. ESB виконують ряд функцій, включаючи маршрутизацію повідомлень між різними додатками, перетворення даних між різними форматами, а також управління безпекою і надійністю зв'язку [13].

Інтерфейси взаємодії інформаційних систем відіграють вирішальну роль у забезпеченні інтеграції різних систем і додатків. Забезпечуючи стандарти і протоколи для комунікації та обміну даними, вони дозволяють різним системам безперешкодно працювати разом, підвищуючи ефективність, зменшуючи дублювання зусиль і збільшуючи продуктивність.

1.4 Аналіз предметної області та технологій інтеграції застосунків

Технологія інтеграції застосунків є важливою складовою універсального інтерфейсу, що дозволяє забезпечити зв'язок між різними додатками та платформами. Її основна мета полягає в об'єднанні різних програмних рішень для ефективної взаємодії та спільного використання ресурсів.

Ця технологія надає можливість забезпечити сумісність та зручність управління різноманітними додатками та платформами, спрощуючи процеси обміну даними та комунікації між ними. Шляхом інтеграції додатків, їх функціональні можливості стають доступними з єдиного інтерфейсу, що сприяє зручності роботи користувачів і покращує продуктивність.

Одним із ключових аспектів технології інтеграції додатків є забезпечення взаємодії між різними додатками та платформами. Це досягається шляхом розробки спеціальних протоколів та стандартів комунікації, які дозволяють передавати дані та виконувати команди між різними системами. Таким чином, різні додатки можуть обмінюватися інформацією та взаємодіяти між собою без проблем.

Крім того, технологія інтеграції додатків надає можливість інтегрувати дані з різних джерел у єдину базу даних. Це дозволяє зберігати та управляти інформацією з різних додатків у зручному та структурованому форматі.

Інтеграція даних сприяє усуненню дублювання інформації, полегшує процеси аналізу та забезпечує єдинообразність даних для використання в різних контекстах.

Таким чином, технологія інтеграції додатків виступає важливим інструментом для забезпечення єдиності, сумісності та ефективної взаємодії різних програмних рішень.

Аналіз предметної області. На тему технології інтеграції додатків існує безліч літератури і ресурсів для навчання. Дослідники та фахівці активно вивчають цю область, розробляють нові методи та підходи до інтеграції додатків для поліпшення їх функціональності та продуктивності. Крім того, наявні різноманітні програмні засоби, які дозволяють будувати та тестувати інтегровані додатки.

Далі наведені типові технології, що використовуються для цих цілей. Вони допомагають реалізувати інтеграцію додатків шляхом розробки спеціальних інструментів, протоколів та стандартів, які сприяють взаємодії та обміну даними між різними системами та платформами.

Аналіз технології брокеру повідомлень. Брокерська технологія - це архітектура розподілених обчислень, яка існує з 1990-х років. Концепція брокера була вперше представлена компанією IBM у вигляді системи обміну повідомленнями MQSeries, яка була розроблена для забезпечення надійної асинхронної черги повідомлень між додатками. З того часу розроблено багато різних систем-брокерів, включаючи Apache ActiveMQ, RabbitMQ та Microsoft Message Queuing (MSMQ). Зараз технологія брокерів широко використовується в додатках корпоративного рівня і стала важливим компонентом багатьох архітектур розподілених обчислень [14].

В архітектурі на основі брокерів є центральний компонент, який називається брокер, що виконує роль черги повідомлень для різних додатків. Додатки можуть надсилати повідомлення брокеру, який потім перенаправляє їх відповідному одержувачу. Брокер виступає посередником між відправником і одержувачем, гарантуючи, що повідомлення будуть доставлені надійно і в правильному порядку [14].

Основний робочий процес в архітектурі на основі брокера виглядає наступним чином (див. рис. 3.1):

– застосунок-відправник створює повідомлення і надсилає його брокеру;

– брокер отримує повідомлення і ставить його в чергу;

– застосунок-отримувач отримує повідомлення з черги;

– застосунок-отримувач обробляє повідомлення і надсилає відповідь відправнику, яка також може бути направлена через брокера.

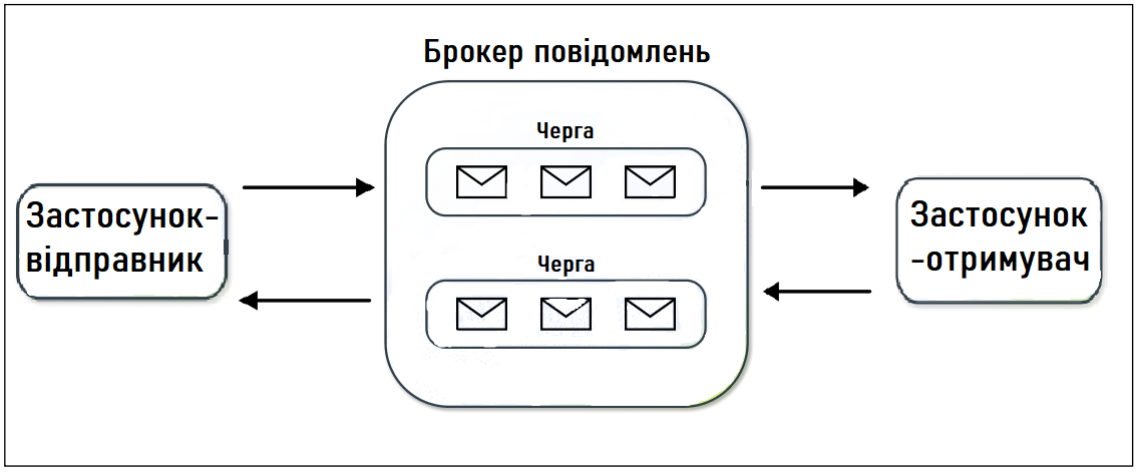

Рисунок 3.1 – Схема брокеру повідомлень

Однією з ключових переваг цієї архітектури є те, що вона дозволяє додаткам спілкуватися асинхронно, тобто їм не потрібно бути постійно підключеними один до одного. Це може бути особливо корисно в розподілених системах, де програми можуть працювати на різних машинах або в різних місцях.

Брокерська технологія використовується в широкому спектрі додатків, особливо в системах корпоративного рівня, де важливий надійний та асинхронний зв'язок. Ось кілька поширених прикладів використання брокерських технологій:

– фінансові торгові системи: брокерська технологія зазвичай використовується у фінансових торгових системах для маршрутизації повідомлень між торговими програмами та біржами. Це гарантує, що угоди виконуються швидко і надійно [14];

– системи охорони здоров'я: брокерські технології використовуються в системах охорони здоров'я для безпечного та ефективного обміну даними про пацієнтів між різними додатками;

– електронна комерція: брокерські технології використовуються в системах електронної комерції для обробки замовлень, обробки платежів та інших важливих функцій [14];

– інтернет речей (IOT): брокерські технології використовуються в системах IoT, щоб дозволити пристроям взаємодіяти один з одним і з хмарними додатками.

Деякі приклади брокерських технологій включають:

– Apache ActiveMQ;
– RabbitMQ;
– Microsoft Message Queuing (MSMQ);
– IBM MQ.

На закінчення, технологія брокерів відіграє вирішальну роль у побудові надійних і масштабованих розподілених систем, забезпечуючи розділення, масштабованість і відмовостійкість. Такі приклади, як Apache ActiveMQ, RabbitMQ, Microsoft Message Queuing, IBM MQ та Amazon SQS, демонструють широке використання та універсальність брокерських технологій у різних галузях та додатках.

Аналіз гетерогенні обчислювальні середовища. Гетерогенні обчислювальні середовища (системи) розвиваються з 1990-х років. У минулому обчислювальні системи були гомогенними, тобто використовували однотипні процесори, пам'ять та комунікаційні протоколи [15]. Однак зі зростанням попиту на високопродуктивні обчислення і необхідністю швидко обробляти великі обсяги даних виникла потреба в обчислювальних системах нового типу. ГОС з'явилися як спосіб об'єднати різні типи процесорів, такі як CPU і GPU, для паралельного виконання різних завдань.

ГОС призначені для об'єднання різних типів процесорів та апаратного забезпечення для спільної роботи для досягнення спільної мети. Ключовим моментом у роботі ГОС є використання спеціалізованого програмного забезпечення, яке керує різними апаратними ресурсами та розподіляє завдання по всій системі. Рівень програмного забезпечення зазвичай складається з операційної системи, прикладного програмного забезпечення та проміжного програмного забезпечення. Проміжне програмне забезпечення виступає у ролі моста між різними програмними та апаратними компонентами, забезпечуючи їх безперебійну взаємодію та спільну роботу [15].

У ГОС завдання розподіляються між різними апаратними компонентами на основі їхніх сильних і слабких сторін. Наприклад, завдання, що вимагають високого рівня обчислень і математичної обробки, призначаються графічним

процесорам, тоді як завдання, що вимагають послідовної обробки і управління даними, призначаються центральним процесорам. Такий розподіл завдань дозволяє ГОС обробляти великі обсяги даних швидко та ефективно.

ГОС використовуються в різних галузях, включаючи наукові дослідження, інженерію та аналітику даних. Вони особливо корисні в додатках, що вимагають обробки великих обсягів даних, таких як прогнозування погоди, фінансове моделювання та біоінформатика.

ГОС також використовуються в хмарних обчисленнях, де кілька користувачів можуть отримати доступ до одних і тих самих обчислювальних ресурсів і використовувати їх одночасно. Хмарні ГОС можна використовувати за для таких додатків, як аналіз даних, машинне навчання та для штучного інтелекту [16].

Деякі приклади ГОС включають:

– CUDA: платформа паралельних обчислень та інтерфейс прикладного програмування (API), розроблений NVIDIA для використання з її графічними процесорами;

– OpenCL: фреймворк з відкритим вихідним кодом для розробки додатків, які можна виконувати на різних типах обладнання, включаючи CPU та GPU;

– Apache Hadoop: програмний фреймворк з відкритим вихідним кодом, який дозволяє розподілено обробляти великі масиви даних на кластерах комп'ютерів;

– OpenMPI: програмна бібліотека, яка дозволяє розробляти паралельні додатки, що можуть виконуватися на декількох процесорах і апаратних архітектурах.

Аналіз Enterprise Service Bus (ESB). Технологія ESB була представлена на початку 2000-х років як рішення для гнучкої, надійної та масштабованої інтеграції корпоративних систем. Вона була розроблена як розширення технологій проміжного програмного забезпечення і спрямована на подолання обмежень підходів до інтеграції «точка-точка» [13].

ESB виступає в ролі комунікаційного проміжного програмного забезпечення між декількома корпоративними додатками, сервісами та джерелами даних. Він забезпечує централізований центр для інтеграції, управління та моніторингу зв'язку між різними системами в розподіленому середовищі [13]. ESB спирається на набір стандартних протоколів обміну повідомленнями, таких як SOAP і HTTP, щоб забезпечити зв'язок між різними компонентами інтегрованої системи [13].

Технологія ESB зазвичай використовується в складних корпоративних середовищах, які вимагають інтеграції різноманітних додатків і джерел даних. Вона часто використовується в таких галузях, як банківська справа, страхування, охорона здоров'я та логістика, де існує потреба в інтеграції застарілих систем з новими системами та хмарними додатками.

Існує багато прикладів впровадження ESB в різних галузях. Наприклад, в галузі охорони здоров'я ESB використовується для інтеграції електронних медичних записів з іншими клінічними системами, такими як лабораторні інформаційні системи і радіологічні інформаційні системи. У банківській сфері ESB використовується для інтеграції основних банківських систем з системами мобільного та інтернет-банкінгу.

Аналіз middleware. Проміжне програмне забезпечення існує вже багато років, його коріння сягає 1980-х років, коли компанії почали використовувати проміжне програмне забезпечення, орієнтоване на повідомлення, для з'єднання різних додатків. З розвитком технологій розвивалося і проміжне програмне забезпечення, з розробкою корпоративних сервісних шин (ESB) та інших інструментів інтеграції [11].

Проміжне ПЗ виступає в ролі посередника між різними додатками, системами та джерелами даних, дозволяючи їм спілкуватися та обмінюватися інформацією. Це відбувається шляхом надання спільної платформи для взаємодії різних технологій між собою. Проміжне програмне забезпечення може також включати в себе готові роз'єми та адаптери для полегшення інтеграції між різними системами.

Проміжне програмне забезпечення використовується в широкому спектрі галузей, від фінансів до охорони здоров'я та роздрібної торгівлі. Його часто використовують у великих організаціях зі складною IT-інфраструктурою, де різні системи та додатки повинні безперебійно працювати разом [11].

Одним із прикладів застосування middleware є сфера фінансових послуг, де банки можуть використовувати middleware для зв'язку своїх основних банківських систем з клієнтськими додатками, такими як мобільні банківські додатки та інтернет-портали. Іншим прикладом є охорона здоров'я, де проміжне програмне забезпечення може використовуватися для інтеграції систем електронних медичних записів з іншими клінічними додатками, такими як системи візуалізації або аптечні системи [11].

Аналіз API-інтеграції. API-інтеграція - це процес інтеграції різних застосунків, сервісів і систем через використання API, що дає змогу різним застосункам взаємодіяти один з одним [17].

API-інтеграція виникла у зв'язку з необхідністю інтеграції різнорідних систем і застосунків, що працюють на різних платформах, для поліпшення процесу збору та обробки даних. Раніше цей процес був ручним, що призводило до високої ймовірності помилок і затримок. У світі API-інтеграція стала невід'ємною частиною розробки застосунків і програмного забезпечення.

API-інтеграція зазвичай включає в себе ряд кроків. Спочатку програміст налаштовує API, щоб він дозволяв доступ до потрібних функцій та даних. Далі, розробник додатку використовує цей API для взаємодії з сервісом. Запити та відповіді передаються між додатком та сервісом за допомогою мережевого протоколу [17]. API може бути реалізований за допомогою різних протоколів, таких як REST, SOAP, GraphQL та інші.

API-інтеграція використовується в багатьох галузях технологій, включаючи електронну комерцію, соціальні мережі, фінансові послуги, медіа та багато іншого. В багатьох випадках це дозволяє додаткам та сервісам

працювати разом та взаємодіяти між собою. Наприклад, веб-застосунок може використовувати API фінансової компанії для обробки платежів, а соціальна мережа може використовувати API мапи для відображення місцезнаходження користувачів.

Деякі приклади API-інтеграції включають:

інтеграція електронної комерції з системою управління замовленнями;

інтеграція CRM-системи з маркетинговою платформою для автоматизації процесу маркетингу.

Аналіз Enterprise Application Integration. EAI - це підхід до інтеграції додатків, який виник у відповідь на необхідність зв'язку між системами, розробленими на різних платформах і мовах програмування. Концепція EAI вперше з'явилася в 1990-х роках, коли компанії почали усвідомлювати, що для підвищення ефективності та зменшення витрат необхідно інтегрувати різні системи [18].

EAI дає змогу зв'язати різні додатки та системи, забезпечуючи їм обмін даними та спільне використання ресурсів. Цей підхід використовує спеціальні програмні компоненти, які забезпечують передачу даних між системами та виконання необхідних операцій. EAI може використовуватися як для інтеграції систем у межах однієї компанії, так і для зв'язку між компаніями або і між відділеннями.

EAI широко використовується у великих компаніях, які мають безліч додатків і систем, що працюють на різних платформах. Це може бути банківська справа, фінанси, охорона здоров'я, виробництво тощо. EAI також використовується для інтеграції систем електронної комерції та управління відносинами з клієнтами [19].

Одним із найвідоміших прикладів реалізації EAI є платформа MuleSoft, яка надає низку інструментів для інтеграції різних застосунків і систем. Іншим прикладом є IBM WebSphere EAI, який дає змогу зв'язати додатки та системи, що працюють на різних платформах і мовах програмування. Крім того, Microsoft BizTalk Server і Oracle Fusion Middleware також надають інструменти для EAI.

Порівняльна характеристика технологій. В процесі мого дослідження я провів аналіз технологій інтеграції додатків, таких як технології брокеру повідомлень, Гетерогенні обчислювальні середовища, ESB, middleware, API-інтеграція та Enterprise Application Integration. Кожна з цих технологій має свої переваги та обмеження.

Нижче наведена таблиця порівнянь цих технологій:

Таблиця 1.2 – Порівняння переваг та недоліків існуючих технології

Технологія	**Переваги**	**Недоліки**
Технологія брокеру повідомлень	Масштабованість: Брокерська технологія дозволяє додаткам горизонтально масштабуватися, тобто до системи можна додавати додаткові вузли, якщо це необхідно для обробки зростаючих навантажень; Надійність: Брокерська технологія забезпечує надійну постановку повідомлень в чергу і доставку, що гарантує, що повідомлення не будуть втрачені або доставлені в неправильному порядку; Асинхронний зв'язок: Брокерська технологія дозволяє додаткам взаємодіяти асинхронно, що зменшує зв'язок між додатками і робить систему більш стійкою до збоїв;	Складність: Архітектури на основі брокерів можуть бути складними у розробці, впровадженні та управлінні, особливо у великих системах з великою кількістю різних додатків; Затримка: Оскільки повідомлення маршрутизуються через брокера, в системі може виникнути додаткова затримка, яка в деяких випадках може вплинути на продуктивність; Єдина точка відмови: Якщо брокер виходить з ладу, це може вплинути на всю систему, що може бути критичною проблемою в системах, які вимагають високої доступності.

Продовження таблиці 1.2

Технологія	Переваги	Недоліки
Технологія брокеру повідомлень	Безпека: Брокерська технологія може забезпечити безпечну маршрутизацію та шифрування повідомлень, що важливо в системах, де передаються конфіденційні дані. Гнучкість і масштабованість: ESB забезпечує гнучку і масштабовану архітектуру, яка дозволяє додавати нові компоненти і сервіси, не порушуючи роботу існуючих;	Складність: Технологія ESB є складною і вимагає досвіду в різних сферах, таких як протоколи обміну повідомленнями, інтеграція додатків та управління проміжним програмним забезпеченням;
Enterprise Service Bus	Централізоване управління: ESB надає централізовану платформу управління, яка забезпечує кращий моніторинг, усунення несправностей та аудит зв'язку між різними компонентами інтегрованої системи; Стандартизація: ESB покладається на стандартні протоколи обміну повідомленнями, які забезпечують кращу сумісність між різними системами та компонентами.	Висока вартість: Технологія ESB може бути дорогою у впровадженні та підтримці, особливо у великих корпоративних середовищах; Накладні витрати на продуктивність: ESB додає рівень накладних витрат на зв'язок і обробку даних, що може вплинути на продуктивність інтегрованої системи.

Продовження таблиці 1.2

Технологія	**Переваги**	**Недоліки**
Middleware	Стандартизація комунікації: middleware надає стандартизовані механізми для обміну даними та комунікації між різними додатками та системами; Підтримка різних платформ: middleware може працювати на різних платформах і операційних системах, що забезпечує переносимість додатків;	Додаткові шари складності: Використання middleware може призвести до збільшення складності системи через необхідність настроювання та управління додатковим програмним забезпеченням;
API-інтеграція	Поліпшення ефективності та продуктивності бізнес-процесів; Збільшення швидкості розробки та висока гнучкість в інтеграції; Зменшення помилок і затримок, пов'язаних із ручним введенням даних; Зниження витрат на розробку та підтримку інтеграцій.	Складність інтеграції застосунків з різними API; Ризик безпеки під час передачі даних між додатками; Необхідність постійної підтримки та оновлення додатків для сумісності з новими версіями API.

Продовження таблиці 1.2

Технологія	**Переваги**	**Недоліки**
Enterprise Application Integration	Поліпшена ефективність: EAI дозволяє оптимізувати обмін даними та комунікацію між різними системами, що призводить до поліпшення продуктивності та ефективності бізнес-процесів; Підвищена надійність: EAI допомагає управляти та контролювати обмін даними, забезпечуючи надійність і цілісність інформації між системами; Зменшення витрат: Використання EAI дозволяє економити витрати на розробку та супровід застосунків, оскільки існуючі системи можуть бути інтегровані замість створення нових додатків з нуля; Підвищена гнучкість бізнесу: EAI дозволяє легко розширювати та змінювати функціональність систем, що забезпечує гнучкість відповідно до змінних потреб бізнесу.	Висока вартість розгортання: Впровадження технології EAI може бути дорогим процесом, пов'язаним з придбанням необхідного обладнання та програмного забезпечення, а також налаштуванням та інтеграцією систем; Складність налаштування та інтеграції: EAI вимагає високого рівня експертизи та знань для успішної настройки та інтеграції різних систем, що може бути складним завданням; Потреба у фахівцях: Використання технології EAI вимагає наявності кваліфікованих фахівців з досвідом у галузі EAI, що може бути витратним і складним з точки зору підбору персоналу.

Описане вище рішення, яке використовує middleware з API-інтеграцією, має кілька переваг порівняно з іншими підходами, такими як технологія брокеру повідомлень, ГОС, ESB і EAI.

Технологія брокеру повідомлень, хоч і забезпечує певний рівень взаємодії між додатками, вона може бути складною у впровадженні та налаштуванні. Вона зазвичай потребує встановлення та налаштування додаткового програмного забезпечення, а також залежить від специфічних протоколів комунікації. У порівнянні з middleware з API-інтеграцією, яке є більш простим та швидким для впровадження, технологія брокеру повідомлень може стати перешкодою для розгортання та інтеграції систем.

ГОС використовуються для інтеграції різних систем, що працюють на різних платформах та мовах програмування. Однак цей підхід може бути складним у впровадженні та потребує детального вивчення технологій і стандартів, що використовуються в різних системах. В порівнянні з middleware з API-інтеграцією, яка спрощує взаємодію та забезпечує гнучкість, гетерогенні обчислювальні середовища можуть вимагати більшої кількості зусиль для успішної інтеграції.

ESB є платформою, яка надає послуги маршрутизації та обміну даними між додатками. Він може бути корисним у випадках, коли потрібна складна маршрутизація або перетворення даних. Однак використання ESB може бути складним та потребувати значних зусиль для його впровадження. В порівнянні з middleware з API-інтеграцією, яке забезпечує простий та прямий обмін даними між додатками, ESB може бути надмірним і займати більше ресурсів.

EAI є підходом до інтеграції, який ставить за мету створити єдине середовище для обміну даними між різними додатками. Він може використовувати різні технології та протоколи для забезпечення взаємодії між системами. Проте, впровадження EAI може бути складним та вимагати значних зусиль для інтеграції різноманітних систем. У порівнянні з middleware з API-інтеграцією, яке може бути швидко впроваджено та забезпечити ефективний обмін даними, EAI може бути більш громіздким та менш гнучким підходом.

Отже, на основі вищезазначених переваг, рішення, яке використовує middleware з API-інтеграцією, є більш легким, сучасним та ефективним варіантом для впровадження інтеграції між різними ІС. Воно спрощує процес взаємодії та консолідує важливі документи, забезпечуючи швидку та надійну інтеграцію систем. Порівняно з іншими підходами, такими як Технологія брокеру повідомлень, Гетерогенні обчислювальні середовища, ESB і EAI, middleware з API-інтеграцією має переваги у простоті впровадження, швидкості обміну даними, гнучкості та ефективності. Враховуючи ці переваги, впровадження такого підходу до інтеграції систем буде раціональним та високоефективним для організацій.

2 ПРОЕКТУВАННЯ СИСТЕМИ

У данному розділі поясненно вибір програмних інструментів, архітектурних рішень та функціональності систем.

2.1 Опис інструментарію для розробки програмного забезпечення

Універсальний інтерфейс побудований як веб-сервіс. Який надає уніфікований інтерфейс для взаємодії між різноплатформними інформаційними системами. Він функціонує як посередник між цими системами, незалежно від їх архітектури, мов програмування чи операційних систем.

Універсальний інтерфейс дозволяє системам обмінюватися даними та спілкуватися одна з одною, використовуючи стандартизовані протоколи та формати обміну даними. Він надає зручний спосіб передавати інформацію, викликати методи, обмінюватися повідомленнями та синхронізувати дані між різними системами.

Універсальний інтерфейс забезпечує єдину точку доступу для взаємодії з різними системами, спрощуючи процес комунікації та забезпечуючи зручну та надійну взаємодію між ними. Він може бути реалізований як частина веб-сервісу, який надає API для взаємодії з іншими системами. Універсальний інтерфейс використовує стандартні протоколи, такі як HTTP або SOAP, та формати даних, такі як JSON або XML, для обміну інформацією між системами.

Для його реалізації обрано такі інструменти, як:

- мова програмування C#;
- середовище розробки Visual Studio Community 2019;
- бібліотека Newtonsoft.Json;
- технологія ASP.Net Core версії 5.0;
- бібліотека System.Security.Cryptography.

C# є мовою програмування від компанії Microsoft, яка відрізняється своєю простотою, потужністю та статичною типізацією. Вона належить до сімейства мов програмування C і має знайомий синтаксис для програмістів, які працювали з C, C++, Java та JavaScript. C# успішно поєднує найкращі особливості своїх попередників, таких як мови C, C++, Modula та Object Pascal, використовуючи практичний досвід з їхнього застосування.

Технологія ASP.NET Core є відкритою, кросплатформенною і високопродуктивною платформою для побудови веб-застосунків та веб-сервісів [20]. Вона є наступником популярної технології ASP.NET і була повністю переписана для забезпечення більшої продуктивності, гнучкості і масштабованості. Вона має багато переваг для побудови веб-сервісів та серверної розробки [20]. Ось деякі з них:

– кросплатформенність: ASP.NET Core може працювати на різних операційних системах, таких як Windows, macOS і Linux. Це дозволяє розробникам використовувати технологію на будь-якій платформі за їхнім вибором;

– висока продуктивність: ASP.NET Core побудований з фокусом на швидкодію та продуктивність. Він має оптимізовану обробку запитів і відповідей, що дозволяє обробляти більше запитів за одиницю часу та забезпечує швидку відповідь на клієнтські запити;

– масштабованість: ASP.NET Core надає потужні засоби для масштабування додатків. Він підтримує розподілені системи, що дозволяє горизонтально масштабувати застосунок за допомогою додавання нових серверів до кластера. Це забезпечує здатність додатку обробляти більше навантаження при збереженні продуктивності;

– гнучкість: ASP.NET Core надає розробникам велику гнучкість вибору різних інструментів та бібліотек для розробки. Ви можете використовувати мови програмування, такі як C# або F#, а також вибирати серед різних бібліотек і фреймворків для вирішення конкретних завдань;

– безпека: ASP.NET Core надає вбудовану підтримку безпеки та ідентифікації. Він має вбудовану підтримку автентифікації та авторизації, а також можливості захисту від атак, таких як перехоплення сесії і введення шкідливих кодів;

– легкість розгортання: ASP.NET Core дозволяє легко розгортати додатки на різних платформах та середовищах. Ви можете розгорнути застосунок в Docker-контейнерах, на хмарних платформах або на власних серверах залежно від вашого вибору.

Також для роботи з файлами JSON обрано Newtonsoft.Json. Вона є потужною бібліотекою для роботи з JSON у мові програмування C# [21]. Вона надає ряд переваг для розробників:

1) простота використання: Newtonsoft.Json має простий та зрозумілий API, що дозволяє легко серіалізувати (перетворювати об’єкти в JSON) та десеріалізувати (перетворювати JSON у об’єкти) дані;

2) висока продуктивність: Бібліотека володіє високою продуктивністю та ефективним використанням ресурсів;

3) гнучкість: Бібліотека підтримує різноманітні сценарії, такі як серіалізація/десеріалізація простих типів даних, колекцій, складних об’єктів, а також вбудованих типів даних C#.

Окрім цього для шифрування даних обрано **Бібліотека System.Security.Cryptography**. Вона є частиною платформи .NET Framework і надає набір класів та функцій для роботи з криптографією та забезпечення безпеки даних в програмах, розроблених на мовах програмування, які підтримують .NET.

Основна мета бібліотеки System.Security.Cryptography – це забезпечення конфіденційності, цілісності та автентифікації даних за допомогою різноманітних криптографічних алгоритмів. Вона надає можливість генерації та перевірки цифрових підписів, шифрування та розшифрування даних, гешування повідомлень та багато інших криптографічних операцій.

2.2 Система взаємодії кросплатформних інформаційних систем на основі універсальних інтерфейсів

Аналізуючи сучасний стан комунікації фермерів та трейдерів, можна помітити, що швидкість їх комунікації вигідна обом сторонам. Ситуація на агробіржах визначає ціни на агропродукцію, тому чим швидше фермер відправляє свої документи, тим швидше трейдер може прийняти рішення щодо покупки продукції. В результаті обидві сторони здобувають вигоду з цього процесу. Швидка комунікація дозволяє фермерам оперативно представляти свою продукцію на ринку, залучати зацікавлених трейдерів та забезпечувати швидкий обіг товарів. З іншого боку, трейдери можуть швидко реагувати на доступні пропозиції фермерів та ефективно управляти своїми закупівлями. Це створює сприятливі умови для обох сторін, сприяючи ефективній взаємодії та взаємній вигоді.

Та оскільки з розвитком цифрової технології, зростає рівня цифровізації у сільському господарстві, то все більше аграріїв та зернових брокерів створюють власні інформаційні системи для ефективного управління своїми операціями. Однак, цей процес призводить до розподіленості систем на різних платформах, що створює проблему взаємодії та обміну даними між ними. Додатково, виникає складність у забезпеченні сумісності інтерфейсів та синхронізації даних.

У такому контексті виникає потреба у розробці універсального інтерфейсу, який може об’єднати ці різнорідні інформаційні системи на різних платформах. Універсальний інтерфейс дозволить аграріям та зерновим брокерам спростити процес комунікації та обміну даними, незалежно від платформи, на якій працює їхні власна система. Цей інтерфейс може функціонувати як веб-сервіс, забезпечуючи їм доступ до потрібних функціональностей та можливість обміну даними з іншими системами.

Використання універсального інтерфейсу дозволить аграріям швидше та ефективніше відправляти свої документи та інформацію зерновим брокерам, а трейдерам зерна – швидше приймати рішення про покупку продукції. Це

сприятиме покращенню комунікації, збільшенню швидкості обробки даних та зниженню часу на прийняття важливих рішень. Універсальний інтерфейс стане цінним інструментом для аграріїв та зернових брокерів, допомагаючи їм ефективно взаємодіяти та забезпечувати взаємний успіх.

Для досягнення поставлених цілей сформульовано наступні вимоги до систем:

1) застосунок «Фермер»:

– формування та збереження документів, що стосуються продажу агропродукції;

– забезпечення взаємодії з універсальним інтерфейсом для отримання списку трейдерів та відправки пакету документів.

2) сервер універсального інтерфейсу:

– реєстрація трейдерів та забезпечення захищеного обміну документами;

– підтримка стандартів, таких як SOAP та REST, для забезпечення універсального комунікування з іншими системами;

– при обміні документів, усі документи повинні бути підписані, та зашифровані.

Вимоги до системи визначені з метою забезпечити зручну та безпечну обмін інформацією між фермерами та трейдерами. Використання IOS-застосунку «Фермер» дозволить фермерам ефективно формувати та зберігати документи, пов’язані з продажем агропродукції, а універсальний інтерфейс забезпечить їх комунікацію з трейдерами через захищений обмін документами та підтримку стандартів. Таке рішення сприятиме покращенню процесу обміну даними та спрощенню взаємодії між різними системами.

2.3 Формулювання вимог до функціональних модулів системи

Загальна архітектура взаємодії різнорідних кросплатформних систем (див. рис. 2.1), забезпечена сервером універсального інтерфейсу, пропонує ефективний спосіб безпроблемної інтеграції різних систем. Цей сервер

виступає посередником між різними компонентами, які можуть бути мобільними та підключаються до нього. Він забезпечує зручний канал комунікації між цими компонентами та різними модулями систем.

Рисунок 2.1 – Загальна архітектура системи

Завдяки цій загальній архітектурі, наш сервер універсального інтерфейсу сприяє зручній і надійній інтеграції різнорідних систем, забезпечуючи безшовну комунікацію та обмін даними між компонентами системи.

Система серверу універсального інтерфейсу має наступну структуру, що складається з різних функціональних підсистем:

1) підсистема реєстрації: відповідає за процес реєстрації трейдерів у системі універсального інтерфейсу. Вона повинна забезпечити такий функціонал:

– створення запису інформації про доступні API автентифікованих трейдерів;

– надавати трейдерам унікальний індентифікаційний номер у межах системи;

– забезпечити API до списку індентифікаційний номерів зарегестрованих трейдерів;

– підсистема реєстрації гарантує, що лише автентифіковані користувачі матимуть доступ до системи.

2) підсистема адаптації даних: відповідає за перетворення та адаптацію даних між різними платформами та форматами, дозволяючи ефективно обмінюватись даними між трейдерами та фермерами. Вона повинна забезпечити такий функціонал:

– забезпечення сумісність та інтеграцію існуючих ІС;

– забезпечити працю с протоколами SOAP та REST;

– коректна трансформацію данних з JSON файла у данні в XML файлі та навпаки.

3) підсистема безпеки: забезпечує захист конфіденційності, цілісності та доступності даних у системі. Отже, гарантує що дані трейдерів та фермерів залишаються конфіденційними та захищеними. Вона повинна забезпечити такий функціонал:

– реєстрація ключів трейдерів та фермерів у центрі сертифікаіції;

– створення відповідного сертифікату завдяки OpenSSL за стандартом X509;

– відправка сертифікату системам для підтвердження реєстрації.

4) підсистема моніторінгу: забезпечує постійний нагляд та контроль за роботою системи універсального інтерфейсу. Це дозволяє вчасно виявляти та вирішувати можливі проблеми та несправності, забезпечуючи безперебійну роботу системи. Вона повинна забезпечити ісінтрументами для збору та візуалізіції даних про функціонування системи.

Такий розподіл функціональних підсистем у системі універсального інтерфейсу дозволяє ефективно організувати та описати функціонал системи, забезпечуючи гнучкість, безпеку та високу якість комунікації між трейдерами та фермерами. Кожна підсистема має свою роль та взаємодіє з іншими для досягнення поставлених цілей системи.

Система «Фермер» це мобільний застосунок побудований для платформи IOS, повинен бути написаним на мові Swift, та мати наступну структуру, що складається з різних функціональних підсистем:

1) підсистема реєстрація: відповідає за процес реєстрації та авторизації фермерів у застосунку. Вона повинна забезпечити такий функціонал:

– реєстрація фермера у застосунку: підсистема надає можливість фермерам створювати облікові записи та реєструватись у системі. Цей процес включає збір необхідної інформації про фермера та створення унікального ідентифікатора для нього;

– авторизація фермера у застосунку: після реєстрації фермер може авторизуватись у системі, використовуючи свій обліковий запис та відповідний пароль. Ця процедура гарантує, що лише авторизовані користувачі мають доступ до функціоналу застосунку.

2) підсистема формування документів: відповідає за процес створення документів для продажу агропродукції. Вона повинна забезпечити функціонал:

– створення документу за вибраними параметрами: фермер може вибрати необхідні параметри та відповідні дані для створення документа про продаж агропродукції. Це включає вибір типу продукції, якості, встановлення ціни та інших важливих параметрів.

3) підсистема управління документами: відповадає за управління документами. Вона повинна мати такий функціонал:

– перегляд документа у вигляді PDF-файлу: фермер може переглядати створені документи у зручному форматі PDF;

– видалення документів: фермер має можливість видаляти непотрібні документи зі свого акаунту;

– сортування документів: підсистема надає можливість сортувати документи за різними критеріями, наприклад, за датою створення або типом документа;

– відправлення документів універсальному інтерфейсу: фермер може надсилати документи за допомогою універсального інтерфейсу до інших систем або користувачів.

4) підсистема моніторінгу: забезпечує постійний нагляд та контроль за роботою системи «Фермера». Вона повинна мати такий функціонал:

– запис інформації про стан системи до файлу: підсистема фіксує важливі дані про роботу системи та зберігає їх у файлі для подальшого аналізу;

– сортування інформації по даті: інформація про стан системи сортується за датою, що дозволяє здійснювати зручний аналіз динаміки роботи системи;

– інструмент візуалізації даних: підсистема надає можливість візуалізувати дані про роботу системи у зручній формі, наприклад, у вигляді графіків чи діаграм.

Такий розподіл на функціональні підсистеми дозволяє забезпечити ефективну роботу додатку, зручну реєстрацію та авторизацію фермерів, створення документів продажу агропродукції з врахуванням різних параметрів, управління документами та моніторинг стану системи.

2.4 Опис концептуальної моделі даних.

Описуючи концептуальну модель взаємодії систем, яка зображена на рисунку 2.2, можна виділити три основні сутності: фермер, сервер універсального інтерфейсу та трейдера. Далі будуть описані зв’язки між цими сутностями:

1) связки Фермера з сервером універсального інтерфейсу:

– процес «Отримання списку доступних трейдерів»: сервер універсального інтерфейсу надсилає запит до сервера універсального інтерфейсу для отримання списку трейдерів, які доступні для фермера. Цей процес дозволяє фермеру ознайомитися зі списком потенційних трейдерів для подальшої співпраці;

– процес «Відправлення даних (документа про продаж) з номером Трейдера»: фермер надсилає дані (наприклад, документ про продаж агропродукції) до універсального інтерфейсу, вказуючи номер трейдера, до якого він хоче направити ці дані. Універсальний інтерфейс отримує ці дані та передає їх на сервер універсального інтерфейсу для подальшої обробки та передачі до відповідного трейдера.

2) связки Трейдера з Сервером універсального інтерфейсу:

– процес «Реєстрації свого SOAP або REST API»: Трейдер здійснює реєстрацію свого SOAP або REST API на сервері універсального інтерфейсу. Це дозволяє серверу універсального інтерфейсу встановити зв’язок з трейдером та забезпечити можливість обміну даними між ними;

– процес «Відправлення даних (документа) за зазначеним протоколом (SOAP або REST)»: Сервер універсального інтерфейсу надсилає дані (наприклад, документ про продаж агропродукції) до трейдера за зазначеним протоколом (SOAP або REST). Це дозволяє серверу універсального інтерфейсу передавати дані до відповідного трейдера для обробки та подальшої взаємодії.

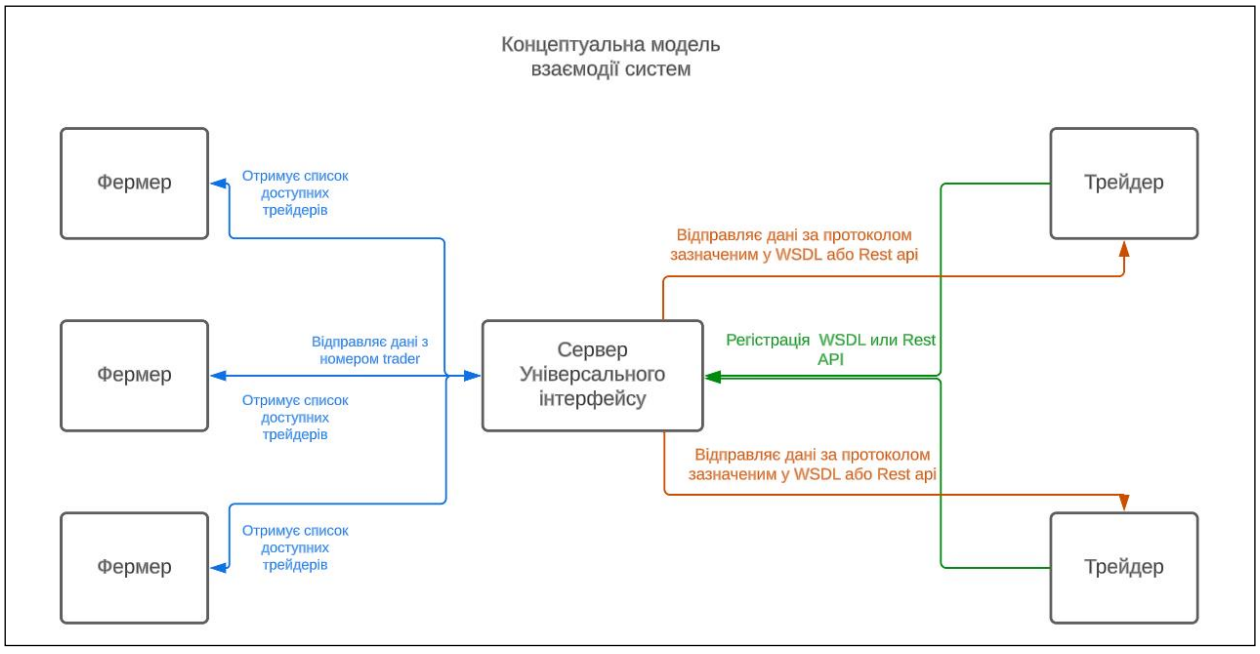

Рисунок 2.2 – Концептуальна модель взаємодії системи

Ця концептуальна модель відображає основні зв’язки та процеси взаємодії між фермером, універсальним інтерфейсом та трейдером. Вона створена з метою полегшити обмін даними та спростити комунікацію між цими системами у контексті продажу агропродукції.

Застосовані документи, описані у вищезазначених процесах, мають свою структуру, яка відповідає вимогам та потребам системи фермера. Дана структура може включати такі елементи:

1) заголовок документа містить інформацію про тип документа та його унікальний ідентифікатор;

2) дані про фермера: включають особисту інформацію про фермера, таку як його ім’я, прізвище, контактні дані та інші ідентифікаційні відомості;

3) дані про агропродукцію містять інформацію про продукцію, яка продається, включаючи її тип, якість, кількість та інші характеристики;

4) ціна та умови продажу визначають ціну продукції та додаткові умови, які встановлюються для угоди між фермером та трейдером;

5) підписи та авторизація включають підписи сторін, які підтверджують згоду та легітимність документа, а також механізми авторизації, що забезпечують доступ до документа тільки вповноваженим особам.

Також у даній системі є концептуальна модель безпеки зображена на рисунку 2.3.

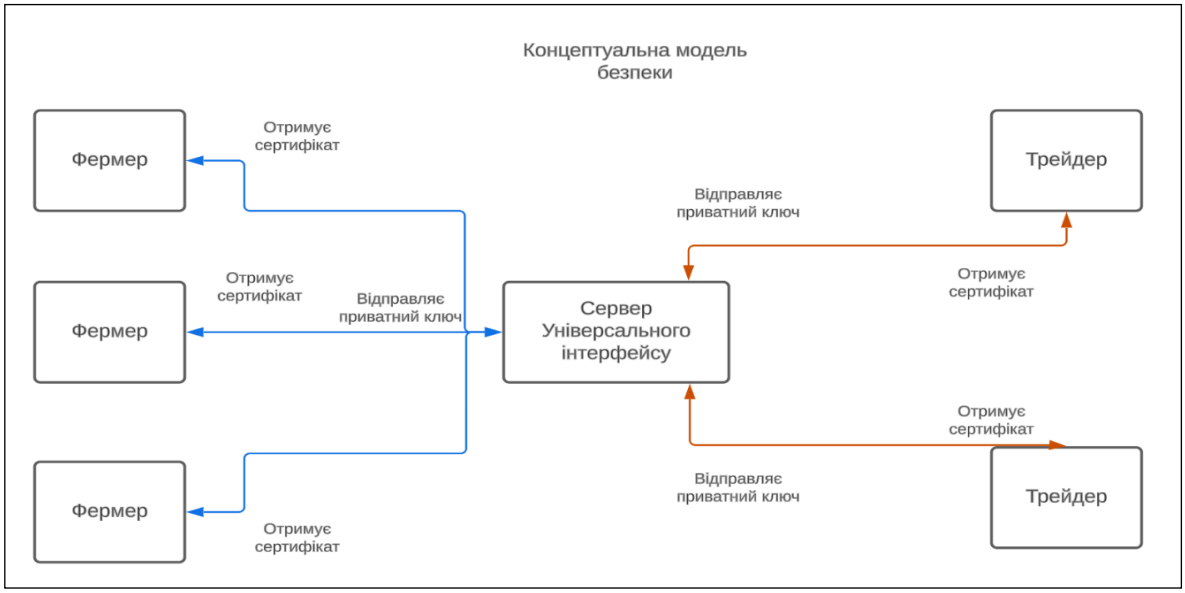

Рисунок 2.3 – Концептуальна модель безпеки

Концептуальна модель безпеки у системі включає наступні елементи, пов’язані зі зв’язками між фермерами, трейдерами та сервером універсального інтерфейсу:

1) від «Трейдера» або «Фермера» до «Сервера універсального інтерфейсу» іде процес «Відправка приватного ключа». Цей приватний ключ використовується для шифрування даних, які відправляються між сторонами. Приватний ключ надсилається на сервер універсального інтерфейсу для його реєстрації в центрі сертифікації, щоб підтвердити легітимність ключа та забезпечити безпеку комунікації.

2) від «Сервера універсального інтерфейсу» до «Трейдера» або «Фермера» іде процес «Отримання сертифіката». Цей сертифікат є електронним документом, який підтверджує ідентичність та достовірність сторін, зокрема, трейдера або фермера. Сервер універсального інтерфейсу видає сертифікат відповідній стороні, яка його запитала. Сертифікат містить публічний ключ, який використовується для розшифрування даних, які були зашифровані за допомогою приватного ключа.

Ці два процеси забезпечують безпеку передачі даних між фермерами, трейдерами та сервером універсального інтерфейсу. Вони гарантують захищеність даних шляхом використання шифрування та підтверджують

ідентичність сторін за допомогою сертифікатів. Такі заходи безпеки дозволяють забезпечити конфіденційність, цілісність та автентичність даних, що передаються у системі.

Проведено детальний аналіз концептуальної моделі взаємодії систем, враховуючи важливі аспекти безпеки. На основі цього аналізу здійснено опис функціональних блоків системи та розроблено рішення для реалізації програмного підходу.

Описані функціональні блоки системи надають зрозуміле уявлення про компоненти, які повинні бути реалізовані, а також про спосіб їх взаємодії. Цей аналіз створив міцну основу для розробки системи, дозволяючи приступити до реалізації з чітким розумінням функціональних вимог.

Таким чином, після завершення аналізу та розробки функціональних блоків, можна перейти до реалізацію системи.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

У даному розділі розглянута програмна структура систем та їх детальна реалізація. Також розглянуті перспективи розвитку системи.

3.1 Загальна структура систем.

Сервер універсального інтерфейсу – являє собою веб-сервіс, який складається з деяких контролерів, безлічі моделей та сервісу. Для детального дослідження поглянемо на рис. 3.1.

Розглянемо кожен елемент детальніше.

Блок «Контролери». У цьому блоку розміщені програмні контролери, які забезпечують комунікацію серверу з іншими системами. Ці контролери приймають запити від клієнтів і виконують відповідні дії на сервері, обробляючи ці запити.

Блок «Сервіси». У цьому блоку знаходиться клас CertificationCenter, який відповідає за реєстрацію ключів шифрування у центрі сертифікації. Цей сервіс забезпечує безпеку обміну інформацією між системами шляхом генерації та управління ключами шифрування.

Блок «Моделі». У цьому блоку знаходяться класи, які використовуються для отримання та обміну інформацією з системами. Ці моделі визначають структуру даних, які передаються між сервером і іншими системами, і допомагають узгоджувати формати та типи даних, що передаються в процесі комунікації.

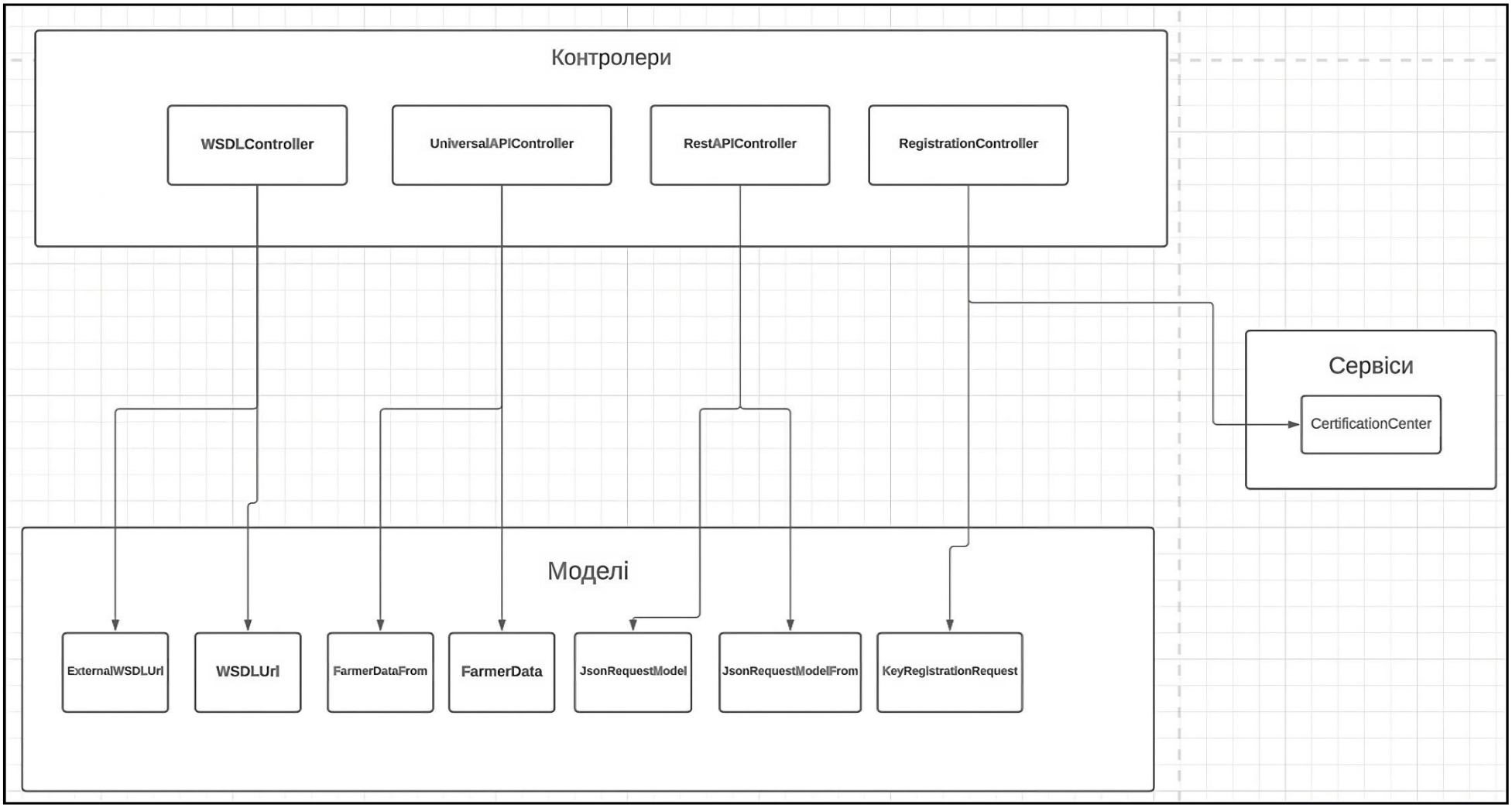

Рисунок. 3.1 – Загальна архітектура серверу унівrсального інтерфейсу

Система «Фермер» – являє собою нативний IOS застосунок, який також складається з моделей, контролерів та користувацьких інтерфейсів. Для детального дослідження поглянемо на рис. 3.2.

Розглянемо кожен елемент детальніше.

Блок «Контролери». У цьому блоку розміщені програмні контролери, які відповідають за обробку подій, взаємодію з користувачем та координацію роботи інших компонентів додатка. Основною відповідністю цим класам є реалізація патерну проектування Model-View-Controller (MVC). Контролери приймають вхідні події, аналізують їх та приймають рішення про подальші дії. Вони також відповідають за координацію роботи інших компонентів додатка.

Блок «Сервіси». У цьому блоку знаходиться клас «CertificationManager», який відповідає за генерацію ключів для шифрування інформації. Цей сервіс забезпечує безпеку обміну інформацією. Він виконує функції згенерування та управління ключами шифрування, що гарантує захищеність передаваних даних в додатку.

Блок «Моделі». У цьому блоку знаходяться класи, які використовуються для отримання та обміну інформацією з системами. Ці моделі визначають структуру даних, які передаються між застосунком та сервером, і допомагають узгоджувати формати та типи даних, що передаються в процесі комунікації. Моделі відображають дані, що використовуються в додатку, і забезпечують правильну передачу цих даних між компонентами.

Блок «Користувацькі інтерфейси». У цьому блоку знаходяться представлення, в яких описані правила побудови інтерфейсу мобільного додатку. Представлення відповідають за відображення даних користувачу та взаємодію з ним. Вони створюють та підтримують графічний інтерфейс додатку, забезпечуючи зручний та інтуїтивний спосіб користування додатком.

Кожен з цих елементів має свою важливу роль у роботі iOS додатку, забезпечуючи взаємодію з користувачем, безпеку даних, обмін інформацією та побудову інтерфейсу.

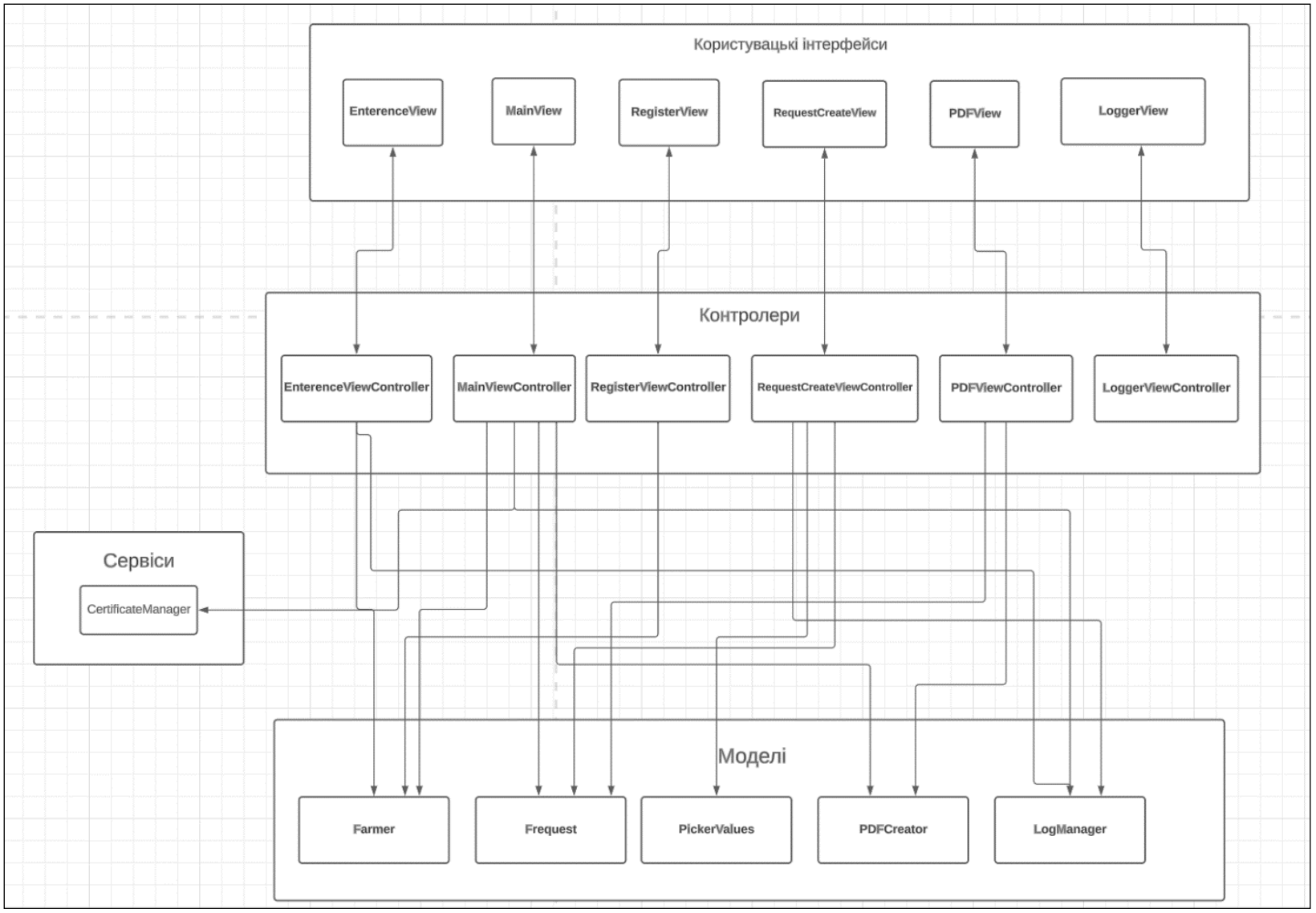

Рисунок 3.2 – Загальна архітектура системи Фермер

3.2 Реалізація базових класів системи «Фермер»

Зважаючи на функціональні підсистеми, які були згадані раніше, перейдемо до опису контролерів, які реалізують ці підсистеми в системі Фермер. Вказані контролери будуть виконувати обробку подій, взаємодію з користувачем та координацію роботи інших компонентів додатку. Основною відповідністю цим класам є реалізація патерну проектування Model-View-Controller (MVC).

Підсистему реєстрації забезпечує RegisterViewController та EnterenceViewController.

Перший реєструє користувача у систему та його дані зберігаються у локальній БД, реалізація якої представлена у додатку А. Її ER-діаграмму можно побачити на рис 3.3.

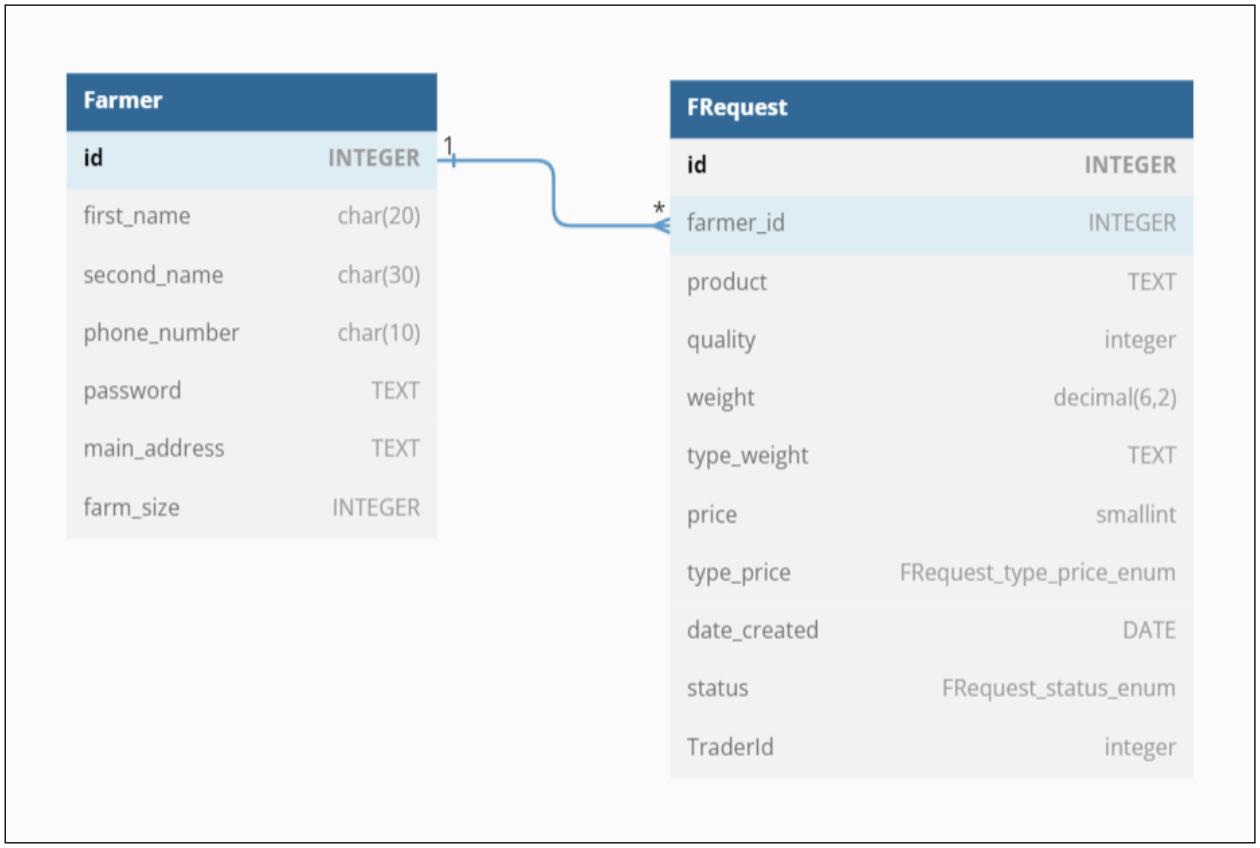

Рисунок 3.3 – ER-діаграмма БД системи Фермер

А вже другий контролер робить звернення до БД та аунтентіфікує користувача, частина реалізація якого представлена у додатку А.

Підсистему формування документа забезпучує контролер RequestCreateViewController. Він забезпечує валідацію даних та запис їх до БД. Частина реалізації якого представлена у додатку А.

Підсистему управління документами забезпечує два контролери, а саме MainViewController та PDFViewController.

MainViewController виконує такі завдання, як:

- видалення документів;
- сортування документів;
- відправлення документів.

А PDFViewController бере на себе останнє завдання по перегляду документів у вигляді PDF-файлів. Частина реалізації якого представлена у додатку А.

Підсистема моніторінгу забезпечує контролер LoggerViewController. Він вирішує такі задачі, як:

- запис інформації про стан системи до файлу;
- сортування інформації по даті;
- інструмент візуалізації даних.

Оскільки всі класи, що забезпечують реалізацію наявних підсистем, були описані, можна перейти до відповідних користувацьких інтерфейсів, які надають доступ до цих функціональних можливостей. Користувацькі інтерфейси взаємодіють з контролерами та моделями, щоб відображати дані та обробляти користувацькі взаємодії.

Для кожної підсистеми створено відповідний користувацький інтерфейс, який включатиме в себе необхідні елементи управління, візуальні компоненти та логіку обробки подій.

За підсистему реєстрація відповідає представлення RegisterView (див. рис. 3.4) та EnterenceView (див. рис. 3.5).

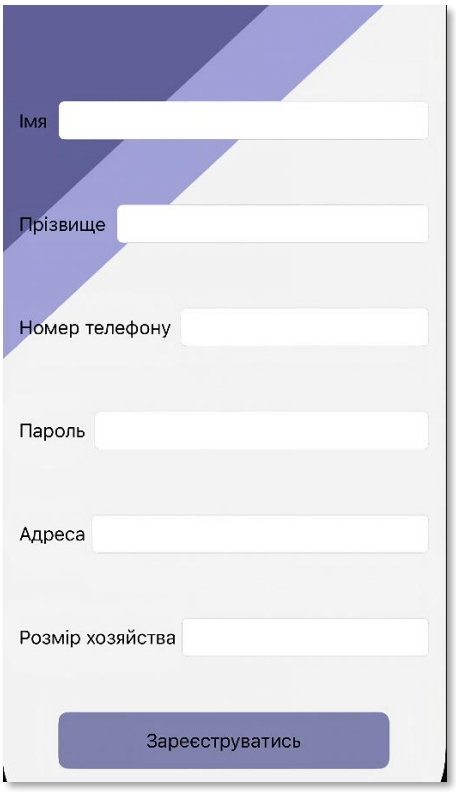

Рисунок 3.4 – Представлення RegisterView

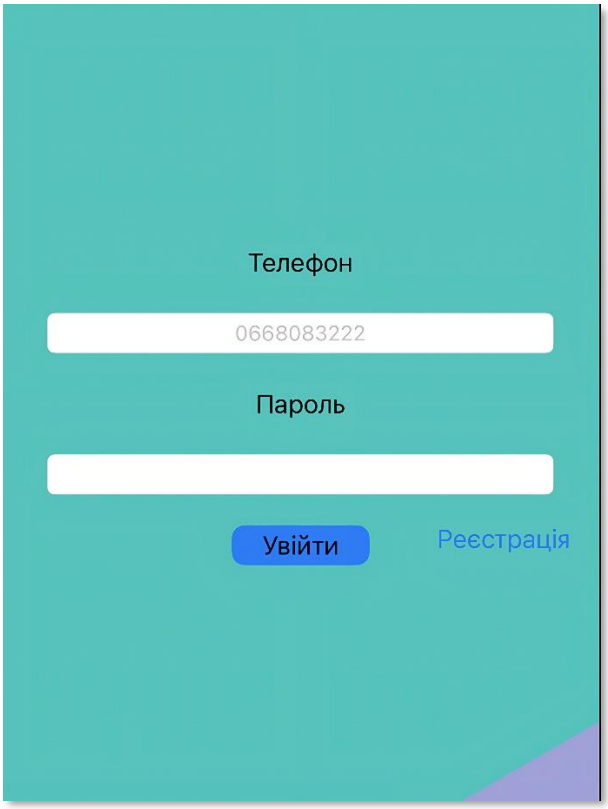

Рисунок 3.5 – Представлення EnterenceView

За підсистему формування документа відповідає представлення RequestCreateView (див. рис. 3.6).

Рисунок 3.6 – Представлення RequestCreateView

За підсистему управління документами відповідає декілька представлень, а саме MainView (див. рис. 3.7) та PDFView (див. рис. 3.8).

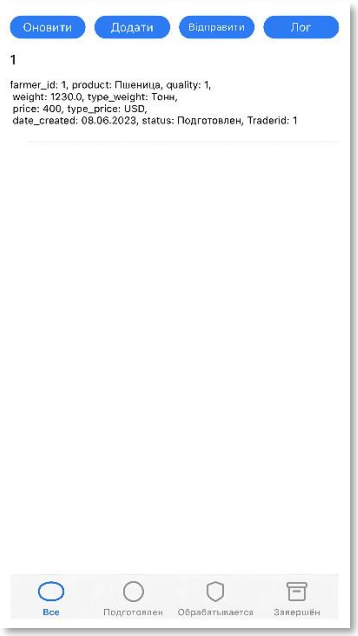

Рисунок 3.7 – Представлення MainView

Рисунок 3.8 – Представлення PDFView

За підсистему моніторінгу відповідає декілька представлень, а саме LoggerView (див. рис. 3.9).

Рисунок 3.9 – Представлення LoggerView

3.3 Реалізація базових класів сервера універсального інтерфейса

Зважаючи на функціональні підсистеми, які були згадані раніше, перейдемо до опису контролерів, які реалізують ці підсистеми в системі універсальний інтерфейс. Розглянемо кожен елемент детальніше.

Підсистему реєстрації реалізує два контролери WSDLController та RestAPIController.

WSDLController відповідає за реєстрацію SOAP API трейдера. Точніше кажучі клас WSDLController представляє собою контролер веб-застосунку і відповідає за обробку HTTP-запитів, пов'язаних з керуванням інформацією про WSDL-сервіси трейдерів. Він має атрибут [Route("API/WSDLController")], який визначає маршрут для доступу до методів контролера через API. Усередині класу визначено два приватних поля: combinedWSDLPath та idPath, які містять шляхи до файлів, що використовуються для збереження даних. Метод DownloadAndCombineWSDL дозволяє трейдеру записати свої дані на сервері. Коли трейдер відправляє POST-запит на маршрут /SendUrlWSDL, він передає об'єкт ExternalWSDLUrl, який містить інформацію про URL-адресу та метод WSDL-сервісу трейдера.

Отже, клас WSDLController служить для обробки запитів, пов'язаних з завантаженням та комбінуванням інформації про WSDL-сервіси трейдерів та їх ідентифікаторах. Він забезпечує взаємодію з файлами та оновлення даних, необхідних для подальшої роботи системи. Детальніше з частиною реалізацією можно ознайомитись у додатку А.

RestAPIController відповідає за реєстрацію REST API трейдера. Точніше кажучі клас RestAPIController є контролером веб-сервісу і відповідає за обробку HTTP-запитів, пов'язаних з керуванням інформацією про REST API трейдерів, та містить одну функцію під назвою ProcessJsonFile. Він має маршрут /API/RestAPIController для доступу до методів контролера через API.

Функція ProcessJsonFile виконує обробку отриманого POST-запиту. Вона здійснює наступні дії: зчитування останнього ідентифікатора з файлу

id_trader.json, збільшення ідентифікатора на 1 і запис нового значення назад у файл id_trader.json. Далі функція створює список для збереження об'єктів JsonRequestModel та отримує інформацію з отриманого POST-запиту. На основі цієї інформації створюється новий об'єкт JsonRequestModel, який додається до списку.

Потім функція зчитує вміст файлу restAPI.json, десеріалізує його у словник і перевіряє наявність певного ключа у словнику. Якщо ключ існує, нові дані додаються до відповідного списку. У випадку, якщо ключ не існує, створюється новий запис з відповідним ключем у словнику.

Далі словник серіалізується у відформатований JSON-рядок, який записується назад у файл restAPI.json. На завершення функція повертає відповідь з позитивним статусом для підтвердження успішного виконання запиту.

Детальніше з частиною реалізацією можно ознайомитись у додатку А.

Підсистему адаптації даних реалізує контролер UniversalAPIController. Клас **UniversalAPIController** є контролером, який виконує обробку HTTP-запитів до API. У цьому контролері є два методи:

1) метод ProcessJsonFileAsync обробляє POST-запит, який містить JSON-дані. Цей метод отримує дані з запиту, шукає відповідний API-сервіс (REST або WSDL) за допомогою ідентифікатора Traderid, і відправляє дані на цей сервіс для подальшої обробки. Якщо сервіс є REST API, дані надсилаються за допомогою HTTP-запиту POST. Якщо сервіс є WSDL API, формується SOAP-запит з даними і відправляється на відповідний URL. Метод повертає результат обробки запиту;

2) метод GetTradesTotalCount обробляє GET-запит і повертає кількість зареєстрованих трейдерів.

У цьому контролері також є допоміжні методи, такі як FindTraderByIdInJSON, SendDataToRestAPI і SendToWSDLSerivice, які виконують відповідні операції з даними, такі як пошук трейдера за ідентифікатором, надсилання даних на REST API або WSDL API тощо.

Цей контролер також містить шляхи маршрутизації, що вказуються за допомогою атрибута [Route]. Детальніше з частиною реалізацією можно ознайомитись у додатку А.

Підсистема безпеки реалізується завдяки класу CertificationCenter. Клас **CertificationCenter** представляє собою центр сертифікації, який відповідає за генерацію та збереження сертифікатів. В ньому реалізовані методи для реєстрації ключа, генерації сертифіката, а також відправки сертифіката на сервер.

При створенні екземпляра **CertificationCenter** ініціалізуються словник certificates для збереження сертифікатів та об'єкт httpClient для відправки HTTP-запитів.

Метод **RegisterKey** приймає ключ і URL сервера як параметри. Усередині методу виконуються наступні дії:

1) генерується сертифікат для переданого ключа за допомогою методу GenerateCertificate;

2) згенерований сертифікат додається до словника certificates з ключем, що відповідає переданому ключу;

3) сертифікат відправляється на сервер за допомогою методу SendCertificateToServer;

4) метод повертає згенерований сертифікат.

Метод **GenerateCertificate** створює об'єкт RSA для роботи з ключами, генерує випадковий ідентифікатор сертифіката, створює запит на сертифікат з використанням переданого ключа, створює самопідписаний сертифікат та повертає його.

Метод **SendCertificateToServer** приймає сертифікат і URL сервера як параметри. Усередині методу виконуються наступні дії:

1) сертифікат конвертується в байти за допомогою методу Export;

2) байти сертифіката відправляються на сервер за допомогою HTTP-запиту;

3) залежно від статусу відповіді сервера, виводиться відповідне повідомлення в консоль.

Таким чином, клас **CertificationCenter** надає зручний інтерфейс для генерації та збереження сертифікатів, а також їх відправки на сервер. Детальніше з частиною реалізацією можно ознайомитись у додатку А.

Підсистема моніторінгу реалізована завдяки команди для виводу інформації про стан системи на консоль. Ці команди дозволяють отримувати актуальну інформацію про різні аспекти системи, такі як стан компонентів, параметри, помилки або події.

У цьому розділі описано реалізацію базових класів сервера універсального інтерфейсу. Були розглянуті ключові аспекти, пов'язані з розробкою цих класів, включаючи архітектурний підхід, розробку API, обробку запитів та відповідей, а також управління сесіями.

Процес реалізації базових класів сервера універсального інтерфейсу включав ретельне проектування, розробку та тестування. Були використані передові практики програмування та принципи об'єктно-орієнтованого дизайну для створення ефективного та розширюваного серверного компонента.

Завдяки реалізації базових класів сервера універсального інтерфейсу стало можливим забезпечити обробку різноманітних запитів та взаємодію з клієнтами через стандартизований інтерфейс. Це дозволяє забезпечити легку інтеграцію з різноманітними системами та розширити функціональні можливості сервера.

3.4 Перспективи розвитку системи

Головною перевагою сервера універсального інтерфейсу є його легкість розширення, зокрема здатність легко підключати інші системи і надавати їм можливість комунікації. Отже, для поліпшення системи важливо зосередитися на покращенні його продуктивності.

Крім того, варто зазначити, що сучасні технології продовжують розвиватися, а апаратні можливості з кожним роком зростають. Злочинці, які

мають намір незаконно отримати доступ до приватних даних, стають все більш активними [22]. Тому, для уникнення сценаріїв зламу, необхідно розглядати покращення безпеки системи з перспективи.

Також була розроблена методика оцінки ефективності інтеграції та розробки інформаційних систем. Для досягнення цієї мети був проведений детальний аналіз та зібрана інформація про різні технології та їх вартість. Результати цього дослідження були представлені у вигляді таблиці нижче:

Таблиця 3.1 – Таблиця оцінки ефективності технології.

Технологія	**Швидкість**	**Кошторис**
Сервер універсального інтерфейсу	Середня 20 м/с, залежить від віддаленості хостінг серверу.	Оплата хостінгу в середньому від $3 до $15 на місяць
Mulesoft integration(EAI)	Залежить від пакету	Оплата $80,000 на рік
IBM MQ	Залежить від виду ліцензії	10000 до 15000 долларів за ліцензію

З отриманих результатів видно, що майже всі аналоги дуже дорогі. У порівнянні з ними, використання серверу універсального інтерфейсу може бути більш доступним з вартістю від $3 до $15 на місяць.

ВИСНОВКИ

В результаті аналізу предметної області сформульовано задачі, які необхідно виконати для досягнення мети проекту. Також порівняно існуючі види кросплатформних програмних компонентів та порівняні сучасні технології інтеграції систем, такі як:

- технологія брокеру повідомлень;
- ГОС;
- ESB;
- EAI;
- middleware;
- API-інтеграція.

На основі аналізу переваг та недоліків сформульовано рішення, що використання middleware з API-інтеграцією є найбільш легким, сучасним та ефективним варіантом для впровадження інтеграції між різними ІС.

Для створення серверу універсального інтерфейсу обранна технологія ASP.NET Core, яка дозволяє створювати веб-сервіси, для роботи з JSON використовувалась бібліотека Newtonsoft.Json, та для забезпечення безпеки використовувалась бібліотека System.Security.Cryptography, та усе написано на мові програмування C#.

Проведено детальний аналіз концептуальної моделі взаємодії систем, враховуючи важливі аспекти безпеки. На основі цього аналізу здійснено опис функціональних блоків системи та розроблено рішення для реалізації програмного підходу. Описані функціональні блоки системи надають зрозуміле уявлення про компоненти, які повинні бути реалізовані, а також про спосіб їх взаємодії. Цей аналіз створив міцну основу для розробки системи, дозволяючи приступити до реалізації з чітким розумінням функціональних вимог.

Завдяки цьому успішно створено застосунок «Фермер» та сервер універсального інтерфейсу, які реалізують усі функціональні підсистеми визначені на етапі планування.

Дана система ще не ідеальна, тому визначенно основні напрямки удосконалення системи, такі як безпека та продуктивність.

За результатами даної роботи опубліковано тези на всеукраїнській конференції та відбулось впровадження системи у процесс продажу агропродукції (див. додаток Б).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Сергатий Є. Ю., Дослідження існуючих технологій забезпечення інтеграції різних інформаційних систем в єдине ціле / Є. Ю. Сергатий, О. С. Максимов // Інформатика, інформаційні системи та технології: тези доповідей двадцятої всеукраїнської конференції студентів і молодих науковців. Одеса, 28 квітня 2023 р. - Одеса, 2023. – С.128-130.
2. Коваленко О. Є. Системна інженерія та життєвий цикл систем // Електронне моделювання, т. 40, № 6, С. 61–82, 2018.
3. Roco M. C., Bainbridge W. S. The new world of discovery, invention, and innovation: convergence of knowledge, technology, and society // Journal of Nanoparticle Research, т. 15, № 9, pp. 1-17, 2013.
4. Roco M. C. Convergence-Divergence Process // Handbook of Science and Technology Convergence / W. Bainbridge, M. Roco, eds. – Cham, Springer, 2016, pp. 79-93.
5. da Silva A. R. Model-Driven Engineering: A Survey Supported by a Unified Conceptual Model // Computer Languages, Systems & Structures, № 43, p. 139–155, 2015.
6. Лекція 19. Крос-платформне програмування (2016 р.) (victana.lviv.ua) [Електронний ресурс]. – Режим доступу: https://victana.lviv.ua/knyhy/konspekty-lektsii/133-kros-platformenne-prohramuvannia-ta-khmarni-servisy/567-lektsiia-19-kros-platformne-prohramuvannia-2016-r
7. Cross-platform software – CodeDocs [Електронний ресурс]. – Режим доступу: https://codedocs.org/what-is/cross-platform-software
8. What Are Cross Platform Programming Languages? (embarcadero.com) [Електронний ресурс]. – Режим доступу: https://blogs.embarcadero.com/what-are-cross-platform-programming-languages/

9. A Guide to Cross Platform App Development Frameworks in 2023 (netsolutions.com) [Електронний ресурс]. – Режим доступу: https://www.netsolutions.com/insights/cross-platform-app-frameworks-in-2019/
10. What Are Interfaces? (With Definition and Examples) | Indeed.com [Електронний ресурс]. – Режим доступу: https://www.indeed.com/career-advice/career-development/what-are-interfaces
11. What is an API (Application Programming Interface)? – GeeksforGeeks [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/what-is-an-API/
12. Що це таке і як це працює? - techukraine.net [Електронний ресурс]. – Режим доступу: https://techukraine.net/що-це-таке-і-як-це-працює/
13. What is an ESB (Enterprise Service Bus)? | IBM [Електронний ресурс]. – Режим доступу: https://www.ibm.com/topics/esb
14. What are Message Brokers? | IBM [Електронний ресурс]. – Режим доступу: https://www.ibm.com/topics/message-brokers
15. Heterogeneous Computing | Gigabyte [Електронний ресурс]. – Режим доступу: https://www.gigabyte.com/Glossary/heterogeneous-computing
16. Heterogeneous Computing Is About Optimizing Resources - EE Times Gigabyte [Електронний ресурс]. – Режим доступу: https://www.eetimes.com/heterogeneous-computing-is-about-optimizing-resources
17. API Integration - What is API Integration? | API Glossary (rAPIdAPI.com) [Електронний ресурс]. – Режим доступу: https://rAPIdAPI.com/blog/API-glossary/API-integration/
18. Enterprise Integration: What It Is and Why It’s Important | IBM [Електронний ресурс]. – Режим доступу: https://www.ibm.com/cloud/blog/enterprise-integration
19. What is application integration? | IBM [Електронний ресурс]. – Режим доступу: https://www.ibm.com/topics/application-integration

20. Overview of ASP.NET Core | Microsoft Learn [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-5.0
21. Newtonsoft Json.NET Documentation [Електронний ресурс]. – Режим доступу: https://www.newtonsoft.com/json/help/html/Introduction.htm
22. Войтович В.С., Гриник Р.О. Основні безпекові проблеми кіберпростору України. // Зб. тез доповідей Міжнародна науково-практична конференція “Інформаційна безпека в сучасному суспільстві” (м. Львів, 24-25 листопада 2016 р.). Львів : ЛДУБЖД, 2016. С. 23–24.

ДОДАТОК А

Вихідний код програмної реалізації

Створення БД для системи Фермер.

```
CREATE TABLE FRequest (
    id INTEGER PRIMARY KEY AUTOINCREMENT CHECK(id >0),
    farmer_id INTEGER not null,
    product TEXT not null CHECK(product IN ('Пшеница',
'Кукуруза', 'Подсолнечник', 'Ячмень', 'Соя')) ,
    quality integer not null check(quality > 0 and quality < 6),
    weight decimal(6,2) not null,
    type_weight TEXT not null CHECK(type_weight IN
('Тонн','Куб')),
    price smallint not null check(price > 0),
    type_price TEXT not null CHECK(type_price IN ('UAH', 'USD',
'EUR')),
    date_created DATE not null,
    status TEXT not null CHECK(status IN ('Подготовлен',
'Обрабатывается', 'Завершён')),
    TraderId integer not NULL,
    FOREIGN KEY (farmer_id) REFERENCES Farmer(id)  on update
Cascade
);
```

Частина реалізації контролера EnterenceViewController.

```
@IBAction func enterButtonPressed(_ sender: UIButton) {

        errorLabel.text = ""
        farmers = []
        if(phoneTextField.text != "" && passwordTextField.text
!= "")
        {
            let enteredPhoneNumber = phoneTextField.text!
            let enteredPassword = passwordTextField.text!
            print("\(enteredPhoneNumber) \(enteredPassword)")
            try? dbQueue?.read { db in
                let rows = try Row.fetchAll(db, sql: "SELECT *
FROM Farmer where phone_number = ? and password = ?", arguments:
[enteredPhoneNumber,enteredPassword])

                for row in rows {
                    farmers.append(Farmer(id: row[0],
first_name: row[1], second_name: row[2], phone_number: row[3],
password: row[4], main_address: row[5], farm_size: row[6]))
                }

            }
```

```
            if let value = farmers.first?.id
            {
                //створення Логера та запис в нього інформації
                myLogger =
LogManager(String(farmers.first!.phone_number))
                myLogger.log("Пользователь
\(farmers.first!.first_name) успешно вошёл")

                self.performSegue(withIdentifier: "goToData",
sender: self)
            }
            else
            {
                print("error")
                errorLabel.text = "Ошибка такого аккаунта не
существует"
            }
        }
        else
        {
            errorLabel.text = "Не все поля заполненные"
        }
    }
```

Частина реалізації контролера RequestCreateViewController.

```
@IBAction func addReqButtonPressed(_ sender: UIButton) {

            if (!checkTextFields())
            {
                errorLabel.text = "Не все поля были заполненны"
            }
            else
            {
                var newRequest: Frequest = Frequest(id: 0,
farmer_id: farmerID, product: selectedTypeProduct!, quality:
selectedQuality!, weight:
Double(weightTextField.text!)!,type_weight: selectedTypeWeight!
, price: Int(priceTextField.text!)!, type_price:
selectedTypePrice!, date_created: currentDate!,status:
selectedStatus!, Traderid: TraderId!)

                do{
                    try dbQueue?.write { db in
                        try db.execute(sql: "INSERT INTO
FRequest (farmer_id, product, quality, weight,type_weight,
price,type_price, date_created,status,Traderid) VALUES
(?,?,?,?,?,?,?,?,?,?)", arguments: [newRequest.farmer_id,
newRequest.product, newRequest.quality, newRequest.weight,
newRequest.type_weight, newRequest.price,
```

```
newRequest.type_price,newRequest.date_created,newRequest.status,
newRequest.Traderid])
                    }
            }
                catch {
                    print("Произошла неизвестная ошибка:
\(error)")
                }
                self.dismiss(animated: true)
            }
    }

    func checkTextFields() -> Bool
    {
        var flag:Bool = true

        if(weightTextField.text! == "" || priceTextField.text!
== "")
        {
            flag = false
        }
        return flag
    }

    func TextsFieldChanged()
    {
        incomeLabel.text = "Заработок: "

        if(checkTextFields())
        {
            if let weightText = weightTextField.text, let
priceText = priceTextField.text, let selectedType =
selectedTypePrice {
                if let weight = Double(weightText), let price =
Double(priceText) {
                    let income = weight * price
                    let incomeText = String(income) + " " +
selectedType
                    incomeLabel.text! += incomeText
                }
            }
        }
    }
```

Частина реалізації контролера EnterenceViewController.

```
private func openOrCreatePDFFile() {
            // Проверка, существует ли файл по указанному пути
            if FileManager.default.fileExists(atPath:
pdfURL.path) {
                // Если файл существует, открыть его
                // Если файл существует, удалить его
```

```
                do {
                    try FileManager.default.removeItem(at:
pdfURL)
                } catch {
                    print("Ошибка при удалении существующего
PDF-файла: \(error.localizedDescription)")
                }
                createAndDisplayPDF()

            }
            else {
                // Если файл не существует, создать и открыть
новый файл
                createAndDisplayPDF()
            }
        }

    private func createAndDisplayPDF() {


            let body: String = " Номер заявки:
\(additionalData.id)\n Создана: \(creator_name)\n По продаже
товара: \(additionalData.product)\n Качества:
\(additionalData.quality)\n Вес всей продукции:
\(additionalData.weight) \(additionalData.type_weight)\n По цене
\(additionalData.price)\(additionalData.type_price) за 1
\(additionalData.type_weight)\n Дата создания заявки:
\(additionalData.date_created)\n Статус заявки:
\(additionalData.status)\n Общая сума:
\(Double(additionalData.price) * Double(additionalData.weight))
\(additionalData.type_price)"

            let pdfCreator = PDFCreator(title: "Информация об
заявке на продажу", body: body)

            pdfView.document = PDFDocument(data:
pdfCreator.createFlyer())

        }

    @objc private func closeButtonTapped() {
           dismiss(animated: true, completion: nil)
       }
```

Частина реалізації контролера WSDLController.

```
[Route("API/WSDLController")]
    public class WSDLController : Controller
    {
        private readonly string combinedWSDLPath = @"G:\Учёба\4
курс  - 2 семестр\Диплом\Програмная
```

```
реализация\ConfigService\ConfigService\APIFiles\combined_wsdl.js
on";
        private readonly string idPath = @"G:\Учёба\4 курс  - 2
семестр\Диплом\Програмная
реализация\ConfigService\ConfigService\APIFiles\id_trader.json";


        [HttpPost]
        [Route("SendUrlWSDL")]
        public async Task<IActionResult>
DownloadAndCombineWSDL([FromBody] ExternalWSDLUrl externalWSDL)
        {
            // Чтение последнего идентификатора из файла
id_trader.json
                    string idJson =
System.IO.File.ReadAllText(idPath);
                    dynamic idData =
JsonConvert.DeserializeObject(idJson);
                    string lastId = idData.id.ToString();

                    // Увеличение идентификатора на 1
                            int newId = int.Parse(lastId) + 1;

                            // Запись нового идентификатора
обратно в файл id_trader.json
                                    idData.id = newId;
                                    string newIdJson =
JsonConvert.SerializeObject(idData, Formatting.Indented);

System.IO.File.WriteAllText(idPath, newIdJson);

            // Создаем список для хранения объектов
ExternalWSDLUrl
            List<WSDLUrl> traders = new List<WSDLUrl>();

            // Создаем объект ExternalWSDLUrl с использованием
нового идентификатора
            var trader = new WSDLUrl
            {
                TraderId = lastId.ToString(),
                url = externalWSDL.url,
                method = externalWSDL.method
            };

            // Добавляем объект ExternalWSDLUrl в список
            traders.Add(trader);


            // Загружаем содержимое файла combined_wsdl.json
            string jsonFromFile =
System.IO.File.ReadAllText(combinedWSDLPath);
```

```
            // Десериализуем содержимое файла в объект
            Dictionary<string, List<WSDLUrl>> jsonData =
JsonConvert.DeserializeObject<Dictionary<string,
List<WSDLUrl>>>(jsonFromFile);

            // Проверяем, существует ли ключ "Traders" в словаре
            if (jsonData.ContainsKey("Traders"))
            {
                // Если ключ существует, добавляем новые данные
в список
                jsonData["Traders"].AddRange(traders);
            }
            else
            {
                // Если ключ не существует, создаем новую запись
в словаре
                jsonData.Add("Traders", traders);
            }

            // Сериализуем объект обратно в JSON-строку
            string updatedJson =
JsonConvert.SerializeObject(jsonData, Formatting.Indented);

            // Записываем обновленную JSON-строку обратно в файл
            System.IO.File.WriteAllText(combinedWSDLPath,
updatedJson);

            return Ok();

        }
    }
```

Частина реалізації контролера RestAPIController.

```
using Microsoft.AspNetCore.Mvc;
using Newtonsoft.Json;
using System.IO;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.OpenAPI.Models;
using ConfigService.Models;

namespace ConfigService.Controllers
{
    [Route("API/RestAPIController")]
    public class RestAPIController : Controller
    {
        private readonly string restJsonPath = @"G:\Учёба\4 курс
- 2 семестр\Диплом\Програмная
реализация\ConfigService\ConfigService\APIFiles\restAPI.json";
```

```
        private readonly string idPath = @"G:\Учёба\4 курс  - 2
семестр\Диплом\Програмная
реалізация\ConfigService\ConfigService\APIFiles\id_trader.json";

        [HttpPost]
        [Route("SendAPIJson")]
        public IActionResult ProcessJsonFile([FromBody]
JsonRequestModelFrom requestModel)
        {
            // Чтение последнего идентификатора из файла
id_trader.json
            string idJson = System.IO.File.ReadAllText(idPath);
            dynamic idData =
JsonConvert.DeserializeObject(idJson);
            string lastId = idData.id.ToString();

                // Увеличение идентификатора на 1
                int newId = int.Parse(lastId) + 1;

                    // Запись нового идентификатора обратно в
файл id_trader.json
                    idData.id = newId;
                    string newIdJson =
JsonConvert.SerializeObject(idData, Formatting.Indented);
                    System.IO.File.WriteAllText(idPath,
newIdJson);

            // Создаем список для хранения объектов
JsonRequestModel
            List<JsonRequestModel> traders = new
List<JsonRequestModel>();

            // Получаем информацию из запроса
            var trader = new JsonRequestModel
            {
                TraderId = newId.ToString(),
                Endpoint = requestModel.Endpoint,
                HttpMethod = requestModel.HttpMethod,
                Parameters = requestModel.Parameters
            };

            // Добавляем объект JsonRequestModel в список
            traders.Add(trader);


            // Загружаем содержимое файла combined_wsdl.json
            string jsonFromFile =
System.IO.File.ReadAllText(restJsonPath);

            // Десериализуем содержимое файла в объект
            Dictionary<string, List<JsonRequestModel>> jsonData
= JsonConvert.DeserializeObject<Dictionary<string,
List<JsonRequestModel>>>(jsonFromFile);
```

```
            // Проверяем, существует ли ключ "Traders" в словаре
            if (jsonData.ContainsKey("Traders"))
            {
                // Если ключ существует, добавляем новые данные
в список
                jsonData["Traders"].AddRange(traders);
            }
            else
            {
                // Если ключ не существует, создаем новую запись
в словаре
                jsonData.Add("Traders", traders);
            }

            // Сериализуем объект обратно в JSON-строку
            string updatedJson =
JsonConvert.SerializeObject(jsonData, Formatting.Indented);

            // Записываем обновленную JSON-строку обратно в файл
            System.IO.File.WriteAllText(restJsonPath,
updatedJson);

            return Ok();
        }
    }
}
```

Частина реалізації контролера UniversalAPIController.

```
[Route("API/UniversalAPIController")]
    public class UniversalAPIController : Controller
    {

        string fileRestPath = @"G:\Учёба\4 курс  - 2
семестр\Диплом\Програмная
реалізация\ConfigService\ConfigService\APIFiles\restAPI.json";
        string fileWSDLPath = @"G:\Учёба\4 курс  - 2
семестр\Диплом\Програмная
реалізация\ConfigService\ConfigService\APIFiles\combined_wsdl.js
on";


        [HttpPost]
        [Route("SendAPIJson")]
        public async Task<IActionResult>
ProcessJsonFileAsync([FromBody] FarmerDataFrom requestModel)
        {
            // Получаем информацию из запроса

            #region Читаем и получаем данные з полученого json'a
```

```
            int id = requestModel.id;
            int farmer_id = requestModel.farmer_id;
            string product = requestModel.product;
            int quality = requestModel.quality;
            double weight = requestModel.weight;
            string type_weight = requestModel.type_weight;
            int price = requestModel.price;
            string type_price = requestModel.type_price;
            string date_created = requestModel.date_created;
            string status = requestModel.status;



            string Traderid =
Convert.ToString(requestModel.Traderid);


            // Создаем объект, представляющий информацию
            var requestData = new FarmerData
            {
                id = id,
                farmer_id = farmer_id,
                product = product,
                quality = quality,
                weight = weight,
                type_weight = type_weight,
                price = price,
                type_price = type_price,
                date_created = date_created,
                status = status
            };


            #endregion Читаем и получаем данные з полученого
json'a



            #region <---- Ищем нужный зарегестрированный API-
сервис ---->


                    #region <---- Ищем нужный зарегестрированный
RestAPI-сервис ---->
                    // Достаём id и смотрим в каком файле оно
есть

                    //string restAPIPath = "restAPI.json"; //
Путь к файлу restAPI.json

                    string restAPIData =
System.IO.File.ReadAllText(fileRestPath); // Чтение содержимого
restAPI.json
```

```
//string wsdlData = System.IO.File.ReadAllText(wsdlPath); //
Чтение содержимого combined.wsdl


                    JsonRequestModelFrom jsonRestFromFile =
null;

   try
    {
      JToken selectedTrader = FindTraderByIdInJSON(restAPIData,
Traderid);
if (selectedTrader != null)
                        {
        var tmp = new JsonRequestModelFrom
         {
     Endpoint = selectedTrader["Endpoint"].ToString(),
                                HttpMethod =
selectedTrader["HttpMethod"].ToString(),
                                Parameters =
selectedTrader["Parameters"].ToObject<Dictionary<string,
string>>()
      };

  jsonRestFromFile = tmp;
  }
    else
                        {
                            Console.WriteLine($"JSON-объект с
TraderId {Traderid} не найден.");



                        }
                    }
                    catch (JsonReaderException ex)
                    {
                       Console.WriteLine($"Ошибка при чтении
JSON: {ex.Message}");
                    }
                    catch (Exception ex)
                    {
                        Console.WriteLine($"Ошибка:
{ex.Message}");
                    }

            #endregion <---- Ищем нужный зарегестрированный
RestAPI-сервис ---->



            // Тут мі считаем с json где описан всдл и если уже
там нету такого номера то вернём ошибку
```

```
            ExternalWSDLUrl jsonWSDLFromFile = null;

                if (jsonRestFromFile == null)
                {

                    string soapAPIData =
System.IO.File.ReadAllText(fileWSDLPath); // Чтение содержимого
restAPI.json

                    try
                    {
                        JToken selectedTrader =
FindTraderByIdInJSON(soapAPIData, Traderid);

                        if (selectedTrader != null)
                        {
                            var tmp = new ExternalWSDLUrl
                            {
                                url =
selectedTrader["url"].ToString(),
                                method =
selectedTrader["method"].ToString()
                            };

                            jsonWSDLFromFile = tmp;
                        }
                        else
                        {
                            Console.WriteLine($"WSDL-объект с
TraderId {Traderid} не найден.");


                        }
                    }
                    catch (JsonReaderException ex)
                    {
                        Console.WriteLine($"Ошибка при чтении
JSON: {ex.Message}");
                    }
                    catch (Exception ex)
                    {
                        Console.WriteLine($"Ошибка:
{ex.Message}");
                    }




            }

            #endregion <---- Ищем нужный зарегестрированный API-
сервис ---->
```

```
            // Если это rest то создаём http client и отправляем
данные но уже без id

            if (jsonRestFromFile != null)
            {
                // Проверяем, является ли HttpMethod равным
"POST"
                if (jsonRestFromFile.HttpMethod.Equals("POST",
StringComparison.OrdinalIgnoreCase))
                {
                    string result = await
SendDataToRestAPI(jsonRestFromFile.Endpoint, requestData);
                    if (result == "OK")
                    {
                        return Ok(); // Возвращаем "Ok" в случае
успеха
                    }
                    else
                    {
                        return StatusCode(500, result); //
Возвращаем ошибку с соответствующим сообщением
                    }
                }
                else
                {
                    Console.WriteLine("Метод HTTP не
поддерживается.");
                }
            }
            else if(jsonWSDLFromFile != null)
            {
                SendToWSDLSerivice(jsonWSDLFromFile.url,
jsonWSDLFromFile.method, requestData);
            }

            // Если wsdl, то динамический код надо

            return StatusCode(500, "Не найденно");
        }
 }
```

Реалізації класу CertificationCenter.

```
public class CertificationCenter
    {
        private Dictionary<string, X509Certificate2>
certificates;
        private HttpClient httpClient;

        public CertificationCenter()
        {
```

```
            certificates = new Dictionary<string,
X509Certificate2>();
            httpClient = new HttpClient();
        }

        public async Task<X509Certificate2> RegisterKey(string
key, string serverUrl)
        {
            // Генерируем сертификат для ключа
            X509Certificate2 certificate =
GenerateCertificate(key);

            // Сохраняем сертификат в центре сертификации
            certificates.Add(key, certificate);

            // Отправляем сертификат нашего сервера
            await SendCertificateToServer(certificate,
serverUrl);

            // Возвращаем сертификат
            return certificate;
        }

        private X509Certificate2 GenerateCertificate(string key)
        {
            // Создаем объект RSA для ключей
            using (RSA rsa = RSA.Create())
            {
                // Генерируем случайный идентификатор
сертификата
                string certificateId =
Guid.NewGuid().ToString();

                // Создаем запрос на сертификат
                CertificateRequest request = new
CertificateRequest(
                    new X500DistinguishedName($"CN=Certificate
for key: {key}"),
                    rsa,
                    HashAlgorithmName.SHA256,
                    RSASignaturePadding.Pkcs1);


                // Создаем самоподписанный сертификат
                X509Certificate2 certificate =
request.CreateSelfSigned(DateTimeOffset.Now,
DateTimeOffset.Now.AddYears(1));

                // Возвращаем сертификат
                return certificate;
            }
        }
```

```
        private async Task
SendCertificateToServer(X509Certificate2 certificate, string
serverUrl)
        {
            // Конвертируем сертификат в байты
            byte[] certificateBytes =
certificate.Export(X509ContentType.Cert);

            // Отправляем сертификат нашего сервера
            var content = new
ByteArrayContent(certificateBytes);
            var response = await httpClient.PostAsync(serverUrl,
content);

            if (response.IsSuccessStatusCode)
            {
                Console.WriteLine("Certificate sent to
server.");
            }
            else
            {
                Console.WriteLine("Failed to send certificate to
server.");
            }
        }
```

ДОДАТОК Б

Довідка про впровадження

ДОВІДКА

про впровадження інформаційної системи

На основі універсальних інтерфейсів взаємодії інформаційних систем при застосуванні кросплатформних програмних компонентів яка була розроблена студентом Одеського національного університету імені І.І. Мечникова **Сергатим Євгеном Юрійовичем** під час виконання дипломної роботи бакалавра, побудовано перший реліз інтегрованої крос-платформної інформаційної системи збору та обробки заявок фермерів і формування пакету документів для роботи зернового трейдера.

Данна система прийнята до опитної експлуатації.

Директор «

Подписано цифровой подписью:
Олександр Максимов
Дата: 2023.06.16 21:30:33 +03'00'