КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп’ютерних наук та кібернетики
Кафедра інтелектуальних програмних систем

Кваліфікаційна робота

на здобуття освітнього рівня магістра

за спеціальністю 121 Інженерія програмного забезпечення

на тему:

РОЗРОБКА SAAS ДОДАТКУ ДЛЯ СТВОРЕННЯ «NO-CODE» РІШЕНЬ АВТОМАТИЗАЦІЇ ДЛЯ БІЗНЕСУ

Виконав студент 2-го курсу магістратури
Дмитро КУНЦЬО
____
(підпис)

Науковий керівник:
асистент, кандидат фіз.-мат наук
Костянтин ЖЕРЕБ
____
(підпис)

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

Студент __________
(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри інтелектуальних програмних систем
« 10 » травня 2023 р.,
протокол № 9
Завідувач кафедри
Олександр ПРОВОТАР __________
(підпис)

Київ – 2023

РЕФЕРАТ

Обсяг роботи 70 сторінок, 27 ілюстрації, 1 таблиця, 27 джерел посилань.

ВЕБ РОЗРОБКА, КОДОГЕНЕРАЦІЯ, «ЧИСТА» АРХІТЕКТУРА, ANGULAR, ASP.NET CORE, AWS, DOCKER, IONIC, LLM, LOW-CODE, NO-CODE, PYTHON, ROSLYN, SEQ2SEQ.

Об’єктом роботи є розробка платформи для автоматизації малого та середнього бізнесу.

Предметом роботи є програмний продукт, що автоматизує ділянку певної галузі та налаштовується без написання додаткового коду.

Метою роботи є проектування та розробка додатку для автоматизації бізнесу малого та середнього сегменту, використовуючи засоби кодогенерації, такі як шаблони, рефлексія та моделі штучного інтелекту.

Інструменти розробки: Visual Studio 2022 Community, Visual Studio Code, мова програмування C#, Python, TypeScript, Angular, Ionic, Docker, Amazon Web Services, Langchain, HuggingFace.

Були отримані наступні результати: виконано загальний огляд проектування архітектури програмних додатків. Спроектовано та розроблено додаток, що налаштовується без написання додаткового коду. Розроблено механізм кодогенерації з використанням моделі штучного інтелекту, для якої проведено тонке налаштування з урахуванням специфіки предметної галузі.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ 5

ВСТУП 6

РОЗДІЛ 1 РІШЕННЯ АВТОМАТИЗАЦІЇ БІЗНЕСУ НА ОСНОВІ ГЕНЕРАЦІЇ КОДУ ТА ВЕБ-ЗАСТОСУНКІВ 8

1.1 «Low-code», «No-code» системи 8

1.2 Мова опису доменної специфікації DSL 12

1.3 Кодогенерація 15

1.3.1 Рефлексія 21

1.3.2 Natural Language Processing (NLP) 24

1.4 Архітектура Web додатку 30

РОЗДІЛ 2 РЕАЛІЗАЦІЯ ВЕБ ДОДАТКУ 36

2.1 Уточнена постановка задачі 36

2.2 Архітектура додатку 36

2.2.1 Domain Models компонента 40

2.2.2 RabbitMQ компонента 40

2.2.3 AuthService компонента 40

2.2.4 RealTimeCommunicator компонента 41

2.2.5 SocialMessagingService компонента 41

2.2.6 ComponentExplorer компонента 41

2.2.7 DataRepository компонента 42

2.2.8 WorkflowEngine компонента 44

2.3 Генератор доменних моделей та клієнтського коду 46

2.3.1 DSL для кодогенерації моделей 46

2.3.2 Сервіс для кодогенерації доменних моделей 49

2.3.3 DSL для кодогенерації клієнтських сторінок 51
2.3.4 Додаток, що генерує клієнтські сторінки з DSL 53
2.3.5 Клієнтський інтерфейс для кодогенераторів 54
2.3.6 Сервіс для перекомпіляції моделей та оновлення структури сторінок 57
РОЗДІЛ 3 ВИКОРИСТАННЯ ШТУЧНОГО ІНТЕЛЕКТУ ДЛЯ КОДОГЕНЕРАЦІЇ 59
3.1 Постановка задачі 59
3.2 Загальна реалізація 59
3.3 Підготовка датасету 60
3.4 Вибір методу тренування 61
3.5 Тонке налаштування моделі 62
3.6 Оцінювання моделі 63
3.7 Інтеграція моделі з веб додатком 65
ВИСНОВКИ 67
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 68

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

API – Application Programming Interface, прикладний програмний інтерфейс;

AWS – Amazon Web Services, хмарний сервіс Amazon;

CRM – Customer relationship management, управління відносинами з клієнтами;

DSL – Domain Specific Language, мова специфічного домену;

EC2 – Amazon Elastic Compute Cloud, сервіс хмарних гнучких обчислень;

EF – Entity Framework, фреймворк сутностей;

ERP – Enterprise resource planning, планування ресурсів підприємства;

FT – Fine-Tuning, тонке налаштування;

HTTP – HyperText Transfer Protocol, протокол передачі гіпертекстових документів;

HTTPS – HTTP Secure, безпечний HTTP;

IDE – Integrated Development Environment, інтегроване середовище розробки;

IP – Internet Protocol, інтернет протокол;

JSON – JavaScript Object Notation, запис об’єктів JavaScript;

LINQ – Language Integrated Query, запити, інтегровані в мову;

LLM – Large Language Model, велика мовна модель;

NLP – Natural Language Processing, обробка природньої мови;

REST – Representational State Transfer, передача репрезентативного стану;

SSL – Secure Sockets Layer, рівень захищених сокетів.

ВСТУП

Оцінка сучасного стану об'єкта розробки. Сьогодні з кожним днем зростає попит на різного роду послуги та товари. Звідси слідує велике зростання нових підприємств, особливо в сегменті малого та середнього бізнесу.

Важливо, що у всіх цих підприємствах зазвичай є певні бізнес процеси, за допомогою яких бізнес є стабільним та має змогу розвиватись. Для автоматизації цих процесів, власники часто замовляють створення програмних продуктів, які можуть покрити їх потреби. Проте розробка нових продуктів не дешеве задоволення, тому є велика потреба в додатках, що дозволяють власникам налаштовувати їх функціональність та бізнес логіку додатків за допомогою гнучкого клієнтського інтерфейсу та без написання додаткового коду.

Актуальність роботи та підстави для її виконання. Тема є надзвичайно актуальною, власники підприємств намагаються зменшити кількість задіяних ресурсів в своєму бізнесі для підняття рівня прибутку, за допомогою автоматизації їх бізнес процесів. В цьому зараз надзвичайно корисними є додатки для автоматизацій бізнесу, такі як системи управління відносинами з клієнтами (CRM) чи системи планування ресурсів підприємства (ERP), проте здебільшого вони доволі дорогі і розраховані на сегмент великого бізнесу. Також важливо, щоб при запровадженні даної системи власник бізнесу не оплачував додаткову розробку під конкретно його сферу діяльності. На ринку на даний момент кількість таких систем росте, проте оскільки вони знаходяться на початкових стадіях розробки, то їх функціональність не відповідає очікуванням клієнтів та користувачів.

Мета й завдання роботи. Метою кваліфікаційної роботи є створення програмного продукту для автоматизації бізнес підприємств малого та середнього сегменту, з можливістю налаштування додатку без додаткової розробки. Для досягнення цієї мети поставлено такі завдання:

- дослідити існуючі додатки для автоматизації бізнесу, що відповідають підходам «Low-code»/«No-code» на ринку;
- розробити технічне завдання до продукту;
- розробити легко масштабований додаток;
- проаналізувати методи генерації програмного коду;
- розробити програмне забезпечення для генерації програмного коду;
- розробити модель штучного інтелекту для кодогенерації з природньої мови.

Об’єктом даної роботи є розробка платформи для автоматизації малого та середнього бізнесу.

Предметом дослідження є програмний продукт, що автоматизує ділянку певної галузі та налаштовується без написання додаткового коду.

В якості інструменту створення програмного засобу було обрано Visual Studio 2022 Community – інтегроване середовище розробки мовою програмування C#, яке є – безкоштовним і вільно поширюваним від корпорації Microsoft.

C# – мова, яка є об’єктно-орієнтованою, та є популярним вибором для розробки веб додатків.

Angular – фреймворк для розробки клієнтської частини веб додатку.

Python – мова програмування, що використовувалась для тренування моделі штучного інтелекту.

Можливі сфери застосування. Програмний продукт, розроблений в рамках даної роботи, може бути застосованим в різних сферах бізнесу, таких як онлайн комерція, маркетинг, управлінням різного роду ресурсами.

РОЗДІЛ 1 РІШЕННЯ АВТОМАТИЗАЦІЇ БІЗНЕСУ НА ОСНОВІ ГЕНЕРАЦІЇ КОДУ ТА ВЕБ-ЗАСТОСУНКІВ

1.1 «Low-code», «No-code» системи

«Low-code» і «No-code» системи – це нові технології, які дозволяють створювати програмне забезпечення без необхідності в написанні коду [1]. Ці рішення стають все більш популярними в сучасному світі, оскільки вони дозволяють створювати програми значно швидше і ефективніше. Також вони зменшують залежність від висококваліфікованих розробників, що забезпечує збільшення продуктивності та зниження затрат.

«Low-code» і «No-code» системи часто включають графічний інтерфейс, що дозволяє розробникам взаємодіяти з додатком шляхом перетягування та розміщення блоків, що репрезентують функції додатка. «Low-code» і «No-code» системи підтримують інтеграцію з іншими системами та сервісами, що дозволяє розробникам легко підключити свій додаток до інших сервісів, таких як соціальні мережі, платіжні системи та інші. Також, більшість «Low-code» і «No-code» систем мають вбудовану автоматизацію, що дозволяє розробникам швидко створювати складні додатки з мінімальними зусиллями. Наприклад, система може автоматично створити таблицю в базі даних або зберегти деякі дані у зовнішньому джерелі. Окрім цього, такі системи зазвичай мають вбудовані функції перевірки безпеки, які дозволяють забезпечити безпеку додатку.

З джерела [1] історія розвитку «Low-code» і «No-code» починається з 1980-х років з виникненням RAD (Rapid Application Development) систем, які дозволяють швидко розробляти програми з мінімальним програмуванням без необхідності висококваліфікованих розробників. У 2002 році компанія Salesforce створила першу систему «Low-code» для створення корпоративних застосунків. З тих пір

багато компаній випустили свої власні «Low-code» і «No-code» системи, і вони стали більш доступними для користувачів з різних галузей.

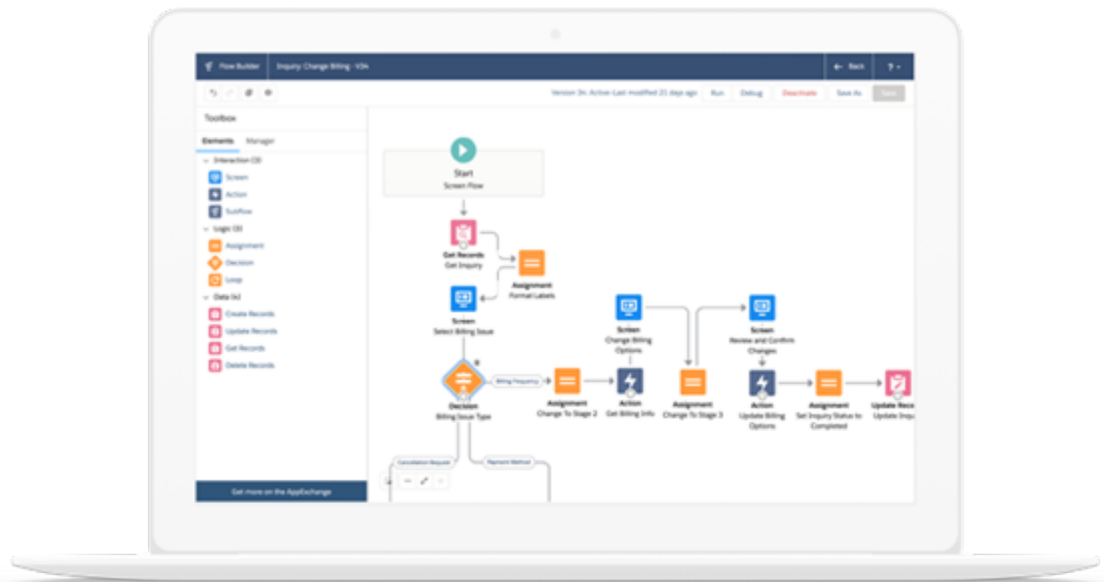

Рисунок 1.1 – Клієнтський інтерфейс продукту Salesforce

На рисунку 1.1 ми можемо бачити сучасну Salesforce систему з використання «flow-chart» діаграм для побудови необхідного бізнес-процесу для CRM системи під запити клієнту.

На сьогоднішній день на ринку існує багато таких систем, кожна з яких має свої переваги та недоліки. Нижче наведено приклади декількох таких систем:

– Microsoft PowerApps [2] – ця система дозволяє створювати додатки для платформи Microsoft. Вона має зручний інтерфейс та дозволяє інтегруватися з багатьма іншими сервісами. Однак, вона обмежується платформою Microsoft, тому не може бути використана для створення додатків для інших платформ;
– Google App Maker [3] – ця система дозволяє створювати додатки для Google Workspace. Вона також має багато вбудованих функцій та зручний інтерфейс, проте, вона обмежується платформою Google, і також не може бути використана для створення додатків для інших платформ;

- Zoho Creator [4] – ця система дозволяє створювати додатки для різних платформ, включаючи веб та мобільні додатки. Вона дозволяє інтегруватися з багатьма іншими сервісами, однак, вона має обмежену кількість функцій порівняно з іншими системами;
- Appian [5] – ця система має широкий спектр можливостей, включаючи інтеграцію з різними джерелами даних, мобільні додатки та інше. Вона дозволяє створювати додатки для різних платформ. Проте, ця система є однією з найбільш дорогих на ринку;
- Bubble [6] – ця система дозволяє створювати веб-додатки без необхідності в програмуванні. Вона має зручний інтерфейс та дозволяє інтегруватися з багатьма іншими сервісами. Однак, ця система має обмежену функціональність та не підходить для створення складних додатків.

Загалом, вибір «Low-code» і «No-code» системи залежить від потреб користувача та його технічних знань. Деякі системи, такі як Microsoft PowerApps та Google App Maker, підходять для користувачів, які вже працюють з платформою Microsoft або Google, тоді як інші системи, такі як Zoho Creator та Appian, підходять для користувачів, які шукають широкий спектр можливостей та інтеграцію з різними сервісами.

Найкраще застосовувати «Low-code» і «No-code» систем для швидкого розроблення прототипів та додатків, які не потребують складного програмування. Ці рішення можуть бути корисними для невеликих бізнесів та стартапів, які швидко хочуть випустити свій продукт на ринок. Ось деякі галузі та сфери в яких можна використовувати ці системи:

- розробка внутрішніх програм – багато компаній використовують «Low-code» і «No-code» системи для створення внутрішніх програм, таких як системи управління відпустками, ведення списків та інші. Це дозволяє їм збільшити продуктивність та зменшити витрати на розробку;

– створення веб-додатків – це може бути корисним для невеликих бізнесів, які не мають ресурсів для наймання повноцінних розробників;
– мобільна розробка – «Low-code» і «No-code» системи дозволяють розробникам швидко створювати мобільні додатки для різних платформ;
– аналітика та звіти – «Low-code» і «No-code» системи можуть бути використані для створення систем аналітики та звітів, що допомагають компаніям збирати та аналізувати дані.

Ці технології стають все більш популярними завдяки своїй простоті та ефективності [7]. «Low-code» і «No-code» системи є дуже важливим інструментом для розвитку програм та додатків у сучасному світі. Вони дозволяють швидко реагувати на зміни в ринку та потреби користувачів, забезпечуючи ефективну та швидку розробку програм та додатків без необхідності в глибоких знаннях програмування. Також вони дозволяють залучати до розробки програм більше людей, які раніше не мали можливості розвивати свої ідеї через відсутність необхідного досвіду. І дозволяють розробникам програмного забезпечення та бізнес-аналітикам більш ефективно працювати разом, тому що немає необхідності в навчанні всіх учасників команди програмуванню.

Проте, незважаючи на те, що «Low-code» і «No-code» системи можуть бути корисними для багатьох різних галузей, вони не є універсальним рішенням для всіх задач. Вони можуть мати обмеження щодо функціональності та інтеграції з різними сервісами. Також, деякі з них можуть бути дуже дорогими для користувачів з обмеженим бюджетом.

Незважаючи на недоліки, в останні роки зростання «Low-code» і «No-code» платформ посилилося через брак кваліфікованих розробників програмного забезпечення та необхідністю покращити час виконання проектів розробки, щоб можна було швидше вирішувати бізнес-проблеми.

1.2 Мова опису доменної специфікації DSL

У сучасному світі, де швидкість технологічного розвитку стрімко зростає, необхідно швидко та ефективно розробляти програмне забезпечення. Одним з інструментів, що дозволяє значно спростити та прискорити процес розробки, є мови опису доменної специфікації DSL [8].

DSL – це мова специфічна для домену, яка призначена для опису певної області знань або домену [8]. Вона дозволяє ефективно моделювати та описувати складні системи та процеси у межах обраного домену, що робить її дуже корисною для розробки програмного забезпечення.

DSL може бути створена для вирішення різних завдань, від тестування програмного забезпечення до автоматизації бізнес-процесів. Вона дозволяє зменшити кількість коду та підвищити його читабельність, що робить DSL більш доступною для фахівців з інших областей.

В залежності від специфічних потреб та вимог області, для якої розробляється DSL, існують різні типи DSL. Їх можна поділити на текстові та графічні, внутрішні та зовнішні. Розглянемо детальніше кожен тип [8]:

- текстові DSL зазвичай використовують спеціальні мови програмування для опису вимог та специфікацій домену. Вони забезпечують можливість зручної взаємодії з текстовим редактором та можуть бути ефективно використані для розробки програмного забезпечення в різних областях;
- графічні DSL використовують графічні елементи та діаграми для представлення вимог та специфікацій домену. Це дозволяє використовувати DSL з меншою кількістю коду та забезпечує більш інтуїтивний підхід до розробки програмного забезпечення;
- внутрішні DSL розробляються для конкретної галузі знань, забезпечують зручну та просту розробку програм з даними, які належать до цієї галузі знань;

– зовнішні DSL використовуються для розв'язання конкретної проблеми, яка може виникнути в галузі знань. Зовнішні DSL можуть бути створені на основі внутрішніх DSL або на основі стандартних мов програмування.

Використання DSL має свої переваги та недоліки, які важливо розумітти для правильного вибору між DSL та загальною мовою програмування при розробці програмного забезпечення.

З переваг DSL можна виділити:

– ефективність – DSL дозволяє швидко та ефективно описувати складні системи та процеси в межах обраного домену;
– читабельність – DSL дозволяє зменшити кількість коду та підвищити його читабельність, що полегшує розуміння коду для фахівців з інших областей;
– інтеграція – DSL забезпечує зручну інтеграцію з існуючми системами, тому що вона спеціально створена для роботи з даними, які належать до певної галузі знань;
– простота – DSL дозволяє легко створювати програми, спрощуючи процес розробки програмного забезпечення;
– розширюваність – DSL дозволяє легко додавати нові функції, що відповідають потребам користувачів з певної галузі знань.

Недоліки DSL включають:

– обмеження – DSL може бути обмежена в здатності до опису деяких аспектів системи та процесів;
– вартість – розробка та підтримка DSL може бути дорогим процесом;
– спеціалізація – DSL може бути призначена тільки для певної області та може бути непридатною для інших завдань.

В загальному, DSL може бути застосована в різних областях, для програмування вбудованих систем, розробки бізнес-додатків, створення документації, тестування, моделювання, автоматизації процесів та багато іншого.

Розглянемо декілька реалізованих прикладів використання DSL:

- Ruby on Rails [9] – це фреймворк для веб-розробки. В ньому використовується внутрішній DSL для створення веб-додатків;
- Hadoop [10] – програмна платформа для організації розподіленого зберігання і обробки наборів великих даних. Використовує внутрішній DSL для розподіленої обробки даних. Це дозволяє програмістам легко виконувати складні завдання обробки даних та забезпечує ефективну розподілену обробку даних;
- SQL [11] – мова програмування, для роботи з реляційними базами даних. SQL є внутрішнім DSL для роботи з базами даних;
- HTML [12] – мова розмітки, що використовується для створення веб-сторінок. HTML є зовнішнім DSL для створення веб-сторінок.

DSL може бути створено за допомогою різних інструментів та технологій, таких як ANTLR, Xtext, MPS та багато інших [13]. Кожен з цих інструментів має свої переваги та недоліки, тому вибір того, який саме інструмент використати, залежить від потреб проекту та власних навичок програмістів.

Основні елементи DSL включають в себе наступне [8], [13]:

- синтаксис – визначає, які слова, символи та правила можуть використовуватися для побудови виразів, інструкцій та конструкцій мови. Для кожної DSL визначається свій синтаксис, який може бути природним для конкретної доменної області;
- семантика – визначає, який зміст мають побудовані на основі мови вирази, інструкції та конструкції. Семантика визначається в залежності від конкретної доменної області та завдань, які потрібно розв'язати за допомогою DSL;
- лексика – визначає, які слова та символи можуть використовуватися в мові. Вона може бути обмеженою або необмеженою, залежно від конкретної доменної області;

– граматика – визначає, як можуть бути побудовані вирази, інструкції та конструкції мови. Граматика має формальний характер та визначається в залежності від синтаксису та семантики мови;
– транслятор – це програмний засіб, який перетворює вихідний код DSL в код мови загального призначення. Транслятор може бути реалізований у вигляді компілятора, інтерпретатора, транслятора в байт-код або іншого програмного засобу.

Ці елементи можуть бути взаємопов'язані та залежати один від одного. Для створення ефективної DSL необхідно враховувати конкретні потреби та вимоги доменної області, для якої вона розробляється.

Загалом, DSL є потужним інструментом для програмістів та розробників програмного забезпечення, який дозволяє створювати програмне забезпечення, яке відповідає специфічним потребам користувачів та галузей знань. Нові технології та розвиток штучного інтелекту дозволяють програмістам створювати DSL з більшою легкістю та ефективністю. Використання DSL може знизити час розробки програмного забезпечення, покращити його якість та забезпечити ефективну роботу з даними.

1.3 Кодогенерація

Кодогенерація – це процес автоматичної генерації програмного коду з використанням певних алгоритмів та інструментів. Цей процес є дуже важливим для створення програм з максимальною швидкістю та ефективністю.

Ідея автоматичної генерації програмного коду з'явилася ще в 1950-х роках, коли було розроблено перші комп'ютери та мови програмування. У цей період кодогенерація була дуже простою та базувалася на генерації коду з використанням статичних шаблонів.

Згодом, з розвитком комп'ютерів та програмного забезпечення, підходи до кодогенерації стали більш складними та ефективними. У 1970-х роках з'явилися

перші інструменти для автоматичної генерації коду з використанням мов програмування, таких як Lisp та Prolog [14].

У 1990-х роках з'явилися перші інструменти для автоматичної генерації коду з використанням графічного інтерфейсу користувача (GUI), такі як Visual Basic та Delphi. Ці інструменти дозволяли розробникам швидко створювати програми за допомогою перетягування та розміщення різних елементів інтерфейсу користувача.

У 2000-х роках з'явилися нові IDE – Eclipse та NetBeans, що зокрема містять інструменти для автоматичної генерації коду, які дозволяють розробникам швидко та ефективно створювати програмне забезпечення з використанням різних мов програмування.

Один із найпопулярніших підходів на якому базується кодогенерація – це використання шаблонів та генерації коду на основі цих шаблонів. Основний принцип кодогенерації полягає в тому, що розробник визначає шаблон для генерації коду та вказує параметри, які можуть бути змінені. Після цього код генерується автоматично з використанням вказаних параметрів. Іншим підходом є використання API для створення чи модифікації коду.

Можна виділити такі основні компоненти що використовуються під час кодогенерації [14]:

- шаблони (templates) – це структуровані блоки коду, які можна використовувати для створення нового коду. Шаблони зазвичай містять різні параметри та інструкції, які дозволяють генерувати різний код, залежно від потреб користувача;
- генератори (generators) – це програмні засоби, які використовують шаблони для автоматичної генерації коду. Генератори можуть бути написані на різних мовах програмування та використовувати різні технології для генерації коду;

- метамоделі (metamodels) – це моделі, які визначають структуру та характеристики коду, який генерується. Метамоделі зазвичай описують сутності, атрибути та взаємодії між ними;
- мови моделювання (modeling languages) – це мови, які використовуються для створення метамоделей та описування коду, який генерується. Різні мови моделювання можуть мати свої сильні та слабкі сторони залежно від конкретного завдання;
- редактори моделей (model editors) – це програмні засоби, які дозволяють створювати та редагувати моделі, які використовуються для генерації коду. Редактори моделей можуть включати різні функції, такі як перевірку правильності моделі, автодоповнення, підказки та інші.

Існує декілька підходів до кодогенерації, включаючи генерацію коду з використанням моделей, макро-процесорів, автоматичну генерацію коду з використанням мов програмування, графічних інтерфейсів користувача, шаблонів, та схем даних.

Генерація коду з моделі – це процес автоматичної генерації програмного коду з вже створених моделей, таких як UML та SysML. Цей підхід дає змогу швидко та ефективно створювати програмне забезпечення, забезпечуючи високу якість та точність коду.

Одним зі стандартів для моделювання систем є мова моделювання UML (Unified Modeling Language). UML зазвичай використовується для моделювання поведінки та структури програмного забезпечення. Генерація коду з UML-моделі може бути виконана з використанням різних підходів [4]:

- за допомогою використання інструментів, які автоматично генерують код зі структури UML-моделі. Ці інструменти зазвичай використовують шаблони коду, які відповідають структурі UML-моделі, і відтворюють цю структуру в програмному коді. Однак, цей

підхід може бути досить обмеженим, оскільки він може не забезпечити достатньої гнучкості при генерації програмного коду;

– використання машинного навчання та штучного інтелекту для генерації коду з моделі. В цьому підході модель UML моделюється як вхідні дані для глибинної нейронної мережі, яка навчається генерувати програмний код зі структури моделі. Цей підхід може забезпечити більшу гнучкість та точність генерації коду.

Іншим стандартом для моделювання систем є SysML [15]. SysML (Systems Modeling Language) – це розширення мови UML, яке використовується для моделювання систем з використанням методології системного інженерінгу і дозволяє моделювати системи з більш високим рівнем абстракції. SysML використовується для моделювання систем, які складаються з різних компонентів та взаємодіють з іншими системами. Генерація коду з SysML-моделі може бути виконана з використанням тих саме підходів, що і для генерації коду з UML-моделі.

Але, окрім цього, генерація коду з SysML-моделі може базуватися на підходах до моделювання, які відрізняються від традиційних підходів до генерації коду з UML-моделі.

У SysML використовуються блокові діаграми для моделювання систем та їх компонентів. Ці блоки представляють окремі частини системи, а їх взаємодії моделюються з використанням діаграм взаємодії. Генерація коду з такої моделі може включати автоматичне створення коду для кожного блоку системи та генерацію коду для їх взаємодії.

Іншим способом генерації коду з SysML-моделі є використання підходу, який базується на генерації коду з UML-моделі з використанням засобів, які вміють працювати з SysML. У такому випадку SysML-модель перетворюється на UML-модель, яка потім використовується для генерації програмного коду. Цей підхід може бути використаний для систем, які мають складність, що перевищує можливості SysML.

Окрім генерації програмного коду, генерація коду з моделі може також використовуватися для генерації тестових сценаріїв. Така генерація може забезпечити автоматичне створення тестових сценаріїв на основі вхідних даних та поведінки системи, яку необхідно протестувати.

Загалом, процес генерації коду з моделі зазвичай починається зі створення моделі, в якій описуються всі компоненти системи та їх взаємодії. Потім з моделі генерується програмний код з використанням підходів, описаних вище. Після генерації коду розробник може перевірити його на правильність та знайти можливі помилки або неточності.

Окрім генерації коду з моделі, існують і інші підходи для кодогенерації, які згадувалися вище.

Макро-процесори є стандартними інструментами для генерації коду з використанням шаблонів. Вони дозволяють визначати макроси, які замінюються на фрагменти коду. Тоді кодогенератор може використовувати цей макро для генерації коду з використанням параметрів, переданих йому. Це сприяє зручності та швидкості генерації коду, однак макро-процесори мають свої обмеження, такі як обмежену функціональність та складність відлагодження.

Інший підхід до кодогенерації полягає в використанні мов програмування для автоматичної генерації коду. Цей підхід зазвичай використовується для створення коду на високорівневих мовах програмування з використанням шаблонів та параметрів.

Один з прикладів такого підходу є мова програмування Python зі своїм фреймворком Django, який має вбудовані генератори коду для автоматичного створення моделей, форм та представлень.

Ще одним підходом до кодогенерації є використання графічних інтерфейсів користувача. Цей підхід зазвичай використовується для генерації коду для мобільних додатків та веб-сторінок з використанням drag-and-drop інтерфейсів та візуального програмування.

Один з прикладів – сервіс Appery.io [16], який дозволяє розробникам створювати мобільні додатки з використанням графічного інтерфейсу та вбудованих генераторів коду.

Далі, генерація коду за допомогою шаблонів є підходом, що полягає у створенні шаблонів коду, які можуть бути заповнені даними, що отримуються з інших джерел. Цей підхід зазвичай використовується для створення повторюваного коду, такого як генерація коду для створення таблиць баз даних, валідації форм та інших елементів інтерфейсу користувача.

Інший підхід до генерації коду полягає в використанні схем даних, які описують структуру даних, що використовуються в програмному забезпеченні. За допомогою цього підходу можна згенерувати код для створення баз даних, класів, структур та інших елементів програми.

При виборі підходу до розробки, необхідно також розуміти переваги та недоліки кодогенерації. Основними перевагами кодогенерації є:

- швидкість – кодогенерація дозволяє швидко створювати програми без необхідності писати кожен рядок коду вручну, що зменшує час та зусилля, необхідні для написання коду;
- коректність – код, згенерований з використанням кодогенерації, знижує кількість помилок та забезпечує високу точність програмного коду;
- гнучкість – використовувати різні методології розробки програмного забезпечення та розширювати функціональність програмного забезпечення шляхом редагування моделі.

Проте, існують також деякі недоліки генерації коду з моделі:

- налаштування – потребує певної кількості часу та зусиль для налаштування інструментів кодогенерації;
- оптимізація – може привести до створення погано оптимізованого або неправильної структури програмного коду;

– складність редагування – у випадку коли генератор лише створює код, а його модифікації є відповідальністю розробника, згенерований код може бути складним для редагування, особливо якщо він генерується зі складних моделей. Проте у випадку підтримки модифікації коду генератором цей недолік нівелюється.

У будь-якому випадку, генерація коду з моделі стає все більш популярною в світі програмування і використовується в багатьох галузях, включаючи програмування вбудованих систем, автономні автомобілі та інші індустрії [14]. Також, мови програмування постійно розвиваються. Це означає, що для забезпечення сумісності програмного забезпечення з новими та «застарілими» API та мовними конструкціями необхідні регулярні проекти технічного обслуговування. А згенерований код лише вимагає налаштування генератора коду, а не цілого програмного забезпечення.

1.3.1 Рефлексія

Рефлексія – це механізм, за допомогою якого програма може отримувати доступ до внутрішніх властивостей та структури свого власного коду під час виконання. Це дозволяє програмі здійснювати динамічний аналіз та модифікацію свого коду під час виконання.

В більшості сучасних мов програмування, таких як Java, C# та Python, є рефлексивні можливості. Рефлексія дозволяє програмі отримувати доступ до класів, методів, властивостей та інших об'єктів свого коду в режимі виконання. За допомогою рефлексії можна динамічно створювати об'єкти, викликати методи, зчитувати та змінювати значення властивостей.

Основна ідея рефлексії полягає в тому, що програма може звертатися до метаданих, які описують структуру коду, замість безпосереднього доступу до конкретних об'єктів. Наприклад, можна отримати доступ до класу і його методів за

допомогою імені класу, а не самого об'єкта класу. Це дозволяє програмі створювати об'єкти з коду, який не був відомий на момент компіляції [17].

Рефлексія може бути використана для багатьох різних задач [17]:

- для динамічного завантаження класів, які необхідні під час виконання програми. Наприклад, якщо програма потребує певний клас, але його немає у вихідному коді програми, вона може завантажити клас за допомогою рефлексії;
- для створення нових об'єктів, які не були відомі на момент компіляції коду. Це може бути корисно в деяких випадках, коли потрібно створити новий об'єкт на основі даних, які отримані з іншої програми або з бази даних;
- для отримання інформації про клас, такої як його ім'я, методи, поля та конструктори. Це може бути корисно, якщо програма потребує додаткової інформації про клас;
- для виклику методів та доступу до полів об'єктів, навіть якщо вони є приватними. Це може бути корисно, коли потрібно дізнатися про деякі властивості або викликати методи класу, який не був відомий на момент написання коду;
- використання рефлексії для створення анотацій, які можуть бути використані для позначення коду та визначення додаткових властивостей для класів, методів та полів;
- для серіалізації та десеріалізації об'єктів. Це означає, що програма може зберігати стан об'єктів у файлі або передавати через мережу, а потім відновлювати об'єкти з цих даних. Для цього програма використовує метадані, щоб отримати доступ до структури об'єкта та його властивостей;
- для валідації вхідних даних. Наприклад, програма може перевіряти, чи відповідають вхідні дані очікуваному формату або типу даних, використовуючи метадані класів.

Рефлексія також може бути використана для оптимізації в тому випадку, коли неможливо виконати деякі операції в статичний спосіб, або коли вони вимагають виконання додаткових обчислень протягом виконання програми.

Нижче наведено декілька способів, які можна використовувати для оптимізації за допомогою рефлексії [17]:

- кешування результату та кешування раніше знайдених методів. Цей спосіб використання рефлексії полягає у збереженні раніше знайдених методів та кешування результатів виклику. Це дозволяє уникнути повторних викликів методів та зменшити час, потрібний для виконання програми;
- використання кешування значень констант. Якщо програма використовує значення констант, вони можуть бути закешовані для покращення продуктивності. Наприклад, можна створити мапу для збереження констант та використовувати цю мапу замість того, щоб викликати методи в рантаймі для отримання значень констант.

Однак, слід зазначити, використання рефлексії може мати і свої недоліки:

- це може впливати на продуктивність програми, оскільки вона вимагатиме більше обчислювальних ресурсів та затримок. Проте, вплив на продуктивність може бути дуже різним в залежності від конкретного застосування рефлексії;
- деякі операції, пов'язані з рефлексією, можуть бути небезпечними з точки зору безпеки. Наприклад, доступ до приватних полів та методів може бути використаний для зловживання та порушення безпеки програми.

У загальному, використання рефлексії може бути корисним інструментом, який дозволяє динамічно взаємодіяти з класами та об'єктами в процесі виконання програми. Однак, варто пам'ятати про можливість негативного впливу на продуктивність та безпеку програми та використовувати рефлексію тільки тоді, коли це дійсно необхідно. Якщо правильно використовувати рефлексію, вона може

стати корисним інструментом для розробки програмного забезпечення та підвищення її гнучкості та розширюваності.

1.3.2 Natural Language Processing (NLP)

Natural Language Processing (NLP) – це галузь комп'ютерних наук, яка фокусується на взаємодії між людьми та комп'ютерами, використовуючи природну мову. NLP займається розробкою алгоритмів та моделей, які дозволяють комп'ютерам розуміти, інтерпретувати та генерувати людську мову.

NLP використовується для розв'язання різноманітних завдань, пов'язаних з мовою, таких як машинний переклад, аналіз тональності, виявлення іменованих сутностей, автоматична класифікація тексту, генерація тексту та багато іншого.

З джерела [18], історія виникнення NLP сягає середини 20 століття, коли почалися перші спроби комп'ютерної обробки природної мови. На той час використовувалися переважно правила та евристики, щоб зрозуміти та згенерувати текст.

У 1950-х роках було розроблено перші мовні моделі, зокрема, модель Маркова, що дозволила розробити перші програми машинного перекладу.

У 1960-х роках розвиток NLP став більш інтенсивним. Були розроблені перші системи з розпізнавання мови та машинного перекладу. Однак, через обмежену потужність комп'ютерів та обробки об'ємних даних, досягнення були обмеженими.

У 1970-х роках були розроблені нові методи та алгоритми, такі як алгоритм Кокона, який дозволяє швидко знаходити синоніми та інші стилістичні відмінності у текстах.

У 1980-х роках були розроблені перші системи з аналізу та синтезу мови, які здатні інтерпретувати та генерувати текст за допомогою граматик.

У 1990-х роках з'явилися перші комерційні продукти для розпізнавання та синтезу мови, а також віртуальні асистенти, які використовують NLP для розуміння та відповіді на запити користувачів.

У наш час, з розвитком глибокого навчання та великих даних, NLP стає все більш потужним та ефективним інструментом. Сучасні NLP-моделі здатні розпізнавати та генерувати текст з високою точністю та ефективністю, відкриваючи нові можливості для застосування у різних галузях

Наразі у галузі обробки природньої мови (NLP) існує безліч моделей, які використовуються для різних задач, таких як машинний переклад, класифікація тексту, генерація тексту, розпізнавання іменованих сутностей, аналіз тональності, розпізнавання мовленнєвих актів та інших.

Деякі з найбільш відомих моделей NLP:

- Recurrent Neural Networks (RNN) – це моделі, які можуть обробляти послідовності довільної довжини, використовуючи зв'язки між попередніми і поточними вхідними значеннями. RNN добре підходять для задач, які вимагають розуміння послідовності;
- Convolutional Neural Networks (CNN) – це моделі, які використовуються для обробки зображень, але їх також можна використовувати для обробки текстової інформації, якщо текст відображається у вигляді матриці. CNN добре підходять для класифікації тексту та відповідей на запитання;
- Transformer-based Models – це моделі, які використовують механізми уваги та трансформера для обробки послідовностей. Трансформерні моделі, такі як BERT, GPT та T5, демонструють дуже хорошу продуктивність у багатьох задачах NLP, зокрема в машинному перекладі, класифікації тексту та генерації тексту;
- Seq2Seq – це моделі, які приймають послідовність як вхід та генерують іншу послідовність як вихід. Seq2Seq моделі добре підходять для машинного перекладу, генерації тексту та відповідей на запитання;

- Generative Adversarial Networks (GAN) – це моделі, які використовуються для генерації тексту та інших типів даних, які мають розподіл. GAN складається з двох моделей: генератора та дискримінатора. Генератор намагається створити нові дані, які схожі на навчальні дані, а дискримінатор намагається розрізнити між навчальними даними та згенерованими даними;
- Recursive Neural Networks (Tree-based Models) – це моделі, які працюють зі структурованими даними, такими як дерева та графи. Recursive Neural Networks добре підходять для задач, які вимагають аналізу структури тексту, таких як відношення між різними частинами речення.

Це не повний список моделей, які використовуються в NLP. Більшість з них мають велику кількість варіацій та додаткові функції, які роблять їх більш спеціалізованими для різних завдань та даних.

Розглянемо детальніше моделі Seq2Seq та «Трансформер» моделі [20].

Моделі Seq2Seq є одними з найбільш популярних моделей глибокого навчання, використовуваних в задачах обробки природної мови, таких як машинний переклад, стенограма до мовлення та генерація тексту.

Принцип роботи моделей Seq2Seq полягає в тому, що вони приймають на вхід послідовність даних, таку як рядки тексту або звукові сигнали, та перетворюють їх у вихідну послідовність. Наприклад, модель машинного перекладу може приймати англійський текст та генерувати відповідний переклад на іншу мову.

Основна структура моделей Seq2Seq складається з двох основних компонентів: енкодера та декодера (рис. 1.2) [19]. Енкодер отримує послідовність вхідних даних та перетворює її в вектор фіксованої довжини, який представляє семантику цієї послідовності. Цей вектор, що називається контекстним вектором, передається на вхід декодеру. Декодер отримує контекстний вектор та використовує його для генерації вихідної послідовності.

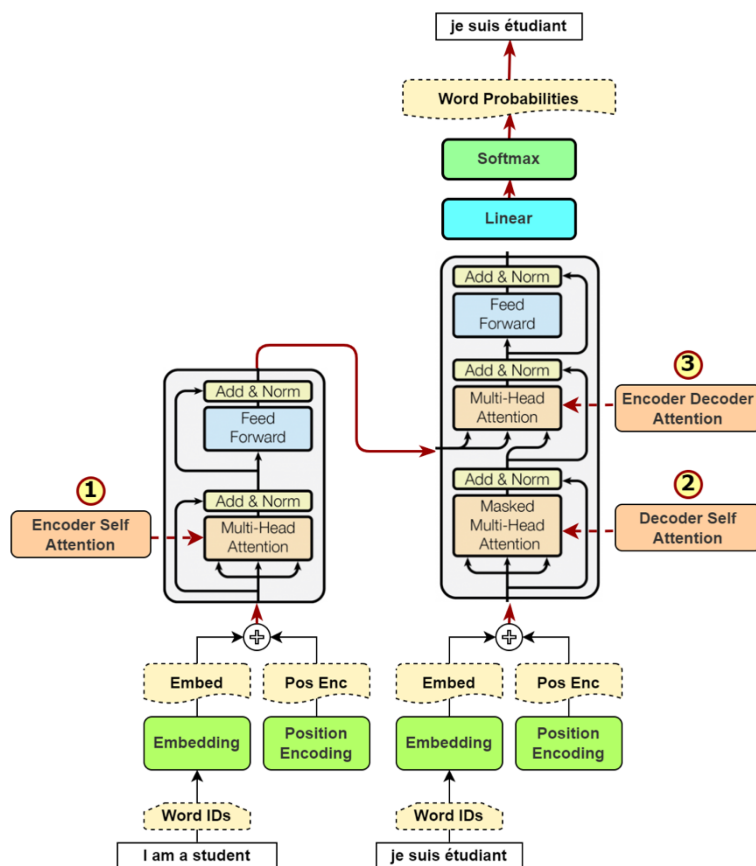

Рисунок 1.2 – Архітектура моделі Seq2Seq [19]

Енкодер та декодер зазвичай реалізуються за допомогою рекурентних нейронних мереж (RNN), зокрема, довготривалої пам'яті (LSTM) та глибоких RNN (GRU). Кожен елемент послідовності подається на вхід енкодеру по черзі, і кожен раз він оновлює свій стан, використовуючи інформацію з попередніх елементів. Після проходження всіх елементів, енкодер видає останній стан, який інтерпретується як контекст, тобто вектор фіксованої довжини, що представляє семантику всієї послідовності.

Після цього декодер приймає на вхід спеціальний токен – початок послідовності і починає генерувати вихідну послідовність, використовуючи контекст та попередні згенеровані елементи.

На кожному кроці декодер використовує контекст і попередні згенеровані елементи для генерації наступного елемента вихідної послідовності. Цикл

продовжується доти, поки не буде згенерований токен – кінець послідовності або не буде досягнуто максимальної довжини вихідної послідовності.

Під час навчання Seq2Seq моделей використовується метод зворотного поширення помилки (backpropagation), щоб мінімізувати різницю між фактично згенерованою послідовністю та очікуваною послідовністю.

Використання Seq2Seq моделей не обмежується лише машинним перекладом. Їх також успішно використовують для генерації тексту, виконання завдань відповіді на запитання, згортання тексту та вибірки, навіть для генерації музики та зображень. Seq2Seq моделі є одними з найбільш потужних і універсальних моделей в галузі обробки природних мов (NLP) та знань з розуміння мови (NLU).

Розглянемо наступну модель. Трансформери (Transformers) – це новий тип нейромережевих моделей в області обробки природної мови, який з'явився в 2017 році. Вони використовуються для різних задач обробки природної мови, таких як машинний переклад, розпізнавання іменованих сутностей, відповіді на запитання, генерація тексту та багато інших.

Трансформерні моделі працюють на основі механізму уваги (self-attention) [19], який дозволяє моделі фокусуватись на найбільш важливих частинках вхідного тексту. Вони складаються з кількох енкодерів та декодерів, кожен з яких містить блоки механізмів уваги, що дозволяють моделі "думати" над великою кількістю тексту одночасно.

Трансформерні моделі мають декілька переваг порівняно з іншими моделями, такими як LSTM та RNN. Одна з основних переваг трансформерів полягає в тому, що вони можуть працювати з послідовностями будь-якої довжини, без обмежень на кількість кроків, які можна зробити під час проходження нейромережі. Крім того, трансформери можуть ефективно використовувати паралельні обчислення на багатоядерних процесорах та GPU.

Серед трансформерних моделей можна виділити такі, як BERT (Bidirectional Encoder Representations from Transformers), GPT (Generative Pre-trained Transformer), RoBERTa (Robustly Optimized BERT Pretraining Approach), T5 (Text-to-Text Transfer Transformer), XLNet та інші [19]. Кожна з цих моделей має свої особливості та застосування в різних задачах обробки природної мови:

- BERT був випущений в 2018 році компанією Google і показав вражаючі результати на багатьох завданнях в обробці природньої мови. BERT базується на двохзв'язному кодувальному процесорі, який може використовуватись для виконання багатьох завдань в обробці природньої мови;
- GPT був випущений компанією OpenAI і використовується для генерації тексту. GPT навчається на великому корпусі тексту, що дозволяє йому здійснювати передбачення і генерацію тексту з високою якістю;
- Transformer-XL – розроблена для роботи з послідовнісними даними, такими як текстові рядки. Transformer-XL використовує механізми змінної довжини послідовностей для ефективного моделювання залежностей між словами в тексті.

Інші модифікації трансформерних моделей, такі як RoBERTa, ALBERT, T5 та інші, покращують результати на різних завданнях в обробці природньої мови.

Трансформерні моделі зазвичай використовуються в задачах кодогенерації з двох підходів: заснованих на правилах та заснованих на даних.

Підхід, заснований на правилах, використовує заздалегідь встановлені правила, щоб визначити, який код має бути згенерований для заданого тексту. Наприклад, можна визначити правило, що якщо в тексті згадується функція "sort", то генерується код, який сортує вказані дані. Трансформерна модель використовується для перекладу тексту в код згідно з встановленими правилами.

Підхід, заснований на даних, використовує трансформерну модель для навчання на великому наборі текстів та відповідних кодів. Модель навчається

здійснювати відповідність між текстом та кодом, щоб можна було згенерувати код на основі заданого тексту. При цьому, трансформерна модель використовується для здійснення перекладу з тексту в код.

Для використання трансформерних моделей в кодогенерації використовують Seq2Seq моделі, описані раніше. Однак, замість рекурентних нейронних мереж, в яких зазвичай використовувалися LSTM або GRU шари, використовуються трансформери.

У цьому підході, вхідний код представляється в форматі послідовності токенів, а вихідна послідовність є кодом мови програмування. Таким чином, для кодогенерації, трансформерна модель приймає послідовність токенів, яка представляє код на вхід, і генерує послідовність токенів, що відповідає коду мови програмування на виході.

1.4 Архітектура Web додатку

Веб-додатки є однією з найбільш поширених технологій в Інтернеті, і їх використовують у різних сферах, включаючи бізнес, соціальні мережі та онлайн-ігри. Для успішної реалізації веб-додатків необхідно визначити архітектуру, яка відповідає потребам користувачів та вимогам системи. Розглянемо основні архітектурні типи веб-додатків:

– клієнт-серверна архітектура – у цій архітектурі клієнтська частина забезпечує інтерфейс для користувача, а серверна частина забезпечує обробку запитів та доступ до баз даних. Основною перевагою цієї архітектури є можливість розділення функціональності між клієнтською та серверною частинами, що дозволяє збільшити швидкість та продуктивність додатку. Крім того, така архітектура забезпечує більшу масштабованість та забезпечує можливість розвитку додатку у майбутньому;

– архітектура односторінкових додатків – полягає в тому, що весь контент веб-додатку завантажується одним разом при першому завантаженні, а потім додаток працює без перезавантаження сторінок. Ця архітектура може бути реалізована за допомогою фреймворків, таких як Angular та React. Вона забезпечує більш швидкий та плавний досвід користувача, оскільки не потребує перезавантаження сторінок при кожному запиті. Крім того, ця архітектура сприяє більшій масштабованості та забезпечує зниження навантаження на серверну частину додатку;
– мікросервісна архітектура: полягає у розбитті веб-додатку на окремі функціональні блоки, які називаються мікросервісами. Кожен мікросервіс відповідає за виконання однієї конкретної функції, і мікросервіси можуть бути розгорнуті на різних серверах. Ця архітектура забезпечує більшу гнучкість та масштабованість додатку, оскільки можливо змінювати та оновлювати окремі мікросервіси без впливу на решту системи. Крім того, така архітектура забезпечує більшу надійність та забезпечує швидкість розгортання нових функцій.

Архітектура веб-додатків – це ключовий елемент у розробці високопродуктивних, безпечних та масштабованих додатків. При виборі архітектури веб-додатку, слід враховувати такі фактори, як масштабованість, продуктивність, безпеку, взаємодію з користувачем та інші потреби проекту. Важливо також розглянути можливості використання новітніх технологій, середовищ та фреймворків для забезпечення найбільш ефективного та прогресивного досвіду користувача.

Розглянемо основні принципи «чистої» архітектури, якими керуватимемось, в процесі створення веб додатку [21]. Для розуміння чистої архітектури, потрібно спочатку розібратись, що таке архітектура програмного забезпечення в цілому.

Архітектура програмної системи – це форма, яка надається системі її творцями. Ця форма утворюється поділом системи на компоненти, їх організацією

та визначенням способів взаємодій між ними. Мета форми – спростити розробку, розгортання і супровід програмної системи, що міститься в ній. Головна стратегія такого спрощення в тому, щоб якомога довше мати якомога більше варіантів. Головне призначення архітектури – підтримка життєвого циклу системи. Гарна архітектура робить систему легкою в освоєнні, простою в розробці, супроводі і розгортанні. Кінцева її мета – мінімізувати витрати протягом терміну служби системи і максимізувати продуктивність програміста [21].

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

- гексагональна архітектура (Hexagonal Architecture, також відома як архітектура – портів і адаптерів), розроблена Алістером Кокберн і описана Стівом Фріманом і Натом [22];
- DCI (дані, контекст, взаємодія), запропонована Джеймсом Копліеном і Трюгве-Реенскаугом [23];
- BCE (границі, контроль, модель), запропонована Іваром Якобсоном [24].

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

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

- незалежність від фреймворків – архітектура не залежить від наявності будь-якої бібліотеки. Це дозволяє розглядати фреймворки як інструменти, замість того щоб намагатися втиснути систему в їх рамки;
- простота тестування – бізнес-правила можна тестувати без користувацького інтерфейсу, бази даних, веб-сервера і будь-яких інших зовнішніх елементів;

- незалежність від призначеного для користувача інтерфейсу – призначений для користувача інтерфейс можна легко змінювати, не зачіпаючи решту системи. Наприклад, веб-інтерфейс можна замінити консольним інтерфейсом, не міняючи бізнес-правил;
- незалежність від бази даних – можливість поміняти Oracle або SQL Server на Mongo, BigTable, CouchDB або щось ще. Бізнес-правила не прив'язані до бази даних;
- незалежність від будь-яких зовнішніх агентів – бізнес-правила нічого не знають про інтерфейси, що ведуть до зовнішнього світу.

Приклад «чистої» архітектури наведений на рисунку 1.3. Концентричні кола представляють різні рівні програмного забезпечення. Чим ближче до центру, тим вище рівень. Зовнішні кола – це механізми. Внутрішні – політики. Головним правилом, що призводить цю архітектуру в дію, є правило залежностей (Dependency Rule): залежності в вихідному коді повинні бути спрямовані всередину, в сторону високорівневих політик. Ніщо у внутрішньому колі нічого не знає про зовнішні колах. Наприклад, імена, оголошені в зовнішніх колах, не повинні згадуватися в коді, що знаходиться у внутрішніх колах. Сюди відносяться функції, класи, змінні і будь-які інші іменовані елементи програми. Точно так формати даних, оголошені в зовнішніх колах, не повинні використовуватися у внутрішніх, особливо якщо ці формати генеруються фреймворком в зовнішньому колі. Ніщо в зовнішньому колі не повинно впливати на внутрішні кола.

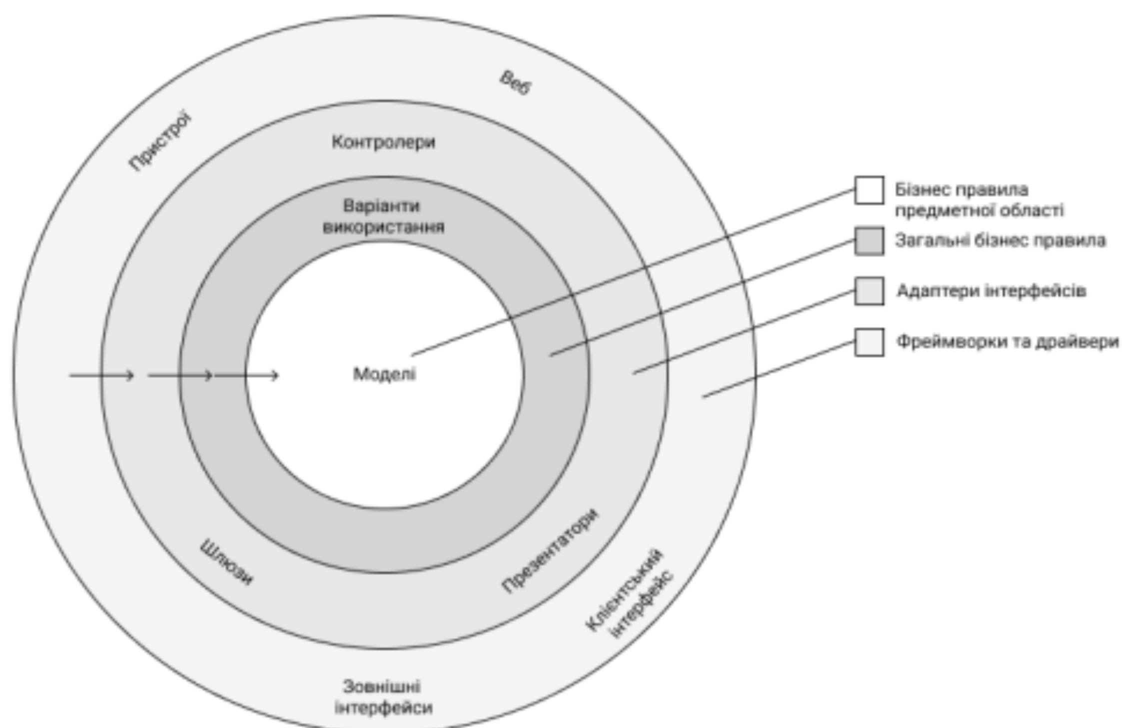

Рисунок 1.3 – «Чиста» архітектура додатку

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

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

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

Самий зовнішній рівень моделі на рисунку 1.3 зазвичай складається з

фреймворків і інструментів, таких як база даних і веб-фреймворк. Як правило, для цього рівня потрібно писати не дуже багато коду, і зазвичай цей код грає роль сполучної ланки з наступним внутрішнім світом.

Кола на рисунку 1.3 лише схематично зображують основну ідею: іноді може знадобитися більше чотирьох кіл. Фактично немає ніякого правила, який стверджує, що кіл має бути саме чотири. Але завжди діє правило залежностей. Залежності в вихідному коді завжди повинні бути спрямовані всередину.

РОЗДІЛ 2 РЕАЛІЗАЦІЯ ВЕБ ДОДАТКУ

2.1 Уточнена постановка задачі

Метою даної розробки є задоволення потреби користувачів у легкому способі створення платформи для управління власним бізнесом, та автоматизації процесів в ньому. Власник бізнесу, котрий виступає клієнтом даного продукту, повинен мати змогу описати доменну модель свого бізнесу за допомогою зручного клієнтського інтерфейсу. Після чого в замовника має бути можливість налаштувати клієнтську частину свого додатку, а саме розташування різних полів для вводу даних, розташування таблиць з даними, налаштування бізнес правил для обов'язковості, відображення чи фільтрації даних. Також у клієнта повинна бути можливість налаштувати бізнес процеси його компанії без написання коду. Архітектура додатку повинна дотримуватись принципів «чистої» архітектури та не бути залежною від будь-якого хмарного сервісу.

2.2 Архітектура додатку

Розробка «low-code» системи для автоматизації бізнесу потребує створення додатків, які можуть ефективно виконувати завдання без необхідності глибокого розуміння програмування. Для досягнення цієї мети була розроблена архітектура додатку, яка дозволяє максимально гнучко налаштовувати та розширювати його функціональність. Більше того, цей додаток можна легко супроводжувати та розміщувати в хмарі без необхідності прикладання додаткових зусиль.

При проектуванні веб-додатку були використані всі принципи чистої архітектури, що дозволило досягти максимальної ефективності та легкості розробки. Архітектура додатку, представлена на рисунках 2.1 та 2.2, є результатом ретельного аналізу та оптимізації всіх компонентів системи. Вона дозволяє використовувати різноманітні технології та рішення для досягнення оптимальних

результатів. Зокрема, до архітектури входять такі компоненти, як база даних, веб-сервер, клієнтська частина та бізнес-логіка, що забезпечує зручне та просте використання додатку.

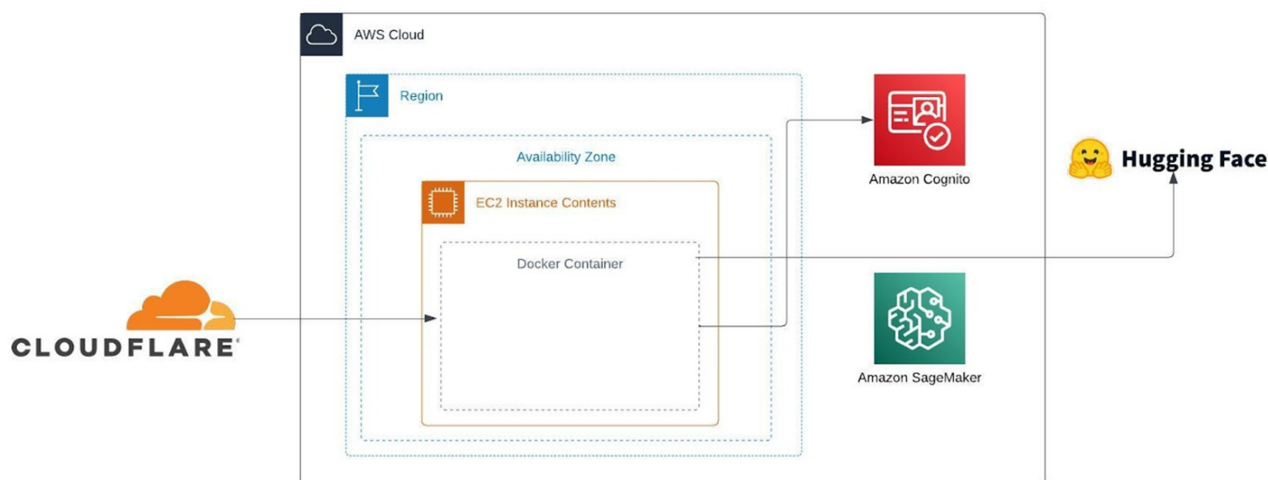

Рисунок 2.4 – Архітектура розміщення додатку в хмарі

Розглянемо більш детально архітектуру зображену на рисунку 2.1. Для розташування додатку в хмарі використовується хмарний сервіс Amazon Web Service (AWS). З постановки задачі та вимог до системи додаток створювався як максимально незалежний від хмарних сервісів (cloud agnostic), тому використання сервісів хмарного провайдера має максимально відповідати цьому принципу, тобто використання будь-яких сервісів має бути максимально обмежено, а вся інфраструктура додатку описана структурно в коді додатку. Проте для хостингу додатку потрібно все ж використовувати сервіс, що надає доступ до віртуальних машин, тобто для даного додатку використовується є сервіс EC2, що дає доступ до віртуальних машин на яких можна з легкістю розташовувати додатки.

Оскільки додаток обслуговує невелику кількість користувачів та для заощадження коштів, додаток розташовується на одній віртуальній машині в одному регіоні. Тобто, не існує інших віртуальних машин для горизонтального масштабування чи для забезпечення доступності у випадку відмови першої машини. Проте, варто зазначити, що для розгортання інфраструктури додатку

використовується принцип «Інфраструктура як код (infrastructure as a code)», а саме вся інфраструктура описана за допомогою Docker файлів та nginx файлів конфігурації, що дає змогу дуже легко та швидко розгортати додаток.

Як було вже зазначено, вся інфраструктура описана за допомогою Docker файлів, які згодом використовуються для запуску Docker контейнерів на віртуальних машинах. Оскільки контейнеризація не залежить від машини на якій вона запускається, а потрібна лише наявність Docker та підтримки віртуалізації, це дає змогу в майбутньому легко розгортати контейнери в будь-якому іншому хмарному сервісі чи навіть локально для розробки та тестування.

Проте, в даній архітектурі легко помітити порушення принципу незалежності від хмарного сервісу, що представляє AWS сервіс Cognito. Цей сервіс використовується для зберігання користувачів та їхньої автентифікації в додатку. Це єдина залежність, яка присутня в додатку на конкретного хмарного провайдера, використовується вона тому, що це був чи не перший сервіс який розроблявся, і для швидкої реалізації було прийнято рішення використати саме AWS Cognito, оскільки воно надає зручний інтерфейс для взаємодії та зрозумілий інтерфейс для налаштування. Проте, далі з більш детальної архітектури сервісу авторизації стане зрозуміло, що в коді не присутні прямі залежності на цей сервіс, всі вони знаходяться за добре спроектованими інтерфейсами, що дає змогу в майбутньому легко підмінити реалізацію на більш незалежний від хмарного сервісу метод, як IdentityService.

Також в рамках AWS використовується сервіс SageMaker. Додаток не залежить від цього сервісу, оскільки він використовувався для тренування та оцінювання моделі для кодогенерації.

За рамками AWS, використовується сервіс HuggingFace, що дає змогу розташовувати модель та взаємодіяти з нею через зручний інтерфейс.

Також використовується сервіс CloudFlare, що відіграє роль брандмауера третього покоління, тобто передбачає захист додатку від різного роду мережевих атак.

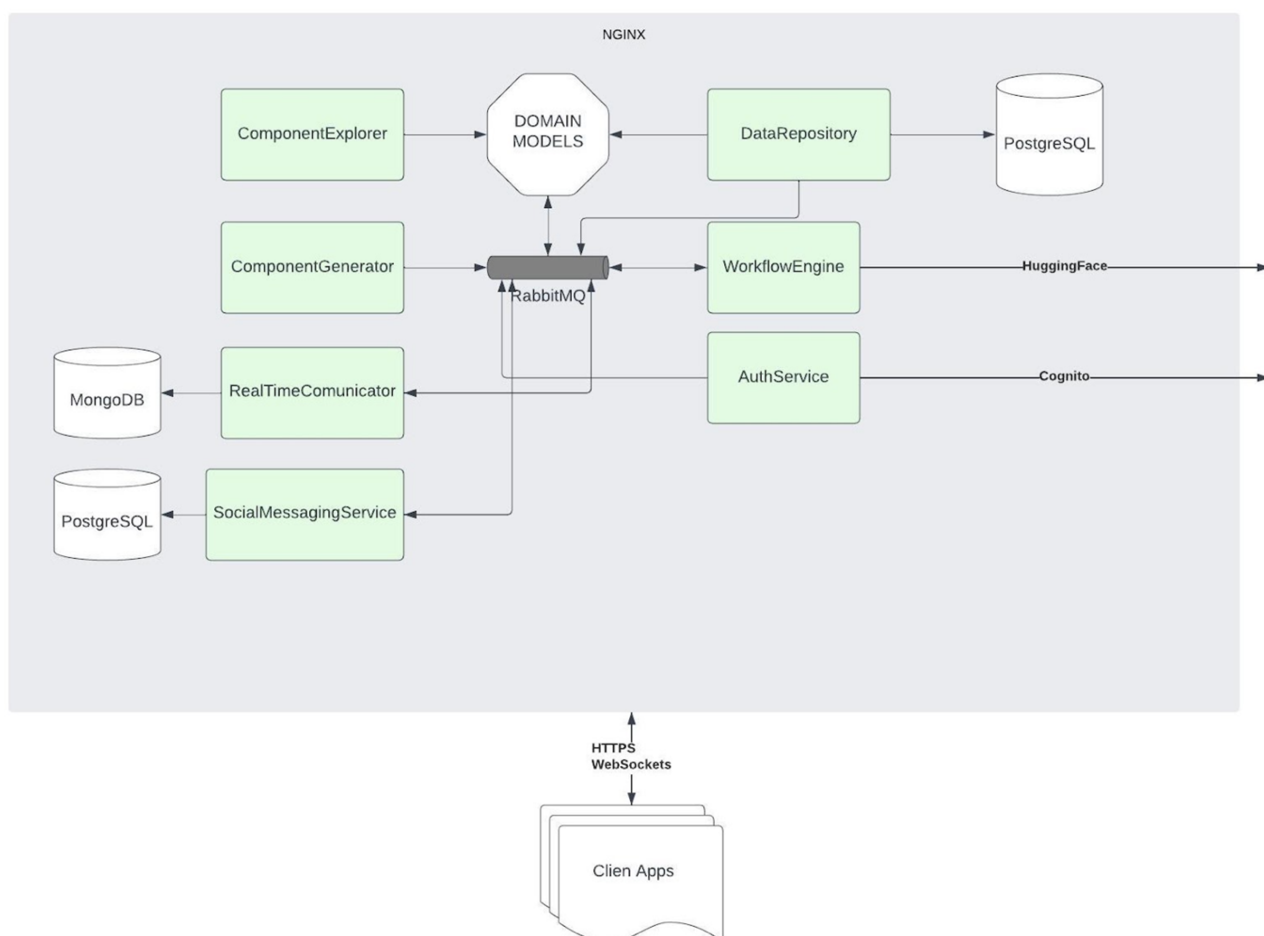

Рисунок 2.5 – Архітектура компонент додатку

На рисунку 2.2, представлено діаграму яка більш детально описує архітектура самого додатку, що розміщується в Docker контейнері. Кожен зелений блок представляє окремий сервіс, а з допомогою Docker контейнеризації кожен цей сервіс може бути винесений в окремий контейнер, зо дасть змогу горизонтально масштабувати окремий сервіс. На даний момент, завантаження додатку не є високим, тому для заощадження коштів та ресурсів всі сервіси розміщуються в одному контейнері та використовується технологія nginx, що відіграє роль

своєрідної точки входу в додаток та в майбутньому може давати змогу балансувати між декількома сервісами.

Розглянемо більш детально кожен з компонент даної діаграми.

2.2.1 Domain Models компонента

Бачимо що основною компонентою є Domain Models який відповідає за бізнес логіку проекту, всі направлення залежностей (стрілки) направлені в сторону до цієї компоненти вона є ядром системи, все інше деталі, які легко можуть бути замінені на інші. Саме ця компонента зберігає код для всіх моделей в додатку, які генеруються.

2.2.2 RabbitMQ компонента

В якості центрального хабу для спілкування між різними сервісами була використана технологій брокеру повідомлень, що дає змогу асинхронно спілкуватись сервісами. В якості брокеру був використаний застосунок RabbitMQ. Програмна взаємодія з брокером відбувається за допомогою бібліотеки MassTransit, що дає змогу легко створювати та розширювати приймачів повідомлень. Також дана бібліотеки підтримує інших брокерів, тому брокер не є важкою залежністю і може бути легко замінений на інший.

2.2.3 AuthService компонента

AuthService – це сервіс, що відповідає за авторизацію та автентифікацію. В даній реалізації продукту використовується AWS Cognito сервіс для роботи з авторизацією та автентифікацію. Для взаємодії з даним сервісом використовувались інтерфейси представлені AWS CDK. Для кожного користувача зберігається інформація про його ім’я, електронну адресу, зашифрований пароль, ролі до яких він належить та додаткова інформація що може бути корисною для

бізнес логіки додатку. Всі ці дані зберігаються в AWS Cognito, який гарантує безпеку даних.

Рольова модель, що використовується для авторизації є плоскою, тобто не ієрархічною. Ролі використовуються в подальшому для визначення доступу до інформації, чи сервісів.

2.2.4 RealTimeCommunicator компонента

Дана компонента відповідає за спілкування між додатком та клієнтською частиною в режимі реального часу, тобто ця компонента забезпечує підтримку протоколу WebSockets. Також компонента дає змогу використання технології WebPush для відправки повідомлень з серверу на клієнт в режимі реального часу. Для зберігання релевантної інформації використовується документо-орієнтована NoSQL база даних MongoDB.

2.2.5 SocialMessagingService компонента

Дана компонента забезпечує користувача можливістю використовувати внутрішній чат додаток або ж Twitter подібний сервіс для спілкування з іншими користувачами чи коментування певних подій. Збереження повідомлень відбувається в базі даних PostgreSQL, проте варто зазначити, що архітектура даного сервісу була написана з дотриманням всіх вимог DDD (Domain Driven Design), тому база даних яка є частиною інфраструктури можу бути легко підмінена на іншу.

2.2.6 ComponentExplorer компонента

Сервіс що конвертує код моделей в DSL опису цих моделей в JSON форматі. Надалі опис моделей використовується в різних частинах додатку, для кодогенерації клієнтського інтерфейсу, чи при новій кодогенерації моделей.

2.2.7 DataRepository компонента

DataRepository – це компонент який відповідає за доступ до даних. Ядровий сервіс здатен сам вибирати з якою базою даних він хоче працювати, ця інформація передається до компоненти даних, яка в результаті розуміє з якою базою даних вона працюватиме. Таким чином ми не залежимо від якоїсь конкретної бази даних. На даний момент продукт підтримує PostgreSQL та MSSQL бази даних. Для доступу до даних також був розроблений контракт для фільтрації та вибору певних колонок, і для цього контракту був створений динамічний будівельник, що перетворював об’єкт цього контракту в LINQ запит, після чого LINQ запит конвертується в SQL запит за допомогою EF. Компонента підтримує два контракти для доступу до даних. Перша версія використовувала JSON формат, для опису запитів до сервісу. Приклад такого об’єкту представлено на рисунку 2.3. Як бачимо це структурований запит, в якому користувач має змогу описати фільтри, які він хоче застосувати до даних, чи вказати які дані він хотів би отримати. Мінусом даного контракту є те що він є досить об’ємним, що сповільнює передачу даних в мережі, а також що з таким контрактом неможливо використовувати метод GET в HTTP протоколі, адже такий контракт може бути переданий тільки в тілі запиту. Це є вагомим мінусом, адже неможливість використання GET методу виключає певні базові функції в браузері як кешування, що знову робить систему більш повільною.

Саме тому був розроблений новий контракт, версія 2, що представлений на рисунку 2.4. Як бачимо він набагато компактніший, та дозволяє використовувати себе як параметр запиту, а не тіло. Можна помітити, що даний контракт досить схожий на контракт що використовується в GraphQL, тому варто зазначити чому не був використаний GraphQL. Було оцінено, що імплементація GraphQL та інтеграція уже з існуючим функціоналом займе більше часу чим розробка власної, до того ж розширення власного функціоналу здійснюватиметься набагато простіше, адже GraphQL не передбачає можливості легкої розширюваності.

```json
{
  "filters": {
    "expression": {},
    "logicalOperator": 0,
    "filters": [
      {
        "expression": {
          "columnPath": "Code",
          "value": "adas-tiritaji-un-kondicionieri",
          "comparisonType": 0
        },
        "logicalOperator": 0,
        "filters": []
      }
    ]
  },
  "columns": {
    "columnNames": [
      "Id",
      "Name"
    ],
    "relatedColumns": [
      {
        "columnNames": [
          "Id",
          "Name",
          "Code"
        ],
        "columnName": "Parent",
        "relatedColumns": []
      }
    ]
  },
  "pageSize": 1,
  "pageIndex": 0
}
```


Рисунок 2.6 – Контракт версії 1

```
select=Name,Code,Contact.Name & \
filter=($count(Products) $gte 4 $or Contact.Account.Name $contains 'test') & \
limit=10 & offset=10
```

Рисунок 2.7 – Контракт версії 2

На рисунку 2.5 представлено діаграму класів, що забезпечує кодогенерацію з контракту в код, що може бути виконаний двигуном ORM EF. Бачимо, що при кодогенерації LINQ запитів враховується не тільки запит, а також роль користувача та його права на доступ до даних. Цей кодогенератор підтримує багато різних SQL подібних операцій, таких як вибірка даних, фільтрація, обмеження та агрегація. Також враховується ознака підтримки різних мов додатком, таким чином ті дані що є локалізованими відповідно до певної мови повертаються відповідно до мови запиту.

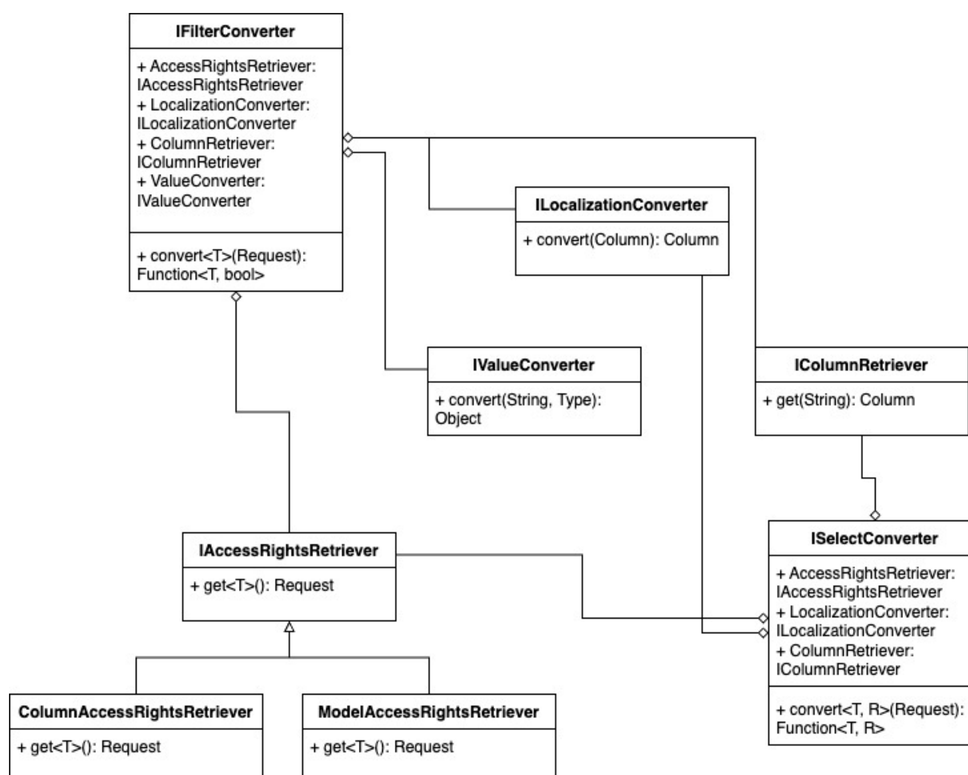

Рисунок 2.8 – Діаграма класів конвертування в LINQ

2.2.8 WorkflowEngine компонента

Дана компонента є важливою оскільки вона дозволяє клієнтам розширювати їх бізнес логіку за допомогою конструктора бізнес процесів. Для імплементації даної компоненти було використано застосунок Elsa [25]. Це публічно доступний проект написаний на C# мові програмування, що дає змогу будувати та виконувати бізнес процеси.

Цей застосунок було успішно інтегровано в додаток, таким чином, що він використовує авторизацію та автентифікацію а також вміє спілкуватись з іншими сервісами за допомогою брокеру повідомлень. Яскравий приклад застосування є події при створенні, оновленні чи видаленні даних. Ці події зазвичай є критичними для бізнес логіки додатків, тому потрібно механізм, щоб користувач мав змогу реагувати на них. Саме ця компонента дає змогу налаштовувати це досить легко використовуючи клієнтський інтерфейс та базове розуміння бізнес процесів.

Програмний процес взаємодії під час реагування на ці події зображено на діаграмі на рисунку 2.6.

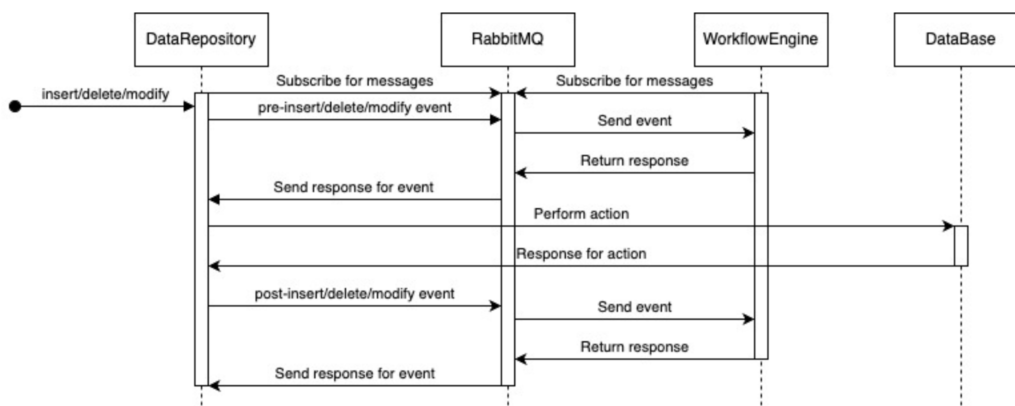

Рисунок 2.9 – Діаграма реагування на подію збереження змін

Бачимо, що під час події компонента DataRepository успішно відправляє повідомлення до брокеру перед подією та після. Після чого компонента WorkflowEngine приймає це повідомлення, та якщо є відповідний процес що очікує на така повідомлення запускається, виконується та результат повертається до брокеру, після чого компонента доступу до даних має змогу отримати відповідь та відповідно до неї вирішити наступні кроки по опрацюванню запиту.

Іншим яскравим прикладом є імплементація різних сервісів для проведення оплати. Оскільки таких сервісів багато, то неможливо розробити гнучкий додаток для всіх, тому використання компоненти бізнес процесів дозволяє імплементувати специфічну клієнтську логіку в рамках процесу, і відповідно не виносити цю логіку до загальних сервісів.

Elsa Workflow є досить могутнім додатком, що дає змогу описувати будь-які бізнес процеси, а також архітектура додатку дозволяє легко розширювати додаток не порушуючи будь-який SOLID принципів. До прикладу, можна створювати процеси що стартують по таймеру (крон завдання), HTTP сервіси, відправляти електронні листи, чи навіть писати код за допомогою Liquid або JavaScript.

Саме ця компонента також є точкою доступу до нетренованої моделі для кодогенерації запитів до компоненти доступу до даних, а саме за допомогою створення HTTP точки входу далі ми обробляємо запит і надсилаємо на HuggingFace, де розташована модель.

2.3 Генератор доменних моделей та клієнтського коду

ComponentGenerator компонента є центральною для даної роботи, адже тема роботи це саме фокус на кодогенерації, за що і відповідає дана компонента. Компонента включає в себе набір сервісів та додатків, а саме:

- сервіс для кодогенерації C# моделей з описаного контракту DSL у форматі JSON;
- клієнтський інтерфейс для користувацького додатку, котрий вміє сам генерувати код на базі фреймворку Angular Ionic з описаного контракту DSL у форматі JSON;
- клієнтський інтерфейс для створення вищезазначених контрактів DSL у форматі JSON;
- сервіс для перекомпіляції моделей, оновлення структури клієнтських сторінок та оновлення всіх сервісів що залежать від моделей без втрати доступу до додатку.

2.3.1 DSL для кодогенерації моделей

Для реалізації генератора моделей, в першу чергу потрібно зробити DSL для представлення моделей. Був обраний синтекс представлення моделей в форматі JSON. На рисунку 2.7 представлено структуру моделі.

```
{
  "tableName": "Page",
  "caption": "Page",
  "isLocalized": true,
  "displayValueField": "RelativeUrl",
  "booleanFieldsStructure": [...],
  "textFieldsStructure": [...],
  "lookupFieldsStructure": [],
  "guidFieldsStructure": [...],
  "numberFieldsStructure": [],
  "dateTimeFieldsStructure": [...],
  "imageFieldsStructure": [],
  "fileFieldsStructure": [],
  "multiCurrencyAmountFieldsStructure": [],
  "entityCollectionsStructure": [...]
},
```

Рисунок 2.10 – Структура моделі

Розглянемо всі атрибути які наявні в даному представленні моделі:

– «tableName» – це назва таблиці в базі даних якій відповідає модель;
– «caption» – це назва моделі, яка буде відображатись користувачу, вони є локалізованою, локалізації зберігаються на сервері;
– «isLocalized» – вказує чи є дана модель локалізованою, тобто чи зберігаємо ми переклади для текстових полів в базі даних;
– «displayValueField» – вказує на назву колонки, яка в моделі є її текстовим представленням (наприклад, для моделі «Місто» в якій є колонка «Ім’я», саме колонка «Ім’я» буде представляти цю модель).

Інші атрибути є переліком колонок різного типу, а саме:

– «booleanFieldsStructure» – перелік колонок булевого типу;
– «textFieldsStructure» – перелік колонок текстового типу;
– «lookupFieldsStructure» – перелік колонок, що мають тип який посилається на інші моделі (наприклад для моделі «Місто» такою колонкою може виступати колонка «Країна»);
– «guidFieldsStructure» – перелік колонок 128-бітного ідентифікатора;
– «numberFieldsStructure» – перелік колонок числового типу;
– «dateTimeFieldsStructure» – перелік колонок, що вказують на дату та/або час;

- «imageFieldsStructure» – перелік колонок, що відображають певну картинку;
- «fileFieldsStructure» – перелік колонок, що відображають певний файл;
- «multiCurrencyAmountFieldsStructure» – перелік колонок, що відображають числове значення та валюту, і перераховується при зміні валюти;
- «entityCollectionsStructure» – перелік колонок які є зворотніми до «lookupFieldsStructure», тобто зберігають інформацію які моделі посилаються на нашу.

Зрозуміло, що у кожного типу колонок є своя структура, розглянемо до прикладу колонки числового типу, структура яких представлена на рисунку 2.8.

```
{
  "minValue": -1.7976931348623157e+308,
  "maxValue": 1.7976931348623157e+308,
  "precision": 14,
  "scale": -1,
  "dbFieldName": "FeeAmount",
  "name": "FeeAmount",
  "caption": "FeeAmount",
  "isReadOnlyField": false,
  "isDisplayValueField": false,
  "isRequiredField": false,
  "requiredFieldErrorMessage": null
},
```

Рисунок 2.11 – Структура колонки числового типу

З рисунка 2.8 видно, що користувач може задавати мінімальне та максимальне значення для даної колонки, може вказувати точність, якщо використовується число з плаваючою точністю. Також є набір стандартних полів, такий як «name» – відображає назву в базі даних чи «isRequiredField» – чи є це поле обов’язковим для заповнення.

2.3.2 Сервіс для кодогенерації доменних моделей

Коли у нас є відповідна DSL, ми можемо приступити до створення генератора коду, що буде перетворювати дану структуру в код мови .Net. Для вирішення даної задачі була обрана бібліотека з відкритим кодом Roslyn, яка є API для компіляції та аналізу коду мови C#.

Підхід при створенні чи видаленні коду моделей є простим, ми просто виконуємо потрібну операцію. Більше труднощів викликає операція модифікації, адже тут є два підходи для її реалізації. Перший – видаляємо старий код і генеруємо новий за новою описаною схемою, цей підхід значно зменшує шанс помилок, адже ми робимо повне заміщення, при цьому так як ми видаляємо і створюємо заново, він працює дещо повільніше чим його конкурент. Другий підхід – робити розумне об’єднання, при цьому потрібно створити певний клас, що буде відповідати за правильне об’єднання схем. Цей підхід зменшує час при оновленні моделі, проте він є складнішим і важчим в розробці. Виходячи з цього було рішення слідувати першому алгоритму, при цьому код написаний так, що в разі чого можна буде легко розширити його до використання другого підходу.

На рисунку 2.9 представлена діаграма послідовності для процесу створення моделі з схеми описаної вище.

Спочатку користувач надсилає запит з заданою моделлю, після чого цей запит обробляється будівельником моделі (ModelBuilder), котрий створює відповідний клас, добавляє для цього класу відповідні атрибути, і за потреби наприклад якщо наша модель повинна бути локалізованою, то надсилає запит на створення моделі локалізацій для даної моделі. Після цього будівельник моделей надсилає всі колонки моделі до будівельника колонок (ColumnBuilder), який генерує код колонок, і так як у колонок можуть бути різні атрибути, наприклад максимальна довжина тексту чи точність дробового числа, будівельник колонок запитує декоратора (ColumnDecorator), для того щоб він додав відповідні атрибути в залежності від опису колонки.

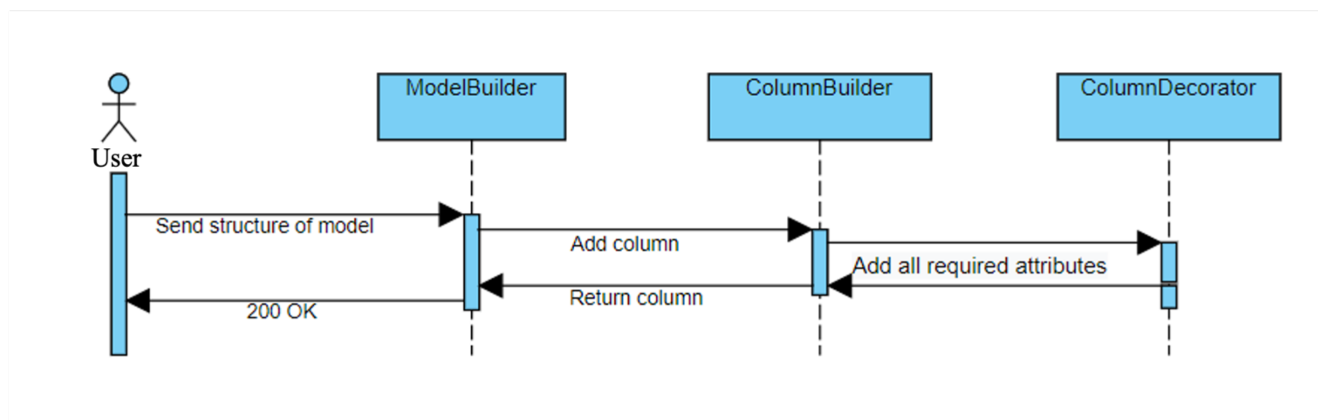

Рисунок 2.12 – Діаграма послідовностей для кодогенерації моделей

На виході отримуємо код однієї чи декількох моделей згенерований мовою C# і який підтримується EF. Код згенерований даним генератором представлений на рисунку 2.10. Як видно з рисунку генератор додає всі необхідні атрибути для класу та елементів класу, а також наслідує клас моделі від правильної базової сутності. Для числових колонок були додані атрибути, котрі описують цю колонку з специфікації наведеної розділом вище. Також генератор успішно додав колонку, що посилається на інший клас в додатку, та згенерував правильне ім'я для неї в базі даних, як ключ що вказує на зовнішню таблицю.

Також варто відмітити, що даний кодогенератор також добавляє всі необхідні using команди, щоб код компілювався.

```
[Table("Guard")]
[ChangeTrackingStore(typeof(GuardChangeTrackingEntity))]
public class GuardEntity : BaseEntity {
    [Display(Name = "GuardQualificationIdCaption")]
    [Required]
    public Guid? GuardQualificationId { get; set;}
    [ForeignKey("GuardQualificationId")]
    [Display(Name = "GuardQualificationCaption")]
    public virtual GuardQualificationEntity GuardQualification { get; set; }

    [Display(Name = "AdditionalInfoCaption")]
    [Column(TypeName = "text")]
    public string AdditionalInfo { get; set; }

    public Guid? SalaryAmountId { get; set; }
    [Range(0, 1000000)]
    [Precision(2)]
    [MultiCurrencyAmount]
    [ForeignKey("SalaryAmountId")]
    [Display(Name = "SalaryAmountCaption")]
    public MultiCurrencyAmountEntity SalaryAmount { get; set; }

    [Display(Name = "CityIdCaption")]
    [Required]
    public Guid? CityId { get; set;}
    [ForeignKey("CityId")]
    [Display(Name = "CityCaption")]
    public virtual CityEntity City { get; set; }
}
```

Рисунок 2.13 – Згенерований код моделі

2.3.3 DSL для кодогенерації клієнтських сторінок

Так само як і для генерації моделей, спочатку створимо DSL для опису клієнтського інтерфейсу. Для прикладу візьмемо частину клієнтського інтерфейсу, що задає форми, котрі використовують дані безпосередньо з моделей, які ми генерували темою вище. На рисунку 2.11 представлений приклад опису такої форми.

```
{
    "name": "dataitem-edit-page",
    "component": "edit-page",
    "parameters": {
        "entity": "DataItem",
        "rules": [...
        ],
        "children": [...
        ]
    }
},
```

Рисунок 2.14 – Структура опису клієнтської сторінки

В даній схемі присутні наступні атрибути:

а) «name» – назва форми;

б) «component» – компонента, що відповідає даній формі;

в) «parameters» – параметри компоненти, що включають в себе:

1) «rule» – бізнес правила на даній сторінці
2) «entity» – назва відповідної моделі для якої ця форма створена;
3) «children» – компоненти, які відображатимуться на даній сторінці, до прикладу це можуть бути колонки чи таблиці з зв’язаними даними.

Для функціоналу бізнес правил була використана бібліотека «json-rules-engine», розглянемо як саме задаються бізнес правила. На рисунку 2.12 представлений синтаксис опису бізнес правила для відображення чи ховання певної колонки.

```
{
    "conditions": {
        "all": [
            {
                "fact": "dataitem-edit-page",
                "operator": "equal",
                "value": "f30188bc-4f0c-4822-85e4-46c8a47c5c5a",
                "path": "$.dataItemType.id"
            },
            {
                "fact": "dataitem-edit-page",
                "operator": "equal",
                "value": "a705cdc1-275f-46d1-8f5f-dfad35ae89e7",
                "path": "$.color.id"
            }
        ]
    },
    "event": {
        "type": "show",
        "params": {
            "target": "dataitem-edit-page.primary-column-group.Heading",
            "triggeredBy": [
                "DataItemType",
                "Color"
            ]
        }
    }
},
```

Рисунок 2.15 – Структура опису бізнес правила

В описі бізнес правила присутні дві основні компоненти – «conditions» та «event». Кондиції задають правила, і якщо ці правила в певний момент часу виконуються, то відбудеться тригер події описаного в відповідній секції. Також цікавим тут є атрибут «target» який є по суті ідентифікатором елемента на сторінці, і саме він вказує який елемент потрібно сховати чи показати, тому правило приховання чи показування елементів працює не тільки для колонок, а може працювати для будь-якого елементу на сторінці. В системі було реалізовано наступні типи бізнес правил:

- приховування/показування елементів сторінки;
- фільтрація полів з вибіркою даних за певних умов;
- автозаповнення полів за певних умов;
- блокування чи дозвіл на редагування полів;
- фільтрація таблиць з даними.

Розглянемо опис структури для колонок, що відображаються в формі, представлено на рисунку 2.13.

```
{
    "name": "primary-column-group",
    "styles": {},
    "component": "column-group",
    "parameters": {
        "columns": [
            {
                "name": "Title",
                "size": 10
            },
            {
                "name": "Icon",
                "filter": {
                    "expression": {
                        "columnPath": "IsSocialLink",
                        "value": false,
                        "comparisonType": 0
                    },
                    "logicalOperator": 0,
                    "filters": []
                },
                "size": 2
            },
            {
                "name": "Image",
                "size": 12
            }
        ]
    }
},
```

Рисунок 2.16 – Структура колонок для клієнтських сторінок

Як бачимо з структури, колонки не поміщаються на пряму на сторінку форми, в першу чергу вони агрегуються в певні групи колонок, що дозволяє користувачу розбивати свою сторінку по певних групах колоно. У кожної колонки є ім’я, яке відповідає назві колонки в об'єкті, а також розмір, тобто можна коригувати розмір колонки, враховуючи що сторінка це таблиця з 12 колонками, і ми вибираємо розмір куди помістити поле виходячи з цього. Також у колонк, що є вибіркою даних, можна задавати фільтр для фільтрації даних.

2.3.4 Додаток, що генерує клієнтські сторінки з DSL

Додаток, що генерує клієнтські сторінки з вище розглянутої DSL специфікації, є додатком що і рендерить їх і показує користувачу. Тобто це

кінцевий клієнтський інтерфейс який бачить користувач та взаємодії з усіма вище переглянутими сервісами.

Для реалізації даного додатку було обрано фреймворк Angular та Ionic для подальшої можливості конвертації даного веб додатку в мобільний додаток. Для кожного типу колонки був створений окремий генератор вихідного контенту, і була також створена компонента, що генерує сторінку форми.

Результат роботи такого генератору, представлено на рисунку 2.14.

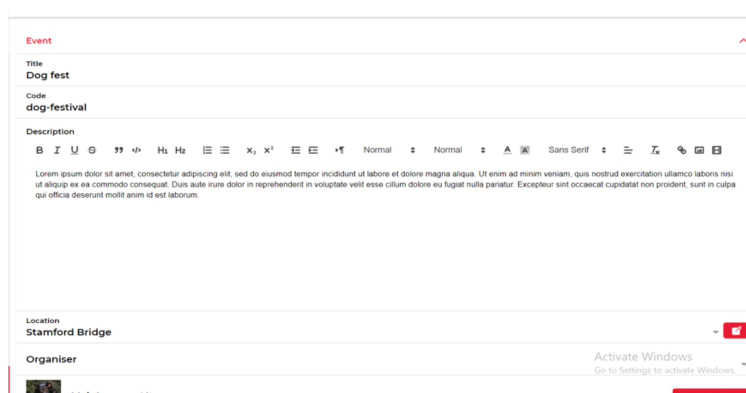

Рисунок 2.17 – Згенерована клієнтська сторінка

2.3.5 Клієнтський інтерфейс для кодогенераторів

Для розробки клієнтського інтерфейса для взаємодії з кодогенератором використовувались технології Angular та Ionic. Використання технології Ionic дає змогу в майбутньому легко перекомпілювати проект в мобільний застосунок, що дасть змогу розширити сфери застосування додатку.

Клієнтська частина включає кодогенерацію як моделей так і клієнтських сторінок, що відображатимуться кінцевому користувачу.

Клієнтський інтерфейс для кодогенератора є відносно простим, адже основна мета – це дати змогу зручно редагувати DSL структуру описану вище для моделей. Саме тому з початку користувач бачить список наявних моделей в системі та має

змогу обрати для редагування якусь з них чи створити нову, що продемонстровано на рисунку 2.15.

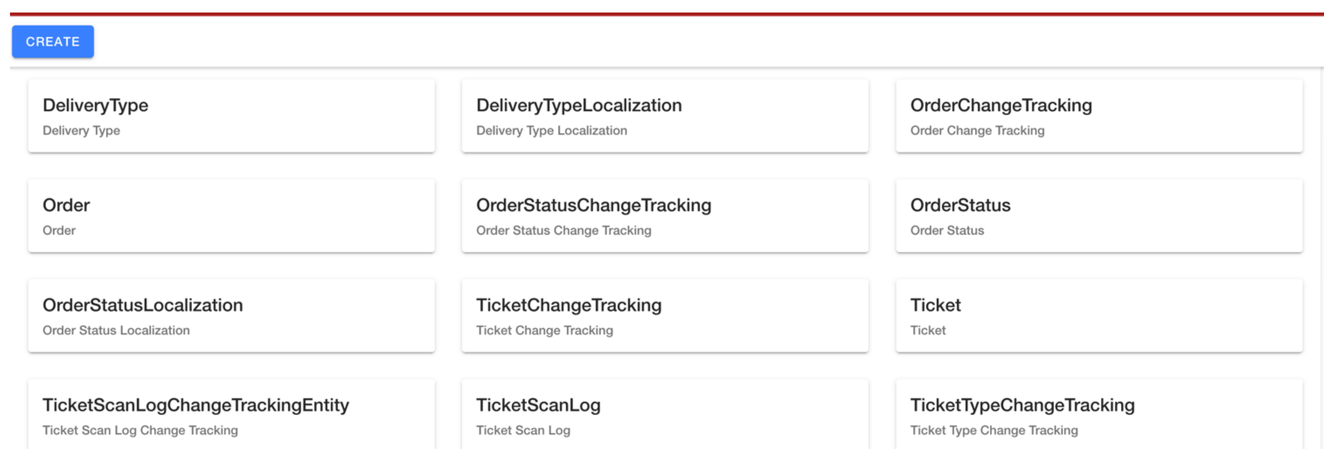

Рисунок 2.18 – Сторінка генератора з існуючими моделями

При створенні нової чи редагуванні існуючої моделі, користувач має змогу бачити та редагувати існуючі властивості моделі, а також змінювати, додавати нові чи видаляти колонки моделі (рисунок 2.16).

Після чого користувач має змогу зберегти зміни або ж відхилити їх. Під час збереження змін, всі зміни проходять верифікацію на конфлікти, до прикладу якщо користувач видалив колонку, що використовується то зміни не будуть збережені та користувач побачить діалогове вікно з помилкою. Проте ця верифікація не є кінцевою, адже при перекомпіляції та міграції змін до бази даних можуть бути знайдені нові помилки, в такому разі користувач отримає помилку в асинхронному режимі та його зміни будуть відхилені.

Складнішим в реалізації є клієнтський інтерфейс для кодогенерації клієнтських сторінок, адже тут використовуються різноманітні компоненти від простих колонок до складних таблиць чи графіків.

Початкові дії взаємодії користувача з кодогенератором сторінок є досить схожими на взаємодію з кодогенератором моделей. Адже початково користувач повинен обрати сторінку для редагування чи створити нову.

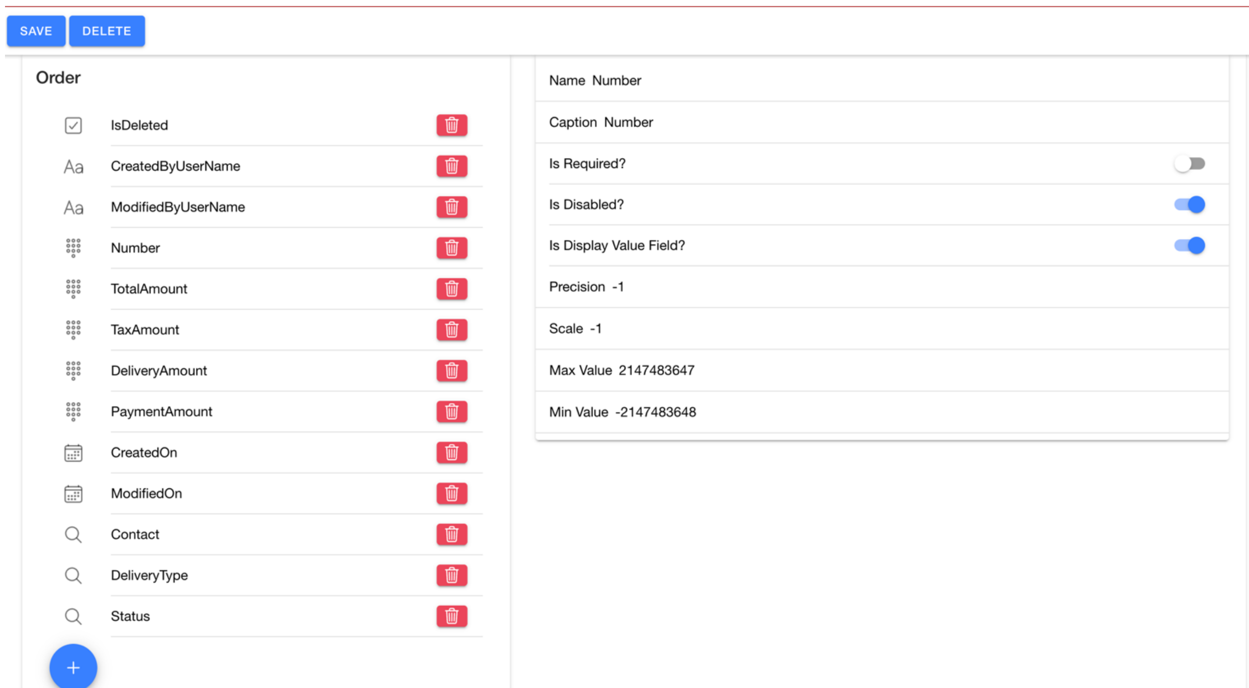

Рисунок 2.19 – Сторінка генератора створення/редагування моделі

При створенні нової сторінки користувач повинен обрати тип сторінки (сторінка для моделі чи таблиця для моделі) та модель для якої ця сторінка створюється. Після чого, інтерфейс створення є еквівалентним інтерфейсу редагування, перед користувачем знаходиться безкінечне полотно яке розділено на рядки та в кожному рядку по 12 колонок. В кожну комірку цього полотна користувач має змогу перетягнути певний елемент клієнтського інтерфейсу (колонку, таблицю і т.д.) та за потреби розширити межі цього елементу використовуючи функцію розтягування. Також, кожну компоненту користувач може окремо налаштовувати натиснувши на неї, до прикладу налаштувати бізнес правила для колонки. Приклад даного конструктора зображений на рисунку 2.17.

Далі користувач знову має змогу зберегти чи відхилити зміни, при збереженні зміни будуть верифіковані системою.

Рисунок 2.20 – Сторінка генератора клієнтських сторінок

2.3.6 Сервіс для перекомпіляції моделей та оновлення структури сторінок

Після збереження нових чи змінених моделей та клієнтських сторінок, додаток потребує перекомпіляції цих моделей та підміни їх в усіх залежних сервісах, щоб працювати без помилок.

Для цього після збереження моделі та її верифікації відбувається компіляція проекту з моделями, якщо вона пройшла успішно, відправляється повідомлення до брокеру про нові моделі, після чого усі зацікавлені сервіси оновлюють посилання на моделі та перезавантажуються. У випадку успіху всіх цих дій, вважаємо що оновлення пройшло успішно, проте якщо певний сервіс не зміг оновити моделі чи компіляція пройшла не успішно, усі зміни відхиляються та сервіси знову використовують старі моделі. Даний процес наведено на діаграмі послідовності на рисунку 2.18.

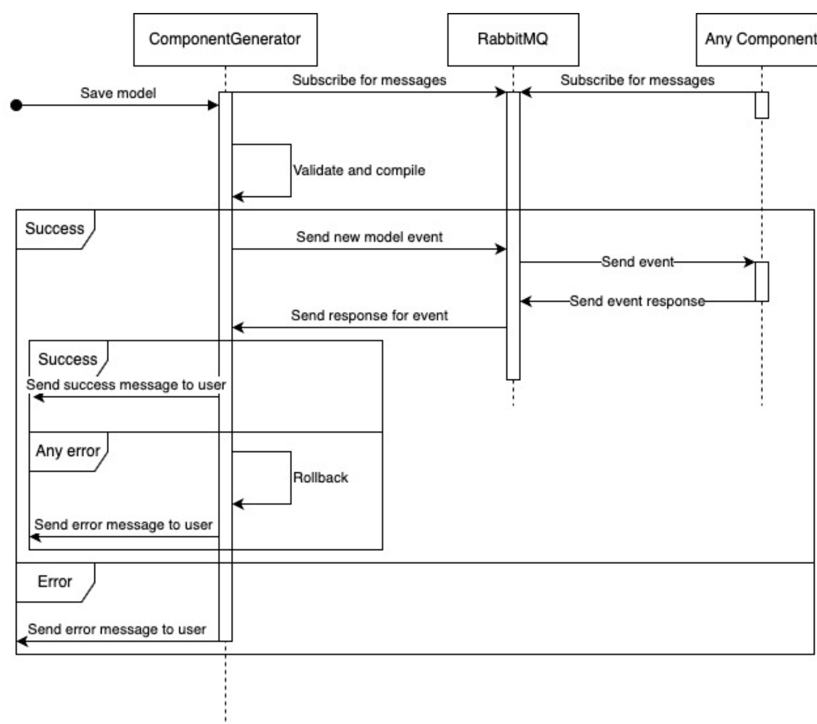

Рисунок 2.18 – Діаграма процесу перекомпіляції

Під час оновлення структури клієнтських сторінок, перекомпіляція не є потрібною, тому цей процес є значно простішим. Все що відбувається після збереження змін структури клієнтських сторінок – це відправлення повідомлення до брокеру, після чого це повідомлення обробляється сервісом комунікації в режимі реального часу, тобто відправляє повідомлення на клієнтський додаток про оновлення за допомогою WebSocket. Після чого, клієнтський додаток загружає нову версію структури та використовує її.

РОЗДІЛ 3 ВИКОРИСТАННЯ ШТУЧНОГО ІНТЕЛЕКТУ ДЛЯ КОДОГЕНЕРАЦІЇ

3.1 Постановка задачі

Метою даної розробки є задоволення потреби користувачів у легкому способі доступу до даних за допомогою пошукача, що використовується на етапі використання додатку. Користувач повинен мати змогу отримувати дані відповідно до його текстового запиту. Наприклад, користувач вводить запит до пошукача: «покажи всі замовлення, де загальна ціна більше тисячі гривень», додаток повинен зрозуміти запит, згенерувати запит у форматі, що може бути опрацьований компонентом DataRepository, що був описаний в розділі 2.2.7.

3.2 Загальна реалізація

З постановки зрозуміло, що додаток повинен певним чином зрозуміти запит користувача, написаний на природній мові, та згенерувати контракт (розділ 2.2.7), що може бути опрацьований сервісом доступу до даних. Завдання розуміння природньої мови та генерації представлення запиту є класичною задачею штучного інтелекту з NLP.

Тому для розробки застосунку було використано технології штучного інтелекту та була натренована відповідна модель для опрацювання запитів. Загальний підхід до вирішення проблеми:

- підготовка датасету для тренування моделі;.
- вибір методу тренування моделі;
- тренування моделі;
- оцінювання моделі;
- розміщення моделі в хмарі для взаємодії з нею;

– інтегрування моделі з існуючим веб додатком.

Враховуючи контракт вихідного формату, модель повинна не тільки генерувати його, а ще розуміти структуру доменних моделей, для того щоб запит міг бути опрацьований веб додатком. Для цього модель повинна виконати дві дії:

– зрозуміти до якої моделі відноситься запит користувача, з прикладу наведеного в постановці – модель повинна зрозуміти, що запит відноситься до доменної моделі замовлень;
– знаючи структуру доменної моделі, згенерувати коректний запит.

3.3 Підготовка датасету

Отже першим завдання є підготовка датасету для тренування моделі та в цілому потрібно зрозуміти дані з якими буде працювати додаток. Отже, з постановки задачі зрозуміло, що на вхід модель повинна отримати текст та згенерувати представлення цього запиту в певній структурі, що може бути опрацьована веб додатком. Для представлення запиту будемо використовувати контракт версії 2 представлений в розділі 2.3.7.

Отже ми зрозуміли природу даних з якими працюватиме модель, тепер потрібно підготувати відповідний датасет, що дасть змогу моделі ефективно навчитись опрацьовувати запити. Зрозуміло, що вихідний формат є досить специфічним для даного додатку, тому знайти датасет в відкритому доступі, що задовільнить наші потреби неможливо. Тому для створення відповідного датасету був обраний підхід синтетичної генерації даних.

Для генерації датасету використовувалась мова програмування Python та бібліотека jinja2, що дає змогу використовувати шаблони та підміняти ключі в шаблонах. В якості шаблонів використовувалось декілька для генерації датасетів з розуміння доменної моделі та генерації запиту на основі моделі. Приклад шаблону для розуміння доменної моделі зображено на рисунку 3.1. Приклад для генерації запиту зображено на рисунку 3.2.

```
Request: {{ search_word }} {{ model_alias }} where {{ col_alias1 }} {{ op_name1 }} {{ col_value_alias1 }}
Model:[-SPLITER-]{{ model_name }}
```

Рисунок 3.21 – Шаблон для генерації даних розуміння доменної моделі

```
Request: {{ search_word }} {{ model_alias }} where count of {{ col_alias }} {{ op_name }} {{ num_word }}
Query:[-SPLITER-]$count({{ col_name }}) {{ op }} {{ num }}
```

Рисунок 3.22 – Шаблон для генерації даних генерації запиту

Всі ключі знаходяться в подвійних фігурних дужках і динамічно підставляються при виконанні коду.

Після створення датасетів, для того щоб не втратити дані, датасети було об'єднано та розміщено за допомогою застосунку HuggingFace – аналог GitHub для уніфікованого доступу до моделей штучного інтелекту та датасетів. Даний датасет може бути знайдений на платформі за ідентифікатором dkuntso/gen-qm-17000.

3.4 Вибір методу тренування

Коректне тренування моделі є центральним в здатності моделі генерувати якомога більше коректних відповідей. Саме тому важливо зрозуміти який саме тип моделі вирішуватиме поставлену задачу та як правильно натренувати модель.

Сьогодні популярними є великі мовні моделі або ж трансформери, і це не безпідставно адже ці моделі є величезними та здатні вирішувати завдання різного роду. Ці моделі чудово вирішують завдання NLP, проте щоб натренувати таку модель з нуля, потрібно багато ресурсів, що означає багато коштів. Саме тому для використання цих моделей для специфічних завдань існує інший підхід, що називається тонке налаштування (fine tuning).

Тонке налаштування дозволяє обрати існуючу модель та на її збережених параметрах продовжити тренування для специфічного датасету, щоб вирішувати конкретну проблему. Саме цей підхід використовується в даній роботі, адже щоб

зробити тонке налаштування потрібно значно менше ресурсів, проте на виході отримується могутня модель, що здатна вирішити проблему.

Для тренування та оцінювання моделі використовувалась мова програмування Python, з бібліотекою HuggingFace, що дає змогу використати простий інтерфейс для тренування, тонкого налаштування чи оцінювання моделі. Також дана бібліотека дозволяє зберігати модель, так само як і датасет.

Також після тонкого налаштування, для покращення результатів роботи моделі використовувались підходи контекстного навчання (in-context learning) [26], що передбачає додавання певного контексту до запиту, наприклад під час виконання запиту з постановки як контекст можна передати список існуючих доменних моделей з їх назвами, таким чином модель зможе краще знаходити доменну модель до якої відноситься запит. В якості контектсу для генерації запиту, передавався список з усіх колонок з їх іменами та типами для визначеної доменної моделі на попередньому кроці.

Також був застосований підхід навчання з одним прикладом (one-shot learning) [27], який передбачає передавання моделі не тільки контексту, а і одного прикладу з відповіддю.

3.5 Тонке налаштування моделі

Для тонкого налаштування використовувався сервіс AWS SageMaker, що дає змогу використовувати Jupiter Notebook, для написання коду та вже має встановленні основні залежності. Тип віртуальної машини на якій працював сервіс, що використовувався є p3dn.24xlarge – тобто машина з 8 ядрами GPU, для швидкого тонкого налаштування.

З технологій використовувалась бібліотека HuggingFace з Pytorch для тонкого налаштування. За базову модель було обрано трансформер t5-small від компанії Google, що знаходиться в публічному доступі. Датасет, що використовувався був описаний вище та мав розмір в 17000 записів. Це не великий

датасет, тому з ним неможливо отримати хороших результатів під час оцінювання, але при збільшенні розміру збільшився і час тонкого налаштування, оскільки налаштування передбачає декілька епох тренування, то зі збільшенням розміру датасету час збільшується майже з експоненціальною швидкістю. В якості токенайзеру слів використовувався T5Tokenizer від Google.

В результаті отримали модель, що може бути знайдена на HuggingFace за ідентифікатором dkuntso/gen-qm-17-small, що виконує поставлену задачу. На рисунку 3.3, представлено графік залежності втрат моделі під час навчання від епохи. Бачимо, що втрати зменшувались з кожною епохою, що означає що модель дійсно навчалась опрацьовувати запити.

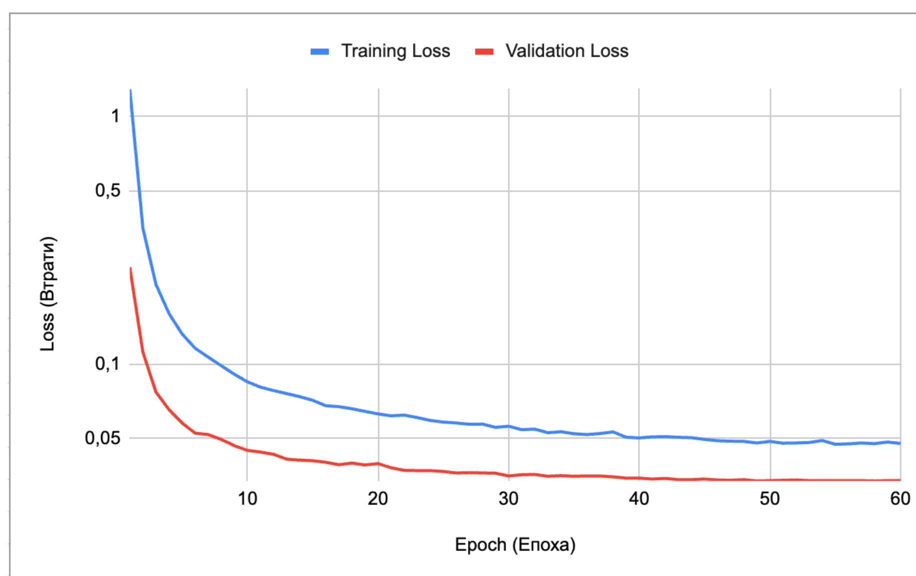

Рисунок 3.23 – Графік залежності втрат від епохи при навчанні

3.6 Оцінювання моделі

Для оцінювання моделі використовувалась бібліотека HuggingFace evaluate, що дала змогу використовувати різні метрики такі як rouge чи bleu, проте дані метрики не є сильно показовими для даного завдання, в даному випадку, так як модель здебільшого повинна генерувати сталу відповідь на запит, то можна використати стандартну метрику точності для оцінювання моделі. В таблиці 3.1

наведено результати оцінювання використовуючи різні підходи при надсиланні запиту до моделі.

З таблиці 3.1 видно, що модель добре вміє знаходити доменну модель до якої відноситься запит, особливо використовуючи підкоди контекстного навчання та одно-пострільного. Результати генерації запитів дещо гірші, адже це складніше завдання, більш того на оцінювання впливало те, що згенерований запит міг бути коректний, проте відрізнятись від того що знаходився в датасеті.

Таблиця 3.1 – Точність моделі в залежності від підходу та завдання

Завдання	Підхід	Точність
Знаходження доменної моделі	Без модифікацій	54%
Знаходження доменної моделі	З контекстним навчанням	72%
Знаходження доменної моделі	З one-shot навчанням	67%
Знаходження доменної моделі	З контекстним навчанням та one-shot	86%
Генерація запиту	Без модифікацій	32%
Генерація запиту	З контекстним навчанням	54%
Генерація запиту	З one-shot навчанням	34%
Генерація запиту	З контекстним навчанням та one-shot	58%

Варто зазначити, що розмір датасету та синтетичність даних впливають на оцінювання моделі також, модель натренована на синтетичних даних може бути добре оцінена на них, проте на даних від реальних користувачів зазнає краху. Але процес тонкого налаштування дозволяє тренувати модель регулярно, тому всі дані

які вводять користувачі будуть зберігати та використовуватись для подальшого навчання моделі.

Було також проведено порівняння роботи моделі з існуючою моделлю OpenAI, що доступна за допомогою додатку ChatGPT. В даному випадку тонке налаштування даної моделі не передбачене, проте було використане контекстне та навчання з прикладом для покращення результатів. Точність роботи моделі OpenAI досягла 95% для завдання знаходження доменної моделі та 89% для генерації запиту. Дані числа є набагато більшими чим отримані при тонкому налаштування, але це є логічно враховуючи параметри тонкого налаштування, та те що модель OpenAI є набагато більшою (~300 мільярдів параметрів), що дає їй змогу виконувати з високою точність будь-які завдання при правильному контексті запиту. Використання власної моделі має певні переваги над використанням моделей OpenAI, до прикладу – незалежність від сервісів API, питання безпеки та приватності даних користувачів (дані не потраплятимуть до зовнішніх застосунків).

3.7 Інтеграція моделі з веб додатком

Як уже зазначалось раніше, модель зберігається у відкритому доступі на платформі HuggingFace, який дає змогу взаємодіяти з моделлю за допомогою API. Звичайно цей API не є швидким, і краще розміщувати модель ближче до веб додатку, але для економії ресурсів та враховуючи невелику навантаження на модель, було вирішено для початку використовувати модель за допомогою API HuggingFace. Приклад взаємодії з моделлю представлено на рисунках 3.4 та 3.5.

Також раніше згадувалось, що даний API використовується в компоненті WorkflowEngine, яка в свою чергу надає зручний інтерфейс взаємодії для клієнтського додатку. Приклад використання даного функціоналу в клієнтському додатку зображено на рисунку 3.6.

Hosted inference API

Text2Text Generation

Instructions: Based on request and possible Model Names fetch the model name from request. Usually model in the request can be referenced by alias. The mapping between alias and model name can be found in Model Names, for example Contact as contact,person,human where Contact is model name and contact,person,human are aliases. All possible model names and aliases are separeted by semicolumn.
Model Names: Page as page;PageMetaValue as page meta value;Organiser as organiser;DeliveryType as delivery type;Order as order;OrderStatus as order status;Ticket as ticket;TicketScanLog as ticket scan log;TicketType as ticket type,hucket type;MetaKey as meta key;Message as message;MessageAttachment as message attachment;MessageStopPhrases as message stop phrases;City as city;Country as country;Location as location;LocationType as location type;Image as image;Honorific as honorific;Gender as gender;Event as experience,event;EventCategory as experience category,event category;EventCategoryInType as event category in type,experience category in type;EventCategoryType as experience category type,event category type;EventDate as event date,experience date;EventFlag as event flag,experience flag;EventImage as experience image,event image;EventInCategory as experience in category,event in category;EventLineUp as event line up,experience line up;EventMember as experience member,event member;EventMemberType as event member type,experience member type;EventMetaValue as event meta value,experience meta value;EventPlace as event place,experience place;EventReview as experience review,event review;EventTag as event tag,experience tag;FavouriteEvent as favourite event;Tag as tag;Contact as contact;Currency as currency;Language as language;MultiCurrencyAmount as multicurrencyamount,multi currency amount
Request: look up order where subtitle of delivery type like xniVD
Model:

Compute ⌘+Enter 0.0

Computation time on Intel Xeon 3rd Gen Scalable cpu: 0.087 s

Order

Рисунок 3.24 – Генерація доменної моделі

Hosted inference API

Text2Text Generation

Instructions: Based on Request and Model Description generate query with represents requests filter. Generaly query statement consists of path to the models column on the left, operator of comparison in the middle started with $ and comparison value on the right. Also query can contain more than one statement combined with $and or $or operator.
Model Description: EventLink as link to website,event link;EventTitle as experience title,event title;CreatedByUserName as created by user name;ModifiedByUserName as modified by user name;CreatedOn as created on;ModifiedOn as modified on;Event.IsActive as is visible on site of experience,is active of event;Event.Title as title of experience;Event.Code as code of experience;Event.Description as description of experience;Event.Link as link of experience;Event.LocationTitle as location title of experience;Event.CreatedByUserName as created by user name of experience;Event.ModifiedByUserName as modified by user name of experience;Event.MinPrice as min price, base currency of experience,min price of event;Event.MaxPrice as max price, base currency of experience,max price of event;Event.LikesCount as likes count of experience;Event.TotalTicketsCount as total tickets count of experience;Event.SoldTicketsCount as sold tickets count of experience;Event.SoldTicketsAmount as sold tickets amount of experience,sold tickets amount, base currency of event;Event.StartDate as start date of experience;Event.EndDate as end date of experience;Event.CreatedOn as created on of experience;Event.ModifiedOn as modified on of experience;Event.EventImages as experience images of experience,event images of event;Event.EventInCategories as experience in categories of experience,event in categories of event;Event.EventFlags as event flags of experience,experience flags of event;Event.FavouriteEvents as favourite events of experience;Event.EventReviews as event reviews of experience,experience reviews of event;Event.EventMetaValues as event meta values of experience,experience meta values of event;Event.Dates as dates of experience
Request: select favourite event where link of experience equal localhost
Query:

Compute ⌘+Enter 0.3

Computation time on Intel Xeon 3rd Gen Scalable cpu: 0.139 s

Event.Link $eq localhost

Рисунок 3.25 – Генерація запиту

IDEA PROVIDER Global Search

show me all client orders where total price is less than 15 euros

Hi null null,
Welcome back!
Global search
Specifikācijas veids
Pakalpojumu konsultāc...
apkalpošana
Produktu konsultāciju p...
Produkts
Maksājuma veids
Lappuse
Rikojums
Pasūtījuma statuss
ArticleEntityCaption
ArticleBlockTypeEntityC...
ArticleBlockSizeEntityC...
EN

Add

Rikojums

Numurs
1001010
Kopējā summa
27.8
Nodokļu summa
4.82
Piegādes summa
0
Maksājuma summa
0
Saziņāties
broaden.your@outlook.com
Adrese
Maksājuma veids
Piegādes veids
Saņemt uzvietas darbnīcā
Pasūtījuma statuss

Pasūtīt produktu variantu

Cena	Daudzums	Produkta variācija
13.9	1	TĒJAS UN KAFIJAS TRAIPU TĪRĪTĀJS / REMOVE IT! TEA & COFFEE STAIN REMOVER (13.9)
13.9	1	TĒJAS UN KAFIJAS TRAIPU TĪRĪTĀJS / REMOVE IT! TEA & COFFEE STAIN REMOVER (13.9)

Add

Рисунок 3.26 – Пошукова система даних інтегрована в веб додаток

ВИСНОВКИ

У процесі створення кваліфікаційної роботи для автоматизації бізнесу малого та середнього сегменту, що дозволятиме налаштовувати даний додаток під кожного клієнта без написання додаткового коду було зроблено наступне:

- досліджено поточний стан розвитку додатків «Low-code» та «No-code»;
- проаналізовано основні методи кодогенерації;
- застосовано кодогенерацію за допомогою шаблонів, рефлексії та штучного інтелекту;
- досліджено принципи «чистої» архітектури програмних додатків;
- спроектовано архітектуру додатка та реалізовано її;
- розроблено процес розгортання та супроводу додатка за допомогою Docker;
- використано сервіси AWS для розміщення веб додатку;
- розроблено додаток для генерації компонент клієнтського інтерфейсу на основі структурованих описів цих компонент;
- проведено тонке налаштування моделі штучного інтелекту для кодогенерації, а також застосовано контекстне навчання та одно-пострільне навчання для покращення результатів роботи моделі.

В подальшому планується покращення алгоритмів для модифікації існуючих моделей в кодогенерації. Також розширення мови опису моделей новими типами та атрибутами. Збереження клієнтських запитів до моделі штучного інтелекту, для подальшого їх анотування та використання в наступних ітераціях навчання.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. What is low-code/no-code application development? | SAP Insights [Електронний ресурс] // SAP. – Режим доступу: https://www.sap.com/products/technology-platform/low-code/what-is-low-code-no-code.html. – Назва з екрана.
2. Business apps | microsoft power apps [Електронний ресурс] // Business Apps | Microsoft Power Apps. – Режим доступу: https://powerapps.microsoft.com/en-gb. – Назва з екрана.
3. AppSheet: no-code app development | google cloud [Електронний ресурс] // Google Cloud. – Режим доступу: https://cloud.google.com/appsheet. – Назва з екрана.
4. Zoho creator: best low-code custom application development platform [Електронний ресурс] // Zoho. – Режим доступу: https://www.zoho.com/creator/. – Назва з екрана.
5. Appian [Електронний ресурс] // Appian Platform for Process Automation -Low-Code - Process Mining. – Режим доступу: https://appian.com/. – Назва з екрана.
6. The best way to build web apps without code | Bubble [Електронний ресурс] // Bubble. – Режим доступу: https://bubble.io. – Назва з екрана.
7. Simon P. Low-Code/No-Code: citizen developers and the exciting future of business applications / Phil Simon. – [Б. м.] : Racket Publishing, 2023. – 278 c.
8. Mernik M. When and how to develop domain-specific languages / Marjan Mernik, Jan Heering, Anthony M. Sloane // ACM computing surveys. – 2005. – Vol. 37, no. 4. – P. 316–344.
9. Solymosi M. Creating a ruby DSL: a guide to advanced metaprogramming | toptal® [Електронний ресурс] / Máté Solymosi // Toptal Engineering Blog.

– Режим доступу: https://www.toptal.com/ruby/ruby-dsl-metaprogramming-guide. – Назва з екрана.

10. Apache hadoop [Електронний ресурс] // Apache Hadoop. – Режим доступу: https://hadoop.apache.org/. – Назва з екрана.
11. Koelewijn A. SQL is a DSL [Електронний ресурс] / Andrej Koelewijn. – Режим доступу: https://www.andrejkoelewijn.com/blog/2008/10/27/sql-is-a-dsl. – Назва з екрана.
12. DSL::HTML - declarative dsl(domain specific language) for writing HTML templates within perl. - metacpan.org [Електронний ресурс] // Search the CPAN - metacpan.org. – Режим доступу: https://metacpan.org/pod/DSL::HTML. – Назва з екрана.
13. Fowler M. Domain-specific languages / Martin Fowler. – Upper Saddle River, NJ : Addison-Wesley, 2011. – 597 с.
14. Herrington J. Code generation in action / Jack Herrington. – Greenwich, CT : Manning, 2003. – 342 с.
15. What is SysML? | OMG SysML [Електронний ресурс] // OMG SysML Home | OMG Systems Modeling Language. – Режим доступу: https://www.omgsysml.org/what-is-sysml.htm. – Назва з екрана.
16. Appery.io [Електронний ресурс] // Appery.io. – Режим доступу: https://appery.io/. – Назва з екрана.
17. Dollard K. Code generation in microsoft .NET / Kathleen Dollard. – [Б. м.] : Apress, 2004. – 730 с.
18. Grinder J. Origins of neuro linguistic programming : навч. посіб. / John Grinder. – [Б. м.] : Crown House Publishing, 2013. – 322 с.
19. Alammar J. The illustrated transformer [Електронний ресурс] / Jay Alammar // Jay Alammar – Visualizing machine learning one concept at a time. – Режим доступу: https://jalammar.github.io/illustrated-transformer. – Назва з екрана.
20. Attention is all you need / Ashish Vaswani [та ін.] // NIPS'17: Proceedings of the 31st International Conference on Neural Information Processing

Systems, Long Beach, 4–9 груд. 2017 р. – New York, 2017. – С. 6000–6010.

21. Fowler M. Patterns of enterprise application architecture / Martin Fowler. – [Б. м.] : Addison-Wesley Professional, 2002. – 560 с.
22. Martinez P. Hexagonal Architecture, there are always two sides to every story [Електронний ресурс] / Pablo Martinez // Medium. – Режим доступу: https://medium.com/ssense-tech/hexagonal-architecture-there-are-always-two-sides-to-every-story-bc0780ed7d9c. – Назва з екрана.
23. Reenskaug T. The DCI architecture: a new vision of object-oriented programming [Електронний ресурс] / Trygve Reenskaug, James O. Coplien // artima. – Режим доступу: https://www.artima.com/articles/the-dci-architecture-a-new-vision-of-object-oriented-programming. – Назва з екрана.
24. Kosar V. Boundary control entity architecture pattern [Електронний ресурс] / Vaclav Kosar // Vaclav Kosar's Software & Machine Learning Blog. – Режим доступу: https://vaclavkosar.com/software/Boundary-Control-Entity-Architecture-The-Pattern-to-Structure-Your-Classes. – Назва з екрана.
25. Elsa workflows - add workflow capabilities to any .NET project. [Електронний ресурс] // Elsa Workflows - Add workflow capabilities to any .NET project. – Режим доступу: https://v2.elsaworkflows.io/ (дата звернення: 13.05.2023). – Назва з екрана.
26. Solving a machine-learning mystery [Електронний ресурс] // MIT News | Massachusetts Institute of Technology. – Режим доступу: https://news.mit.edu/2023/large-language-models-in-context-learning-0207. – Назва з екрана.
27. Logunova I. Neural networks: one-shot learning [Електронний ресурс] / Inna Logunova // Serokell Software Development Company. – Режим доступу: https://serokell.io/blog/nn-and-one-shot-learning. – Назва з екрана.