Міністерство освіти і науки України
Львівський національний університет імені Івана Франка
Факультет електроніки та комп’ютерних технологій

Кафедра радіоелектронних
та комп’ютерних систем

КУРСОВА РОБОТА

«Створення патерну програмування для реалізації тригерів у CRM-системах»

Виконав: ст.гр. ФеІм-14
_____ Сенько В.Я.
(дата)
Науковий керівник:
_____ доц. Оленич І.Б.
(дата)

Львів – 2023

АНОТАЦІЯ

У курсовій роботі був розглянутий спосіб написання патерну проектування для реалізації тригерів у CRM-системах, на прикладі системи Salesforce. Реалізація патерну відбулась за допомогою мови програмування Apex та метаданих, що використовуються у цій CRM-системі.

ANNOTATION

In the course work, the method of writing a design pattern for the implementation of triggers in CRM-systems was considered, using the example of the Salesforce system. The pattern was implemented using the Apex programming language and metadata used in this CRM-system.

ЗМІСТ

ВСТУП.....3

РОЗДІЛ 1. ОБРАНІ ТЕХНОЛОГІЇ.....4

1.1 Основні поняття та визначення.....4

1.2 Основні можливості Salesforce.....6

1.3 Практичне застосування Salesforce.....7

1.4 Особливості та переваги використання мови програмування Apex.....8

1.5 Особливості використання тригерів у Apex.....9

1.6 Custom Setting і Custom Metadata в Salesforce.....11

1.7 Патерни проектування.....13

1.8 Особливості середовища програмування Intellij Idea.....15

РОЗДІЛ 2. ПРОЦЕС СТВОРЕННЯ.....18

2.1 Алгоритм роботи.....18

2.2 Процес створення патерну програмування.....18

2.3 Демонстрація роботи проекту.....22

ВИСНОВКИ.....25

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ.....26

ДОДАТОК 1 Реалізація алгоритму створення патерну програмування з використанням мови Apex та CRM-системи Salesforce.....27

ВСТУП

На даний момент CRM-системи користуються попитом серед компаній, це програмне забезпечення, допомагає керувати бізнесом, взаємодіяти з клієнтами, збирати інформацію.

Однією з функцій CRM-системи є можливість автоматично реагувати на події, що відбуваються в системі, і виконувати відповідні дії. Ці дії виконуються за допомогою тригерів. Тригери можуть бути використані для автоматизації бізнес-процесів, забезпечення послідовності дій та маніпулювання даними. Проте, розробка та реалізація ефективних тригерів може бути досить складним завданням, яке вимагає обґрунтованого підходу та професійних навичок. За допомогою патерну програмування для реалізації тригерів навіть недосвідчений користувач матиме змогу в кілька кроків реалізовувати потрібну бізнес логіку.

Метою даної курсової роботи є розробка та впровадженні патерну програмування, який дозволить створити гнучкі, повторно використовувані та легко зрозумілі реалізації тригерів у CRM-системах. Окрім цього, у цій курсовій роботі ми розглянемо проблематику створення патерну програмування для реалізації тригерів у CRM-системах, дослідимо існуючі патерни проектування та проаналізуємо їх застосування у контексті CRM-систем.

РОЗДІЛ 1. ОБРАНІ ТЕХНОЛОГІЇ

1.1 Основні поняття та визначення

Salesforce - це відома та впливова хмарна CRM-платформа (Customer Relationship Management), яка надає широкий спектр функцій та можливостей для управління взаємодією з клієнтами, продажами, маркетингом та іншими аспектами бізнесу. Вона використовується компаніями різного розміру та галузей, починаючи від малих стартапів до великих корпорацій [10].

Salesforce з часом перетворився на потужний SaaS, пропонуючи кілька хмарних платформ для спеціальних цілей. У серпні 2022 року Salesforce оголосила, що має дохід у розмірі 7,72 мільярда доларів, що на 22% зросло за рік.

Основна передумова Salesforce — це доступне програмне забезпечення CRM як онлайн-сервіс. До появи Salesforce більшість компаній розміщували програмне забезпечення CRM на своїх серверах або використовували локальні ресурси, що вимагало багато часу та фінансових інвестицій.

Salesforce пропонує модель підписки з оплатою за використання та зберігає всі дані в хмарі, що робить їх легко доступними з будь-якого пристрою, підключеного до Інтернету. Salesforce пропонує кілька інноваційних технологій, які допомагають об’єднати клієнтів, компанії, розробників і ділових партнерів.

Apex — це об’єктно-орієнтована мова програмування, яка дозволяє розробникам виконувати оператори керування потоком і транзакціями на платформі Salesforce. Apex інтегрований, простий у використанні, орієнтований на дані, розміщений, підтримує роботу з кількома клієнтами, автоматично оновлюється, простий у тестуванні та версії [1].

Visualforce — це фреймворк, який дозволяє розробникам створювати динамічні багаторазові інтерфейси, які можуть бути розміщені на Salesforce. Вони можуть створювати цілі власні сторінки всередині організації Salesforce або пов’язувати свою логіку з класом контролера, написаним на Apex. Розробники можуть використовувати сторінки Visualforce для заміни стандартних кнопок і сторінок огляду вкладок, визначення користувальницьких

вкладок, вбудовування компонентів у детальні макети сторінок, створення компонентів інформаційної панелі, налаштування бічних панелей у консолі Salesforce і додавання пунктів меню.

Lightning — це вдосконалена версія Salesforce Classic. Його компонентна структура дозволяє розробникам створювати адаптивні додатки SaaS для будь-якого пристрою. Він дає змогу бізнес-користувачам із мінімальними навичками програмування або без них створювати сторонні користувацькі додатки поверх додатків Salesforce.

Salesforce Einstein — це комплексна технологія AI для CRM, розроблена для платформи Salesforce Customer Success Platform. Einstein призначений для надання відділам продажів і маркетингу більш повного й актуального уявлення про клієнтів і потенційних клієнтів. Його розроблено, щоб зробити Salesforce Customer 360 інтелектуальнішим і надати штучний інтелект новаторам у всьому світі.

1.2 Основні можливості Salesforce

Salesforce вважається найпопулярнішою CRM системою, а свою популярність ця система завоювала не просто так. У ній реалізовано великий системний функціонал для побудови бізнес системи саме для ваших потреб, якщо системного функціоналу недостатньо, завжди є можливість розробити зручну бізнес логіку за допомогою Apex коду. Розглянемо основні можливості Salesforce, за допомогою яких він завоював таку популярність:

1. Управління клієнтськими даними - Salesforce дозволяє зберігати та організовувати інформацію про клієнтів, контакти, облікові записи, можливості та інші дані. Це допомагає підтримувати взаємодію з клієнтами, відстежувати їхні потреби та покращувати обслуговування.
2. Автоматизація продажів - Salesforce надає інструменти для ефективного керування процесом продажу. Ви можете створювати та відстежувати угоди, прогнозувати продажі, управляти комунікацією з клієнтами та виконувати інші завдання, пов'язані з продажами.

3. Маркетинг та автоматизація маркетингу - Salesforce дозволяє вам створювати та виконувати маркетингові кампанії, відстежувати результати, аналізувати дані та взаємодіяти зі своїми клієнтами через різні канали комунікації.
4. Сервісна підтримка клієнтів - Salesforce надає інструменти для організації та виконання сервісної підтримки клієнтів. Ви можете керувати запитами, відстежувати проблеми та взаємодіяти з клієнтами для вирішення їхніх потреб.
5. Аналітика та звіти - Salesforce має вбудовані засоби аналітики, які дозволяють вам отримувати дані та створювати звіти для аналізу різних аспектів вашого бізнесу. Ви можете відстежувати ключові показники ефективності, спостерігати за трендами та приймати обґрунтовані рішення.
6. Розширення функціональності - Salesforce надає можливості розширення своєї функціональності шляхом використання додатків і розширень з AppExchange - маркетплейсу Salesforce, де доступні різноманітні готові рішення сторонніх розробників.

1.3 Практичне застосування Salesforce

Завдяки своїй гнучкості та широкому спектру функціональності, Salesforce може бути адаптований до різних бізнес-потреб та галузей. Розглянемо кілька реальних випадків використання Salesforce:

1. Компанії можуть використовувати Salesforce для ефективного керування процесом продажу, відстеження угод, управління комунікацією з клієнтами та аналізу результатів.
2. Маркетологи можуть використовувати Salesforce для створення та виконання маркетингових кампаній, відстеження результатів та аналізу ефективності.

3. Компанії можуть використовувати Salesforce для організації та виконання сервісної підтримки клієнтів, вирішення проблем та підтримки взаємодії з клієнтами.
4. Salesforce надає засоби для аналізу даних та створення звітів, що дозволяє компаніям відстежувати та аналізувати ключові показники ефективності та приймати обґрунтовані рішення.

1.4 Особливості та переваги використання мови програмування Apex

Мова програмування Apex була розроблена компанією Salesforce для використання в їх хмарній платформі Salesforce. Вона була представлена в 2007 році як мова спеціально для розробки на платформі Salesforce.

Apex базується на мові Java та має схожий синтаксис і структуру. Однак, Apex є мовою, орієнтованою на об'єктно-орієнтоване програмування (ООП) та зосереджена на роботі з даними та функціональністю, які надає Salesforce [3].

Apex використовується для розробки тригерів (triggers), обробників подій (event handlers), кастомних класів та контролерів, які виконують бізнес-логіку та маніпулюють даними на платформі Salesforce. Вона має доступ до API Salesforce для взаємодії з базою даних Salesforce, об'єктами, процесами та іншими компонентами платформи.

Apex також надає можливості для тестування коду та безпечного деплойменту. Він інтегрується з іншими функціями Salesforce, такими як веб-сервіси, аналітика та звіти, дозволяючи розробникам створювати потужні та настроювані додатки на платформі Salesforce.

Apex є важливою частиною розробки на платформі Salesforce і дозволяє розробникам створювати кастомні рішення, розширювати функціональність та автоматизувати бізнес-процеси на цій платформі.

Особливості та переваги використання мови програмування Apex [7]:

1. Тригери та обробники подій: Apex використовується для написання тригерів, які автоматично реагують на події, що відбуваються в Salesforce, такі як створення, оновлення або видалення записів. Це дозволяє розробникам виконувати додаткові дії та бізнес-логіку при зміні даних.
2. Інтеграція з базою даних Salesforce: Apex надає можливість взаємодіяти з базою даних Salesforce, виконувати запити, оновлювати та створювати записи. Це дозволяє розробникам здійснювати потужні операції з даними та забезпечувати їх цілісність.
3. Обробка подій та бізнес-логіка: Apex дозволяє розробникам створювати кастомні обробники подій, які реагують на дії користувачів або системні події. Це дозволяє виконувати різні операції, такі як створення, оновлення або видалення даних, автоматично виконувати певні дії або запускати бізнес-логіку.
4. Велика спільнота та ресурси: Apex має велику та активну спільноту розробників Salesforce, що дозволяє знайти підтримку, документацію та відповіді на питання. Існує також багато ресурсів, таких як форуми, блоги та онлайн-курси, що допомагають розробникам вивчити та вдосконалити свої навички Apex.
5. Безпека та обмеження доступу: Apex дотримується механізмів безпеки Salesforce, що дозволяє контролювати доступ до даних та функціональності. Розробники можуть встановлювати права доступу, обмеження та правила безпеки для забезпечення конфіденційності та цілісності даних.
6. Тестування та деплоймент: Apex надає можливості для написання тестів, що допомагають перевірити правильність роботи коду перед його впровадженням. Крім того, Salesforce надає механізми для контролю версій та безпечного деплойменту коду.

Це лише кілька особливостей та переваг мови програмування Apex. Вона дозволяє розробникам розширювати та налаштовувати функціональність

Salesforce, створювати кастомні рішення та автоматизувати бізнес-процеси з використанням потужних інструментів, які надає ця мова.

1.5 Особливості використання тригерів у Apex

Тригери в Salesforce - потужний механізм, який дозволяє виконувати автоматичні дії на об'єктах Salesforce при виконанні певних подій. Тригери забезпечують можливість здійснювати автоматизацію бізнес-процесів та налаштовувати поведінку системи згідно з вимогами вашої організації.

Тригер - це блок коду, написаний на мові Apex, який автоматично виконується при виконанні певних подій на об'єктах Salesforce, таких як створення запису, оновлення запису, видалення запису тощо [9]. Є такі види тригерів: Before Triggers та Before Triggers. Before Triggers — тригери, що виконуються перед збереженням запису у базі даних. After Triggers — тригери, що виконуються після збереження запису у базі даних.

Тригер виконується на вказаному об’єкті після DML операції, наприклад: Insert, Update, Delete, Undelete. Тригери, зазвичай, використовують у таких випадках:

1. Валідація та перевірка даних перед збереженням.
2. Зміна значень полів на основі певних умов.
3. Виклик інших операцій або сервісів.
4. Створення або оновлення пов'язаних записів.
5. Реагування на зміни даних та спрацювання додаткової бізнес-логіки.

На рисунку 1.1 зображено приклад використання After Update Trigger, що оновлює спеціальне поле на записі облікового запису (Account), залежно від кількості пов'язаних записів контактів (Contact):

```
1   trigger UpdateAccountField on Contact (after insert, after update, after delete) {
2       List<Account> accountsToUpdate = new List<Account>();
3
4       if (Trigger.isInsert || Trigger.isUpdate) {
5           for (Contact contact : Trigger.new) {
6               accountsToUpdate.add(new Account(
7                   Id = contact.AccountId,
8                   CustomField__c = [SELECT COUNT() FROM Contact WHERE AccountId = :contact.AccountId]
9               ));
10          }
11      } else if (Trigger.isDelete) {
12          for (Contact contact : Trigger.old) {
13              accountsToUpdate.add(new Account(
14                  Id = contact.AccountId,
15                  CustomField__c = [SELECT COUNT() FROM Contact WHERE AccountId = :contact.AccountId]
16              ));
17          }
18      }
19
20      update accountsToUpdate;
21  }
22
```

Рисунок 1.1 - Приклад After Trigger в Apex

Тригери в Salesforce - це потужний механізм автоматизації та налаштування бізнес-процесів. Вони дозволяють реагувати на події, що відбуваються на об'єктах Salesforce, та виконувати дії відповідно до визначеної логіки.

1.6 Custom Setting і Custom Metadata в Salesforce

Custom Settings та Custom Metadata - це два різні механізми зберігання даних в Salesforce. Ці функціональності дозволяють зберігати настроювальні дані та метадані, які можна використовувати в своїх додатках та процесах. Вони надають можливість зручного управління настроювальними параметрами та метаданими без необхідності внесення змін у вихідний код.

Custom Settings - це механізм зберігання налаштувань в Salesforce [6]. Вони можуть бути використані для зберігання налаштувань, які повинні бути доступними для всього додатку. Custom Settings не можна пакувати та розгортати, а також не можна оновлювати на пісочницях навіть без даних. Є два види Custom Settings:

1. List Custom Setting - дозволяє створити настроювальний список з декількома значеннями.

2. Hierarchy Custom Setting - надає можливість створити ієрархію настроювальних параметрів, які можуть бути успадковані.

Зазвичай Custom Setting використовують для зберігання глобальних настроювальних параметрів додатку, зберігання URL-адреси сервісу, до якого потрібно виконувати запити та збереження конфігураційних параметрів, таких як часові рамки, ліміти.

На рисунку 1.2 зображено приклад використання Custom Setting:

```
1   // Створення List Custom Setting з назвою MySetting__c та полями Name і Value
2
3   // Клас для отримання значення з Custom Setting
4   public class MySettingUtil {
5       public static String getSettingValue(String settingName) {
6           return MySetting__c.getInstance(settingName).Value__c;
7       }
8   }
9
10  // Використання Custom Setting у класі Apex
11  public class MyApexClass {
12      public void doSomething() {
13          String settingValue = MySettingUtil.getSettingValue('SettingA');
14          // Використання значення з Custom Setting
15          System.debug('Value from Custom Setting: ' + settingValue);
16      }
17  }
18
```

Рисунок 1.2 - Приклад Custom Setting в Apex

Custom Metadata - це механізм зберігання метаданих в Salesforce [5]. Вони можуть бути використані для зберігання метаданих, які повинні бути доступними для всього додатку. Custom Metadata можна пакувати та розгортати, а також їх можна оновлювати на пісочницях навіть без даних. Custom Metadata має більше функцій порівняно з Custom Setting та менше функцій порівняно з Custom Object. Основною перевагою є те, що Custom Metadata має можливість деплоїти дані разом зі значеннями на організацією, Custom Settings такої можливості немає, дані потрібно заповнювати вручну або за допомогою Apex коду. Custom Metadata можуть бути використані в формулах та мати зв’язки з іншими об’єктами.

Зазвичай Custom Metadata використовують для збереження динамічних даних, які змінюються в процесі виконання додатка, збереження даних про конфігурацію, наприклад, розміщення екранів, налаштування зовнішнього вигляду тощо.

На рисунку 1.3 зображено приклад використання Custom Metadata:

```
1   // Створення Custom Metadata з назвою MyMetadata__mdt та полями Name і Value
2
3   // Клас для отримання значення з Custom Metadata
4   public class MyMetadataUtil {
5       public static String getMetadataValue(String metadataName) {
6           MyMetadata__mdt metadata = [SELECT Value__c FROM MyMetadata__mdt WHERE Name = :metadataName LIMIT 1];
7           return metadata.Value__c;
8       }
9   }
10
11  // Використання Custom Metadata у класі Apex
12  public class MyApexClass {
13      public void doSomething() {
14          String metadataValue = MyMetadataUtil.getMetadataValue('MetadataA');
15          // Використання значення з Custom Metadata
16          System.debug('Value from Custom Metadata: ' + metadataValue);
17      }
18  }
19
```

Рисунок 1.3 - Приклад Custom Metadata в Apex

Custom Setting і Custom Metadata є потужними інструментами для збереження настроювальних даних та метаданих у Salesforce. Вони дозволяють ефективно управляти налаштуваннями додатків та впроваджувати зміни без необхідності внесення змін у вихідний код. Розуміння їх використання та відмінностей допоможе вам створити більш гнучкі й ефективні рішення на платформі Salesforce.

Таблиця порівняння Custom Setting і Custom Metadata [4]:

	Custom Setting	Custom Metadata
Зміна даних в runtime	Так	Так
Доступ до метаданих	Неможливо	Можливо
Залежність від релізу	Ні	Ні
Можливість міграції	Ні	Так
Підтримка ієрархії	Ні	Так

1.7 Патерни проектування

Патерни проектування - це розроблені та визнані методики та рекомендації, які допомагають розробникам вирішувати типові проблеми при проектуванні програмного забезпечення [8]. Вони представляють собою високорівневі шаблони або розв'язки, які можна застосовувати для вирішення певного класу проблем у конкретному контексті.

Патерни проектування дозволяють розробникам створювати гнучкі, повторно використовувані та легко зрозумілі архітектурні рішення. Вони засновані на принципах об'єктно-орієнтованого програмування і надають шаблони для взаємодії між класами, об'єктами та компонентами системи.

Використання патернів дозволяє розділити функціональність програмного забезпечення на окремі компоненти, що полегшує внесення змін та розширення системи без впливу на інші компоненти. Використання патернів проектування допомагає враховувати кращі практики та стандарти розробки, що сприяє створенню високоякісного програмного забезпечення.

Приклади найпопулярніших патернів програмування [2]:

Singleton — використовується коли потрібно мати тільки один екземпляр класу, доступний з будь-якої точки програми. Приклад реалізації singleton патерну зображено на рисунку 1.4.

```
1   public class Singleton {
2       private static Singleton instance;
3
4       private Singleton() {
5           // Приватний конструктор
6       }
7
8       public static Singleton getInstance() {
9           if (instance == null) {
10              instance = new Singleton();
11          }
12          return instance;
13      }
14  }
```

Рисунок 1.4 - Приклад використання патерну проектування Singleton

Factory Method — використовується коли потрібно створювати об'єкти різних підкласів на основі специфічних параметрів. На рисунку 1.5 наведено приклад реалізації класу за допомогою патерну factory.

```
1   public abstract class Product {
2       // Абстрактний клас продукту
3       public abstract void operation();
4   }
5
6   public class ConcreteProductA extends Product {
7       // Конкретний продукт A
8       public void operation() {
9           // Логіка для продукту A
10      }
11  }
12
13  public class ConcreteProductB extends Product {
14      // Конкретний продукт B
15      public void operation() {
16          // Логіка для продукту B
17      }
18  }
19
20  public abstract class Creator {
21      // Абстрактний клас творця
22      public abstract Product createProduct();
23  }
24
25  public class ConcreteCreatorA extends Creator {
26      // Конкретний творець A
27      public Product createProduct() {
28          return new ConcreteProductA();
29      }
30  }
31
32  public class ConcreteCreatorB extends Creator {
33      // Конкретний творець B
34      public Product createProduct() {
35          return new ConcreteProductB();
36      }
37  }
```

Рисунок 1.5 - Приклад використання патерну проектування Factory

Використання патернів проектування є важливою практикою в розробці програмного забезпечення, оскільки вони допомагають розробникам побудувати ефективні, гнучкі та підтримувані системи.

1.8 Особливості середовища програмування Intellij Idea

Intellij Idea є одним з найпопулярніших та потужних середовищ програмування для розробки програмного забезпечення. Розроблений

компанією JetBrains, Intellij Idea надає широкі можливості для розробників, щоб працювати з різними мовами програмування та платформами. Серед особливостей можна відзначити:

1. Intellij Idea має потужний та інтуїтивно зрозумілий редактор коду, який надає розширений набір функцій, включаючи автодоповнення, підказки, рефакторинг, виправлення помилок та багато інших. Це дозволяє розробникам писати код швидше та зменшує кількість помилок.
2. Універсальна підтримка мов - Intellij Idea підтримує широкий спектр мов програмування, включаючи Java, Kotlin, JavaScript, TypeScript, PHP, Python, Ruby, HTML/CSS та інші. Це робить його ідеальним вибором для проектів з різними технологіями.
3. Intellij Idea постачається з багатьма інструментами, що полегшують розробку програмного забезпечення. Він має вбудовану систему контролю версій, інтеграцію з Git, підтримку збірки та розгортання проекту, систему автоматичного тестування, засоби аналізу коду та багато інших.
4. Розширення та плагіни - це середовище розробки підтримує розширення за допомогою плагінів, які дозволяють розширити функціональність середовища програмування.
5. Є можливість інтеграції з іншими інструментами, Intellij Idea інтегрується з багатьма іншими інструментами розробки, такими як системи збірки проекту (Maven, Gradle), сервери додатків (Tomcat, JBoss), фреймворки (Spring, Hibernate) та інтеграція з Illuminate Cloud. Intellij Idea забезпечує вбудовану підтримку для розробки з використанням Illuminate Cloud. Illuminate Cloud є хмарною платформою, спеціалізованою на управлінні проектами та забезпеченні швидкої розробки програмного забезпечення за допомогою Apex.

Intellij Idea є потужним та універсальним середовищем програмування, яке надає розробникам інструменти для швидкої та ефективної розробки програмного забезпечення. Його багатий функціонал, підтримка різних мов та

розширюваність роблять його популярним вибором серед розробників у всьому світі. Дизайн та функціонал середовища розробки Intellij Idea наведено на рисунку 1.6.

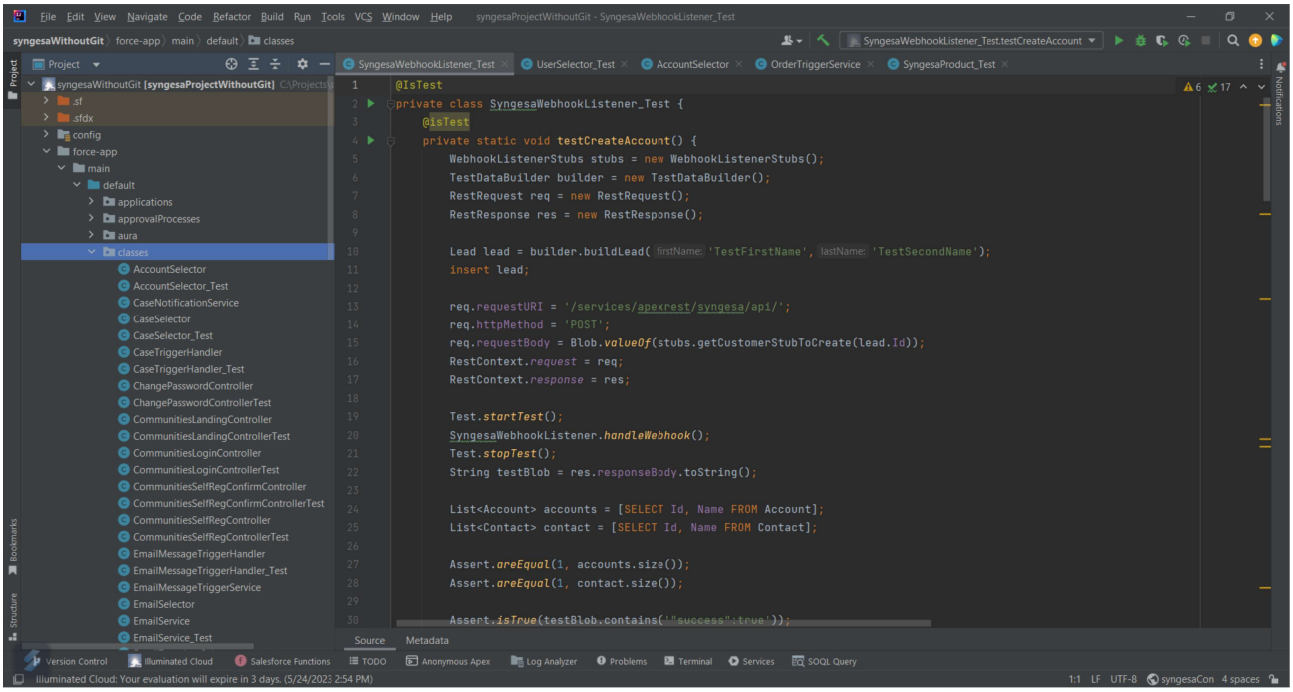

Рисунок 1.6 - Середовище розробки Intellij Idea

РОЗДІЛ 2. ПРОЦЕС СТВОРЕННЯ

2.1 Алгоритм роботи

У даному проекті для реалізації патерну програмування для тригерів потрібно спочатку створити та налаштувати організацію у CRM системі Salesforce. Після створення та налаштування організації ми отримаємо повноцінне середовище розробки, де в подальшому зможемо реалізовувати патерн.

У Salesforce є правила створення тригерів, серед них є правило, що на один об’єкт створюється тільки один тригер, для невеликих проектів цього достатньо. Проте системного обмеження немає, за бажання, користувач має змогу створити і більше тригерів для одного об’єкту, але тоді можуть виникати проблеми, а саме: тригери спрацьовуватимуть в будь-якому порядку, що може привезти до некоректних змін даних та виникнення помилок. У великих проектах це не припустимо, оскільки там може бути дуже багато бізнес логіки, і тригер спрацює не у визначеному порядку, то є можливість втрати або пошкодження даних, що для проектів та компаній не припустимо. Також можна створювати тільки один тригер, а у обробнику додати всю потрібну бізнес логіку. Цей підхід працюватиме, але тут теж є проблема, з часом у такому обробнику накопичиться дуже багато коду, який важко буде змінювати та модернізувати.

Створення цього патерну дозволить спростити роботу з тригерами, оскільки в нас буде кілька тригерів на одному об’єкті, де кожен відповідатиме за частину бізнес логіки. Окрім цього, матимемо змогу керувати порядком виконання тригерів.

2.2 Процес створення патерну програмування

Як вже згадувалось раніше потрібно створити патерн програмування для забезпечення більшої гнучкості та контролю над виконанням логіки тригерів, що допомагає підтримувати чистоту коду та уникати небажаних проблем, таких як рекурсивність.

Першим кроком створюємо звичайний тригер на об’єкті. Використаємо об’єкт Contact (Рисунок 2.1).

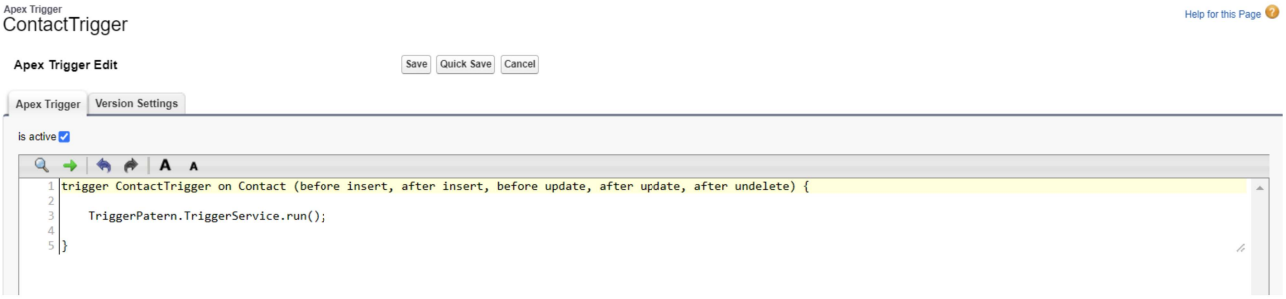

Рисунок 2.1 - Приклад тригеру на об'єкті Contact

Як видно на рисунку в тригері ми викликаємо TriggerService — це клас в якому і буде реалізовуватись основна логіка тригер патерну.

```
public static void run() {
    run(new Map<String, Object>());
}

public static void run(Map<String, Object> config) {
    if (skipAllHandlers == true) {
        return;
    }

    String mode, subscriber; {
        mode = BaseUtilities.coalesce(String.valueOf(config.get(CONFIG_PARAM_MODE)), MODE_NORMAL);
        subscriber = String.valueOf(config.get(CONFIG_PARAM_SUBSCRIBER));
    }

    if (BaseUtilities.isEquivalent(mode, MODE_EVENT_ASYNC)) {
        List<SObject> scope; {

            if (batchSize == null || batchSize <= 0) {
                scope = (List<SObject>)Trigger.new.clone();
            } else {
                scope = (List<SObject>)CollectionUtilities.limitTo(Trigger.new, batchSize);
                String replayId = String.valueOf(scope[scope.size() - 1].get('ReplayId'));
                EventBus.TriggerContext.currentContext().setResumeCheckpoint(replayId);
            }
```

Рисунок 2.2 - TriggerService клас

На рисунку 2.2 частина класу TriggerService, в якому є метод run, що запускає роботу патерну. Цей метод ми викликаємо з нашого Contact тригера. TriggerService клас розширює сервіс клас, в якому буде описана алгоритм роботи тригера. Перевагою цього патерну є те, що його можна використовувати на будь-якому об’єкті, немає прив’язки до певних об’єктів. Після виклику тригера, потрібно визначити, який тип DML операції це зробив, щоб правильно визначити метод обробки. Більш детально алгоритм роботи можна побачити на рисунку 2.3:

```
public static void callHandler(TriggerContext ctx, TriggerHandler h) {

    String objectType = String.valueOf(ctx.getSObjectType());

    String handlerClassName = h.getName();


    Integer cpuTimeStart = Limits.getCpuTime();
    Integer dmlRowsStart = Limits.getDMLRows();
    Integer dmlStatementsStart = Limits.getDMLStatements();
    Integer queriesStart = Limits.getQueries();
    Integer queryRowsStart = Limits.getQueryRows();

    Long startTime = System.now().getTime();


    String phaseTag = null;

    String startHeader = '    '.repeat(level) + 'Trigger Handler Metrics: START {0}: ObjectType = ' + objectType + ', HandlerName = ' + handlerClassName;

    if (!ctx.getIsSelect()) {
        level++;
    }

    if (ctx.getIsBefore()) {

        if (ctx.getIsInsert()) {

            phaseTag = 'BEFORE INSERT';

            System.debug(
                LoggingLevel.WARN,
                String.format(
                    startHeader,
                    new List<String> {
                        phaseTag
                    }
                )
            );

            h.beforeInsert(ctx);
```

Рисунок 2.3 - Метод, що визначає контекст з якого викликають тригер

В методі виконуються спочатку перевірки для визначення типу операції, чи before чи after, а також, після якої операції викликається тригер, на цьому рисунку вказано для before insert. Коли контекст з якого викликається тригер вже визначено, переходимо до наступного кроку — створення екземпляру класу, для потрібного нам класу обробника.

Це можна реалізувати за допомогою Custom Metadata: в тригер сервісі ми дістаємо записи custom metadata та за допомогою системного методу Type.forName визначатиме тип класу обробника для тригера. Реалізацію цього функціоналу наведено на рисунку 2.4.

```
public static List<TriggerHandler> createHandlers(Context ctx) {
    List<TriggerHandler> handlers = new List<TriggerHandler>();

    for (HandlerWrapper hw : getHandlersPerObjectType(ctx)) {

        try {

            Type handlerType = ClassUtilities.getType(hw.name);

            if (handlerType != null) {

                TriggerHandler instance = (TriggerHandler)handlerType.newInstance();

                instance.init(hw.name, hw.sType, hw.params, hw.isAsync);

                handlers.add(instance);
            } else {

                System.debug(
                    LoggingLevel.ERROR,
                    'TriggerService::onTrigger can\'t instantiate handler for setting: ' + hw
                );
            }

        } catch (Exception exp) {

            System.debug(
                LoggingLevel.ERROR,
                'TriggerService::onTrigger exception instantiating handler for setting: ' + hw + ' Exception: ' + exp.getMessage()
            );
        }
    }

    return handlers;
}
```

Рисунок 2.4 - Метод, що створює потрібний обробник для операції, що виконується в тригері

Якщо на одному об’єкті використовується кілька тригерів, то їх виконання відбуватиметься відповідно до порядку, який користувач вказав при створені запису в Custom Metadata. Також перевіряється чи обробник тригера зараз активний (Рисунок 2.5).

Apex Class	Order	Is Active
AccountPrimaryContactAction	1	✓
ContactPrimaryContactAction	1	✓
LiveFeedbackAction	1	✓

Рисунок 2.5 - Визначення порядку виконання та статусу обробника для тригера

Порядок виконання та статус тригера визначено, переходимо до наступного кроку. Коли ми вже визначили, який саме клас обробник нам потрібно, створюємо його екземпляр та виконуємо логіку, що описана в ньому. В тригері може бути вказана будь-яка бізнес логіка, проте зазвичай тригери використовують, щоб змінити дані в залежних записах на інших об’єктах, для створення нових записів на залежних об’єктах або для надсилання сповіщень користувачам, власникам записів.

2.3 Демонстрація роботи проекту

Після реалізації патерну користувач матиме змогу створювати кілька тригерів для одного об’єкту. Окрім цього є можливість вмикати або вимикати потрібно тригери та визначати порядок їх виконання. Далі опишемо кроки, які потрібно використати користувачу для налаштування тригер патерну:

1. В першу чергу користувач повинен додати розроблений нами тригер патерн до свого проекту, поки це можна зробити додавши вручну, проте є змога цей код винести в пакет. Таким чином користувач просто встановить пакет з кодом, не додаючи кожен файл окремо до свого проекту. Також це краще з точки зору безпеки, бо таким чином кінцевий користувач не матиме змоги редагувати код патерну, а просто використовуватиме його.
2. Створюємо, наприклад, тригер на об’єкті Contact, назвемо його ContactTrigger, та визначимо перед якими або після яких DML операцій тригер спрацьовуватиме. У нашому випадку це будуть такі операції: before insert, after insert, before update, after update, after undelete. На рисунку 2.6 показано процес створення ContactTrigger тригера.

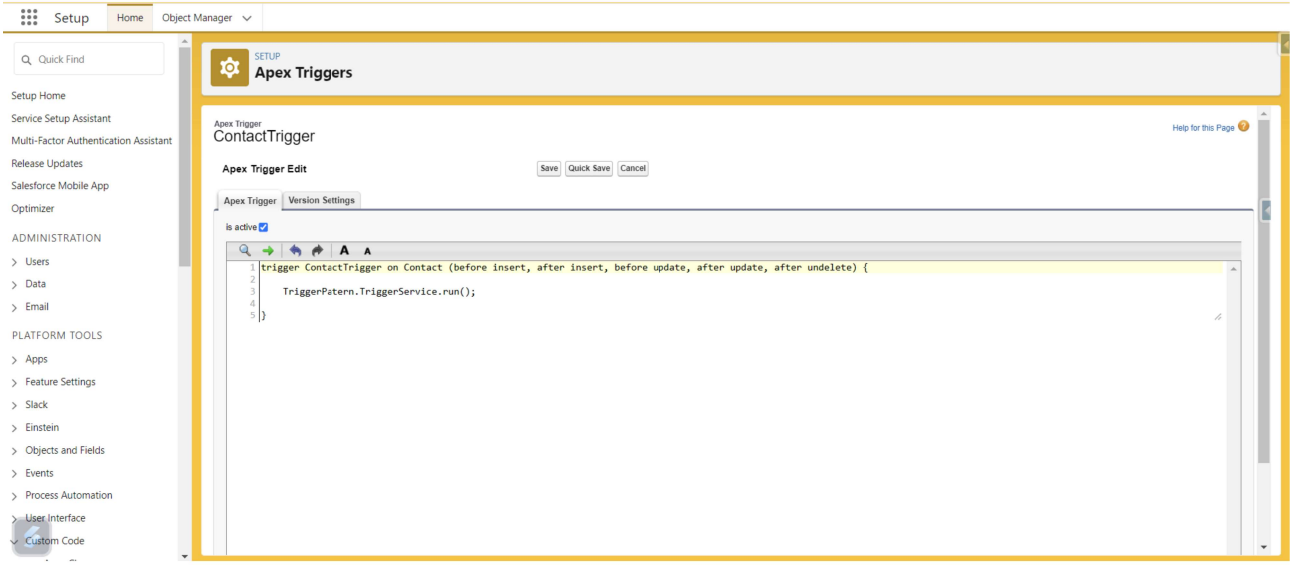

Рисунок 2.6 - Приклад тригеру на об'єкті Contact

3. Наступним кроком потрібно створити Custom Metadata оскільки за допомогою саме цього ми діставати потрібний нам тригер та викликатимемо його.

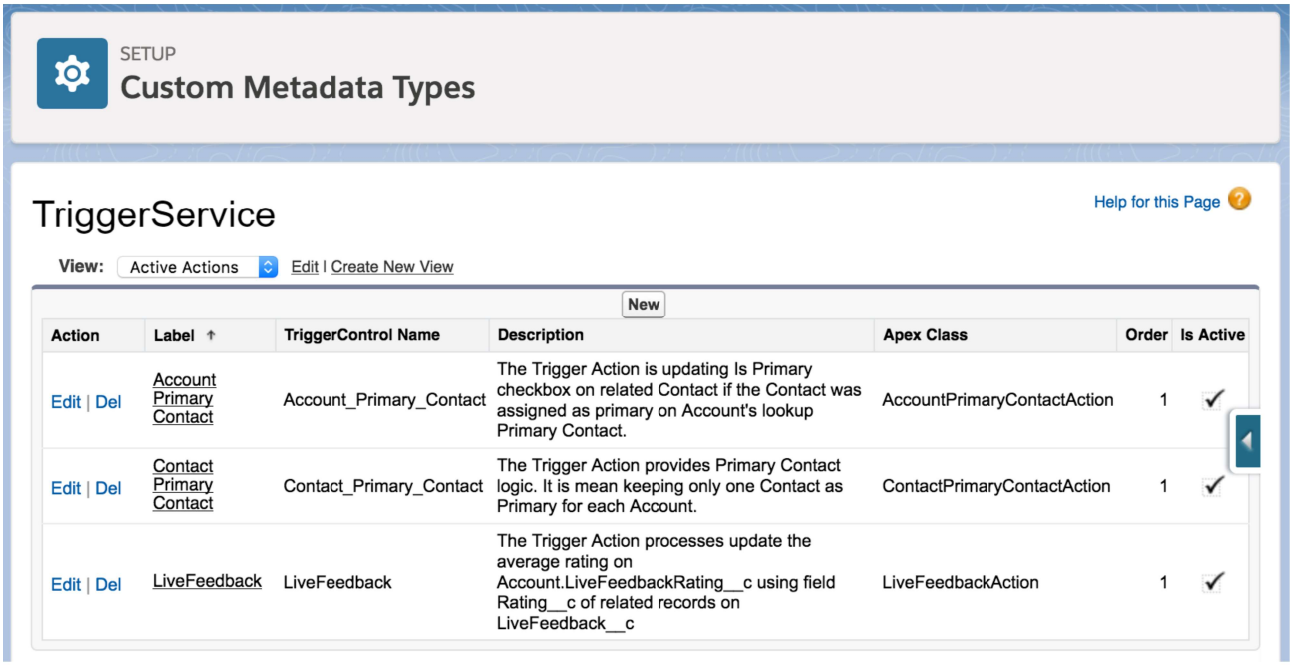

Рисунок 2.7 - Приклад Custom Metadata рекорду

4. Створюючи custom metadata рекорд ми визначили клас обробник, що викликатиметься після спрацювання тригера, порядок виконання тригерів та поле Is Active, що дає змогу користувачу деактивовувати тригер, це може бути корисно, коли потрібно за допомогою стороннього API додати велику кількість записів.

Таким чином користувачу потрібно виконати всього три кроки для імплементації створеного патерну програмування для реалізації тригерів у CRM-системі.

ВИСНОВКИ

Під час виконання курсової роботи я проаналізував основні принципи та патерни програмування, а також розібрав особливості реалізації тригерів у CRM-системах, розглянув різні аспекти створення тригерів.

Об’єктом курсової роботи став патерн програмування, для впровадження тригерів у CRM-системи, за допомогою якого можна автоматизувати операції обробників та спростити управління бізнес-процесами. Використання патернів дозволяє підтримувати структуру коду, забезпечувати його читабельність та облегшує підтримку системи у майбутньому. Для розробки патерну програмування було використано мову програмування Apex та CRM-систему Salesforce. Виконуючи цю курсову роботу я отримав багато досвіду у роботі з тригерами, обробниками та вбудованим функціоналом CRM-системи.

Наступним кроком буде створення керованого пакету у який буде додано код патерну, щоб користувачі могли швидко та легко налаштовувати своє середовище розробки та використовувати патерн.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. A Beginner's Guide to CRM Systems (електронний ресурс) – https://www.salesforce.com/eu/learning-centre/crm/crm-systems/
2. Apex Design Patterns (електронний ресурс) – https://www.apexhours.com/apex-design-patterns/
3. Apex Developer Guide (електронний ресурс) – https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_dev_guide.htm
4. Difference between custom setting and custom metadata types (електронний ресурс) – https://stories.salesforcecodex.com/2022/07/salesforce/custom-setting-and-custom-metadata-type/
5. Custom Metadata in Salesforce (електронний ресурс) – https://developer.salesforce.com/docs/atlas.en-us.sfFieldRef.meta/sfFieldRef/salesforce_field_reference_CustomMetadataType.htm
6. Custom Setting in Salesforce (електронний ресурс) – https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_customsettings.htm
7. Salesforce Apex Best Practices (електронний ресурс) – https://www.salesforceben.com/12-salesforce-apex-best-practices/
8. Software design pattern (електронний ресурс) – https://en.wikipedia.org/wiki/Software_design_pattern
9. Triggers in Salesforce (електронний ресурс) – https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_triggers.htm
10. What is Salesforce? (електронний ресурс) – https://www.salesforce.com/products/what-is-salesforce/

ДОДАТОК 1 Реалізація алгоритму створення патерну програмування з використанням мови Apex та CRM-системи Salesforce

TriggerService.cls

```
public class TriggerService {
   private static void resetHandlersCache() {
      allHandlerWrappers = null;
   }

   private static List<TriggerHandler__mdt>
applyUserPermissions(List<TriggerHandler__mdt> handlers, Set<String>
permissions) {
      List<TriggerHandler__mdt> allowedHandlers = new
List<TriggerHandler__mdt>();
      Set<String> inversedPermissions = inversePermissions(permissions);

      for (TriggerHandler__mdt handler : handlers) {
         if (String.isBlank(handler.PermissionLevel__c)) {
            continue;
         }

         if (handler.PermissionLevel__c.contains(BaseUtilities.COMMA)) {

            Set<String> handlerPermissions =
               BaseUtilities.toSet(

handler.PermissionLevel__c.toLowerCase().split(BaseUtilities.CSV_DELIMITER_R
EGEX)
               )
            ;
            if (
               BaseUtilities.intersect(
                  permissions,
                  handlerPermissions
               ).isEmpty()
               ||
               !
               BaseUtilities.intersect(
                  inversedPermissions,
                  handlerPermissions
               ).isEmpty()
```

```
            ) {
                continue;
            }
            allowedHandlers.add(handler);
        } else {

            if (permissions.contains(handler.PermissionLevel__c.toLowerCase())) {
                allowedHandlers.add(handler);
            }
        }
    }

    return allowedHandlers;
  }

  public static void skipHandlersForObject(SObjectType objectTypeToSkip) {
    skipHandlersForObject(objectTypeToSkip, PERMISSION_LEVEL);
  }

  public static void skipHandlersForObject(SObjectType objectTypeToSkip, String
permissionLevel) {
    TriggerHandler__mdt skipHandler =
        new TriggerHandler__mdt(
            HandlerName__c = TriggerService.SkipAllHandler.class.getName(),
            IsActive__c = true,
            Description__c = 'Handler to skip processing of all next handlers in a
chain.',
            Type__c = String.valueOf(objectTypeToSkip),
            PermissionLevel__c = permissionLevel,
            IsAsync__c = false,
            Params__c = null,
            Subscriber__c = null
        )
    ;

    prependStaticHandler(
        skipHandler
    );
  }

  public static String onBeforeSelect(String query) {
    Context ctx = new Context(query);
    executeActions(ctx);

    return ctx.query;
```

```
    }

    public static List<SObject> onAfterSelect(List<SObject> records) {
        Context ctx = new Context(records);
        executeActions(ctx);

        return ctx.records;
    }

    public static List<SObject> onBeforeInsert(List<SObject> records) {
        Context ctx = getBeforeInsertVirtualContext(records);
        executeActions(ctx);

        return ctx.newList;
    }


    public static List<SObject> onAfterInsert(List<SObject> records) {
        Context ctx = getAfterInsertVirtualContext(records);
        executeActions(ctx);

        return ctx.newList;
    }

    public static void run() {
        run(new Map<String, Object>());
    }

    public static void run(Map<String, Object> config) {
        if (skipAllHandlers == true) {
            return;
        }

        String mode, subscriber; {
            mode =
BaseUtilities.coalesce(String.valueOf(config.get(CONFIG_PARAM_MODE)),
MODE_NORMAL);
            subscriber = String.valueOf(config.get(CONFIG_PARAM_SUBSCRIBER));
        }

        if (BaseUtilities.isEquivalent(mode, MODE_EVENT_ASYNC)) {
            List<SObject> scope; {

                if (batchSize == null || batchSize <= 0) {
                    scope = (List<SObject>)Trigger.new.clone();
```

```
            } else {
                scope = (List<SObject>)CollectionUtilities.limitTo(Trigger.new,
batchSize);
                String replayId = String.valueOf(scope[scope.size() - 1].get('ReplayId'));

EventBus.TriggerContext.currentContext().setResumeCheckpoint(replayId);
            }
        }

        Context ctx; {
            ctx =
                new Context(
                    null,
                    null,
                    null,
                    null,
                    Trigger.isBefore,
                    Trigger.isAfter,
                    Trigger.isInsert,
                    Trigger.isUpdate,
                    Trigger.isDelete,
                    Trigger.isUndelete,
                    subscriber
                )
            ;

            ctx.isVirtual = true;
            ctx.mode = mode;

        }

    } else if (BaseUtilities.isEquivalent(mode, MODE_NORMAL)) {

        if (!Trigger.isExecuting) {
            throw
                new TriggerServiceException(Label.Error_Not_Trigger_Context);
        }

        Context ctx =
            new Context(
                Trigger.old,
                Trigger.oldMap,
                Trigger.new,
                Trigger.newMap,
                Trigger.isBefore,
```

```
            Trigger.isAfter,
            Trigger.isInsert,
            Trigger.isUpdate,
            Trigger.isDelete,
            Trigger.isUndelete,
            subscriber
        );

    ctx.mode = mode;

    executeActions(ctx);
  } else {
    throw new TriggerServiceException('Unsupported mode: ' + mode);
  }
}

public static void executeAct(Context ctx) {
  if (skipAllHandlers == true) {
    return;
  }

  SObjectType objectType = ctx.getSObjectType();

  if (objectType == null) {
    return;
  }

  List<TriggerHandler> handlers =
    createHandlers(ctx);

  for (TriggerHandler h : handlers) {

    if (h instanceof TriggerService.Handler) {

      TriggerService.Handler localHandler = (TriggerService.Handler)h;

      if (localHandler.skipAllHandlersPre(ctx)) {
        break;
      }
    }

    if (h.skipExecution(ctx)) {
      continue;
    }
```

```
        if (
            h.isAsyncMode(ctx) == true
            &&
            System.isQueueable() == false
            &&
            System.isBatch() == false
            &&
            Limits.getQueueableJobs() < Limits.getLimitQueueableJobs()
            &&
            (ctx.isAfter == true && ctx.isSelect != true)
        ) {
            try {

                System.enqueueJob(
                    new HandlerRunnerAsync(
                        ctx,
                        h
                    )
                );

            } catch (Exception exp) {
                callHandler(ctx, h);
            }

        } else {
            callHandler(ctx, h);
        }

        if (h instanceof TriggerService.Handler) {
            TriggerService.Handler localHandler = (TriggerService.Handler)h;

            if (localHandler.skipAllHandlersPost(ctx)) {
                break;
            }
        }
    }
}

public static void callHandler(TriggerContext ctx, TriggerHandler h) {

    String objectType = String.valueOf(ctx.getSObjectType());

    String handlerClassName = h.getName();
```

```
Integer cpuTimeStart = Limits.getCpuTime();
Integer dmlRowsStart = Limits.getDMLRows();
Integer dmlStatementsStart = Limits.getDMLStatements();
Integer queriesStart = Limits.getQueries();
Integer queryRowsStart = Limits.getQueryRows();

Long startTime = System.now().getTime();


String phaseTag = null;

String startHeader = '    '.repeat(level) + 'Trigger Handler Metrics: START {0}:
ObjectType = ' + objectType + ', HandlerName = ' + handlerClassName;

if (!ctx.getIsSelect()) {
    level++;
}

if (ctx.getIsBefore()) {

    if (ctx.getIsSelect()) {

        phaseTag = 'BEFORE SELECT';

        System.debug(
            LoggingLevel.WARN,
            String.format(
                startHeader,
                new List<String> {
                    phaseTag
                }
            )
        );

        h.beforeSelect(ctx);

    } else if (ctx.getIsInsert()) {

        phaseTag = 'BEFORE INSERT';

        System.debug(
            LoggingLevel.WARN,
            String.format(
                startHeader,
                new List<String> {
```

```
                phaseTag
            }
        )
    );

    h.beforeInsert(ctx);

} else if (ctx.getIsUpdate()) {

    phaseTag = 'BEFORE UPDATE';

    System.debug(
        LoggingLevel.WARN,
        String.format(
            startHeader,
            new List<String> {
                phaseTag
            }
        )
    );

    h.beforeUpdate(ctx);

} else if (ctx.getIsDelete()) {

    phaseTag = 'BEFORE DELETE';

    System.debug(
        LoggingLevel.WARN,
        String.format(
            startHeader,
            new List<String> {
                phaseTag
            }
        )
    );

    h.beforeDelete(ctx);

} else if (ctx.getIsUndelete()) {

    phaseTag = 'BEFORE UNDELETE';

    System.debug(
        LoggingLevel.WARN,
```

```
            String.format(
                startHeader,
                new List<String> {
                    phaseTag
                }
            )
        );

        h.beforeUndelete(ctx);
    }


} else if (ctx.getIsAfter()) {
    if (ctx.getIsSelect()) {

        phaseTag = 'AFTER SELECT';

        System.debug(
            LoggingLevel.WARN,
            String.format(
                startHeader,
                new List<String> {
                    phaseTag
                }
            )
        );

        if (ctx.getIsCount()) {

            h.afterCount(ctx);

        } else if (ctx.getIsAggregate()) {

            h.afterAggregate(ctx);

        } else {

            h.afterSelect(ctx);

        }

    } else if (ctx.getIsInsert()) {

        phaseTag = 'AFTER INSERT';
```

```
        System.debug(
            LoggingLevel.WARN,
            String.format(
                startHeader,
                new List<String> {
                    phaseTag
                }
            )
        );

        h.afterInsert(ctx);

    } else if (ctx.getIsUpdate()) {

        phaseTag = 'AFTER UPDATE';

        System.debug(
            LoggingLevel.WARN,
            String.format(
                startHeader,
                new List<String> {
                    phaseTag
                }
            )
        );

        h.afterUpdate(ctx);

    } else if (ctx.getIsDelete()) {

        phaseTag = 'AFTER DELETE';

        System.debug(
            LoggingLevel.WARN,
            String.format(
                startHeader,
                new List<String> {
                    phaseTag
                }
            )
        );

        h.afterDelete(ctx);

    } else if (ctx.getIsUndelete()) {
```

```
            phaseTag = 'AFTER UNDELETE';

            System.debug(
                LoggingLevel.WARN,
                String.format(
                    startHeader,
                    new List<String> {
                        phaseTag
                    }
                )
            );

            h.afterUndelete(ctx);
        }
    }

    Integer cpuTimeEnd = Limits.getCpuTime();
    Integer dmlRowsEnd = Limits.getDMLRows();
    Integer dmlStatementsEnd = Limits.getDMLStatements();
    Integer queriesEnd = Limits.getQueries();
    Integer queryRowsEnd = Limits.getQueryRows();

    Long endTime = System.now().getTime();
    duration += endTime - startTime;

    if (!ctx.getIsSelect()) {
        level--;
    }

}

private static String getSObjectType(Context cont) {
    return (cont.isDelete ? cont.oldList :
cont.newList).getSObjectType().getDescribe().getName();
}

private static List<HandlerWrapper> getHandlersPerObjectType(Context ctx) {

    String objectName = String.valueOf(ctx.getSObjectType());

    Set<String> objectTypesCriteria =
        new Set<String> {
            objectName,
            ALL_OBJECT_TYPES
```

```
        };

    if (ObjectUtilities.isPlatformEvent(objectName)) {
        objectTypesCriteria.add(
            ALL_EVENT_TYPES
        );
    }

    List<HandlerWrapper> handlerWrappers = new List<HandlerWrapper>(); {
        for (HandlerWrapper handlerWrapper : allActiveHandlerWrappers) {
            if (
                objectTypesCriteria.contains(handlerWrapper.sType)
                &&
                handlerWrapper.subscriber == ctx.subscriber
            ) {
                handlerWrappers.add(handlerWrapper);
            }
        }
    }

    handlerWrappers.sort();

    return handlerWrappers;
}


public static List<TriggerHandler> createHandlers(Context ctx) {
    List<TriggerHandler> handlers = new List<TriggerHandler>();

    for (HandlerWrapper hw : getHandlersPerObjectType(ctx)) {

        try {

            Type handlerType = ClassUtilities.getType(hw.name);

            if (handlerType != null) {

                TriggerHandler instance = (TriggerHandler)handlerType.newInstance();

                instance.init(hw.name, hw.sType, hw.params, hw.isAsync);

                handlers.add(instance);
            } else {

                System.debug(
```

```
                LoggingLevel.ERROR,
                'TriggerService::onTrigger can\'t instantiate handler for setting: ' + hw
            );
        }

    } catch (Exception exp) {

        System.debug(
            LoggingLevel.ERROR,
            'TriggerService::onTrigger exception instantiating handler for setting: ' +
hw + ' Exception: ' + exp.getMessage()
        );
    }
}

return handlers;
}

public virtual class Handler implements TriggerHandler {

    protected Map<String, Object> params = new Map<String, Object>();

    protected String name;

    public void init(string name, string sType, string params, boolean isAsync) {
        this.params = BaseUtilities.isNullOrEmpty(params) ?
            new Map<String, Object>() : JsonUtilities.toObject(params);
        this.name = name;
        this.objectName = sType;
        this.asyncMode = isAsync;
    }

    public virtual Boolean skipAllHandlersPre(Context cont) {
        return false;
    }

    public virtual Boolean skipAllHandlersPost(Context cont) {
        return false;
    }

    public virtual Boolean skipExecution(Context cont) {
        return false;
    }

    public virtual Boolean isAsyncMode(Context cont) {
```

```
        return asyncMode;
    }

    public Map<String, Object> getParams() {
        return params;
    }


    public String getStringParam(String paramName) {
        return (params <> null && params.containsKey(paramName))
            ? String.valueOf(params.get(paramName))
            : null;
    }


    public Integer getIntegerParam(String paramName) {
        return (params <> null && params.containsKey(paramName))
            ? NumberUtilities.toInteger(params.get(paramName), null)
            : null;
    }

    public virtual void beforeSelect(Context cont) {}
    public virtual void afterSelect(Context cont) {}
    public virtual void afterCount(Context cont) {}
    public virtual void afterAggregate(Context cont) {}
    public virtual void beforeInsert(Context cont) {}
    public virtual void afterInsert(Context cont) {}
    public virtual void beforeUpdate(Context cont) {}
    public virtual void afterUpdate(Context cont) {}
    public virtual void beforeDelete(Context cont) {}
    public virtual void afterDelete(Context cont) {}
    public virtual void beforeUndelete(Context cont) {}
    public virtual void afterUndelete(Context cont) {}


    public string getName() { return this.name; }

    public boolean skipExecution(TriggerContext context) { return
skipExecution((Context)context); }
    public boolean isAsyncMode(TriggerContext context) { return
isAsyncMode((Context)context); }

    public void beforeInsert(TriggerContext context)
{ beforeInsert((Context)context); }
    public void afterInsert(TriggerContext context) { afterInsert((Context)context); }
```

```
        public void beforeUpdate(TriggerContext context)
{ beforeUpdate((Context)context); }
        public void afterUpdate(TriggerContext context)
{ afterUpdate((Context)context); }
        public void beforeDelete(TriggerContext context)
{ beforeDelete((Context)context); }
        public void afterDelete(TriggerContext context) { afterDelete((Context)context);
}
        public void beforeUndelete(TriggerContext context)
{ beforeUndelete((Context)context); }
        public void afterUndelete(TriggerContext context)
{ afterUndelete((Context)context); }

        public void beforeSelect(TriggerContext context)
{ beforeSelect((Context)context); }
        public void afterSelect(TriggerContext context)
{ afterSelect((Context)context); }
        public void afterCount(TriggerContext context)
{ afterCount((Context)context); }
        public void afterAggregate(TriggerContext context)
{ afterAggregate((Context)context); }
    }

    public virtual class EventHandler extends Handler {


        private void handle(List<SObject> events, Context ctx) {
            List<SObject> eventsToProcess = events;

            if (this.isChunkMode(ctx)) {

                Integer chunkSize = this.getChunkSize();

                eventsToProcess = (List<SObject>)CollectionUtilities.limitTo(events,
chunkSize);

                if (events.size() > chunkSize) {
                    EventBus.TriggerContext.currentContext().setResumeCheckpoint(
                        (String)eventsToProcess[chunkSize - 1].get('replayId')
                    );
                }

            }
            SavePoint mainSavePoint = (this.rollbackOnFailure() == true) ?
Database.setSavepoint() : null;
```

```
        try {
            this.process(eventsToProcess, ctx);
        } catch (Exception exp) {
            if (mainSavePoint != null) {
                Database.rollback(mainSavePoint);
            }

            if (this.isRetryMode(ctx)) {
                Integer maxRetryAttempts = this.getMaxRetryAttempts();
                Integer retryAttemptNumber =
EventBus.TriggerContext.currentContext().retries;

                if (retryAttemptNumber < maxRetryAttempts) {

                    throw new EventBus.RetryableException(
                        String.format(
                            RETRY_MESSAGE_TEMPLATE,
                            new List<String> {
                                String.valueOf(
                                    retryAttemptNumber
                                ),
                                String.valueOf(
                                    maxRetryAttempts
                                )
                            }
                        )
                    );
                }
            }

            try {
                LogService.Logger logger = LogService.getLogger();
                logger.error(
                    'Run out of retry attempts. Exception when processing event: ' +
exp.getMessage(),
                    TriggerService.EventHandler.class.getName()
                );

                logger.persistLogs();

            } catch (Exception innerException) {
                System.debug(
                    System.LoggingLevel.ERROR,
```

```
                TriggerService.EventHandler.class.getName() + ': Run out of retry
attempts. Exception when processing event: ' +  exp.getMessage()
            );
        }
    }
}

public class Context implements TriggerContext {
    public Integer count { get; set; }
    public SObjectType sourceType { get; set; }

    public String query { get; set; }
    public List<SObject> records { get; set; }

    public List<SObject> oldList { get; set; }
    public Map<Id, SObject> oldMap { get; set; }

    public List<SObject> newList { get; set; }
    public Map<Id, SObject> newMap { get; set; }

    public Boolean isBefore { get; set; }
    public Boolean isAfter { get; set; }
    public Boolean isSelect { get; set; }
    public Boolean isInsert { get; set; }
    public Boolean isUpdate { get; set; }
    public Boolean isDelete { get; set; }
    public Boolean isUndelete { get; set; }
    public String subscriber { get; set; }
    public String mode { get; set; }
    public Map<String, Object> globalScope { get; set; }

    public Context(Integer count, SObjectType sourceType) {
        this();
        this.isSelect = true;
        this.isCount = true;
        this.isAfter = true;
        this.isVirtual = true;
        this.count = count;
        this.sourceType = sourceType;
    }


public class HandlerWrapper implements Comparable {

    public String name { get; set; }
```

```
        public Boolean active { get; set; }
        public String description { get; set; }
        public Integer order { get; set; }
        public String sType { get; set; }
        public String permission { get; set; }
        public Boolean isAsync { get; set; }
        public String params { get; set; }
        public String subscriber { get; set; }

        public HandlerWrapper(String name, Boolean active, String description, Integer
order, String sType, String permission) {
            this(name, active, description, order, sType, permission, false);
        }

        public HandlerWrapper(String name, Boolean active, String description, Integer
order, String sType, String permission, Boolean isAsync) {
            this(name, active, description, order, sType, permission, false, null);
        }


        public HandlerWrapper(String name, Boolean active, String description, Integer
order, String sType, String permission, Boolean isAsync, String params) {
            this(name, active, description, order, sType, permission, isAsync, params,
null);
        }


        public HandlerWrapper(String name, Boolean active, String description, Integer
order, String sType, String permission, Boolean isAsync, String params, String
subscriber) {
            this.name = name;
            this.active = active;
            this.description = description;
            this.order = order;
            this.sType = sType;
            this.permission = permission;
            this.isAsync = isAsync;
            this.params = params;
            this.subscriber = subscriber;
        }
    }

    private class TriggerServiceException extends Exception {  }
```

```
    public class HandlerRunnerAsync implements Queueable,
Database.AllowsCallouts {
        private TriggerContext ctx;
        private TriggerHandler hnd;

        public HandlerRunnerAsync(TriggerContext ctx, TriggerHandler hnd) {
            this.ctx = ctx;
            this.hnd = hnd;
        }


        public void execute(QueueableContext asyncContext) {
            callHandler(ctx, hnd);
        }

    }

    public inherited sharing class SkipAllHandler extends TriggerService.Handler {
        public override Boolean skipAllHandlersPre(Context cont) {
            return true;
        }
    }
}
```