Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет *Комп'ютерних наук*
(повна назва)

Кафедра *Системотехніки*
(повна назва)

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

МАГІСТР
(освітньо-кваліфікаційний рівень)

ГЮИК.501500.006 ПЗ
(позначення документа)

Дослідження особливостей розробки сервісів орієнтованих на підтримку динамічного розвитку бізнесу
(тема)

Виконав: студент *VI* курсу, групи *ІТПм-19-1*
напряму підготовки (спеціальності) _____
122 – Комп'ютерні науки
(шифр і назва напряму, спеціальності)

Стрименешенко О.С.
(прізвище, ініціали)

Керівник *проф. Іванов В.Г.*
(прізвище, ініціали)

Рецензент _____
(прізвище, ініціали)

Допускається до захисту

Зав. кафедри СТ _____ *проф. Гребеннік І.В.*
(підпис) (прізвище, ініціали)

2020 р.

Харківський національний університет радіоелектроніки

Факультет *Комп'ютерних наук*
Кафедра *Системотехніки*
Освітньо-кваліфікаційний рівень *Магістр*
Напрям підготовки *6.050101 – Комп'ютерні науки*
(шифр і назва)
Спеціальність *122 – Комп'ютерні науки*
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 20 ___ р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

студентові *Стрименешенку Олександру Сергійовичу*
(прізвище, ім'я, по батькові)

1. Тема роботи (проекту) *Дослідження особливостей розробки сервісів орієнтованих на підтримку динамічного розвитку бізнесу*

затверджена наказом по університету від "02 "11.2020 р. № *1517 Ст*
2. Термін подання студентом роботи (проекту) *24 грудня 2020 р.*
3.Вихідні дані до роботи (проекту) *Дослідження особливостей розробки сервісів орієнтованих на підтримку динамічного розвитку бізнесу. Перелік використаних програмних засобів: NotePad++, Microsoft Word 2016, ОС Microsoft Windows 10. Технічне забезпечення: персональний комп'ютер зі встановленим браузером. На ПК має бути встановлена Java 8, JDK та середовище розробки IntelliJ IDEA.*

4. Зміст пояснювальної записки (перелік питань, що потрібно розробити) *4.1Вступ. 4.2 Огляд та аналіз розробки сервісів як процесу прискорення і вдосконалення створення додатків. 4.3 Постановка задачі 4.4 Практична ефективність розбиття монолітної системи на сервіси. 4.5 Дослідження розбиття системи та компоновка веб-сервісів. 4.6 Дослідження технології створення сервісів для підтримки динамічного розвитку. 4.7 Дослідження вибору СУБД. 4.8 Розробка API для базового функціоналу бізнес-функцій. 4.9 Тестування розроблених сервісів. 4.10 Висновки*

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслеників, плакатів) *5.1 Еволюція веб-додатків до веб-сервісів та ключові архітектурні відмінності. 5.2 Архітектурні відмінності між (а) монолітним додатком із інтегрованими функціями та (б) розподіленим додатком із використанням функцій веб-сервісів. 5.3 Надсилання та отримання повідомлень веб-сервісу для побудови калькулятора підвищення ціни акцій. 5.4 Складаючи разом послуги, що виставляються багатьма корпораціями, щоб створити окрему пропозицію послуг. 5.5 Приклад веб-програми, яка спілкується з іншими. 5.6 Структура JWT. 5.7 Список ресурсів*

6. Консультанти розділів роботи (проекту)

Найменування Розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Спеціальна частина	*Проф. Іванов В.Г.*		

7. Дата видачі завдання *01.10.2020*

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи (проекту)	Термін виконання етапів проекту (роботи)	Примітка
1	*Отримання завдання на магістерську*		
	Роботу	*01.10.2020*	
2	*Аналіз завдання, літератури та аналогів за*		
	темою роботи	*16.10.2020*	
3	*Аналіз предметної області та її роль у розробці*	*18.10.2020*	
4	*Структурне проектування ПЗ.*		
	Розробка алгоритму та структури ПЗ	*25.10.2020*	
5	*Аналіз і вибір засобів створення веб-сервісів*	*01.11.2020*	
6	*Аналіз і вибір прикладу веб-сервісів*	*10.11.2020*	
7	*Запровадження веб-сервісів для обраної*	*20.11.2020*	
	системи. Порівняння результатів		
8	*Оформлення пояснювальної записки*	*03.12.2020*	
9	*Оформлення графічної частини та*		
	Матеріалів	*20.12.2020*	
10	*Представлення на рецензування*	*22.12.2020*	
11	*Подання магістерської роботи в ЕК*	*22.12.2020*	

Студент _____
(підпис)

Керівник роботи (проекту) _____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Магістерська атестаційна робота: пояснювальна записка: 65 с., 18 рис., 4 табл., 3 додатка, 17 джерел; графічний матеріал – 7 аркушів.

Об'єкт дослідження – процес розбиття монолітного додатку на сервіси.

Предмет дослідження – розбиття статичного коду на сервіси з мінімальною зв’язністю між собою, які мають можливість подальшої інтеграції в різноманітні додатки.

Мета магістерської роботи – дослідження предметної області сервісів орієнтованих на підтримку динамічного розвитку бізнесу і на основі отриманої інформації розбити статичний додаток на незалежні сервіси.

Методи дослідження: Resful архітектура створення веб-сервісів та методи об’єктно-орієнтованої розробки.

Результати: проаналізована предметна область, після чого виявлено особливості розробки сервісів; У результаті розибито прогармний додаток на сервіси, які компонуються між собою. В ході роботи був проведений аналіз предметної області, розглянуті існуючі рішення, вивчені переваги і недоліки різних архітектурних рішень. Розроблено і детально описана архітектура системи, яка відповідає заявленим функціональним і нефункціональним вимогам. Обрані і вивчені програмні засоби реалізації розробленої архітектури. І, нарешті, система розбита на нові компоненти і протестована. Крім того, програмний інтерфейс програми був забезпечений докладною документацією у вигляді інтерактивної веб-сторінки.

Область застосування – процес створення веб-додатків в IT-компаніях.

JAVA, ВЕБ-ДОДАТОК, SPRING FRAMEWORK, HIBERNATE FRAMEWORK, MONGODB, NOSQL, REST, JSON

ABSTRACT

Master's Thesis: 65 pages, 18 pictures, 4 tables, 3 applications, 17 sources; graphic part – 7 posters.

The object of study - the process of dividing the monolithic application into services.

The subject of the study is the division of static code into services with minimal interconnection, which have the ability to further integrate into various applications.

The purpose of the master's work is to study the subject area of services focused on supporting the dynamic development of business and based on the information to develop a static application for independent services.

Research methods: Resful architecture of creating web services and methods of object-oriented development.

Results: the subject area is analyzed, after which the peculiarities of service development are revealed; As a result, the program application for services that are composed among themselves is broken. In the course of the work the analysis of the subject area was carried out, the existing decisions were considered, the advantages and disadvantages of various architectural decisions were studied. The architecture of the system, which meets the stated functional and non-functional requirements, has been developed and described in detail. Selected and studied software tools for implementing the developed architecture. Finally, the system is broken down into new components and tested. In addition, the program interface of the program was provided with detailed documentation in the form of an interactive web page.

Scope - the process of creating web applications in IT companies.

JAVA, WEB APPLICATION, SPRING FRAMEWORK, HIBERNATE FRAMEWORK, MONGODB, NOSQL, REST, JSON

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

БД – база даних;

СУБД – система управління базами даних;

HTML – HyperText Markup Language (мова гіпертекстової розмітки);

HTTP – HyperText Transfer Prorocol (протокол передачі гіпертекстових файлів);

JSP – Java Server Pages (розширення технології сервлетів для спрощення роботи з web-вмістом);

JSTL – Java Standard Tag Library (бібліотеки стандартних тегів на мові Java);

MS – Microsoft;

MVC – Model-View-Controller (архітектурний шаблон підходу до створення об'єктно-орієнтованих web-додатків);

SQL – Structured Query Language (мова структурованих запитів);

URL – Uniform Resource Locator (уніфікований покажчик інформаційного ресурсу);

XML – eXtensible Markup Language (розширена мова розмітки документів);

REST – Representational State Transfer (передача репрезентативного стану)

ВСТУП

На сьогоднішній день ніхто не сумнівається в доцільності винесення бізнес-функцій розроблюваних програмних продуктів в окремі сервіси. Метою будь-якого проекту є забезпечення якості та швидкості виконання продукту, що розробляється. Використання готового функціоналу підвищує ефективність, терміни виконання, дозволяє створювати гнучкі додатки, які можна розбивати по модулям, а, отже, покращує якість створюваного програмного забезпечення.

Мережа Інтернет стала загальновизнаним фактором ділового та суспільного життя. Широка поширеність і велика пропускна здатність створюють умови, при яких вигідно вирішувати багато завдань за допомогою інтернет-технологій. Однак Інтернет об'єднує в собі багато різних платформ, а інформація міститься в різноманітних джерелах даних. Тому актуальна проблема зв'язку таких різнорідних даних, а також створення способу, який дозволяє отримувати їх у вигляді зручному для подальшої обробки.

Концепція сервісів (Services) покликана вирішити цю задачу об'єднання, інтеграції різнорідних систем на основі відкритих стандартів.

На сьогоднішній день величезна кількість людей користується послугами сервісів.

Відомо, що сучасним програмним продуктам властива висока складність: велика кількість компонентів і функцій, складні взаємозв'язки. Одним з рішень проблеми обмежених ресурсів і необмежених завдань як раз і є використання сервісів. Без інтеграції готових програмних модулей в розроблювану систему - ресурси на створення та підтримання продукту будуть значно більші. Тому використання сервісів - невід'ємна частина розробки великих та серйозних проектів.

У даній роботі досліджується процес підтримки якості програмного продукту за допомогою використання сервісів; класифікація сервісної архітектури і напрямок використання. Далі наводиться опис впровадження даного напрямку по

відношенню до веб-додатку. Аналізується доцільність використання сервісів в період розбиття програмного додатку на модулі.

У відповідність з метою були поставлені і реалізовані наступні завдання:

- дослідити особливості використання сервісів;
- узагальнити і систематизувати рівні і типи сервісів;
- проаналізувати ефективність використання сервісів в цілому;
- проаналізувати доцільність застосування веб-сервісів на існуючому додатку і інструменти для їх реалізації;
- реалізувати сценарії використання обраного функціоналу веб-додатку.

Метою роботи є дослідження предметної області сервісів орієнтованих на підтримку динамічного розвитку бізнесу і на основі отриманої інформації розробити статичний додаток на незалежні сервіси, які можна використовувати в будь-яких комбінаціях в середині програми та зовнішніми викликами.

1 ОГЛЯД ТА АНАЛІЗ РОЗРОБКИ СЕРВІСІВ ЯК ПРОЦЕСУ ПРИСКОРЕННЯ І ВДОСКОНАЛЕННЯ СТВОРЕННЯ ДОДАТКІВ

1.1 Актуальність даної теми

Веб-сервіси представляють нову парадигму в архітектурі та розробці програм. Важливість веб-служб полягає не в тому, що вони нові, а в тому, що ця нова технологія відповідає потребам розробки додатків. Щоб зрозуміти цю парадигму, давайте спочатку розглянемо парадигму додатків, яка передувала веб-програмам веб-служб.

Веб-програми - це програми, доступні через Інтернет і дозволяють будь-якому користувачеві в будь-якій точці світу отримати доступ до своїх можливостей. Це на відміну від старих клієнт-серверних додатків, в яких лише спеціалізовані клієнти могли отримати доступ до програм, що перебувають на сервері. Веб-додатки розширили базу користувачів з кількох сотень клієнтських машин, що здійснюють доступ до клієнт-серверної програми, до мільйонів користувачів по всій Мережі, що отримують доступ до веб-програми [1].

Веб відкрив шлюзи для веб-програм, дозволивши користувачам просто вказати URL-адресу у веб-браузері. Веб-додатки також збільшують труднощі розробки додатків, оскільки клієнт додатків (браузер ПК) не знає вимог до зв'язку програми та основних систем. Для подолання цього розриву між клієнтами веб-додатків та самими веб-додатками використовувались галузеві стандартні технології, такі як HTTP та HTML. Сервери додатків та інше проміжне програмне забезпечення з’явилися, щоб зменшити складність побудови веб-програм, дозволяючи при цьому всебічний доступ до кожної веб-програми.

Веб-служби спираються на модель веб-додатків та розширюють її. Веб-програми дозволяють будь-якому веб-браузеру отримувати доступ до своїх

функціональних можливостей за допомогою інтерфейсу користувача програми, представленого через браузер. Веб-служби роблять цей крок далі і дозволяють будь-яким клієнтським програмам отримувати доступ та використовувати свої можливості.

Веб-програма надає універсальний доступ користувачів до своїх можливостей, підтримуючи стандартні інтерфейси користувача до свого інтерфейсу. Вони не дозволяють розширювати чи додавати свої можливості за допомогою програмного доступу. Щоб використати функціональність веб-програми та розробити її, слід використовувати складні та часто ненадійні методи, такі як вишкрібання екрану. Веб-служби вирішують цю проблему, дозволяючи програмний доступ до можливостей веб-служб, використовуючи стандартні галузеві інтерфейси та протоколи. Еволюція веб-додатків до веб-служб показана на рисунку 1.1.

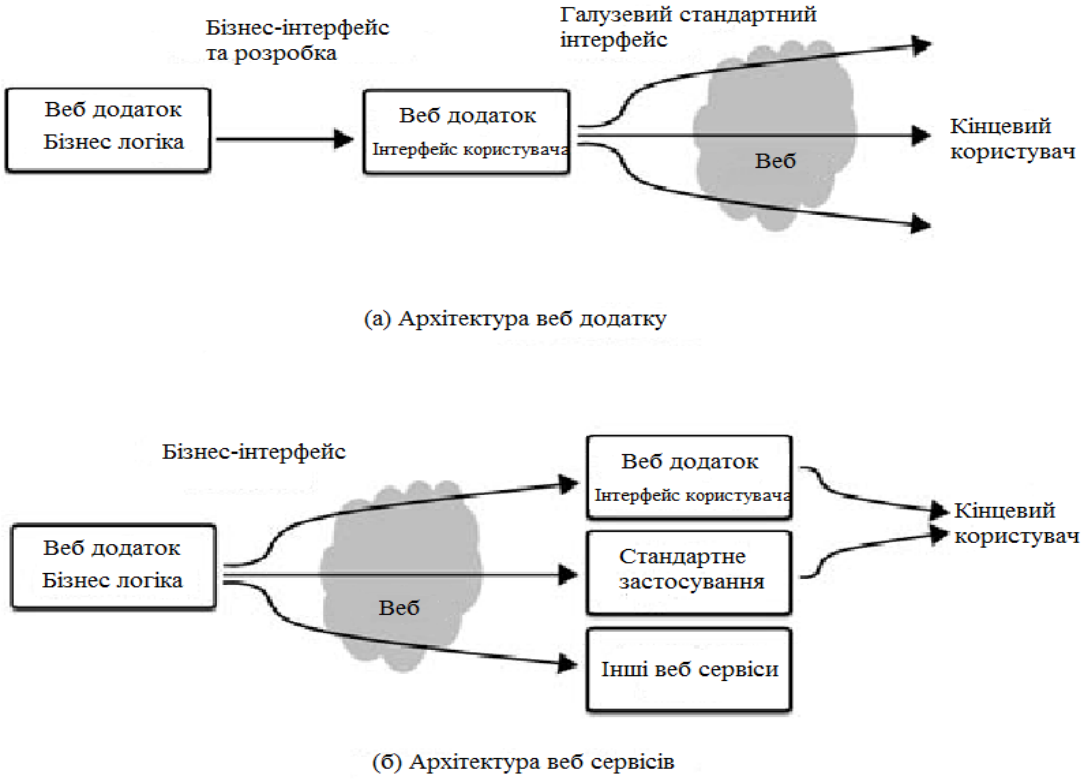


Рисунок 1.1 – Еволюція веб-додатків до веб-сервісів та ключові архітектурні відмінності

Веб-сервіси виступають за архітектуру, орієнтовану для програм, в яких програмний компонент забезпечує свою функціональність як сервіс, який можуть використовувати інші програмні компоненти [2]. Така модель служби абстрагує багато складних проблем, що виникають внаслідок інтеграції компонентів програмного забезпечення, включаючи сумісність платформи, тестування та обслуговування.

Оскільки клієнти веб-сервісу не мають інформації, [3] необхідної для спілкування з веб-службою, необхідний набір стандартів, щоб дозволити комунікацію багато до багатьох. Стандарти веб-сервісу базуються на попередніх стандартах комунікацій та представлення даних, таких як HTTP та HTML.

Ключовим фактором, що сприяє веб-службам, є XML. Незважаючи на те, що HTML та XML схожі за тим, що обидві є зручними для читання мовами розмітки, HTML призначений для розмітки презентацій, тоді як XML - для семантичної розмітки. Цей критичний атрибут XML підтримує вираження прикладної та функціональної семантики незалежно від платформи, що дозволяє обмінюватися інформацією від будь-якого до будь-якого.

Деякі стверджують, що веб-служби не є нічим новим; вони є просто останнім втіленням розподілених обчислень. У якомусь сенсі це може бути правдою, але щось у веб-сервісах викликає неймовірний галас. Давайте розбиратися далі.

1.2 Огляд та аналіз сучасного стану проблеми

На перший погляд, веб-послуги представляються ризикованою пропозицією для підприємств. Чому IT-організації, які вимагали повного контролю над усіма аспектами корпоративних додатків, прийматимуть розподілену та спільну архітектуру програмного забезпечення, яка переміщує адміністративний контроль над різними частинами програм поза межами корпоративного брандмауера? Характеристики виконання додатків, що базуються на веб-послугах, матимуть

критичну залежність від віддаленого розміщення та дистанційного управління зовнішніми компаніями. Це суттєвий відхід від централізованого управління, а також гарантованої передбачуваності та надійності, які стали ознаками корпоративного програмного забезпечення та ІТ-організацій, які ними керують.

Причини такої перерви зрозумілі. Веб-сервіси забезпечують потік даних та бізнес-процесів між діловими партнерами між підприємствами, а також між кількома організаціями або групами на підприємстві до такої міри, яка раніше була неможливою. Компанії, які раніше не могли спілкуватися, та програми, які раніше не могли взаємодіяти, тепер можуть це зробити.

Веб-сервіси дозволяють компаніям стимулювати зростання, об'єднуючи різні послуги та запроваджуючи нові послуги, що приносять прибуток. У той же час веб-служби спрощують інтеграцію, скорочуючи час виходу на ринок та витрати, а також підтримують операційну ефективність, яка впорядковує суть.

Потенційні переваги веб-сервісів величезні. Ризики однаково великі, якщо не більші. Підприємницькі ІТ-організації опиняться посередині, відповідальними за узгодження переваг та ризиків використання веб-сервісів на підприємстві.

ІТ-організації, намагаючись закріпити позицію контролю над ризикованим та потенційно шкідливим трафіком веб-служб, наполягатимуть на контролі над тим, які програми веб-служб взаємодіють між собою. Неправильна веб-служба просто буде відключена від взаємодії з будь-якими корпоративними програмами; такі відсікання можуть бути навіть превентивними, якщо є історія проблем або сприйняття загрози [4].

Для цього ІТ відіграє більш стратегічну роль в організаціях і тісніше узгоджується з окремими бізнес-підрозділами. Критичні рішення ділових підрозділів, таких як партнери, від яких надходитимуть компоненти, повинні бути очищені ІТ, якщо веб-служби цих партнерів будуть взаємодіяти з програмами або веб-службами компанії.

Це матиме серйозні наслідки для архітектур корпоративних програм. Підприємницькі додатки підтримуватимуть динамічні та замінні веб-сервіси. Більше того, ІТ використовуватимуть середовища управління для розгортання

загальнокорпоративних політик щодо веб-служб, які відстежуватимуть і суворо застосовуватимуть веб-служби, якими можуть користуватися додатки.

Немає сумнівів, що використання веб-служб на підприємстві потребуватиме змін. Багато з цих змін стосуватимуться встановлених процедур та існуючої політики, що підтверджується багаторічним досвідом та мільярдами доларів. Тим не менше, потенційні вигоди як фінансові, так і стратегічні для впровадження веб-служб є достатньо великими, щоб виправдати такі зміни.

1.3 Мета створення сервісів

Веб-сервіси насправді є технологією розподілених обчислень, і існує одна критична відмінність між веб-службами та розподіленими обчислювальними технологіями, які існували раніше. Людина, яка впроваджує веб-службу, може бути майже на сто відсотків впевненою, що хтось інший може спілкуватися з нею та користуватися нею. Прорив веб-служб - це саме комунікації, які вони дозволяють. Рівень довіри, який породжують веб-служби у своїх розробників, подібний до рівня веб-сторінок HTML. Розробник HTML-сторінки впевнений, що будь-хто з браузером може переглядати веб-сторінку.

Веб-послуги виросли з-за потреби в розподіленому обчислювальному середовищі додатків, яке не було так складно розгорнути, як загальна архітектура посередницьких запитів об'єктів (CORBA) або об'єктна модель розподілених компонент Microsoft (DCOM), а також пропонували більшу сумісність. І CORBA, і DCOM мали на меті забезпечити розподілене обчислювальне середовище в різнорідних середовищах. На жаль, не підтримуються середовища та технології, які були б настільки далекосяжними, щоб забезпечити неоднорідні комунікації в масштабі будь-кого.

У певному сенсі веб-сервіси жертвують багатством можливостей, що надаються попередніми розподіленими обчислювальними середовищами, які

необхідні невеликій групі всіх додатків, для набагато більш простого та повсюдного рішення, яке застосовується для переважної більшості додатків. Це не означає, що веб-служби обмежують їх використання. Додаткові можливості можна розкласти поверх платформи веб-служб для задоволення різних потреб.

Додатки, які є веб-службами, мають велику базу інших програм (які також є веб-службами), з якими можна спілкуватися. Оскільки вони базуються на простих і відкритих галузевих стандартах (або фактичних стандартах), веб-служби роблять значний прорив у напрямку повсюдної взаємодії. Сумісність тут має масштаби Інтернету чи Інтернету, а не лише групи чи організації.

Засновані на галузевих стандартах та підтримуючи взаємодію ні з ким, веб-сервіси готові стати платформою, яка задовольняє потреби електронного бізнесу. Всі компанії взаємодіють з іншими компаніями в процесі ведення свого бізнесу. Виробничі компанії взаємодіють з постачальниками комплектуючих, дистриб'ютори - з виробничими компаніями, роздрібні торговці - з дистриб'юторами тощо. Спочатку ці взаємодії проводились вручну, проводились поштою, телефоном та факсом.

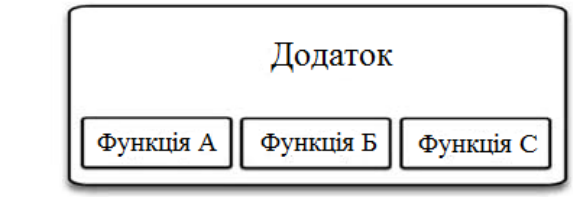
Веб-програми дозволяли компаніям взаємодіяти між собою, виставляючи деякі свої можливості та бізнес-процеси іншим в Інтернеті. Але, здебільшого, це все одно вимагало взаємодії людини з веб-додатком з іншого боку. Веб-служби усувають необхідність постійного втручання людини, поки компанії взаємодіють, дозволяючи програмні розмови між програмами [5].

Усуваючи цей бар'єр для взаємодії електронного бізнесу, веб-служби забезпечують нові ділові відносини, а також більш плавні відносини, які можна налаштувати та переналаштувати на льоту. Незважаючи на те, що веб-послуги пропонують численні переваги, вони також представляють багато викликів та ризиків у традиційному корпоративному середовищі. Далі ми обговорюємо веб-сервіси та те, як вони відповідають підприємствам.

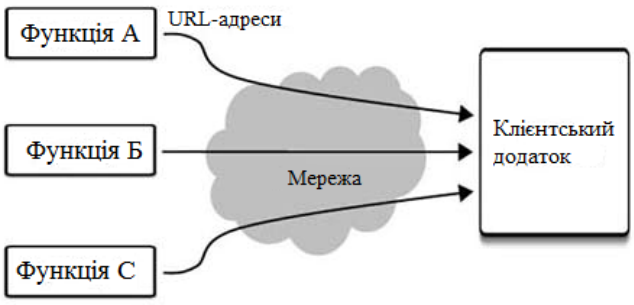
1.4 Роль сервісів в процесі розробки ПЗ

Сервіси грають значну роль під час розробки додатків. Навіщо ще раз винаходити велосипед, якщо вже існують готові проектні рішення, які можна перевикористовувати.

Веб-служби реалізують можливості, доступні для інших програм (або навіть інших веб-служб) через стандартні мережеві інтерфейси та інтерфейси додатків та протоколи. Додаток може використовувати можливості веб-служби, просто викликаючи її через мережу, не потребуючи її інтеграції. Таким чином, веб-служби представляють собою багаторазові будівельні блоки програмного забезпечення, які мають URL-адресу. Архітектурні відмінності між монолітними, інтегрованими програмами та програмами, що базуються на веб-послугах, зображені на рисунку 1.2.



(а) Монолітний додаток с інтегрованими функціями



(б) Клієнтський додаток, що викликає віддалені функції веб-служб

Рисунок 1.2 – Архітектурні відмінності між (а) монолітним додатком із інтегрованими функціями та (б) розподіленим додатком із використанням функцій веб-сервісів.

Деякі з цих можливостей важко або непрактично інтегрувати в сторонні програми. Коли ці можливості виставляються як веб-сервіси, їх можна вільно поєднати, тим самим досягаючи переваг інтеграції, не створюючи при цьому труднощів.

Веб-служби надають свої можливості клієнтським програмам, а не їх реалізації. Це дозволяє реалізовувати веб-сервіси будь-якою мовою та на будь-якій платформі і при цьому бути сумісними з усіма клієнтськими програмами.

Кожен будівельний блок (веб-служба) є автономним. Він описує власні можливості, публікує власний програмний інтерфейс та реалізує власні функціональні можливості, доступні як розміщена послуга. Бізнес-логіка веб-служби працює на віддаленій машині, яка доступна іншим програмам через мережу. Клієнтська програма просто викликає функціональність веб-служби, надсилаючи їй повідомлення, отримує відповідні повідомлення від веб-служби, а потім використовує результати в додатку. Оскільки немає необхідності інтегрувати веб-службу всередині клієнтської програми в єдиний монолітний блок, тим самим скорочуються терміни розробки та тестування, витрати на обслуговування та загальні помилки.

Припустимо, ви хочете створити простий додаток-калькулятор, який визначає подорожчання ціни акцій будь-якої компанії, враховуючи її корпоративне найменування та дату придбання акцій. Заявка повинна робити наступне[6]:

- значте символ біржової біржі компанії на основі назви компанії.
- значте останню ціну акції, виходячи з символу білета.
- значте історичну ціну акції на дану дату, виходячи із символу білета.
- числіть різницю між двома цінами на акції та презентуйте її користувачеві.

Ця, здавалося б, тривіальна програма насправді надзвичайно складна. З самого початку є проблеми. Нам потрібно створити базу даних з назвами всіх компаній у країні та відповідним символом біржового котирування. Що ще важливіше, ми повинні підтримувати цю базу даних, оскільки компанії перелічуються, перераховуються, змінюються їхні назви або символи, або об’єднуються. Щоб отримати доступ до ціни акцій у реальному часі, ми повинні

мати стосунки з фінансовою або брокерською фірмою. Юридичні складності та суперечки в організації таких відносин досить погані, не кажучи вже про IT-інфраструктуру, яка також повинна бути створена.

Якщо ви не працюєте в брокерській компанії або не займаєтесь підтримкою інформації про запаси, час і витрати, необхідні для створення інфраструктури, необхідної для підтримки калькулятора оцінки запасів, величезні і, в більшості випадків, непомірні. Поки брокерська фірма сама не вирішила надати такий калькулятор, клієнтам доведеться обійтися без нього.

Веб-служби спрощують і багато в чому усувають необхідність будувати для себе інфраструктуру підтримки як юридичну, так і технічну. Калькулятор можна розробити, просто передаючи повідомлення між програмою калькулятора та відповідним набором веб-служб. На рисунку 1.3 графічно зображений потік повідомлень та фундаментальна архітектура калькулятора оцінки ціни акцій на основі веб-служб.

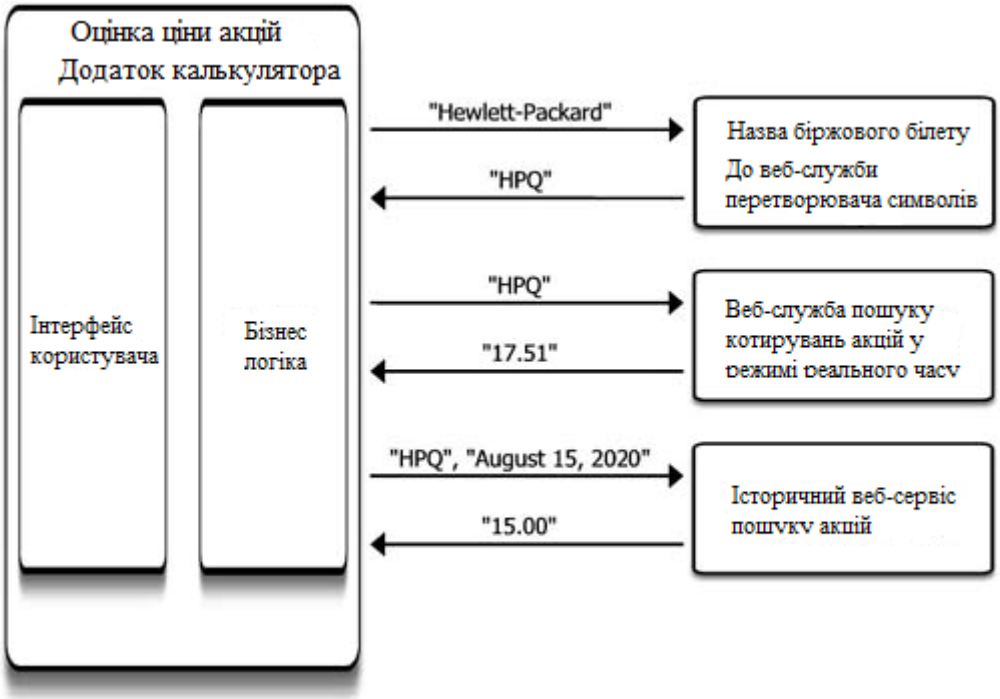


Рисунок 1.3 – Надсилання та отримання повідомлень веб-сервісу для побудови калькулятора підвищення ціни акцій

Повідомлення надсилаються між програмою калькулятора та такими трьома веб-службами:

- Назва біржового білета в конвертер символів, сервіс який приймає назву компанії та надає символ акції.
- Пошук котирувань акцій у режимі реального часу, сервіс який надає останню ціну акції на основі символу стрічки.
- Історичний веб-сервіс пошуку акцій, сервіс який надає історичну ціну акції на основі символу та бажаної дати.

Оскільки кожна з цих трьох веб-служб надається, розміщується та управляється іншою компанією, розробник додатку калькулятор повинен зосередитись лише на своєму ключовому розумінні або внеску. Складні, специфічні для домену проблеми, такі як той факт, що символом акції Hewlett-Packard був HWP і лише нещодавно став HPQ, вирішуються (або повинні вирішуватися) безпосередньо веб-службами. Використовуючи ці три веб-сервіси, програма може легко визначити підвищення цін акцій Hewlett-Packard з 15 серпня 2020 р. До 17,51 - 15,00 дол. США = 2,51 дол. США. На основі даних веб-сервісів програма калькулятора може забезпечити подальший аналіз, наприклад, підвищення відсоткового співвідношення, та подати всю інформацію в зрозумілій графічній формі.

Припускаючи, що необхідні можливості існують і доступні як веб-сервіси, розробники можуть зосередитись на своїй унікальній частині з доданою вартістю та використовувати сторонні веб-служби для решти функціональних можливостей. Переваги використання веб-сервісів очевидні:

- Різко зменшують витрати на розробку додатків, зосередившись на власному внеску на додану вартість та використовуючи сторонні веб-служби для всього іншого.
- Інтегрують дані та бізнес-процеси з учасниками ринку та діловими партнерами, які мають знання або можливості в області.
- Зменшують або усувають багато помилок, що виникають із складних та великих монолітних додатків.

- Спрощують обслуговування та налаштування додатків, сегментуючи додаток у клієнтському додатку та кожному з його споживаних веб-служб.
- Значно скорочують час виходу на ринок.

По мірі того, як ми продовжуємо ідею, і все більше і більше компаній виявляють деякі свої внутрішні можливості у якості веб-служб, справжня цінність веб-послуг полягає у складі набору веб-служб. Розглянемо наступні дві компанії. Одна - компанія з обслуговування дорожнього руху, яка контролює автомобільний рух на основних дорогах та магістралях та передбачає очікуваний час поїздки. Друга - це компанія, що займається послугами бронювання таксі, яка дозволяє клієнтам забронювати таксі для вивезення у визначеному місці та часом. Кожна з цих компаній та їх продукція переконливі самі по собі. Однак, якщо ці компанії виставлять свої можливості веб-сервісів, ці послуги можуть бути об'єднані в єдину, більш привабливу та корисну послугу як однією з цих двох компаній, так і третьою компанією.

Як приклад можна взяти таксі до аеропорту, перш ніж ловити рейс на зустріч. Використовуючи можливості обох компаній за допомогою відповідних веб-служб, мандрівник може зарезервувати таксі і бути впевненим, що якщо аварія чи інші умови дорожнього руху спричинять несподіване збільшення її часу в дорозі, бронювання таксі може бути проведено та надіслано попередження мандрівник, що повідомляє їй про оновлений розклад таксі, а також про дорожню ситуацію, яка спричинила зміни. Просто та розумно поєднуючи окремі послуги двох компаній, ми можемо створити більш привабливу та корисну послугу для мандрівників. Склад веб-сервісів різних підприємств зображений на малюнку 1.4. Технологіями, що складають основу веб-сервісів, є SOAP, WSDL та UDDI.

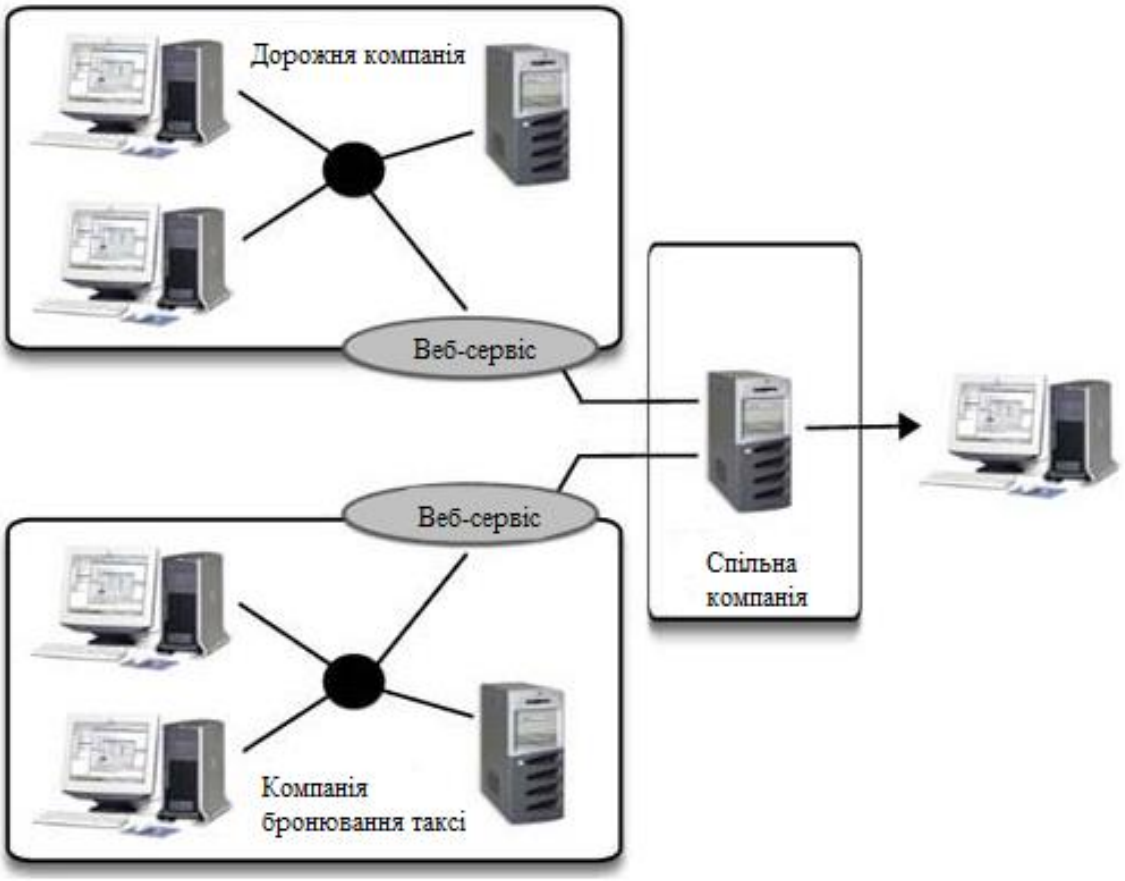


Рисунок 1.4 – Складаючи разом послуги, що виставляються багатьма корпораціями, щоб створити окрему пропозицію послуг

1.5 Огляд існуючих сервісних архітектур

На сьогоднішній день найбільшого поширення набули такі протоколи реалізації веб-сервісів[7]:

- SOAP (Simple Object Access Protocol) - по суті це трійка стандартів SOAP / WSDL / UDDI
- REST (Representational State Transfer)
- XML-RPC (XML Remote Procedure Call)

Насправді, SOAP стався від XML-RPC і є наступним етапом його розвитку. У той час як REST - це концепція, в основі якої лежить швидше архітектурний стиль, ніж нова технологія, що базується на теорії маніпуляції об'єктами CRUD (Create Read Update Delete) в контексті концепцій WWW. Нижче наведено основні сервісні архітектури.

1.5.1 Загальна архітектура брокера об'єктних запитів (CORBA)

У 1980-х почалося активне використання корпоративних мереж і клієнт-серверної архітектури. Виникла потреба в стандартному способі взаємодії додатків, які створені з використанням різних технологій, виконуються на різних комп'ютерах і під різними ОС. Для цього була розроблена CORBA. Це один зі стандартів розподілених обчислень, що зародився в 1980-х і розквітлий до 1991 року.

Стандарт CORBA був реалізований декількома вендорами. Він забезпечує:

- не залежні від платформи виклики віддалених процедур (Remote Procedure Call);
- транзакції (в тому числі віддалені);
- безпека;
- події;
- незалежність від вибору мови програмування;
- незалежність від вибору ОС;
- незалежність від вибору обладнання;
- незалежність від особливостей передачі данних та звя’яку;
- набір данних через мову опису інтерфейсів (Interface Definition Language, IDL);

Принцип роботи заключается в тому, що спочатку нам потрібно отримати брокер об'єктних запитів (Object Request Broker, ORB), який відповідає специфікації

CORBA. Він надається вендором і використовує мовні перетворювачі (language mappers) для генерування «заглушок» (stub) і «скелетів» (skeleton) на мовах клієнтського коду. За допомогою цього ORB і визначень інтерфейсів, які використовують IDL (аналог WSDL), можна на основі реальних класів генерувати в клієнті віддалено викликані класи-заглушки (stub classes). А на сервері можна генерувати класи-скелети (skeleton classes), обробні вхідні запити і викликають реальні цільові об'єкти[8].

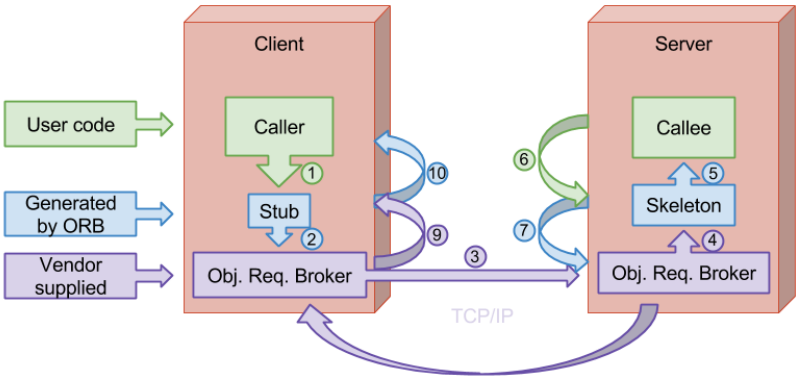


Рисунок 1.5 – Принцип роботи COBRA

Викликаюча програма (caller) викликає локальну процедуру, реалізовану заглушкою.

- заглушка перевіряє виклик, створює повідомлення-запит і передає його в ORB;
- клієнтський ORB шле повідомлення по мережі на сервер і блокує поточний потік виконання;
- серверний ORB отримує повідомлення-запит і створює екземпляр скелета;
- скелет виконує процедуру в викликається об'єкті;
- викликаний об'єкт проводить обчислення і повертає результат;
- скелет пакує вихідні аргументи на повідомлення-відповідь і передає його в ORB;
- ORB шле повідомлення по мережі клієнтові;
- клієнтський ORB отримує повідомлення, розпаковує і передає інформацію заглушці;

- заглушка передає вихідні аргументи зухвалому методу, розблокує потік виконання, і зухвала програма продовжує свою роботу.

Переваги архітектури:

- незалежність від обраних технологій (не рахуючи реалізації ORB);
- незалежність від особливостей передачі даних / зв'язку;

Недоліки:

- Незалежність від місця розташування: клієнтський код не має поняття, чи є виклик локальним або віддаленим. Звучить непогано, але тривалість затримки та види збоїв можуть сильно варіюватися. Якщо ми не знаємо, який у нас виклик, то програма не може вибрати відповідну стратегію обробки викликів методів, а значить, і генерувати вилучені виклики всередині циклу. В результаті вся система працює повільніше;
- складна, роздута і неоднозначна специфікація: її зібрали з декількох версій специфікацій різних вендорів, тому (на той момент) вона була роздутою, неоднозначною і важкою в реалізації;
- заблоковані канали зв'язку (communication pipes): використовуються специфічні протоколи поверх TCP / IP, а також специфічні порти (або навіть випадкові порти). Але правила корпоративної безпеки та файрволи часто допускають HTTP-з'єднання тільки через 80-й порт, блокуючи обміни даними CORBA;

1.5.2 Веб-сервіси

Хоча сьогодні можна знайти застосування для CORBA, але ми знаємо, що потрібно було зменшити кількість вилучених звернень, щоб підвищити продуктивність системи. Також був потрібний надійний канал зв'язку і простіша специфікація обміну повідомленнями. І для вирішення цих завдань в кінці 1990-х почали з'являтися веб-сервіси.

Це мав бути надійний канал зв'язку, тому: HTTP став за замовчуванням працювати через порт 80. Для обміну повідомленнями почали використовувати платформо-незалежну мову (на зразок XML або JSON). Потрібно було зменшити кількість вилучених звернень, тому: віддалені з'єднання стали явними, так що тепер ми завжди знаємо, коли виконується віддалений виклик. Замість численних віддалених викликів об'єктів ми звертаємося до віддалених сервісів, але набагато рідше. Потрібно було спростити специфікацію обміну повідомленнями, тому: перша чернетка SOAP з'явилася в 1998-му, стала рекомендацією W3C у 2003-му, після чого перетворився в стандарт. SOAP увібрав в себе деякі ідеї CORBA, на кшталт шару для обробки обміну повідомленнями та «документа», що визначає інтерфейс за допомогою мови опису веб-сервісів (Web Services Description Language, WSDL)[9].

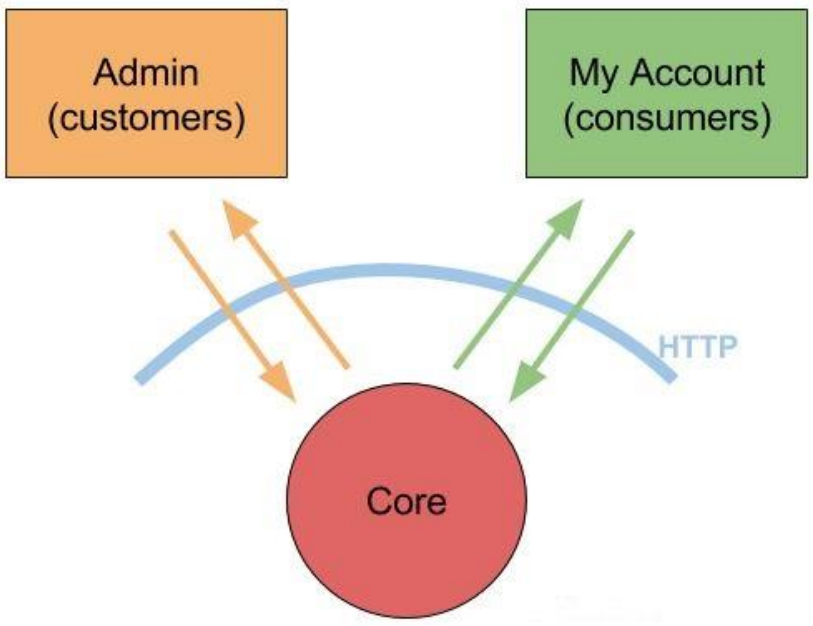


Рисунок 1.6 – Принцип роботи веб-сервісів

Завдяки мікросервісам ми перейшли в парадигмі SOA від віддаленого виклику методів об'єкта (CORBA) до передачі повідомлень між сервісами.

Але потрібно розуміти, що в рамках SOA веб-сервіси - не просто API загального призначення, всього лише надають CRUD-доступ до бази даних через

HTTP. У якихось випадках ця реалізація може бути корисною, але заради цілісності ваших даних необхідно, щоб користувачі розуміли лежить в основі реалізації модель і дотримувалися бізнес-правила. SOA має на увазі, що веб-сервіси є обмеженими контекстами бізнес-субдоменів (business sub-domain) і відокремлює реалізацію від розв'язуваних веб-сервісами завдань.

Переваги:

- незалежність набору технологій, розгортання і масштабованості сервісів;
- стандартний, простий і надійний канал зв'язку (передача тексту по HTTP через порт 80);
- оптимізований обмін повідомленнями;
- стабільна специфікація обміну повідомленнями;
- ізольованість контекстів доменів (Domain contexts);

Недоліки:

- різні веб-сервіси важко інтегрувати через відмінності в мовах передачі повідомлень. Наприклад, два веб-сервісу, що використовують різні JSON-представлення однієї і тієї ж концепції;
- синхронний обмін повідомленнями може перевантажити системи;

1.5.3 Черга повідомлень

У нас є кілька додатків, які асинхронно спілкуються один з одним за допомогою платформо-незалежних повідомлень. Черга повідомлень покращує масштабованість і підсилює ізольованість додатків. Їм не потрібно знати, де знаходяться інші додатки, скільки їх і навіть що вони собою представляють. Однак всі ці додатки повинні використовувати одну мову обміну повідомленнями, тобто заздалегідь певний текстовий формат представлення даних[10].

Черга повідомлень використовує в якості компонента інфраструктури програмний брокер повідомлень (RabbitMQ, Beanstalkd, Kafka і т. Д.). Для реалізації зв'язку між додатками можна по-різному налаштувати чергу:

- Запит / Відповідь. Клієнт шле в чергу повідомлення, включаючи посилання на «розмову» («conversation» reference). Повідомлення приходить на спеціальний вузол, який відповідає відправнику іншим повідомленням, де міститься посилання на той же розмову, так що одержувач знає, на яку розмову посилається повідомлення, і може продовжувати діяти. Це дуже корисно для бізнес-процесів середньої і великої тривалості (ланцюжків подій, sagas).
- Публікація / Підписка за списками. Черга підтримує списки опублікованих тим підписок (topics) і їх передплатників. Коли черга отримує повідомлення для якоїсь теми, то поміщає його у відповідний список. Повідомлення зіставляється з темою за типом повідомлення або по заздалегідь визначеному набору критеріїв, включаючи і зміст повідомлення.
- Публікація / Підписка на основі мовлення. Коли черга отримує повідомлення, вона транслює його всім вузлам, прослуховуючих чергу. Вузли повинні самі фільтрувати дані і обробляти тільки цікавлять повідомлення.

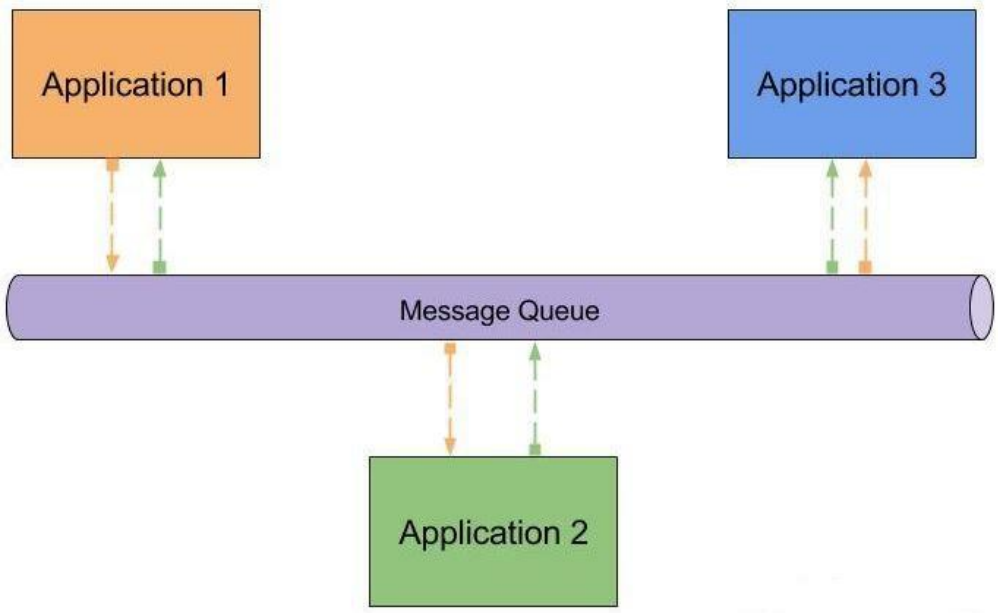


Рисунок 1.7 – Принцип роботи черги повідомлень

Переваги:

- незалежність набору технологій, розгортання і масштабованості сервісів;
- стандартний, простий і надійний канал зв'язку (передача тексту по HTTP через порт 80);
- оптимізований обмін повідомленнями;
- стабільна специфікація обміну повідомленнями;
- ізольованість контекстів домену (Domain contexts);
- простота підключення і відключення сервісів;
- асинхронність обміну повідомленнями допомагає керувати навантаженням на систему;

Недоліки:

- різні веб-сервіси важко інтегрувати через відмінності в мовах передачі повідомлень. Наприклад, два веб-сервісу, що використовують різні JSON-представлення однієї і тієї ж концепції;

1.5.4 Сервісна шина підприємства (ESB)

Сервісна шина підприємства використовувала веб-сервіси вже в 1990-х, коли вони тільки розвивалися (можливо, деякі реалізації спочатку використовували CORBA).

ESB виникла за часів, коли в компаніях були окремі додатки. Наприклад, одне для роботи з фінансами, інше для обліку персоналу, третє для управління складом, і т. д., і їх потрібно було якось пов'язувати між собою, якось інтегрувати. Але всі ці додатки створювалися без урахування інтеграції, не було стандартної мови для взаємодії додатків (як і сьогодні). Тому розробники додатків передбачали кінцеві точки для відправки і прийому даних в певному форматі. Компанії-клієнти потім інтегрували додатки, налагоджуючи між ними канали зв'язку і перетворюючи повідомлення з однієї мови додатка в інший[11].

Черга повідомлень може спростити взаємодію додатків, але вона не здатна вирішити проблему різних форматів мов. Втім, була зроблена спроба перетворити чергу повідомлень з простого каналу зв'язку в посередника, який доставляє повідомлення і перетворює їх в потрібні формати / мови. ESB став наступною сходинкою в природної еволюції простої черги повідомлень.

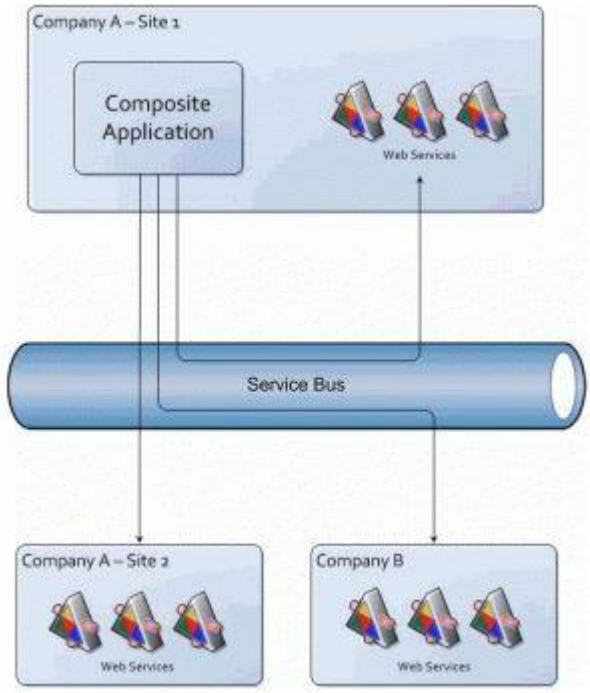


Рисунок 1.8 – Схема сервісної шини підприємства

У цій архітектурі використовується модульний додаток (composite application), зазвичай орієнтоване на користувачів, яке спілкується з веб-сервісами для виконання якихось операцій. У свою чергу, ці веб-сервіси теж можуть спілкуватися з іншими веб-сервісами, згодом повертаючи з додатком якісь дані. Але ні додаток, ні бекенд-сервіси нічого один про одного не знають, включаючи розташування і протоколи зв'язку. Вони знають лише, з яким сервісом хочуть зв'язатися і де знаходиться сервісна шина.

Клієнт (сервіс або модульне додаток) відправляє запит на сервісну шину, яка перетворює повідомлення в формат, підтримуваний в точці призначення, і перенаправляє туди запит. Вся взаємодія йде через сервісну шину, так що якщо вона падає, то з нею падають і всі інші системи. Тобто ESB - ключовий посередник, дуже складний компонент системи.

Це дуже спрощений опис архітектури ESB. Більш того, хоча ESB є головним компонентом архітектури, в системі можуть використовуватися і інші компоненти на зразок доменних брокерів (Domain Broker), сервісів даних (Data Service), сервісів

процессной оркестровки (Process Orchestration Service) і обробників правил (Rules Engine). Той же патерн може використовувати інтегрована архітектура (federated design): система розділена на бізнес-домени зі своїми ESB, і все ESB з'єднані один з одним. У такої схеми вище продуктивність і немає єдиної точки відмови: якщо якась ESB впаде, то постраждає лише її бізнес-домен[12].

Переваги:

- незалежність набору технологій, розгортання і масштабованості сервісів;
- стандартний, простий і надійний канал зв'язку (передача тексту по HTTP через порт 80);
- оптимізований обмін повідомленнями;
- стабільна специфікація обміну повідомленнями;
- ізольованість контекстів домену (Domain contexts);
- простота підключення і відключення сервісів;
- асинхронність обміну повідомленнями допомагає керувати навантаженням на систему;
- єдина точка для управління версіонірованія і перетворенням;

Недоліки:

- нижче швидкість зв'язку, особливо між уже сумісними сервісами;
- централізована логіка;
- єдина точка відмови, здатна обрушити системи зв'язку всієї компанії;
- велика складність конфігурації і підтримки;
- згодом можна прийти до зберігання в ESB бізнес-правил;
- шина так складна, що для її управління вам потрібно ціла команда;
- висока залежність сервісів від ESB;

1.5.5 Мікросервіси

В основі мікросервісной архітектури лежать концепції SOA. Призначення у неї той же, що і у ESB: створити єдиний загальний корпоративний додаток з декількох спеціалізованих додатків бізнес-доменів.

Головна відмінність мікросервісов і шини в тому, що ESB була створена в контексті інтеграції окремих додатків, щоб вийшов єдиний корпоративний розподілений додаток. А мікросервісна архітектура створювалася в контексті швидко і постійно мінливих бізнесів, які (в основному) з нуля створюють власні хмарні додатки.

Тобто у випадку з ESB у нас вже були додатки, які нам не «належать», і тому ми не могли їх змінити. А у випадку з мікросервісами ми повністю контролюємо додатки (при цьому в системі можуть використовуватися і сторонні веб-сервіси).

Характер побудови / проектування мікросервісів не вимагає глибокої інтеграції. Мікросервіси повинні відповідати бізнес-концепції, обмеженому контексту. Вони повинні зберігати свій стан, бути незалежними від інших мікросервісів, і тому вони менше потребують інтеграції. Тобто низька взаємозалежність і висока зв'язність привели до чудового побічного ефекту - зменшення потреби в інтеграції.

Головним недоліком архітектури ESB було дуже складний централізований додаток, від якого залежали всі інші програми. А в мікросервісной архітектурі цей додаток майже цілком прибрано.

Ще залишилися елементи, які пронизують всю екосистему мікросервісів. Але у них набагато менше завдань в порівнянні з ESB. Наприклад, для асинхронного зв'язку між мікросервісами досі застосовують чергу повідомлень, але це лише канал для передачі повідомлень, не більше того. Або можна згадати шлюз екосистеми мікросервісів, через який проходить весь зовнішній обмін даними.

Виділяють вісім принципів мікросервісной архітектури. А саме:

- проектування сервісів навколо бізнес-доменів. Це може дати нам стабільні інтерфейси, високосвязані і мало залежать один від одного модулі коду, а також чітко визначені розмежовані контексти;
- культура автоматизації. Це дасть нам набагато більше свободи, ми зможемо розгорнути більше модулів;
- приховування подробиць реалізації. Це дозволяє сервісів розвиватися незалежно один від одного;
- повна децентралізація. Децентралізоване прийняття рішень і архітектурні концепції, надайте командам автономність, щоб компанія сама перетворилася в складну адаптивну систему, здатну швидко пристосовуватися до змін;
- незалежне розгортання. Можна розгортати нову версію сервісу, не змінюючи нічого іншого;
- спочатку споживач. Сервіс повинен бути простим у використанні, в тому числі іншими сервісами;
- ізолювання збоїв. Якщо один сервіс падає, інші продовжують працювати, це робить всю систему стійкою до збоїв;
- зручність моніторингу. В системі багато компонентів, тому важко встежити за всім, що в ній відбувається. Нам потрібні складні інструменти моніторингу, що дозволяють заглянути в кожен куточок системи і відстежити будь-яку ланцюжок подій;

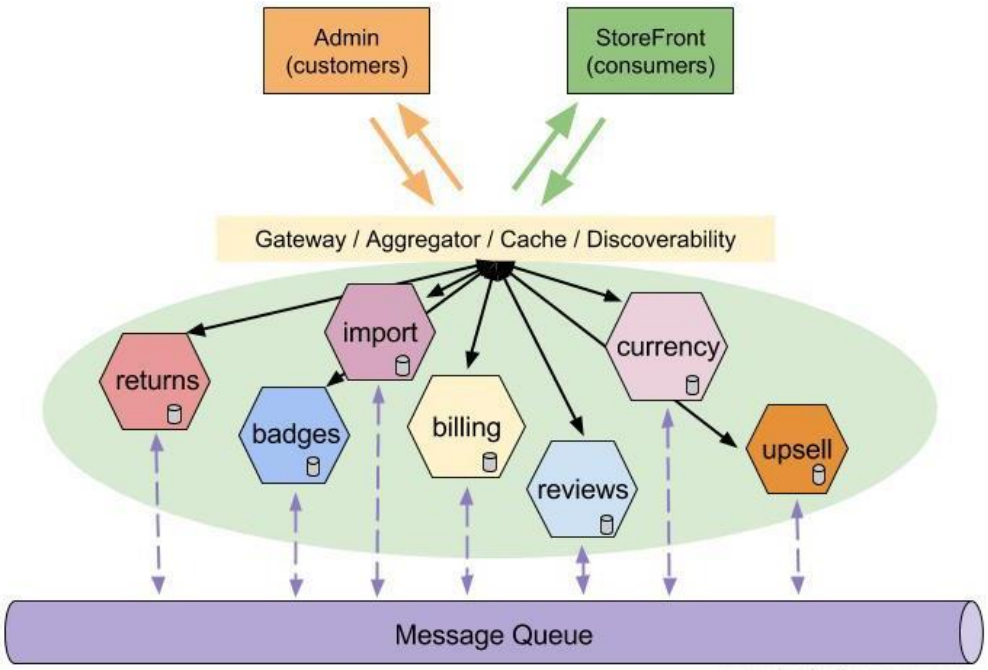


Рисунок 1.9 – Принципи взаємодії мікросервісів

Переваги:

- незалежність набору технологій, розгортання і масштабованості сервісів;
- стандартний, простий і надійний канал зв'язку (передача тексту по HTTP через порт 80);
- оптимізований обмін повідомленнями;
- стабільна специфікація обміну повідомленнями;
- ізольованість контекстів домену (Domain contexts);
- простота підключення і відключення сервісів;
- асинхронність обміну повідомленнями допомагає керувати навантаженням на систему;
- синхронність обміну повідомленнями допомагає керувати продуктивністю системи;
- повністю незалежні і автономні сервіси;
- бізнес-логіка зберігається тільки в сервісах.

- дозволяють компанії перетворитися в складну адаптивну систему, що складається з декількох маленьких автономних частин / команд, здатну швидко адаптуватися до змін;

Недоліки:

- використання численних технологій і бібліотек може вийти з-під контролю;
- потрібно акуратно управляти змінами вхідних / вихідних API, тому що ці інтерфейси будуть використовувати багато програм;
- використання «узгодженості в кінцевому рахунку» (eventual consistency) може привести до серйозних наслідків, які потрібно враховувати при розробці програми, від бекенд до UX.
- тестування ускладнюється, тому що зміни в інтерфейсі можуть непередбачуваним чином на інші сервіси.

2. ПОСТАНОВКА ЗАДАЧІ

У роботі необхідно обґрунтувати та проаналізувати використання методів сервісного облаштування програмних додатків для побудови адаптивних систем управління бізнесу.

Система представлена набором програмних модулів написана на мові програмування Java та знаходиться на локальному сервері.

У даній роботі необхідно визначити особливості використання обраної функції в конкретних умовах експлуатації.

В існуючому додатку є наступний набір основних функцій:

- можливість завантажувати файли (видео, картинки, документи, тощо);
- пуш повідомлення;
- відправка емейлів на пошту;
- взаємодія з базою даних;
- вхід у систему;
- вихід із системи;
- реєстрація користувача;
- управління особистим кабінетом;

Згідно з тим, що маємо великий статичний додаток, який потребує масштабування та перестройки функціональності. Методи встроєні статично і викликаються лише в конкретних місцях. Для того щоб внести зміни у функціональності необхідно ворушити велику кількість місць. Це завдання виконати досить складно. Ось тут на допомогу і приходить розбивка моноліту на сервіси.

Далі нам необхідно реалізувати побудову цих функцій таким чином, що б у них була сервісна технологія і організувати збірку під заданий варіант використання системи наборів сервісів з наявних у відповідності з бізнес цілями.

Ми маємо вихідний набір функцій які необхідно інкапсулювати, завернути в сервісну технологію, задати опис можливостей сервісів, реалізувати їх складання і

на підставі цього кінцевий користувач отримує відповідний можливості в зв'язку з його бізнес цілями. Практична частина роботи містить невеликий приклад, що демонструє розробку веб-сервісів на базі RESTful технології, що дозволяє звертатись до методів за допомогою URL, де не потрібні громіздкі WSDL документи для описання функціоналу того чи іншого сервісу. Приклад ґрунтується на реалізації концепції веб-сервісів в рамках Java-технологій. Для розуміння прикладу досить базових знань Java.

Кожна функція має особливості застосування в тому чи іншому середовищі. З цього ми функції розбиваємо на складові і кінцеву на мікросервіси і вони в свою чергу дозволяють отримати будь-яке забарвлення системи в залежності від використання та кінцевих бізнес-цілей користувача. Що значно прискорить розробку, помітно зменшить затрати по ресурсам та додаток набуде більшої гнучкості, що дозволить с легкістю масштабувати.

Для досягнення поставленої мети необхідно дослідити особливості розробки сервісних архітектур, приділити особливу увагу веб-сервісам. Визначити, який тип бази даних краще використовувати. Проаналізувати ефективність впровадження сервісів у статичних додатках. Проаналізувати можливості компоновки сервісів між собою. Реалізувати тестові сценарії для обраного функціоналу сервісів, забезпечити захист інформації та контроль доступу до методів сервісів. Надати результати і зробити відповідні висновки.

3. ПРАКТИЧНА ЕФЕКТИВНІСТЬ РОЗБИТТЯ МОНОЛІТНОЇ СИСТЕМИ НА СЕРВІСИ

3.1 Дослідження розбиття системи та компоновка веб-сервісів

Мікросервіси – це досить просто до тих пір, поки потрібно виконати дії, що охоплюють кілька служб.

Наївне перехресне впровадження сервісів - це просто зробити виклик наступної служби з попередньої. Прикладом може бути реєстрація нового учасника (Служба учасника), якому потрібно розмістити в мережі декілька файлів (Служба файлів) та метаінформацію про нього (Служба метаінформації).

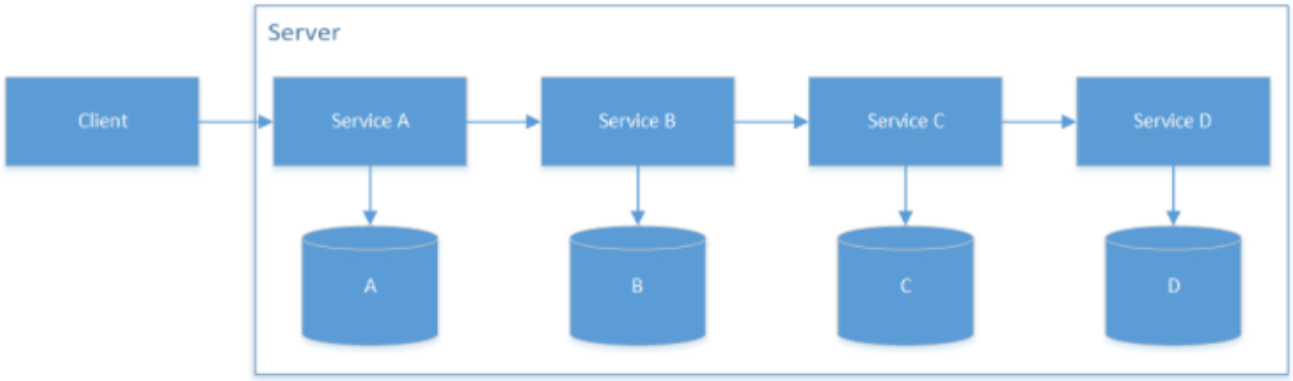


Рисунок 3.1 – Приклад послідовного впровадження сервісів

Основними проблемами такого підходу є:

- відсутність обробки несправностей, будь-яка з цих подальших служб може бути в автономному режимі, що означає, що частина потоку вийде з ладу, і система залишиться в нестабільному стані;

- зв'язування збільшується, що погано, оскільки, будуючи мікросервісну архітектуру, нам слід уникати створення залежностей. Кожна служба повинна мати можливість стояти окремо та мати доступ до неї RESTfully;
- складність зростає, це означає, що зміни бізнес-процесів можуть означати зміну декількох компонентів, коли все, що необхідно зробити, це додати новий крок;
- використання ресурсів зростає, оскільки кілька потоків заблоковано відкритими, оскільки кожна служба чекає завершення свого поточного;
- практично неможливо зрозуміти, в якому процесі система зараз перебуваєте;

Тому можна зробити висновок, що не треба реалізовувати сервісну взаємодію саме таким чином. Вона має стільки недоліків і нуль переваг.

Коли ми починаємо моделювати дедалі складнішу логіку, доводиться мати справу з проблемою управління бізнес-процесами, які виходять за межі окремих послуг.

Як дослідили вище, це з'єднання є катастрофічним з точки зору складності та кількості потоків, що утримуються відкритими, потенційно протягом тривалого часу (тобто понад пару секунд).

А щодо мікросервісів - ця межа буде досягнута раніше, ніж зазвичай.

Довіра до HTTP для здійснення міжслужбових викликів дорожча, ніж одна велика куля брудних послуг, тому межу буде досягнуто навіть раніше.

Коли справа стосується фактичної реалізації цього потоку, існує два стилі архітектури, яких можливо дотримуватися. Під час оркестровки покладаємося на центральний мозок, який керує та керує процесом, подібно до диригента в оркестрі.

Впроваджуючи цей підхід до організації, ми зменшуємо кількість відкритих потоків до двох і розв’язуємо мікросервіси, що добре.

Ми також можемо продовжувати синхронно, змушуючи клієнта чекати, поки Оркестрація буде успішною або невдалою. Це буде залежати від системного процесу та віри у своєчасне завершення в 99% випадків[17].

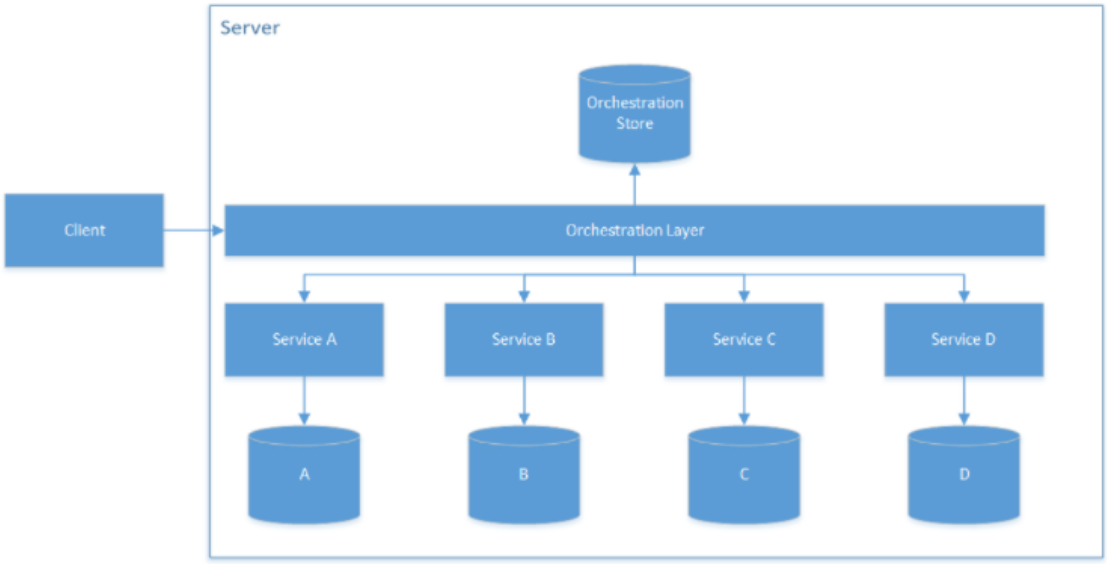


Рисунок 3.2 – Схема оркестровки

Але ця модель сама по собі не розглядає випадки відмов; для цього існує спеціальна схема роботи зі складними бізнес-процесами.

Впровадження оркестрації вирішує проблему зчеплення, але все ще має недоліки:

Він синхронно блокує Orchestrator і здебільшого однопотоковий процес, коли, можливо, все можна робити паралельно.

Зміни в бізнес-процесі вимагають змін в Orchestrator.

Але це не повинно бути так, ми можемо зробити краще за допомогою іншого методу.

За допомогою хореографії ми повідомляємо кожну частину системи про свою роботу, і дозволяємо їй опрацьовувати деталі, як танцівники, які всі знаходять свій шлях і реагують на оточуючих у балеті.

Хореографічні служби можуть розмістити свої виходи в черзі для отримання наступної служби, але це зберігає зв'язок, тому таким шляхом не йдемо. Як варіант використовувати eventing, кожна служба має тему, яку вона слухає, і надсилає повну подію на шину подій, коли це буде зроблено.

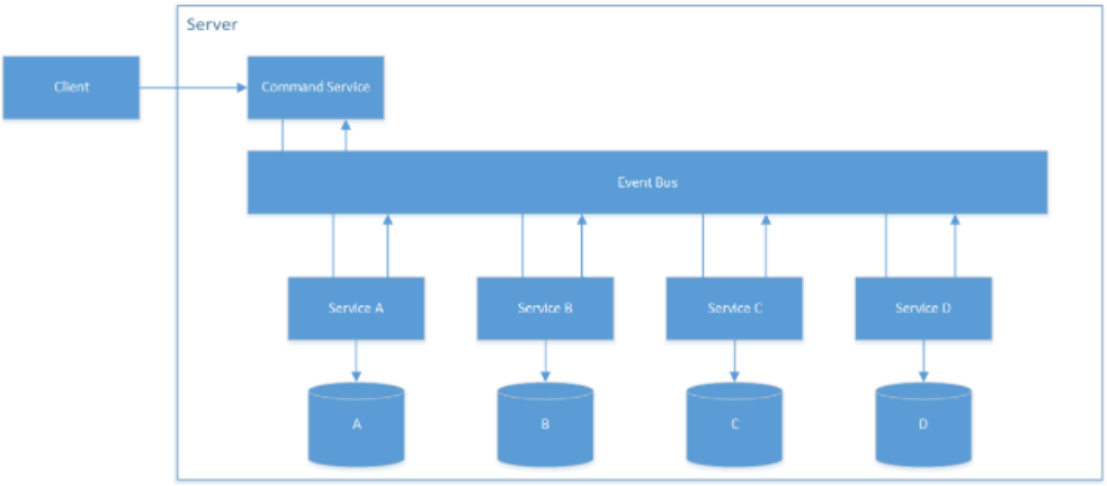


Рисунок 3.3 – Приклад схеми хореографії

Використання подій означає, що отримуємо кілька інших переваг:

– можемо виконати паралельний процес;

– можемо додавати деталі в процес без значних змін коду;

Також можна створити сховище подій, щоб мати можливість реалізувати джерело подій.

Тепер розглянемо нашу досліджувану систему. В результаті використання оркестровки та хореографії отримаємо наступні можливості, використовуючи REST архитектуру, які наведено нижче в таблиці.

Таблиця 3.1 – Приклад адаптивних запитів системи

HTTP Request	Результат запиту
GET /users/	Список користувачів системи
POST /users/	Створення користувача
GET /users/1	Отримати данні користувача з номером 1
DELETE /users/1	Видалити данні користувача з номером 1
PUT /users/1	Обновити данні користувача з номером 1
GET /files/	Отримати список файлів

Таблиця 3.1 – Продовження прикладу адаптивних запитів системи

POST /files/	Створити новий файл
GET /files/1	Отримати файл з номером 1
DELETE /files/1	Видалити файл з номером 1
GET /users/1/files/	Отримати файли користувача з номером 1, які були створені ним

3.2 Дослідження технології обраної для створення сервісів підтримки динамічного розвитку

З наступних причин використано саме Restful:

Неоднорідні мови та середовища - це одна з фундаментальних причин, яка така ж, як ми бачили і для SOAP.

Це дозволяє веб-програмам, побудованим на різних мовах програмування, взаємодіяти між собою.

За допомогою служб Restful ці веб-програми можуть розміщуватися в різних середовищах, деякі можуть бути в Windows, а інші - в Linux[13].

Але врешті-решт, незалежно від середовища, кінцевий результат повинен завжди бути однаковим, щоб вони мали можливість спілкуватися між собою. Інші веб-служби пропонують цю гнучкість для програм, побудованих на різних мовах програмування та платформ для спілкування між собою.

На малюнку нижче наведено приклад веб-програми, яка вимагає спілкування з іншими програмами, такими як Facebook, Twitter та Google.

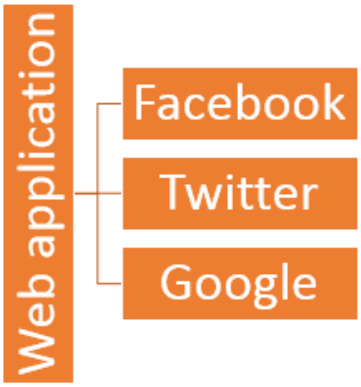


Рисунок 3.4 – Приклад веб-програми, яка спілкується з іншими

Тепер, якби клієнтська програма мала працювати з такими сайтами, як Facebook, Twitter тощо, вони, мабуть, повинні були б знати, на якій мові побудовані Facebook, Google і Twitter, а також на якій платформі вони побудовані.

Виходячи з цього, ми можемо написати інтерфейсний код для нашого веб-додатку, але це може виявитись кошмаром.

Facebook, Twitter і Google надають свої функціональні можливості у вигляді веб-служб Restful. Це дозволяє будь-якій клієнтській програмі викликати ці веб-служби через REST.

Сьогодні все повинно працювати скрізь, будь то мобільний пристрій, ноутбуки чи навіть автомобільні системи.

Чи можете ви уявити, скільки зусиль намагаються кодувати програми на цих пристроях для спілкування із звичайними веб-програмами? Знову ж таки, Restful API можуть спростити цю роботу, оскільки, вам дійсно не потрібно знати, що є основним рівнем для пристрою.

Зараз все рухається до хмари. Програми повільно переходять до хмарних систем, таких як Azure або Amazon. Вони надають багато API на основі архітектури Restful. Отже, додатки тепер потрібно розробляти таким чином, щоб вони стали сумісними з Хмарою. Отож, оскільки всі архітектури, засновані на хмарі, працюють за принципом REST, для веб-сервісів має більше сенсу програмуватись на архітектурі, що базується на REST, для найкращого використання хмарних служб.

REST слід використовувати, якщо для вас дуже важливо звести до мінімуму зв'язок між клієнтськими та серверними компонентами в розподіленому додатку.

Це може бути так, якщо вашим сервером буде користуватися багато різних клієнтів, над якими ви не маєте контролю. Це може також статися, якщо ви хочете мати можливість регулярно оновлювати сервер без необхідності оновлювати клієнтське програмне забезпечення.

Досягти такого низького рівня зчеплення непросто. Дуже важливо дотримуватися всіх обмежень REST, щоб досягти успіху. Підтримувати чисто зв’язок важко. Вибрати правильні типи носіїв та втиснути ваші дані у формати досить складно. Створення власних типів медіа може бути ще складнішим.

Адаптація насиченої поведінки сервера до єдиного інтерфейсу HTTP може заплутати і часом здається педантичною в порівнянні з відносно простим підходом RPC.

Незважаючи на труднощі, переваги полягають у тому, що у вас є послуга, яку розробник клієнта повинен легко зрозуміти завдяки послідовному використанню протоколу HTTP. Послуга повинна бути легко доступною завдяки гіпермедіа, а клієнт повинен бути надзвичайно стійким до змін на сервері.

Переваги гіпермедіа та уникнення стану сеансу робить збалансування навантаження простим, а розділення послуг можливим. Суворе дотримання правил HTTP робить доступність таких інструментів, як налагоджувачі та кешування проксі-серверів, чудовою річчю.

REST - це не офіційний стандарт, а скоріше архітектурний стиль мережевих систем, що складаються з клієнтів та серверів. Клієнти ініціюють запити до серверів; сервери обробляють запити та повертають відповідні відповіді. Запити та відповіді будуються навколо передачі "подань" "ресурсів". Ресурсом може бути, по суті, будь-яка послідовна та значуща концепція, до якої звертаються. Представлення ресурсу, як правило, є документом, який фіксує поточний або запланований стан ресурсу.

Мотивація веб-служб REST - це те, що рухає прогрес технологій: спрощувати складні речі. Веб-служби першого покоління покладалися на обмін XML-пакетами,

що відповідають специфікації SOAP (Простий протокол доступу до об'єктів) за допомогою протоколу HTTP. Насправді в специфікації веб-служб не сказано, що SOAP-повідомлення повинні обмінюватися через HTTP. Саме в цьому напрямі пішов Microsoft зі своєю архітектурою WCF. Це відокремлює обмін повідомленнями SOAP від базового протоколу зв'язку, що дозволяє TCP / IP та інші способи обміну SOAP. Але ... чи потрібно нам взагалі МИЛО?

Прихильники архітектури REST вважають SOAP та XML занадто важкими. Сам HTTP має достатньо можливостей, щоб програми могли спілкуватися по мережі. Насправді HTTP є тим, що забезпечує Інтернет, і він має дуже багатий словник з точки зору дієслів, URI, заголовків запитів та відповідей та типів Інтернет-засобів масової інформації. У сервісах REST також часто використовується JSON (JavaScript Object Notation) як зручна, «знежирена» альтернатива XML. JSON, будучи підмножиною JavaScript, є ідеальним форматом даних для побудови чистих клієнтів HTML, що працюють у веб-браузерах із вбудованою логікою JavaScript, та асинхронного доступу до ресурсів REST за допомогою AJAX для високочутливих інтерфейсів користувача.

З вибухом Інтернету, а тепер і Web 2.0, лише зараз ми починаємо бачити htt: автоматичну взаємодію між різними веб-сайтами, а також між веб-сайтами та клієнтськими програмами, веб-сайтами та бізнес-базами даних, як частину більш глобальної взаємозв’язок.

В Інтернеті дані рухаються швидше, ніж ми можемо їх переглянути. Отже, існує великий попит на програми, які можуть знаходити, відстежувати та контролювати інформацію, що надходить з різних джерел, включаючи дані про продажі, фінансову інформацію, інтернет-спільноти, маркетингові кампанії, щоб назвати декілька.

Техніка веб-сервісу, яка швидко розвивається, може змінити спосіб роботи Інтернету для бізнесу. Хоча в даний час веб-служби добре працюють для додатків від бізнесу до споживача, вони не працюють для програм для бізнесу. Наприклад, окремий клієнт може легко придбати книгу у інтернет-продавця, але для книгарні, яка бажає зробити обсяг покупки, це набагато складніше. У книгарні потенційно

потрібно буде використовувати кілька додатків для відстеження продажів, визначення повторних замовлень та відстеження відвантаження. Часто дані однієї програми потрібно повторно вводити в інші програми, роблячи весь процес неефективним. Веб-служби повинні мати можливість вирішити цю проблему[14].

REST також може змінити спосіб архітектури складних систем. Традиційна сервісно-орієнтована архітектура (SOA) повільно рухається до веб-орієнтованої архітектури (WOA) - терміну, придуманого Діоном Хінчкліффом - де додатки використовують багату мережу ресурсів REST. Замість кількох точкових служб SOA, корпоративні дані будуть представлені через мільйони детальних ресурсів REST, таких як сама Інтернет.

REST швидко стає найкращою технологією для створення довільних додатків, які спілкуються по мережі. REST повністю використовує протоколи та стандарти, що працюють у Всесвітній павутині, і простіший за традиційні веб-сервіси на базі SOAP. З появою хмарних обчислень та зростаючим інтересом до веб-розміщених додатків, REST-засновані технології можуть допомогти як у розробці розширених клієнтських інтерфейсних клієнтів, що звертаються до віддалених серверів; і при розробці власне серверів для маніпулювання структурами даних у клієнтському додатку (написаному будь-якою мовою) або безпосередньо у браузері.

3.3 Дослідження вибору СУБД

В останні роки багато говорять про бази даних NoSQL. Бази даних, такі як MongoDB, з’явились у відповідь на потребу бізнесу у спритності, продуктивності та масштабі. Отже, бази даних NoSQL можуть підтримувати широкий набір випадків використання, включаючи дослідницьку та прогнозну аналітику в режимі реального часу.

Сказавши це, традиційні реляційні бази даних, такі як SQL Server та MySQL, як і раніше виконують життєво важливу функцію у сфері зберігання, доступу та управління даними. Оскільки як реляційні бази даних, так і NoSQL мають свої сильні та слабкі сторони, важливо відповідати потребам та передбачуваним моделям використання до найкращого типу БД для роботи.

MongoDB - це підтип баз даних NoSQL, який називається база даних документів, що означає зберігає дані у схожих на JSON документах за допомогою BSON. Він використовує двійкове кодування для зберігання документів у колекціях. Він додає підтримку типів даних, таких як Date і двійкові файли, які не підтримуються в JSON.

Як і інші бази даних NoSQL, MongoDB - це розподілена, нереляційна база даних, яка призначена для великомасштабного зберігання даних та паралельної високопродуктивної обробки даних на великій кількості серверів. Бази даних NoSQL, такі як MongoDB, призначені для масштабування по горизонталі і розроблені для масштабування до сотень мільйонів і навіть мільярдів користувачів, які роблять оновлення та читають.

Кілька областей, де MongoDb - і бази даних NoSQL загалом - мають перевагу над своїми реляційними братами, включають:

- Зберігання великих обсягів даних без структури. База даних NoSQL не обмежує типи даних, що зберігаються. Крім того, ви можете додавати нові типи, коли бізнес змінюється.
- Використання хмарних обчислень та зберігання. Хмарне сховище - чудове рішення, але для масштабування потрібно легко розподілити дані на декількох серверах. Використання доступного обладнання на місці для тестування, а потім для виробництва у хмарі - це те, для чого розроблені бази даних NoSQL.
- Швидкий розвиток. Якщо ви розробляєте сучасні гнучкі методології, реляційна база даних сповільнить вас. База даних NoSQL не вимагає рівня підготовки, який зазвичай необхідний для реляційних баз даних.

Також хочу навести деякі переваги реляційної бази даних.

Хоча бази даних, що не містять схеми, такі як MongoDB, полегшують початок роботи, вони можуть спричинити труднощі. Якщо ви не вирішите, як дані зберігатимуться заздалегідь, нові шаблони запитів не будуть добре підтримуватися - особливо, якщо вони вимагають виконання найкращих налаштувань.

Реляційні бази даних дійсно добре зменшують надмірність даних, усуваючи необхідність зберігати однакові поля в декількох таблицях. Роблячи це, організації можуть заощадити на сховищі.

Реляційна база даних реалізує стандарти ACID (атомність, послідовність, ізоляція, довговічність), щоб забезпечити надійність обробки транзакцій. ACID - це набір властивостей, які використовуються при зміні бази даних, що гарантує дійсність транзакцій у випадку, якщо ви можете зіткнутися з помилкою, збоєм живлення, збоєм тощо.

Ще один момент полягає в тому, що для MongoDB існує багато чудових служб баз даних на базі хмарних технологій. Вони пропонують багато-багато інших послуг, які ви можете інтегрувати у свою базу даних, включаючи аналітику, ШІ, заплановані завдання та безліч інших корисних послуг.

Аналізуючи типи баз даних зупинилися все ж таки на MongoDB. Бо ми не знаємо чітку структуру даних, які захоче передавати клієнт. Наприклад, для реєстрації користувача комусь досить логіна та пароля, але бувають випадки, коли обов’язково потрібен номер телефону. Тому наші сервіси повинні бути максимально гнучкими.

3.4Розробка API для базового функціоналу бізнес-функцій

До безпосереднього проектування архітектури серверної частини повинні бути проаналізовані та сформовані вимоги, які мають відповідати цій архітектурі. Вимоги до системи поділяються на функціональні та нефункціональні. Функціональні описують можливості програмного інтерфейсу додатків, тобто

сценарії взаємодії клієнта з системою, тоді як нефункціональні склади додаткові обмеження на систему, не відносяться до перегляду.

Проаналізувавши предметну область та існуючі рішення, було сформовано функціональні вимоги до додатків API (таблиця 3.2). До системи був також визначений ряд нефункціональних вимог. Їх опис наведено в таблиці 3.3.

Таблиця 3.2 – Функціональні вимоги до системи

№	Вимога	Опис
1	Реєстрація та авторизація користувачів	Для використання сервісу необхідно передати кількість параметрів необхідних для реєстрації користувача. Це дозволить розмежувати дисковий простір , створити набір даних для додатку, який викликає сервіс і в подальшому забезпечить контроль доступу до збережених даних.
2	Загрузка файлів	Сервіс повинен мати можливість завантажувати файли
3	Перегляд і зміна метаданих файлів	Метадані про всі файли повинні зберігатися сервісом, користувач повинен мати можливість переглядати, змінювати і додавати власні метадані
4	Видалення файлів	Сервіс повинен мати можливість видаляти окремі файли з усім їхнім вмістом
5	Пошук даних	Сервіс повинен мати можливість шукати файли по одному або декільком параметрам
6	Обмін файлами	Сервіс повинен мати можливість поділитися своїми файлами з іншими сервісами

Таблиця 3.3 – Нефункціональні вимоги до системи

№	Вимога	Опис
1	Зручність використання	API додатка повинен відповідати сучасним вимогам до проектування програмного забезпечення, бути уніфікованим і інтуїтивно зрозумілим. Розроблений інтерфейс повинен бути детально задокументовано, що також спростить його використання
2	Розмежування доступу	Кожному обліковому запису повинна бути присвоєна одна або кілька ролей в системі, за допомогою яких розмежовується доступ до інформації. Користувач повинен мати доступ тільки до власних даних
3	Обмеження на обсяг збережених і переданих файлів	Сервіс повинний накладати обмеження на розмір файлів і на максимальний обсяг дискового простору
4	Шифрування даних при їх передачі	Всі взаємодії клієнта з сервером повинні відбуватися за протоколом HTTPS, що забезпечує конфіденційність даних при їх передачі
5	Надійність	Система повинна бути надійною. Додаток має продовжувати функціонувати при виникненні позаштатних ситуацій
6	Простота тестування	Архітектура повинна забезпечувати простоту тестування компонентів системи

Визначивши вимоги до системи, можна приступати безпосередньо до проектування архітектури.

Взаємодія клієнта з програмним інтерфейсом додатку заснована на використанні архітектурного стилю Representational State Transfer (скорочено REST). Фактично REST API - це всього лише набір кінцевих точок (веб-адрес), звертаючись до яких за допомогою HTTP-запитів, клієнт отримує у відповідь інформацію з сервера у форматі JSON, XML, HTML і т. д. Вся інформація на сервері представлена у вигляді ресурсів і підресурсів: користувачі програми, завантажені файли і т. д. У кожного ресурсу є свій унікальний ідентифікатор, ресурс має стан, і клієнт може отримувати або змінювати стан ресурсу за допомогою уявлень (як уже згадувалося раніше, під поданням можна розуміти JSON, XML, текст в певному форматі або що завгодно, що дозволяє нам розуміти поточний стан ресурсу). При цьому серверна частина ніяк не залежить від клієнтської, клієнтом веб-сервісу може виступати браузер користувача, мобільний додаток, інший сервер і т. д.

Нижче представлено схематичне зображення взаємодії компонентів в REST-архітектурі.

У програмному інтерфейсі REST для маніпулювання ресурсами використовуються стандартні HTTP-методи відповідно до специфікації протоколу[15]:

- GET використовується для отримання поточного подання ресурсів;
- POST використовується для створення нових ресурсів;
- PATCH використовується для повного або часткового оновлення існуючих ресурсів;
- PUT використовується для створення нового ресурсу або відновлення існуючого. Якщо ресурс із заданим ідентифікатором знайдений, інформація про нього повинна бути оновлена, в іншому випадку буде створено новий ресурс;
- DELETE використовується для видалення існуючих ресурсів.

Запити в такій архітектурі є самодостатніми, у сервера при їх обробці немає необхідності отримувати контекст додатку, оскільки клієнт включає в запит всі необхідні дані, використовуючи для цього заголовки і тіло запиту. Такий підхід підвищує продуктивність, спрощує дизайн і реалізацію серверних компонентів

системи. Узагальнюючи вищесказане, можна виділити кілька основних принципів, яким повинна слідувати розроблювальна архітектура:

- використання клієнт-серверної моделі;
- використання стандартних методів протоколу HTTP;
- використання унікальних ідентифікаторів ресурсів;
- маніпуляція даними через уявлення у форматі JSON;
- відсутність стану на стороні сервера.

Дотримання перерахованих принципів підвищує надійність і продуктивність сервісів. Розроблене програмне забезпечення має такі переваги, як простота і однаковість інтерфейсів, портативність програмних компонентів, легкість внесення змін, а також здатність еволюціонувати і пристосовуватися до нових вимог.

Авторизація за допомогою сревісу здійснюється з використанням протоколу OAuth 2.0. Цей протокол дає можливість стороннім додаткам отримувати доступ до захищених ресурсів сервера від імені власника цих ресурсів, при цьому додаток не має доступу до реєстраційних даних користувача.

У протоколі OAuth 2.0 існують чотири учасники процесу авторизації:

- власник ресурсу. Під власником мається на увазі користувач, якому належать дані на сервері. Він авторизує додаток, тим самим дозволяючи йому доступ до своїх даних;
- клієнт. Клієнт - це програма, яка хоче отримати доступ до даних користувача;
- сервер ресурсів. Всі дані користувача, що мають відношення до функціоналу додатку, зберігаються на сервері ресурсів, а клієнт може взаємодіяти з ним, використовуючи API сервісу;
- сервер авторизації. Використовується для зберігання і перевірки реєстраційних даних клієнта, а також видачі токенів (ключів доступу). Токен передається серверу ресурсів з кожним запитом і необхідний для ідентифікації клієнта.

Для використання протоколу OAuth клієнту необхідно мати власний обліковий запис на сервері. При реєстрації кожному клієнту надається унікальний

ідентифікатор і секрет, у нього також можуть бути вказані адреса перенаправлення (redirect URI) і області видимості (scopes). Коли додаток відправляє запит на отримання нового токена, в параметрах запиту вказуються області видимості, що визначають, який тип доступу йому потрібно отримати (читання, запис, повний доступ і т. д.). А на вказану адресу додаток буде перенаправлено сервером авторизації після того, як дані клієнта були перевірені, і користувач дозволив доступ до запитуваних областей видимості.

Для отримання токенів в протоколі визначені чотири можливі сценарії, причому вибір конкретного способу залежить від типу програми і підтримки цього сценарію сервером авторизації:

- неявний доступ (implicit grant). Спочатку клієнт переходить на сервер авторизації, надаючи при цьому свій ідентифікатор і redirect URI (секрет при цьому не використовується). Після перевірки даних і отримання від користувача дозволу, сервер перенаправляє клієнта за вказаною адресою і передає йому токен у вигляді фрагмента адреси. Цей сценарій підходить для мобільних і веб-додатків, які не можуть зберігати секрет в таємниці;
- код авторизації (authorization code grant). Використовується переважно серверними додатками, оскільки їх вихідний код недоступний третім обличчям. Аналогічний сценарієм неявного доступу, за винятком необхідності надання секрету, а також наявності додаткового кроку для отримання коду авторизації та подальшого запиту з його допомогою токена доступу;
- облікові дані власника (resource owner password credentials grant). Полягає в передачі клієнту імені та пароля користувача безпосередньо і в подальшому отриманні з їх допомогою токена доступу. Підходить для надійних додатків, яким дозволено мати прямий доступ до реєстраційних даних користувача;

- облікові дані клієнта (client credentials grant). Необхідні в тому випадку, якщо клієнт хоче отримати доступ до власної облікового запису, щоб змінити адресу перенаправлення або секрет.

Хоча протокол OAuth виглядає досить громіздким і складним, він фактично став стандартом в індустрії програмного забезпечення.

У системі, буде реалізований як сервер ресурсів, так і сервер авторизації. Це дасть можливість іншим розробникам використовувати програмний інтерфейс сервісу для створення власних клієнтських додатків.

У проектованої системі використовується нереляційна модель даних. Для відповідності функціонала додатка функціональним вимогам в MongoDB були виділені чотири основні колекції бази даних:

- User (користувач). Зберігає реєстраційні дані всіх користувачів додатку (адреса електронної пошти, хеш пароля в форматі bcrypt, ім'я та т. д.). У кожного користувача є набір належних йому файлів;
- Role (роль). Визначає ролі, які призначаються користувачам, причому один користувач може мати кілька ролей;
- File (файл). Зберігає метадані для користувача файлів і розташування файлу на жорсткому диску. Кожен файл належить одному власникові і може мати кілька додаткових властивостей;
- Property (властивість). Властивості файлу - це додаткові метадані, які були самостійно додані користувачем. Кожна властивість має поле key (ключ) і value (значення). У файлу може бути тільки одна властивість із заданим ключем. Для відображення сутностей на таблиці бази даних використовується технологія об'єктно-реляційного відображення (ObjectRelational Mapping).

Збереження файлів. Коли файл завантажується на сервер, йому присвоюється ідентифікатор UUID - це унікальний 128-бітний номер, причому ймовірність згенерувати один і той же UUID двічі так мала, що її можна вважати нульовою. На основі цього ідентифікатора генерується унікальний шлях до файлу. З UUID витягуються три перші пари символів і використовуються в якості імен папок,

кожна наступна з яких вкладена в попередню. Сам ідентифікатор стає ім'ям файлу. Вміст файлу копіюються на жорсткий диск цим шляхом, а всі метадані, включаючи його справжнє ім'я і шлях до файлу, зберігаються в базу даних.

Рисунок 3.5 – Приклад сгенерованого шляху до файлу

Такий спосіб зберігання допомагає знизити навантаження на файлову систему, оскільки велика кількість файлів в одній папці сильно погіршує продуктивність при роботі з ними. Використовуваний алгоритм вирішує цю проблему, розподіляючи файли рівномірно. Крім того, оригінальні імена файлів можуть містити заборонені символи, і операційна система не дозволить зберегти файли з такими іменами. У UUID ж використовуються тільки символ «-» і цифри шістнадцятирічної системи, будь-яка комбінація з яких завжди є коректним ім'ям файлу.

Для обміну файлами між користувачами був використаний механізм створення тимчасових посилань. Таке посилання дозволяє завантажити файл будь-якому користувачеві і воно активне протягом декількох годин після його створення.

Для генерації тимчасових посилань використовуються маркери JSON Web Tokens. JWT - це стандарт для створення токенов доступу, які складаються з трьох частин в форматі JSON: заголовка (header), корисного навантаження (payload) і цифрового підпису (signature). Тема містить метадані, що описують токен, в корисному навантаженні містяться безпосередньо дані, а підпис - це хеш-сума, яка обчислюється на підставі заголовка, корисного навантаження і секретного ключа і необхідна для підтвердження достовірності токена.

Стандартним способом підпису токена є використання алгоритму HMAC-SHA256. Для генерування хеш-суми цей алгоритм, крім даних вихідного повідомлення, використовує також секретний ключ. Будь-яка зміна даних або

значення хешу викличе розбіжність підпису. А оскільки секрет для підпису відомий тільки серверу, такий токен неможливо підробити.

Після того, як всі три елементи токена визначені, його можна перетворити в компактний формат. Для цього заголовок і корисне навантаження кодуються з використанням схеми кодування Base64, до них додається хеш-сума, і всі складові частини поділяються точками.

Рисунок 3.6 – Структура JWT

Коли користувач запитує посилання на файл, сервер генерує новий токен, в якості корисного навантаження якого використовуються ідентифікатор файлу і поле exp, що позначає час, коли токен стане невалідним. До корисного навантаження додається заголовок і підпис, після чого токен перетворюється в компактне уявлення. Клієнту повертається посилання виду https://www.example.com/links/token, де змінна token - це згенерований компактний JWT. Якщо користувач перейде по такому посиланню, сервер витягне токен, перевірить його підпис, чи не минув його термін дії, а потім знайде файл за ідентифікатором і поверне вміст.

У такого підходу є як свої плюси, так і мінуси. З одного боку, немає необхідності зберігати створені посилання в базі даних, оскільки JWT несуть в собі всю необхідну інформацію. З іншого боку, такі посилання не можна відкликати.

Поки термін дії токена не закінчився, і файл, на ідентифікатор якого посилається токен, що не знаходиться на відстані, посилання на скачування буде активна.

Файли додатки розподілені по каталогам відповідно до структури проекту Maven: основні класи знаходяться в /src/main/java, тестові класи - в /src/test/java, а каталог /src/main/resources містить файли, необхідні для запуску і налаштування програми .

Далі буде розглянуто реалізацію основних компонентів програми, описана їх структура і задачі.

Для конфігурації програми використовуються класи з пакета config і файли з директорії /src/main/resources.

Директорія resources містить наступні файли:

- application.properties. Spring Boot дозволяє налаштовувати більшість компонентів додатка, використовуючи для цього властивості Java, які зберігаються в цьому файлі у вигляді пар ключ-значення;
- import.sql. SQL-скрипт для завантаження вихідних даних. додає ролі користувача і адміністратора, облікові записи тестових користувачів, їх кореневі папки;
- keystore.p12. Сховище ключів і сертифікатів, необхідних для установки захищеного з'єднання по протоколу HTTPS.

Сервіси можна назвати центральними компонентами програми, оскільки саме ці класи містять логіку управління даними. Кожен сервіс представлений інтерфейсом з пакета service і забезпечений докладними коментарями JavaDoc, що описують його призначення, функціонал методів, параметри і повертаються значення. Винятком є клас DefaultUserDetailsService, який повертає об'єкт UserPrincipal з пакета security, який представляє поточного користувача, і використовується фреймворком Spring при аутентифікації облікового запису.

У проекті реалізовані наступні сервіси:

- FileService. Надає методи для роботи з файлами, включаючи створення токенів JWT для генерації тимчасових посилань;
- FolderService. Надає методи для роботи з папками;

- PropertyService. Дозволяє додавати, переглядати і видаляти властивості файлів;
- SearchService. Дозволяє здійснювати пошук файлів і папок за певними параметрами, а також переглядати недавно змінені файли і папки;
- UserService. Надає методи для роботи з обліковими записами користувачів програми.

У кожному інтерфейсі сервісу використовується анотація @PreAuthorize, яка служить механізмом контролю доступу на рівні планування для. Перевірка виконується перед викликом методу на підставі значення анотації. Якщо значення виразу одно true, то метод виконується, в іншому випадку Spring порушить виняток AccessDeniedException.

Лістинг 3.1 – Способи контрою доступів на рівні визиваємого методу

```
List<File> findFilesBySizeGreaterThan(Long size);

@Query("SELECT f FROM File f WHERE f.size > :size")
List<File> findFilesJPQL(@Param("size") Long size);

@Query(value = "SELECT * FROM files WHERE size > :size", nativeQuery = true)
List<File> findFilesSQL(@Param("size") Long size);
```

Розглянемо приклад з лістингу 3.1, в якому представлені два методи: отримання вмісту кореневої папки по id власника і отримання вмісту папки по id папки. У першому випадку перевірка доступу тривіальна. Необхідно лише перевірити, що id поточного користувача збігається з параметром, переданим методу. Але для аналогічної перевірки у другому прикладі нам необхідно знати власника папки з заданим id. У таких випадках в анотації @PreAuthorize використовується вираз hasPermission, яке делегує перевірку класу StorageItemPermissionEvaluator з пакета security. Цей клас знаходить об'єкти із заданим id і перевіряє умови доступу до них.

Кожен метод в реалізації сервісу відзначений анотацією @Transactional. Ця інструкція означає, що при виклику методу буде автоматично створена нова

транзакція, а при успішному поверненні з нього ця транзакція буде зафіксована. У разі виникнення в методі виключення відбудеться відкат транзакції.

Класи сервісів також використовують механізм публікації подій, що дає можливість виконати деякі дії, такі як видалення файлу з жорсткого диска, тільки після успішного завершення або скасування транзакції, забезпечуючи тим самим цілісність даних. Події з пакета event є спадкоємцями класу ApplicationEvent і містять інформацію про створені або віддалених сутності. Для публікації подій використовується клас ApplicationEventPublisher, а методи з сервісу DefaultFileService, відмічені анотацією @TransactionalEventListener, який обробляє ці події.

Контролери відзначаються анотаціями @Controller або @RestController. Кожен з них прив'язаний до певного веб-адресою і HTTP-методу, використовуючи анотацію @RequestMapping або її похідні (@GetMapping, @PostMapping і т. Д.).

Завдання контролерів полягає в перетворенні об'єктів передачі даних в формат JSON і назад за допомогою анотацій @RequestBody і @ResponseBody, валідації синтаксису запиту за допомогою анотації @Validated, добуванні параметрів запиту за допомогою анотацій @RequestParam і @PathParam, роботі з заголовками та встановлення статусу відповіді. Крім того, в методах контролерів використовується анотація @AuthenticationPrincipal, що дозволяє отримати об'єкт UserPrincipal, який представляє поточного користувача.

Контролери також можуть обробляти виключення, що виникають на нижчих рівнях, і повертати клієнту повідомлення про помилки. Прикладом такого контролера служить клас ExceptionHandlerController. Він відзначений анотацією @RestControllerAdvice, а його методи - анотацією @ExceptionHandler, яка описує, який тип виключення цей метод обробляє. Цей спосіб генерації повідомлень про помилки аналогічний використанню анотації @ResponseStatus.

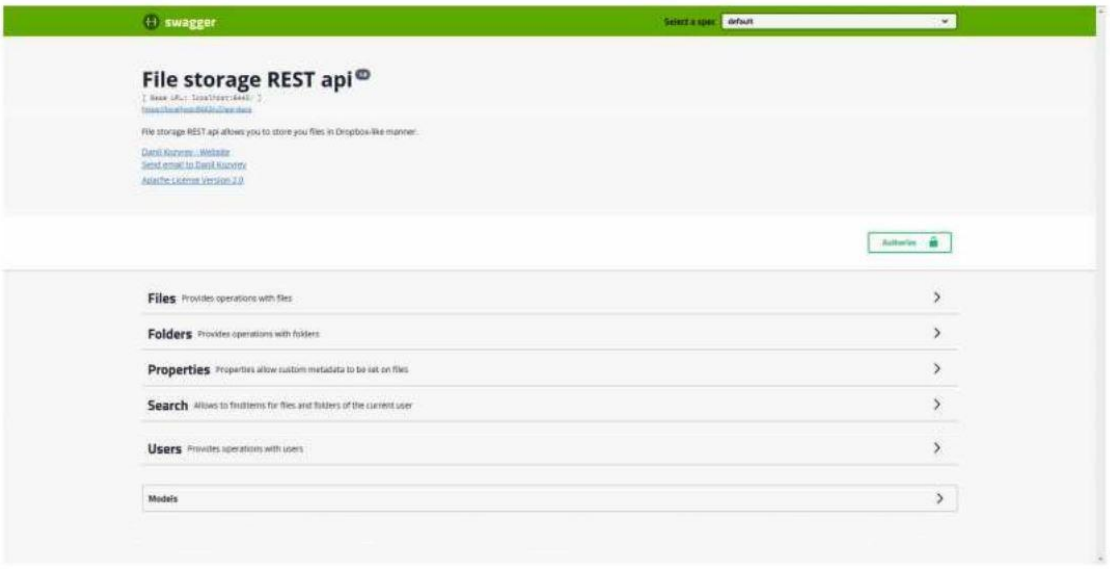


Рисунок 3.7 – Список ресурсів

Swagger автоматично генерує документацію до програмного інтерфейсу на основі використовуваних контролерів і анотацій @Api, @ApiOperation, @ApiResponses. Документація використовується в якості домашньої сторінки сервера і доступна за адресою: https: // localhost:8443.

Ресурси в документації групуються відповідно до параметру tags анотації @Api. Для кожної кінцевої точки наведені її адресу і HTTP-метод, параметри, які необхідно передати серверу, а також приклад успішної відповіді і можливі повідомлення про помилки. Кнопка «try it out» дозволяє

відправити запит на сервер, але для цього необхідно авторизуватись, натиснувши кнопку «Authorize» і ввівши дані облікового запису.

Секція Models містить інформацію про використовувані в API структурах даних. Як можна помітити, ці моделі відповідають об'єктам передачі даних, оскільки саме їх повертають і приймають контролери.

3.5 Тестування розроблених сервісів

Модульне тестування передбачає тестування кожного компонента системи в ізоляції. Для досягнення цієї мети використовуються об'єкти-заглушки, створювані бібліотекою Mockito за допомогою анотації @Mock. Поля тестованого класу заповнюються цими об'єктами за допомогою анотації @InjectMocks. Крім того, при необхідності можна «захопити» аргументи, передані методам заглушок, використовуючи об'єкти класу ArgumentCaptor.

Розглянемо приблизний алгоритм тестування методу:

- створюється тестовий метод, зазначений анотацією @Test;
- за допомогою класів-будівельників створюються тестові об'єкти;
- використовуючи методи when і thenReturn, встановлюються значення, які повинні повертати заглушки при виклику їх методів
- викликається тестований метод сервісу;
- використовуючи метод verify, перевіряється, що очікувані методи заглушок були викликані тестованим класом;
- використовуючи метод assert, перевіряється, що значення полів об'єктів відповідають очікуваним.

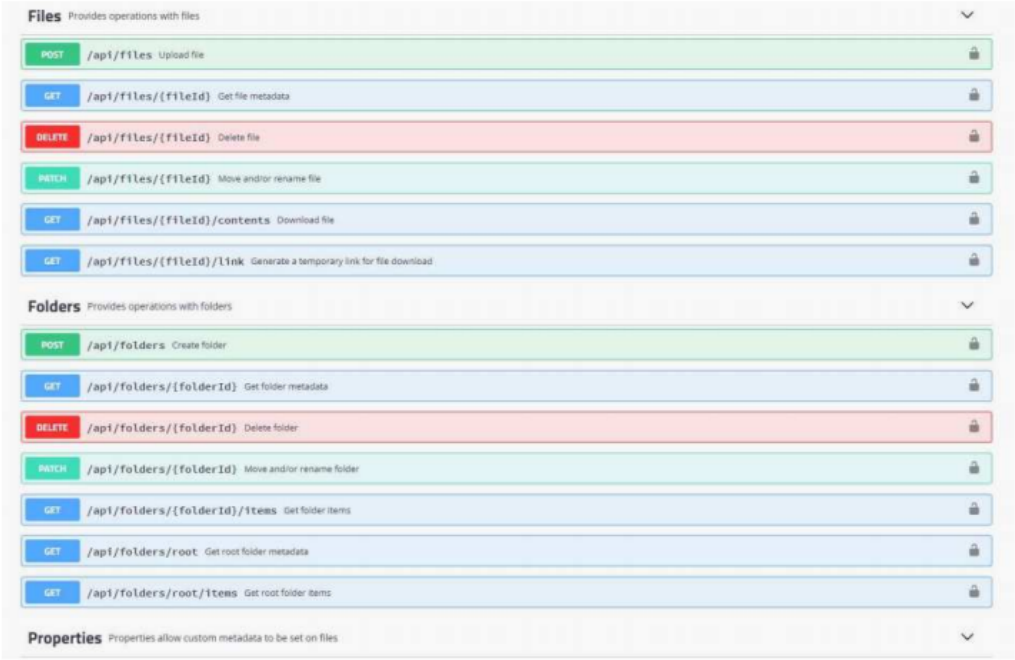


Рисунок 3.8 – Список кінцевих точок

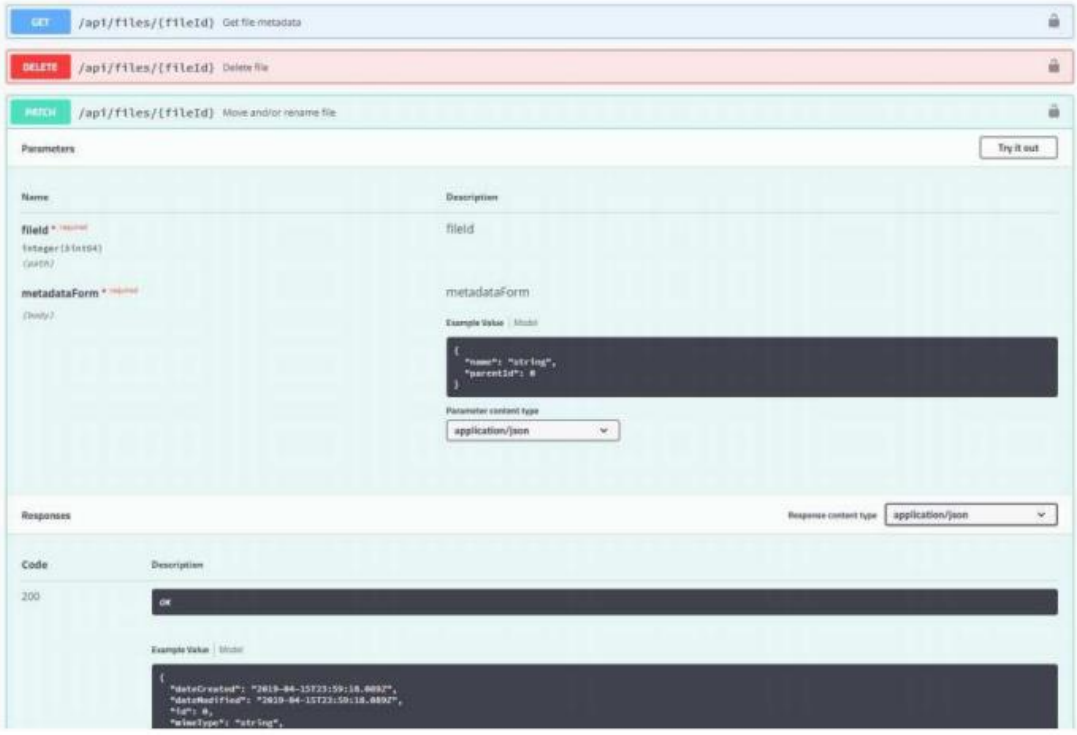


Рисунок 3.9 – Інформація о кінцевій точці

ВИСНОВКИ

Для виконання атестаційної роботи була поставлена задача дослідити процес підтримки якості програмного продукту за допомогою використання сервісів; класифікації сервісної архітектури, напрямків їх використання та розробити декілька сервісів.

Проаналізувавши різні архітектурні рішення для розв’язання задачі було обрано Restful архітектуру створення веб-сервісів.

У відповідність з метою були поставлені і реалізовані наступні завдання:

- дослідити особливості використання сервісів;
- узагальнити і систематизувати рівні і типи сервісів;
- проаналізувати ефективність використання сервісів в цілому;
- проаналізувати доцільність застосування веб-сервісів на існуючому додатку і інструменти для їх реалізації;
- реалізувати сценарії використання обраного функціоналу веб-додатку;

В ході роботи був проведений аналіз предметної області, розглянуті існуючі рішення, вивчені переваги і недоліки різних архітектурних рішень. Розроблено і детально описана архітектура системи, яка відповідає заявленим функціональним і нефункціональним вимогам. Обрані і вивчені програмні засоби реалізації розробленої архітектури. І, нарешті, статична система розбита на сервіси і протестована. Крім того, програмний інтерфейс програми був забезпечений докладною документацією у вигляді інтерактивної веб-сторінки.

Завдяки застосуванню сучасних програмних засобів і підходів до розробки реалізована система легко розширювана і може бути використана з будь-яким клієнтом. Таким чином, подальше пріоритетний напрямок розвитку проекту - це збільшити кількість сервісів, які будуть надавати базовий функціонал для створення додатків, який матиме можливість з легкістю комбінуватись між собою.

ПЕРЕЛІК ПОСИЛАНЬ

1. Екель, Брюс Філософія Java [Текст]./ Б. Екель – 4-е повне видання — СПб.: Пітер 2015. — 13 с.

2. Йенер, Мурат Java EE. Паттерни проектування для професіоналів [Текст]./ Алекс, Фидом — СПб.: Пітер 2016. — 55 с.

3. Шефер, Кріс Spring 4 для професіоналів [Текст]./ Х. Кларес – 4-е повне видання — ООО «І. Д, Віл'ямс» 2015. — 78 с.

4. Крістіан, Бауєр Java Persistence API та Hibernate [Текст]./ Гевін Кінг – 4-е повне видання — СПб.: Пітер 2016.. — 34 с.

5. Шварц, Берон NOSQL по максимуму. Оптимізація, реплікація, резервне копіювання [Текст]./ Петр Зайцев – 3-е повне видання —Пітер 2016.. — 22 с.

6. Крістіан, Бауєр Java Persistence API та Hibernate [Текст]./ Гевін Кінг – 4-е повне видання — СПб.: Пітер 2016.. — 34 с.

7. Martin Kalin, Java Web Services: Up and Running: A Quick, Practical, and Thorough Introduction [Текст] – 2-е повне видання – Reading Minds – 50 с.

8. Bill Burke, RESTful Java with JAX-RS 2.0 [Текст] – 2-е повне видання – Reading Minds – 100 с.

9. Leonard Richardson, RESTful Web Services [Текст] – 1-е повне видання – Reading Minds – 33 с.

10. Joshua Bloch, Effective Java [Текст] – 1-е повне видання – Reading Minds – 65 с.

11. Balaji Varanasi, Spring REST [Текст] – 1-е повне видання – Reading Minds – 14 с.

12. Brian Goetz, Java Concurrency in Practice [Текст] – 1-е повне видання – Reading Minds – 120 с.

13. Craig Walls, Spring Boot in Action [Текст] – 1-е повне видання – Reading Minds – 47 с.

14. Kathy Sierra, JHead First Java, 2nd Edition [Текст] – 2-е повне видання – Reading Minds – 222 с.

15. Робер Мартін, Чиста архітектура. Мистецтво розробки програмного забезпечення. [Текст] – 1-е повне видання – Reading Minds – 165 с.

16. Grebennik I., Reshetnik V., Ovezgeldyyev A., Ivanov V., Urniaieva I. (2019) Strategy of Effective Decision-Making in Planning and Elimination of Consequences of Emergency Situations In: Murayama Y., Velev D., Zlateva P. (eds) Information Technology in Disaster Risk Reduction. ITDRR 2018. IFIP Advances in Information and Communication Technology. Springer, Cham Scopus – 12с.

17. Harihara Subramanian, Hands-On RESTful API Design Patterns and Best Practices [Текст] – 1-е повне видання – 204-220 с.