КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ТАРАСА ШЕВЧЕНКА

факультет інформаційних технологій

кафедра програмних систем і технологій

На правах рукопису

УДК___________

ВИПУСКНА КВАЛІФІКАЦІЙНА МАГІСТЕРСЬКА РОБОТА

Тема Дослідження інтеграційних процесів між системи управління даними

Спеціальність 121 Інженерія програмного забезпечення»

Виконав студент

Курмаз Ілля Владиславович

Науковий керівник

к.ф-н, доцент Доценко Сергій Іванович

Робота допущена до захисту

на засіданні кафедри протокол № _ від «__» __ 2021

завідувач кафедри Програмних систем і технологій

____

2021

Рішенням Екзаменаційної комісії

випускна кваліфікаційна робота студента

____

захищена з оцінкою

____

Голова Екзаменаційної комісії

професор, доктор техн. наук Онищенко В.В

Форма завдання на випускну кваліфікаційну магістерську роботу

Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій

Кафедра програмних систем і технологій

Спеціальність 121 Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:

Завідувач кафедри

програмних систем і технологій

________________(О.С.Бичков)

„___”_________________ 20___р.

ЗАВДАННЯ

НА ВИПУСКНУ КВАЛІФІКАЦІЙНУ МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Курмазу Іллі Владиславовичу

1. Тема випускної кваліфікаційної магістерської роботи Дослідження і розробка системи інтеграції CRM та ERP систем

затверджені наказом вищого навчального закладу від „11” листопада 2020р. № 6

2. Строк здачі студентом закінченої роботи ______________________________

3. Вихідні дані до роботи ____

____

____

____

____

____

4. Зміст пояснювальної записки (перелік питань, що їх належить розробити)

____

____

____

Форма завдання на випускну кваліфікаційну магістерську роботу

6. Консультанти з роботи із зазначенням розділів роботи, що їх стосуються

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання ____________

Керівник ____________ ________________

(підпис) (розшифровка підпису)

Завдання прийняв до виконання ____________ ________________

(підпис) (розшифровка підпису)

АНОТАЦІЯ

Випускна кваліфікаційна магістерська робота: 76 сторінки, 9 рисунків, 1 додаток, 11 джерел.

Тема: Дослідження інтеграційних процесів між системи управління даними.

Об'єкт дослідження: інтеграційні процеси між системами управління даними

Предмет дослідження: інтеграція між системами управління даними

Мета дослідження: Дослідити інтеграційні процеси між системами управління даними та побудувати на їх основі інтеграційний застосунок.

Задачі дослідження:

– Дослідження існуючих систем та підходів до побудови інтеграційних процесів
– реалізація уніфікованої системи для інтеграції з можливістю побудови користувацьких процесів

Результати дослідження: Реалізовано інтеграційний застосунок, що працює на розробленому стандарті формату даних та їх передачі.

Висновок: Поставлені задачі були виконані. Результатом дипломної роботи є робочий додаток, що реалізує розроблений стандарт інтегрування даних. Застосунок є придатним до використання та відкритим для розширення функціоналу.

Ключові слова: Інтеграція, інтеграційний процес, дані, CRM, ERP, керування даними

АННОТАЦИЯ

Выпускная квалификационная магистерская работа: 76 страницы, 9 рисунков, 1 приложения, 11 источника.

Тема: Исследование интеграционных процессов в системах управления данными

Объект исследования: интеграционные процессы в системах управления данными

Предмет исследования: интеграция между системами управления данными

Цель исследования: исследовать интеграционные процессы между системами управления данными и

Задачи исследования:

- Исследование существующих систем и подходов к построению интеграционных процессов

- реализация унифицированной системы для интеграции с возможностью построения пользовательских процессе

Результаты исследования: Реализовано интеграционный приложение, работающее на разработанном стандарте формата данных и их передачи

Вывод: Поставленные задачи были выполнены. Результатом работы является рабочий приложение, реализующее разработан стандарт интеграции данных. Приложение является пригодным к использованию и открытым для расширения функционала

Ключевые слова: Интеграция, интеграционный процесс, данные, CRM, ERP, управление данными.

ABSTRACT

Graduation qualification master's thesis: 76 pages, 9 figures, 1 complement, 11 sources.

Subject: Research of integration processes between data management systems.

Object of the research: integration processes between data management systems

Subject of the research: integration between data management systems

Purpose of the research: Investigate the integration processes between data management systems and build an integration application based on them.

The research objectives:

- Research of existing systems and approaches to building integration processes
- implementation of a unified system for integration with the ability to build user processes

Research results: Implemented an integration application that works on the developed standard of data format and their transmission.

Conclusion: The tasks were completed. The result of the thesis is a working application that implements the developed standard of data integration. The application is usable and open to extension functionality.

Keywords: Integration, integration process, data, CRM, ERP, data management.

ЗМІСТ

РОЗДІЛ 1 ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1 Визначення предметної термінології

1.2 Актуальність та проблематика

РОЗДІЛ 2 ТЕХНОЛОГІЇ РОЗРОБКИ

2.1 Обґрунтування систем керування даними

2.2 Обґрунтування обраного технологічного стеку

2.3 Вимоги до застосунку

2.3.1 Вимоги до дизайну

2.3.2 Функціональні вимоги

2.3.3 Нефункціональні вимоги

РОЗДІЛ 3 ДОСЛІДЖЕННЯ ІНТЕГРАЦІЙНИХ ОСОБЛИВОСТЕЙ

3.1 Особливості створення інтеграцій

3.2 Види інтеграцій

3.3 Форми комунікації

3.4 Побудова інтеграційного стандарту

РОЗДІЛ 4 ПРОЕКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ

4.1 Проектування архітектури системи

4.2 Проектування бази даних

РОЗДІЛ 5 УПРАВЛІННЯ ДАНИМИ В СИСТЕМІ

5.1 Життєвий цикл даних

5.2 Діяльність у системі

5.3 Процесінг даних

5.4 Зіставлення відповідностей

5.4.1 Стандартний мапінг

5.4.2 Мапінг залежних об’єктів

5.4.3 Зворотній мапінг

5.4.4 Пошуковий мапінг

5.4.5 Мапінг функціями

РОЗДІЛ 5 ПОБУДОВА КОНЕКТОРІВ

5.1 Метадата Salesforce

5.2 Побудова Salesfroce конектору

5.3 Побудова QuickBooks Desktop конектору

ВСТУП

Протягом останніх років суспільство закріпило тренд на активну інформатизацію суспільства, серед основних потреб якої є побудова інформаційних систем, що здатні накопичувати упорядковані сукупності документів, масивів документів та інформаційних технологій із використанням обчислюваної техніки і зв’язку, що реалізують інформаційні процеси. Розвиток інформаційних систем призвів до експоненціального зростання кількості зберігаємої інформації. Така інформація, зазвичай має специфічний формат, шаблони, метадані, методи її зберігання, відтворення та видобутку.

Основою задачею роботи є побудова взаємно однозначного стандарту спілкування між системами управління даними та реалізація програмного забезпечення, що працює поверх цього протоколу. Для досягнення цієї мети у роботі розглянуті додаткові задачі:

1. Дослідження основних типів систем управління даними та методів їх комунікації.
2. Дослідження комунікаційних протоколів та меж їх використання.
3. Аналіз систем, що реалізують інтеграційні процеси між системами управління даними.

В результаті роботи було розроблено мета-мову опису даних, що охоплює весь спектр задач, які постають під час побудови інтеграційного процесу між системами управління даними, а також побудовано систему, що представляє собою уніфіковану інтеграційну платформу для побудову сучасних інтеграційних процесів. Система забезпечує високу швидкість та гнучкість у налаштування інтеграційних процесів та реалізує необхідні протоколи, щодо забезпечення безпеки даних під час їх передачі у інформаційному середовищі.

Існуючі бізнес системи можуть вирішувати свою задачу із різним ступенем якості, але зазвичай вони несумісні, відповідно до різних задач що постають перед ними. Значна різниця у ER-моделях, протоколах комунікації, архітектурних особливостей помножена на мільйонні об'єми даних породжують значну кількість проблем для людей, що мають стосунок до використання, підтримки, актуалізації та маніпуляції цими даними.

Використання такої системи є дуже затребуваним для бізнес сегменту, активна інформатизація якого призводить до значних проблем в управління даними та методах співставлення відповідності під час використання кластеру таких систем вузького призначення. Зокрема, в роботі окремо значну увагу приділено вирішенню цієї проблеми для "систем планування ресурсами" та "систем управління відносинами із клієнтами".

У роботі наведені аргументи, щодо переваг використання систем, аналогічних до розроблюваної у вигляді "платформа як застосунок", у яких системні інтегратори будують інтеграційні процеси за допомогою кастомного коду, візуальних інструментів та інших засобів.

РОЗДІЛ 1

ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1 Визначення предметної термінології

Оскільки термінологія даної області не є загальновживаної, необхідно формалізувати перелік основних понять, на які у подальшому буде спиратися робота.

Інтеграція – напрямок процесу розвитку, пов'язаний з об'єднанням в єдине ціле раніше різнорідних частих і елементів. Тлумачний словник Оксфордського університету розглядає інтеграцію як «акт чи процес поєднання двох чи більше частин таким чином, щоб вони функціонували разом» . Інтеграційні процеси відбуваються як у межах вже створеної системи, що призводить до підвищення рівня її цілісності та організованості, так і під час виникнення нової системи з раніше не пов'язаних елементів. Окремі частини інтегрованого цілого можуть мати різний ступінь автономії. Під час інтеграції збільшується обсяг та інтенсивність взаємозв'язків між складовими її системи, зокрема, надбудовуються нові рівні управління. Результатом інтеграції є певний ступінь інтегрованості – стан упорядкованого функціонування частин цілого.

Управління даними – процес, що передбачає ефективну, економічну, безпечну організацію процесів збору, збереження та використання даних. Ціль управління даними – оптимізація процесів із урахуванням політик та правил для того, щоб кінцеві користувачі мали змогу приймати рішення і діяти найбільш вигідним для компанії шляхом.

Система управління даними – це програмна система, що підтримує наповнення та маніпулювання даними, що мають цінність для користувачів при вирішення прикладних завдань. Основною особливістю та водночас причиною

проблем, яким присвячена ця робота, є наявність процедур введення і зберігання не тільки самих даних, але і описів їх структури.

Дослідити інтеграційні процеси – сформувати уявлення про цілі, методи та засоби побудови процесів, розглянути архітектурні, апаратні, програмні обмеження та особливості проміжних систем відповідно до цілей побудови інтеграційного застосунку.

Побудувати комунікаційний протокол – сформувати чітку, двосторонню систему визначень, назв, стандартів, що забезпечують передачу даних між однієї системи керування до іншої, не зважаючи на порядок, кількість та взаємну зв'язаність цих систем, по відношення до інших інформаційних акторів. Такий протокол може реалізовувати, або не реалізовувати протоколи безпеки, транспорту та інші, відповідно до поставлених цілей.

Зазначений результат роботи може бути застосований для будь-якої системи управління даними, що має аналог відкритого програмного інтерфейсу, бо розроблений формат отримується шляхом перетворення будь-яких даних, але найбільшого інформаційного різноманіття досягають «системи управління ресурсами» та «системи управління відносинами із клієнтами», тому доречно буде сфокусувати зусилля на демонстрації роботи саме для цих тип типів систем.

CRM або Управління відносинами з клієнтами — поняття, що охоплює концепції, котрі використовуються компаніями для управління взаємовідносинами зі споживачами, включаючи збір, зберігання й аналіз інформації про споживачів, постачальників, партнерів та інформації про взаємовідносини з ними. Сучасна CRM направлена на вивчення ринку і конкретних потреб клієнтів. На основі цих знань розробляються нові товари або послуги і таким чином компанія досягає поставлених цілей і покращує свій фінансовий показник.

Існує три CRM-підходи, кожен з яких може бути реалізованим окремо від інших:

1. Оперативний — автоматизація споживчих бізнес-процесів, що допомагає персоналу з роботи з клієнтами виконувати свої функції.
2. Співробітницький — програма взаємодіє зі споживачами без участі персоналу з роботи з клієнтами.
3. Аналітичний — аналіз інформації про споживачів із різноманітними цілями.

Класифікують можливості (модулі) CRM-систем за функціональністю та рівнем обробки інформації. За функціональністю можна згрупувати блоки процесів: маркетинг, обробка заявок та побажань, продажі, сервісне обслуговування. Як окремі складові зазвичай виділяють:

1. call-центри — центри обробки вхідних викликів. Спочатку це були телефонні дзвінки, а останнім часом сюди почали включати усі канали взаємодії;
2. функції (модулі) обробки інформації:
3. оперативна функція — реєстрація та оперативний доступ до первинної інформації за розділами бази даних: Події, Компанії, Проекти, Контакти, Документи тощо;
4. аналітична функція — звітність на основі первинних даних і найголовніше — глибший аналіз інформації у різних розрізах;
5. кооперативна функція — організація тісної взаємодії з кінцевими споживачами та клієнтами аж до впливу клієнта на внутрішні процеси компанії (опитування для зміни характеристик продукту чи порядку обслуговування, Web-сторінки для відслідковування клієнтами стану замовлення тощо).

ERP або Планування ресурсів підприємства — Система планування ресурсів підприємства) — корпоративна інформаційна система (КІС), призначена для автоматизації обліку й керування. Зазвичай ERP-системи будуються за модульним принципом і в тому або іншому ступені охоплюють всі ключові процеси діяльності компанії. Класичні системи ERP забезпечують керування задачами:

1. керування фінансами;
2. керування виробництвом;
3. керування формуванням та розподілом запасів;
4. керування реалізацією та маркетингом;
5. керування утриманням покупців
6. керування постачанням;
7. керування проектами;
8. керування сервісним обслуговуванням;
9. керування процедурами забезпечення якості продукції.

Впровадження

Класичні ERP-системи, на відміну від так званого програмного забезпечення «в коробці», належать до категорії «важких» замовних програмних продуктів — їхні вибір, придбання і впровадження зазвичай вимагають ретельного планування в рамках тривалого проєкту з участю партнерської компанії — постачальника або консультанта. Оскільки КІС будуються за модульним принципом, замовник часто (принаймні, на ранній стадії таких проєктів) купує не повний спектр модулів, а обмежений їхній комплект. У ході впровадження проєктна команда зазвичай протягом декількох місяців (до року) здійснює налаштування модулів, що поставляються.

Переваги

Використання ERP системи дозволяє використовувати одну інтегровану програму замість декількох розрізнених. Єдина система може керувати обробкою, логістикою, дистрибуцією, запасами, доставкою, виставлянням рахунків, бухгалтерським обліком, податковим обліком, програмою лояльності. Реалізована в ERP-системах система розмежування доступу до інформації, призначена (в комплексі з іншими заходами інформаційної безпеки підприємства) для протидії як зовнішнім загрозам (наприклад, промисловому шпигунству), так і внутрішнім (наприклад, розкраданням). Впроваджувані в зв'язці з CRM-системою і системою контролю якості, ERP-системи націлені на максимальне задоволення потреб компаній в засобах управління бізнесом.

Недоліки

Безліч проблем, пов'язаних з ERP, виникають через недостатнє інвестування в навчання персоналу, а також через недосконалість політики занесення і підтримки актуальності даних в ERP.

1. Невеликі компанії не можуть дозволити собі інвестувати достатньо грошей в ERP і адекватно навчити всіх співробітників.
2. Іноді ERP складно або неможливо адаптувати під документообіг компанії і її специфічні бізнес-процеси.
3. Система може страждати від проблеми «слабої ланки» — ефективність всієї системи може бути порушена одним департаментом або партнером.
4. Опір департаментів в наданні інформації зменшує ефективність системи.
5. Проблема сумісності з попередніми системами.
6. Помилки розробників у системі приводять до відчутних втрат коштів та частки на ринку.

1.2 Актуальність та проблематика

Оскільки поняття інтеграційного процесу не є загально визначеним у контексті використання у системах керування даних, то можна однозначно сказати що його процеси, методи та засоби не є детермінованими. Таким чином процес інтеграції розглядається лише як додаток до впровадження систем керування даними. Таким чином, визначення методів, стандартів та засобів імплементації зазначеного процесу залишається на другому плані і формується в залежності від побудованої інфраструктури. Такий підхід є дійовим, коли інтеграція не є пріоритетом, або коли розробники планують будувати сімейство продуктів, у яких можливо самостійно конфігурувати усі апаратні та програмні важелі. Однак, це рішення не завжди є виваженим.

Тенденція останніх років спрямована на збільшення дивергенції у програмних продуктах та системах. Таким чином, з'являється все більше продуктів які добре виконують лише одну функцію. Це можуть бути системи звіту, побудови графічних матеріалів, платіжні системи, поштові та будь-які інші. Такий підхід є дуже зручним і поставляє на ринок великий об'єм якісного програмного забезпечення, проте певні бізнес-задачі потребуються координованих зусиль таких систем – так з'являються сервіси-агрегатори. У даному контексті ми не розглядаємо агрегатори, що представляють собою звичайні веб-інтерфейси, у які можна вбудовувати плагіни, а системи, що мають на меті агрегувати дані з 2 та більше систем, трансформувати і доставити кінцевому адресату. Прикладами таких сервісів можуть бути системи для генерації лідів та продажу, які виконують певні бізнес-операції базуючись на зазначених бізнес-потребах.

Збільшення не детермінованості у світі таких систем створює складні завдання для команд, що займаються побудовою інтеграцій, що особливо відчутно коли такий процес потрібно побудувати між системами різних поколінь

– різниця у технологічній частині настільки значна, що потребує побудови складних програмних рішень для встановлення взаємної комунікації.

На сьогоднішній день вирішення цієї проблем існує – величезні системні агрегатори на кшталт Dell Boomie або Microsoft Power Automate. Такі системи забезпечують неперервну інтеграцію між багатьма системами, але ці програмні продукти розроблювалися роками і кожна приєднана система оброблена окремо, відповідно до архітектурних та програмного особливостей, що закладені їх розробниками. Збільшення кількості систем, оновлення їх публічних програмних інтерфейсів, еволюція програмних продуктів – фактори, які на довготривалому проміжку часу зроблять підтримку таких продуктів дуже складним та дорогим явищем, яке потребує значних людських та програмних ресурсів.

Дана роботи присвячена першому етапу вирішення цієї проблеми – побудові стандарту формату даних, та методів їх маніпуляції. Цей стандарт та програма, що працює поверх цього стандартну стане основою для міграційних можливостей систем керування даних. У сукупності із розглянутими у роботі особливостями таких систем, продукт дозволяє перейти до більш складних та фундаментальних речей, а саме – побудови окремого інтеграційного протоколу, який і реалізує та вирішує вищезазначені проблеми.

РОЗДІЛ 2

ТЕХНОЛОГІЇ РОЗРОБКИ

2.1 Обґрунтування систем керування даними

Для побудови змістовного та виваженого висновку щодо інтеграційних процесів, у роботі розглянута значних кількість систем керування даними, проте для демонстраційних цілей та перевірки розробленого рішення необхідно обрати декілька систем, що будуть індикаторами якості та змістовності розробленого застосунку. Системи, що містять найбільшу кількість зав'язків та які постійно використовуються у тандемі - це CRM та ERP – системи. Саме тому, на мою думку, доречно використати їх представників для візуального зображення результату.

Першою з таких представників є Salesforce – CRM система, що розповсюджується виключно за моделлю PAAS.

Найбільш цікавою для мети роботи є внутрішні механізми обробки та даних. Salesforce працює на основі архітектури Model-view-controller і використовує такі інструменти:

1. Apex - це мова програмування, яку використовує Salesforce. Є строго типізованою та нечутливою до регістру. Apex можна використовувати для написання користувацького коду, що виконується під час певних визначених операцій (приклад – створення об'єкту, оновлення об'єкту чи певна комбінація цих подій). Сама ця технологія дозволяю будувати користувацькі інтеграційні процеси довільно, без прив'язки до стандартного функціоналу платформи. Наприклад, таким чином можна збирати дані про створені об'єкти та надсилати їх до третіх осіб, проте

такий підхід може бути вразливий до атак, якщо не продумати надійну систему авторизації запитів.

Представником із систем управління ресурсами є Quickbooks – більш стара система, що реалізує поєднує архаїчні протоколи комунікації разом із новітніми практиками. QuickBooks - це програмний пакет бухгалтерського обліку, розроблений та проданий Intuit. Продукти QuickBooks орієнтовані в основному на малий та середній бізнес і пропонують локальні програми бухгалтерського обліку, а також хмарні версії, які приймають бізнес-платежі, управляють та оплачують рахунки та функції оплати праці.

На поточний момент існує два різновиди системи Quickbooks – Desktop та Online. Перша цікава тим, що написана більш ніж 30 років тому, але все одно використовується більш ніж 600 тис. клієнтів і по сьогоднішній день, серед основних причин цього – система не має доступу до інтернету, що створює значні проблеми для побудови інтеграційних процесів. Онлайн версія є більш сучасної і підтримує усі новітні підходи до побудови PaaS – систем, як-то oauth2-авторизацію, публічний програмний інтерфейс та веб-перехоплювачі. Інтеграція з такою системою, у контексті теми роботи, є дуже важливою, тому що успішна інтеграція із таким архаїчним продуктом доводить життєздатність рішення на полі систем керування даними.

2.2 Обґрунтування обраного технологічного стеку

Стек розробки було обрано відповідно до основної мови розробки - Javascript. Вибір цієї мови є оптимальним для задачі обробки динамічних даних та їх перетворення під час роботи системи. Також Javascript нативно підтримує функції, що виконуються динамічно, що є зручним під час створення співвідношень, що можуть змінюватися. Значна підтримка спільноти та великий обсяг програмного забезпечення, що знаходиться у вільному доступі також є визначним фактором для системи. Під час побудови системи, було важливо не прив'язуватися до будь-яких комерційних рішень, тому система керування базами даних також є проектом з відкритим кодом – PostgreSQL.

2.2.1 Система керування базами даних PostgreSQL

PostgreSQL це об'єктно-реляційна система керування базами даних, написана на мові програмування C. Ця система набула широкого розповсюдження через свої значні програмні можливості (використовуючи PL/pgSQL), а також потужні механізми реплікації, шардінгу та підтримку однозначної модифікації БД декількома користувачами да допомогою механізму MVCC.

2.2.2 Фреймворк Odi

Odi - це гнучкий фреймворк для розробки веб-застосунків, побудований на основі Node.js, що надає просте інтуїтивне API і дозволяє сконцентруватися на бізнес логіці замість довгої роботи над спрощенням архітектури застосування, надає величезний набір методів, має в своєму розпорядженні безліч допоміжних HTTP-методів та проміжних обробників але на відміну від аналогів таких як Express або Koa підтримує Dependency Injection, що спрощує написання та тестування коду через виконання приципу Dependency Inversion. Також Odi є найшвидшим з інших фреймворків на Node.Js.

2.2.3. React

React - Javascript фреймворк для розробки фронтенду що, дозволяє розробляти динамічні веб застосування. Має виличезний набір допоміжних безкоштовних бібліотек. Розробка ведеться в декларативномії парадигмі програмування , що дозволяє дуже легко підтримувати додаток і робить код більш передбачуваним.

2.2.4 Платформа Node.js

Node.js - це JavaScrip t– оточення побудоване на JavaScript–движці Chrome V8. Node.js використовує дієву, неблокуючу I/O модель, що робить його легким та ефективним. Пакетна екосистема Node.js, npm, в свою чергу є найбільшою у світі екосистемою бібліотек з відкритим кодом, яка дозволяє швидко та просто встановлювати всі наявні бібліотеки, а також створювати свої та викладати їх для загального користування.

2.2.5 Amazon Web Services

Amazon Web Services це хмарна платформа що надається в платну оренду користувачам. Базується на серверних фермах по всьому світу і підтримувана компанією Amazon. Значна кількість наявних сервісів додає варіативності та варіантів використання під час вибори стратегії розміщення додатку. Так, стандартне розміщення за допомогою EC2 може бути, відносно просто, перенесено на ELC.

2.2.6 Docker

Docker — інструментарій для управління ізольованими Linux-контейнерами. Docker доповнює інструментарій LXC більш високорівневим API, що дозволяє керувати контейнерами на рівні ізоляції окремих процесів. Зокрема, Docker дозволяє не переймаючись вмістом контейнера запускати довільні процеси в режимі ізоляції і потім переносити і

клонувати сформовані для даних процесів контейнери на інші сервери, беручи на себе всю роботу зі створення, обслуговування і підтримки контейнерів.

2.2.7 Bitbucket

Bitbucket — веб-сервіс для хостингу проектів на базі систем керування версіями: Mercurial та Git. Bitbucket надає як безкоштовні так і платні послуги. Bitbucket є аналогом GitHub, проте на відміну від GitHub, у якого при безкоштовному профілі файли зберігаються лише у відкритому доступі, Bitbucket дозволяє безкоштовно створювати приватні репозиторії з можливістю спільної роботи з файлами до 5-ти користувачів. Bitbucket інтегрований з іншими програмними продуктами Atlassian, такими як, JIRA, Confluence, Bamboo та HipChat.

2.3 Основні вимоги до застосунку

2.3.1 Вимоги до дизайну застосунку

– Сайт повинен бути виконаний у єдиному стилі, з використанням м’яких тонів, що забезпечують зручне використання
– Інтерфейс має бути інтуїтивно зрозумілий для користувача
– Усі функціональні розділу сайту мають бути доступі з навігаційної панелі сайту
– Дизайн сайту повинен відповідати стандартом сучасних технік створення сайтів
– Інтерфейс має бути графічно та функціонально-орієнтованим. Тобто не містити зайвої інформації.

2.3.2 Вимоги до доступу в систему

– Відповідно до специфіки системи сайт є закритою одиницею. Тобто усі сторінки не мають бути доступними без аутентефікації
– При спрові отримати будь-яку сторінку системи користувач повинен авторизуватися. Для цього користувач буде перенаправлений на сторінку авторизації. У разі помилок у наданій інформації – подальші дії у системі будуть заблоковані.
– Навігаційна система повинна бути адиптвною відповідно до ролей у системі. Тобто для звичайного користувача не будуть доступні розділи адміністрування системи.
– Спроба перейти до сторінки, яку користувач не повинен бачити через нестачу рольових привелеїв(наприклад переходячи за абсолютним посиланням на сторінку у URL) має бути зупинена переадресацією на головну сторінку додатка, або в разі, якщо користувач не авторизований – на сторінку авторизації.

2.3.3 Функціональні вимоги

- Загальносистемні
    1. Авторизація
    2. Вихід із системи
- Форма авторизації
    1. Вказання ідентефікатора у системі
    2. Вказання паролю
- Наявність можливості створення інтеграції
    1. Створення нової інтеграції та вказання її імені
    2. Додавання процесів до створеної інтегарції
    3. Підтвердження створенної інтеграці
- Наявність можливості створення процесу
    1. Задання імені процесу
    2. Задання API-імені об’єкту, що буде вихідним для поточного процесу
    3. Задання запиту, що буде використаний для вибірки даних з вихідної системи
    4. Задання розміру групи, за числом яких буде проходити обробка даних
    5. Задання параметру, що ідентефікує, чи буде процес запущений одразу при запуску інтеграційного процесу
    6. Задання параметру, що ідентефікує, чи буде в процесі використаний процес зворотнього мапінгу
    7. Можливість додавання нового інтеграційного правила або декількох правил
    8. Можливість задавання імені об’єкту, що буде цільовим для інтеграційного правила
    9. Можливість задати операцію яка буде використана для визначення специфіки обробки даних

10. Можливість задати співвідношення у мапінгу у вигляді пар з вихідного поля та цільового поля
11. Можливість додавати до співвідношення функції, що будуть впливати на принцип обробки поточного співвідношення
12. Можливість додати співвідношення для залежних об’єктів
13. Можливість задати параметр, що ідентефікує чи потрібно зберігати в пам’яті системи інформацію про результати поточного інтеграційного правила
14. Можливість видалити будь-який елемент співвідношення
15. Можливість підтвердити коректність створеного співвідношення
16. Можливість змінити формат заповнення на звороній мапінг. Додати ім’я об’єкту для зворотнього мапінгу та аналогчні до прямого – зворотній мапінг для вихідної системи.
17. Порахувати кількість потенційних елементів, що буде оброблена до моменту збережння процесу.

РОЗДІЛ 3
ДОСЛІДЖЕННЯ ІНТЕГРАЦІЙНИХ ОСОБЛИВОСТЕЙ

3.1 Особливості створення інтеграцій

Більшість компаній мають швидко зростаюче число наявних веб-системи, що дозволяють кожному - працівникам, клієнтам та діловим партнерам доступ до важливої функціональності та доступ до важливої інформації, необхідної для їх роботи. Ці всі нові програми розроблені з інтерфейсом браузера, в першу чергу для його універсального доступу характеристики.

Однак із збільшенням кількості веб-систем, що зростають в геометричній прогресії, як усередині, так і поза фізичних мережевих шлюзів до організації, часто дуже важко дістатись до цих систем працювати разом у будь-який інтегрований спосіб. Майже у кожному випадку вони будувались самостійно.

Різні групи, що використовують різні технології. Однак універсальним є те, що всі веб-програми використовувати HTTP як транспортний протокол, а HTML як синтаксис програми. Насправді це універсальне використання обидва для майже всіх веб-додатків представляють особливий тип Інтерфейсу програмування програм(API), який можна використати для інтеграції.

Проблеми веб-інтеграції:

Сучасні підприємства стикаються з дедалі важливішими та складнішими проблемами інтеграції. Незважаючи на всі зусилля, спрямовані на стандартизацію та координацію роботи в межах компанії, більшість IT-організацій стикаються з численними внутрішньо розробленими та пакетними програмами, які не були розроблені для взаємодії між собою. З веб-додатками вони тепер стикаються з ще більшим викликом - необхідністю доступу та інтеграції із програмами за межами підприємства для підтримки клієнтів, партнерів та постачальників. Традиційно для вирішення цих проблем

застосовуються стандартні підходи до інтеграції додатків. Зовсім недавно експерти та засоби масової інформації пропонували веб-служби як панацею для інтеграції програм. Проте більшість організацій стикаються з низкою серйозних проблем за допомогою цього методу.

1. Час на реалізацію

Незважаючи на обмежені бюджети та обмежені ресурси, сьогодні більшість користувачів вимагають, щоб їхні технологічні групи надавали рішення швидше, ніж будь-коли раніше. У світі інтеграції вирішальний крок - визначити, який API використовувати для інтеграції. Такі галузеві підходи, як веб-сервіси, сприяють єдиному програмному інтерфейсу для всіх додатків, але всі нормальні проблеми розвитку залишаються. Перед їх розгортанням необхідно створити та ретельно протестувати веб-служби або навіть ручні рішення. Зазвичай це передбачає зміну програми, щоб викрити її логіку, що вимагає великих ресурсів та вимагає багато часу. Прагматичніше, такий підхід можливий лише тоді, коли ваша компанія володіє цілою програмою.

2. Непрактичність

Для веб-додатків за межами підприємства зміни, які вимагають традиційні методи інтеграції, часто неможливі. Навіть коли партнер або клієнт готові відкрити свої системи, можливість інтеграції може бути недоцільною. Як результат, більшість важливих для бізнесу проектів, що потребують інтеграції із зовнішніми програмами, в кращому випадку надзвичайно важкі, а в гіршому неможливі. Для більшості організацій це може призвести безпосередньо до втрачених можливостей для бізнесу та збільшення операційних витрат.

3. Ціна

Традиційна інтеграція додатків покладається на повний контроль над програмою, тому її можна змінювати внутрішньо. Це часто призводить до значних витрат, оскільки більшість проектів інтеграції програм представляють дуже великі зусилля з розробки з використанням висококваліфікованого персоналу. Цим великим проектам властивий високий ризик затримки та можливий вплив на загальну інфраструктуру організації.

3.2 Види інтеграцій

За принципами та формами можна визначити 4 основні види інтеграцій:

1. Ручне перенесення даних - є однією з найпростіших форм інтеграції і вимагає взаємодії людини. Передача вручну, як правило, передбачає експорт даних у одне місце та імпорт їх кудись ще. Наприклад, клієнт може експортувати всі свої замовлення зі свого веб-сайту, а потім імпортувати свої результати у QuickBooks. У довгостроковій перспективі передача даних вручну може виявитися трудомісткою і схильною до людських помилок.
2. Серверна передача даних - схожа на ручну, але вона має елементарний рівень автоматизації. У більшості випадків система А потрапляє у файлову систему системи B і розміщує файл у системі B. Тоді система B, яка діє на цей файл, наприклад імпортує у вашу базу даних відстеження замовлень. Ця система, як правило, негнучка, але може бути швидким рішенням для інтеграції, яке матиме жорсткі правила, яких легко дотримуватися. У багатьох випадках робочий процес відбувається у запланований час і щоразу однаковий, без різниці, це може бути простий спосіб автоматичної обробки повторюваних завдань. Наприклад, сервер веб-сайту може використовувати FTP для передачі щоденного списку всіх ваших замовлень на сервер бази даних продуктів.
3. Прямий доступ до сервера - це місце, де одна система має доступ до ресурсів іншої системи. Наприклад, веб-сайт буде входити безпосередньо у вашу базу даних про товари та отримувати всю інформацію за потреби. Прямий доступ до сервера створює багато проблем з безпекою для більшості IT-підрозділів та сторонніх постачальників, і небагато, якщо такі взагалі дозволяють. Однак, якщо його можна безпечно і надійно виконати, його можна використовувати для дуже надійної розробки додатків.

4. Веб – інтерфейси - це сучасний спосіб взаємодії систем. Веб-інтерфейси виступають безпечним набором інструментів, спрямованих назовні. Замість програмування жорстких структур, які повторюються, тепер ви можете взаємодіяти в режимі реального часу, коли ваш веб-сайт може запитувати власну інформацію з іншої системи, коли користувач взаємодіє з сайтом. За допомогою веб-інтерфейсів системи не просто обмінюються інформацією, вони можуть взаємодіяти між собою. На відміну від будь-якої іншої системи, ваш веб-інтерфейс також портативний. Веб-інтерфейс не тільки забезпечує спосіб взаємодії вашого веб-сайту з ним, але також дозволяє взаємодіяти з інтерфейсом все, що підключено до Інтернету. Використання стандартів є найкращим варіантом при використанні інтерфейсу. Спеціальні інтерфейси можуть бути проблематичними, але багато разів існують обмежувальні фактори, що перешкоджають використанню стандартизованого веб-інтерфейсу. Це можуть бути апаратні чи програмні обмеження, обмеження навичок програміста чи час, а іноді веб-інтерфейс, який у вас є, просто не робить всього необхідного. Спеціальну інтеграцію потрібно робити часто, і це не повинно стримувати вас на шляху інтеграції до веб-інтерфейсу. Як приклад, ви можете використовувати програмне забезпечення, яке забезпечує веб-інтерфейс, але воно не використовує визнаний стандарт W3C. Зазвичай це змушує розробників писати спеціальне програмне забезпечення для взаємодії з цією системою. Багато разів це виключає багато наявних у їх розпорядженні інструментів, подовжує час розробки, час тестування.

3.3 Форми комунікації

Після класифікації інтеграцій за їх типом та формою, необхідно визначити прикладні інструменти, які дозволяють реалізовувати ті чи інші інтеграційні підходи.

1. Відкритий програмний інтерфейс

Підхід, за якого розробники системи закладають інфраструктуру для отримання та створення даних системи. При побудові додатків API спрощує програмування, абстрагуючись від базової реалізації та виставляючи лише об'єкти або дії, необхідні розробнику. Хоча графічний інтерфейс для поштового клієнта може надавати користувачеві кнопку, яка виконує всі дії для отримання та виділення нових електронних листів, API для введення / виведення файлів може дати розробнику функцію, яка копіює файл з одного місця в інше без вимагаючи, щоб розробник розумів операції з файловою системою, що відбуваються за кадром. За такої схеми, можливе передбачення двох потенціальних систем аутентифікації користувачів:

1. Базова аутентифікація – у контексті HTTP, метод для користувача для надання імені користувача та пароля під час надсилання запиту. Запит має містити поле заголовку у формі Authorization: Basic <credentials>, де облікові дані - це кодування Base64 ідентифікатора та пароля, об'єднані однією двокрапкою. Така система аутентифікації є найпростішою, але водночас і найбільш схильна до зламу, тому використовувати її для промислових програмних інтерфейсів треба обережно, та бажано, із надійних контекстів, наприклад – Tableau Trusted Authentication. Така авторизація стандартизована у RFC 7617.
2. Авторизація на основі Oauth2 - це відкритий стандарт авторизації, який дозволяє користувачам відкривати доступ до своїх приватних даних

(фотографії, відео, списки контактів), що зберігаються на одному сайті, іншому сайту, без необхідності вводу імені користувача та паролю. OAuth дозволяє користувачам роздавати сайтам маркери доступу, до даних що розміщуються на сайтах-сервісах. Кожен маркер доступу надає доступ конкретному сайту (наприклад, сайту редагування відео) до конкретних ресурсів (наприклад, тільки відео від конкретного альбому) та на визначений термін (наприклад, на наступні 2 години). Це дозволяє користувачам надавати доступ третім сайтам до їх інформації, що зберігається на інших сайтах — постачальниках послуг, не передаючи повною мірою самих даних та без застосування імені/паролю. Такий варіант авторизації є значно надійнішим і широко використовується для будь-яких промислових інтерфейсів, наприклад – Stripe Oauth2. Така авторизація стандартизована у RFC 6749.

Загалом, на сьогоднішній день майже не існує системи яка не надає відкритий програмний інтерфейс разом із своїм продуктом. Тому цю форму комунікації із системою можна вважати загальноприйнятою і впроваджувати як стандарт. Однак деякі системи оформлюють свій програмний інтерфейс як окремий продукт, який не можна отримати безкоштовно, чи не будучи членом компанії – партнером, наприклад – Glassdoor API.

2. Веб-перехоплювачі (Web-hooks) – це визначені користувачем зворотні виклики. Зазвичай їх викликає якась подія, як-то створення певного об’єкту у системі, його оновлення чи видалення. Коли така подія відбувається вихідна система робить HTTP-запит до певної URL-адреси, налаштованої для веб-перехоплювача. Цей підхід є типовим представником моделі інтеграції, що заснована на подіях. Такий метод комунікації також є загально вживаним, оскільки може забезпечувати доставку інформації за лічені секунди. Авторизація веб-перехоплювачів може бути реалізована у різних варіантах:

1. Кінцева точка, яка отримує, може вибрати збереження списку IP-адрес для відомих джерел, з яких будуть прийняті запити.
2. Базову автентифікацію HTTP можна використовувати для автентифікації клієнта.
3. Веб-хук може містити інформацію про тип події, а також секрет або підпис для перевірки веб-хука.
4. Підпис HMAC може бути включений як заголовок HTTP. GitHub та Stripe використовують цю техніку.
5. Facebook підписує їхні запити за допомогою SHA-1.
6. Взаємну автентифікацію TLS можна використовувати, коли встановлено з'єднання. Кінцева точка (сервер) може потім перевірити сертифікат клієнта.

3. Власні розробки – ще один варіант комунікації, за якого усі методи, засоби та варіанти використання лягають на відповідальність розробника. Такий метод є можливим, лише за умови, коли система підтримує внутрішні програмні методи, або навіть мову програмування. Такий підхід дозволяє отримати максимальну гнучкість і вирішує проблеми там, де потрібні складні поєднання даних, або специфічні умови до авторизації та/або передачі даних. Основним недоліком такого підходу є велика вартість розробки і мала стійкість до помилок та зміни вимог. Вдалими прикладами систем, що підтримують такий підхід є – Salesforce (за допомогою Apex Controllers) та NetSuite (за допомогою Restlets). Такі розробки не стандартизовані і відповідають внутрішнім потребам авторів системи.

3.4 Побудова інтеграційного стандарту

Відповідно до проведеного дослідження комунікаційних протоколів можна зробити певні висновки щодо уніфікації інтеграційного стандарту.

1. Використання стандартного API – найкраща практика. Оскільки розробка публічного або приватного відкритого програмного інтерфейсу є загальновживаною практикою, то необхідно сфокусуватися на такому підході до інтеграції із системами управління даними. Цей метод є значно гнучкішим, ніж наприклад використання веб-перехоплювачів, тому що можна самостійно керувати життєвим циклом і гарантувати доставку даних. Використання веб-перехоплювачів також може бути ускладнено проблемами із інтернет-з'їданням (хоча певні системи вирішують це, наприклад - Stripe), або знаходження за фаерволом, що унеможливлює нормальну роботу системи. Серед додаткових переваг, можна зазначити ручне управління об'ємами даних, тобто ми можемо самостійно обирати розмір і кількість об'єктів, що підлягають інтеграції
2. Динамічне керування кількістю даних – необхідність. Системи значно різняться за своїми підходами, так само можуть змінюватися кількість полів і відсоток їх наповнення, тому коректна реалізації та можливість динамічно визначати розмір інтегрованих об'єктів – системна необхідність.
3. Композиція даних - важливий аспект. Інтеграційні процеси будуються поверх складних та заплутаних бізнес процесів, що зазвичай побудовані не оптимально, а іноді являють собою повну протилежність оптимальності. Але коректні процеси розробляти дорого і довго, тому реальна їх зміна маловірогідна, тому ми повинні мати можливість динамічно компонувати дані, відповідно до плаваючих бізнес-потреб.
4. Час інтеграції – критично важливий. Тут необхідно зазначити обидва розуміння часу у інтеграційному контексті. Час виконання – необхідно

виконувати інтеграційні процеси швидко відносно об'єму даних, тому що для певних систем керування даними, актуальність даних може лічитися хвилинами, що може призвести до некоректної роботи залежних людей. Щодо розуміння часу, як періоду у який виконується інтеграційний процес – необхідно забезпечити гнучкий механізм налаштування цього параметру, бо певні організації лімітують зміни у свої системах у певних часових межах.

5. Кількість вихідних і кінцевих потоків може відрізнятися. Враховуючи різні структури систем керування даними, вхідні об'єкти можуть відповідати різним об'єктам на виході. Для цього кожному процесу інтеграції треба одне чи більше правило (але не менше одного) для формування вихідного співставлення.
6. Забезпечення цілісності даних. Під час передачі даних можливі потенціальні записи, що не можуть бути інтегровані відповідно до бізнес правил. Для того, щоб забезпечити цілісність даних та візуальне розуміння того, які дані були інтегровані правильно. Тому необхідно забезпечити механізм зворотного співставлення для запису ідентифікатора зворотної системи або ідентифікатора коду помилки.
7. Опис даних має бути наочним. Необхідно забезпечити простий спосіб задання процесів та правил інтеграції, тому для системи не будуть використані більш оптимальні методи (бінарні методи) для запису. Замість цього основним протоколом опису має бути JSON.
8. Зберігання динамічних даних. Ця вимоги виникає як наслідок композиції даних, тому що композиція може бути реалізована не лише на рівні вихідних даних, а і на вхідних (декілька об'єктів однієї системи мають створити один, але у нашому процесі може бути зазначений лише один сирцевий об'єкт). Крім того, такий підхід є необхідним для забезпечення

кешування даних під час виконання «дорогих» операцій (прикладом таких операції є операція пошуку)

9. Налаштування даних. Різність форматів та форм надання даних вимагає від підтримувати не лише відповідність типу «один до одного», а й варіації, що допускають «один до багатьох»(у контексті полів) та «один до одного із модифікацією сирцевих полів». Оскільки така функціональність має велике значення і неможливо визначити границю необхідного функціоналу – реалізувати треба увесь спектр операції над строковими та чисельними даними.
10. Порядок. Оскільки кінцевою метою створення стандарту є зменшення не детермінованості – необхідно забезпечити чіткий порядок виконання операцій. Це також необхідно для забезпечення бізнес потреб у випадку багаторівневого інтегрування типу «батько-нащадок».

РОЗДІЛ 4
РЕАЛІЗАЦІЯ ІНТЕГРАЦІЙНОГО ЗАСТОСУНКУ

4.1 Проектування архітектури системи

Проектування архітектури програмного забезпечення – це процес створення, що виконується після етапу аналізу і збору вимог. Задача проектування – перетворення зібраних вимог до розроблюваної системи у вимоги до програмного забезпечення та побудова на основі аналізу релевантної архітектури. Основним методом побудови архітектури є визначення цілей та меж системи, її вхідних та вихідних даних, декомпозиція системи на складові (підсистеми), компоненти та модулі, розробка загальної структури додатку. Проєктування архітектури використовує різні методи для вирішенні задачі:

1. Стандартизований метод
2. Об’єктно-орієнтований метод
3. Компонентний метод

Кожен з цих методів визначає свій шлях до побудови архітектури, а саме визначення моделей за допомогою відповідних структурних елементів (діаграми, графи, схеми).

У контексті розроблюваного застосунку, важливо сфокусуватися на елементах верхнього рівня та методі їх використання, тому доречно буде використати об’єктний підхід.

Проєктування системи може здійснюватися на основі об'єкто-орієнтованого моделювання методом UML, який дозволяє враховувати аспекти, властиві діючим особам системи, створювати сценарії виконання системи тощо. Об’єктний стиль проєктування — це декомпозиція майбутньої системи на окремі підсистеми (пакети), визначення функціональних і нефункціональних вимог і

об'єктної моделі предметної області. Носіями інтересів, можливостей і дій в системі є діючі особи — актори. Пакет може складатися з об'єктної моделі, варіантів використання, що визначають сценарії поведінки системи, склад об'єктів і методів їхньої взаємодії. Поведінка об'єктів відображується діаграмами, що задають послідовність взаємодії об'єктів (діаграми послідовностей, взаємодії), правилами переходу від стану до стану (діаграми станів), а також діаграмами кооперації, в яких діючі особи зображуються графічно. Об'єкти і відповідні їм діаграми варіантів використання задають загальну архітектурну схему системи, у рамках якої здійснюється реалізація структури і специфіки поведінки компонентів. Загальна концепція об'єктного проєктування — це побудова всіх сценаріїв, екранних діаграм для керування ними і їх випробування в різних варіантах використання. На вибір варіантів використання впливають нефункціональні вимоги (наприклад, забезпечення конфіденційності, швидкодії й ін.). На основі моделі опису вимог і понять проводиться уточнення складу і змісту функцій системи, методів їхньої реалізації у вигляді сценаріїв і діаграм потоків даних, у яких відображається взаємодія об'єктів як обмін повідомленнями між елементами системи для передачі даних і одержання відповіді після виконання операцій. Моделі вимог визначають призначення і місце вимог у таких системах. Цьому сприяють розроблені національні, корпоративні і відомчі стандарти. Вони фіксують правила формування нефункціональних вимог, у яких відображаються відомості про взаємодію і захист даних у системі. При цьому поведінка об'єктів представляється діаграмами UML, вони можуть уточнюватися при перегляді моделей вимог і складу об'єктів системи. Перегляд починається з вимог і пошуку місць локалізації для внесення необхідних змін у модель. Зміни можуть стосуватися функціональних і нефункціональних вимог у зв'язку з уточненням замовником обмежень на структуру системи, використовувані ресурси й умови середовища її функціонування.

Система спроектована таким чином, щоб кожний сервер виконував свою, незалежну від інших окрему функцію. Для того щоб сервери могли спілкуватися і довіряти дані один одному через увесь додаток пронизана JWT авторизація.

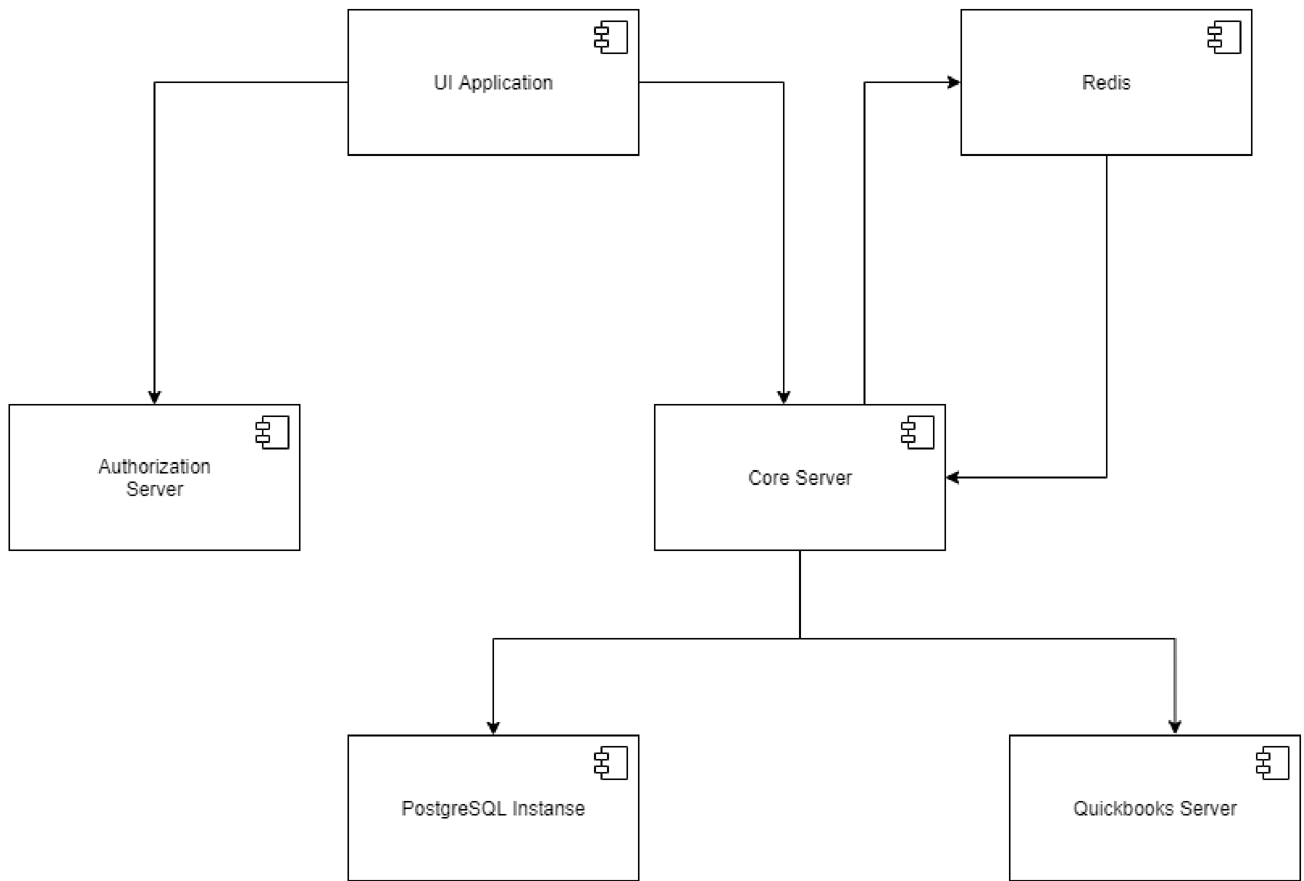


Рисунок 4.1.1 – Загальна архітектурна модель системи

1. Core – сервер. Відповідає за процесінг даних, створення, запуск та управління конфігураціями. Сервер використовує експериментальний фреймворк Odi, що є високорівневою обгорткою над Fastify.js.
2. UI Application – React додаток, що є вхідною точкою для усіх ролей системи. Звідси можливо створювати нові конфігурації, редагувати за запускати процеси, спостерігати прогрес виконання.
3. Authorization Server – Сервер, що забезпечує JWT-авторизацію, що пронизує усю систему.

4. QuickBooks Server – Сервер, що забезпечує комунікацію між додатком та QuickBooks Web Connector – вбудовану технологію в QuickBooks для забезпечення інтеграційних процесів.
5. Redis - розподілене сховище пар ключ-значення, які зберігаються в оперативній пам'яті, що дозволяє розподіляти інтеграційні сутності і виконувати «навігацію» під час виконання програми.

4.2 Проектування бази даних

Розглянемо спочатку концептуальне проектування - побудову семантичної моделі предметної області, тобто інформаційної моделі найбільш високого рівня абстракції.

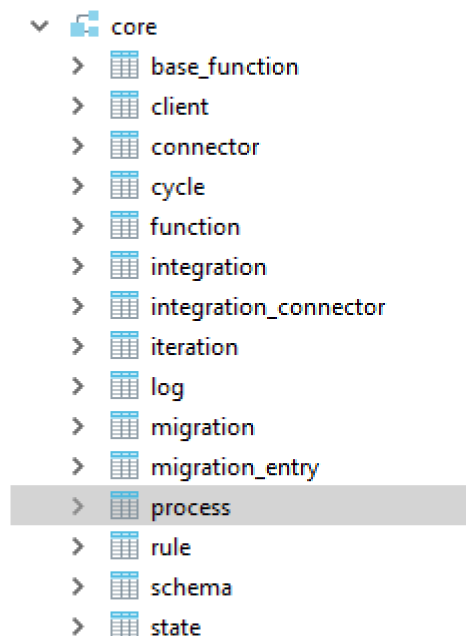


Рисунок 4.2.1 – Перелік сутностей у базі даних

Client – користувач застосунку що має власні інтеграції, міграції, конектори та має 2 рівні доступу: адмін та користувач. Адмін має доступ до всіх клієнтів застосування.

Connector – конектор до CRM системи, ERP системи, або будь-якої іншої PaaS-системи, містить дані для входу, лінк для входу та тип підключення.

Integration – об’єкт інтеграції що містить в собі process, має зв’язок до користувача та має два конектори – джерело та ціль.

Рисунок 4.2.2 – Об’єкт інтеграції

Process – процес інтеграції що містить настройки інтеграції та зв’язок до інтеграції, інтеграція може містити багато процесів. Інформація про процес це sourceObject – об’єкт з якого читати дані, cronSettings -- настройки розкладу роботи, enabled: чи треба стартувати процес, onInitRun -- чи стартувати процес при створенні інтеграції. BatchSize – кількість об’єктів які проходять за один раз. ProcessName – назва процесу, Query – кастомна функція для обробки даних в процесі, яка приймає стан процесу, та стан ранера процесу. Та масив об’єктів Rules.

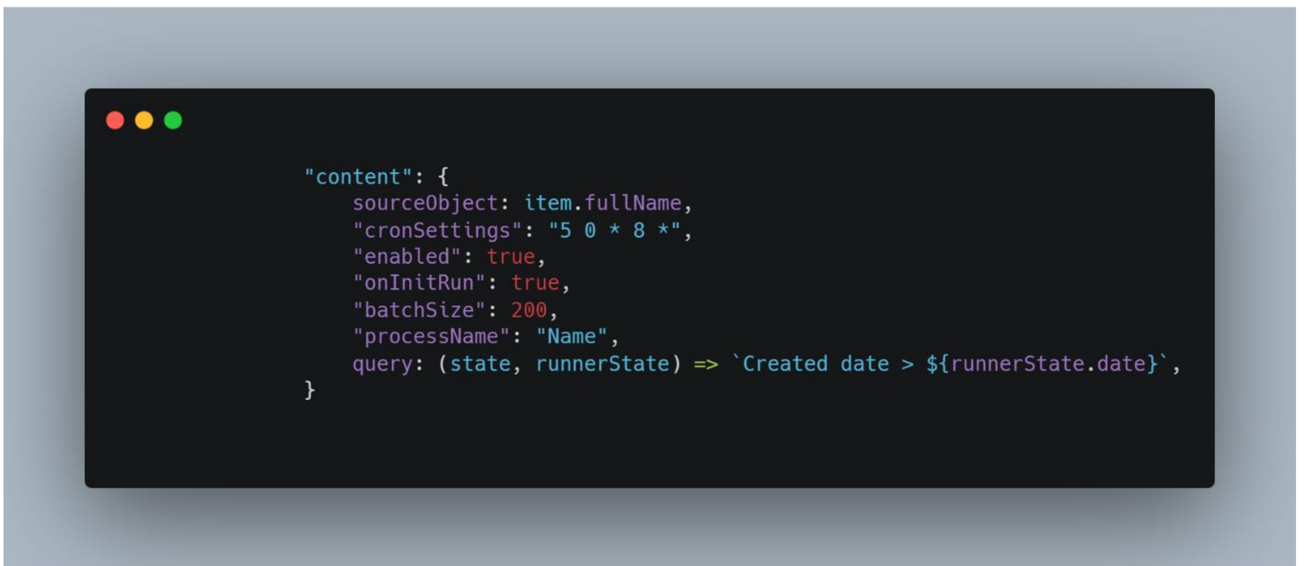


Рисунок 4.2.3 – Об’єкт процесу

Rules - об’єкт правил для процесу що існує щоб з одного об’єкту можна було записувати дані в декілька об’єктів або для оновлення об’єктів на льоту у випадку якщо наприклад потрібно поставити поле батьківський об’єкт, яке містить посилання на об’єкт такого ж типу. Поля правила(rule): Name – назва процесу, enabled - чи включене правило, targetObject – об’єкт в який записуємо,

Mapping – об’єкт правил для полів, onBatchWrote – hook для обробки виконаного батчу, operation – визначає яка це операція (можливі 4 варіанти: insert, update, delete, upsert), writeback – чи включений зворотній запис для зв’язку об’єктів у таргетовій системі з сусрс системою, writebackMapping – правила запису для зворотнього запису.

```
                rules:
                    [
                        {
                            "name": "SF Account to SF Account",
                            "enabled": true,
                            "targetObject": ‘Account’,
                            mapping: 'Legacy_ID__c': { alias: 'Id' },'CreatedDate': { alias:
'CreatedDate' },'CreatedById': { func: '(entry) => entry', alias: `SFLookup('SELECT Id, STL_User_ID__c
FROM User WHERE STL_User_ID__c =CreatedById')` }},
                            "onBatchWrote": "onSFBatchMigrationEntry",
                            "operation": "insert",
                            "writeback": false,
                            "writebackMapping": {
                        ‘Account_id_new’ : {
                            alias : ‘Id’
                        }
                            }
                        }
                    ]
```

Рисунок 4.2.4 – Об’єкт правила

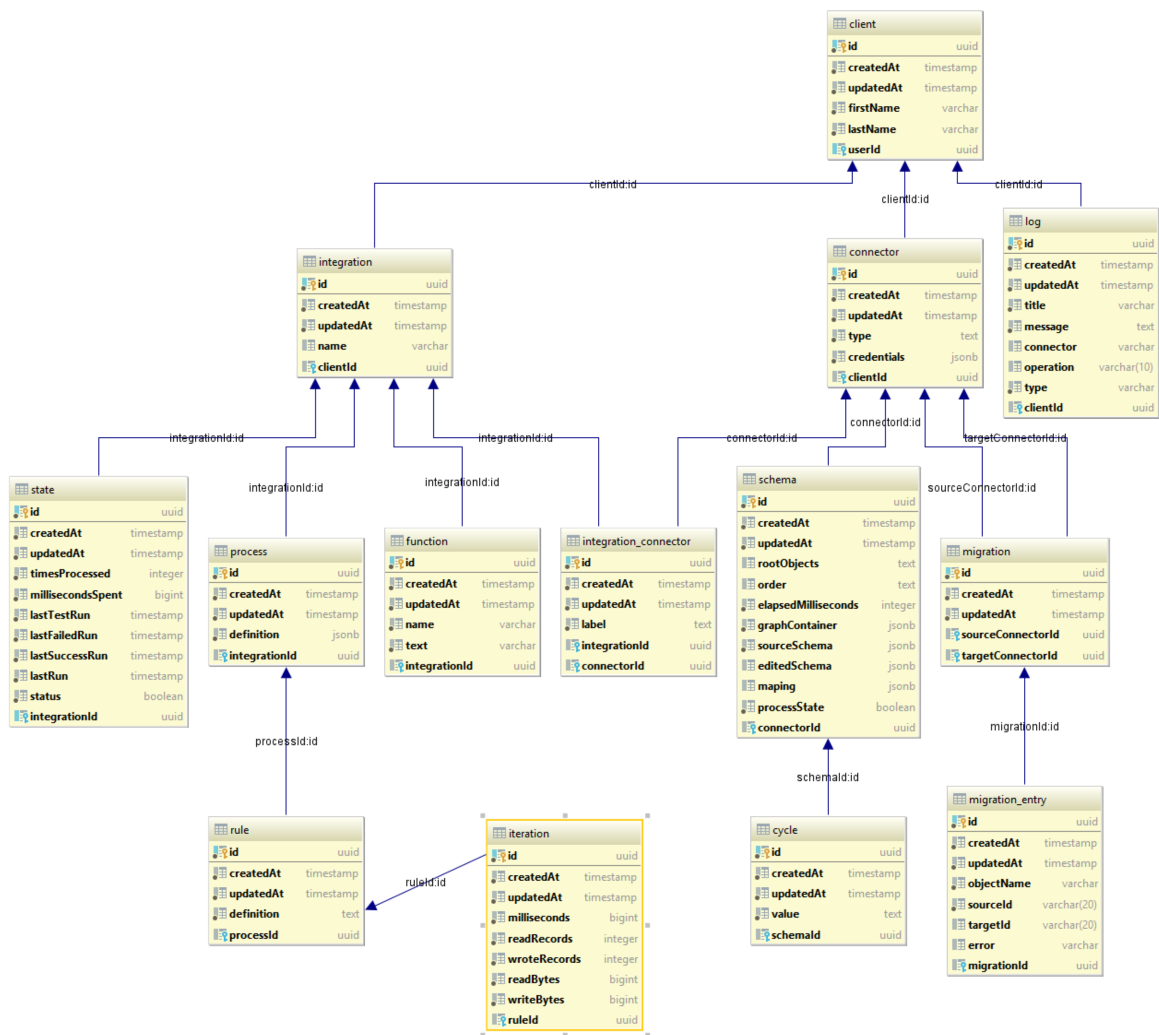


Рисунок 4.2.5 – Схема бази даних

РОЗДІЛ 5

УПРАВЛІННЯ ДАННИМИ В СИСТЕМІ

5.1 Життєвий цикл даних

Щоб описати життєвий цикл необхідно ввести наступні поняття:

- Інтеграція – найвищий рівень організації оперування над даними, характеризується сирцевим і цільовим конектором, є агрегацією процесів
- Процес – проміжний організаційний рівень, визначає характер отримання даних з сирцевого конектору, характер та періодичність виконання та інші налаштування
- Правило – конкретний опис маніпуляцій над даними, як-то: мапінг, цільовий об’єкт, операцію та зворотній запис.

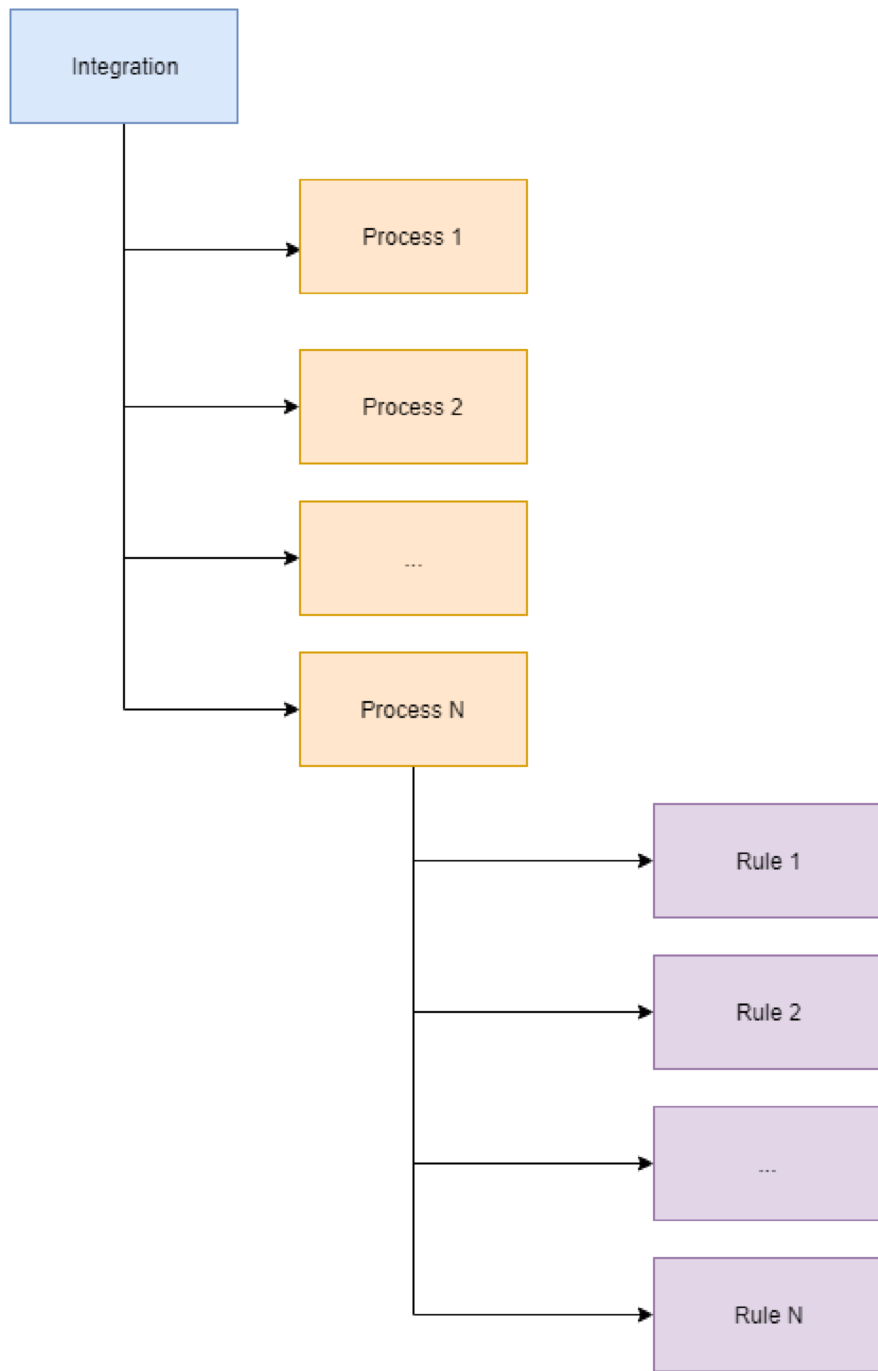


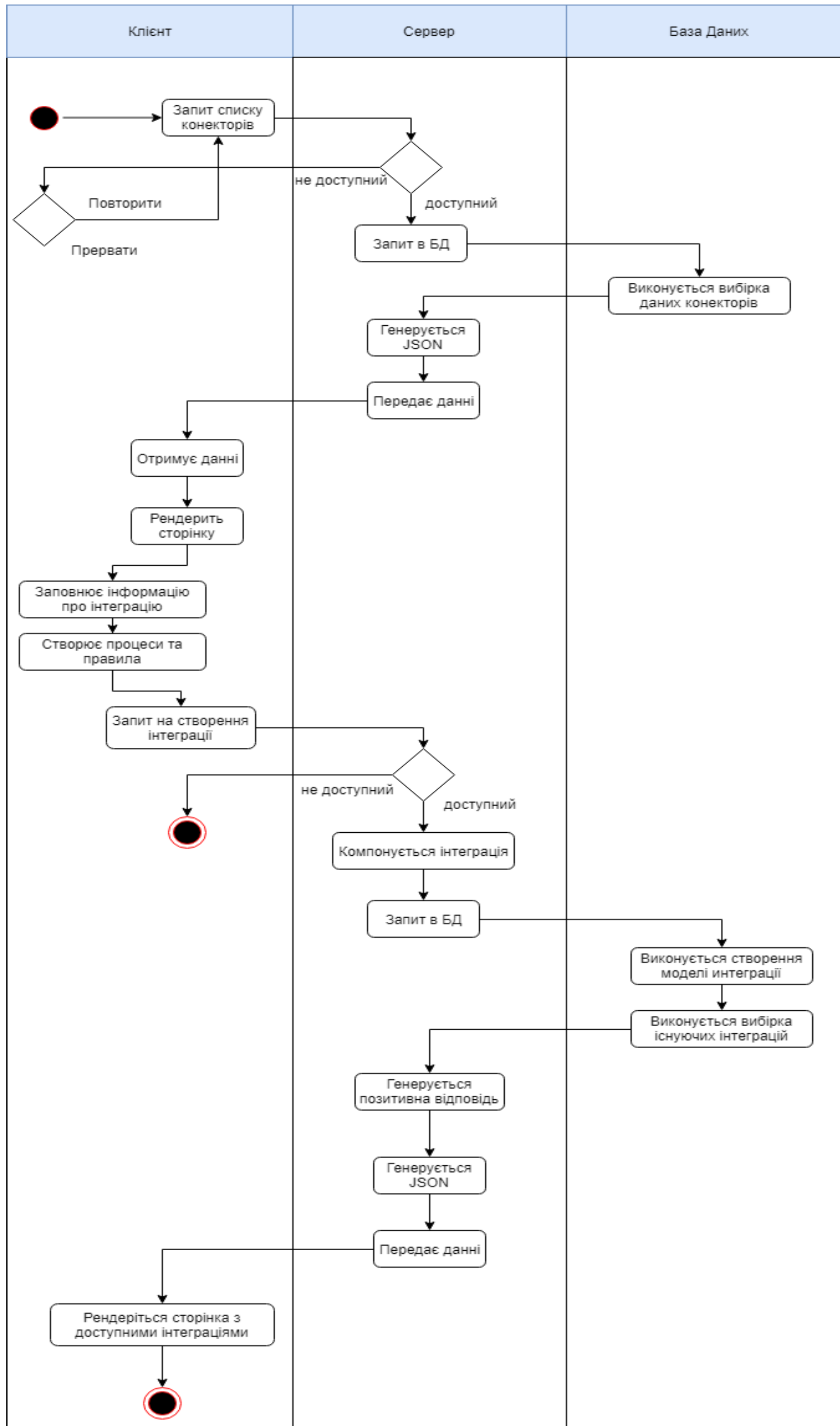
Рисунок 5.1.1 – Схема ієрархії інтеграційних сутностей

5.2 Діяльність у системі

Система в першу чергу має задовольняти різноманітні інтеграційні потреби, отже необхідно надати змогу користувачу самостійно створювати інтеграційні профіля та конфігурувати їх. Мінімальним набором конфігурації для побудови інтеграції є:

1. Вихідний та цільовий конектор
2. Перелік процесів, в яких зазначені:
   2.1 Частота оновлення даних
   2.2 Вихідний об'єкт
   2.3 Запит до вихідної системи(може бути опущений за замовчуванням – тоді буде використаний стандартний запит, специфічний до конкретної системи)
3. Перелік правил, в яких зазначені
   3.1 Цільовий об'єкт
   3.2 Назва операції
   3.3 Співвідношення між вихідними полями та цільовими(необхідно щоб було зазначено хоча б одне таке співвідношення, а також назви полів мають відповідати API-іменам у кожній конкретній системі)
   3.4 Опціонально над кожним співвідношенням можливо додати функцію, що буде форматувати дані, замість відображення один до одного(за замовчуванням значення функції буде опущено). Також треба зазначити що функції повинні бути визначені до моменту створення інтеграції, тому як користувач буде зазначений перелік

уже існуючих функцій(як базових так і створених користувачем самостійно



.

Рисунок 5.2.1 – Діаграма діяльності створення інтеграції

Опис інтеграційної конфігурації:

1. integrationName – ім’я інтеграції, використовується лише для логічно організованої ієрархії. В подальшому відображається на сторінці списку інтеграцій
2. clientId – унікальний ідентифікатор клієнта до якого буде прив’язана інтеграція у базі даних.
3. connectors - агрегована інформація про конектори
   3.1 Унікальний ідентифікатор конектора у системі
   3.2 label – мітка, що зазначає чи є конектор вихідним (source) або цільовим (target)
   3.3 name – ім’я конектору, використовується для кращою орієнтації у відповідних об’єктах системи. Відображається на сторінці усіх конекторів.
4. process
   4.1 processName: ім’я процесу, використовується лише для логічно організованої ієрархії
   4.2 sourceObject: об’єкт який буде отримуватися з вихідного конектора (повинен відповідати API-імені об’єкта)
   4.3 cronSettings: період часу за яким буде виконуватися процес. Визначається задопомогою cron-нотації.
   4.4 enabled: мітка, що визначає у якому статусі знаходиться процес, вимкненому (disabled) або увімкнутому (enabled)
   4.5 batchSize: кількість об'єктів яка будуть оброблені за одну операцію під час записування даних у цільовий конектор

4.6 onInitRun: ідентифікатор, чи запускати процес при виклику оперції старту інтеграції. Якщо значення негативне, то перший виклик процесу буде з плином часу, зазначеному у крон-налаштуваннях.

4.7 query: опціональний параметр, в якому можливо написати кастомну query в вихідний конектор. Параметр є опціональним і за його відсутності система використовує запит за замовчуванням, характерий до конкретної системи.

4.8 order: порядок процесу виконання у інтеграції

5. Rules

5.1 enabled: мітка, що визначає у якому статусі знаходиться правило, вимкненому (disabled) або увімкнутому (enabled)

5.2 order: порядок процесу виконання у процесі

5.3 writeback: позначення чи потрібно записувати результат зворотньо

5.4 name: ім’я правила

5.5 targetObject: об’єкт в який буде записаний результат

5.6 operation: вид операції

5.7 saveResponse: ідентифікатор, чи потрібно зберігати в пам’яті результат

5.8 mapping: опис відповідностей для об’єктів

5.9 lineItemMapping: опис мапінгу для об’єктів, що є дочірніми і прив’язаними до основних об’єктів системи.

5.10 writebackMapping: опис відповідностей для зворотної інтеграції, яка відбувається одразу ж після основної. Основна мета такої інтеграції, зазвичай, повідомити вихідну систему про виконану синхронізацію, що допоможе скорегувати роботу системи при подальшому процесінгу даних.

5.3 Процесінг даних

На цьому етапі відбувається ініціація процесу інтеграції. Система дістає інформацію про конфігурацію системи та її конектори. Наступним кроком відбувається ініціалізація конекторів. Система, використовуючи логін, пароль, токен та шлях надсилає запит до системи, щоб отримати токен авторизації, який забезпечить можливість здійснювати транзакції у системі. Аналогічні дії виконуються і для цільового конектору. Якщо для будь якого з конекторів аутентифікація завершилася помилкою, то виконання інтеграції буде припинено до наступної ітерації, а інформація про помилку буде відправлена до UI середовищі за допомогою технології сокетів.

Для кожного з процесів буде створена Process Job. Така конструкція в Node.js дозволяє створювати періодично виконувані завдання. Період виконання таких завдань задається як крон-запис. Таким чином можливо задати в системі будь-який варіант часового виконання. Кожний Process Job зберігається у пам’яті у словнику вигляду «ідентифікатор» - «пара конекторів». Ці дії необхідні для того, щоб при наступній ітерації ми могли звертатися до них, а не ініціалізовувати повторно, а також під час виконання правил інтеграційного процесу.

Наступним етапом відбувається видобування даних з вихідного конектору. Згідно до проектування конектору, кожен екземпляр має містити метод, що дозволяє видобувати дані за певними фільтрами та/або за допомогою написаного власноруч запиту. Система, до якої виконаний запит, повертає дані та метадані. Перші підуть для подальшого процесінгу даних, останні – для моніторингу статусу процесу та вчасної зупинки процесу під час процесінгу даних групами. Це необхідно для того, щоб економити ресурси сторонніх систем, так як більшість з них мають обмежені ліміти щодо API-запитів, а процесінг групами дозволяє зменшити кількість таких запитів.

5.4 Зіставлення відповідностей

На етапі мапінгу виконуються дії по перетворенню даних вихідної системи до формату/вигляду даних цільової системи. Мапінг відбувається у декілька етапів та дій.

1. **Стандартний мапінг**

   Ця дія виконується для тих полів, для яких не потрібні жодні маніпуляції з даними, тобто співвідношення представляє собою вигляд «один до одного». Відповідний сервіс проходить по кожному елементу і створює з нього об'єкт-обгортку. Така трансформація необхідна для того щоб за допомогою особливого синтаксису у нас була можливість отримувати не лише значення з об'єктів верхнього рівня, а й з об'єктів нижнього рівня та вкладених об'єктів. Рівні вкладеності записуються у властивості через тильду, як-то : BillingAdress~City. Це означає, що система проходить рекурсивна по вузлам об'єкту та вибере з об'єкту верхнього рівня BillingAdress значення вкладеного об'єкту City. Таким чином можливо уніфікувати процес звернення до даних для будь-якої системи та здійснювати видобування даних на будь-яку глибину. Після цього значення цього поля записується у його відповідний аналог на цільовому об'єкті. Яка ім'я задати цільовому полю зазначено у конфігурації самого правила як alias. Параметр order не є обов'язковим усюди, але це специфічна доробка яка обов'язкова необхідна для роботи із QuickBooks Desktop. Справа в тому, що система розроблена дуже давно і має специфічний протокол спілкування, тому необхідно зберігати поля у порядку, який зазначений у метадаті кожної конкретної версії цього конектору. Об'єкт із зіставленими відповідностями у полях трансформується у відповідний формат даних

(JSON, XML, QBXML) та вже готовий до використання в системі в подальшому

2. **Мапінг залежних об'єктів**

Залежні об'єкти – це об'єкти що не можуть існувати у системі без батьківського об'єкту. Вони є його частиною, а точніше списком певної інформації, що кількісно характеризує об'єкт. Фактично, для бази даних такі об'єкти є розширенням зв'язку «один до багатьох», тільки містять ще чимало інформації.

Робота з ними не відрізняється від роботи із стандартними об'єктами у плані зіставлення, але потребує додатковий маніпуляцій щоб вичленити їх із батьківських даних. Також тут можуть бути розглянуті два випадки:

1. Об'єкт буде оброблений як частина батьківського об'єкту. Такий сценарій не дуже розповсюджений, але для певних випадків можливо зустріти і такий варіант. Тоді дочірній об'єкт зіставляється стандартним образом, але кінцевим пунктом призначення є елемент верхнього рівня. Звичайно, якщо для об'єкту верхнього рівня буде визначено декілька об'єктів нижнього рівня, то кінцеві дані будуть визначатися за останнім елементом.
2. Об'єкт буде оброблений як окремий цільовий об'єкт. У конфігураціях системи можливо задати значення поля process separated, що відповідає за цей варіант обробки даних. У цьому випадку система сприймає залежний об'єкт як об'єкт вищого рівня і для нього справедливі усі твердження, які зазначені у варіанті із стандартними мапінгом. Ім'я цільового об'єкту задається у полі separated Object. За таким API-іменем, вже підвищені у ранзі, об'єкти будуть передані до цільової системи.

3. **Зворотній мапінг**

Зворотній мапінг, це мапінг який виконується у процесі зворотної інтеграції. Основною метою зворотної інтеграції є повідомлення вихідної системи про результати синхронізації даних. Така інформація дозволяє ефективніше управляти даними та/або спрощує пошук відповідностей між системами.

У випадку зворотнього мапінгу необхідно розглянути два можливих варіанта використання.

1. Об’єкт буде оброблений як вихідний об’єкт. Стандартна практика полягає у тому, щоб записувати ідентефікатор новоствореного об’єкту у вихідну систему. Для доступу для таких даних необхідно використовувати вище описаний синтаксис через тильду. Але замість основного об’єкту першим елементом цього ланцюга стає строковий літерал result. Від ідентифікує, що дані були отримані саме з відповіді цільової системи. Якщо операція була завершена не успішно, то в цей ідентифікатор не буде нічого записано і будь-які подальші операції над цим об’єктом втрачають сенс. Так як в нашому випадку ми залишаємося частиною вихідного об’єкту, то операція, що буде виконана – оновлення. Для оновлення необхідний ідентифікатор вихідного об’єкту, що отримується аналогічно, але першим елементом ланцюга стає строковий літерал entry. Від ідентифікує, що дані були отримані саме з початкової інформації. З точки зору самого процесу такий мапінг не відрізняється від стандартного мапінгу і для нього справедливі усі твердження, що і у випадку стандартного мапінгу.

2. Об’єкт буде оброблений як окремий інший об’єкт. Зазвичай, відповідно до специфіки таких систем, деякі об’єкти не є аналогічними за їх бізнес значенням. Тому у деяких випадках логічно записувати результати виконаної роботи у інший об’єкт, який більш точно характеризує цю

відповідність. Прикладом можуть бути виставлені рахунки, якими оперує кожна ERP-система, але які не є базовими елементами у CRM-системі. Для обробки мапінгу таким чином необхідно задати додатнім значенням параметр writeback.

4. **«Пошуковий» мапінг**

Дуже часто об'єкт має посилання на інші об'єкти і у більшості випадків такі посилання є обов'язковими. Таким чином ми не можемо створити об'єкт у цільовій системі на основі наявного об'єму даних. Цю проблемо можливо вирішити якщо у нашому наборі даних є зовнішні ідентифікатори необхідних нам об'єктів (за умови що такі об'єкти вже зіставлені у попередніх процесах, але це вже стосується правильного порядку в інтеграційних процесах).

Таким чином маючі зворотні ідентифікатори ми можемо виконати запит до цільової системи і отримати необхідні нам ключі. Це дуже зручно, але може сильно вдарити по продуктивності системи. Тут можливі два виходи, але кожен з них має свої області використання

1. Обробляти посилання по одному. На маленьких об'ємах даних і невеликої (1-2) кількості посилань - варіант з один запитом до системи для кожного посилання не є катастрофічним. Але реальні потреби клієнтів зазвичай значно ширші, тому цей варіант є лише плацдармом для вирішення проблеми
2. Обробляти усі дані разом. Протилежним варіантом є варіант, за яким ми отримуємо усі дані із системи та вже в пам'яті оперуємо ними та достаємо необхідні нам ідентифікатори. Цей підхід значно кращий за перший, бо використовує лише один API – виклик для такої операції. У системи не буде значних проблем з пам'яттю, тому здається що можливо це рішення проблеми. Але постає нова проблема – більшість систем мають обмеження

за кількістю даних, які система може віддати за один запит. Для Salesforce, наприклад, оскільки кількість становить 2000 одиниць, та навіть якщо уявити що ми можемо отримати більше це не панацея. Деякі об'єкти мають особливість швидко нарощувати свою кількість і за 1-2 роки роботи у системі їх кількість становить декілька мільйонів. Прикладом такого об'єкта можуть бути Case або Email. Тому ми не можемо оперувати такими великими масивами даних у пам'яті.

3. Використовувати групи даних. Рішення, що повністю задовольняє потреби системи є більш складним у реалізації. Ми повинні визначати групи даних та отримувати їх з системи за певним критерієм. Це може бути дата створення, модифікацій або авто-інкрементований порядковий номер. Таким чином ми можемо регулювати об'єми даних и не впиратися у ліміти систем з видобування даних. Але такий підхід має свою ціну – для кожної конкретної системи таких механізм повинен бути розроблений окремо, що підвищує складність розробки та підтримки системи.

Алгоритм специфікації такого посилання необхідно задати у зіставленні кожного конкретного поля. Мапінг поля характеризується також додатковим параметром func, який буде детальніше описаний нижче. Але для обробки посилань необхідно використовувати спеціальну функцію, що є унікальною для кожної системи, та специфічний синтаксис у полі alias. Використання такої функції, наприклад SFLookup для системи Salesforce дозволяє системі зрозуміти, що вона обробляє не звичайне поле, а поле що необхідно додаткового обробити, зіставив посилання з цільової системи. У полі alias необхідно написати запит, який буде діставати необхідну інформацію з цільової системи. Системи використовую різні протоколи взаємодії, навіть різні мови запиту, тому такий запит є унікальним для кожної системи. Розглянемо на прикладі Salesforce:

```
"Pricebook2Id": {
```

```
  "func": "SFLookup",
  "alias": "SELECT Id FROM Pricebook2 WHERE IsStandard=true"
}
```

Синтаксис у Salesforce аналогічний до мови SQL, тому після select ми передаємо першим параметром поле, яке ми хочемо бачити у якості посилання, а далі йде умовна фільтрація, що дозволяє знайти необхідне значення. У системі також є можливість вибрати декілька параметрів, а потім обробити їх за допомоги функцій. Для цього параметри передаються через кому після першого з них.

5. **Мапінг функціями**

Розповсюдженим варіантом використання мапінгу є трансформування даних за допомогою не стандартних функцій. Саме тому необхідно підхід до створення, застосування і пере використання користувацький функцій. Для цього в системі використовується не зовсім безпечний, але дуже потужний механізм функції eval.

Метод eval () виконує JavaScript код, представлений строкою. Таким чином ми можем прямо в JSON конфігурації визначати функції у строковому представленні та виконувати їх під час виконання програма. Стандартною практикою для таких випадків у статично типізованих мовах програмування є використання рефлексії. Але кожний виклик рефлексії завдає шкоду продуктивності системи, що в випадку із великою кількість таких запитів є критичним. Але TypeScript компілюється у JavaScript і насправді не має типів – тому ми можемо собі дозволити підвищити продуктивність за рахунок використання низькорівневих інтерфейсів.

Для опису такої функції у мапінгу кожного конкретного поля необхідно задати параметр func. На UI – частині це відбувається за допомогою вибору з випадаючого списку. У списку присутні стандартні функції, які визначають

функціональні операції над строками. Прикладом таких функції є split, contact, і таке інше. Також користувач може заздалегідь самостійно визначити функції, що будуть використані під час інтеграційного процесу. Зазначмо, що функції не можуть бути транспортовані між різними інтеграціями, а лише скопійовані та про ініційовані повторно.

```
(value, entry, state) => {
        for (const array of state) {
            for (const element of array) {

                let entryElement = element.entry;

                if (entryElement.ListID.split('-')[0] == entry.ListID.split('-')[0]) {
                    return element.result.id;
                }
            }
        }
}
```

Рисунок 5.5.1 – Приклад функції для мапінгу

Розглянемо основні елементи такої функції.

1. Value – значення елементу
2. Entry - параметр, в який в коді буде передне реальне значення поля, що оброблюється на поточній операції
3. State – параметр, який відповідає сховищу, до якого є доступ у runtime. Це необхідно для того, щоб під час виконання послідовних запитів, процесів була можливість зберегти результати виконання і пере використати в наступних процесах. Це дуже корисно коли інтеграція вимагає окремої обробки батьківських та дочірніх об’єктів, або поточний об’єкт має посилання на об’єкт інвентаризації, як-то продукт або ціновий лист. Для того щоб зберегти значення будь якого правила, необхідно задати в його налаштуванні saveResponse у значення дійсно. State представляє собою

масив масивів і кожне збережене правило буде займати окрему комірку з інформацією.

При такому підході ми можемо використовувати усі функції JavaScript та навіть будь-які функції, що зазанчені в нашому коді та знаходяться у тієї ж самої області видимості. На рисунку зображений приклад використання стандартної функції split. Також аналогічним чином можливо реалізовувати логувані із середині функції.

```
(value, entry, state) => {
        console.log('Start processing')
        for (const array of state) {
            for (const element of array) {

                let entryElement = element.entry;

                console.log(entryElement)

                if (entryElement.ListID.split('-')[0] == entry.ListID.split('-')[0]) {
                    console.log('Finish processing')
                    return element.result.id;
                }
            }
        }
}
```

Рисунок 5.5.2 – Приклад функції для мапінгу з логуванням

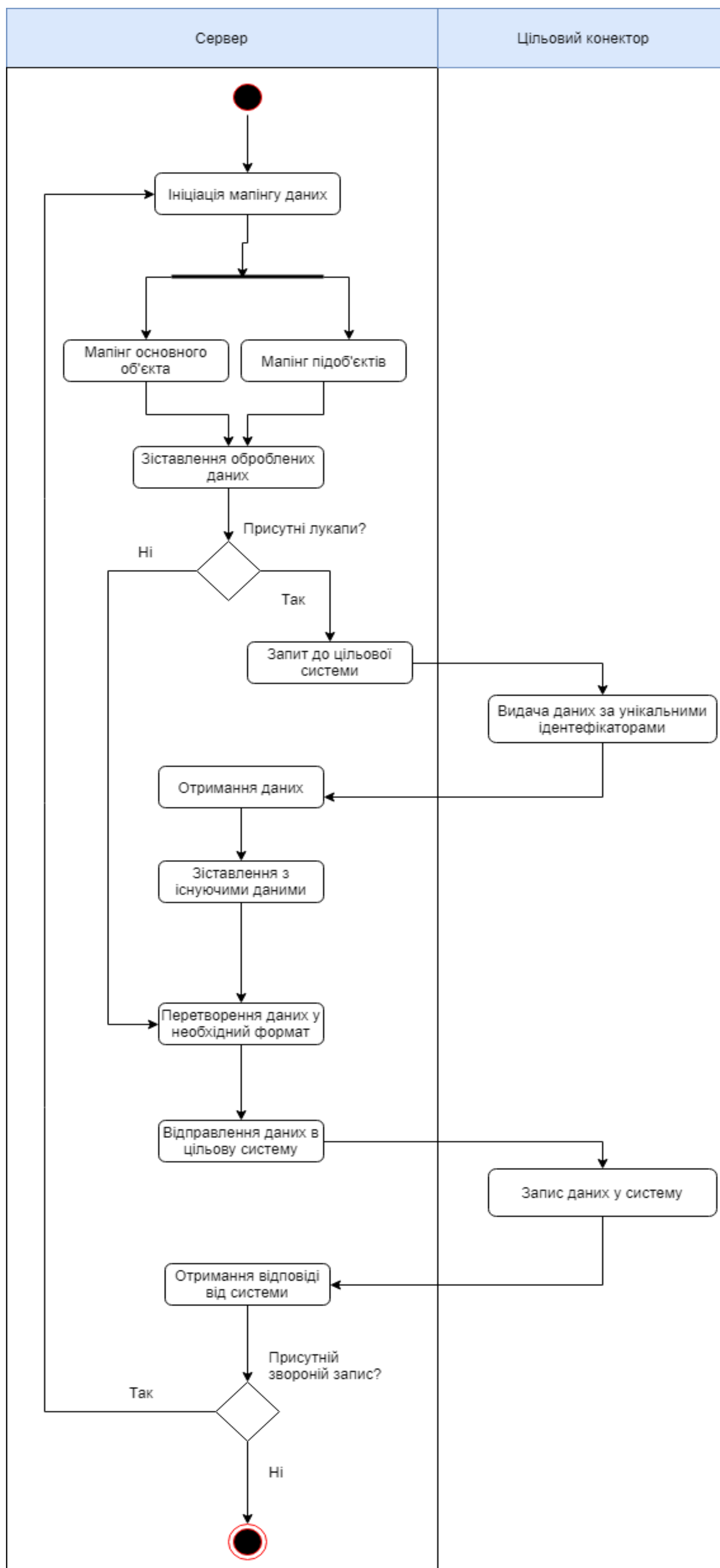


Рисунок 5.5.3 – Приклад діяльності запуску інтеграції

РОЗДІЛ 5

ПОБУДОВА КОНЕКТОРІВ

5.1 Метадата Salesforce

Для того щоб будувати будь-які процеси з CRM-системою Salesforce необхідно розміти як організована метадата.

Компоненти та типи метаданих

Компоненти метаданих не засновані на sObjects, як об'єкти в API. Натомість вони засновані на типах метаданих, таких як ApexClass і CustomObject, які розширюють метадані, базовий клас для всіх типів метаданих. Компонент - це екземпляр типу метаданих. Наприклад, CustomObject - це тип метаданих для призначених для користувача об'єктів, а компонент MyCustomObject__c - екземпляр спеціального об'єкта.

Тип метаданих може бути ідентифікований у метаданих WSDL як будь-який комплексний тип, який розширює метадані complexType. Комплексний тип, який є типом метаданих, містить у своєму визначенні WSDL наступний елемент:

<xsd:extension base="tns:Metadata">

CustomObject і BusinessProcess розширюють метадані, тому вони є типами метаданих; ActionOverride не розширює метаданих, тому це не тип метаданих.

Можливо індивідуально розгорнути або отримати компонент для типу метаданих. Наприклад, ви можете отримати окремий компонент BusinessProcess, але ви не можете отримати окремий компонент ActionOverride. Ви можете отримати компонент ActionOverride, лише отримавши його компонент CustomObject.

Компонентами метаданих можна керувати за допомогою асинхронних викликів API метаданих або декларативних (або файлових) викликів API метаданих.

Поля в метадаті

Кожне поле компонента має певний тип поля. Ці типи полів можуть відповідати іншим компонентам, визначеним у WSDL, або примітивними типами даних, наприклад рядком, які зазвичай використовуються в сильно типізованих мовах програмування.

Ці типи даних поля використовуються в повідомленнях SOAP, які обмінюються між вашим клієнтським додатком і API. Під час написання клієнтської програми дотримуйтесь правил введення даних, визначених для мови програмування та середовища розробки. Інструмент розробки обробляє відображення набраних даних у мові програмування з тими SOAP-типами даних.

Перелікові поля

Деякі поля компонентів мають тип даних, який є перерахуванням. Перелік – це еквівалент API списку вибору. Допустимі значення поля обмежуються строгим набором можливих значень, які мають однаковий тип даних. Ці значення перелічені у стовпці опису поля для кожного поля переліку. Див. SortBy для прикладу поля переліку типу string. Наведений нижче XML показує приклад визначення переліку рядкових типів у WSDL.

```
1 <xsd:simpleType name="DashboardComponentFilter">
2     <xsd:restriction base="xsd:string">
3         <xsd:enumeration value="RowLabelAscending"/>
4         <xsd:enumeration value="RowLabelDescending"/>
5         <xsd:enumeration value="RowValueAscending"/>
6         <xsd:enumeration value="RowValueDescending"/>
7     </xsd:restriction>
8 </xsd:simpleType>
```

5.2 Побудова Salesforce конектору

Застосунок базується на принципах поліморфізму тобто для роботи з різними системами в ідеалі достатньо лише змінити імплементацію інтерфейсів з’єднувача. В реальності ж більшість ERP систем настільки, що під кожну доводиться дописувати власний функціонал, або ж подавати дані в дуже специфічному форматі без мінімальних відхилень. Також все часто ускладняється відсутністю правильної не застарілої документації, або не існуючою службою підтримки. Проте для CRM систем, локальних або хмарних баз даних та CSV файлів додаткових проблем майже не виникає, тому в більшості випадків достатньо реалізувати абстрактний клас Connector.

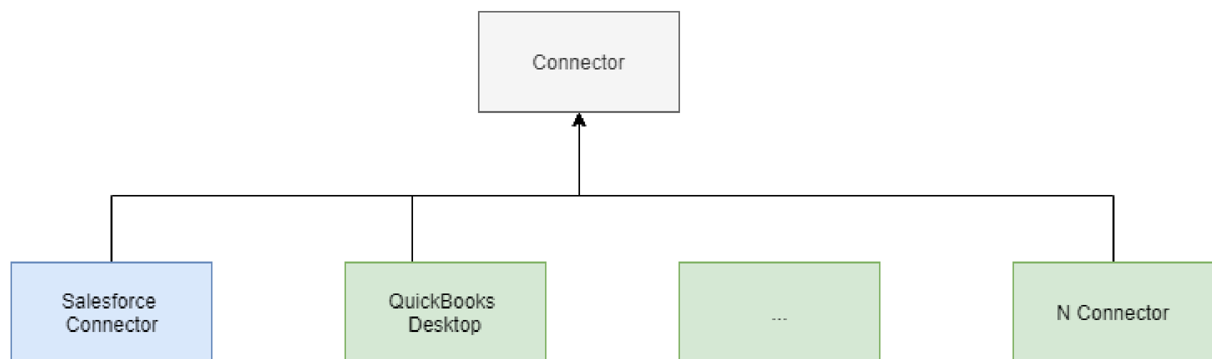


Рисунок 5.2.1 – Загальна схема реалізації конектору

Перлік визначенийх операцій:

1. Init – операція з якої починаєтсья будь-яка робота з конектором. Система передає інформацію для авторизації і отримує підключення до цільової системи.
2. DeInit – для певних систем необхідно закривати підключення, або відправляьт певну інформацію про завершення роботи. Не використовується для Salesforce або QuickBooks Desktop.
3. FindById, FindByClose – стандартні операції пошуку за іменем об’єкта у першому випадку та кастомним запитом в останньому.

4. CUD-операції – стандартні операції оперування даними в API.
5. AOP Hooks – набір операцій, що визначать дії після певної події. Такі тригери на події допомагають краще управляти процесом інтеграції і підключенням. Загалом визначено 4 хуки – beforeAll, afterAll, beforeAction, afterAction.

```
export default abstract class Connector<Connection = unknown, Credentials = unknown> {
    protected id: string;

    protected userId: string;

    protected connection: Connection;

    private readonly nonImplementedMessage = "Method not implemented";

    constructor(id: string, userId: string) {

        if (!id) {
            return;
        }

        this.id = id;
        this.userId = userId;
    }

    /* AOP Hooks */
    public async beforeAll() {}

    public async afterAll() {}

    public async beforeAction() {}

    public async afterAction() {}

    public getConnection() {
        return this.connection;
    }

    /* Initialization */
    public abstract init(credentials: Credentials, ...args: any[]): Promise<void | Connection>;

    public abstract deinit(): Promise<void>;

    /* Data retrieval */
    public abstract findById(objectName: string, id: string): Promise<object>;

    public abstract findByClause(objectName: string, offset: number, limit: number, query: object |
string, meta?: any, ...args: any[]): Promise<any>;

    /* Batch entity */
    public abstract insert(objectName: string, data: MappingResult): Promise<object[]>;

    public abstract upsert(objectName: string, data: MappingResult, externalId?: string):
Promise<object[]>;

    public abstract update(objectName: string, data: MappingResult, externalIdField?: string):
Promise<object[]>;

    /* Signle entity */
    public abstract insertOne(objectName: string, data: SingleMappingResult): Promise<object>;

    public abstract upsertOne(objectName: string, data: SingleMappingResult, externalId: string):
Promise<object>;

    public abstract updateOne(objectName: string, data: SingleMappingResult): Promise<object>;

  /* validation */
    public async validate(credentials: Credentials): Promise<object> {
        throw Error(this.nonImplementedMessage);
    }
}
```

Рисунок 5.2.2 – Опис протоколу реалізації методів конектору

Для того щоб не будувати підключення руками і не працювати напряму з Salesforce API у системи використана обгортка над API – Jsforce.

JSforce - ізоморфна бібліотека JavaScript, яка використовує API Salesforce: він працює як у браузері, так і з Node.js.Він перетворює доступ до різних API, наданих Salesforce в асинхронних викликах функцій JavaScript.Вона також має інтерфейс командного рядка (CLI), який дає інтерактивну консоль (REPL), так що ви можете вивчити використання без проблем.

Підтримувані API для Salesforce такі:

- REST API (SOQL, SOSL, опис і т.д.)
- Apex REST
- API Analytics
- Масовий API
- Chatter API
- API метаданих
- SOAP API
- Потоковий API
- API інструментів

Нижчено наведено приклад використання Jsforce для створення підключення до Salesforce API.

Щоб авторизуватися за допомогою такого метода, необхідно створити конектор з такою конфігурацією

```
{
  "username": "test1@test.com",
  "password": "password1",
  "loginUrl": "https://login.salesforce.com"
}

{
  "username": "test2@test.com",
  "password": "password2",
  "loginUrl": "https://test.salesforce.com"
}
```

У першому випадку система буде авторизуватися у production версію Salesforce і в тестовий у другому. Також, опціонально, при певних налаштуваннях системи необхідно додати токен.

5.3 Побудова QuickBooks Desktop конектору

З точки зору розробленого застосунку реалізація цього конектору не відрізняється від попередника. Але з цією системою є одна велика проблема – її проектували 30 років тому і не передбачалась будь-яка інтеграція зі сторонніми системами. Тому QuickBooks Desktop не має API та навіть підключення до інтернету. У цьому і полягає головна особливість і складність інтеграції з цією системою.

Побудувати інтеграцію можливо наступним чином. Розробити окремий додаток, сервер, що буде реалізувати специфічній протокол комунікації QuickBooks Desktop та управляти формуванням даних у формат QBXML. Такий додаток буде генерувати .qbw файл, який можна завантажити на QB Web Service. Тоді цей додаток буде помічений системою і кожні 60 секунд вона буде перевіряти чергу на наявність будь-яких запитів до неї.

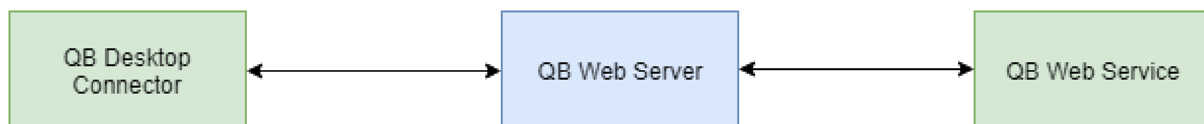


Рисунок 5.2.3 – Схематичне зображення зв'язків між модулями

Основною проблемою є сам формат даних – QBXML. Формат маэ набір специфічних атрибутів та відрізняється від однієї версії системи до іншої. Саме для цих потреб був розроблений QbXmlGenerator який будує необхідний запит відповідно до об’єкта, типу метода та додаткових атрибутів.

Визначений набір операцій містить: select, update, insert, deleteList, deleteTransaction.

```
export class QBXmlGenerator {

    callMethod(method: QbMethod, args: QbMethodArgs): string {
        return this[method](...args);
    }

    select(id: string, objectName: string, fields: FieldsMap) {
        return this.generateRequest(id, objectName, fields, "QueryRq");
    }

    update(id: string, objectName: string, fields: FieldsMap) {
        return this.generateRequest(id, objectName, fields, "ModRq");
    }

    deleteList(id: string, objectName: string, fields: FieldsMap) {
        return this.generateRequest(id, objectName, fields, "ListDel");
    }

    deleteTransaction(id: string, objectName: string, fields: FieldsMap) {
        return this.generateRequest(id, objectName, fields, "TxnDelRq");
    }

    insert(id: string, objectName: string, fields: FieldsMap) {
        return this.generateRequest(id, objectName, fields, "AddRq");

    }

    accpetMethod(name: string): boolean {
        const supported = ['select', 'update', 'insert', 'deleteList', 'deleteTransaction'];

        return supported.includes(name);
    }


    private generateRequest(id: string, objectName: string, fields: FieldsMap, mod: string) {
        let base = this.generateBase()
            .ele(`${objectName}${mod}`)
            .att("requestID", id);

        if(mod === 'QueryRq') {
            if(fields.iteratorID) {
                base.att("iterator", 'Continue');
                base.att("iteratorID", fields.iteratorID);

                delete fields.iteratorID;
            } else
                base.att("iterator", 'Start');
        }

        base = this.generateFields(base, fields);
        return this.generateClosing(base);
    }

    private generateBase() {
        return builder.create('QBXML', { version: '1.0', encoding: "utf-8" })
            .ele('QBXMLMsgsRq', { 'onError': 'stopOnError' })

    }

    private generateClosing(xml: XMLElementOrXMLNode) {
        return xml.end({ 'pretty': true }).replace('<?xml version="1.0" encoding="utf-8"?>', `<?xml
version="1.0" encoding="utf-8"?>\n<?qbxml version="13.0"?>`);
    }

    private generateFields(xml: XMLElementOrXMLNode, fields: FieldsMap | FieldsMap[]) {
        if (Array.isArray(fields)) {
            for (const fieldsmap of fields){

                this.generateFields(xml, fieldsmap);
            }

            return xml;
        }

        for (const name of sortFields(fields)) {

            if (Array.isArray(fields[name])) {
                for (const fieldsmap of (fields as any)[name]){

                    this.generateFields(xml, {[name]: fieldsmap});
                }

                continue;
            }
            let subxml;

            const { value, order, ...children } = fields[name];

            if (children && Object.keys(children).length > 0) {
                subxml = xml.ele(name);

                this.generateFields(subxml, children);
            } else {
                subxml = xml.ele(name).t(String(value));
                xml.up();
            }
        }

        return xml;
    }
}
```

Рисунок 5.2.3 – Реалізація генератору запитів

ВИСНОВКИ

У роботі було розглянуто особливості форматів даних та методів їх передачі для систем керування даними. Виокремлено критичні аспекти та методи їх реалізації. Описано стандарт формату даних, що охоплює увесь спектр операції над полем даних.

Під час роботи були вирішені проблеми, що пов'язані із доступом до даних та лімітами систем щодо певних системних операцій. Так у QuickBooks відсутня вбудована операція Upsert, яку довелося симулювати через Get => insert/update, відсутня операція Delete, а операцію Update неможливо провести без отримання запису який ми хочемо оновити через динамічний унікальний ідентифікатор який прив'язаний до timestamp.

Проведено дослідження і тестування застосунку на таких системах як Salesforce, QuickBooks, Epicor та NetSuite. Ці системи є лідерами індустрії з керування та передачі бізнес-критичних даних, тому успішні проведені тести із зазначеними системами можуть бути гарантом правильності зроблених висновків, щодо стандарту побудови інтеграційних процесів.

Для розробки застосунку було використано сучасні технології, що дозволили побудувати гнучке, масштабоване рішення і повністю виправдали затрати на час їх впровадження.

Враховуючи коректність зроблених припущень і реалізацію застосунку до кінцевого продукту – завдання роботи виконано повністю.

Список джерел

1. Система керування базами даних PostrgeSQL [Електронний ресурс]. – Режим доступу : https://www.postgresql.org/
2. Документація CRM системи Salesforce [Електронний ресурс]. – Режим доступу : https://developer.salesforce.com/docs/
3. Фреймворк ODI [Електронний ресурс]. – Режим доступу : https://github.com/Odi-ts/odi
4. Платформа Node.js [Електронний ресурс]. – Режим доступу : https://nodejs.org/
5. Бібліотека React [Електронний ресурс]. – Режим доступу : https://reactjs.org/
6. Хмарна платформа Amazon Web Services [Електронний рурс]. – Режим доступу : https://aws.amazon.com/
7. QB Online Documentation [Електронний ресурс]. –
8. Режим доступу: https://developer.intuit.com/app/developer/qbo/docs/develop
9. Технології веб-інтеграцій [Електронний ресурс]. – Режим доступу : http://www.redoaksw.com/products/webclipper/RedOak_WebIntegrationOverview.pdf
10. Форми інтеграцій [Електронний ресурс]. – Режим доступу : https://www.orbitmedia.com/blog/4-basic-forms-web-integration/
11. Стандарт RFC7617 [Електронний ресурс]. – Режим доступу : https://datatracker.ietf.org/doc/html/rfc7617

ДОДАТКИ

Додаток А Лістинг інтеграційного застосунку

```
export type QuickBooks = typeof QuickBooks;
export interface QuickBookOnlineDetails {
  accessToken: string;
  refreshToken: string;
  realmId: string;
  sandbox: boolean;
  quickBookAppId: string;
  quickBookAppSecret: string;
}

export default class QuickBookOnlineConnector extends Connector<QuickBooks, QuickBookOnlineDetails> {

  protected connection: QuickBooks;

  private readonly MAX_RECORDS_IN_BATCH = 30;

  private readonly OFFSET = 7 * 24 * 60 * 60 * 1000;

  private readonly ACCESS_TOKEN = "access_token";

  private readonly REFRESH_TOKEN = "refresh_token";
  private credentials: QuickBookOnlineDetails;

  private readonly notificationLogger: NotificationLogger = getContainer().get(NotificationLogger) as NotificationLogger;

  // tslint:disable-next-line: no-empty
  public async deinit(): Promise<void> {
  }

  public insertOne(objectName: string, data: SingleMappingResult): Promise<object> {
    throw new Error("Method not implemented.");
  }

  public upsertOne(objectName: string, data: SingleMappingResult, externalId: string): Promise<object> {
    throw new Error("Method not implemented.");
  }

  public updateOne(objectName: string, data: SingleMappingResult): Promise<object> {
    throw new Error("Method not implemented.");
  }

  public async beforeAll() {
    const result = await this.refreshToken() as any;

    this.initConnection({
      ...this.credentials,
      refreshToken: result[this.REFRESH_TOKEN],
      accessToken: result[this.ACCESS_TOKEN]
    });

    await this.updateConnector(result);

  }

  public async init(details: QuickBookOnlineDetails) {
    this.credentials = details;
```

```
        this.initConnection(details);
    }

    public async findById(objectName: string, id: string): Promise<any> {
        return this.getAction<QBFindPromise>(QuickBookOnlineActions.GET, objectName)(id);
    }

    public async findByClause(objectName: string, offset: number, limit: number, query: object | string, meta: any, lastRunDate:
number, lastIterationRunDate: number , dateSelector: string, dateBased: boolean): Promise<any> {

        let filters: any[] = [];

        if (dateBased) {
            const filledDate = (lastIterationRunDate ? lastIterationRunDate * 1000 : lastRunDate);

            const date = new Date(filledDate);

            date.setHours(date.getHours() - Number.parseInt(process.env.TIME_OFFSET ?? "7"));

            filters = this.buildFilters(moment(date).format("YYYY-MM-DDTHH:mm:ss"), dateSelector, limit);
        }

        const queryAction = this.getAction<QBFindPromise>(QuickBookOnlineActions.FIND, objectName);

        try {
            const { QueryResponse } = filters.length === 0 ? await queryAction(query) : await queryAction(filters);

            const fetched = QueryResponse[objectName];

            return { fetched };
        } catch (ex) {
            await this.notificationLogger.emitNotification(this.userId, NotificationType.Error, "his message is a caution that the
Quickbooks Online instance can not be reached. Please contact support if this message repeats");
            await createLog({ title: "Find by Clause", operation: OperationTypes.Find, message: `Quickbooks Online querying
failed.
            Query: ${filters.map(f => `field: ${f.field}, value: ${f.value}, operator: ${f.operator}`)}.
            Exception: ${ex}`, connector: this.id, type: LogTypes.Error }, this.userId);
        }
    }

    public async insert(objectName: string, data: MappingResult): Promise<any> {

        const { mapped } = data;

        const batch = this.getAction<QBBatchPromise>(QuickBookOnlineActions.BATCH, objectName);

        const mappedBatch = mapped.map(mappings => ({
            bId: uuid(),
            [objectName]: {
                ...mappings
            },
            operation: "create"
        }));

        const chunks = _.chunk(mappedBatch, this.MAX_RECORDS_IN_BATCH);

        const result: object[] = [];

        for (const entries of chunks) {
            try {
                const batchResult = await batch(entries);
```

```
            (batchResult.BatchItemResponse as object[]).forEach(value => {
               result.push((value as any)[objectName]);
            });

         } catch (ex) {
            await this.notificationLogger.emitNotification(this.userId, NotificationType.Error, "his message is a caution that the Quickbooks Online instance can not be reached. Please contact support if this message repeats");
            await createLog({ title: "Insert", operation: OperationTypes.Find, message: `Insert to Quickbooks Online failed. ObjectName: ${objectName}. Exception: ${JSON.stringify(ex.fault.error, null, 4)}`, connector: this.id, type: LogTypes.Error }, this.userId);
         }

      }
      return result;
   }

   public async upsert(objectName: string, data: MappingResult, externalIdField: string): Promise<any> {
      const result: object[] = [];

      const { mapped, entries } = data;

      const update = this.getAction<QBUpdatePromise>(QuickBookOnlineActions.UPDATE, objectName);
      const create = this.getAction<QBCreatePromise>(QuickBookOnlineActions.CREATE, objectName);

      for (let index = 0; index < mapped.length; index++) {
         const elem = mapped[index];
         const entryData = entries[index] as any;

         const nestingEntry = new NestingObject(entryData);

         const externalIdValue = nestingEntry.getProperty(externalIdField);

         const prst = externalIdValue ? await this.findById(objectName, externalIdValue) : null;

         try {
            if (prst) {
               result.push(await update({ SyncToken: prst.SyncToken, ...prst, ...elem }));
            } else {
               result.push(await create(elem));
            }
         } catch (err) {
            await createLog({ title: "Upsert", operation: OperationTypes.Find, message: `Upsert to Quickbooks Online failed. ObjectName: ${objectName}. Exception: ${JSON.stringify(err, null, 4)}`, connector: this.id, type: LogTypes.Error }, this.userId);
            result.push(elem);
         }
      }
      return result;
   }

   public async update(objectName: string, data: MappingResult, externalIdField: string): Promise<any> {
      const update = this.getAction<QBUpdatePromise>(QuickBookOnlineActions.UPDATE, objectName);

      const { mapped, entries } = data;

      for (let index = 0; index < mapped.length; index++) {
         const elem = mapped[index];
         const entryData = entries[index];

         if (!entryData[externalIdField]) {
            continue;
```

```
            }

            const prst = await this.findById(objectName, entryData[externalIdField]);

            try {
                await update({ ...prst, elem });
            } catch (ex) {
                await createLog({ title: "Update", operation: OperationTypes.Find, message: `Update to Quickbooks Online failed.
ObjectName: ${objectName}. Exception: ${JSON.stringify(ex, null, 4)}`, connector: this.id, type: LogTypes.Error },
this.userId);
            }
        }
    }

    private async updateConnector(result: any) {

        await runTransaction(IsolationLevel.READ_COMMITTED, async manager => {
            const connectorRepository = manager.getRepository(ConnectorModel);

            const connector = await connectorRepository.findOne(this.id);

            if (!connector) {
                throw new IHttpError("Connector does not exist", 400);
            }

            const credentials = connector.credentials as QuickBookOnlineDetails;

            credentials.accessToken = result[this.ACCESS_TOKEN];
            credentials.refreshToken = result[this.REFRESH_TOKEN];

            await connectorRepository.update(connector.id, { credentials });
        });

    }

    private getAction<T>(operation: QuickBookOnlineActions, objectName: string): T {
        let alias = objectName;

        if (operation === QuickBookOnlineActions.BATCH) {
            return promisify<any>(this.connection.batch);
        }

        if (operation === QuickBookOnlineActions.FIND) {
            alias += "s";
        }

        return promisify<any>(this.connection[operation + alias]);
    }

    private buildFilters(lastRunDate: string, dateSelector: string, limit: number) {

        const formattedSelector = dateSelector.includes("~") ? dateSelector.split("~").join(".") : dateSelector;

        const formattedLastDate = lastRunDate.toString().includes(".") ? lastRunDate.toString().split(".")[0] : lastRunDate;

        if (!lastRunDate || !formattedSelector) {
            return [{
                field: "limit",
                value: limit
            }];
        }
```

```
        return [{
            field: formattedSelector,
            value: formattedLastDate,
            operator: ">"
        },
        {
            field: "limit",
            value: limit
        }];
    }

    private refreshToken() {
        return new Promise((resolve, reject) => this.connection.refreshAccessToken((err: any, result: {} | PromiseLike<{}> |
undefined) => err ? reject(err) : resolve(result)));
    }

    private initConnection({ refreshToken, accessToken, realmId, sandbox, quickBookAppId, quickBookAppSecret }:
QuickBookOnlineDetails) {

        this.connection = new QuickBooks(
            quickBookAppId,
            quickBookAppSecret,
            accessToken, /* oAuth access token */
            false, /* no token secret for oAuth 2.0 */
            realmId,
            sandbox, /* use a sandbox account */
            false, /* turn debugging on */
            null, /* minor version */
            "2.0", /* oauth version */
            refreshToken /* refresh token */
        );
    }
}
```