Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Центр післядипломної освіти
(повна назва)
Кафедра Програмної інженерії
(повна назва)

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

другий (магістерський)
(рівень вищої освіти)

(позначення документа)

Дослідження методів розробки сервіс-орієнованих додатків для використання у публічних хмарах
(тема)

Виконав: студент 2 курсу, групи ІПЗмзд-17-1
спеціальності
121 Інженерія програмного забезпечення
(код і повна назва спеціальності)
освітньо-наукова програма
Інженерія програмного забезпечення
(повна назва спеціалізації)
Колосов І.Ю.
(прізвище, ініціали)
Керівник доц. Лановий О.Ф.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри ___________ Дудар З.В.
(підпис) (прізвище, ініціали)

2019 р.

Харківський національний університет радіоелектроніки

Факультет Центр післядипломної освіти

Кафедра Програмної інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 121 Інженерія програмного забезпечення
(код і повна назва)

Освітньо-наукова програма Інженерія програмного забезпечення
(повна назва)

ЗАТВЕРДЖУЮ:

“____»________20__р __________

Зав. кафедри проф. З.В.Дудар

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

магістрантові ***Колосову Ігорю Юрійовичу***

1. Тема роботи “ *Дослідження методів розробки сервіс-орієнованих додатків для використання у публічних хмарах»* затверджена наказом по Університету №________ від «____» квітня 2019 р.
2. Термін здачі студентом закінченої роботи «____» червня 2019 р.
3. Вихідні дані до проекту (роботи): *принципи розробки SOA додатків, методи розподілених обчислень, типи розгортання хмарних додатків, програмне забезпечення як сервіс. Організація даних: база даних та зовнішнє джерело даних. Форма діалогу: інтерактивно-графічний режим. Перелік використаних програмних засобів: ОС Microsoft Windows 10, Mac OS, IDE WebStorm, MongoDB 3.2, AllFusionProcessModeler 7, RationalRose. Технічне забезпечення: IBM–сумісний ПК Core i5 та вище.*
4. Зміст пояснювальної записки (перелік питань, що їх належить розробити) *аналіз предметної області, аналіз першоджерел інформації, постановка задачі дослідження та розробка моделі системи, визначення системи обмежень, дослідження особливостей розробки хмарних застосувань, побудова функціональної моделі системи, вибір засобів програмної реалізації та розгортання, програмування системи та проведення експериментів, висновки по роботі.*
5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень) *Мета роботи, сервіс-орієнтована розробка, постановка задачі, можливі шляхи розв’язання проблеми, опис моделі системи, технології розробки та розгортання, архітектура системи, інтерфейс програмної системи, висновки по роботі.*

Календарний план

№	Назва етапів дипломної роботи	Термін виконання етапів роботи	Примітка
1	Аналіз задачі дослідження	11-02-2019	виконано
2	Вибір засобів та технологій	10-03-2019	виконано
3	Проектування архітектури додатку	15-04-2019	виконано
4	Розробка коду програми	08-05-2019	виконано
5	Тестування і налагодження програми	27-05-2019	виконано
6	Підготовка пояснювальної записки.	01-06-2019	виконано
7	Підготовка презентації та доповіді	16-06-2019	виконано
8	Попередній захист	18-06-2019	виконано
9	Нормоконтроль, рецензування	18-06-2019	виконано
10	Занесення диплома в електронний архів	19-06-2019	виконано
11	Допуск до захисту у зав. кафедри	20-06-2019	виконано

Дата видачі завдання «______»_______________20___ р.

Керівник доц. _______________ Лановий О.Ф.

Завдання прийняв до виконання_________________ Колосов І.Ю.

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до атестаційної роботи: __ с., ___ рис., __ табл., ___ джерела.

Об'єкт проектування – комплекс рекомендацій стосовно створення і використання систем, що використовують дані як сервіс. Мета проектування – аналіз існуючих галузей щодо використання систем агрегації даних як сервісу. Метод проектування – збір інформації стосовно існуючих систем використання даних та великих даних як сервісу, особливостей застосування цих систем та отримання відповідного ефекту.

У результаті роботи створено програмну систему з агрегування і подальшого використання даних як сервісу. Також наведено апаратно-програмний інструментарій щодо розробки таких систем.

DAAS, BDAAS, ВІДКРИТІ ДАНІ, JAVA, ВЕБ-СЕРВІС, ПАРСИНГ, АРХІТЕКТУРА СИСТЕМИ, ВІЗУАЛІЗАЦІЯ ДАНИХ, ФОРМАТИ ДАНИХ.

Master`s thesis: ___ p. ___ pic., ___ tabl., ___ lit.

Project design target is a set of recommendations for the creation and usage of the systems which use data as a service.

The main purpose of target system design is analysis of existing domains of usage the data aggregation systems as a service. Design approach – information collection on existing systems for data usage and Big Data as a service as well as peculiarities of the usage of such systems, obtaining an appropriate effect and further analysis of result advantages and disadvantages.

Application with capabilities of collection, aggregation and usage of data as a service was created as a project outcome. Also software toolset was provided for the development of similar systems.

DAAS, BDAAS, OPEN DATA, JAVA, WEB SERVICE, PARSING, SYSTEM ARCHITECTURE, VISUALIZATION OF DATA, DATA FORMAT.

ЗМІСТ

Вступ 5
1 Аналіз проблемної галузі та постановка задачі 8
1.1 Сервісні технології та послуги 8
1.2 Сервіс-орієнтоване програмне забезпечення 12
1.3 Грануляція (зернистість) сервісів 16
1.4 Недоліки використання хмарних сервісів 18
1.5 Постановка задачі 19
2 Дослідження базових принципів розробки SOA 22
2.1 Мікросервіси 22
2.2 Мікросервісний підхід до рефакторінгу існуючих додатків 27
2.3 Контейнери 29
2.4 Інструментарій для створення веб-сервісів 31
2.5 Вимоги до програмної системи 37
3 Проектні рішення 39
3.1 Рішення, що підтримують розробку SOA додатків 39
3.2 Вибір технологій програмування для реалізації проекта 42
3.3 Архітектура системи що розробляється 60
4 Опис програмної реалізації системи 65
4.1 Розгортання проекту 65
4.2 Програмний інтерфейс 65
4.3 Технічна специфікація 70
4.4 Дослідження SOA-додатку 73
Висновки 76
Перелік джерел посилання 77
Додаток А 79
Додаток Б 87

ВСТУП

Сучасний світ – це світ інформаційних технологій. Мабуть, одним із найбільших досягнень людства у XX ст. стали електронно-обчислювальні машини та Інтернет. Вважається, що людство вже давно досягло тієї позначки, коли головним ресурсом на планеті є інформація. Відповідно до цієї тези, володіння інформацією є основною цінністю в бізнесі, економіці, політиці тощо. Саме з цих міркувань з'явилася знаменита крилата фраза Ротшильда «Хто володіє інформацією, той володіє світом». З кожним роком ця фраза стає все більш актуальною.

Актуальність теми дослідження визначається тим, що сфера послуг є найбільш швидкозростаючим сегментом світової економіки, в якому вже сьогодні зайнято більше половини працездатного населення планети [1,2,6]. Системи сервісів є динамічними конфігураціями людей, технологій, організацій та засобів обміну інформацією, які створюють і забезпечують цінність для користувачів, постачальників та інших зацікавлених сторін. Сфера послуг потребує створення своєї теоретичної бази, яка може містити наступні чотири основні блоки: основи знань про сервіси, інженерію сервісів, менеджмент сервісів і технологію реалізації сервісів.

Зараз настав час переходу на нову парадигму програмування, пов'язану ні з об'єктами, а з бізнес-процесами і їх складовою частиною - бізнес-функціями. Її ідея - компоновка застосувань шляхом виявлення і виклику сервісів, доступних в мережі, для виконання певної задачі. В результаті можна визначити сервіс-орієнтовану архітектуру (SOA) наступним чином: це архітектура застосувань, побудована на основі формалізованих бізнес-процесів, функції яких представлені у вигляді багаторазово використовуваних сервісів з прозорими описаними інтерфейсами. При цьому створення, впровадження або зміна бізнес-процесу пов’язані з композицією вже раніше розроблених сервісів, призначених для автоматизації бізнес-функцій. Компанії, що мають за мету діяльність з продажу або

обробки інформації, зараз знаходяться на етапі стрімкого розвитку. Основною причиною популярності та успіху такого виду бізнесу є той факт, що він доступний практично кожному. Однак найважливішим тут є те, що інформація здатна вплинути на діяльність індивідуальних підприємців, компаній і навіть держав. За допомогою неї можна коригувати існуючі процеси і ефективно приймати рішення. Наслідки цього можуть мати виключно позитивний ефект на прибутковості підприємницької діяльності.

Ідея хмарних обчислень з'явилася ще в 1960 році, коли Джон Маккарті висловив припущення, що колись комп'ютерні обчислення проводитимуться за допомогою так званих «загальнонародних утиліт». Дана теорія почала набувати популярності з 2007 року завдяки швидкому розвитку каналів зв'язку і стрімко зростаючим потребам користувачів [1].

Сервіс-орієнтоване програмне забезпечення (або SOA-додатки) є найбільш відомою сервісною моделлю доступу до хмарних застосувань. При використанні SOA-додатків з різних хмарних сервісів клієнти отримують можливість використовувати це сервіс-орієнтоване програмне забезпечення через браузер, без необхідності його встановлювати та підтримувати відповідними апаратними засобами. Як очікують фахівці, до 2020 року 15% міжнародного ринку програмного забезпечення мігрує у «хмару» [1]. Цей факт впливає не лише на сам ринок, але і змушує враховувати ці аспекти при розробці нового програмного забезпечення.

Для бізнесу SOA означає прискорене задоволення потреб клієнтів, реальну гнучкість бізнесу, швидкий час виходу на ринок, простоту співпраці і низьку вартість бізнесу. Для IT-організацій SOA означає більшу продуктивність, зниження витрат на IT-рішення за рахунок прискорення розробки застосувань, більш м'якого повторного використання сервісів, більш високої якості застосувань, і в цілому більш швидкого реагування на запити бізнес-клієнтів для поліпшення і модифікації системи. Сьогодні процес проектування, розробки та забезпечення функціонування програмного забезпечення значно еволюціонував. Вимогами до сучасних програмних систем є не лише функціональні вимоги, а й вимоги до продуктивності, надійності та ефективності рішень. Тому значну роль у задоволенні вимог до

програмних систем відіграє вибір архітектури реалізації. Під час розробки розподілених обчислювальних систем, зокрема грід та хмарних сервісів для розв’язання ресурсоємних задач щодо створення власних ІТ- рішень та керування бізнес-процесами, доцільно розробити власну архітектуру програмної системи.

Таким чином актуальність теми атестаційної роботи обумовлена поширенням хмарних сервісів. Робота присвячена дослідженню необхідних змін в сервіс-орієнтованій архітектурі розробки додатків у зв’язку з загальною тенденцією переносу прикладних застосувань у хмарне середовище В рамках написання атестаційної роботи планується дослідити особливості процесу інтеграції різних сервіс-орієнтованих додатків, що підтримуються різними хмарними середовищами та запропонувати методику їх програмної реалізації з використанням контейнерного завантаження на рівні мікросервісів, що базується на особливостях їх розгортання у приватних хмарах.

1 АНАЛІЗ ПРОБЛЕМНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Сервісні технології та послуги

Класичне визначення терміну сервіс дано фірмою IBM у вигляді: «сервіс – це видимий ресурс, що виконує повторюване завдання і описаний зовнішньої інструкцією» [1]. Таке визначення сервісу має два основні загально прийняті значення: бізнес-орієнтоване значення та IT – орієнтоване значення: «веб-сервіс» або «сервіс-орієнтована архітектура». У першому випадку акцент робиться на взаємодію при обміні і на нематеріальний характеру сервісу, а у другому – на технології програмного забезпечення, які дозволяють підтримувати сумісність різних програмних модулів.

Поширення інтегрованого Інтернету, корпоративні додатки, користувальницький досвід, мережі та мобільні сервіси надають більше можливостей для підвищення якості сервісів.

Розвиток науки про сервіси спирається на два таких відомих технічних нововведення в інформаційних технологіях, як Softwareas a Service (SaaS), коли програмне забезпечення використовується і орендується через Інтернет, і Service-Oriented Architecture (сервіс-орієнтована архітектура, SOA) як архітектурний стиль для створення IT-архітектури підприємства, що використовує принципи орієнтації на сервіси для досягнення тісного зв'язку між бізнесом і інформаційними системами, що його підтримують.

Сервіс-орієнтована архітектура впроваджує в промисловість нові можливості, нові шляхи для співпраці, нові інфраструктури та нові типи програмних додатків. Зростання складності сучасних інформаційних систем, включаючи грід/хмарні-інфраструктури, зумовило поширення модульного підходу до розробки їх програмного забезпечення з використанням стандартизованих за можливості інтерфейсів між частинами, як це передбачено концепцією сервіс-орієнтованої архітектури. Теоретичні переваги SOA полягають у підвищенні гнучкості при одночасному зниженні операційних витрат на створення системи

сервісів. Дана архітектура представлена у вигляді набору сервісів і процесів, які можна комбінувати, а також змінювати з часом відповідно до змін вимог за допомогою планувальників потоку завдань (workflows). Однак практичне застосування всього потенціалу SOA ускладняється через часту необхідність використовувати окремі сервіси з несумісними інтерфейсами.

В цілому, SOA являє собою модель взаємодії компонент, яка пов'язує різні функціональні модулі додатків (сервіси) між собою за допомогою чітко визначених інтерфейсів. Інтерфейси самі по собі не залежать від апаратних платформ, операційних систем або мов програмування, використовуваних для розробки цих додатків. Це дозволяє окремим сервісів взаємодіяти між собою одним і тим же стандартним, але в той же час універсальним способом. Така особливість використання інтерфейсу, незалежна від оточення і платформи, отримала назву моделі "слабкий зв'язок". На слабо пов’язаних сервісах будуються додатки, що об’єднують різні сервіси. Необхідність виклику декількох сервісів для задовольняння потреби вимагає певних координаційних заходів, які необхідно вирішувати. Основні підходи до координації веб-сервісів відомі як оркестровка та хореографія. При пошуку одного сервісу оркестровка пов’язана з реалізацією синхронного шаблону «запит-відповідь», а хореографія – з реалізацією асинхронної взаємодії між сервісом та споживачем сервісу за шаблоном «публікація-підписка».

Хмарні технології – це парадигма, що передбачає віддалену обробку та зберігання даних. Ця технологія надає користувачам мережі Інтернет, доступ до комп'ютерних ресурсів сервера і використання програмного забезпечення як онлайн-сервісу. Тобто якщо є підключення до Інтернету то можна виконувати складні обчислення, опрацьовувати дані використовуючи потужності віддаленого серверах [1]. Хмарні сервіси, що дозволяють перенести обчислювальні ресурси й дані на віддалені інтернет-сервери, в останні роки стали одним з основних трендів розвитку IT-технологій. Хмарні обчислення – це модель забезпечення зручного мережевого доступу на вимогу до загальних обчислювальних ресурсів (наприклад, мереж передачі даних, серверів, пристроїв зберігання даних, прикладним

програмам, програмам і сервісам - як разом, так і окремо), які можуть бути оперативно надані та звільнені з мінімальними експлуатаційними витратами або зверненнями до провайдера [1].

Хмарний сервіс – це будь-яка послуга, що доступна користувачеві за запитом через Інтернет з серверів хмарних обчислень провайдера, на відміну від тих, що надаються за рахунок власних на локальних серверах компанії. Хмарні сервіси розроблені, щоб забезпечити легкий масштабований доступ до додатків, ресурсів і послуг, а також повністю управляється провайдером хмарних сервісів. Для забезпечення узгодженої роботи вузлів обчислювальної мережі на стороні хмарного провайдера використовується спеціалізоване програмне забезпечення для зв'язку з клієнтами, що забезпечує моніторинг стану обладнання та програм, балансування навантаження, забезпечення ресурсів для вирішення різних завдань.

Хмарні сервіси є комбінацією існуючих технологічних рішень, які взаємно інтегровані для забезпечення максимального автоматизму і мінімізації участі людини в роботі комплексу.

Сервісне середовище створюється як хмарний сервіс. Воно буде в подальшому використовуватися для управління сервісами і системами сервісів з врахуванням кращих можливостей для інновацій бізнес-сервісів, що надаються хмарними технологіями за допомогою віртуалізації і поліпшення інформаційного обслуговування наукових кіл, промисловості, а також інших зацікавлених сторін.

На теперішній час виділяють три головні моделі обслуговування хмарних технологій, які іноді ще називають «шарами хмари» [3–7]. Ці три шари – послуги інфраструктури, послуги платформи та послуги додатків – є відображенням структури не лише хмарних технологій, але і інформаційних технологій загалом.

Стек хмарних технологій складається з трьох частин, кожна з яких представляє окрему модель надання сервісів (рис.1.1). Виділяють наступні основні моделі обслуговування за допомогою хмари:

- Програмне забезпечення як послуга (SaaS).
- Платформа як послуга (PaaS).
- Інфраструктура як послуга (IaaS).

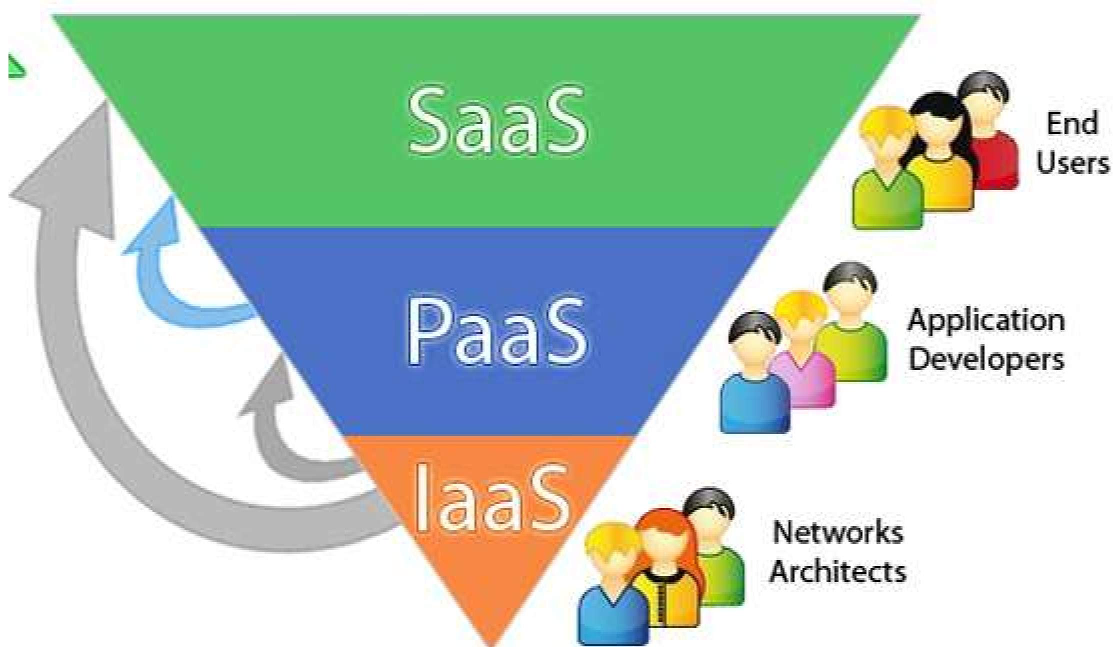

Рисунок 1.1 – Стек хмарних технологій

Програмне забезпечення як послуга (англ. Software as a Service) – це модель розгортання та реалізації програмного забезпечення, при якому постачальник розробляє додаток, ліцензує його, управляє ним, і надає споживачам доступ до нього через Інтернет.

Платформа як послуга (англ. Platform as a Service) – це модель надання хмарних обчислень, при якій споживач отримує доступ до використання інформаційно-технологічних платформ: операційних систем, систем управління базами даних, зв'язного програмного забезпечення, засобів розробки і тестування, розміщених у хмарних провайдерах.

Інфраструктура як послуга (англ. Infrastructure as a Service) – це модель надання хмарних послуг, при якій користувач отримує змогу керувати обчислювальними ресурсами системи, налаштуваннями мережі та розміром сховищ. Користувач має можливість у будь-який час збільшувати і зменшувати обсяги споживаних ресурсів.

1.2 Сервіс-орієнтоване програмне забезпечення

У широкому сенсі SOA – це підхід до розробки додатків, при якому додаток розщеплюється на окремі частини. Ці частини, як правило, розподілені за всією системою і взаємодіють одна з одною через мережу або через API. При цьому до розробки програмного забезпечення, його доставки і розгортання використовується підхід DevOps [16], коли команда розробників відповідає за всі аспекти програмного забезпечення, яке вона будує, навіть за постановку своїпродукції на ринок та за прибуток і збитки. DevOps (акронім від англ. Development і operations) – це набір практик, націлених на активну взаємодію та інтеграцію фахівців з розробки і фахівців з інформаційно-технологічного обслуговування. Базується на ідеї про тісну взаємозалежність розробки та експлуатації програмного забезпечення, і націлений на те, щоб допомагати організаціям швидше створювати і оновлювати програмні продукти і сервіси. Загальна схему формування прикладних додатків «on-demand» включає трьох учасників: провайдера сервісів, брокера і споживача сервісів (рис. 1.2).

Зазвичай, постачальниками доступу до хмарних послуг виступають провайдери телекомунікаційних мереж, що фізично з’єднують та забезпечують передачу інформації між провайдерами хмарних послуг та їх користувачами. Головними вимогами, що висуваються до провайдерів є надання безперервного, надійного та безпечного каналу доступу до провайдера хмарних послуг.

Щодо сервісів, розміщених в репозіторіях, то з ними виконуються наступні процедури [17-20]:

– знайти різні сервіси, які підходять для даної задачі (відкриття);

– вибрати найбільш підходящі сервіси серед доступних (вибір);

– об'єднати сервіси для досягнення мети (композиція);

– усунити невідповідності (дані, протокол, процес) серед комбінованих сервісів (посередництво);

– викликати сервіси згідно програмних конвенцій (виконання);

– контролювати процес виконання (моніторинг);

– сприяти заміні сервісів на еквівалентні (заміщення);

– забезпечити підтримку транзакцій і скасувати або пом'якшити небажані наслідки.

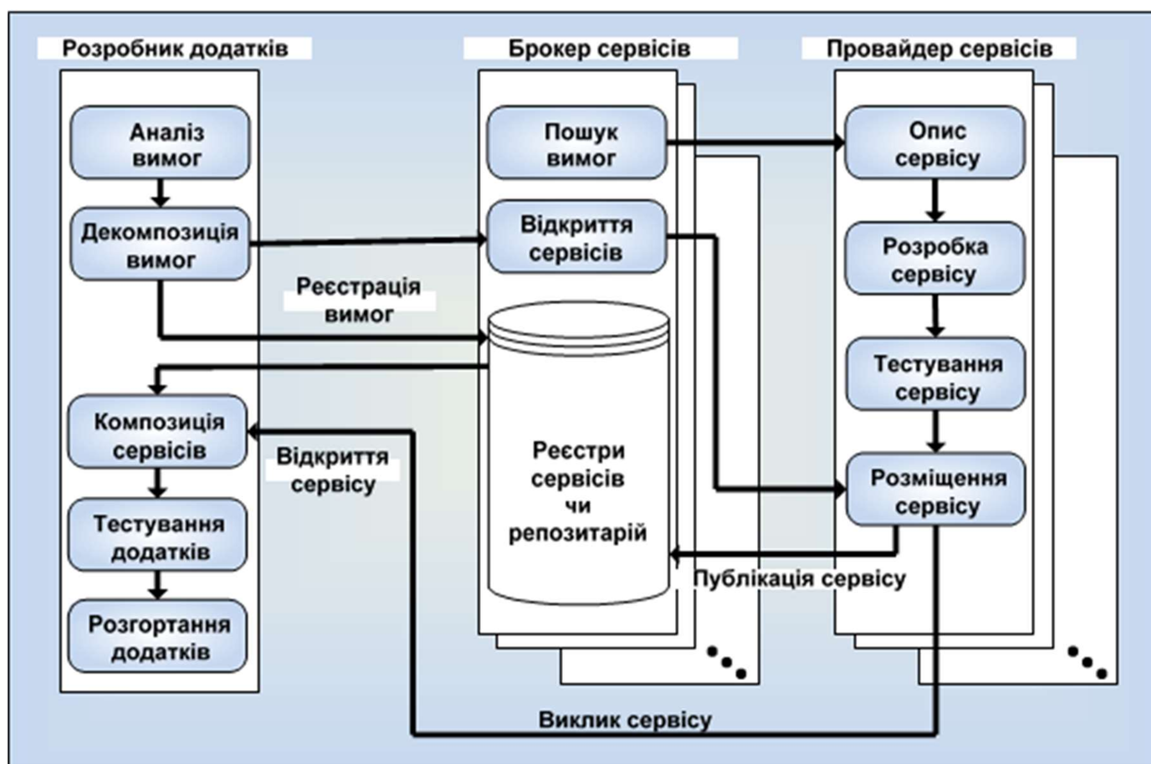

Рисунок 1.2 –Учасники використання SOA

Базова процедура відкриття сервісу ілюструється окремо на рис. 1.3, де показано порівняння запиту користувача та опису сервісу в реєстрі, що рекламується провайдером.

Проблемно-орієнтовані сервіси використовуються, як правило, для вирішення завдань і розробки додатків певного класу. Запити клієнтів на пошук необхідних сервісів, що направляються у реєстр сервісів, містять параметризовані описи завдань у вигляді кінцевого набору вхідних параметрів. Сервіс після обробки запитів клієнтів повертає клієнту результат, оформлений у вигляді кінцевого набору вихідних параметрів.

Сервіс може використовуватися кількома різними додатками. З іншого боку, в одному додатку можуть бути використані декілька сервісів.

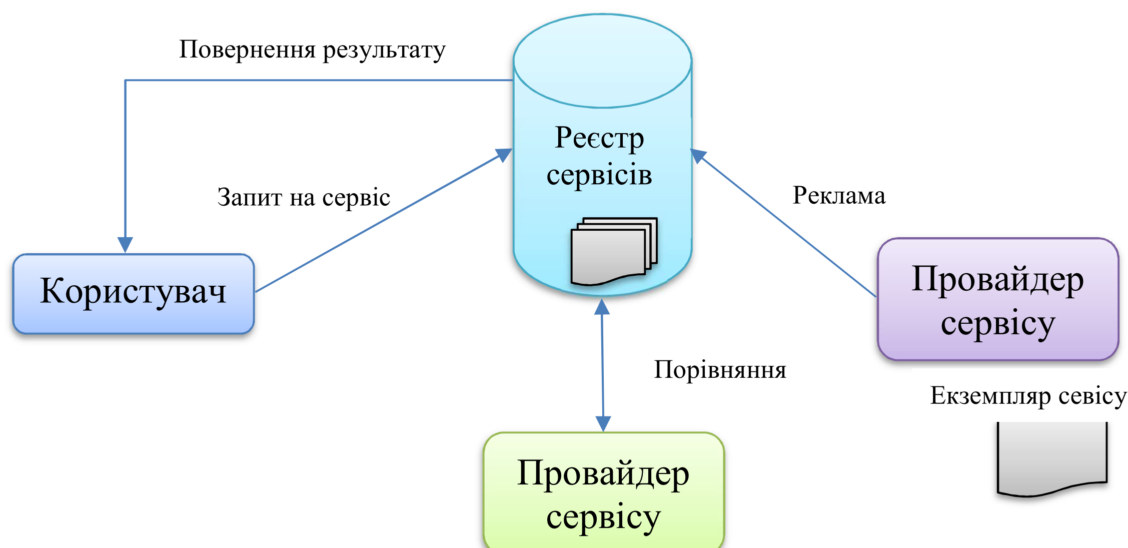

Рисунок 1.3 – Схема відкриття сервісу

Інформація про вхідні та вихідні параметри сервісу, яка необхідна клієнту для взаємодії з цим сервісом, міститься в опублікованому описі сервісу або надається сервісом за запитом клієнта.

Відповідно до моделі розгортання хмари поділяють на приватні, публічні та гібридні [2, 3] (рис.1.4).

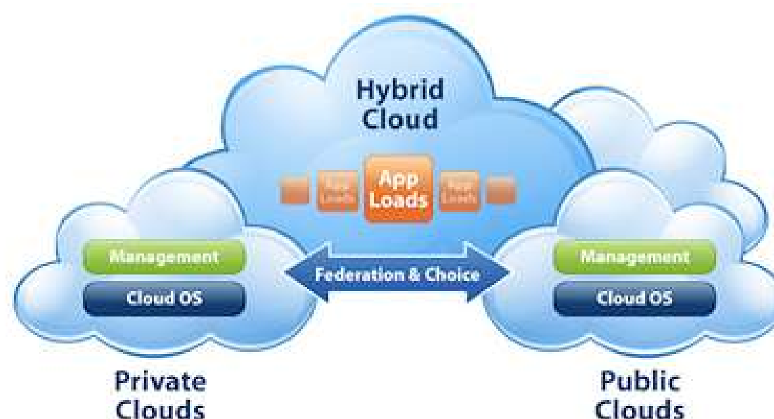

Рисунок 1.4 – Моделі розгортання хмарних обчислень

Private cloud (приватна хмара) — інфраструктура, яка призначена для використання однією організацією, що включає декілька споживачів (наприклад, підрозділів однієї організації), можливо також клієнтами і підрядчиками цієї організації. Приватна хмара може знаходитися у власності, управлінні і експлуатації як самої організації, так і третьої сторони (чи яких-небудь їх

комбінацій), і воно може фізично існувати як усередині, так і поза юрисдикцією власника.

Community cloud (хмара спільноти або громадська хмара) — вид інфраструктури, призначений для використання конкретним співтовариством споживачів з організацій, що мають загальні завдання (наприклад, місії вимог безпеки, політики, і відповідності різним вимогам). Громадська хмара може знаходитися в кооперативній (спільній) власності, управлінні і експлуатації однієї або більше організацій, співтовариств або третьої сторони (чи яких-небудь їх комбінацій), і вона може фізично існувати як усередині, так і поза юрисдикцією власника.

Public cloud (публічна хмара) — інфраструктура, призначена для вільного використання широкою публікою. Публічна хмара може знаходитися у власності, управлінні і експлуатації комерційних, наукових і урядових організацій (чи яких-небудь їх комбінацій). Публічна хмара фізично існує в юрисдикції власника – постачальника послуг. Загальнодоступна хмара - модель, коли незалежний провайдер надає в оренду ПЗ, інфраструктуру або платформи хмарних обчислень за принципом «ПЗ як послуга» (SaaS), «інфраструктура як послуга» (IaaS) або «платформа як послуга» (PaaS).

Hybrid cloud (гібридна хмара) — це комбінація з двох або більше різних хмарних інфраструктур (приватних, публічних або громадських), що залишаються унікальними об'єктами, але пов'язані між собою стандартизованими або приватними технологіями передачі даних і додатків (наприклад, короткочасне використання ресурсів публічних хмар для балансування навантаження між хмарами). Гібридна хмара – архітектура, що поєднує в собі риси приватних і громадських моделей хмарних обчислень. В цьому випадку критично важливі застосування або конфіденційні дані зберігаються в приватній хмарі, що належить самій компанії. У загальнодоступній же частині хмари розміщуються усі інші застосування, особливо складні, які нерегулярно використовуються або вимагають частого оновлення.

1.3 Грануляція (зернистість) сервісів

Під грануляцією сервісів розуміють рівень деталізації обслуговування. Зазвичай в SOA використовують модулі бізнес-логіки досить високого рівня, завдяки чому взаємодія між ними зводиться до обмеження числа повідомлень і зниження навантаження на мережу. На рис. 1.5 показано, як в якості сервісу можна вибрати всю програму (сервіс 1), підпрограму (сервіс 2) чи окрему процедуру (сервіс 3). Але останнім часом застосовуються *мікросервіси*, які реалізують тільки поодинокі функції. Мікросервіси повинні представляти бізнес- функції, а не загальні функції програмного забезпечення, наприклад, пошук у базі даних. Тобто мікросервіси належать до атомарних (простих) сервісів.

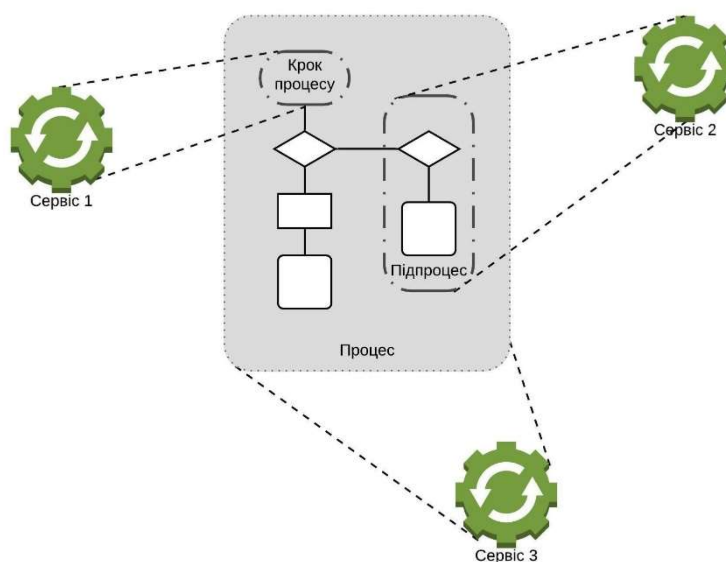

Рис. 1.5. Поділ монолітної програми на сервіси

Властивості, що характеризують сервіс, включають:

– можливість багаторазового застосування;
– можливість визначення сервісу одним або декількома технологічно незалежними інтерфейсами;
– можливість виклику сервісу, слабо зв'язаного з іншими сервісами, що забезпечують можливість взаємодії сервісів між собою.

Сервіси не запам'ятовують інформацію або не зберігають стан від одного виклику до іншого. Сервіси мають бути незалежними, автономними, що не вимагають обміну інформацією чи станом від одного запиту до іншого або залежать від контексту або стану інших сервісів. Коли такі залежності необхідні, вони зазвичай зазначаються у специфікаціях бізнес-процесів, функцій і моделей.

На сьогоднішній день існує кілька способів доставки сервіс-орієнтованих додатків для кінцевих користувачів – традиційно за допомогою е-інфраструктури підприємства, мобільних пристроїв і через хмару. Як раз додатки, розміщені у хмарі, змінюють усі традиційні правила, на яких будуються SOA.

З 2017 року розпочався справжній бум зі створення нового покоління MSA [4], або SOA другого покоління, що базується на використанні мікросервісів.

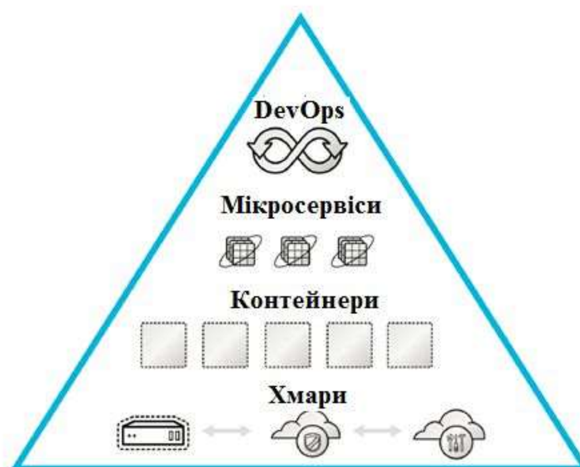

Рис. 1.6. Ієрархія нових концепцій SOA другого покоління

Мікросервіси виглядають так само, як веб-API в тому, що вони можуть бути доступні за допомогою простих HTTP протоколів запиту і форматів представлення даних. Разом з тим вони потребують власного програмного оточення, яке їм забезпечують контейнери. Контейнери відрізняються від віртуальних машин у тому, що додатки, що працюють в контейнерах, розділяють спільно операційну систему. Мікросервіси також розгортаються швидше в контейнерах, ніж на віртуальних машинах. Це може бути дуже корисним під час горизонтального масштабування сервісів зі зміною навантаження або при перерозподілі мікросервісів у зв’язку з відмовою мережевого серверу. Контейнери не є

необхідними для розгортання мікросервісів, але при цьому мікросервіси необхідні для обґрунтування доцільності використання контейнерів.

Сектор сервісів є найбільш швидкозростаючим сегментом світової економіки, в якому вже сьогодні зайнято більше половини працездатного населення планети. Системи сервісів є динамічними конфігураціями людей, технологій, організацій та засобів обміну інформацією, які створюють і забезпечують цінність для користувачів, постачальників та інших зацікавлених сторін.

1.4 Недоліки використання хмарних сервісів

Концепцію моделі хмарних обчислень часто розглядають двояко, деякі в ній бачать ризики для безпеки і нові «вектори загрози», але разом з тим дана система має новими можливостями для підвищення безпеки. Покращена спостережність інфраструктури, автоматизація та стандартизація – всі ці можливості підвищують рівень захищеності інформації. Наприклад, якщо використовувати заздалегідь заданий набір Cloud-інтерфейсів паралельно з централізованим управлінням ідентифікаційної інформацією, поряд з політикою управління доступом, то ми на порядок зменшуємо ризик доступу клієнтів до небажаних ресурсів. Такі заходи безпеки, як виконання обчислювальних сервісів в ізольованих доменах, використання шифрування до даних, значно підвищують збереження інформації, зменшуючи її втрати.

До основних недоліків хмарних технологій можна віднести:

– погрози інформаційній безпеці;

– прив'язка «хмарної» технології до конкретного постачальника послуг, збої на стороні провайдера, вихід з ладу інтерфейсу адміністрування, банкрутство і поглинання оператора;

– втрата зв'язку з мережею провайдера, DDoS-атаки і втрата відповідності вимогам регулювальників.

– збереження призначених для користувача даних залежить від компанії провайдера;

– для отримання якісних послуг користувачеві необхідно мати надійний і швидкий доступ до мережі Інтернет;

– відсутність загальноприйнятих стандартів у напрямах забезпечення безпеки хмарних технологій.

Незважаючи на вказані недоліки процес розвитку хмарних технологій та їх впровадження в усі сфери діяльності людини є незупинним.

1.5 Постановка задачі

На теперішній час потенційний користувач програмного продукту може не лише сформулювати свої побажання до його зовнішнього вигляду та функцій, а й описати їх в формальному вигляді – наприклад, за допомогою уніфікованих графічних діаграм (IDEF, UML, PBMN, EPC та подібні їм). Також замовник може представити своє бачення призначеного для користувача інтерфейсу бажаної системи у вигляді макетів веб-сайту або мобільного додатку. Користувач досить часто не тільки знає, що він хоче, а ще і як – з оголошенням технічних деталей, таких як мова програмування, фреймворк тощо. Більш того, деякі замовники можуть представити навіть початковий прототип системи, який розроблено самостійно. Подальше посилення цієї тенденції з урахуванням переважаючого на сьогодні сервіс-орієнтованого підходу до надання продукту як послуги можливо у вигляді появи сервісів з автоматичного формування програм на підставі вимог замовника і користувача в одній особі.

Застосовуючи цей тренд до задачі дослідження можемо отримати сервісну модель генерації програмного продукту, яка функціонує за принципом

декларативного програмування – від оголошення вимог до результату, тобто модель «чорного» або «сірого» ящика (рис. 1.7). Ступінь «чорноти» ящика визначається рівнем знань та умінь замовника-користувача в сфері IT.

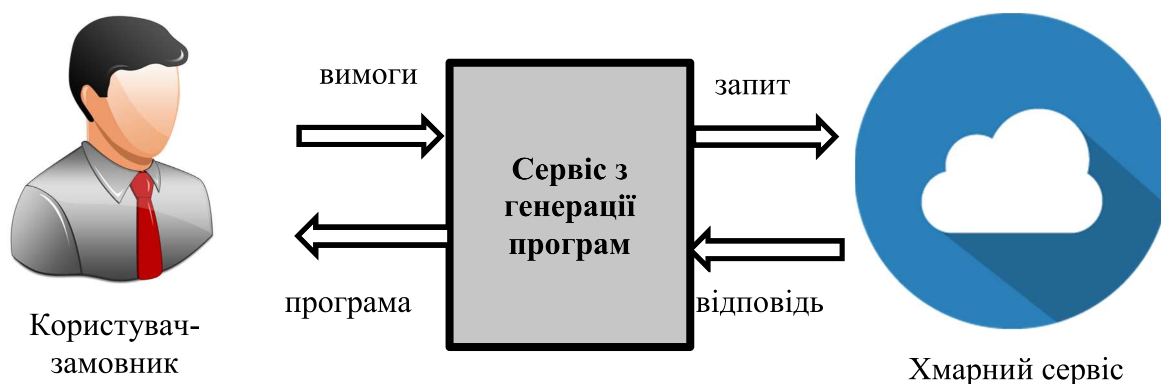

Рисунок 1.7 – Модель інтелектуального хмарного програмування

Метою атестаційної роботи є розробка методики створення SaaS-орієнтованих програмних продуктів. Методика, що розробляється, буде орієнтована на використання W3C-технологій (HTML5, CSS 3, SCCS, TypeScript).

Основною задачею методики є поділ програмного коду на модулі на етапі створення призначеного для користувача рівня SaaS. Для розв’язання цієї задачі в рамках написання атестаційної роботи необхідно виконати наступне:

- виконати аналіз предметної області та здійснити постановку задач користувача відповідно до вимог розроблюваного ПЗ;

- виділити підзадачі, які необхідно вирішити на призначеному для користувача рівні;

- визначити набір модулів, необхідних для розв’язання підзадач. У якості модуля будемо розглядати сукупність файлів HTML (для розмітки інтерфейсу модуля), CSS (для зберігання стилю модуля), JavaScript (для зберігання програмного коду модуля). Приймати рішення про необхідність переведення частини ПЗ в мікросервіс будемо здійснювати на підставі правила: в модуль необхідно виділяти ту і лише ту функціональність, яка використовується більш ніж один раз. Це правило дозволить уникнути створення мікросервісів без

необхідності. Операцію визначення набору мікросервісів необхідно рекурсивно виконувати для кожної частини ПЗ та для кожного вже згенерованого мікросервісу, поки подальше виділення мікросервісів не досягне межі;

- виконати програмування визначених мікросервісів. Цей етап вважається виконаним, якщо функціональність кожного з мікросервісів відповідає меті їх використання. При цьому зробимо припущення про можливість використання для оцінювання загальної функціональності ПЗ «заглушок» для звернення до серверної стороні репозиторія і генерування повідомлень від інших мікросервісів, які також може використовувати цей мікросервіс;

- здійснити проектування контейнерів для мікросервісів і завантажувачів мікросервісів в контейнери (завантажувачі можуть бути каскадними: завантажувач робить завантаження мікросервіса, в процесі якого мікросервіс, що завантажується, також може завантажити необхідний набір мікросервісів, кожен з яких теж може завантажити необхідний набір мікросервісів). В якості веб-контейнера будемо використовувати розмітку частині сторінки, підготовлену до завантаження мікросервісів;

- виконати функціональне тестування розробленого сервіс-орієнтованого додатка та перевірити його працездатність.

Застосування такої методики при розробці мережевих сервіс-орієнтованих додатків дозволить отримати модульні додатки, що функціонують на призначеному для користувача рівні, а також здійснити декомпозицію задачі, що вирішується засобами ПЗ, на множину підзадач, що дозволить різним розробникам в майбутньому повторно використовувати отримані мікросервіси.

В результаті проведеного аналізу предметної області було розглянуто принципи та концепції розробки додатків, що будуються за SOA, визначено поняття сервіс-орієнтованих технологій та сервісів, розглянуті їх переваги та недоліки.

2 ДОСЛІДЖЕННЯ БАЗОВИХ ПРИНЦИПІВ РОЗРОБКИ SOA

2.1 Мікросервіси

Мікросервіс є додатком з однією функцією, такою, наприклад, як маршрутизація мережевого трафіку, онлайн-платіж або аналіз медичного тесту. Така концепція не нова: вона базується на концепції веб-сервісів, а об’єднання мікросервісів (атомарних процесів) у функціональні додатки є еволюцію сервіс-орієнтованої архітектури (SOA) бізнес-процесів, яку вже використовують протягом кількох років. Мікросервісні додатки набагато простіше, ніж традиційні SOA з їх стандартами, включаючи складні сервісні шині підприємства (ESB) у формі проміжного програмного забезпечення, необхідні для спілкування між усіма сервісами [4].

Розпочалася епоха мікросервісів: вони розробляються, розгортаються і масштабуються незалежно, швидко адаптуються до змін, порівняно просто реалізують функції малого бізнесу. Компанії самі можуть з чистого аркуша розробляти мікросервіси, які при необхідності можна швидко і легко змінювати, тому що вони представлені меншим за розміром кодом. Це зовсім не те саме, як мати справу з традиційними монолітними додатками з кодом довжиною у мільйони рядків, що розроблялися раніше. Мікросервіси призначені для роботи у хмарних середовищах, обчислювальні ресурси яких, що необхідні для функціонування певних додатків, можуть збільшуватися і зменшуватися за бажанням. Мікросервіси розробляються так, щоб їх бути легко замінити, якщо трапляються негаразди в інфраструктурі чи якийсь конкретний сервіс припиняє роботу.

З усіма перевагами, які мікросервіси привносять, вони також створюють і складнощі, а саме:

- вимагають інтеграції і оркестрування, оскільки прикладні додатки можуть містити сотні або тисячі мікросервісів;

– потребують більш досконалих засобів автоматизації розгортання і конфігурації, вибору та моніторингу із-за більшої кількості мікросервісів, з яких складається додаток.

Можна виділити наступні заходи з подолання зазначених складнощів і використання переваг мікросервісів в повному обсязі [9]:

– вибір контракту (або внутрішнього зв'язоку) з мікросервісом;
– вилучення (відокремлення) мікросервісів з існуючих монолітних додатків;
– пошук (відкриття) сервісів;
– координація взаємодії;
– управління складністю розгортання сервісів і їх масштабованістю;
– видимість сервісів.

Виконаємо моделювання процесу застосування SOA-додатку.

Нехай необхідно розробити додаток, що забезпечує виконання не менш за n функцій λ (рис.2.1).

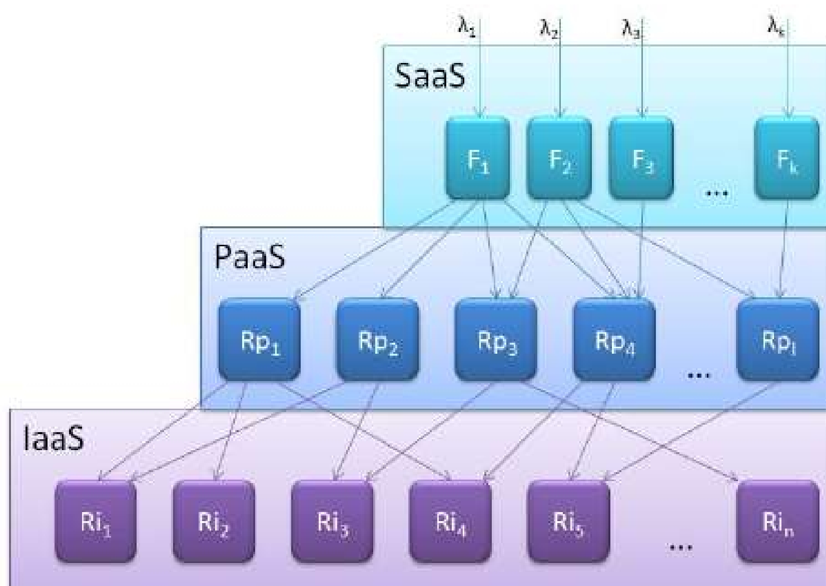

Рисунок 2.1 – Архітектура SOA-додатку

Тоді для створення такого сервіс-орієнтованого додатку спочатку необхідно скласти перелік можливих функцій F, що надаються хмарним сервісом (або сервісами):

$$F = \{f_j\}, j = \overline{1, k}. \quad (1)$$

В якості джерела інформації можуть виступати результати аналізу та/або моніторингу існуючих хмарних сервісів, що створені у вітчизняних або іноземних хмарах, аналіз наукових досліджень та показників ринку хмарних послуг тощо. На базі аналізу складається перелік проектів *P* хмарних сервісів, що становлять інтерес для використання в рамках програмного продукту:

$$P = \{p_i\}, i = \overline{1, l}. \quad (2)$$

Проекти можуть бути розроблені різними фірмами та можуть відповідати деяким типовим хмарним рішенням. Проекти будуть відрізнятися за складом сутностей предметної області, по організації бізнес-процесів їх експлуатації. Відповідно, кожен проект буде реалізовувати деяку підмножину функцій. Це можна відобразити за допомогою матриці:

$$P = \{p_{ij}\}, i = \overline{1, k};\ j = \overline{1, m}. \quad (3)$$

Елементи матриці формується наступним чином:

$$P_{ij} = \begin{cases} 1, \text{якщо функція } j \text{ входить до проекту сервісу } i; \\ 0, \text{якщо функція } j \text{ не входить до проекту сервісу } i. \end{cases}$$

На основі аналізу формуються перелік хмарних платформ та компонентів (на рівні PaaS та IaaS), що здатні забезпечити експлуатацію SaaS-додатку, і перелік функцій платформ. Відповідно формується множина хмарних платформ, до яких може звертатися додаток:

$$RP = \{Rp_n\}, n = \overline{1, kRP}, \quad (4)$$

та множина функцій цих платформ:

$$FRP = \{fRp_l\}, l = \overline{1, mFRP}. \quad (5)$$

В результаті утворюється логічна матриця L, елементи якої формуються наступним чином:

$$L_{ij} = \begin{cases} 1, \text{якщо функція } j \text{ входить до платформи } i; \\ 0, \text{якщо функція } j \text{ не входить до платформи } i. \end{cases}$$

Таким чином можуть бути виділені платформи, які в необхідній мірі відповідають умовам розробки додатку, а також ті платформи, які в достатній мірі близькі до цих умов. Формується матриця сумісності:

$$CPL = \{clp_{in}\}, i = \overline{1,k};\ n = \overline{1,kRP}, \tag{6}$$

елементи якої показують, чи може бути побудований репозиторій сервісів на базі конкретної хмарної платформи або є необхідність в консолідації різних хмарних платформ шляхом створення віртуальної платформи, що поєднує функції різних сервісів у відповідності до функціональних вимог додатку, що розробляється:

$$CRP_{in} = \begin{cases} 1, \text{якщо платформа } n \text{ дозволяє реалізувати сервіс } i; \\ 0, \text{платформа } n \text{ не дозволяє реалізувати сервіс } i. \end{cases}$$

Проводимо об'єднання матриці проектів і матриці платформ. При цьому з декартового добутку множин P та RP видаляються варіанти, які не відображають вимоги до сумісності:

$$\acute{P} = \{(p_i, Rp_n) | cRp_{in} = 1\}, i = \overline{1,k};\ n = \overline{1,kRP}. \tag{7}$$

Множина функцій, які можуть надавати сервіси, об'єднується:

$$\acute{F} = F \cup FRP. \tag{8}$$

У відповідності до отриманих множин $\acute{P}$ та $\acute{F}$ формується матриця варіантів реалізації хмарного сервісу $\acute{VR}$, вигляд якої наведено в таблиці 1.

Враховуючи отримані результати перейдемо до опису моделі програмного забезпечення. Будемо вважати, що програмний продукт, який побудовано з урахуванням варіантів реалізації хмарного сервісу $v_i \in \acute{VR}$, надає інтерфейс для виклику та виконання чітко визначенної предметної функціональності. Тоді для умов розподіленого середовища надання сервісів програмний продукт може бути представлений записом типу:

$$E: I \times \Psi \times R \times \acute{VR} \times O \times \Theta, \tag{9}$$

де E – функція виконання ПЗ, яке використовує хмарні сервіси.

I – множина вхідних даних предметної області;

Ψ – конфігурація запуску додатку;

O – множина вихідних даних;

Θ – множина службових вихідних даних;

Таблиця 1 – Об’єднана матриця варіантів реалізації хмарного сервісу

Варіант реалізації хмарного сервісу SaaS		Функції сервісу			Функції хмарної платформи PaaS		
проект сервісу	хмарна платформа	f_1	…	f_m	frp_1	…	frp_n
P_1	RP_1						
P_1	RP_5						
P_2	RP_3						
P_2	RP_5						
P_0							

Фундаментальним поняттям в рамках SOA є сервіс. Виділяють два види сервісів: атомарні та комплексні. Атомарні сервіси не містять інформацію про структуру та виконуються у вигляді транзакції виду: $S = \{S_1, S_2, ..., S_n\}$. Комплексні сервіси складаються з декількох атомарних сервісів, об'єднаних у відповідності з певною структурою (сценарієм). Застосування комплексних сервісів дозволяє об'єднувати наявні сервіси на базі SOA та (або) створювати нові види послуг.

Аналіз методів динамічного формування комплексних сервісів показує широкі можливості в управлінні рівнем QoS для різних видів послуг за рахунок зміни складу комплексних сервісів. Однак існуючі методи управління складом комплексного сервісу орієнтовані лише на формування сервісу з необхідною функціональністю без врахування обмежень на якість обслуговування, що негативно впливає на характеристики мультисервісного додатку. З метою підвищення ефективності існуючих систем управління мультисервісною мережею, а також враховуючи постійне зростання вимог користувачів до рівня забезпечення QoS послуг у мережі, виникає необхідність в удосконаленні методів управління послугами в мультисервісних SOA додатках.

2.2 Мікросервісний підхід до рефакторінгу існуючих додатків

Цей підхід передбачає розбиття існуючих додатків на мікросервіси відповідно до потреб замовника, що доповнює можливості створення абсолютно нових мікросервісів (рис. 2.2). Потім мікросервіси можуть бути повторно використані у різних контекстах, і це означає, що і для різних комунікаційних потреб. Відокремлення транспортної логіки від логіки мікросервісу є успішною практикою підтвердження важливого значення контексту мікросервісів. При побудові логіки мікросервісу розробник не повинен мікувати про те, як мікросервіс взаємодіє з кінцевою точкою, якою може бути сервер підприємства (XML / SOAP), сервіс хмари (XML / HTTP), мобільний пристрій (JSON / HTTP) або пристрій IoT (протокол TCP низького рівня, MQTT, або, можливо, навіть власний протокол).

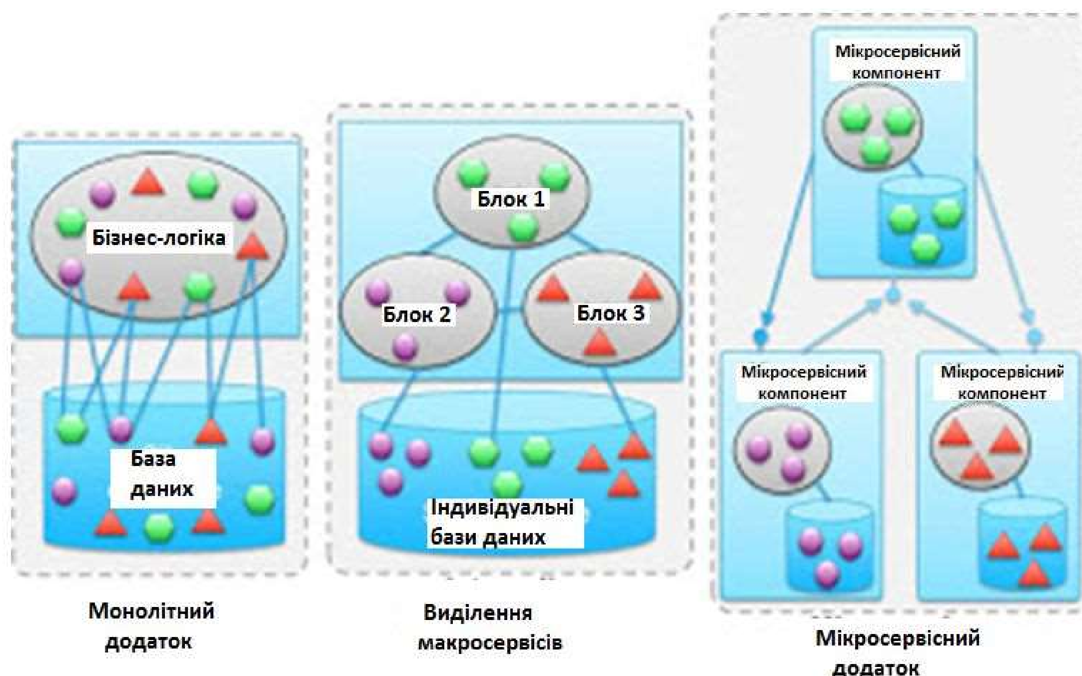

Рисунок 2.2 – Перехід від монолітних додатків до мікросервісів

Розробник додатку також повинен бути в змозі виявити і використовувати інші мікросервіси, які можуть бути опубліковані через сервісний шлюз (API gateway) [12-16]. Шлюз забезпечує споживання сервісів, забезпечує їх

масштабування і надійність і дозволяє повторно їх використовувати у різних контекстах без змін.

Сервісний шлюз робить мікросервіси доступними. Він використовує відкриті стандарти, такі як SAML, Kerberos, OAuth, WS- * або XACML – залежно від вимог [14]. Крім того, розробникам потрібен простий спосіб відкрити необхідних сервісів та їх контрактів. Зазвичай для надання каталогу сервісів й інформації про їх контракти використовується портал самообслуговування.

У той час, як SOAP підтримується кількома фреймворками та інструментами протягом багатьох років, фреймворк API Swagger стає стандартом за замовчуванням для реалізації, виявлення і тестування REST сервісів. Він також інтегрований у багатьох проміжних ПЗ, або middleware [14].

Коли мова йде про відкриття сервісів, то згадують про API, Open API і API керування. Термін API раніше мав на увазі низькорівневі інтерфейси, реалізовані у програмному коді. В останні роки термін змінив значення, і зараз під ним маються на увазі прості інтерфейси, побудовані поверх протоколу HTTP. Зазвичай це еквівалентно REST-інтерфейсам, які надають дані в форматі JSON (іноді – XML) і використовують команди HTTP: PUT, GET, POST і DELETE для опису дій "створити", "прочитати", "змінити" і "видалити". Ці протоколи більш підходять для таких мов програмування, як JavaScript, яка активно використовується для здійснення запитів до API.

Координація взаємодії мікросервісів може здійснюватися різними засобами: зі збереженням стану (stateful) або без збереження (stateless), з орієнтацією на повідомлення сервісів або на події, що виникають. Координація взаємодії без збереження стану є найкращою практикою для одного сервісу у більшості випадків, конкретна координація композитного сервісу може працювати краще з урахуванням стану процесу. Графічний інструмент може бути використаний як для створення окремого мікросервісу, так і для створення композитних сервісів.

2.3 Контейнери

Контейнер є програмним забезпеченням для автоматизації розгортання й управління сервісами у середовищі віртуалізації на рівні операційної системи. Він дозволяє «упакувати» сервіс з усім його оточенням і залежностями в контейнер, який може бути перенесений на будь-яку Linux-систему, а також надає середовище з управління контейнерами.

Контейнеризація за своєю суттю реалізується на рівні віртуалізації операційної системи (на відміну від віртуальних машин (VM), кожна з яких оснащена повною вбудованою ОС). Контейнери легко упаковані, легкі і призначені для роботи в будь-якому місці. Кілька контейнерів можуть бути розміщені в одній віртуальній машині.

Контейнери та мікросервіси не те ж саме. Мікросервіс може працювати у контейнері, але він також може працювати і на виділеній VM. Проте, контейнери є хорошим способом розроблення і розгортання мікросервісів, згрупованих на певні композиції, а інструменти і платформи для запуску контейнерів є хорошим способом для управління мікросервісними додатками [16-18].

Контейнери пропонують новий спосіб реалізації SOA, який є більш ефективним у багатьох відношеннях. За допомогою контейнера можна запустити кожен сервіс (частину програми) всередині контейнера. Контейнери є кращими будівельними блоками для SOA ніж традиційні сервіси з наступних причин [14, 16]:

– контейнери легко переміщаються між хостами (хмарами, вузлами). При використанні традиційних SOA сервіси залежать від конкретних умов хостів. Наприклад, якщо мережевий файл файлової системи встановлюється на сервері CentOS Linux для надання сервісу зберігання даних для SOA, зміна хоста NFS на сервер Ubuntu потребує значних зусиль. На відміну від цього, переміщення контейнера від одного хоста до

іншого значно простіше, оскільки контейнерне середовище хоста завжди однаково.

– контейнери мають високу масштабованість. Оскільки контейнери додатків можуть легко переміщатися між вузлами, контейнерна інфраструктура також легко масштабується шляхом додавання екземплярів певного сервісу, повязаних спільною базою даних. Масштабування в традиційних розподілених системах не реалізується так просто.
– контейнери легко оновити. Хочете оновити додаток, що працює в контейнері? Просто треба відновити зображення контейнера, на якому заснований додаток, а потім перезапустити контейнер. Цей процес є більш простим і безвідмовним у порівнянні з традиційним оновленням сервісу в розподіленої системі. Так же просто відмінити зміни, якщо це необхідно.
– контейнери можуть бути використані повторно. Однією з основних цілей SOA є повторне використання сервісів, їх розділення і зменшування кількості надлишкових сервісів. Контейнери просто використовувати повторно, оскільки зображення контейнерів може бути легко скопійовані і контейнери можуть швидко переміщатися між вузлами.

До складу програмних засобів входить сервер контейнерів, клієнтські засоби, що дозволяють з інтерфейсу командного рядка управляти зображеннями контейнерів, а також API, що дозволяє в стилі REST управляти контейнерами програмно.

Сервер забезпечує повну ізоляцію контейнерів, що запускаються на хості, при цьому кожен контейнер має доступ тільки до прив'язаному до нього мережевого простору імен і відповідним віртуальним мережевим інтерфейсам. В дійсності використовуються кілька хостів для того, щоб створити легко доступну систему контейнерів, коли запускаються багато екземплярів контейнерних зображень, розповсюджуваних між кількома хостами, з'єднаними через мережу.

Набір клієнтських засобів дозволяє запускати процеси в нових контейнерах, зупиняти і запускати контейнери, припиняти і відновлювати процеси в

контейнерах. Набір команд дозволяє здійснювати моніторинг запущених процесів. Нові зображення контейнера можливо створювати зі спеціального сценарного файлу, можливо записати усі зміни, зроблені в контейнері у нове зображення. Крім того, в інтерфейсі командного рядка вбудовані можливості взаємодії з публічним репозиторієм, в якому розміщені попередньо зібрані зображення контейнерів [5, 20].

Навіть якщо контейнери не мають вбудованих ОС, одна спільна ОС, як і раніше, необхідна. Будь-яка стандартна ОС буде працювати, в тому числі Linux або Windows. Проте, фактично необхідні ресурси ОС, як правило, обмежені, тому звичайної операційної системи може бути занадто. Це стало причиною розробки спеціальних контейнерних операційних систем, таких як Rancher OS, CoreOS, VMware Photon, Ubuntu Snappy, Red Hat Project Atomic та Microsoft Nano Server [19, 20].

2.4 Інструментарій для створення веб-сервісів

Перехід до загального репозитарію з безліччю альтернативних варіантів мікросервісів від різних провайдерів, у якому відкриття необхідних мікросервісів буде виконуватися автоматично за запитом користувача, потребує, перш за все, узгодження описів мікросервісів як веб-сервісів з семантичною інформацією.

При оновленні контенту такого репозитарію можна скористатися чи реєстром сервісів FIWARE , побудованих для Інтернету речей (IoT) [16], чи напрацюваннями провідних компаній з розроблення інструментарію для створення сервіс-орієнтованих додатків.

В процесі поширення використання SOA до цього типу додатків сформувався окремий перелік вимог:

– використовують декларативний формат опису процесу встановленні та налаштування додатку, що зводить до мінімуму витрати часу та ресурсів для нових розробників, задіяних у проекті;

– узгоджуються з сучасними хмарними платформами, що забезпечує легке розгортування;

– мінімізують розбіжність між середовищем розробки і середовищем виконання, що дозволяє використовувати безперервне розгортання для отримання максимальної гнучкості;

– можуть масштабуватися без суттєвих змін в інструментах, архітектурі та способі розробки.

Веб-додаток – це сукупність статичних і динамічних веб-сторінок, тобто наперед створених сторінок і програм, що створюють веб-сторінки у відповідь на звернення користувача.

Зазвичай при створенні веб-додатків використовують архітектуру клієнт-сервер. Користувач дає запит веб-додатку, веб-додаток його обробляє та відправляє її через мережу до веб-сервера. Сервер відповідає на запит формуючи веб-сторінку і відправляє її назад веб-додатку також мережею. Вебдодаток в свою чергу вже відображає надану сервером інформацію користувачеві. Зазвичай веб-додаток використовує браузер, як інтерфейс користувача.

Веб-сервіс – це клієнт-серверний додаток, в якому клієнтом виступає браузер, а сервером – веб-сервер. Вони дозволяють відвідувачам сайту відправити і отримати дані з бази даних або внести нову інформацію в базу через Інтернет з використанням веб-браузеру. Дані, представлені для користувача в браузері як інформація, генеруються динамічно (в певному форматі, наприклад, в HTML з використанням CSS) у веб-додаток за допомогою веб-сервера.

Логіка веб-додатку розподілена між сервером і клієнтом, зберігання даних здійснюється, переважно, на сервері, обмін інформацією відбувається по мережі. Однією з переваг такого підходу є той факт, що клієнти не залежать від конкретної операційної системи користувача, тому веб-додатки є сервісами, що не залежать від конкретної платформи.

Отже, веб-додаток – це комп'ютерна програма, яка працює в браузері. Тому, для доступу до програми необхідні браузер та мережа Інтернет. Зберігання та обробка інформації при такій організації обчислень відбувається на віддаленому сервері, а веб-переглядач служить програмою-клієнтом і призначеним для користувача інтерфейсом. Схему роботи веб сервісу представлено на рисунку 2.3.

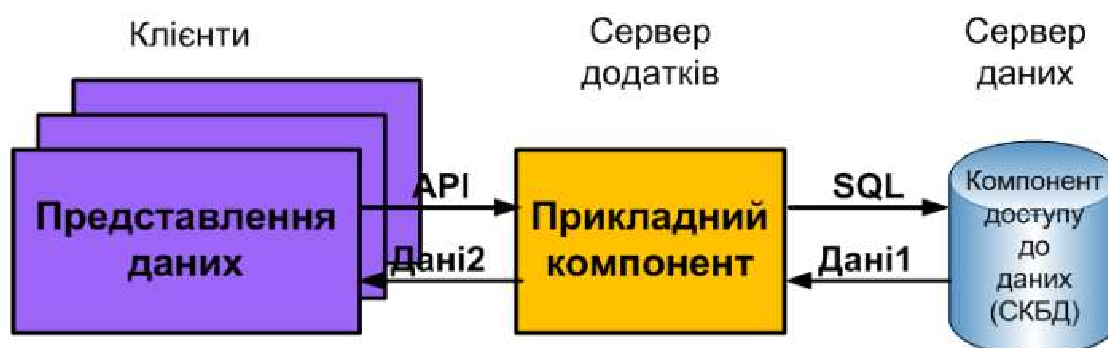

Рисунок 2.3 – Схема роботи WEB-додатку

Розглянемо, які існують способи керувати станом веб-додатку. Оскільки відбувається спілкування клієнта і сервера, то і контекст ділиться на клієнтський і серверний. Далі перераховані способи збереження і відновлення контексту виконання або по-іншому – стану сеансу роботи веб-додатку на стороні клієнта і на стороні сервера.

Контекст виконання на стороні клієнта можна зберігати:

– в оперативній пам'яті програми клієнта, тобто інтернет браузера. Це дуже розумно, оскільки використовується оперативна пам'ять клієнта, а не сервера. З іншого боку, не всі дані є можливість зберігати на стороні клієнта, оскільки не всі дані можна перетворити в текстовий формат і передати на сервер;

– у невеликих фрагментах текстових даних, які зберігаються на стороні клієнта – cookies. Cookies зберігаються в текстових файлах, у розділах, виділених операційною системою для зберігання різної інформації користувача. Ці дані передаються кожен раз з сервера в заголовках HTTP запиту. Окремим недоліком цього способу є те, що прийом cookies може бути заборонений клієнтом.

Контекст виконання на стороні сервера можна зберігати:

– в області оперативної пам'яті, що виділяється веб-сервером (Apache, IIS). Ці дані доступні з усіх сторінок веб-додатки всім його користувачам;

– в області оперативної пам'яті, яка називається стан сеансу. На відміну від стану програми ця область виділяється окремо для кожного користувача і зберігається протягом сеансу його роботи з веб-додатком (від моменту переходу на сторінку програми та до моменту закриття останньої сторінки);

– у структурах баз даних. Це найбільш універсальний і надійний спосіб зберігання контексту програми. Універсальний, бо не залежить від природи веб-сервера, а надійний, тому що життєвим циклом стану програми та стану сеансу управляє веб-сервер, в той час, як інформацію в базі даних контролює веб-додаток.

Ще одним досить надійним способом зберігання даних стану сеансу є використання елементів HTML розмітки. Додаток на стороні клієнта може силами JavaScript і DOM створити приховані елементи з прихованими полями, про існування яких буде знати тільки сервер. Сервер, у свою чергу, обробить ці дані і в них же помістить відповідь, якщо це необхідно. Елементам веб-додатка також доводиться спілкуватися між собою, просто в їхньому випадку проблема стоїть не так гостро, оскільки їх спілкування відбувається на відстані не в рамках одного процесу. З метою організації обміну інформацією між клієнтом і сервером досить часто використовують XML, який чудово підходить для опису чого завгодно, якщо це «що завгодно» має яскраво виражену структуру.

XML поміщають в тіло HTTP запиту і відправляють на сервер. Мова XML – це універсальний будівельний матеріал, придатний для вирішення абсолютно різних завдань, тому на його базі створюються спеціалізовані мови – протоколи мережевої взаємодії, які надалі стають загальноприйнятими стандартами. До таких протоколів слід віднести Web Services Description Language (WSDL) – мова опису інтерфейсів веб-сервісів і Simple Object Access Protocol (SOAP) – протокол обміну структурованими повідомленнями між компонентами розподіленої інформаційної системи [2].

На сьогодні все ще популярним є підхід до розробки веб-додатків, що має назву Ajax. При використанні Ajax сторінки веб-додатки не перезавантажуються

цілком, а лише довантажують необхідні дані з сервера, що робить їх більш інтерактивними і продуктивними. Також дуже популярні “Single Page Applications”, де кожен запит оброблюється на стороні клієнту замість запиту на сервер. Незалежно від мови, якою написана серверна частина веб-додатки, способи обробки запитів та взаємодії з користувачем залишаються ті ж. Основна мова, якою описується графічний інтерфейс веб-додатку – це HTML. Ця мова описує структуру веб-сторінки. Художнє оформлення веб сторінок описується таблицями стилів – CSS [1].

Традиційний підхід до розробки та використання веб-орієнтованих додатків вимагає від розробників багато витрат часу на їх встановлення, розгортання та налаштування. Ця проблема майже зникає, коли користувач орієнтований на використання розподіленого сервіс-орієнтованого програмного забзпечення, яке створюється на основі окремих предметно-орієнтованих сервісів. Поширення таких додатків здійснюється за допомогою бізнес-моделі SaaS, в рамках якої провайдер надає користувачеві в аренду через мережу Інтернет додаток, який функціонує на його технічній платформі. При цьому в якості таких додатків можуть розглядатися як окремі сервіси, так і композитний додаток, що виконує їх агрегацію.

Найбільш поширеним варіантом розробки сервіс-орієнтованого програмного забезпечення передбачає попередню розробку програмного коду або використання вже існуючого коду з метою перетворення його у програмний сервіс з подальшою генерацією усіх складових сервісу, а вже потім інтегрувати отриманий сервіс в програмний додаток. З цією метою під час розробки SOA-додатку необхідно:

– створити додаток, що реалізує функціонал сервісу;

– виконати розгортання сервісу на сервері та протестувати його;

– здійснити генерацію WSDL опису сервісу;

– виконати анотування сервісу;

– створити клієнтський додаток для доступу до сервісу.

Такий підхід майже непридатний для проектування великого програмного проекту, оскільки нераціональним є розробка програмного продукту без

урахування та проектування загальної функціональності системи починаючи з рівня реалізації окремих сервісів.

Однією з технологій, яка дозволяє будувати сервіс-орієнтовані додатки, є Windows Communication Foundation (WCF), відома через .NET Framework [17]. WCF реалізує сучасні галузеві стандарти WS* для сумісності з веб-сервісами. WCF дозволяє використовувати SOAP, JSON, XML і т.і.

ASP.NET – це єдина модель розробки веб-додатків корпоративного класу з мінімальними зусиллями, яка включає сервіси, необхідні для розробника. ASP.NET є частиною .NET Framework, тому при написанні додатків розробник має доступ до класів в .NET Framework. Він може розроблювати свої додатки на будь-якій мові програмуванні, сумісній з CLR (Common Language Runtime), у тому числі Microsoft Visual Basic і C#.

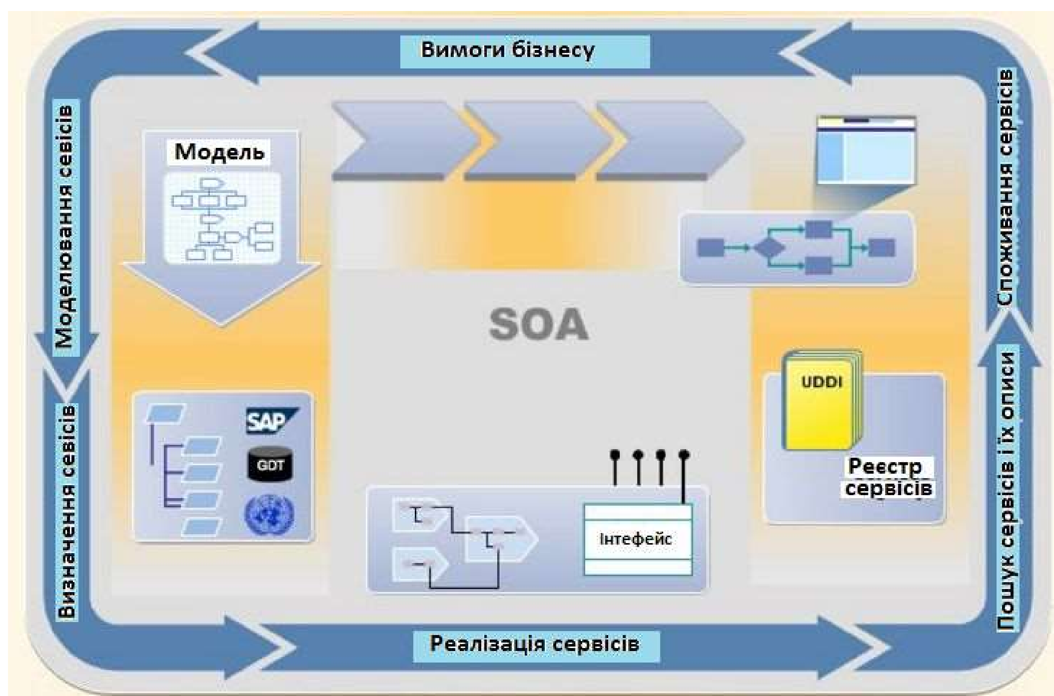

Рисунок 2.4 – Технология Enterprise SOA

Enterprise SOA компанії SAP (рис.2.4) включає не тільки інструментарій для створення веб-сервісів, але і керовані моделі для побудови сервісів і додатків [18]. На відміну від інших провайдерів SAP не лише надає інструменти розробки, а й створює підтримуючу екосистему, що включає сертифікацію, партнерську мережу і співтовариство корпоративних сервісів.

Комплект Oracle SOA Suite 12c [16] дозволяє створювати і впроваджувати сервіс-орієнтовані архітектури (SOA), а також керувати ними. Компоненти пакета використовують ряд загальних функцій, серед яких узгоджений інструментарій, єдина модель впровадження та управління, комплексні функції безпеки і єдиний центр контролю метаданих.

Простий у використанні, орієнтований на багаторазове використання, єдиний інструментарій розробки додатків та підтримка управління життєвим циклом дозволяють суттєво знизити витрати і складність розробки.

Інтегроване рішення для управління та обслуговування програмними комплексами HP SOA Center містить усі елементи, необхідні для успішного створення і експлуатації сервіс-орієнтованої архітектури на підприємстві [7]. З його допомогою можна побудувати і обслуговувати сервіс-орієнтовану архітектуру на підприємстві, ефективно управляти нею і підтримувати необхідні рівні якості і безпеки. HP SOA Center містить усі необхідні інструменти для керівництва та управління на всіх етапах впровадження архітектури SOA, починаючи з тестового запуску і до повного її розгортання.

2.5 Вимоги до програмної системи

В рамках написання атестаційної роботи ставиться за мету перевірка можливість реалізації веб-орієнтованої програмної системи SOA, яка використовує мікросервісний підхід для обробки та агрегації сервісів на вимогу користувача.

Кінцевий програмний продукт повинен задовольняти ряду вимог:

– система повинна передбачати отримання даних з певних джерел;

– інформація має надходити до кінцевої точки незалежно від місцезнаходження джерела (мобільність);

– кількість процесів отримання інформації, що виконуються одночасно повинні обмежуватись лише апаратними засобами системи (робота з багатьма джерелами);

– використання технології «клієнт-сервер»;

– отримання даних та первинна обробка здійснюються на клієнті;

– основна обробка та аналіз даних виконується на сервері;

– для збереження даних використовується СУБД;

– клієнт повинен мати можливість приймати отримувати дані незалежно від наявності зв’язку із сервером;

– система має передбачати інтерфейс для перегляду оброблених та проаналізованих даних, адміністрування та моніторинг протікаючих в системі процесів.

Таким чином, метою програмної реалізації системи в рамках виконання атестаційної роботи магістра, є застосування SOA-методології роботи з хмарними сервісами для проектування віртуалізованого десктопу для доступу до сервісів, які надає провайдер послуг. Перейдемо до опису проектних рішень, що були прийняті під час розробки системи.

3 ПРОЕКТНІ РІШЕННЯ

3.1 Рішення, що підтримують розробку SOA додатків

Веб-сервіси є технологію реалізації концепції дизайну SOA. Існує велика кількість досліджень, присвячених вимогам надійності SOA і, більш конкретно, веб-сервісів. Веб-спільнота розробила ряд специфікацій, які підтримають надійний обмін повідомленнями, управління транзакціями, реплікації і безпеку. Сервіси взаємодіють один з одним за допомогою шаблону синхронного запиту і відповіді, що забезпечує щільне зчеплення між компонентами системи (рис. 3.1). Тобто реалізується зв'язок «один-до-одного» – один з конкретних сервісів викликається у часі тільки одним із споживачів, але зв'язок є двонаправлений.

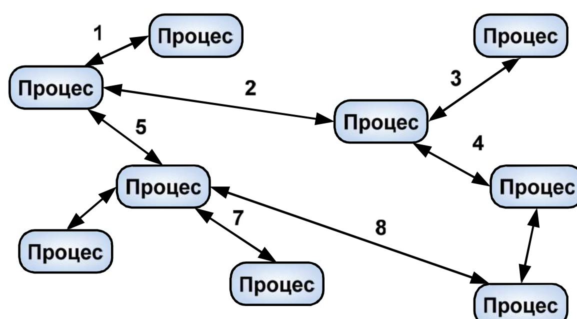

Рисунок 3.1 – Архітектура SOA-додатку, керована запитами

Орієнтація на сервіси та SOA виправдовується краще тоді, коли процеси або їх частини є стандартними, коли вони часто повторюються без змін або коли декільком користувачам потрібний той самий компонент процесу для виконання своїх завдань. Виклик (відкриття) сервісів у SOA реалізується за допомогою віддаленого виклику процедури RPC (Remote Procedure Calls) на вимогу споживача сервісів.

У шаблоні «запит-відповідь» вся логіка маршрутизації повідомлень розташована на кожній кінцевій точці, і сервіси можуть безпосередньо спілкуватися між собою. Підтримка синхронізації даних між мікросервісами стає набагато складнішою у випадку мікросервісних додатків, оскільки кожен

мікросервіс має свої власну базу даних, яка може бути доступна або модифікована тільки за допомогою API REST (рис. 3.2).

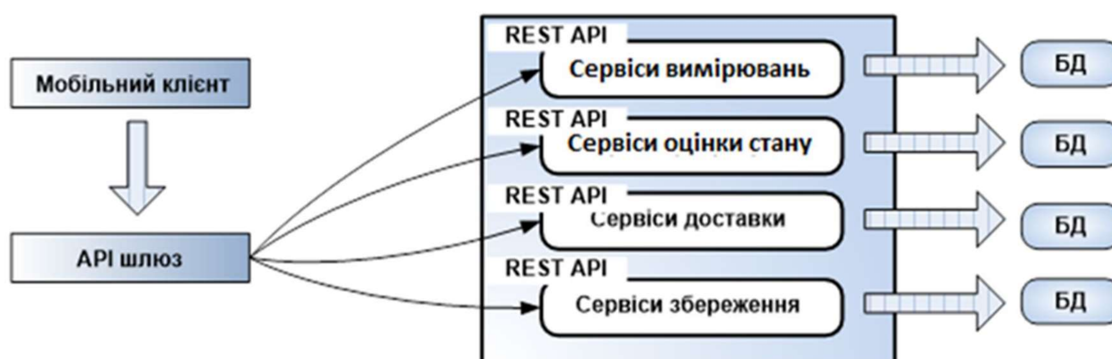

Рисунок 3.2 – Спілкування мікросервісів через API-шлюз

Очевидно, що ця модель придатна для порівняно простих додатків, заснованих на мікросервісах, оскільки зі збільшенням кількості сервісів вона стане більш громіздкою і складною в обслуговуванні. Тому для випадку складних мікросервісних додатків замість шаблону «запит-відповідь» можна використовувати API шлюз в якості єдиної точки входу у систему, побудовану на базі мікросервісів.

Створюючи прикладну програму, яка використовує мікросервіси, потрібно визначитися з організацією взаємодії мікросервісів. Коли сервіс-орієнтована архітектура використовує схему оркестрування для взаємодії сервісів, то між сервісами встановлюється зв'язок «запит-відповідь». Тобто один сервіс запитує API іншого сервісу, в результаті чого створюється мережа шляхів зв'язку між усіма сервісами. Інтеграцію, зміну або видалення сервісів із цієї мережі здійснити важко, оскільки необхідно знати про кожне з'єднання між сервісами.

Оркестрування – це традиційний спосіб взаємодії між різними сервісами у сервіс-орієнтованої архітектурі (SOA) [13-15]. Для реалізації оркестрування, як правило, є один контролер, який виступає у ролі "диригента" (оркестратора) загальної взаємодії сервісів за шаблоном «запит / відповідь» (рис.3.3). Наприклад, якщо потрібно викликати у певному замовленні три сервіси, оркестратор звернеться до кожному з них, чекаючи відповіді, перш ніж викликати наступний.

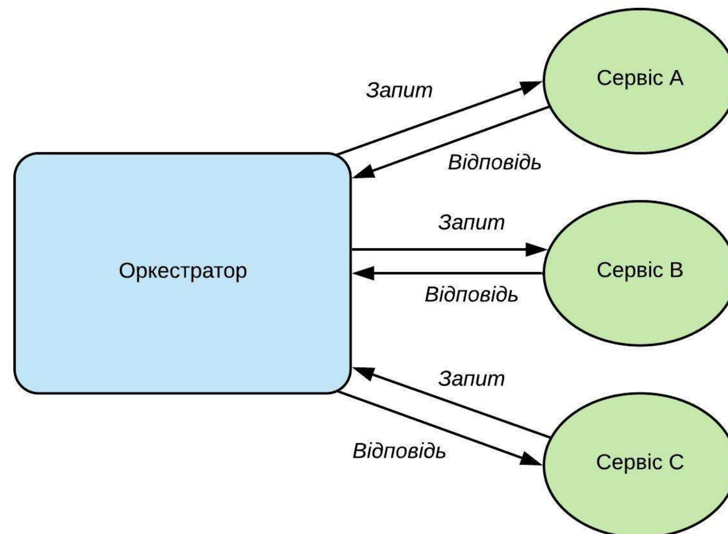

Рисунок 3.3 – Взаємодія сервісів за шаблоном типу «запит / відповідь»

До платформи розробки мікросервісних систем можна сформувати такі вимоги:

– платформа повинна абстрагуватися від архітектурних паттернів сервіс-орієнтованих систем і допомагати розробникам створювати код;

– містити хороші моделі метаданих для опису можливостей мікросервісів;

– підтримувати синхронні (загальнодоступні API) та асинхронні виклики (внутрішні події), обмін повідомленнями, включаючи підтримку протоколів, таких як черга, «публікація - підписка» тощо;

– підтримувати декілька мов реалізації мікросервісів та еластичні режими роботи;

– підтримувати збереження даних;

– мати захищений шлюз API;

– забезпечувати автоматизацію розгортання з підтримкою інструментів для контейнерів та їх оркестрування.

На підставі проведеного порівняльного аналізу хмарних платформ, які підтримують розробку та розгортання додатків з використанням мікросервісів різних платформ та які мають достатню обчислювальну еластичність, для проведення експериментів з розробки та експлуатації програмної системи, яка

розробляється в рамках атестаційної роботи магістра, було обрано Google Cloud Platform.

Веб-сервіс з інтеграції окремих сервісів в SOA додаток з використанням технології контейнерного завантаження мікросервісів повинен мати багатошарову структуру.

Були виділені наступні шари програми: база даних, класи-сутності, DAO-класи, реалізації DAO-класів, класи-сервіси, реалізації класів-сервісів, які працюють із DAO-класами, контролери, які працюють із сервісами, інтерфейс користувача.

Проект реалізовано з використанням таких вимог:

– мова програмування Java;

– веб сервіс має бути крос-платформеним, тобто працювати у середовищі найпоширеніших операційних систем;

– система управління базами даних – MySQL Server;

– використання ORM-системи Hibernate;

– каркас для розробки клієнт серверних додатків – Spring Boot;

– використання технології jsoup для роботи із основним контентом;

– у якості веб-серверу використовувати Apache Tomcat;

– для створення веб сторінок використовувати технологію динамічного формування коду HTML;

– збірка проекту за допомогою технології Maven;

– ведення системного журналу за допомогою технології log4j;

– використання технології mockito для тестування контролерів додатку.

3.2 Вибір технологій програмування для реалізації проекту

За час існування WWW зміст веб-додатків, функції які вони виконують, принципи і архітектура їх побудови зазнали значних змін. З найпростіших засобів

збереження HTML сторінок вони «виросли» до готових рішень, що орієнтовані на підтримку роботи корпоративними інформаційними системами[4, 20].

3.2.1 AJAX (Asynchronous JavaScript and XML).

Технологія описує підхід до побудови інтерфейсу користувача веб-додатків, при якому відповідь на кожен запит користувача не вимагає перезавантаження кожної сторінки браузера, а лише оновлює чи додає нові дані, що йому необхідні [13]. Оскільки не потрібно кожен раз оновлювати сторінку цілком, підвищується швидкість роботи з сайтом та зручність його використання. При цьому запити та їх відправка на сервер формуються і надсилаються у фоновому режимі, а відповіді завантажуються в готову веб-сторінку, оновляючи лише її частинку. Такий підхід дозволяє розробляти зручні веб-інтерфейси для користувачів на тих сторінках сайтів, де необхідна активна взаємодія з користувачем. Асинхронність цього процесу дозволяє користувачеві переглядати контент веб-сторінки під час обробки запиту на сервері.

Класична модель роботи AJAX наступна: користувач завантажує веб-сторінку, натискає на ній будь-який елемент, і браузер відправляє відповідний запит на сервер. Сервер формує лише ту частину веб-сторінки, яка змінилась, та надсилає її у відповідь. Результатом є змінена певна частина веб-сторінки без її повного перезавантаження [4].

AJAX – це концепція використання суміжних певних технологій.

Головною складовою технології AJAX є використання JavaScript –об'єктно-орієнтованої мови програмування, яка дозволяє динамічне завантаження коду з використанням DOM у форматі JSON [5].

Разом з тим слід зазначити, що сторінки на AJAX погано індексуються. Тому фахівці рекомендують передбачити на сайті можливість отримання динамічно довантажувати інформації безпосередньо з посиланнями.

3.2.2 DHTML (Dynamic HTML)

Динамічний HTML отримав свою назву саме за рахунок динамічної зміни змісту сторінки. Ця технологія розробки веб-сайтів розглядає HTML-документ як об'єктну структуру, яка використовує поєднання статичної мови розмітки HTML, вбудованої скриптової мови JavaScript (сценарії виконуються на стороні клієнта), CSS (каскадних таблиць стилів) та DOM (об'єктної моделі документа). Також технологія DHTML використовується для реалізації елементів навігації та для додання інтерактивності формам веб-додатків [22].

Використання XMLHttpRequest дозволяє браузеру формувати API-запит для звернення до сервера у фоновому режимі за протоколом HTTP. Саме це дозволяє не перезавантажувати сторінку повністю. Запит такого формату можна використовувати як для синхронного, так і для асинхронного обміну інформацією в довільному текстовому форматі (наприклад XML, JSON, HTML). Застосування XMLHttpRequest створює уявлення «миттєвої» відповіді серверу у порівнянні з класичними методом перезавантаження усієї сторінки для оновлення представленої на ній інформації.

3.2.3 SPA (Single Page Application)

Популярна технологія розробки веб-додатоків або веб-сайтів, яка використовує єдиний HTML-документ як оболонку для всіх веб-сторінок. За технологією SPA взаємодія між користувачем та сайтом реалізується з використанням DHTML, CSS, JavaScript, зазвичай через AJAX. Односторінкові додатки нагадують звичайні додатки, з тією лише різницею, що виконуються в рамках браузера, а не у власному процесі в середовищі операційної системи [23].

Основними елементами, які використовуються при побудові SPA, є:

– фреймворки для JavaScript, зокрема MVC та MVVM-фреймворки;

– роутінг: навігація між уявленнями (view) проводиться у фронтенді;

– шаблонізатор;

– HTML5;

– API для бекенду, наприклад, в стилі REST;

– Ajax.

Програми, що складаються з однієї сторінки, можуть використовувати стан (state) із зовнішнього джерела (тобто URL-адреси) або спостерігати внутрішній стан додатку. Внутрішній стан SPA обмежений, оскільки існує лише одна точка «входу" у додаток. Одиночний вхід означає, що ви завжди починаєте з кореня додатку при заході у програму. Під час навігації в додатку для внутрішнього використання немає зовнішнього представлення з боку URL-адреси. Якщо ви хочете поділитися певним вмістом, інша особа, яка завантажує додаток, починає роботу з кореня, тому вам доведеться пояснити, як дістатись до потрібного міста у додатку, виконуючи послідовні дії.

За допомогою зовнішнього джерела, у вигляді URL-адреси, в SPA на основі місцезнаходження ви можете поділитися посиланням та бути впевненими, що будь-хто, хто відкрив це посилання, побачить те ж саме, що і ви, тому що під час переміщення місцезнаходження завжди оновлюється (якщо вони мають однакову авторизацію для перегляду вмісту)

Хоча URL-адреса в адресному рядку (рис. 3.3) – це те, з чим користувачі взаємодіють і бачать, SPA використовує браузерний об’єкт window.location. Це дає змогу взаємодіяти з різними частинами URL-адреси, не маючи самостійного аналізу.

Тільки три властивості об'єкту розташування важливі для SPA: шлях до імені (pathname), хешу (hash) та пошуку (search - зазвичай це називається рядком запиту). В односторінкових додатках навігація за будь-яким місцезнаходженням за межами програми виконується регулярно, тому імена хостів і протоколів можуть бути проігноровані.

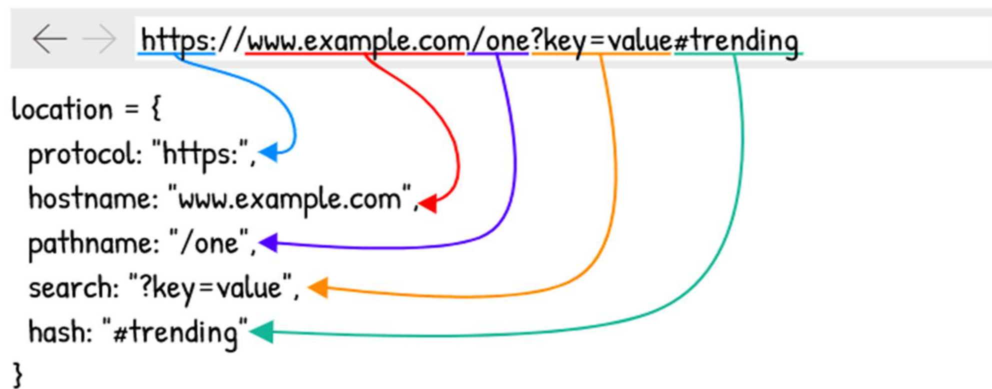

Рисунок 3.4 – Властивості window.location

Шлях до імені, як правило, є найбільш важливою з цих трьох властивостей, оскільки він використовується для визначення того, який вміст потрібно відтворити. Пошуки та хеш більш корисні для завантаження додаткових даних.

Принципи, за якими фреймворк реалізує парадигму SPA, повинні дотримуватися наступних понять і визначень:

– SPA підтримує клієнтську навігації. Всі «переходи» користувача по модулям сторінки фіксуються в історії навігації, причому навігація при цьому є «глибокою», тобто якщо користувач скопіює та відкриє посилання на внутрішній модуль-сторінку в іншому браузері або вікні, він потрапить на відповідну сторінку;

– SPA розміщується на одній web-сторінці, тому всі необхідні для роботи сайту (порталу) скрипти і стилі повинні бути визначені в одному місці проекту – на єдиній web-сторінці;

– SPA зберігає постійно стан (важливі змінні) у роботі клієнта (клієнтського скрипта) в кеші браузера або у Web Storage;

– SPA завантажує всі скрипти, які необхідні для старту програми, під час ініціалізації web-сторінки.

Модульний підхід при побудові SPA дозволяє в будь-який момент масштабувати програму. Клієнтська та серверна частини в SPA є повністю незалежними, що дає змогу ізолювати їх один від одного. Тому зміни у серверній частині ніяким чином не впливають на клієнтську частину.

3.2.4 AngularJS

Це JavaScript-фреймворк з відкритим вихідним кодом. Він призначений саме для розробки односторінкових додатків. Його головна мета – розширення браузерних додатків, що побудовані на основі MVC-шаблону, а також спрощення тестування та розробки веб-орієнтованих додатків. AngularJS спроектований з переконанням, що декларативне програмування найкраще підходить для побудови призначених для користувача інтерфейсів і опису програмних компонентів, в той час як імперативне програмування відмінно підходить для опису бізнес-логіки. Фреймворк адаптує і розширює традиційний HTML, щоб забезпечити двосторонню прив'язку даних для динамічного контенту, що дозволяє автоматично синхронізувати модель і уявлення. В результаті AngularJS зменшує роль DOM-маніпуляцій і покращує її тестованість [24].

Основна мета фреймворку:

– відділення DOM-маніпуляції від логіки додатків, що покращує якість тестування коду;

– відношення до тестування як до важливої частини розробки;

– розділення процесу розробки на клієнтську та серверну, що дозволяє вести їх паралельну розробку;

– проведення замовника через весь шлях створення програми: від проектування бізнес-логіки інтерфейсу користувача до тестування.

Angular дотримується MVC-шаблону проектування і заохочує слабкий зв'язок між поданням, даними і логікою компонентів. Використовуючи впровадження залежності, Angular переносить на клієнтську сторону такі класичні серверні служби, як видо-залежні контролери, що зменшує навантаження на сервер і веб-додаток стає легшим. Двостороннє зв'язування даних в AngularJS є найбільш примітною особливістю і зменшує кількість коду, звільняючи сервер від роботи з шаблонами. Замість цього шаблони відображаються як звичайний HTML, наповнений даними, що містяться в області видимості, яка визначена в моделі.

Програмний сервіс $scope в Angular стежить за змінами в моделі і змінює розділ HTML-виразів в поданні їх через контролер. Крім того, будь-які зміни в поданні відображаються в моделі. Це дозволяє обійти необхідність маніпулювання DOM і полегшує ініціалізацію і прототипування веб-додатків.

3.2.5 Node.js

В основі Node лежить JavaScript [25], причому це майже та сама мова, яка використовується для розробки сценаріїв на боці клієнта. Node.js використовує модель, яка базується на подіях та неблокуючому вводі-виводі, що робить його легким та ефективним. Пакетна екосистема Node.js, npm, є найбільшою у світі екосистемою бібліотек з відкритим кодом. Перш за все Node.js використовується як серверна платформа для роботи з JavaScript. За допомогою Node можна писати повноцінні додатки. Node.js вміє працювати з зовнішніми бібліотеками, викликати команди з коду на JavaScript і виконувати роль веб-сервера.

Для забезпечення обробки великої кількості паралельних запитів у Node.js використовується асинхронна модель запуску коду, заснована на обробці подій в неблокуючому режимі та визначенні обробників зворотніх викликів (callback). Як способи мультиплексування з'єднань підтримується epoll, kqueue, /dev/poll і select. Для мультиплексування з'єднань використовується бібліотека libuv, для створення пулу потоків (thread pool) задіяна бібліотека libeio, для виконання DNS-запитів у неблокуючому режимі інтегрований c-ares. Всі системні виклики, що спричиняють блокування, виконуються всередині пула потоків і потім, як і обробники сигналів, передають результат своєї роботи назад через неіменовані канали (pipe).

Node.js відмінно підходить для створення не тільки API, а й стандартних веб-застосунків. Вона має потужні інструменти для задоволення веб-розробників.

3.2.6 Express.js

Express.js, або просто Express — програмний каркас розробки веб-застосунків для Node.js, реалізований як вільне і відкрите програмне забезпечення під ліцензією MIT. Він спроектований для створення веб-застосунків і API. Де-факто є стандартним каркасом для Node.js. Це мінімалістичний та гнучкий веб-фреймворк надає широкий спектр функцій для мобільних та веб-застосунків. Маючи в своєму розпорядженні значну кількість службових методів HTTP та проміжних оброблювачів, допомагає швидко і легко створювати надійний API. Express надає тонкий шар фундаментальних функцій веб-додатків, які допомагають працювати з функціями Node.js.

Він забезпечує механізми:

– написання обробників для запитів з різними HTTP дієсловами за різними шляхами URL (маршрутами);

– інтеграції з движками візуалізації (view) для створення відповідей серверу, шляхом вставки даних у шаблони;

– встановлення загальних параметрів веб-додатків, таких як порт для підключення, розташування шаблонів, які використовуються для відображення відповідей;

– додавання запитів обробки «проміжного програмного забезпечення» в будь-який момент в рамках обробки запиту до серверу.

Хоча Express є досить мінімалістичним, розробники створили сумісні пакети програмного забезпечення для вирішення майже будь-якої проблеми веб-розробки. Є бібліотеки для роботи з файлами cookie, сеансами, входами користувачів, параметрами URL-адреси, даними POST, заголовками безпеки та багато інших.

Express є бекендом для програмного стека MEAN, разом з базою даних MongoDB і каркасом AngularJS для фронтенду.

3.2.7 Passport.js

Це додаткове програмне забезпечення для перевірки автентичності для Node.js. Надзвичайно гнучкий і модульний, Passport може бути інкапсульований до будь-якого веб-додатку. Комплексний набір стратегій підтримує автентифікацію за допомогою імені користувача та паролю.

Passport надає можливість вибору з більш ніж 140 способів автентифікації. Він дозволяє реалізувати автентифікацію за допомогою локального/віддаленого екземпляра бази даних або використовувати SSO (single sign-on – технологія єдиного входу) за допомогою провайдерів через OAuth (відкритий стандарт авторизації, який дозволяє користувачам відкривати доступ до своїх приватних даних – фотографії, відео, списки контактів –, що зберігаються на одному сайті без необхідності вводу імені користувача та паролю) для Facebook, Twitter, Google та ін. для автентифікації користувачів за допомогою їх акаунтів у соціальних мережах. Під час розробки модулів інкапсуляція є якістю, тому Passport делегує всі інші функціональні можливості основній програмі. Таке відокремлення проблем залишає код чистим та підтримуваним, і робить Passport надзвичайно простим для інтеграції у програму.

У сучасних веб-програмах автентифікація може приймати різні форми. Традиційно користувачі входять, вказуючи ім'я користувача та пароль. Зі зростанням впливу соціальних мереж, єдиний вхід за допомогою постачальника OAuth (Google, Facebook або Twitter) став популярним методом автентифікації. Служби, які виставляють API, часто вимагають захист доступу за допомогою облікових даних на основі токенів.

Passport визнає, що кожна програма має унікальні вимоги до автентифікації (рис. 3.4). Механізми автентифікації, відомі як стратегії, упаковуються як окремі модулі. Програми можуть вибирати, які стратегії використовувати, не створюючи непотрібних залежностей.

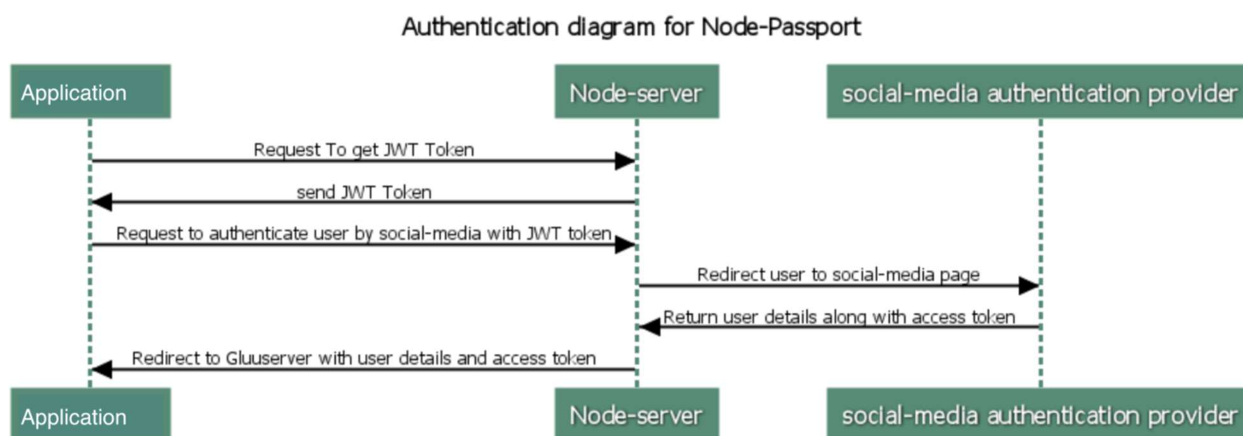

Рисунок 3.5 – Автентифікація з використанням Passport.js

Passport надає тільки механізм для реалізації процедури автентифікації і покладає відповідальність за реалізацію підтримки сесій на розробника програмного забезпечення.

3.2.8 ASP (Active Server Pages)

Ця технологія дозволяє створювати динамічні веб-сторінки HTML, використовуючи об’єктну модель інтерфейсу, створеного на основі ISAPI (Internet Server Application Programming Interface) розширень і фільтрів. Принцип, що закладений в основу інтерфейсу полягає в тому, що на веб-сторінці знаходяться фрагменти коду, який інтерпретується веб-сервером, на якому встановлене розширення ISAPI, і він надає користувачу вже готовий результат виконання вибраних фрагментів коду. Спеціальний «динамічний» тег <%...%> вказує, що в ньому знаходиться потрібний код.

Мови програмування можна використовувати практично будь-які. По замовчуванню підтримуються JavaScript і VBScript.

3.2.9 ISAPI (Internet Server Application Programming Interface)

Інтерфейс до сервера Інтернету фірми Microsoft, призначений для програмного керування сервером, набув значного поширення. ISAPI підтримується більшістю виробників програмних засобів. ISAPI-програми є спеціальним видом додатків, що обробляють запити користувачів та відображають їх у вигляді потоків HTML, які надходять безпосередньо у браузер клієнта.

Додатки ISAPI завантажуються і виконуються в адресному просторі IIS, тому у сервера не має потреби створювати новий процес при кожному HTTP-запиті. Оскільки Windows завантажує динамічно підключаєму бібліотеку лише один раз при першому виклику функції в DLL, то додаток ISAPI залишається завантаженим і не видаляється, поки не буде зупинений або вимкнений веб-сервер.

3.2.10 JSP (Java Server Pages)

Технологія створення веб-додатків, що базується на однократній компіляції Java-коду (сервлету) при першому зверненні до нього з подальшим виконанням методів цього сервлету і розміщенням отриманих результатів в набір даних, які відправляються в браузер [3]. Технологія JavaServer Pages (JSP) дозволяє змішувати звичайний, статичний HTML із динамічно генерованим вмістом сервлетів. Частина JSP-сторінки складається зі звичайного HTML-коду, яка передається клієнту без змін. Частини, які генеруються динамічно, відмічені спеціальними HTML-подібними тегами.

3.2.11 Мова програмування Java

Об'єктно-орієнтована мова програмування Java широко використовується для створення клієнт-серверних додатків та забезпечується набором стандартних бібліотек, що забезпечують функціональність від стандартного вводу та виводу мережевих протоколів до графічних користувацьких інтерфейсів [26].

Надійність і безпека Java значно полегшує створення програмного забезпечення. Сама мова спроектована таким чином, щоб виробляти у програміста звичку писати «правильно». Модель роботи з пам'яттю, в якій виключено використання вказівників, робить неможливими цілий клас помилок, характерних для C та C++.

Компілятор Java використовує байт-коди, тобто модулі-додатки мають архітектурно-незалежний формат, який може бути проінтерпретований на багатьох різноманітних платформах.

Схема роботи системи і набір байт-кодів віртуальної машини Java такі, що дозволяють досягти високої продуктивності на етапі виконання програми:

– аналіз кодів на дотримання правил безпеки проводиться один раз до запуску кодів на виконання, а в момент виконання таких перевірок вже не потрібно, тому коди виконуються максимально ефективно;

– робота з базовими типами максимально ефективна. Для операцій з ними зарезервовані спеціальні байт-коди;

– методи в класах не обов'язково зв'язуються динамічно;

– автоматичний збирач сміття працює окремим фоновим потоком, що не сповільнює основну роботу програми, але в той же час забезпечує своєчасне повернення вільної пам'яті в систему;

– стандарт передбачає можливість написання критичних по продуктивності ділянок програми в машинних кодах [6].

3.2.12 ORM-система Hibernate

ORM (від англ. Object-relational mapping) – це відображення об'єктів будь-якої об'єктно-орієнтованої мови в структури реляційних баз даних. Саме об'єктів, таких, якими вони є, з усіма полями, значеннями, відносинами один до одного.

ORM-рішенням для мови Java, є технологія Hibernate, яка не тільки дбає про зв'язок Java класів з таблицями бази даних (і типів даних Java в типи даних SQL), але також надає можливості для автоматичної побудови запитів і отримання даних та може значно зменшити час розробки, який зазвичай витрачається на ручне написання SQL і JDBC коду. Hibernate генерує SQL виклики і звільняє розробника від ручної обробки результуючого набору даних і конвертації об'єктів, зберігаючи додаток переносним в усі бази даних SQL типу [27].

Hibernate забезпечує прозору підтримку збереження даних, тобто їхньої персистентності (persistence) для “POJO”-об'єктів, тобто для звичайних Java-об'єктів. Єдина сувора вимога до класу, що зберігається – стандартний конструктор без параметрів. Для коректної поведінки у деяких застосуваннях потрібно приділити особливу увагу до методів equals() і hashCode().

Mapping Java класів з таблицями бази даних здійснюється за допомогою конфігураційних XML файлів або Java анотацій. При використанні файлу XML, Hibernate може генерувати скелет вихідного коду для класів тривалого зберігання. У цьому немає необхідності, якщо використовується анотація. Hibernate може використовувати файл XML або анотації для підтримки схеми бази даних.

Забезпечуються можливості з організації відношення між класами «один-до-багатьох» і «багато-до-багатьох». На додаток до управління зв'язками між об'єктами, Hibernate також може керувати рефлексивними асоціаціями, де об'єкт має зв'язок «один-до-багатьох» з іншими примірниками свого власного типу даних.

Hibernate підтримує відображення користувацьких типів значень. Це робить можливим такі сценарії:

– перевизначення типу SQL, який Hibernate вибирає при відображенні стовпчика властивості;

– картування перерахованого типу Java до колонок БД, так, ніби вони є звичайними властивостями;

– картування однієї властивості в декілька колонок.

Колекції об'єктів даних, як правило, зберігаються у вигляді колекцій Java-об'єктів, таких як набір (Set) і список (List). Підтримуються узагальнені класи (Generics), введені в Java 5. Hibernate може бути налаштований на «ледачі» (відкладені) завантаження колекцій. Відкладені завантаження є стандартним варіантом, починаючи з Hibernate 3.

Зв'язані об'єкти можуть бути налаштовані на каскадні операції. Наприклад, батьківський клас, Album (музичний альбом), може бути налаштований або на каскадне збереження, або видалення свого нащадку Track. Це може скоротити час розробки і забезпечити цілісність додатку. Функція перевірки зміни даних (dirty checking) дозволяє уникнути непотрібного запису дій в базу даних, виконуючи SQL оновлення тільки при зміні полів персистентних об'єктів.

Hibernate забезпечує використання SQL-подібної мови Hibernate Query Language (HQL), яка дозволяє виконувати SQL-подібні запити, записані поряд з об'єктами даних Hibernate. Запити критеріїв надаються як об'єктно-орієнтована альтернатива до HQL [8]. Hibernate може використовуватись як у самостійних програмах Java, так і в програмах Java EE, що виконуються на сервері (наприклад, сервлети чи EJB session beans). Також він може включатись як додаткова можливість до інших мов програмування. Hibernate не тільки забезпечує автоматичне генерування Java-класів на основі таблиць БД (а також приведення базових типів Java до типів SQL), а й надає механізми формування запитів та вибірок даних. Також він може істотно знизити час на розробку, яка раніше виконувалася шляхом ручної роботи з даними із використанням SQL і JDBC.

3.2.13 Платформа Spring Boot

Spring boot – це утиліта для швидкого налаштування додатків, що пропонує конфігурацію для збірки Spring. Spring об'єднує широкий діапазон різних модулів таких, як spring-core, spring-data, spring-web (який включає в себе Spring MVC) і так далі. За допомогою цього можна вказати Spring, скільки з них потрібно використовувати, і отримати швидке налаштування для кожного з модулів фреймворку Spring

Spring MVC – це каркас для розробки додатків, який створений і використовується разом з популярною платформою J2EE. Він вкорочує час розробки і робить розробників більш продуктивними, забезпечуючи їх набором інструментів і компонентів для створення клієнт-сервених додатків.

Використання каркасу означає, що програмісту не потрібно витрачати час на створення всієї програми. Він може зосередитися на кодуванні бізнес-логіки та представлення (presentation layer) додатка, а не на частинах "верхнього рівня" таких як опис того як приймати вхідні дані від користувача або як генерувати списки, що випадають на веб-сторінці.

Інша перевага у використанні каркаса полягає в тому, що він дозволяє коду бути у великій мірі незалежним від платформи (як це і є в разі Spring MVC). І це може бути досягнуто в багатьох випадках навіть без рекомпіляції коду – один і той же веб-додаток (або ".war" файл) може просто бути скопійований з одного сервера на інший.

Особливо важлива перевага – це те, що каркас дає для розробників початкового рівня «місце для старту». Очевидно, що простіше мати основу для програми і розширювати її, ніж писати все з нуля. Ця якість Spring MVC зберігає програмістам дні або навіть тижні планування та розробки.

Spring MVC спирається на шаблон проектування Model-View-Controller (MVC, Модель-подання-поведінку). Шаблон MVC широко визнаний як один з

найбільш добре розроблених і зрілих шаблонів проектування які використовуються в даний час [28].

MVC – архітектура програмного забезпечення, в якій модель даних програми, інтерфейс користувача і керуюча логіка розділені на три окремих компонента, так, що модифікація одного з компонентів надає мінімальний вплив на інші компоненти.

Модель надає дані (зазвичай для View), а також реагує на запити (зазвичай від контролера), змінюючи свій стан. Представлення (View). Відповідає за відображення інформації, тобто інтерфейс користувача. Поведінка (Controller). Інтерпретує дані, введені користувачем, та інформує модель і представлення про необхідність відповідної реакції.

Схема паттерну MVC представлена на рисунку 3.5.

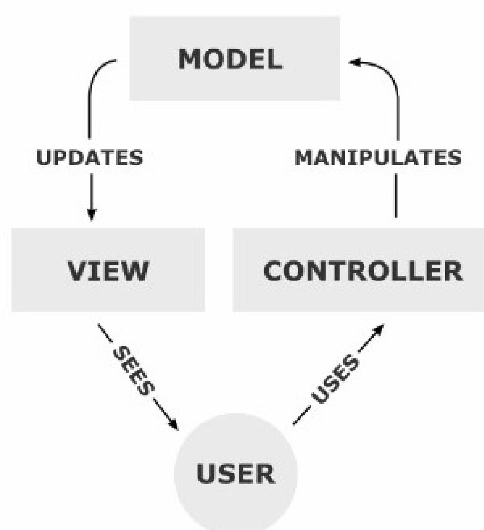

Рисунок 3.5 – Схема патерну проектування MVC

Важливо відзначити, що як представлення, так і поведінка залежать від моделі. Однак, модель не залежить ні від представлення, ні від поведінки. Це одна з ключових переваг подібного поділу. Вона дозволяє будувати модель незалежно від візуального представлення, а також створювати кілька різних представлень для однієї моделі.

Компоненти моделей надають «модель» для бізнес-логіки або дані для Spring програми. Наприклад, в Spring додатку, який управляє даними клієнтів, можливо було б вигідно створити компонент моделі “Customer”, який би надавав програмний доступ до інформації про клієнтів. Компоненти моделей, як правило, є стандартними Java класами. Для моделей не потрібно спеціального формату, тому можна використовувати код, написаний для інших проектів.

Компоненти view – це ті частини програми, які відповідають за презентацію інформації та отримання неї від користувача. У Spring MVC компонентам представлення відповідають веб-сторінки. Вони використовуються для представлення інформації, що зберігається в моделях. Зазвичай для кожної веб-сторінки в Spring MVC додатку є один або більше компонент представлення [29].

Компоненти представлення створюються за допомогою JavaServer Pages (JSP). Технологія JSTL пропонує розробникові велику кількість “JSP Custom Tags”, які розширюють звичайні можливості JSP і спрощують розробку компонентів типу представлення.

Компоненти контролю координують діяльність у додатку. Мається на увазі, наприклад, прийом даних від користувача і оновлення бази даних за допомогою компонента моделі або виявлення помилки back-end системою і її обробка. Контролер приймає дані від користувачів, приймає рішення про те, які компоненти моделей повинні бути оновлені і вибирає компоненти представлення, яким надсилається повідомлення, що сигналізує про необхідність демонстрації оновлених результатів.

Одна з головних переваг компоненту контролю полягає у тому, що вони дозволяють розробнику перенести код, який займається обробкою помилок, з JSP-сторінок в сам додаток. Це може значно спростити логіку в сторінках і полегшити їх розробку та обслуговування.

Компоненти контролю або поведінки в Spring MVC є Java класами і повинні бути створені за певними правилами. У контексті Spring їх зазвичай називають “Controllers”.

Spring MVC є фреймворком, орієнтованим на запити. У ньому визначені стратегічні інтерфейси для всіх функцій сучасної клієнт-серверної системи. Мета кожного інтерфейсу – бути простим і ясним, щоб користувачам було легко його заново імплементувати, якщо вони того побажають. MVC прокладає шлях до більш чистого front-end-коду. Всі інтерфейси тісно пов'язані із Servlet API. Цей зв'язок розглядається деякими як нездатність розробників Spring запропонувати для веб-додатків абстракцію більш високого рівня. Однак, цей зв'язок залишає особливості Servlet API доступними для розробників, полегшуючи все ж роботу з ним.

Вся логіка роботи Spring MVC побудована навколо DispatcherServlet, який приймає і обробляє всі HTTP-запити (з UI) і відповіді на них. Робочий процес обробки запиту DispatcherServlet'ом проілюстрований на рис.3.6.

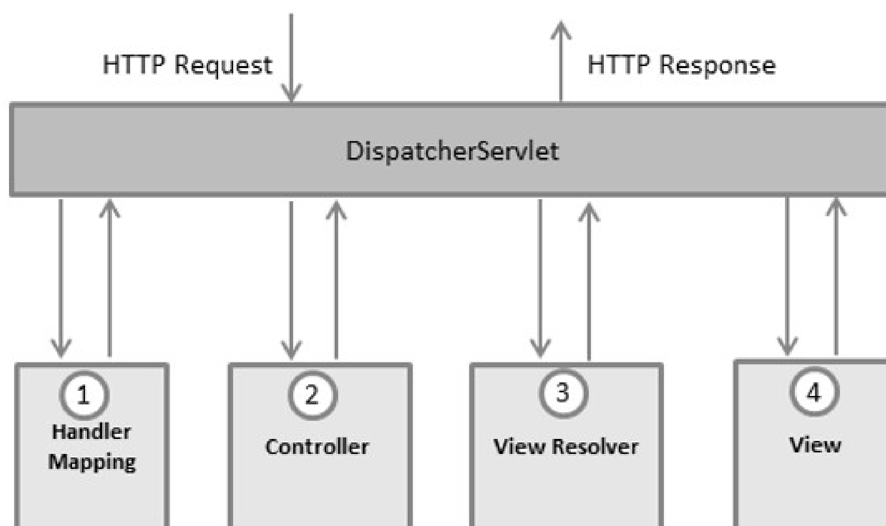

Рисунок 3.6 – обробки запиту DispatcherServlet'ом

Spring MVC надає розробнику наступні можливості: ясний і прозорий поділ між шарами в MVC і запитах. В ньому використовується стратегія інтерфейсів – кожен інтерфейс робить тільки свою частину роботи. Інтерфейс завжди може бути замінений його альтернативною реалізацією.

Інтерфейси тісно пов'язані із Servlet API. Також він реалізує високий рівень абстракції для веб-додатків. Технологія Spring дозволяє розробникам представляти і проектувати складні програми як послідовність щодо простих компонентів

моделей, представлення та контролю. Це веде до кращих, більш однорідних проектів, які також легше підтримуються [29, 30].

3.3 Архітектура системи що розробляється

Відповідно до мети дослідження необхідно розробити модель веб-орієнтованого програмного продукту, яка буде реалізовувати визначену користувачем функціональність за рахунок використання додатку SOA, який забезпечує доступ до сервісів хмари з використанням мікросервісів, розміщенних у власних контейнерах. З метою спрощення системи визначимо наступні обмеження:

– для розгортання будемо використовувати публічну хмару;

– система повинна дозволяти налагоджувати функції додатку незалежно для кожного з користувачів системи;

– питання, пов’язані з адміністрування додатку, в роботі не розглядаються.

Визначимось з користувачами системи.

EndUser – це користувач системи, що використовує програмну систему, працездатність якої забезпечується сервісами SaaS.

SaaSDev – це користувач системи та розробник сервісів, які можуть надавати послуги додатку від EndUser.

Application – каркасний додаток, що забезпечує звернення до сервісів SaaS.

Діаграму варіантів використання системи зображено на рис.3.7.

Архітектуру системи будемо розглядати як веб-сервіс з мікросервісною архітектурою. Вона буде складатися з основного хмарного сервісу та підключаемого репозіторію, в якому зберігаються сервіси.

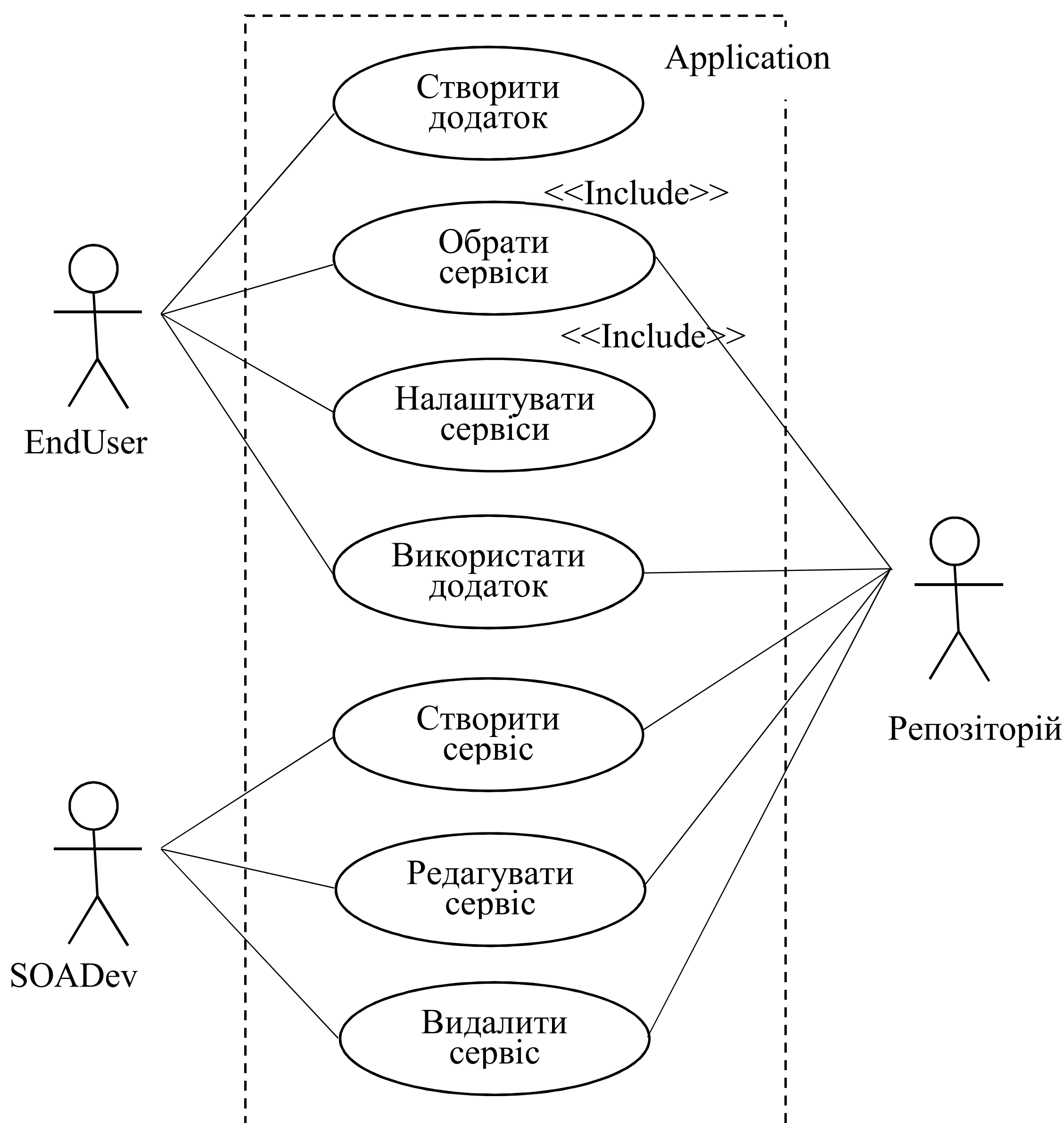

Рисунок 3.7 – Діаграма варіантів використання

Розглянемо архітектуру та функції системи більш детально. Для цього проаналізуємо узагальнену архітектуру системи, яка наведена на рис.3.8.

Компонент автентифікації користувачів – це компонент, який відповідає за ідентифікацію користувачів за допомогою логіна і паролю, відкритого SSH-ключа, а також за допомогою акаунтів із соціальних мереж.

Компонент проектів користувача реалізує взаємодію з користувачем. Він агрегує проекти користувачів та надає зовнішні інтерфейси як у вигляді веб-інтерфейсу, так і у вигляді програмного інтерфейсу, що базується на REST-запитах.

Сервіс зберігає данні про проект, його мету та конфігурацію для усіх користувачів. Для ідентифікації сервіс взаємодіє з мікросервісом автентифікації.

Компонент сервісів та їх атрибутів відповідає за збереження інформації про атрибути застосування та зареєстрованих в системі сервісах. Включає в себе класифікатор атрибутів сервісів та сервіс додатків. При запитах за атрибутами сервісів визначаються відповідні сервіси та необхідні параметри для них.

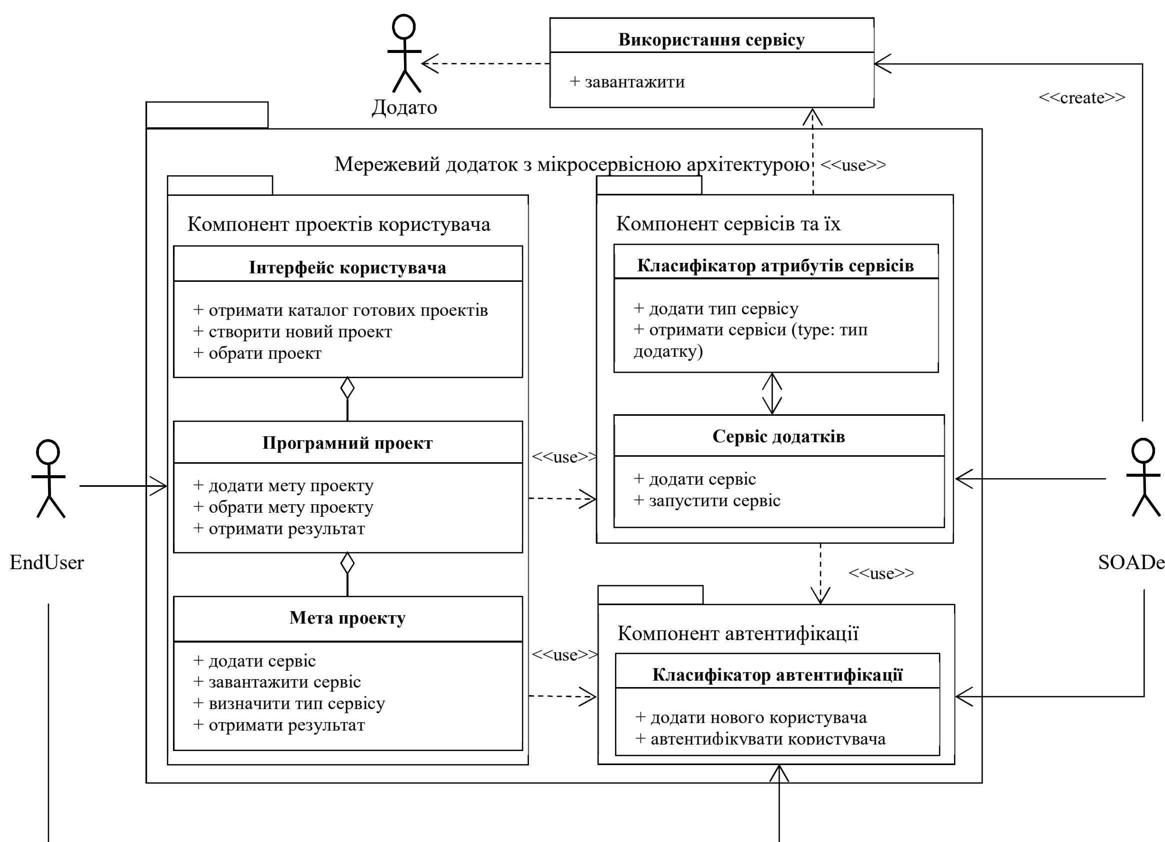

Рисунок 3.8 – Узагальнена архітектура системи

Мікросервіс при використанні сервісу приймає на вхід параметри сервісу та повертає результат. Використання сервісу інтегруються в систему за допомогою реєстрації в компоненті сервісів та їх використання.

Опишемо процес створення проекту додатку з урахванням мети дослідження:

– кінцевий користувач (EndUser) автентифікується в системі;

– кінцевий користувач (EndUser) створює новий проект додатку;

– кінцевий користувач (EndUser) робить опис додатку, вказуючи мету проекту, ім’я та опис;

– на підставі введених даних відповідно до існуючих в системі атрибутів додатка запитуються на їх відповідність сервісам, присутнім в системі;

– кінцевий користувач (EndUser) обирає саме ті атрибути додатка, які підходять для його мети розробки додатка;

– на основі обраних атрибутів компонент проектів користувача звертається до компоненту сервісів та робить запит на використання відповідних сервісів за вказаними атрибутами додатків;

– кінцевий користувач (EndUser) обирає та специфікує всі сервіси, вказуючи необхідні параметри для них: шлях до ресурсів, вихідні файли, обмеження за часом роботи тощо.

На рисунку 3.9 представлено UML-діаграму архітектури хмарної системи. Вона буде складатися з трьох основних компонентів: сервіс авторизації (Autorization Module), сервіс створення нових модулів (Widget Creator Module) та сервіс завантаження та конфігурації віртуального робочого столу (User Dashboard Module).

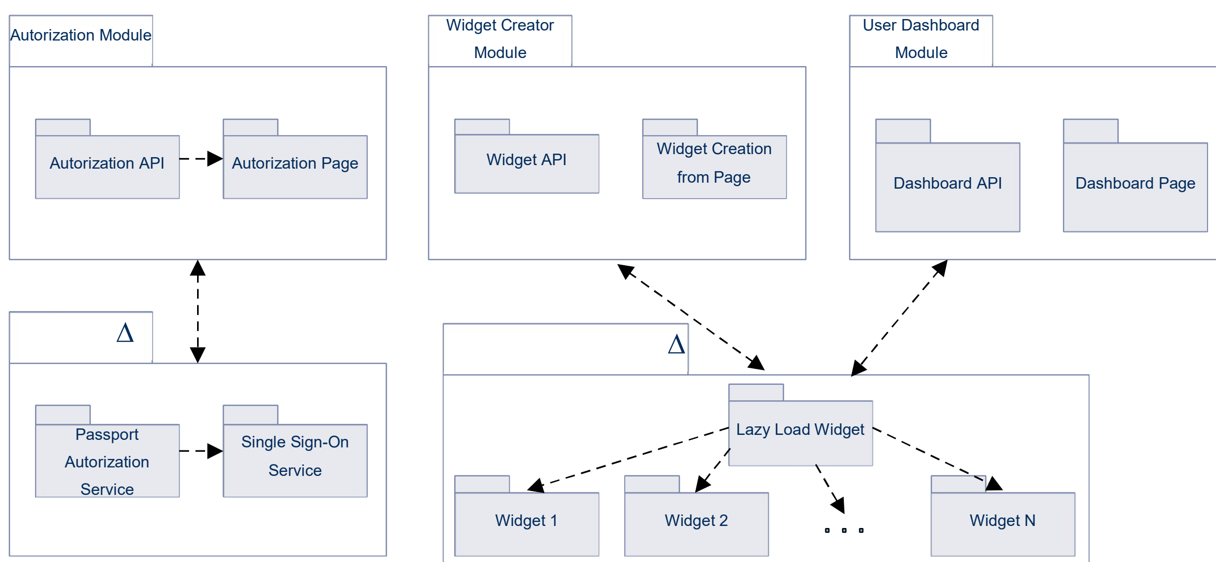

Рисунок 3.9 – UML діаграма архітектури системи

Методологія реалізації хмарних додатків SOA з використанням інтеграції окремих сервісів у програмний додаток є мульти-платформною, тобто не залежить

від принципів розгортання хмари. З метою перевірки працездатності проект буде реалізовано на Google Cloud Platform.

Google Cloud Platform (GCP) надає можливість працювати з різними інструментами для хостингу та обчислень. До складу GCP входять: Google App Engine, що надає сервіс PaaS для розміщення мобільних та веб-додатків, бекендів; Google Container Engine, який дозволяє за допомогою відкритої системи Kubernetes організувати контейнерні обчислення без урахування особливостей розгортання додатків та їх інтеграції в хмарне середовище розташування; Google Compute Engine, яка забезпечує надійну обчислювальну інфраструктуру та надає можливість налагоджувати, адмініструвати та проводити моніторинг систем. Контроль за доступом та видимість ресурсів, які працюють на платформі GCP, захищено моделлю безпеки Google.

Таким чином на підставі проведеного дослідження дійшли до висновку, що для реалізації функції приватної хмари в межах атестаційної роботи магістра буде обрано GCP. Для досягнення мети дослідження за допомогою Kubernetes було розгорнуто кластер “diploma” (територіально розташований у Центральній Америці) з наступними характеристиками:

– розмір кластеру: 3;

– типи віртуальної ЕОМ у кластері: f1-micro;

– обсяг оперативної пам’яті: 1 Гбайт;

– максимальна ємність диску: 50 Гбайт.

Образ клієнтського додатку було створено за допомогою Docker-файлу, після чого його було запушено на віртуальній машині, розгорнутій у кластері. Серверну частину додатку було розроблено з використанням технології Google Cloud. Вбудоване в GCP рішення NodeJs tools забезпечує увесь функціонал для швидкого розгортання та вивантаження працездатного коду на сервер .

Підводячи підсумки слід зазначити, що програмну реалізацію сервіс-інтегратора з використанням мікросервісних контейнерів було виконано у відповідності до мети дослідження. Перейдемо до перевірки працездатності системи та аналізу результатів дослідження.

4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ СИСТЕМИ

4.1 Розгортання проекту

Для розгортання проекту треба зупинитися на засобах розгортання. Docker випустила Docker Networking для включення віртуальних з'єднань між контейнерами та розробляє нові інструменти для підтримки життєвого циклу контейнерних додатків. Docker Universal Control Plane (UCP) забезпечує можливості створення, розгортання та управління додатками на вимогу, необхідні контейнери будуть копіюватися і додатки будуть побудовані з цих контейнерів, так що повне робоче середовище буде відтворене, в тому числі і самі контейнери, баланси навантаження, мережі тощо.

Сьогодні є багато постачальників контейнерів, але продукція фірми Docker лідирує на ринку, задовольняючи на 40% існуючий попит. Google давно використовує контейнери. Компанія є одним з основних вкладників у різні контейнеро-орієнтовані проекти з відкритим вихідним кодом, включали оркестратор Kubernetes і Google Container Engine, що функціонують в Google Cloud Platform, який і було обрано в якості платформи для реалізації проекту з розробки сервіс-орієнтованих додатків в приватній хмарі. Саме такий прототип каталогу було реалізовано в рамках атестаційної роботи магістра.

4.2 Програмний інтерфейс

Завантаження клієнтського додатку здійснюється шляхом відкриття відповідного URL (http://104.154.100.144/dashboard). Загалом клієнтський додаток представляє собою набір окремих модулів для відображення віртуальних сторінок. Головна сторінка додатку – сторінка авторизації, що з’являється при першому вході

в систему (рис.4.1). Для успішного входу в систему та початку роботи необхідно ввести коректні логін (в якості якого виступає електронна пошта) та пароль.

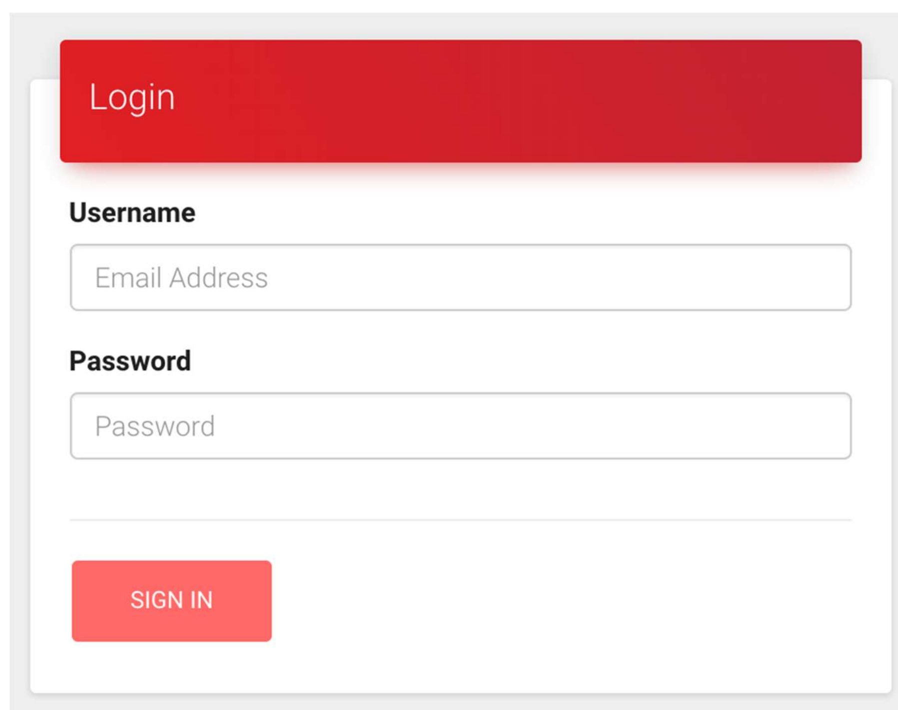

Рисунок 4.1 – Головне вікно входу в систему

Для підтвердження авторизації в системі використовується кнопка входу в систему, яка стає активною лише після заповнення полів логіну та паролю. Після введення необхідних даних та натискання кнопки «Увійти», на веб-сервер направляється POST-запит, що містить у собі об'єкти перевірки прав доступу. У разі присутності в системі інформації про користувача з вказаними валідними даними веб-сервер повертає у відповідь токен авторизації, що у подальшому зберігається у локальному веб-сховищі (local storage).

Після проходження авторизації користувач отримує доступ до головного екрану додатку, який має назву Dashboard (рис. 4.2). Головний екран складається з двох частин. Права частина інформаційна – вона відображає основну інформацію про віртуальну платформу, на який розгорнуто додаток, персональні дані користувача, залишок коштів на рахунку, статистику використання платформи та віджетів. Ліва частина екрану містить систему меню для роботи з додатком.

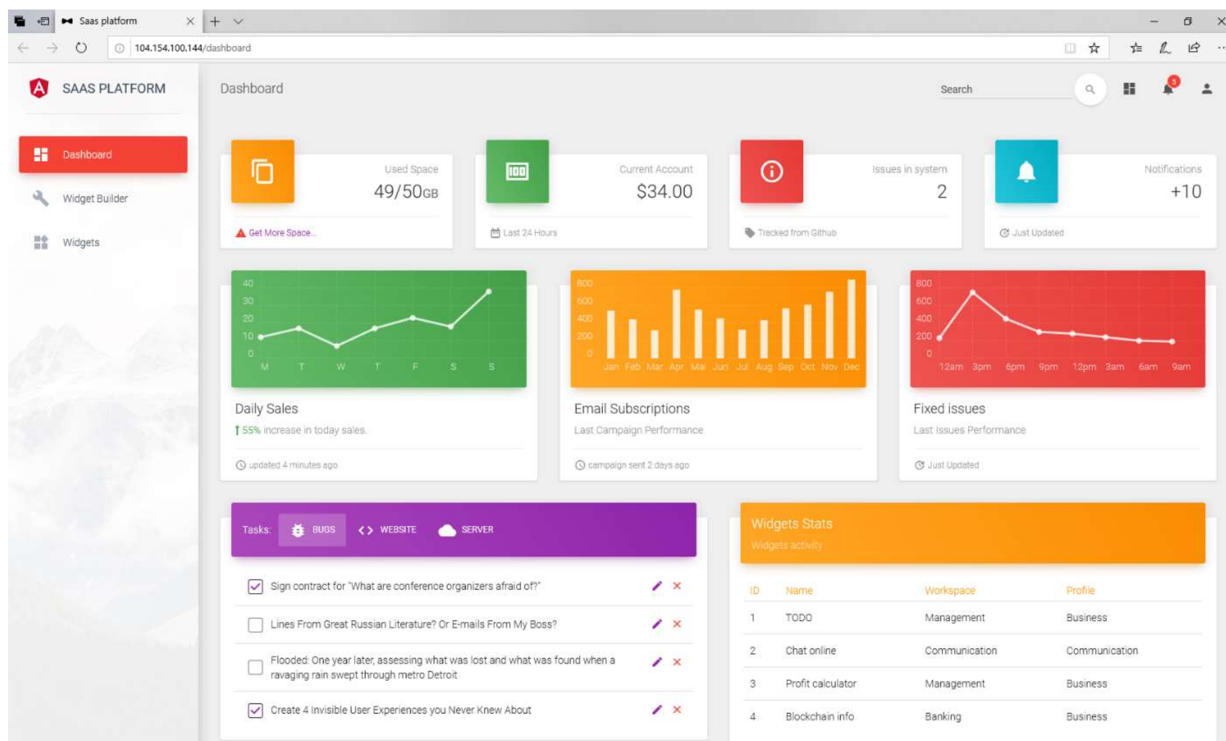

Рисунок 4.2 – Головний екран системи Dashboard

Користувач може користуватися навігацією для швидкого доступу до сторінок Dasboard, Widget creator, Widgets (рис. 4.3).

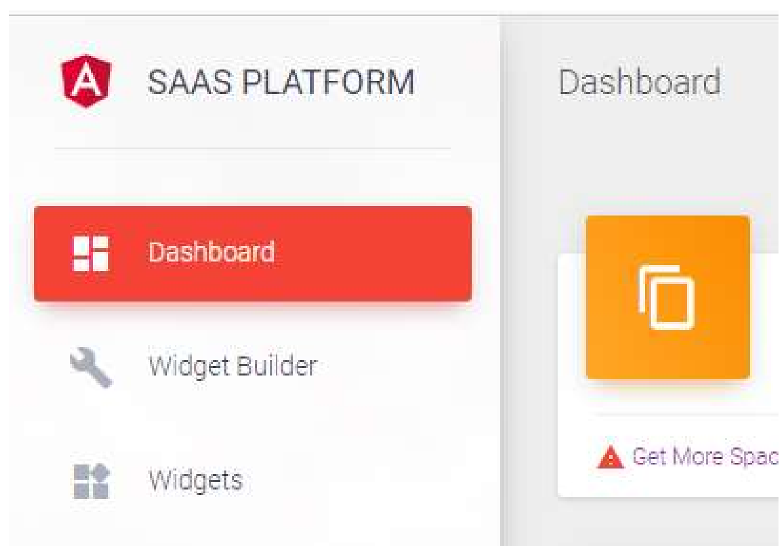

Рисунок 4.3 – Навігаційне меню додатку

Сторінка Widget builder призначена для завантаження в робочу область системи обраного віджету, який вже зареєстровано серед встановлених в глобальній області системи (рис.4.4), або для реєстрації на платформі нового віджету (рис.4.5). З цією метою користувач повинен заповнити відповідні форми, всі поля якої є обов'язковими для заповнення.

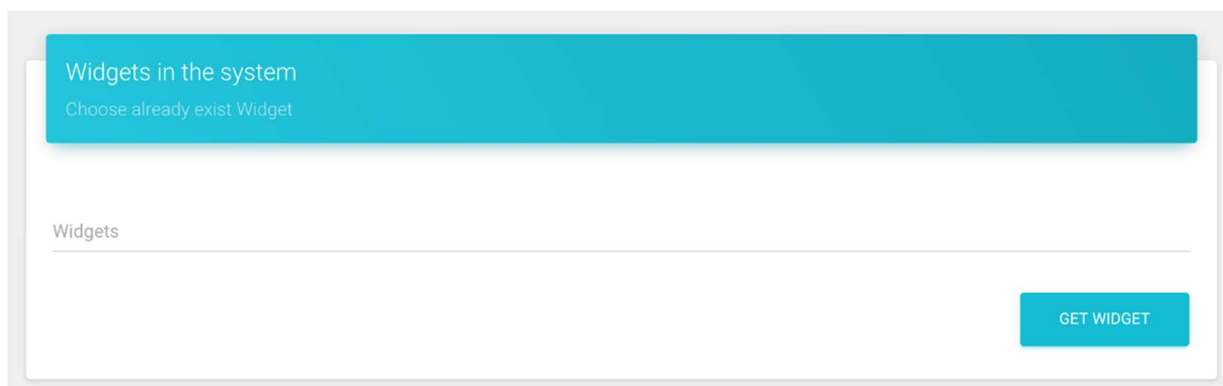

Рисунок 4.4 – Вибір віджету зі списку завантажених в систему

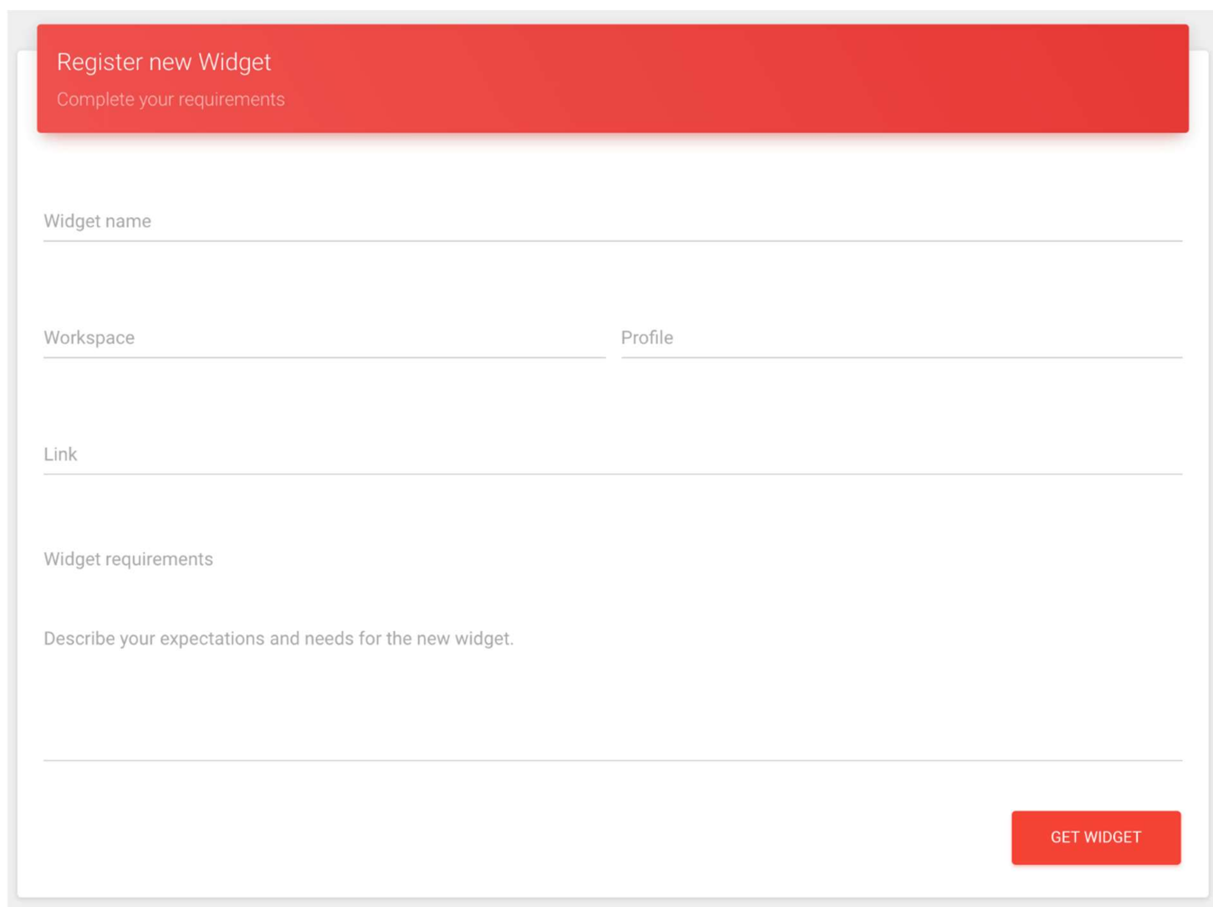

Рисунок 4.5 – Реєстрація в системі нового віджету

Сторінка Widget creator відображає кілька форм для реєстрації нового віджету в системі. Користувач має можливість заповнити форму самостійно для того щоб система підібрала йому потрібний віджет який не встановлено в глобальному контексті системи. Заповнюються поля з ім'ям нового віджету, його глобальна група та призначення, в останніх полях заповнюються URL-адреса на сторінку віджету в інтернеті чи описується його необхідний функціонал, який шукаємо. В останньому випадку адміністраторам системи доведеться самостійно запитувати реалізацію необхідного елемента у інших провайдерів сервісів.

При заповнені іншої форми (див.рис.4.4) користувачу пропонуються для вибору вже готові рішення віджетів, які встановлені в глобальному контексті системи. Ці віджети також додаються адміністратором системи та їх список зберігається на веб сервері – в базі даних.

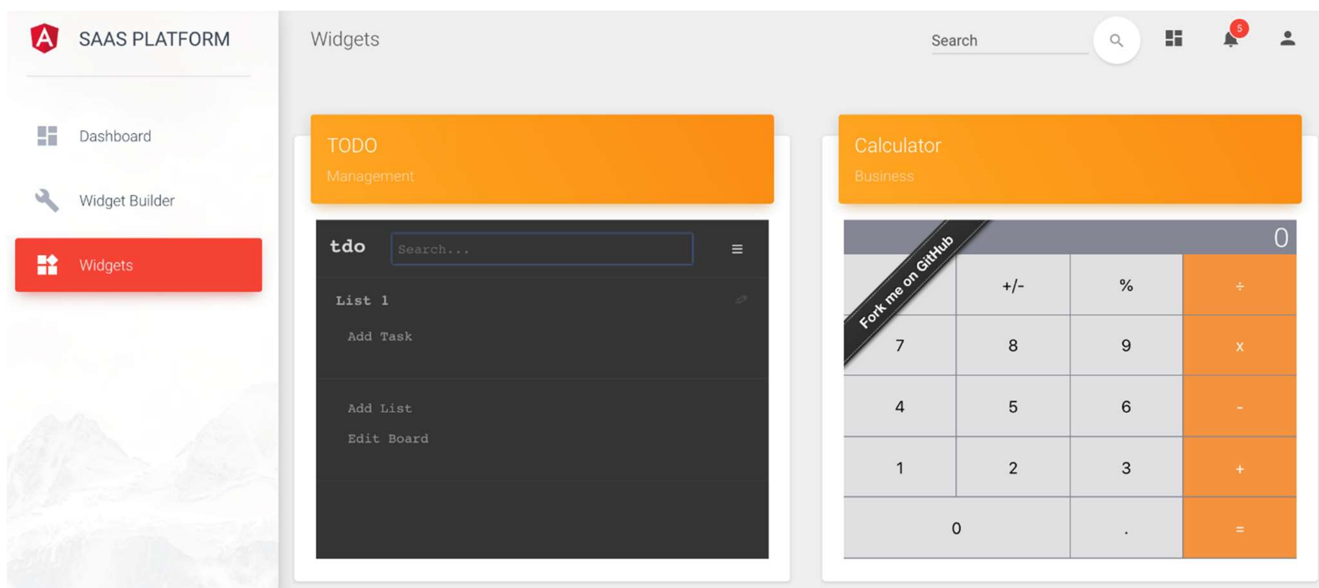

Рисунок 4.6 – Приклад доданих віджетів на сторінці Widgets

Програмний інтерфейс реалізується за допомогою динамічних html сторінок SPA. Додані віджети агрегуються та розташовуються в інтерфейсі користувача на сторінці Widgets. Кожен віджет на цій сторінці окремо зв’язується з обліковим записом користувача і ділиться даними та повідомленнями з самою системою. Віджети інкапсульвані та незалежні від інших і мають свій власний простір. Приклад списку доданих користувачем віджетів наведено на рисунку 4.6.

4.3 Технічна специфікація

Для програмної реалізації веб сервісу було використано:

– мова програмування JavaScript;

– платформа NodeJS;

– засіб автоматизації роботи з програмними проектами Maven;

– Mongoose та технологія доступу до бази даних MongoDB;

– база даних MongoDB;

– CSS3, HTML5, TypeScript, Angular для графічного відображення даних.

Розглянемо окремі особливості реалізації програмної системи. З цією метою розглянемо фрагмент програмного коду для основної сутності Widget, який надає доступ до репозиторію усіх віджетів системи:

```
var mongoose = require('mongoose');
var db = mongoose.createConnection('localhost', 'widgetapp');
var WidgetSchema = require('../models/Widget.js').WidgetSchema;
var Widgets = db.model('widgets', WidgetSchema);

exports.index = function(req, res) {
      res.render('index', {
            title : 'Widgets'
      });
};

exports.list = function(req, res) {
 Widgets.find({}, 'widget', function(error, widgets) {
            res.json(widgets);
      });
};
```

Кажучи про унікальні та встановлені віджети, що інкапсульовані користувачем з інших сервісів, користувачем, треба зазначити що до аккаунту користувача в системі прив’язані ідентифікатори тих віджетів, які були ним збережені.

Приклад моделі користувача із посиланнями на віджети:

```
exports.UserSchema = new mongoose.Schema({
     widgets : {
          id : String,
          required : true
     },
     username : String,
     account: String
     ...
```

Клієнтська частина інтегратора, до якої надається доступ через веб, реалізує для користувача каталог Widgets, що встановлені та розміщенні у власних контейнерах. Користувач може ініціювати GET-request до віджетів на сторінці. Виконання Request супроводжується відповіддю сервера з масивом посилань на список віджетів, який парситься та рендериться на сторінці користувача, завантажуючи необхідний URL віджета в тєг iFrame.

Приклад запиту на стороні клієнта:

```
@Injectable()
export class WidgetsService {
 constructor(private http: HttpClient) {}

 public fetchData(): Observable<LogDataInterface[]> {
 return
this.http.get<WidgetsDataInterface[]>(`${environment.serverUrl}widge
ts`);
 }
}
```

Обробка відповіді від серверу:

```
@Component({
 selector: 'app-widgets',
 templateUrl: './widgets.component.html',
 styleUrls: ['./widgets.component.css']
})
export class WidgetsComponent implements OnInit {

 constructor(private service: WidgetsService) { }

 ngOnInit() {
 this.widgets = this.service.fetchData();
 }
}
```

Шаблон відображення:

```
<div class="card" *ngFor="let widget of widgets | async">
 <div class="card-header" data-background-color="orange">
 <h4 class="title">{{ widget.name }}</h4>
 <p class="category">{{ widget.category }}</p>
 </div>
 <div class="card-content">
 <iframe [src]=" widget.src" frameborder="0"></iframe>
 </div>
</div>
```

Необхідно зазаначити, що на веб-сторінці може бути значна кількість віджетів, які будуть за допомогою iframe посилатися на віддалені ресурси та сервери. З метою запобіганню проблем із завантаженням ресурсів та перенавантаженям слабких клієнтських машин, в інтеграторі було реалізовано «ліниве» завантаження (Lazy Loading) віджетів на сторінці за допомогою вбудованих директив у фреймворк Angular.

```
     <div     class="card"     lazyLoadScroll     [lazyLoadItems]="10"
*ngFor="let widget of widgets | async">
     ...
```

Приклад самої директиви:

```
@Directive({
 selector: '[lazyLoad], [lazy-load-scroll],
})
export class lazyLoadDirective {
 @Input() lazyLoadItems: number = 5;

 constructor(private element: ElementRef, private zone: NgZone) { }

 ngOnChanges({ lazyLoadItems }: SimpleChanges) { }
 ...
}
```

4.4 Дослідження SOA-додатку

Відповідно до мети атестаційної роботи на підставі розробленої моделі було здійснено проектування та розробка хмарного додатку для інтеграції сервісів та мікросервісів різних хмарних платформ в єдиний програмний додаток. Створений програмний продукт є повністю працездатним та надає кінцевому користувачеві зручний інтерфейс для взаємодії з різними сервісами, що завантажені у відповідні контейнери з власним програмним оточенням.

З метою перевірки можливостей додатку було проведено тестування на відповідність функціональним вимогам.

4.3.1 Інтеграція сервісів

За допомогою конструктора віджетів додатку було виконано інтеграцію двох SaaS-сервісів, які належать різним хмарним платформам, в єдиний адресний простір інтегратора DashBoard.

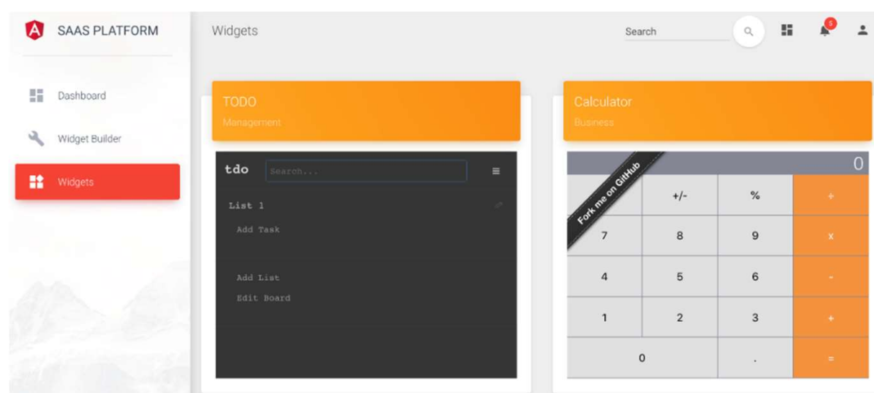

Рисунок 4.7 – Робота з двома сервісами

Разом з тим завантаження кожного з віджетів під час його виклику відбувається з конфігурацією та використанням середовища саме тієї платформи, на який його розгорнуто (рис.4.7).

4.3.2 Незалежне конфігурування

Кожний з вказаних віджетів має власний адресний простір-контейнер, що залежить від конфігурації платформи, на який його розгорнути. Інформацію про властивості віджетів можна отримати на головному екрані додатку (рис.4.8).

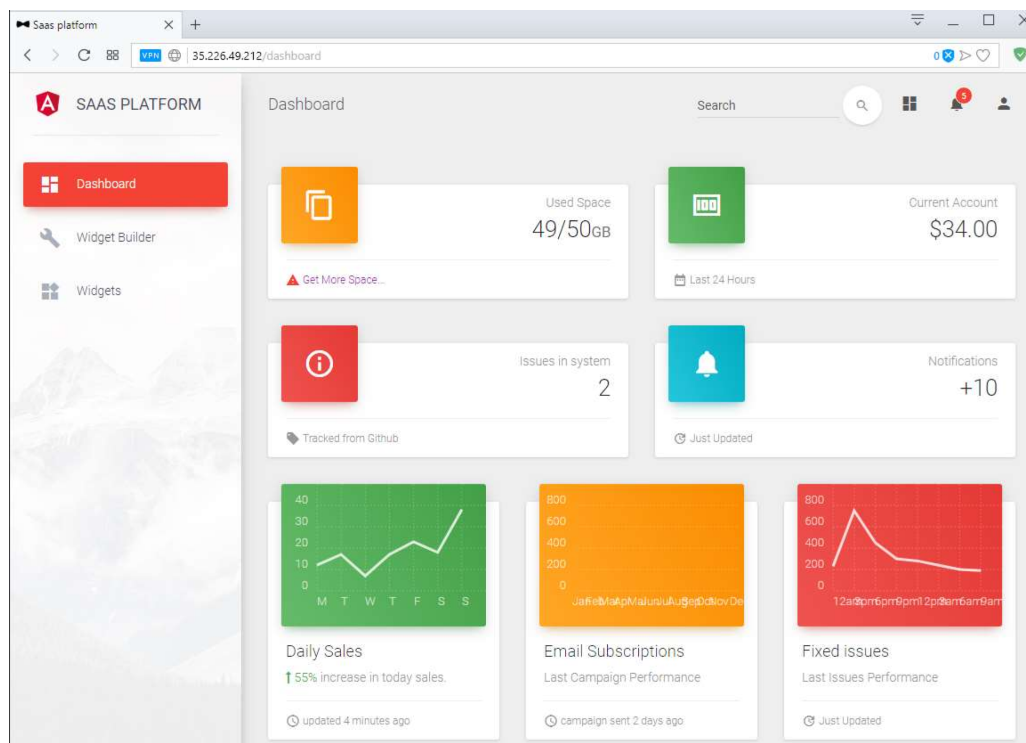

Рисунок 4.8 – Конфігурація сервісів

4.3.3 Перевірка функціональності

Кожний з вказаних сервісів дозволяє виконувати опрацювання інформації незалежно один від одного (рис. 4.9, 4.10).

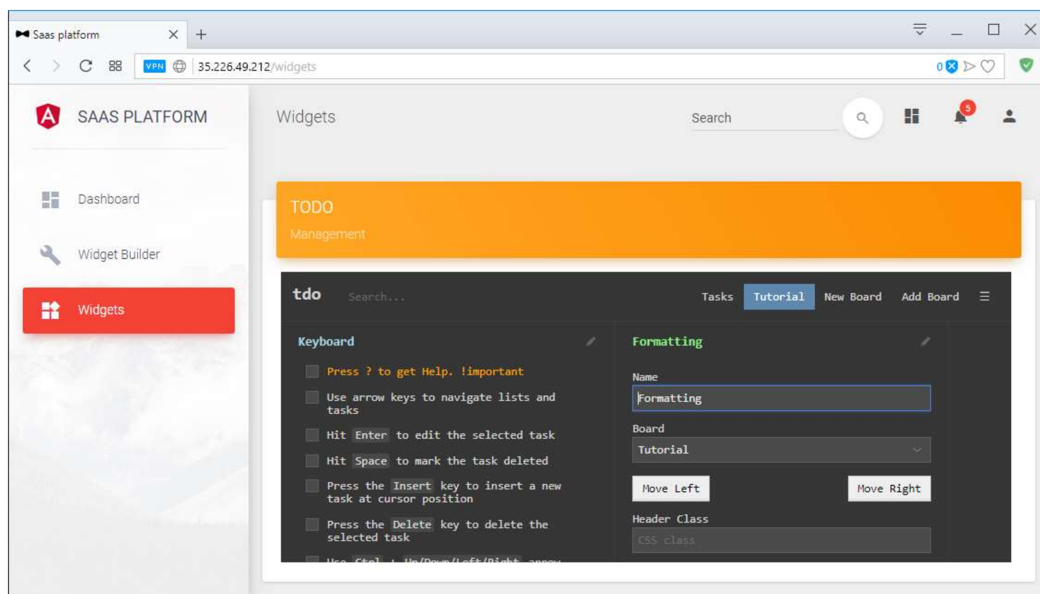

Рисунок 4.9 – Робота з сервісом Google Apps через додаток DashBoard

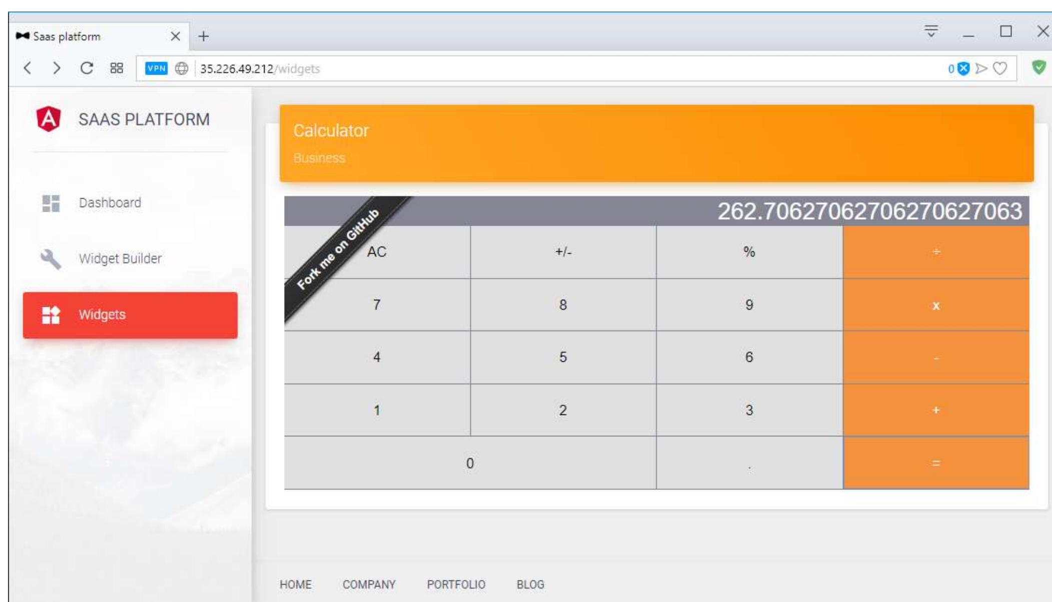

Рисунок 4.10 – Робота з сервісом GitHub через додаток DashBoard

Проведені дослідження доказують, що модель хмарного програмного продукту, яку розроблено в рамках написання атестаційної роботи магістра, є повністю працездатною. Програмна система забезпечує функцію інтеграції сервісів різних хмарних платформ в єдиний веб-додаток, є повністю працездатною та може знайти своє використання при розгортанні приватної хмари, яка буде орієнтована на розв’язання прикладних проблем.

ВИСНОВКИ

Під час виконання атестаційної роботи було проведено дослідження проблем, пов’язаних з організацією розробки програмного забезпечення, яке будується з використанням мікросервісної моделі SOA-додатків при розміщенні їх у хмарних середовищах.

В рамках дослідження було:

– проведено аналіз предметної області;

– виконано постановку задачі дослідження;

– проведено порівняльний аналіз можливих шляхів розв’язання проблеми та зроблено вибір на користь моделі інтеграції SaaS-сервісів в хмарне застосування;

– побудовано модель хмарного застосування, призначеного для інтеграції SaaS-сервісів різних провайдерів;

– проаналізовано можливості різних технологічних платформ реалізації сервісів в хмарі та здійснено обґрунтований вибір на користь Google Cloud Platform;

– здійснено програмну реалізацію та тестування програмної системи DashBoard з використанням контейнерного розгортання сервісів Kubernetes;

– виконано перевірку на дотримання функціональних вимог та відповідності отриманих результатів меті дослідження.

У підсумку слід зазначити, що система, модель якої було розроблено та досліджено в рамках написання атестаційної роботи магістра, реалізує новий підхід до розвитку використання хмарних сервісів та може знайти своє використання в умовах сучасного розвитку інформаційних технологій та їх впровадження в усі сфери людського життя.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. SOA terminology overview, Part 1: Service, architecture, governance, and business terms", https://www.ibm.com/developerworks/library/ws-soa-term1/.

2. Haluk Demirkan. “Service-oriented technology and management: Perspectives on research and practice for the coming decade” / H.Demirkan, R. J. Kauffman,

3. Thomas Erl. “Service-Oriented Architecture, Concepts, Technology, and Design”, Prentice Hall, ISBN : 0-13-185858-0, 2015.

4. . Ньюмен С. “Создание микросервисов” // СПб., Питер, 2016, 304 с. (Серия «Бестселлеры O’Reilly»). ISBN 978-5-496-02011-4.

5. Wähner. “A Good Microservices Architecture = Death of the Enterprise Service Bus (ESB)?”, dzone.com, Dzone, 2017, https://dzone.com/articles/good-microservices.

6. Горбенко А.В. Анализ особенностей создания и эксплуатации гарантоспособных сервисориентированных систем / А.В. Горбенко // Радіоелектронні і комп’ютерні системи. – 2013. – № 5 (64). – С. 237-242..

7. Петренко І. Процесно-орієнтоване проектування програмних комплексів як систем сервісів [Електронний ресурс] / Режим доступу до ресурсу: DOI: https://doi.org/10.20535/SRIT.2308-8893.2016.4.05.

8. Ляшов М.В., Берёза А.Н., Бабаев А.М., Алексеенко Ю.В., Авдеева Т.Г. Применение сервис-ориентированной архитектуры для создания распределенных вычислительных систем // Фундаментальные исследования. – 2016. – № 10-2. – С. 312-316 [Електронний ресурс] // URL: http://www.fundamental-research.ru/ru/article/view?id=40851 (дата звернення: 14.06.2019).

9. Microservices [Електронний ресурс] // URL: http://microservices.io/patterns/microservices.html (Дата звернення 20.04.2019).

10. O’Reilly T., What Is Web 2.0 [Электронный ресурс] // URL: http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html (Дата звернення 22.04.2019).

11. Разработка и внедрение Enterprise Resource Planning (ERP) систем на Open Source технологиях в машиностроительных предприятиях / С. С. Добротворский [и др.] // Вісник Нац. техн. ун-ту "ХПІ" : зб. наук. пр. Сер. : Технології в машинобудуванні– Харків : НТУ "ХПІ", 2018. – № 6 (1282). – С. 67-71 [Електронний ресурс] // URL: http://repository.kpi.kharkov.ua/handle/KhPI-Press/37172 (Дата звернення 14.06.2019).

12. Ricardo T., Marco T.V., Roberto S.B. An approach for extracting modules from monolithic software architectures. Workshop, pages 1–18, 2012.

13. Борисенко О.Д., Турдаков Д. Ю., Кузнецов С. Д., Автоматическое создание виртуальных кластеров Apache Spark в облачной среде OpenStack. Труды Института системного программирования РАН, том 17, 2009 г. Стр 31-50.

14. Остервальдер А., Пинье И. Построение бизнес-моделей. Настольная книга стратега и новатора. — М.: Альпина Паблишер, 2012.

15. Физерс, М. Эффективная работа с унаследованным кодом [Текст] / М. Физерс // Пер. с англ. – М.: Издательский дом «Вильямс», 2009 – 400 с.

16. Gillam L. Cloud Computing: Principles, Systems and Applications. / G. Gillam, N. Antonopoulos. – L.: Springer, 2010. – 379 p.

17. Why Cloud: зачем малому бизнесу облачные технологии [Електронний ресурс] // URL: https://delo.ua/business/why-cloud-zachem-malomu-biznesu-oblachnye-tehnologii-340677/ (Дата звернення 29.05.2019)

18. “Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications” [Електронний ресурс] // URL: https://kubernetes.io/ (Дата звернення 03.06.2019)

19. “Docker - Build, Ship, and Run Any App, Anywhere” [Електронний ресурс] // URL: https://www.docker.com/ (Дата звернення 03.06.2019)

20. “Containers as a Service: Comparing Providers and Evaluating the State of the Market” [Електронний ресурс] // URL: http://sandhill.com/article/containers-as-a-service-comparing-providers-and-evaluating-the-state-of-the-market/ (Дата звернення 03.06.2019)