МАГІСТЕРСЬКА РОБОТА
МР.ІПм – 27.00.00.000 ПЗ
Група ІПм-22-6
Воронич Андрій
2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Воронич Андрій Володимирович
(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

МАГІСТЕРСЬКА РОБОТА

Моделі та методи побудови великих та розподілених WEB-додатків

(назва роботи)

Інженерія програмного забезпечення
(назва освітньої програми)

121 - Інженерія програмного забезпечення
(шифр і назва спеціальності)

Воронич А.В.
(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Лютак Ігор Зіновійович - д.т.н., професор
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту
В.о. завідувача кафедри
доц. Бандура В.В.
(посада) (підпис) (дата) (ініціали та прізвище)

Рецензент

(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітній рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
В.о. зав. кафедрою **ІПЗ**

доц. В.В. Бандура

“ 04 ” вересня 2023 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Вороничу Андрію Володимировичу

(прізвище, ім’я, по-батькові)

1. Тема магістерської роботи “Моделі та методи побудови великих та розподілених WEB-додатків”
керівник проекту (роботи) Лютак Ігор Зіновійович - д.т.н., професор
затверджені наказом закладу вищої освіти від “ 18 ” грудня 2023 р. № 738/7

2. Строк подання студентом проекту (роботи) 15 січня 2024 р.

3. Вихідні дані до проекту (роботи) Теоретичні концепції та формальні моделі побудови та функціонування інформаційних та програмних технологій певного класу

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

1. Аналіз предметної області основ проектування великих та розподілених web-додатків
2. Методологія проектування великих та розподілених web-додатків
3. Реалізація методології побудови архітектури розподілених корпоративних web-додатків

5. Перелік графічного матеріалу (з точним забезпеченням обов’язкових креслень)

1. Логіка роботи обмеження цілісності (рис. 1.1, ст 15)
2. Узагальнена логіка роботи транзакцій (рис. 2.2,, ст.26)
3. Логіка роботи технології MapReduce (рис. 1.3)
4. Загальний вигляд готового рішення (рис. 1.4)
5. Графічний образ моделі водоспаду (рис. 2.6)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата
Нормоконтроль	доц., к.т.н. Вовк Р.Б.	
Перевірка на плагіат	доц., к.т.н. Вовк Р.Б.	

7. Дата видачі завдання 04 вересня 2023 р.

Керівник роботи _____
(підпис)

Завдання прийняв до виконання _____

(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір і вивчення літератури по темі дослідження	01.10.2023	виконано
2	Аналіз предметної області основ проектування великих та розподілених web-додатків	25.10.2023	виконано
3	Методологія проектування великих та розподілених web-додатків	22.11.2023	виконано
4	Реалізація методології побудови архітектури великих та розподілених корпоративних web-додатків	15.12.2023	виконано
5	Затвердження пояснювальної записки роботи завідувачем кафедри	25.01.2024	виконано

Студент – магістр _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Магістерська робота: 76 с., 32 рис., 2 табл., 45 джерел

Тема: Моделі та методи побудови великих та розподілених WEB-додатків

Об'єкт дослідження: є процеси та методологія розробки розподілених корпоративних WEB-додатків

Мета роботи: є розробка та вдосконалення методології розробки і архітектури програмного забезпечення корпоративних, великих і розпределених веб-орієнтовіаних систем.

Предмет дослідження: є моделі, методи та алгоритми проектування та побудови корпоративних веб-додатків.

Результати дослідження:

В результаті дослідження пропонується концептуальний підхід до її вирішення на основі аналізу особливостей побудови розподілених web-додатків. Підхід базується на визначенні стандартної архітектури web-додатків і виборі її складових з використанням формальних методів відповідно до вимог користувача.

Висновок:

У результаті проведеного дослідження розроблено та вдосконалено методологію розробки розподілених корпоративних веб-додатків. В магістерській роботі досліджено питання побудови архітектури сучасних корпоративних додатків на Node.js. Запропоновано архітектуру програмного забезпечення та рекомендації щодо впровадження для розробки хмарних додатків Node.js.

КОРПОРАТИВНИЙ ДОДАТОК, МЕТОДОЛОГІЯ, РОЗПОДІЛЕНИЙ ДОДАТОК, АРХІТЕКТУРА, ФРЕЙМВОРК, ПРОГРАМНА ПЛАТФОРМА, СЕРВІС-ОРІЄНТОВАНА АРХІТЕКТУРА

ANNOTATION

Master's thesis: 76 pp., 32 figures, 2 tables, 45 sources

Topic: Models and methods of building large and distributed WEB applications

Research object: there are processes and methodology for developing distributed corporate WEB applications

The purpose of the work: is the development and improvement of the development methodology and software architecture of corporate, large and distributed web-oriented systems.

Research subject: there are models, methods and algorithms for designing and building corporate web applications.

Research results:

As a result of the study, a conceptual approach to its solution is proposed based on the analysis of the features of building distributed web applications. The approach is based on defined standard web application architectures and selection of its components using formal methods according to user requirements.

Conclusion:

As a result of the conducted research, the methodology for the development of distributed corporate web applications was developed and improved. The issue of building the architecture of modern corporate applications on Node.js was investigated in the master's thesis. Proposed software architecture and implementation guidelines for developing Node.js cloud applications.

CORPORATE APPLICATION, METHODOLOGY, DISTRIBUTED APPLICATION, ARCHITECTURE, FRAMEWORK, SOFTWARE PLATFORM, SERVICE-ORIENTED ARCHITECTURE

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

RUP - Rational Unified Process

XP - Extreme Programming

FDD - Feature Driven Development

TDD - Test Driven Development

XML - eXtensible Markup Language

JSON - JavaScript Object Notation

SQL -Structured Query Language

БД – база даних

ANSI - American national standards institute

СУРБД - система управління реляційними базами даних

MVC - Model-View-Controller

JSP - Java Server Pages

HTTP - HyperText Transfer Protocol

ASP - Active Server Pages

S.O.L.I.D. – single responsibility, open-closed, Liskov substitution, interface segregation, dependency inversion

JDBC - Java DataBase Connectivity

JMX - Java Management Extensions

DNS - Domain Name System

LINQ - Language Integrated Query

REST - Representational State Transfer

DOM - Document Object Model

ВСТУП

Актуальність роботи.

Створення інформаційних систем сьогодні здійснюється на основі сучасних методологічних концепцій, які успадкували найважливіші ідеї класичних методологій типу SADT,IDEFO, одночасно збагативши їх новими ідеями.

Найперспективнішим напрямом розвитку програмних продуктів є клієнт серверна взаємодія в більшості побудована на основі HTTP протоколу, тобто веб застосування, що включають в себе ту саму обробку даних, подаючи її в зручному для користувача вигляді.

Формалізація досить специфічно представлена в сучасних методологіях. У класичних методологіях її розглядали як основу автоматизації. По суті формалізація тоді уявлялася як невід'ємною здатність кожної технології, яка реалізувала методологію, інструмент автоматичного продукування складових інформаційної системи, насамперед програмного забезпечення, баз даних, на підставі їх формальних описів. Сьогодні формалізація більше пов'язана з описанням архітектури системи і описи переважно використовується для постановки задач учасникам проекту, обговорення проміжних результатів, визначення наступних кроків.

Але є реальна можливість надати формалізації більш важливої ролі в сучасних методологіях створення інформаційних систем. По-перше, цьому сприяє стандартизація архітектури і компонентів інформаційних систем. По-друге, у рамках стандартизованих архітектур напрацьовано багато компонентів, які можуть бути використані для проектування нових систем.

По-третє, розвиток математичної логіки і штучного інтелекту останніми роками забезпечив як інструменти формального описання систем, що проектуються з точки зору «що має бути реалізоване», так і описання компонентів з точки зору «що вже реалізоване». По-четверте, з'явилися методології просторової візуалізації систем, що проектуються, які можна

перенести на створення на основі компоненто-базованого підходу інформаційних систем.

Під час практики зроблено спробу використати напрацьовані формальні моделі і ефективні методи для побудови технології автоматизованого проектування інформаційних систем на основі сучасних методологій. Оскільки ця проблема має колосальні масштаби ми виділили один із найбільш готових для реалізації компонент сучасних інформаційних систем. Мова йде про задачу розроблення, яка стає все більш популярною, а саме задачу розроблення систем з web-представленнями.

Актуальність роботи визначається кількома ключовими факторами:

Швидка зміна технологічного середовища - з урахуванням стрімкого розвитку технологій, важливо мати програмне забезпечення, яке не тільки відповідає поточним вимогам, але й здатне працювати ефективно в змінюючомуся середовищі протягом тривалого періоду часу.

Зростання складності програмних продуктів - сучасні програмні продукти стають все складнішими, і вони піддавані впливу різноманітних факторів, таких як зміни в вимогах, вразливості безпеки та інші.

Підвищення вимог до якості - користувачі сподіваються на високу якість та надійність програмних продуктів.

Ефективне управління ризиками - забезпечення стійкості програмного забезпечення дозволяє ефективно управляти ризиками, пов'язаними з можливими витратами часу та ресурсів на розробку, тестування та вдосконалення.

Вартість утримання - програмне забезпечення, яке легко піддається утриманню та апгрейду, може значно зменшити витрати на підтримку та розвиток порівняно з системами, які стають застарілими або непридатними для модернізації.

Підвищення конкурентоспроможності - підприємства, які можуть швидко реагувати на зміни та підтримувати високий рівень стійкості свого програмного забезпечення, зберігають конкурентну перевагу на ринку.

Отже, розробка ефективної методології розробки програмного забезпечення великих і розпределених систем має велике значення для сучасного індустріального та технологічного середовища, сприяючи стабільності та успішності програмних продуктів у тривалій перспективі.

Розроблена методологія виходить за рамки традиційних підходів, об'єднуючи елементи деталізації вимог, гнучкості до змін та автоматизації, спрямовані на забезпечення ефективної розробки корпоративних веб-додатків. Вона може слугувати ефективним інструментом для розробників, що шукають компроміс між традиційними та гнучкими методами розробки.

Мета і задачі дослідження

Метою магістерської роботи є розробка та вдосконалення методології розробки і архітектури програмного забезпечення корпоративних, великих і розпределених веб-орієнтовіаних систем.

Мета визначає загальний цільовий результат, який дослідник прагне досягти. У даному випадку, мета полягає в створенні ефективної методології, яка дозволяє забезпечити стійкість програмного забезпечення протягом тривалого періоду часу.

Досягнення мети включало розв’язання таких **задач**:

- Проблематизація та формулювання цілей дослідження,

- Огляд літератури, аналіз трендів у сфері розробки веб-додатків,

- Визначення вимог до якості, моделювання процесу розробки корпоративних веб-додатків,

- Вдосконалення методики проектування великих та розподілених web-додатків,

- Реалізація методології та архітектури корпоративної веб-орієнтованої системи.

Об’єктом дослідження є процеси та методологія розробки розподілених корпоративних WEB-додатків.

Предметом дослідження є моделі, методи та алгоритми проектування та побудови корпоративних веб-додатків.

Методи дослідження

Для реалізації поставлених завдань використовуються наступні методи: Аналіз літературних джерел, експертний аналіз, моделювання: для створення та тестування моделей процесу розробки та дизайну ПЗ. аналіз трендів: дослідження актуальних технологічних та індустріальних тенденцій у галузі розробки ПЗ. Ці методи спрямовані на систематизацію знань, аналіз існуючих підходів та створення нових концепцій для покращення методології розробки розподілених веб-додатків.

Наукова новизна отриманих результатів

В результаті дослідження пропонується концептуальний підхід до її вирішення на основі аналізу особливостей побудови розподілених web-додатків. Підхід базується на визначенні стандартної архітектури web-додатків і виборі її складових з використанням формальних методів відповідно до вимог користувача.

Практичне значення одержаних результатів

В магістерській роботі досліджено питання побудови архітектури сучасних корпоративних додатків на Node.js. Запропоновано архітектуру програмного забезпечення та рекомендації щодо впровадження для розробки хмарних додатків Node.js.

Структура та обсяг магістерської роботи

Магістерська робота викладена на 76 сторінках друкованого тексту, який складається із вступу, трьох розділів, висновків, списку використаних джерел (45 найменувань). Робота містить 2 таблиці, 32 рисунки.

РОЗДІЛ 1.
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ОСНОВ ПРОЕКТУВАННЯ ВЕЛИКИХ ТА РОЗПОДІЛЕНИХ WEB-ДОДАТКІВ

1.1 Проблематизація. Мета та задачі дослідження

Стійкість у розробці програмного забезпечення – це сфера, яка лише нещодавно почала набирати обертів. Значна частина досліджень дотепер стосувалася екологічного виміру стійкості, наприклад. Визначення технічної стійкості охоплює питання, наприклад придатність до обслуговування, які постійно обговорюються в рамках дисципліни програмної інженерії та є центральними питаннями в галузі, як зазначено вище. Є кілька попередніх робіт, які досліджували сталість у та як нефункціональну вимогу. Це дослідження мало на меті продовжити цю попередню роботу, досліджуючи технічну стійкість шляхом розгляду нефункціональних вимог і якості програмного забезпечення як її компонентів, а також аналізу практики, яка зараз використовується в галузі щодо цих компонентів технічної стійкості.

Мета цього дослідження полягала в тому, щоб дослідити, як можна досягти технічної стійкості підходу до розробки програмного забезпечення в галузі. Для досягнення мети дослідження було сформульовано наступне основне питання дослідження:

• Як можна досягти технічного виміру сталого програмного програмування?

Для відповіді на основне питання дослідження були розроблені підпитання. По-перше, необхідно розуміти, які практики зараз використовуються і чому. Це було розглянуто в першому підпитанні:

Підпитання 1: Які практики зараз використовуються в проектах розробки програмного забезпечення і чому?

Відповівши на перше запитання, наступним кроком було отримати розуміння того, як ці практики діють у відношенні технічної сталості, тобто

яким чином вони сприяють або не сприяють аспектам технічної стійкості. Це було розглянуто в другому підпитанні:

Підпитання 2: Як ці практики діють щодо аспектів технічної стійкості?

Зі знаннями, отриманими під час відповіді на перші два підзапитання, останнє питання, яке потрібно було розглянути, полягало в тому, які зміни в практиках необхідні для їх покращення з точки зору технічної стійкості. Це було розглянуто в останньому підпитанні:

Підпитання 3: Які зміни потрібні для покращення технічної стабільності проектів розробки програмного забезпечення?

Відповівши на три підзапитання вище, можна отримати відповідь на основне питання дослідження і таким чином досягти мети дослідження.

Дослідження було зосереджено на технічному аспекті стійкості сталого розвитку, де нефункціональні вимоги та якість програмного забезпечення є компонентами. Основна увага під час дослідження була зосереджена на аспекті ремонтопридатності програмного забезпечення щодо нефункціональних вимог і якості програмного забезпечення.

Дослідження було вибрано для того, щоб зосередити увагу на споживацькому програмному забезпеченні для стійкої розробки програмного забезпечення; отже, він не охоплював програмне забезпечення «бізнес-бізнес». Для цього дослідження було обрано диференціатор між споживчим програмним забезпеченням і програмним забезпеченням для бізнесу-бізнесу як передбачувану ціль програмного забезпечення; де цільове програмне забезпечення «бізнес-бізнес» — це один конкретний бізнес, а цільове програмне забезпечення для споживачів — це не конкретна юридична особа, а клієнти з певною потребою в програмному забезпеченні.

У цьому розділі описано метод, використаний для виконання цього дослідження. У ньому описано, як проводився збір і аналіз даних. Він також містить обговорення надійності та валідності дослідження. Нарешті, представлено обговорення можливості узагальнення результатів і рішень, прийнятих у дослідженні щодо етики.

Цей розділ починається з огляду змін у програмному забезпеченні та розробці програмного забезпечення, які відбулися за останні десятиліття. Цей розділ охоплює теорію та попередню роботу, пов’язану з досліджуваною сферою. По-перше, розглядається тема стійкої інженерії програмного забезпечення, де представлені її визначення. Також представлено роботу над тим, як включити стійкість у розробку програмного забезпечення .

Потім представлений розділ, присвячений теорії та попереднім роботам щодо ремонтопридатності програмного забезпечення. Нарешті, розглядаються моделі процесу розробки програмного забезпечення та дизайн програмного забезпечення.

У цьому розділі обговорюється, які практики використовуються і чому. Потім обговорення того, як ці практики діють у відношенні технічної стійкості. Обговорення містить інформацію про те, як моделі процесу обробляють вимоги та управління часом щодо технічної стійкості, а також включає обговорення того, як дизайн програмного забезпечення пов’язаний з технічною стійкістю. Нарешті, у цьому розділі представлено обговорення того, які зміни потрібні щодо того, що обговорювалося в цьому.

1.2 Аналіз проблеми розробки web-додатків для обробки великих об’ємів даних

Розробка системи управління базами даних почалася з того, що розробники, яким довелося створювати багато додатків, поступово стали звертати увагу на те, що в різних за своєю суттю додатках їм доводиться програмувати одну і ту ж функціональність, пов'язану з накопиченням, пошуком і аналізом даних. Тенденції роботи з великими обсягами даних росли з кожним роком і прикладних задач, що вимагають обробки цих даних відповідно теж росли. Крім того, багаторазова реалізація дуже схожою функціональності (причому не завжди вдала) значно здорожувала кінцевий програмний продукт. На цьому тлі і виникла ідея розмежування функцій

програми і функцій обробки даних. З'явилася ідея про створення централізованих систем, які забезпечують доступ до даних, тобто ідея про створення високоефективних і якісних універсальних систем, орієнтованих нема на потреби окремих додатків, а виключно на обробку даних. Такого роду системи можуть бути створені один раз. Для їх створення можна використовувати програмістів вищої кваліфікації, а самі системи в подальшому використовувати багаторазово при створенні різноманітних додатків. Ця ідея дозволяла істотно підвищити якість і знизити вартість подальшої розробки додатків. Було запроновано наступні ідеї принципів для систем, керуючих базами даних [1]:

– централізація управління зберіганням і доступом до даних;

– зниження вартості розробки і підвищення якості додатків за рахунок винесення загальної функціональності в СУБД;

– підтримка складних логічних структур; – забезпечення незалежності даних і додатків;

– забезпечення цілісності даних (Integrity);

– підтримка узгодженості даних (Consistency);

– захист від несанкціонованого доступу;

– розмежування прав доступу;

– підтримка високорівневих ефективних мов запитів.

В сучасних системах управління даними значення і зміст багатьох з цих функцій істотно змінилося. Деякі з цих функцій повністю або частково перейшли до інших типів прикладних систем, важливість і наявність інших систем залежать від архітектури прикладної інформаційної системи і типу використовуваної СУБД. Однак в цілому не можна не відзначити, що цей список дійсно визначив напрямок розвитку систем баз даних на багато років вперед. Спробуємо перерахувати основні функції сучасних СУБД і почнемо з функціональності, передбаченої цим списком. Забезпечення незалежності даних і додатків спочатку розглядалося як найважливіший елемент систем

управління. Ті принципи, які розуміються під незалежністю даних і додатків, розбиваються на три можливі групи:

– одні і ті ж дані можуть використовуватися для різних додатків;

– поява нових вимог до даних (наприклад, додавання нових полів, таблиць, бізнес логікі поведінки даних, тощо) не повинно впливати на роботу існуючих додатків;

– припустимо асинхронне впровадження нових версій програм (останній принцип особливо актуальний для крупних клієнтів).

Для підтримки відповідних вимог були, зокрема, введено мови, які декларують опис структури і бізнес логікі поведінки даних. Словники, що представляють відповідну інформацію, що зберігаються в системах управління даними, тощо. Словники даних і мови опису структури і бізнес логікі даних в тому чи іншому вигляді включені майже в усі сучасні традиційні СУБД. Більшість систем строго дотримуються принципів незалежності даних, проте слід зазначити, що останнім часом стали з'являтися СУБД, які відступають від цих, здавалося б, беззастережно корисних принципів. Порушення принципів незалежності першої зі згаданих вище груп означає, що з'являються сховища даних, вузько орієнтовані на специфіку даних і функціональність однієї програми. Це могло б здатися дуже нераціональним, якби не та обставина, що такі додатки носять ексклюзивний характер і затребувані мільйонами користувачів. Порушення принципів незалежності з другої і третьої груп очевидним чином позначається на розробці, супроводі та впровадженні програм, які працюють з такими системами. Крім відповідності структурі описаних даних, система управління даними повинна забезпечувати виконання ряду умов. Цим умовам повинні задовольняти збережені дані (наприклад, квартира в якій живе людина повинна бути від 1 до 100; квартира, по якій визначається її нумерація повинна бути в списку будинків вулиці, тощо). Такого роду умови називають обмеженнями цілісності (Integrity), і вони також описуються за допомогою спеціальних мов опису даних. За перевірку виконання заданих обмежень відповідає СУБД. Будь-які операції, що

намагається порушити ці обмеження, відкидаються. На рисунку 1.1 зображена логіка роботи обмеження цілісності.

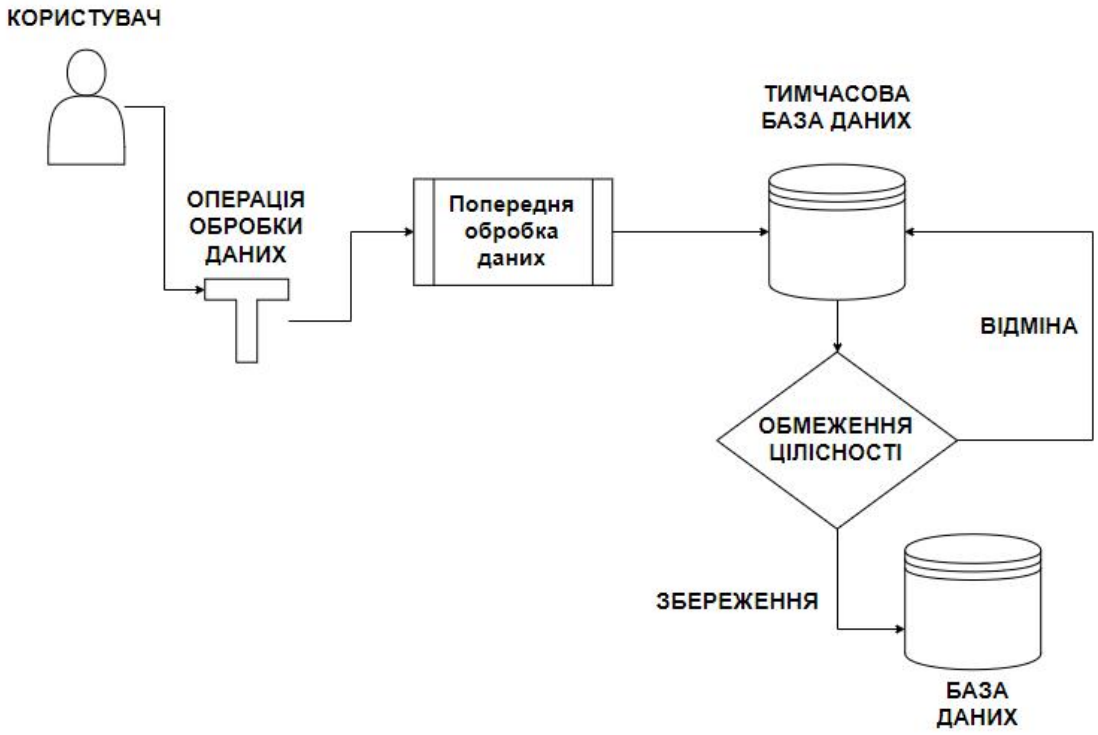

Рис. 1.1. Логіка роботи обмеження цілісності

Однак існують СУБД, в яких правила цілісності або взагалі відсутні, або присутні лише частково. При цьому слід зазначити, що якщо сама природа даних така, що вимагає врахування якихось обмежень, то ці обмеження все одно доведеться враховувати на будь-якому рівні, і якщо немає можливості врахувати їх на рівні СУБД, то розробники будуть змушені їх враховувати на рівні створюваного додатка. Підтримка узгодженості (Consistency) пов'язана зі статками даних. Деякі стану бази даних називаються узгодженими. Завдання СУБД - забезпечення механізмів, що дозволяють переводити дані з одного узгодженого стану в інше. Набір операцій, які переводять базу з одного узгодженого стану в інше, прийнято називати транзакцією. Узагальнена логіка роботи транзакції зображена на рисунку 1.2.

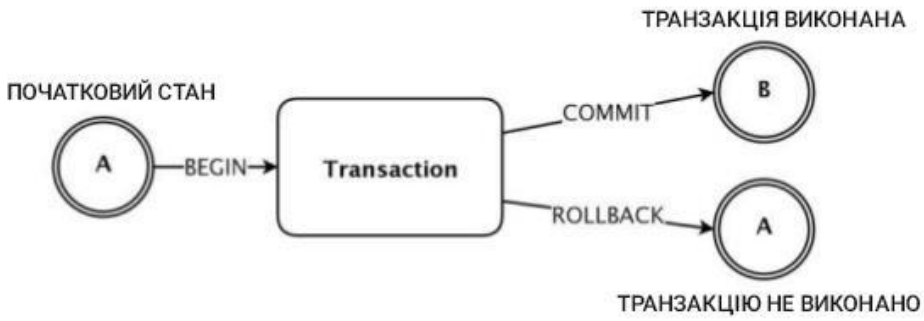

Рис. 1.2. Узагальнена логіка роботи транзакцій

Для багатьох NoSQL систем при обробці даних характерно використання технології MapReduce [8]. Відповідно до цієї технології, обробка даних складається з двох кроків - Map і Reduce (рисунок 1.3).

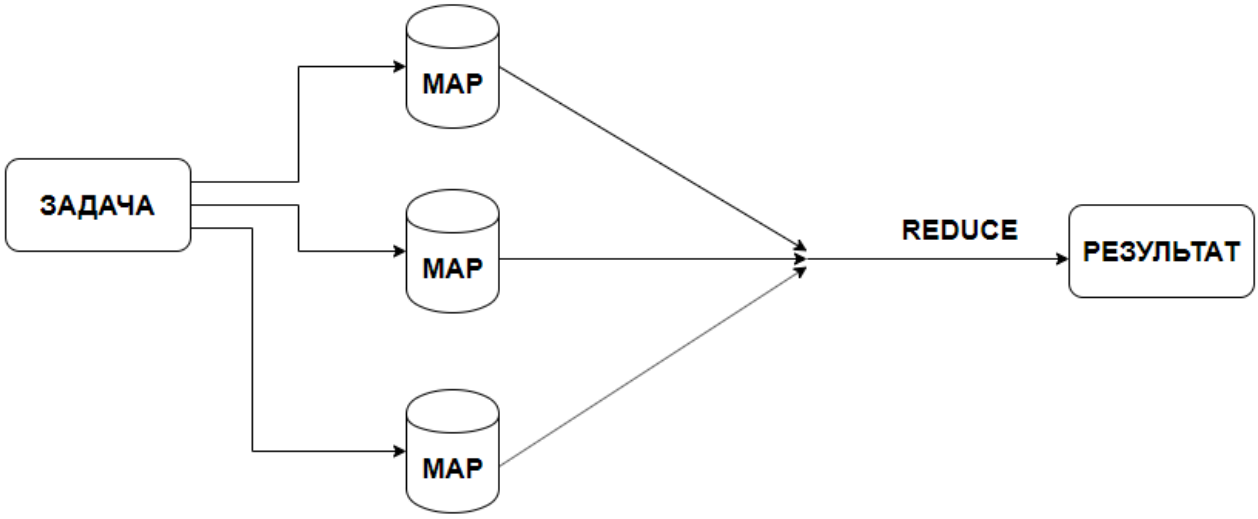

Рис. 1.3. Логіка роботи технології MapReduce

Отже, у підходах до збереження та обробки даних існує багато рішень та підходів для прискорення видачі результату користувачу. Різні моделі баз даних об’єднують у собі лише частину існуючих методів по забезпеченню швидкого, надійного та коректного доступу до даних, що зберігаються. Сучасні СУБД можуть комбінувати ці методи, або використовувати окремі переваги кожного з них. В залежності від типу даних, що зберігаються, та моделі бази

даних, будуть використані різні методи збереження даних, які підтримуються конкретною СУБД.

1.3 Опис задачі побудови архітектури web-рішення

На сьогоднішній день, значний об'єм ринку всіх програмних продуктів займають бізнес системи з веб представленням. При цьому від проекту до проекту змінюються дані з якими працюють розробники і користувачі, але не змінним залишається принцип їх реалізації. Доцільність автоматизації цього класу задач пояснюється значною трудомісткістю проектування і реалізації підзадач, які дублюються у кожному проекті, і при цьому власне зміні підлягають лише дані, з якими працюють, але не основні їх перетворення.

Як зазначалося вище, розробникам варто домовитися про те, що будь-яке застосування вибраного типу має в своїй основі напрацьовану архітектуру.

Також легко домовитися про те, що будь-яке застосування вибраного типу має однаковий загальний вигляд і відрізняється від інших лише наявністю або відсутністю тих чи інших компонентів в залежності від вимог до функціоналу застосування.

Сьогодні при виборі будови застосувань доцільно дотримуватися принципів трирівневої архітектури. Відповідно у типовому застосуванні будемо виділяти представлення, бізнес-логіку і рівень доступу до даних.

Кожен з яких відповідно до принципу декомпозиції на під задачі можна деталізувати на рівень агрегації класів що вирішують ту чи іншу задачу та називаються патерном. В кожному патерні виділяють абстрактну частину, що дає змогу ізолювати конкретну реалізацію від загальної поведінки та імплементацію, що надає відповідним класам абсолютно чітко визначені методи з їх сигнатурами і відповідною реалізацією. Детальніше декомпозиція полягає в структуруванні конкретного класу з наведеного патерну та доповнені його необхідними функціональними одиницями.

Розглянемо приклад детальніше кожний з рівнів архітектури.

Розглянемо приклад типової задачі, що включає в себе всі рівні архітектури та демонструє загальний вигляд системи. Необхідно реалізувати web систему з розмежуванням прав доступу відповідно з двома користувацькими ролями адміністратора та користувача. Користувач після логіну може завантажити файл з розширенням JSON в якому повинна міститись інформація що описує нове обладнання. Система повинна зберегти службову інформацію з файлу опрацювати її і вивести адміністратору на підтвердження. Згідно з вимогами до коду, а саме логіка опрацювання файлу може змінюватись код повинен бути чистим та гнучким, відповідати всім принципам S.O.L.I.D. З точки зору вимог по функціоналу система повинна вміти авторизувати користувачів, опрацьовувати, а саме перевіряти коректність інформації в завантаженому файлі, зберігати історію завантажень, та повідомляти адміністратора ресурсу через електрону пошту про нове не підтверджене завантаження файлу.

Виходячи з цього маємо наступну архітектуру.

• CommandInvoke.Db.Entities – збірка сутностей що відповідають таблицям в базі даних.

• CommandInvoke.Dal.Abstract абстракція з описом сигнатури CRUD операцій з джерела данних оперуючи сутностями з п.1.

• CommandInvoke.Dal.Impl.Ef – реалізація конкретної поведінки по роботі з MSSQL server засобами ORM Entity framework з використанням підходу Code first.

• CommandInvoke.Entities.Ui моделі якими оперує бізнес-логіка для видачі через Api

• CommandInvoke.Abstract абстракція з описом логіки, що покриває всі функціональні вимоги до системи

• CommandInvoke.Bl.Impl – реалізація конкретної поведінки по виконанню процесів системи

• CommandInvoke.Web – Rest API інтерфейс мережевої взаємодії з системою що працює з абстракцією бізнес-логіки і повертає дані для відображення.

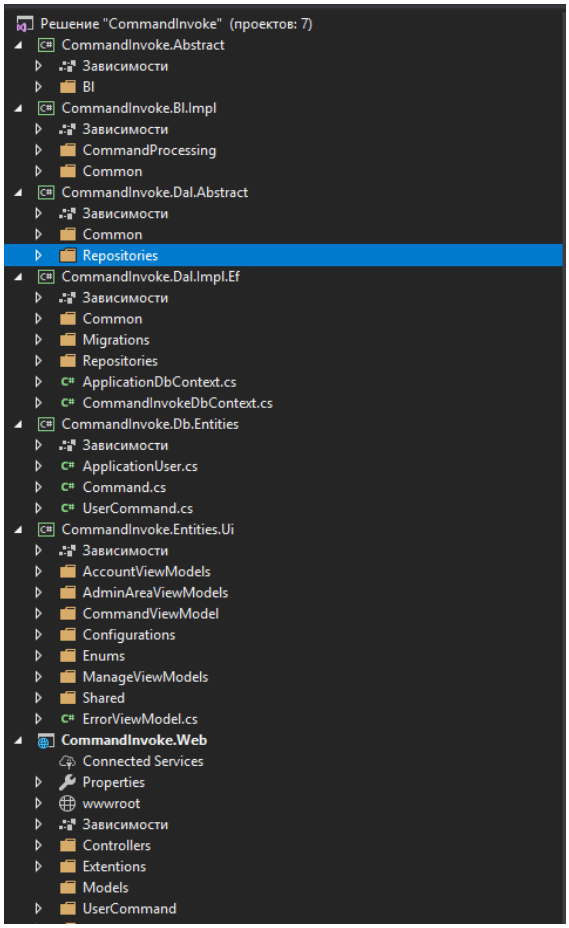

Рис. 1.4. Загальний вигляд готового рішення

Розглянемо шаблони що застосовувались в ході реалізації даного рішення.

Для рівня доступу до даних використовувався патерн репозиторій. Для бізнес-логіки стратегія. І два шаблони рівнем вище котрі повинні в випадку шаблонізації бути задані як окремі модулі, що покривають бізнес-процес авторизації реєстрації, та відправку електронних повідомлень. Як видно з опису задача з одного боку достатньо об’ємна та потребує часових затрат на навчальному прикладі в розмірі 8 годин, а з іншого боку складається виключно з монотонної роботи по зберіганню та завантаженню даних з бази даних і більш

складного процес їх обробки та двох регулярно використовуваних бізнес-процесів авторизації та відправки електронного листа.

Існує два основних підходи, що розв’язують дану задачу:

Репозиторій - це фасад для доступу до бази даних. Весь код програми за межами сховища працює з базою даних через нього і тільки через нього. Таким чином, репозиторій інкасулює в собі логіку роботи з базою даних, це шар об'єктно-реляційного відображення в нашому додатку. Більш точно, репозиторій, або сховище, це інтерфейс для доступу до даних одного типу - один клас моделі, одна таблиця бази даних в простому випадку. Доступ до даних організовується через сукупність всіх репозиторіїв.

Unit Of Work - є класом обгорткою для репозиторію та являє собою контейнер для зберігання та створення репозиторіїв.

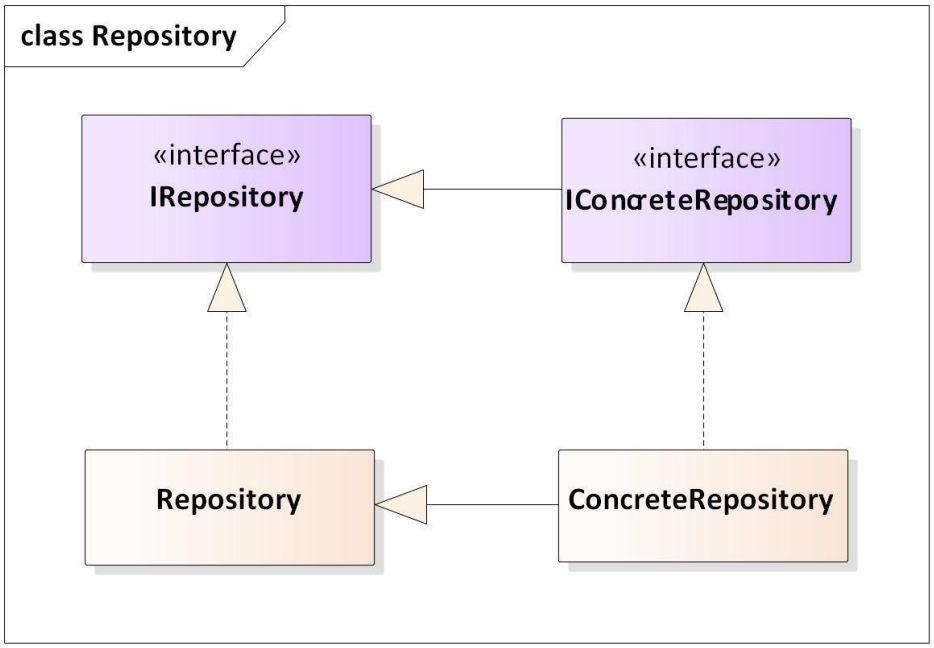

Рис. 1.5. Загальний вигляд патерну репозиторій

Шаблон "Repository", як видно з діаграми, складається з узагальненої абстракції і більш деталізованих її імплементацій. Розглянемо приклад шаблону з точки зору реалізації в коді.

– IRepository<T> шаблоний інтерфейс що описує базову поведінку для класів наслідників по роботі з даними;

– IEntityRepository більш конкретна абстракція задачею якої є зберігання в собі унікальних методів для роботи з поточною сутністю(Entity) та наслідується від IRepository<T> закладуючи абстракцію для базових методів але з чітко визначеним типом;

– BaseRepository<T> є класом що імплементує IRepository<T> та реалізовує всі його методи базові для всіх сутностей і містить прив’язку до певної конкретно вибраної технології по роботі з БД;

– EntityRepository конкретний клас наслідни що імплементує свій конкретний інтерфейс (IEntityRepository) та унаслідує базовий клас BaseRepository<T> для реалізації стандартних методів по роботі з даними.

Тобто роботу з даними визначаємо у традиційний для об’єктно-орієнтованого програмування спосіб. По-перше з кожною сутністю пов’язуємо певну узагальнену поведінку, яка притаманна всім елементам рівня доступу до даних. По-друге доповнена абстракція для кожного елемента рівня доступу до даних розширює його поведінку, додаючи унікальні методи для поточної сутності.

Відповідно конкретний клас сховища успадковує свій інтерфейс, отримуючи базові методи загальні для всіх і свої методи з власного інтерфейсу, а також успадковує клас, який реалізовує базову поведінку, за фактом чого містить реалізацію тільки своїх методів.

Як можна побачити з опису, паттерн має визначену архітектуру і є розширюваним за рахунок засобів додавання нових класів і інтерфейсів для роботи з кожною сутністю. А також патерн Repository є посередником між шаром області визначення і шаром розподілу даних, працюючи, як звичайна колекція об'єктів області визначення. Об'єкти-клієнти створюють опис запиту декларативно і направляють їх до об'єкта-сховища (Repository) для обробки.

Об'єкти можуть бути додані або видалені з сховища, як ніби вони формують просту колекцію об'єктів. А код розподілу даних, прихований в об'єкті Repository, подбає про відповідних операціях в непомітно для розробника. У двох словах, патерн Repository інкапсулює об'єкти,

представлениие в сховище даних і операції, що здійснюються над ними, надаючи більш об'єктно-орієнтоване уявлення реальних даних.

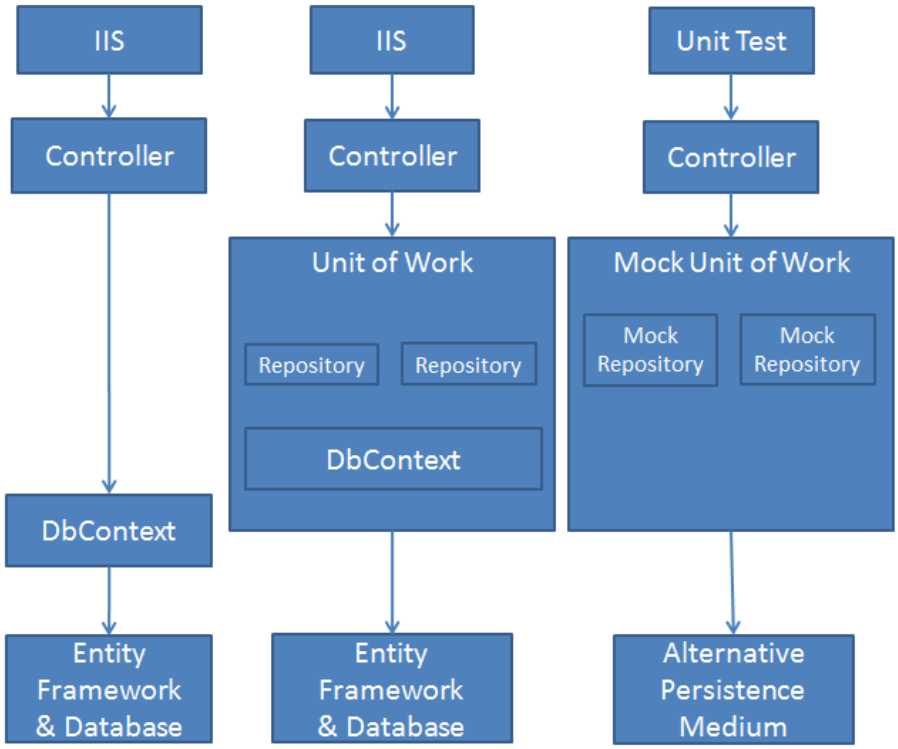

Рис. 1.6. Загальний вигляд патерну Unit of work

Unit Of Work є класом обгорткою для репозиторію та являє собою контейнер для зберігання та створення репозиторіїв. Дана надбудова над репозиторієм застосовується в випадку коли в додатку відсутня DI (Dependency Injection) як механізм підстановки імплементацій за інтерфейсом.

Як можна побачити з опису, паттерн має визначену архітектуру і є розширюваним за рахунок засобів додавання нових класів і інтерфейсів для роботи з кожною сутністю.

1.4. Реалізація бізнес-логіки

Бізнес-логіка переважно полягає у обробленні даних, отриманих з рівня доступу до даних, шляхом використання поведінкових шаблонів, таких як стратегія, також класів поведінки, які описують логіку оброблення даних,

відхиляючись від канонічних шаблонів, в певних випадках команд. Кожний складовий компонент реалізує вже кінцеву функціональність, яку очікує отримати користувач. Тобто присутній перехід між сутністю що описує таблицю бази даних і сутністю що виходить в ході опрацювання бізнес-логікою. Даний процес є посередником між рівнями області визначення і розподілу даних (domain and data mapping layers), використовуючи інтерфейс, схожий з колекціями для доступу до об'єктів області визначення. Система зі складною моделлю області визначення може бути спрощена за допомогою додаткового рівня, наприклад Data Mapper, який би ізолював об'єкти від коду доступу до БД. У таких системах може бути корисним додавання ще одного шару абстракції поверх шару розподілу даних (Data Mapper), в якому б був зібраний код створення запитів. Це стає ще більш важливим, коли в області визначення безліч класів або при складних, важких запитах. У таких випадках додавання цього рівня особливо допомагає скоротити дублювання коду запитів. Згідно закладеної концепції складові компоненти є розширюються за рахунок засобів додавання конкретних класів з вузько націленою поведінкою.

1.4.1 Паттерн стратегії

Паттерн стратегія спрямований на реалізацію різної поведінки в залежності від компоненту, який знаходиться рівнем вище і звертається до даної стратегії.

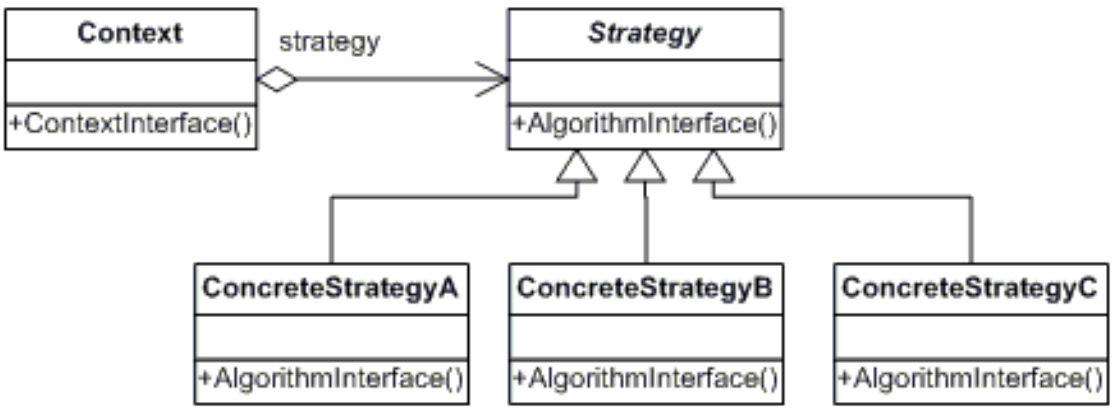

Рис. 1.7. Загальний вигляд патерну стратегії

Як видно з діаграми присутній базовий опис стратегії з відповідним методом, який приймає на вхід необхідні параметри, і відповідно кілька реалізацій даного методу.

1.4.2 Паттерн команди

Сфера застосування патерну команди полягає у здійсненні (виконанні) тієї чи іншої логічної поведінки на вимогу, дозволяючи абстрагуватися від конкретної логіки самої команди всередині механізму.

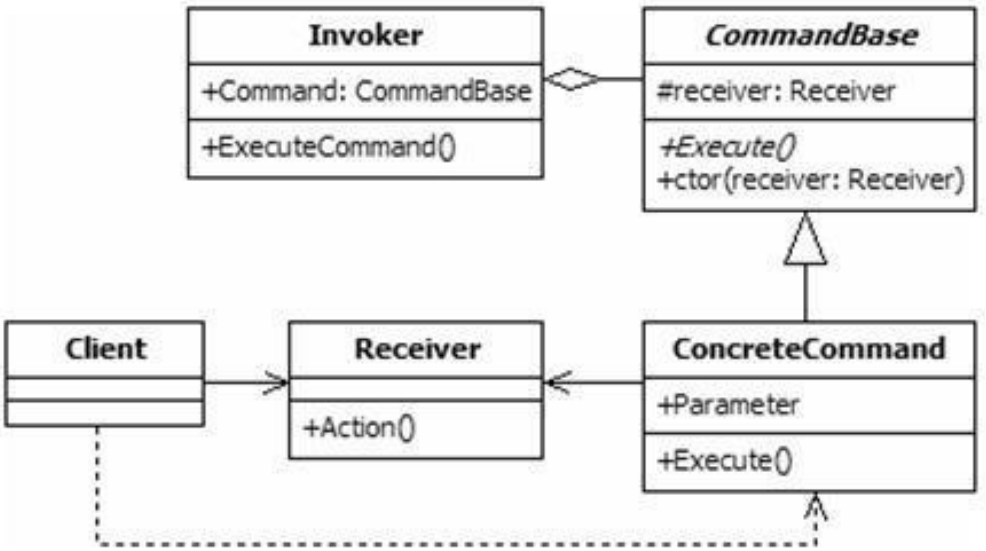

Рис. 1.8. загальний вигляд патерну команди

Ще один механізм, який застосовується, - це інтерфейс з описами поведінки і клас реалізації, що містить елементи обробки даних і формування результату, необхідного для компонента, що знаходиться в архітектурі рівнем вище.

1.4.3 Паттерн ланцюг обов'язків

Оброблювач визначає загальний для всіх конкретних оброблювачів інтерфейс. Зазвичай достатньо описати єдиний метод обробки запитів, але може мати місце оголошення і метод виставлення наступного обробника.

Тобто в ході делегування методу до кінцевої точки за рахунок обходу ланцюжку викликів і передачі відповідальності за рішення до конкретного компоненту якій здатен правильно опрацювати даний запит та на якому припиняється подальший обхід ланцюга.

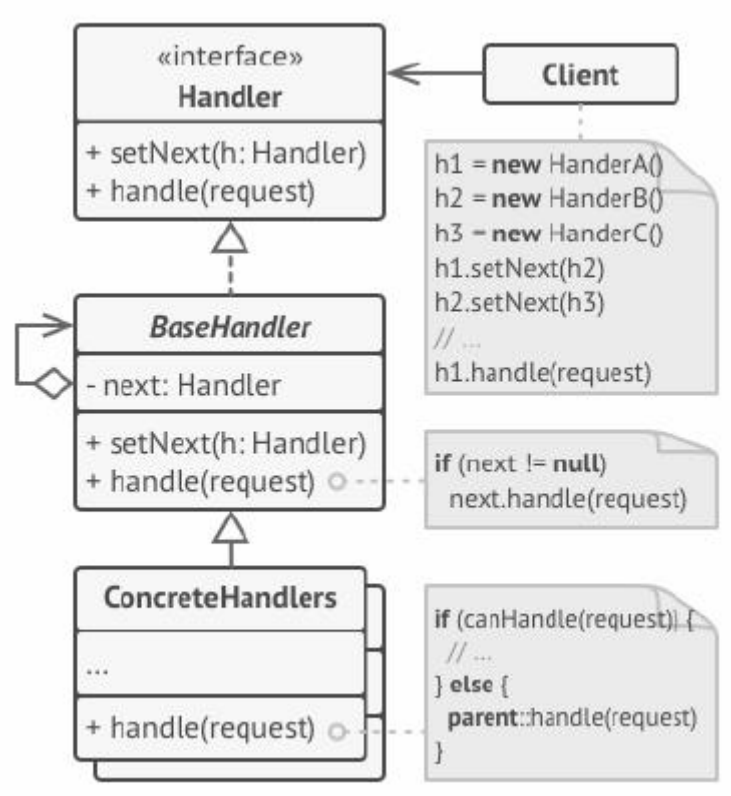

Рис. 1.9. Загальний вигляд патерну ланцюг обов’язків

Даний паттерн застосовується в випадку необхідності динамічного вибору обробника поточної події.

1.4.4 Паттерн посередник

Компоненти - це різноманітні об'єкти, що містять бізнес-логіку програми. Кожен компонент зберігає посилання на об'єкт посередника, але працює з ним тільки через абстрактний інтерфейс посередників. Завдяки цьому, компоненти можна повторно використовувати в іншій програмі, зв'язавши їх з посередником іншого типу.

Посередник визначає інтерфейс для обміну інформацією з компонентами.

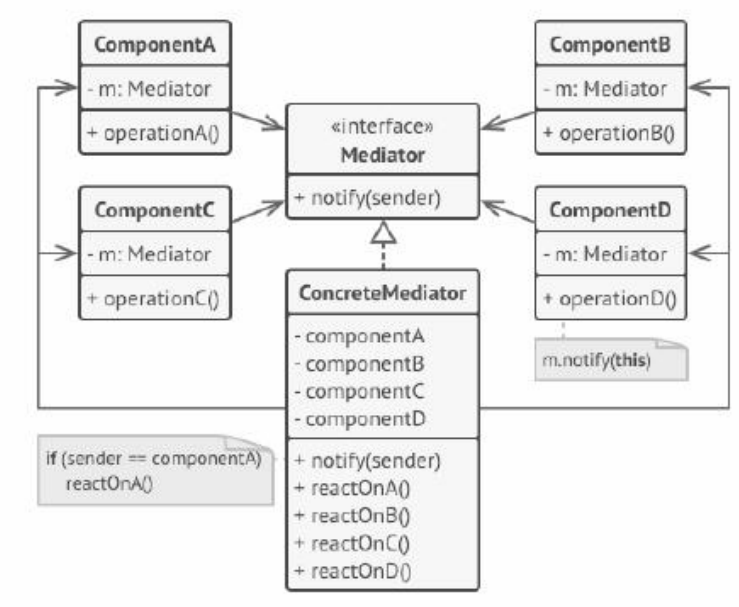

Рис. 1.10. Загальний вигляд патерну посередник

Зазвичай вистачає одного методу, щоб повідомляти посередника про події, що сталися в компонентах. В параметрах цього методу можна передавати деталі події: посилання на компонент, в якому вони є і будь-які інші дані.

Конкретний посередник містить код взаємодії кількох компонентів між собою. Найчастіше цей об'єкт не тільки зберігає посилання на всі свої компоненти, а й сам їх створює, керуючи подальшим життєвим циклом.

Компоненти не повинні взаємодіяти один з одним безпосередньо. Якщо в компоненті відбувається важлива подія, він повинен сповістити свого посередника, а той сам вирішить - чи стосується подія інших компонентів, і чи варто їх сповіщати. При цьому компонент-відправник не знає, хто обробить його запит, а компонент-одержувач не знає, хто його надіслав.

Даний паттерн вирішує задачу абстрагування від жорстких зав’язків та довгих ланцюгів виклику.

Висновки до розділу

В даному розділі, виходячи з аналізу існуючих рішень та наведеного прикладу була сформована наукова актуальність роботи, поставлено завдання

для дослідження та сформовано проблематику, що є підґрунтям для наукової актуальності.

Відповідно до результатів аналізу, було встановлено проблему дослідження та дана точка є відправною для створення рішення, що міститиме переваги рішень, а саме можливість шаблонізації процесу створення додатку з можливістю детального налаштування даного процесу. В поточному розділі було аргументовано загальний підхід декомпозиції глобальної задачі на незалежні одиниці, що піддаються шаблонізації та наведено огляд основних концепцій побудови додатків.

РОЗДІЛ 2.
МЕТОДОЛОГІЯ ПРОЕКТУВАННЯ ВЕЛИКИХ ТА РОЗПОДІЛЕНИХ WEB-ДОДАТКІВ

2.1 Аналіз трендів у сфері розробки програмного забезпечення

Щоб краще зрозуміти концепцію технічної стійкості, ознайомтеся з оглядом якості програмного забезпечення та вимогами , а також того, що було зроблено в попередніх дослідженнях щодо стійкості в розробці програмного забезпечення. Також потрібне розуміння концепцій, які використовуються в дослідженні, таких як підтримуваність програмного забезпечення, моделі процесу розробки програмного забезпечення та проектування ПЗ. По-перше, необхідне базове розуміння подій і змін, що оточують сферу розробки програмного забезпечення.

Щоб зрозуміти, чому програмне забезпечення та процес його розробки набули нинішньої форми, необхідно зрозуміти, як змінювалося його середовище протягом багатьох років. В епоху до персонального комп'ютера програмне забезпечення та його програміст не були в центрі уваги. Це сталося через машини, для яких він був розроблений. Машини займали багато місця, були унікальними і часто був лише один екземпляр машин. Вони також часто ніколи не залишали лабораторію, в якій були розроблені. Програмне забезпечення, розроблене для цих типів машин, повністю залежало від конкретної машини, оскільки кожна машина була унікальною. Оскільки програмне забезпечення повністю залежало від однієї унікальної машини, тривалість життя програмного забезпечення була продиктована тривалістю життя машини [32].

З появою ПК програмне забезпечення можна було розробляти та використовувати набагато більше користувачів, оскільки комп'ютери можна було виробляти у більших масштабах. Проблеми з можливістю повторного використання машин до ПК через їх унікальність були перенесені на

комп’ютери ПК, оскільки вони могли використовувати іншу операційну систему (OS) або платформу. Коли кількість ОС і платформ стабілізувалася, програмісти могли повторно використовувати попередню роботу та знання більшою мірою, ніж раніше. Програмне забезпечення розповсюджувалося серед користувачів на фізичному носії 1, а комп’ютери часто, у випадку споживчого ринку, були ізольовані від інших. Апаратні компоненти для комп’ютерів постійно вдосконалювалися на порядки, що призвело до постійного збільшення продуктивності [32, 70]. Коли Інтернет почав поширюватися та вдосконалюватися, почався перехід до цифрової системи розповсюдження програмного забезпечення. Програмне забезпечення також стало менш ізольованим, оскільки Інтернет дозволив нові способи передачі інформації, де деяке програмне забезпечення стало системами, що використовують архітектуру клієнт-сервер. Всесвітня павутина (WWW) є прикладом такого типу системи з архітектурою клієнт-сервер [11], де клієнт є «браузером», який переглядає файли, що зберігаються на віддаленому (або локальному) сервері.

З початком того, що називають епохою після ПК , коли можливості ПК були реалізовані в інших типах пристроїв. Це відкриває нові можливості для використання програмного забезпечення, коли користувачі мають кілька пристроїв, які діють як клієнти. Клієнти перейшли від простого перегляду файлів до надсилання, отримання, зберігання, зміни та обміну даними. Це поставило перед розробниками програмного забезпечення проблему синхронізації зі складними проблемами, такими як одночасна зміна даних. Клієнти також змінилися і тепер можуть приймати форму програм, розроблених спеціально для платформи/пристрою та веб-програм, які зберігають (у найкращому випадку всі, часто більшість) функціональність рідної програми, але доступні через веб - браузер і має бути незалежною від платформи та пристрою.

Інструменти, які використовуються для створення програмного забезпечення, також змінювалися з роками. Що стосується мов програмування,

розвиток пішов від машинної мови до символічної мови асемблера, а потім до мов високого рівня, які почалися з FORTRAN і COBOT [25]. Парадигми програмування, тобто фундаментальний дизайн комп'ютерного програмування, де зміни пішли від неструктурованого програмування до структурованого [25]. Звідти було створено багато інших парадигм, наприклад, парадигми функціонального, об'єктно-орієнтованого та імперативного програмування. Середовище, у якому можна писати програмне забезпечення, також змінилося: від простих текстових редакторів, де були доступні лише звичайний текст і кілька основних інструментів, до так званого інтегрованого середовища розробки (IDE). Середовище IDE містить кілька утиліт, які спрощують середовище для розробки коду. Прикладами цих утиліт поза редактором вихідного коду (тобто місцем, де ви пишете код) є інструменти автоматизації збірки, налагоджувачі, доповнення коду 3 , системи контролю версій і засоби рефакторинга.

2.2 Якість програмного забезпечення та вимоги

На сучасному етапі розвитку програмної індустрії, при формулюванні вимог використовують ряд технологій. Основними з них є: технологія, яка використовує рекомендації стандарту IEEE 830, структурна та об'єктно-орієнтована. Дані технології мають ряд недоліків, основним з яких є відсутність формалізованого апарату представлення вимог, який дозволив би адекватно та однозначно відобразити потреби замовника ПС на вимоги.

Наслідком відсутності такого апарату при розробці вимог є неоднозначність трактувань сформульованих вимог різними сторонами, що беруть участь в проекті, високий ступінь впливу суб'єктивних факторів, що призводить до негативних результатів на наступних стадіях розробки.

Одним із визначень якості програмного забезпечення є «здатність програмного продукту задовольняти заявлені та неявні потреби при використанні в певних умовах». Часто використовуваною моделлю якості

програмного забезпечення є стандарт ISO/IEC 25010 або його попередник ISO/IEC 9126 [51, 52], див., наприклад , [22, 23, 41, 44]. Модель 25010 складається з восьми якостей продукту, які можна побачити на рисунку 1.2.

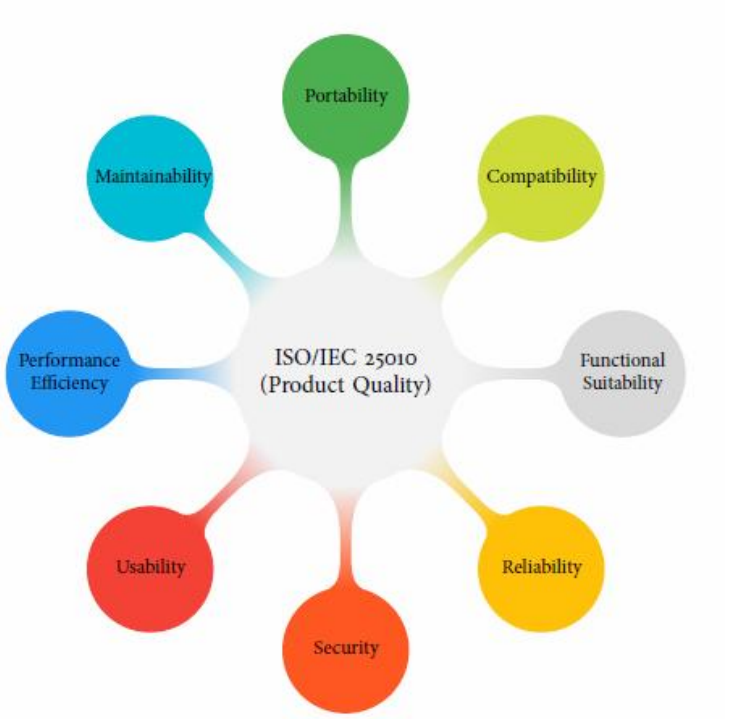

Рис. 2.1. Вісім якостей продукції стандарту ISO/IEC

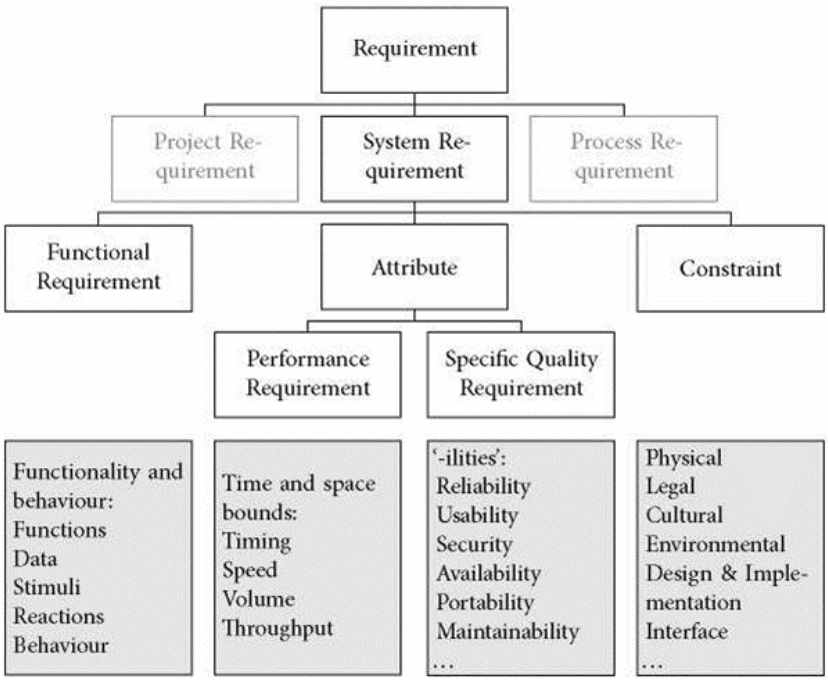

Рис. 2.2. Таксономія вимог

У глосарії IEEE було визначено лише термін функціональні вимоги. Нефункціональні вимоги не були визначені, доки в 2010 році не було опубліковано наступну (поточну) версію глосарію IEEE/ISO/IEC. Нефункціональні вимоги визначаються як «вимога до програмного забезпечення, яка описує не те, що програмне забезпечення буде робити». але як він це зробить», а функціональна вимога визначається як «заява, яка визначає, чого має виконати продукт або процес, щоб забезпечити необхідну поведінку та/або результати». Це схоже на те, що називає правилом, яке часто чують, згідно з яким те, що робить система, є функціональними вимогами, а те, як система поводиться, є нефункціональними вимогами. В [22] запропонували у своєму дослідженні мову моделювання для нефункціональних вимог, яка розглядає нефункціональні вимоги як вимоги над якостями. Метою цієї мови моделювання було вирішення проблем із характеристиками деяких нефункціональних вимог, таких як невизначеність, суб'єктивність і можливість вимірювання.

Широко вживаним визначенням сталого розвитку є «задоволення потреб сьогодення без шкоди здатності майбутніх поколінь задовольняти власні потреби». Для досягнення сталого розвитку існують вимоги в трьох вимірах: економіка, навколишнє середовище та суспільство, які мають бути виконані. Пізніше було стверджено, що Гудленд розширив це, щоб також включити людський вимір, який розглядає розвиток кожної окремої людини протягом її життя, оскільки це є основою для трьох інших вимірів. Ці аспекти відображені у виразі стійкого програмного забезпечення, сформульованому Діком, Науманном і Куном [30]

Визначення 1. «Екологічне та стійке програмне забезпечення — це програмне забезпечення, прямий і непрямий негативний вплив якого на економіку, суспільство, людей і навколишнє середовище внаслідок розробки, розгортання та використання програмного забезпечення є мінімальним та/або яке має позитивний вплив вплив на сталий розвиток».

Науман далі зазначає, що для досягнення екологічного та стійкого програмного продукту організація, яка його розробляє, повинна усвідомлювати негативний і позитивний вплив на сталий розвиток, який може мати місце під час використання продукту. Крім того, весь процес розробки продукту має бути екологічним і екологічним. Науман та ін. висловив процес розвитку щодо цих двох аспектів, як видно з визначення.

Визначення 2. «Екологічна та стійка інженерія програмного забезпечення – це мистецтво розробки екологічно безпечного програмного забезпечення за допомогою екологічного та екологічно чистого процесу розробки. Таким чином, це мистецтво визначення та розробки програмних продуктів таким чином, щоб негативні та позитивні впливи на сталий розвиток, які виникають та/або очікуються в результаті програмного продукту протягом усього його життєвого циклу, постійно оцінювалися, документувалися, і використовується для подальшої оптимізації програмного продукту».

Існує чотири виміри стійкості та визнав чотири аспекти стійкості в розробці програмного забезпечення:

- Процес розробки
- Аспект процесу обслуговування
- Аспект виробництва системи
- Аспект використання системи

Чотири аспекти охоплюють дві точки зору, перші два аспекти охоплюють точку зору процесу розробки, а останні два аспекти охоплюють точку зору продукту під час виробництва та використання. Ці дві точки зору дуже схожі на два визначення Діка, Науманна та Куна та Науманна та ін. [73], оскільки одна частина обох охоплює продукт, а інша частина охоплює процес розробки продукту.

Зазвичай під час розробки програмного забезпечення розробка прогресує, щоб відповідати функціональним і нефункціональним вимогам, установленим для програмного забезпечення. Функціональні вимоги встановлюються таким чином, щоб кінцеве програмне забезпечення включало

всі робочі цілі, які від нього очікуються, наприклад функції та функції. Нефункціональні вимоги зосереджують увагу на аспектах якості програмного забезпечення, наприклад на доступності, точності, ремонтопридатності, надійності та повторному використанні. [3] На відміну від багатьох інших інженерних дисциплін, існує відсутність повідомлень про стійкість як стандартизовану практику в рамках дисципліни програмної інженерії, і припускають, що сталість має бути «першокласною якістю програмного забезпечення».

В [22] представили модель якості на основі ISO/IEC 25010, яка включає сталість як аспект якості. Причиною модифікації ISO/IEC 25010 є те, що він має три такі характеристики [22]:

1. Характеристики, які самі по собі можна визнати пов’язаними зі стійкістю;
2. Характеристики, які не можна розглядати як характеристики елемента 1 , але можуть мати прямий вплив на сталість;
3. Характеристики, де аспекти стійкості не є прийнятними та не потребують застосування.

Виходячи з цих трьох типів, автори вважають, що ті класифіковані як Характеристики пункту 1 представляють собою початкову спробу включити аспекти стійкості в стандарт. Ті, що класифікуються як Характеристики пункту 2 слід змінити на стійку версію, оскільки ці характеристики пов’язані з такими факторами, як характеристики та функціональність продукту. За характеристиками, класифікованими як Характеристики пункту 3 було визнано недоцільним прийняти стійку версію, оскільки вони, як правило, більше пов’язані з можливостями продукту. На основі цього в [22] створили запропоновану модель 25010+S, розширення стандарту ISO/IEC 25010, яка розглядає аспекти сталого розвитку на різних характеристиках.

Потім виконали систематичний огляд літератури щодо заходів стійкості програмного забезпечення, щоб виявити їх сучасність і зробити модель якості 25010+S корисною. Автори знайшли 61 показник, який вважався корисним для

моделі якості 25010+S. З цих 61 показника 59 стосувалися характеристик якості продукту, і лише два показники стосувалися сталого розвитку. Заходи, пов'язані з якістю продукції, охоплювали лише п'ять характеристик: ремонтопридатність, продуктивність, портативність, надійність і зручність використання, і більшість з них були зосереджені на споживанні енергії. Автори дійшли висновку, що потрібна подальша робота, щоб знайти та створити нові міри для всіх характеристик якості в моделі 25010+S.

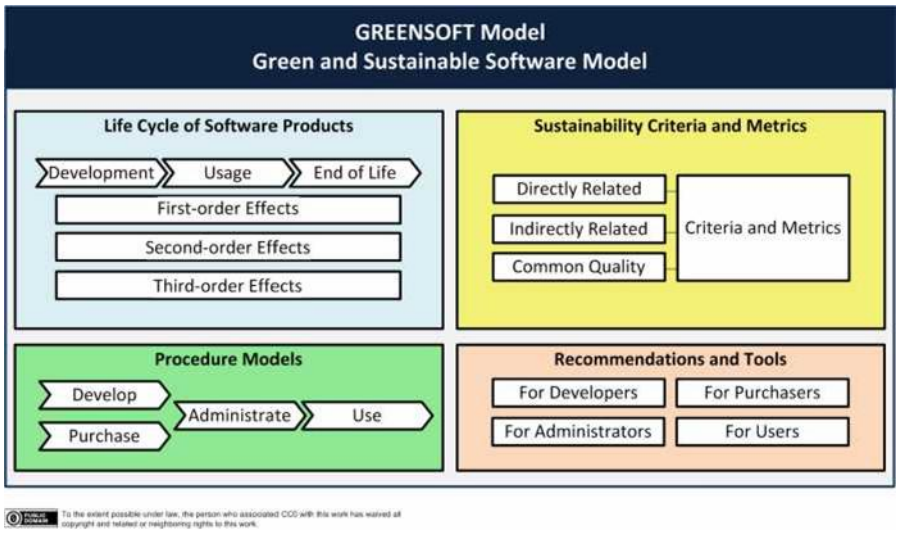

Рис. 2.3. Еталонна Модель GREENSOFT

Модель GREENSOFT — це підхід до розробки програмного забезпечення, який спеціально включає аспекти стійкості для створення, підтримки та використання програмного забезпечення. Він призначений як еталонна модель для стійкого програмного забезпечення та може бути інтегрований у різні моделі процесів, наприклад Scrum та OpenUP [56]. Еталонна модель GREENSOFT складається з чотирьох частин (рис 2.3): «Життєвий цикл програмних продуктів», «Критерії та показники стійкості», «Моделі процедур». Його мета полягає в тому, щоб дозволити зацікавленим сторонам проекту оцінити вплив проекту на основі трьох впливів (таблиця 2.1) , визначених в [10]. Ці впливи зосереджені в основному на екологічних аспектах, але можуть бути уточнені для розгляду як людських, так і соціальних аспектів стійкості в ІКТ. Зазначають, що важливо, щоб зацікавлені сторони були

обережними під час оцінювання, щоб усі рівні були належним чином враховані, як висловив в [1]: «Звичайно, легше зосередитися на впливах першого порядку, оскільки вони негайні та очевидні. Однак впливи третього порядку можуть бути найнебезпечнішими, і до них найважче наблизитися».

Одну критику цієї моделі зробили в [25], який зазначає, що хоча це може бути єдиним підходом у розробці програмного забезпечення, який чітко розглядає аспект стійкості, він має надто специфічне застосування у веб-інженерії. Крім того, вони стверджують, що в галузі відсутня всеохоплююча базова основа для стійкого проектування.

Таблиця 2.1.

Основні впливи ІКТ на навколишнє середовище

ORDER	DESCRIPTION
First order impacts	Direct environmental effects through the production and use of ICT, i.e. resource use and pollution from the hardware production process, energy consumption during usage and the disposal of the hardware devices and components.
Second order impacts	Indirect environmental effects through the use of ICT. There are different types of these effects both positive and negative. The two main types of positive effects are dematerialisation (process optimisation, i.e. larger output for smaller resource input) and substitution (e.g. substituting material products for equivalent immaterial parts) effects
Third order impacts	The indirect long-term environmental effects from the usage of ICT due to rebound effects, e.g. changes that stimulates economic growth and more consumption thus in worst cases outweighing the initial savings

2.3 Дослідження концепції ремонтоздатності програмного забезпечення

Існує кілька визначень ремонтопридатності програмного забезпечення. Згідно з IEEE [49], придатність до експлуатації можна визначити як: «Легкість, з якою програмна система або компонент може бути модифікована для виправлення помилок, покращення продуктивності чи інших атрибутів або адаптації до зміненого середовища». Зазначають, що ремонтопридатність програмного забезпечення є характеристикою програмного забезпечення, яка представляє кількість зусиль, необхідних для досягнення чотирьох завдань:

виправлення помилок, додавання функцій, видалення можливостей та адаптація/модифікація. У стандарті ISO/IEC 25010 ремонтопридатність уточнюється на п’ять додаткових якостей, як показано на рисунку 2.4.

Рис. 2.4. Ремонтопридатність та її вдосконалення в стандарті 1 SO/ 1 EC 25010

Ремонтопридатність і технічне обслуговування програмного забезпечення пов’язані одне з одним, оскільки технічне обслуговування програмного забезпечення – це процес підтримки та зміни програмного забезпечення після його розробки. Фаза обслуговування часто є найбільш трудомісткою і дорогою [40] частиною розробки програмного забезпечення. Зосередивши підтримку програмного забезпечення на етапі розробки, можна значно заощадити ресурси. Стверджують, що дослідження ремонтопридатності є занадто цілеспрямованими з точки зору програмного продукту. Автори стверджують, що для отримання повної картини питання ремонтопридатності необхідно провести систематичне дослідження, яке охоплює всіх людей, організації та процеси, пов’язані з програмним забезпеченням. Дослідження виявили різні типи причин проблем ремонтопридатності, наприклад, технічний борг

Технічний борг

Метафора «Технічний борг» була введена Каннінгемом [27], щоб вказати, що «не зовсім правильний код, який ми відкладаємо, виправляючи

його». Метафора використовується для пояснення компромісів між наданням найбільш відповідного — але, ймовірно, незрілого — продукту в найкоротші терміни. Ця «економія» в короткостроковій перспективі натомість може з’явитися як додаткові витрати у формі технічних боргових відсотків пізніше на етапі обслуговування. Метафора технічного боргу описує, як економія в роботі над нефункціональними вимогами (тобто зручність обслуговування, читабельність тощо) у процесі може з’явитися пізніше з більшими витратами.

Наведено кілька прикладів наслідків, які можуть виникнути, коли технічний борг не буде «погашений». Ці наслідки можуть бути у вигляді перевищення бюджетів, проблем з якістю, проблем із додаванням нових функцій без проблем із наявними функціями, а також того, що програмне забезпечення не витримає очікуваний життєвий цикл. Були зроблені спроби визначити таксономію технічного боргу. Поділяють технічний борг на навмисний і ненавмисний. Навмисний борг – це борг, який був навмисно створений, наприклад, відкладення належного впровадження іменованих констант і замість цього використання «магічних чисел» 4. Навмисну заборгованість також можна поділити на короткострокову та довгострокову Ненавмисна заборгованість – це заборгованість, яка несвідомо створюється через роботу низької якості.

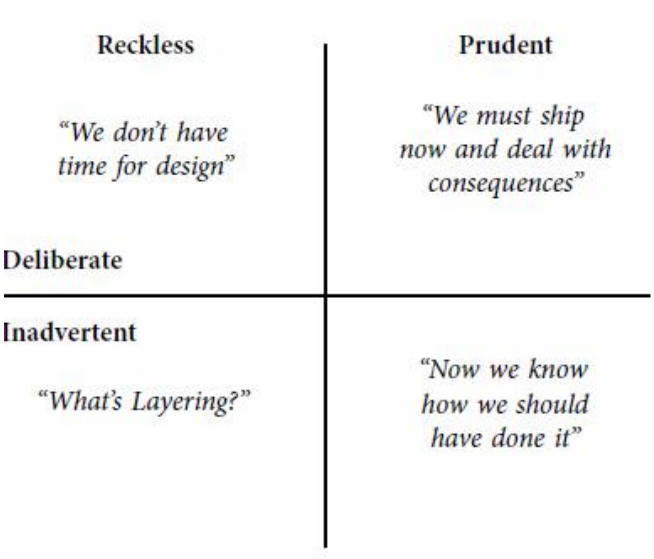

Рис. 2.5. Квадрант технічного боргу

Унікальні значення, які або використовуються в кількох місцях, або значення яких не пояснюється.

В [34] створено квадрант технічного боргу, який має два виміри: безрозсудний/обачливий і навмисний/ненавмисний, як показано в Малюнок 6 . З цього квадранту можна виділити чотири типи технічного боргу: необачний і навмисний борг, обачний і навмисний борг, нерозважливий і ненавмисний борг, розсудливий і ненавмисний борг.

Таблиця 2.2.

Типи технічної заборгованості

TYPE	DEFINITION
Architecture Debt	Refers to the problems encountered in project architecture, for example, violation of modularity, which can affect architectural requirements (performance, robustness, among others). Normally this type of debt cannot be paid with simple interventions in the code, implying in more extensive development activities.
Build Debt	Refers to build related issues that make this task harder, and more time/processing consuming unnecessarily. The build process of a project can contain very unnecessary code to the customer. Moreover, if the build process needs to run ill-defined dependencies, the process becomes unnecessarily slow. When this occurs, one can identify a build debt.
Code Debt	Refers to the problems found in the source code which can affect negatively the legibility of the code making it more difficult to be maintained. Usually, this debt can be identified by examining the source code of the project considering issues related to bad coding practices.
Defect Debt	Software projects may have known and unknown defects in the source code. Defect debt consists of known defects, usually identified by testing activities or by the user and reported on bug track systems, that the change control board agrees should be fixed, but due to competing priorities, and limited resources have to be deferred to a later time. Decisions made by the change control board to defer addressing defects can accumulate a significant amount of technical debt for a product making it harder to fix them later.
Design Debt	Refers to debt that can be discovered by analyzing the source code by identifying the use of practices which violated the principles of good object-oriented design (e.g. very large or tightly coupled classes).
Documentation Debt	Refers to the problems found in software project documentation and can be identified by looking for missing, inadequate, or incomplete documentation of any type. Inadequate documentation is those that currently work correctly in the system, but fail to meet certain quality criteria of software projects.
Infrastructure Debt	Refers to infrastructure issues that, if present in the software organization, can delay or hinder some development activities. Some examples of this kind of debt are delaying an upgrade or infrastructure fix.
People Debt	Refers to people issues that, if present in the software organization, can delay or hinder some development activities. An example of this kind of debt is expertise concentrated in too few people, as an effect of delayed training and/or hiring.
Process Debt	Refers to inefficient processes, e.g. what the process was designed to handle may be no longer appropriate.
Requirement Debt	Requirements debt refers to tradeoffs made with respect to what requirements the development team need to implement or how to implement them. Some examples of this type of debt are: requirements that are only partially implemented, requirements that are implemented but not for all cases, requirements that are implemented but in a way that doesn't fully satisfy all the non-functional requirements (e.g. security, performance, etc.).
Service Debt	The need for web service substitution could be driven by business or technical objectives. The substitution can introduce a technical debt, which needs to be managed, cleared and transformed from liability to value-added. Technical debt can cover several dimensions, which are related to selection, composition, and operation of the service.
Test Automation Debt	Test Automation debt is defined as the work involved in automating tests of previously developed functionality to support continuous integration and faster development cycles.
Test Debt	Refers to issues found in testing activities which can affect the quality of testing activities. Examples of this type of debt are planned tests that were not run, or known deficiencies in the test suite (e.g. low code coverage).

Існує опис технічної заборгованості у різних сферах, наприклад - заборгованість за кодом, дефектом, документацією чи тестуванням. Подібним

чином технічний борг класифікували, який визначив 13 типів технічного боргу за допомогою онтології (таблиця 2.2) . Зазначають, що життєвий цикл процесу програмного проекту може вплинути на суму технічного боргу. Таким чином, гнучкий процес розробки програмного забезпечення створить менше технічного боргу завдяки своїй гнучкій та адаптивній структурі порівняно зі статичною водоспадною моделлю. Через ітераційний процес agile може виникнути переконання, що він захищений від технічного боргу. Однак, як зазначають в [7], швидка розробка та впровадження без часу на належне проектування чи обмірковування довгострокової перспективи, а також відсутність ретельності чи систематичного тестування може призвести до великих сум боргу швидко, незважаючи на можливість ітераційного процесу дозволяє відшкодувати борг.

Для прийняття рішення про те, коли «погасити» борг, пропонують чотири різні підходи до прийняття рішень щодо пріоритетності технічного боргу: підхід аналізу витрат і вигод, процес аналітичної ієрархії, підхід портфеля та підхід опціонів. Стверджують, що рішення про вирішення або відстрочення боргу залежить від накопиченої кількості дефектів. Також було визначено кілька факторів, які впливають на рішення, представлених нижче в порядку зменшення важливості:

- Наявність обхідного шляху
- Терміновість ремонту на вимогу замовника
- Зусилля для впровадження виправлення
- Ризик запропонованого виправлення
- Необхідний обсяг тестування

2.4 Моделі процесу розробки програмного забезпечення та web-додатків

Модель процесу розробки програмного забезпечення — це опис методології, яка має на меті виступати в якості керівництва для інженерів

програмного забезпечення щодо того, як створити повний програмний продукт високої якості. Методологія описує дії та кроки, яких інженери програмного забезпечення повинні дотримуватися, щоб досягти цього. Характеристики цих процесів змінювалися та розвивалися протягом багатьох років.

Початкові «традиційні» процеси були похідними від процесів управління проектами в інших галузях, які були структурованими процесами з послідовними видами діяльності. Ці процеси мали передбачувану характеристику у своїй структурі, де рішення були ретельно задокументовані та сплановані заздалегідь, тобто підхід, керований планом. Початкова модель називається «модель водоспаду», назва моделі походить від її подібності (в графічному зображенні) до набору невеликих серійних водоспадів (рис. 2.6).

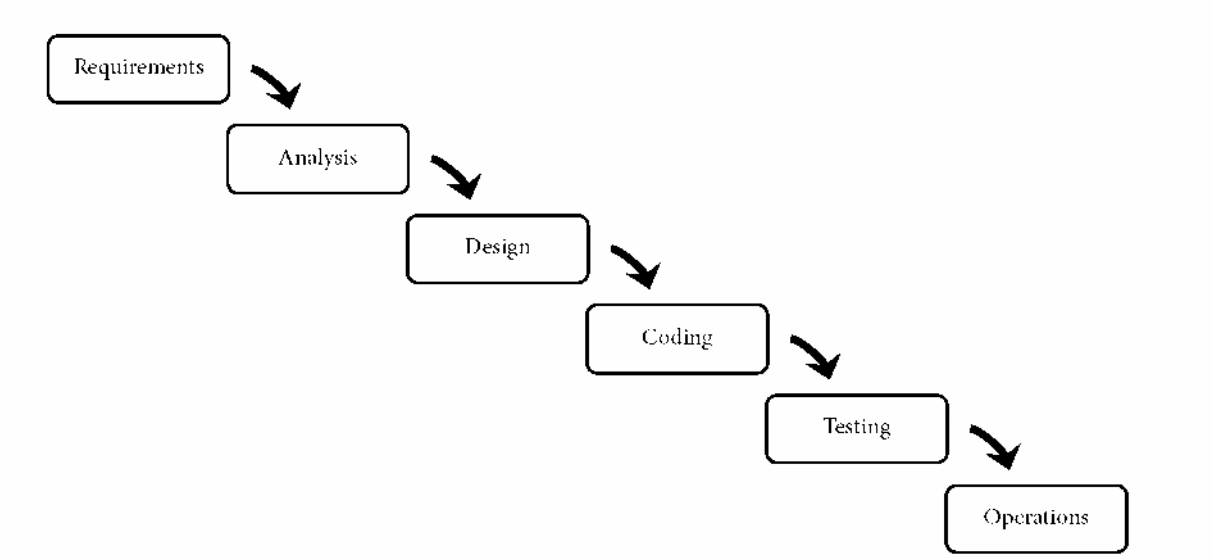

Рис. 2.6. Графічний образ моделі водоспаду

Модель водоспаду набула популярності в індустрії розробки програмного забезпечення, оскільки її було легко зрозуміти та реалізувати. Модель водоспаду завдяки своїм прогнозним характеристикам робить наголос на понятті визначення перед проектуванням і проектування перед кодом. Це поняття було включено в структуру, оскільки кожна дія повинна була бути завершена до того, як можна було розпочати наступну. Після розробки моделі водоспаду були розроблені інші моделі, і кожна модель була розроблена для

того, щоб дозволити інженерам програмного забезпечення успішно розробляти програмне забезпечення, наприклад, модель швидкого прототипування, спіральна модель, базова модель компонентів, модель Rational Unified Process (RUP) і гнучка модель. Кожна модель має різні підходи до успішного програмного проекту. Остання згадана модель, модель Agile, є моделлю, яка протягом останнього десятиліття набула популярності як переважна модель для програмних проектів

Гнучка модель

Гнучка модель була створена для використання в сферах, які вимагають швидкої адаптації до змін і гнучкості. У розробці програмного забезпечення гнучка методологія дотримується чотирьох основних цінностей, проголошених у «Маніфесті гнучкої розробки програмного забезпечення» [8]:

1. Особи та взаємодія процесів та інструментів
2. Працююче програмне забезпечення над повною документацією
3. Співпраця з клієнтом над узгодженням контракту
4. Реагування на зміни замість дотримання плану

Де твердження ліворуч від слова «понад» у пунктах цінуються більше, ніж ті, що розташовані праворуч [8]. «Маніфест гнучкої розробки програмного забезпечення» був створений у 2001 році кількома лідерами руху гнучкої розробки програмного забезпечення. Цей маніфест призначений як керівництво для розробників програмного забезпечення щодо гнучких методологій. Маніфест формулює на додаток до згаданих основних цінностей наступні дванадцять принципів, яких мають дотримуватися інженери
програмного забезпечення:

1. Наш найвищий пріоритет — задовольнити потреби клієнта шляхом швидкої та безперервної доставки цінного програмного забезпечення.
2. Ласкаво просимо до зміни вимог, навіть на пізній стадії розробки. Гнучкі процеси використовують зміни для конкурентної переваги клієнта.

3. Доставляйте робоче програмне забезпечення часто, від кількох тижнів до кількох місяців, віддаючи перевагу коротшим часовим рамкам.

4. Бізнесмени та розробники повинні працювати разом щодня протягом усього проекту.

5. Створюйте проекти навколо мотивованих людей. Забезпечте їм необхідне середовище та підтримку та довірте їм виконання роботи.

6. Найефективнішим і дієвим методом передачі інформації групі розробників і всередині неї є розмова віч-на-віч.

7. Працююче програмне забезпечення є основним показником прогресу.

8. Гнучкі процеси сприяють сталому розвитку. Спонсори, розробники та користувачі повинні мати можливість підтримувати постійний темп нескінченно довго.

9. Постійна увага до технічної досконалості та гарного дизайну підвищує маневреність.

10. Простота — мистецтво максимізувати кількість невиконаної роботи — є важливою.

11. Найкращі архітектури, вимоги та проекти виходять із самоорганізованих команд.

12. Через регулярні проміжки часу команда розмірковує над тим, як стати більш ефективною, а потім відповідно налаштовує та коригує свою поведінку.

Гнучкі методології розробки програмного забезпечення характеризуються ітераційною розробкою, постійною інтеграцією коду та здатністю обробляти зміни у вимогах. Існує багато різних методологій, таких як Scrum, Extreme Programming (XP), Crystal Clear, Kanban development, Lean software development, Feature Driven Development (FDD) і Test Driven Development (TDD), і хоча вони мають різні види діяльності, вони спільні загальні характеристики, наприклад ітераційне прототипування та мінімальна документація.

Гнучка модель була запропонована як альтернатива керованим планом методам, якими розробники були незадоволені, і як спосіб створення високоякісного програмного забезпечення за більш короткий період часу. Головна мета гнучкого процесу — можливість вносити зміни в будь-який час у процесі розробки. Це означає, що бізнес-вимоги та деталі дизайну, які в керованих планом моделях визначалися на початковій стадії, не вимагалися приймати рішення на початку процесу, а були відкритими для змін і вдосконалень. Стверджують, що гнучкі моделі забезпечать кращу якість програмного забезпечення та коротший час виробництва, ніж керовані планом моделі. Цей аргумент знайшов підтримку в деяких дослідженнях, де було показано, що гнучкі моделі були корисними для зменшення кількості дефектів у програмному забезпеченні. Гнучкі моделі можуть підвищити продуктивність, проти гнучких моделей висувається критика щодо того, що хоча продуктивність і витрати на етапі розробки покращуються, витрати насправді можуть бути перенесені на етап обслуговування. Досліджень з цього приводу мало. Дослідження Scrum показало, що Scrum може не призвести до меншої щільності дефектів, ніж метод, керований планом, але він дозволив збільшити шанси на успіх проекту. Проте дослідження також дійшло висновку, що Scrum створює більш стресовий досвід для розробників, щоб забезпечити функціональні можливості вчасно та за бюджетом, ніж метод, керований планом і процесом. Тому розробники можуть знехтувати виконанням певних дій, які б зменшили потенційні зусилля на етапі обслуговування.

Висновки до розділу

В даному розділі представлено методологію проектування великих та розподілених web-додатків, Виконано аналіз трендів у сфері розробки програмного забезпечення, розглянуто аспекти якості програмного забезпечення та вимоги. Наведені моделі процесу розробки програмного забезпечення та web-додатків.

РОЗДІЛ 3.
РЕАЛІЗАЦІЯ МЕТОДОЛОГІЇ ПОБУДОВИ АРХІТЕКТУРИ ВЕЛИКИХ ТА РОЗПОДІЛЕНИХ КОРПОРАТИВНИХ WEB-ДОДАТКІВ

3.1 Огляд архітектури сучасного корпоративного додатку

Побудова сучасних корпоративних додатків, має на увазі, формування єдиної системи, робота якої направлена на поєднання певної кількості модулів взаємодії чи окремих підсистем. Кількість та склад модулів залежить від кінцевої мети розробки та переліку функцій покладених на корпоративну систему. При значній кількості модулів інтеграція між ними безпосередньо вимагає підтримки багатьох зовнішніх інтерфейсів. В результаті з'являються інтеграційні платформи, які закладаються в основу корпоративної інфраструктури додатків.

Node.js виступає платформою з’єднання повного переліку модулів та підсистем, яка надає можливість JavaScript взаємодіяти з пристроями вводу-виводу через свій API, мовою написання якого є C++, підключати інші зовнішні бібліотеки, написані різними мовами, забезпечуючи виклики до них із JavaScript-коду.

Однією з багатьох переваг Node.js є його архітектура, яка дозволяє легко використовувати його як виразну, функціональну мову для програмування на стороні сервера [17]. Хоча розробка на Node.js може бути тривіальною, коли розробник повністю осягає мову, для початківців, щоб ефективно використовувати Node.js, існує багато перешкод перед тим, як вони зможуть реалізувати реальні хмарні програми. Розробнику варто обовьзково враховувати, що Node.js застосовується переважно на сервері, виконуючи роль веб-сервера, але є можливість розробляти на Node.js та десктопні віконні програми (за допомогою NW.js, AppJS або Electron для Linux, Windows та

macOS) і навіть програмувати мікроконтролери (наприклад, tessel, low.js та espruino).

Node.js, як функціональну одиницю, розроблено знизу вгору, щоб обробляти асинхронний запит на введення-вивід, оскільки він побудований за допомогою JavaScript, а JavaScript розроблений як цикл подій. У клієнтському JavaScript подія натискання кнопки є циклом подій.

Хоча інші середовища пропонують цю функцію, вона виконується за допомогою сторонніх бібліотек або не розробляється з нуля для тієї ж мети, що й Node.js, тому вона часто працює повільно або відстає і не належить до стандартної функції. Деякі приклади включають Event Machine, який був створений для Ruby, Twisted, та був представлений для Pythn і доступний з Python 2 і далі, і Apache MINA, який також відомий як «Бібліотека мережевих сокетів» і є ще одним прикладом надання подій, керований та асинхронний, обмежений лише API. Таким же чином Apache AsyncWeb створюється за допомогою Apache MINA та Perl Any Event. Аналогічно, однією з переваг Node.js перед іншими буде його здатність обробляти багато запитів, одночасно діючи як клієнт сторонніх служб, керуючи лише одним потоком. Інші мови, у цьому сенсі, припинять обробку, доки віддалений сервер не відповість на їхній початковий запит, що вимагатиме багатопотокової обробки для виконання. Для порівняння, все, що буде використано у Node, є асинхронним, тому що писати в ньому неасинхронний код буде складно. Крім того, Node.js ніколи не вимагатиме буферизації даних перед їх виведенням, тоді як інші, такі як Event Machine, вимагатимуть обов’язкової буферизації враховуючи масштабність обставин.

Будучи на стороні сервера JavaScript, ще одна помітна перевага Node.js перед іншими полягає в тому, що від розробника потрібно мати знання та досвід лише однієї мови, а саме JavaScript, незалежно від того, чи створює він сценарії на стороні клієнта чи на стороні сервера.

Розробник не зобов'язаний перемикати свій мозковий цикл з однієї мови на стороні клієнта на іншу мову на стороні сервера. Крім того, оскільки Node.js

молодий, він має перевагу, що уникає помилок, які робили інші мови в минулому, такі як помилка зворотної сумісності. Згідно зі статистикою, близько 47 % онлайн-серферів очікують, що сторінка завантажиться протягом 2 секунд, а затримка в 3 секунди знижує задоволеність споживачів на 16 %. Тут лідирує Node.js, оскільки його інтерпретатор менший і швидший, ніж в інших мовах, таких як PHP. На відміну від інших мов, де кожен запуск програми буде виконувати цикли, що потребують таких процесів, як налаштування завантаження, після чого підключення до бази даних, отримання важливої інформації і, нарешті, відтворення мови розмітки, тут програми на стороні сервера завжди підтримуються увімкненими. Node.js, з іншого боку, мінімізує ці кроки, залишаючи програму завжди увімкненою.

Аналізуючи сучасні дослідження [11, 14, 15], варто наголосити, що платформі в цілому бракує стандартів якості коду, і аргументація через файл, який був відредагований різними розробниками, може бути складною. Більше того, оскільки фреймворк є відносно молодим, а програмне забезпечення все ще досягає зрілості, порівняно з традиційними фреймворками, розробники стикаються з відсутністю підтримки документації і можуть зіткнутися з проблемами при отриманні підтримки від спільноти розробників, що дійсно є проблемою. Оскільки він використовує парадигму, керовану подіями/зворотним викликом, код Node.js швидко розвивається, а також його важко налагоджувати. Відсутність готового хостингу для середовищ Node.js наразі є серйозним недоліком. Складні теми в мові JavaScript, такі як успадкування прототипів, анонімні функції та зворотні виклики, ускладнюють вивчення мови, і, як наслідок, її краще вивчати після оволодіння іншою простою мовою. Node.js все ще молода мова, і, як наслідок, професійні програмісти не можуть приєднатися. Інша проблема полягає в тому, що оскільки система є однопоточною, інші запити припиняються, якщо центральний процесор зайнятий навіть на частку секунди. В результаті розробники також змушені мислити в асинхронних термінах, до яких важко адаптуватися.

На сьогодні, ще не існує єдиного стандарту, щодо архітектури програмного забезпечення Node.js чи керівництва щодо впровадження, за яким розробники мали б впливати на створення добре структурованих додатків. Проте, беззаперечним фактом є те, що організація дизайну програми з чіткими межами має вирішальне значення для успішного розгортання в реальному світі. Розробка неструктурованої моделі спричинить логічну складність і труднощі в підтримці й оновленні програми з часом. Більше того, без затверджених інструкцій для налаштування базового середовища розробки додатків потрібен значний час і зайві зусилля при розробці.

Підхід до розробки архітектури сучасного корпоративного додатку на Node.js ґрунтується на застосуванні двох окремих напрямків. Перший напрямок, це «зверху вниз» за відповідними ролями, допомагає розробникам об’єднувати та роз’єднувати модулі, дана структурна організація сформована у чіткому ієрархічному порядку. Це дозволяє розробникам спостерігати за перебігом програми від основного модуля, який відіграє роль основного класу в Java, до інших підмодулів зверху вниз. По суті, це розбивка сучасного корпоративного додатку, щоб отримати уявлення про його композиційні підмодулі. З іншого боку, підхід «розділяй і володарюй» у відповідності до виконуваних завдань, допомагає розробникам розділити завдання на простіші модулі, одночасно дозволяючи розробляти декілька модулів. Початковим підходом до розробки сучасного корпоративного додатку на Node.js є підхід «зверху вниз», а потім підхід «розділяй і володарюй» на наступному кроці, так що на початку визначаються ролі між модулями, перш ніж завдання призначаються кожній ролі, як показано на рисунку 3.1.

Існують різні типи хмарних додатків, які можуть скористатися перевагами запропонованої архітектури сучасного корпоративного додатку на Node.js та рекомендацій щодо впровадження.

Приклади включають програми, які просто реєструють інформацію в хмарі через Інтернет, обмінюються медичною інформацією або інформацією

про медичне обслуговування за допомогою мобільного додатка, а також виконують дії фізичних компонентів у локальній мережі ad-hoc.

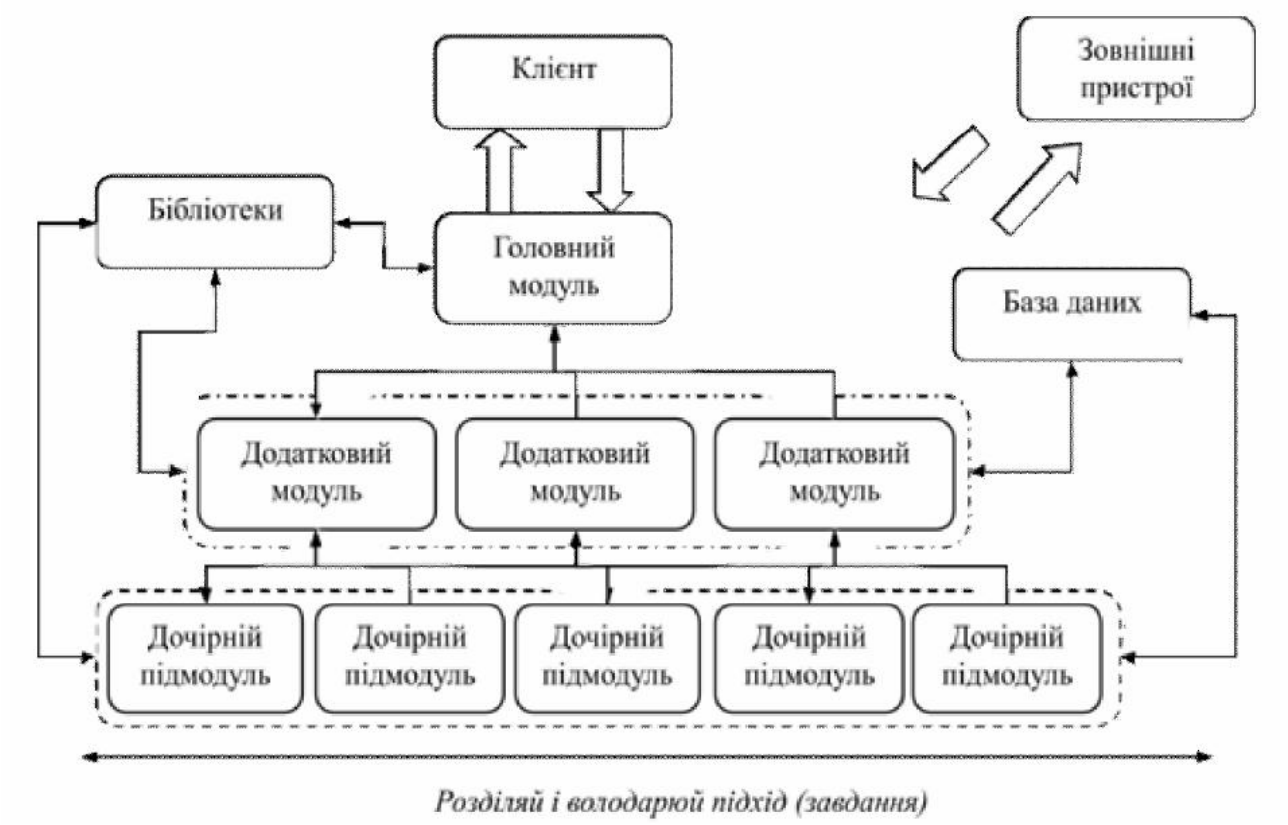

Рис. 3.1. Огляд архітектури сучасного корпоративного додатку на Node.js

Загалом, запропонована розробка може бути застосована до програмування та формування хмарних додатків, які вимагають основи встановлення фундаментальних вимог. Ці вимоги включають надсилання та отримання даних, наявність бази даних для зберігання інформації, виклик зовнішніх виконуваних файлів для алгоритмів машинного навчання та забезпечення простору для розширення програми.

Наукове підґрунтя даної роботи полягає у:

1) Архітектура сучасного корпоративного додатку на Node.js, яка вирішує проблеми підтримки та оновлення хмарних додатків, розроблених за допомогою Node.js. Цей макет дизайну підтримує отриману програму, щоб вона складалася з чітко визначених незалежних компонентів, які покращують

ремонтопридатність. Крім того, нові можливості можуть бути додані до програми без серйозних змін в базовій архітектурі.

2) Інструкція з впровадження, яка містить чіткі, але прості інструкції для розробки прототипу Node.js загального призначення. На базі архітектури вказується, як: створювати прототипи за допомогою обмеженого набору API; розділяти модулі та організовувати функції; дозволяти клієнту спілкуватися з сервером; використовувати базу даних хмарних додатків; обробляти завантаженні файли.

3) Сформована архітектура програмного забезпечення та рекомендації з впровадження можуть бути основою для розробки різноманітних хмарних додатків.

3.2 SOA - Сервіс-орієнтована архітектура

В випадку використання зовнішнього сервісу як джерела данних необхідно розглянути загальну сервісно орієнтовану архітектуру.

SOA - Сервіс-орієнтована архітектура – що представляє собою композицію модулів, як підхід до розробки програмного забезпечення, що заснований розподілених, не звязаних жорстко та замінних компонентах, що оснащенні стандартизованими інтерфейсами для взаємодії за стандартизованими протоколами.

Основні принципи SOA:

- Стандартизований метод комунікації - сервіси мусять бути підпорядкованими стандарту комунікації, що прийнятий колективно.

- Слабкий зв'язок – сервіси мусять бути незалежними однин від одного, і у разі припинення роботи одного, тобто виходу з ладу сервісу, інші сервіси мусять працювати в штатному режимі.

- Повторне використання сервісів – розприділення логіки по сервісах для наявності можливості їх багаторазового використання в межах сервісної архітектури.

- Абстрактність сервісу - крім опису сигнатури концевих точок сервісу за допомогою сервіс-контракту, він приховує внутрішню реалізацію бізнес-логіки від зовнішнього світу.

- Автономність – логіка інкапсульована в сервісі підконтрольна тільки йому.

Переважна більшість технологій та протоколів взаємодії сервісів, побудовані відповідно до принципів SOA, незалежать від мови написання сервісу, що забезпечує, надання користувачам можливості комбінувати сервіси різних типів в розподілених системах, та будувати єдиний шаблон опису сервісів в якості універсальної сполучної ланки.

Структура SOA відображена на рисунку 3.2.

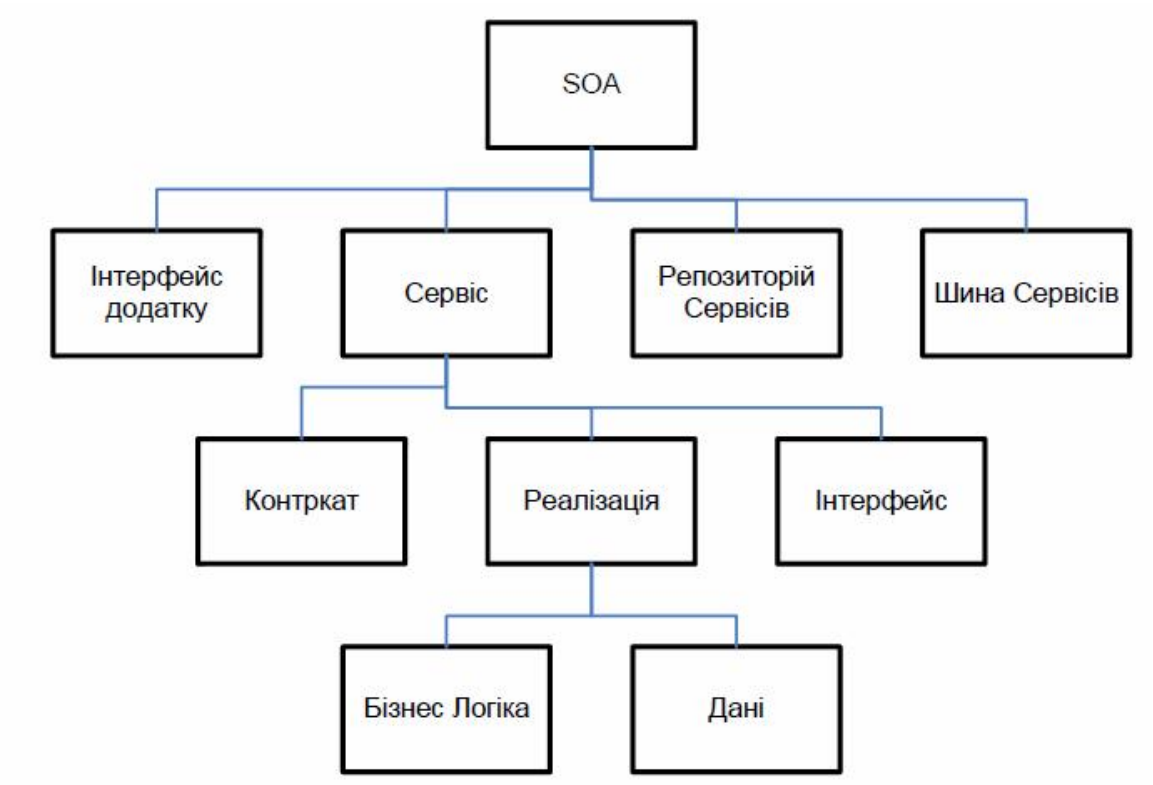

Рис. 3.2. Структура SOA

Веб-сервіси є базою для створення додатків, функціональність яких можне бути використане за допомогою стандартних протоколів Інтернет.

Концепція веб-сервісів може бути побудована за допомогою ряду технологій, стандартизація яких вказана в World Wide Web Consortium (W3C).

Веб-сервіси – один з варіантів побудови компонентної архітектури. XML - фундамент для створення переважної більшості технологій, пов'язаних з веб-сервісами.

Для створення концепції віддаленої взаємодії додатків і відповідних користувачів з веб-сервісами використовується Simple Object Access Protocol (SOAP) [4]. SOAP забезпечує комунікацію в рамках розподілених систем, абстрагуючись від об'єктної моделі, операційної системи або мови програмування. Дані передаються за допомогою визначених форматів між сервісної взаємодії, одним з яких є XML документ. Для SOAP сервісу він містить особливий формат(спосіб задання) згідно якому має бути заданий документ. Існує ряд альтернатив рішенню для взаємодії з веб-сервісами (CORBA, WCF та ін.).

Відповідно до визначення W3C, веб-сервіси це додатки, що доступні по відповідним протоколам, які є стандартизованими для мережі Інтернет. Не існує вимог, стосовно використання веб-сервісами якогось конкретного транспортного протоколу. Специфікація SOAP регламентує, яким чином вибудовується зв'язок повідомленнями SOAP і відповідний протокол передачі.

Частіше всього передача SOAP повідомлень реалізується по протоколу HTTP. Також має місце використання наступних транспортних протоколів, як SMTP, FTP, TCP для вирішення даної задачі.

Відповідно до оприділення W3C, "WSDL - формат XML для задання однозначного опису мережних сервісів в якості набору кінцевих операцій, робота яких здійснюється за допомогою повідомлень, що містять документно-орієнтовану або процедурно-орієнтовану інформацію". Документ WSDL містить повний опис інтерфейсу веб-сервісу із зовнішнім світом. WSDL файл це документ у форматі XML, що найменування методів, що будуть надані веб-сервісом, відповідні параметри даних методів, їх назви та типи, і адресу Listener `а сервісу. В своїй сутності, він є чітко визначеним за загальними принципами

взаємодії прошарком, що надає можливість використання сервісів, створених на базі різних платформ.

WSDL містить все необхідне для опису веб-сервісів, включаючи:

- URL адресу сервісу
- Механізми міжсервісної взаємодії, котрі розуміє веб-сервіс
- Функції, тобто набір методів, які можуть бути виконані веб-сервісом
- Структуру повідомлень веб-сервісу

Приклад WSDL файлу:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="ServiceName"
targetNamespace="http://ServiceUrlSample/ServiceName/"
xmlns:tns="http://ServiceUrlSample/ServiceName/"
38
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlnsгwsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns="http://schemas.xmlsoap.org/wsdl/">
<message name="SampleGetMethod">
<part name="Result" type="xsd:string" />
</message>
<message name="SampleGetRequest">
<part name="prefix" type="xsd:string" />
</message>
<portType name="GuidPortType">
<operation name="getGuid" parameterOrder="prefix">
<input message="tns:SampleGetRequest"/>
<output message="tns:SampleGetMethod" />
</operation>
</portType>
<binding name="GuidEinding" type="tns:GuidPortType">
<soap:binding                                          style="rpc"
transport="http://schemas.xmlsoap.org/soap/http" />
```

```
<operation name="getGuid">
<soap:operation soapAction="urn:ServiceName#SampleGetRequest"/>
<input>
<soap:body use="encoded" namespace="urn:ServiceName"
encodingstyle="http://schemas.xmlsoap.org/soap/encoding/" />
</input>
<output>
<soap:body use="encoded" namespace="urn:ServiceName"
encodingstyle="http://schemas.xmlsoap.org/soap/encoding/" />
</output>
</operation>
</binding>
<service name="GuidService"
<port name="GuidPort" binding="tns:GetBinding">
<soap:address location="http://localhost/php/ServiceName.php"/>
</port>
</service>
</definitions>
```

Більшість платформ самостійно здатні формувати WSDL описи сервісів, забезпечуючи розробнику відповідну можливість роботи з сервісом як із звичайним класом.

Технологія Universal Description, Discovery and Integration (UDDI) забезпечує підтримку ведення реєстру веб-сервісів. В наслідок підключення до цього реєстру, споживачеві надається можливість знаходження веб-сервісів, які є найкращим рішенням відповідно до його потреб. Технологія UDDI забезпечує людигу чи програмний клієнт можливостями публікації та пошуку необхідного сервісу. Публікація і пошук в реєстрі надається програмі- клієнтові як відповідний набір веб-сервісів реєстру UDDI.

Веб-сервіси розцінюються, як програмне забезпечення проміжного шару. Використання веб-сервісів доступне як клієнтським програм, які безпосередньо взаємодіють з користувачем, так і іншим додаткам (у тому числі й іншим веб-сервісам).

Розробниками концепції веб-сервісів запропоновано наступні сценарії застосування веб-сервісів:

1. Веб-сервіси в якості реалізації бізнес-логіки додатка. Тобто, бізнес-логіка нового додатку, буде поміщена у веб сервіс, що в свою чергу розташовано в сервісі застосувань (рис. 3.3).

Рис. 3.3. Сценарій застосування сервісу як реалізації логіки додатку

2. Веб-сервіси як засіб інтеграції. Тобто, застосування веб-сервісу як способу відаленого доступу для клієнтів до внутрішньої інформаційної системи компанії, або в якості взаємодії компонента (наприклад, EJB, COM-компонента) з різними віддаленими клієнтами (див. рисунок 3.4).

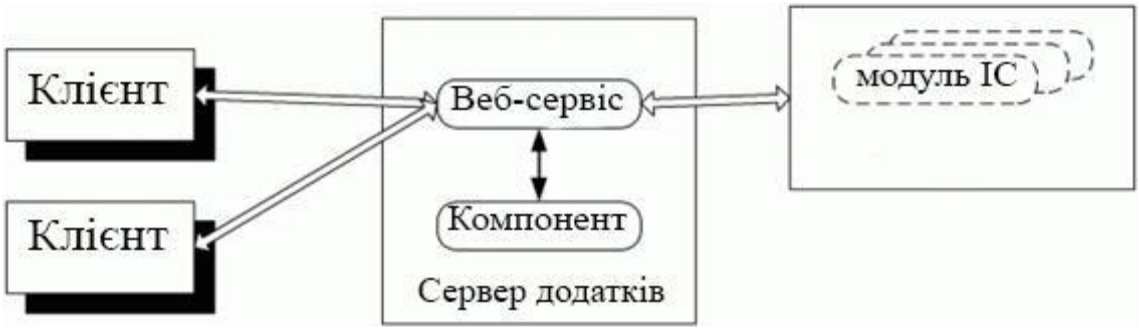

Рис. 3.4. Сценарій застосування сервісу як засобу інтеграції

3. Створення альтенативного сервісу. В даному випадку, при створені нового веб-сервісу буде використано опис інтерфейсу веб-сервісу, що вже існує. Відповідно, сервіс містить велику кількість потенційних клієнтів з

моменту початку роботи, а комунікація з ним стороніх сервісів та клієнтів не вимагає істотних змін.

Концепція веб-сервісів містить в собі можливість ведення реєстру веб-сервісів. З такого реєстру можна отримати опис інтерфейсу. Після впровадження і відповідно створення нового сервісу, має сенс помістити його в реєстр. Тоді при пошуку сервісів клієнти, що реалізують відповідний інтерфейс, отримають посилання і на новий веб-сервіс.

4. Застосування веб-сервісу, як складової частини при створенні програми.

Веб-сервіси можуть бути застосовані додатоком як зовнішні компоненти, що надають чітко визначену та вузько направлену функціональність, відповідно до потреб клієнта, матеріальних та часових обмежень і рівня складності завдання, що забезпечує системі велику гнучкість (рисунок 3.5).

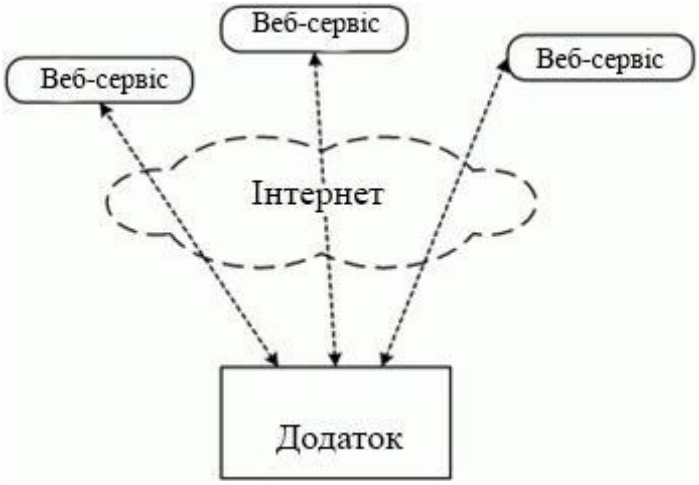

Рис. 3.5. Застосування сервісу як зовняшнього структурного блоку програми

Окрім цього, має місце і інші варіанти застосування веб-сервісів. Прикладом цього є дослідження з використання веб-сервісів в якості елементів для побудови розподілених обчислень та відповідного класу інформаційних систем, до яких належать однорангові, так і зі системи з складною структурою.

Для застосування веб-сервісу клієнту необхідно володіти такою інформацією:

- адреса розташування веб-сервіс;

- ім'я веб-сервісу;

- сигнатура методу, що використовуватиметься.

У найпростішому випадку дана інформація може бути задана при розробці клієнта, що проілюстровано рисунком 3.6.

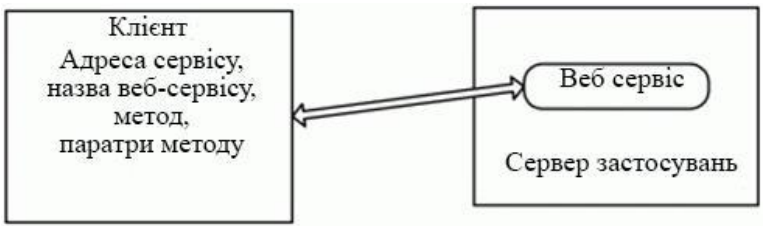

Рис. 3.6. Інформація для застосування веб-сервісу, що є задана при розробці клієнта

Документ WSDL містить повний опис інтерфейсу веб-сервісу із зовнішнім світом, він надає клієнтові засіб для отримання необхідної інформації для подальшого застосування веб-сервісу (рисунок 3.7).

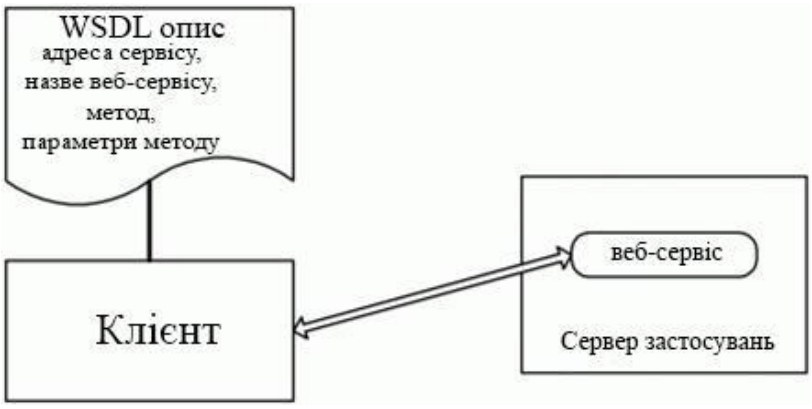

Рис. 3.7. WSDL опис інтерфейсу сервісу документом

Виділяють два ключові підходи, для розв’язання проблеми координації роботи сервісів в розподілених системах:

- Service orchestration (Приклад - BPEL) - метод композиції сервісів, з використанням сервісу-окрестартора, в один багатофункціональний бізнес-сервіс.

- Web Service Choreography – метод, що регламентує інтерфейс взаємодії веб-сервісів між собою. Тобто кооперацію сервісів для виконання одного глобального завдання, завдяки виконанню окремих його частини. Роль сервісу, визначає модель обміну повідомленнями з партнерами. Даний підхід є ефективним для простих завдань, але відповідно до зростання складності поставленого завдання та росту кількості задіяних сервісів стає громіздкою і неефективною.

3.3 Програмні 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. 3.8). Тобто реалізується зв'язок «один-до-одного» – один з конкретних сервісів викликається у часі тільки одним із споживачів, але зв'язок є двонаправлений.

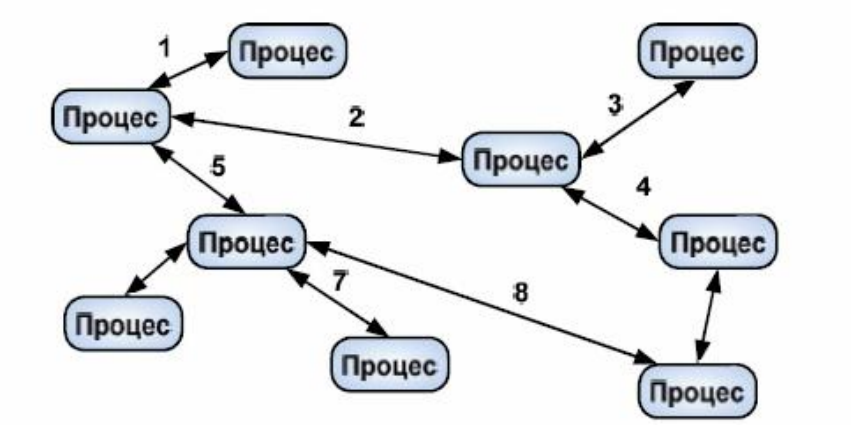

Рис. 3.8. Архітектура SOA-додатку, керована запитами

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. 3.9).

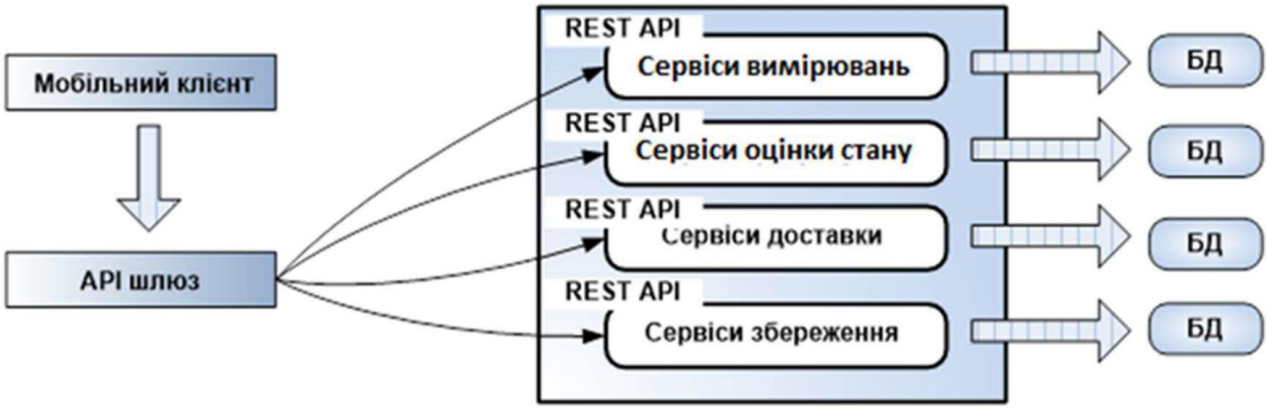

Рис. 3.9. Спілкування мікросервісів через API-шлюз

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.3.10). Наприклад, якщо потрібно викликати у певному замовленні три сервіси, оркестратор звернеться до кожному з них, чекаючи відповіді, перш ніж викликати наступний.

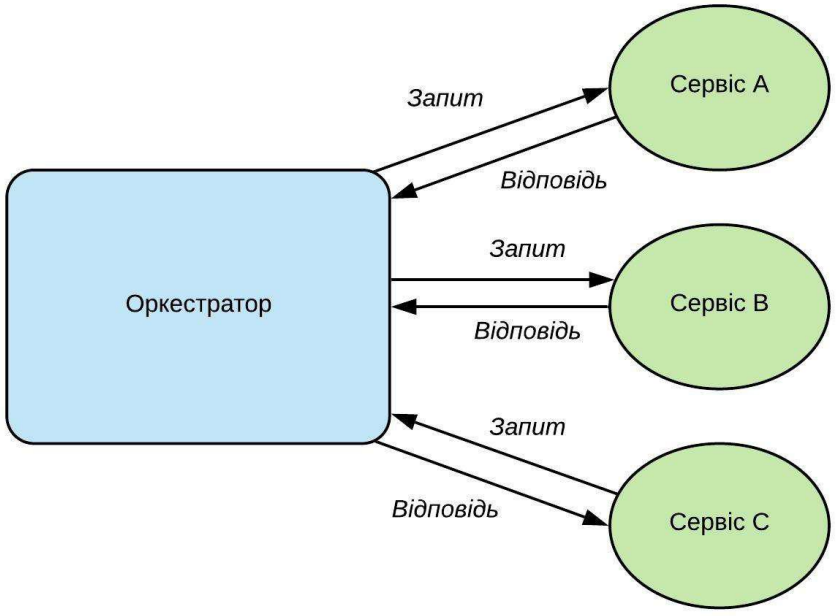

Рис. 3.10. Взаємодія сервісів за шаблоном типу «запит / відповідь»

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 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.

3.4 Архітектура пропонованого рішення

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.

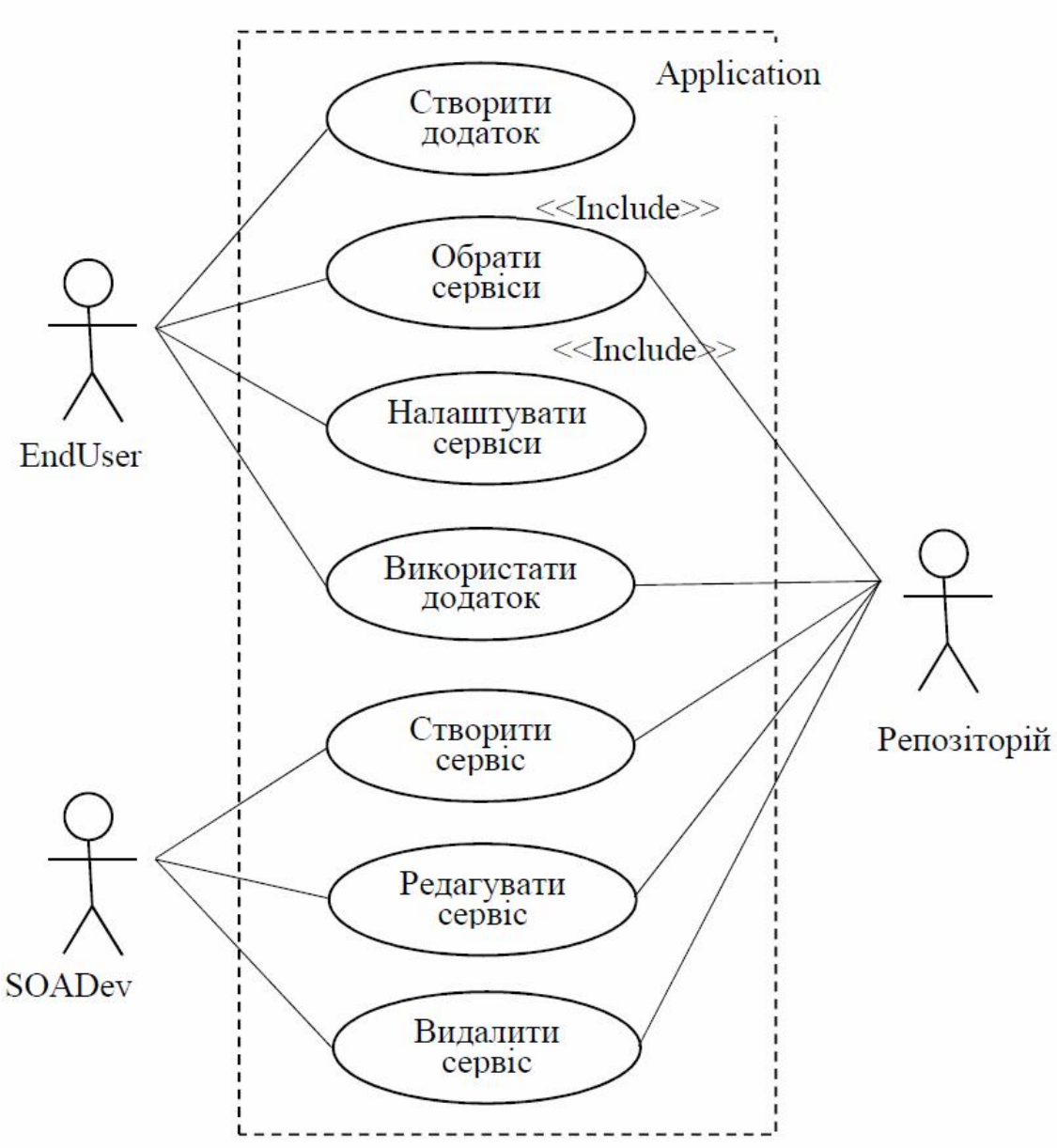

Рис. 3.11. Діаграма варіантів використання

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. 3.11.

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

Розглянемо архітектуру та функції системи більш детально. Для цього проаналізуємо узагальнену архітектуру системи, яка наведена на рис. 3.12.

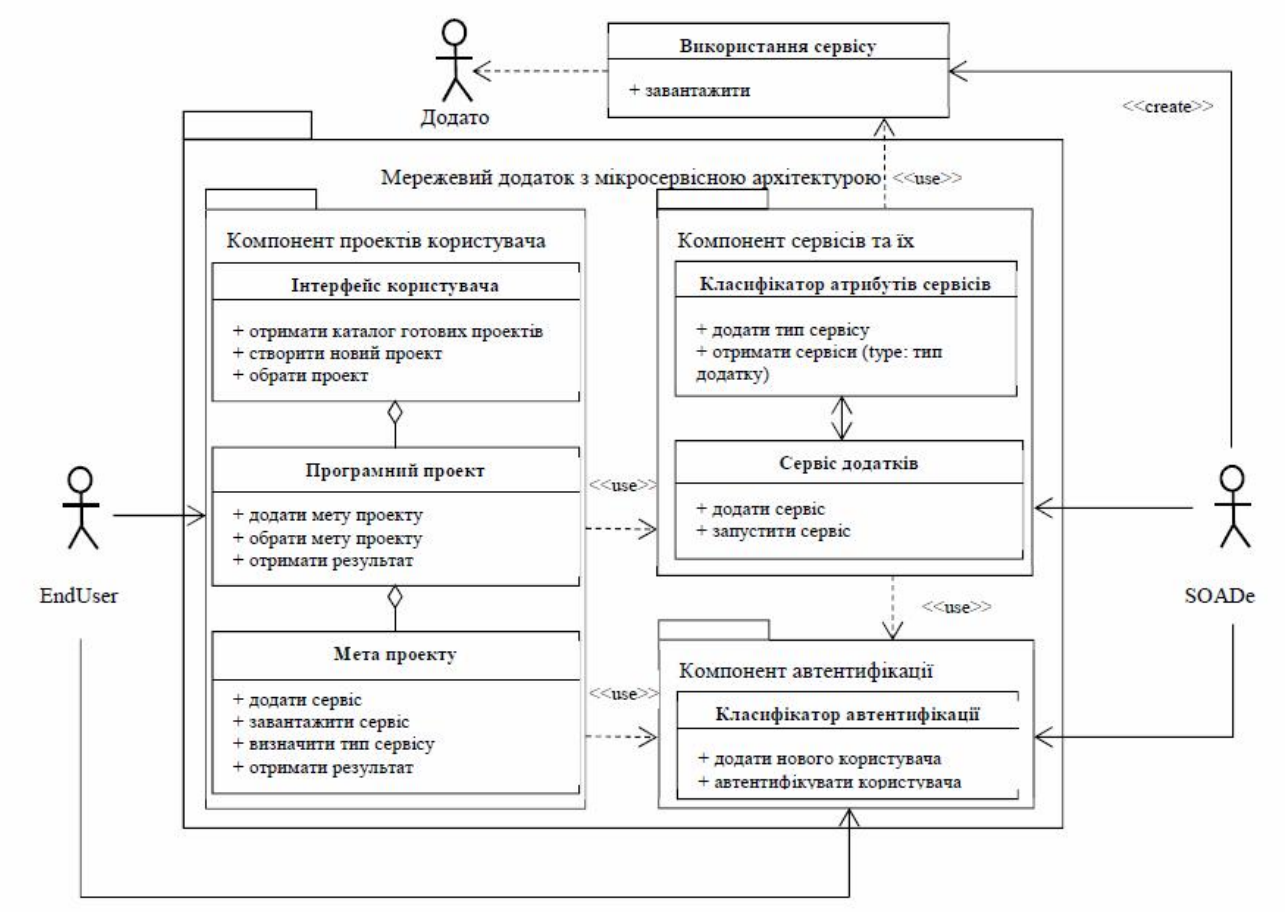

Рис. 3.12. Узагальнена архітектура системи

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.

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 3.13 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).

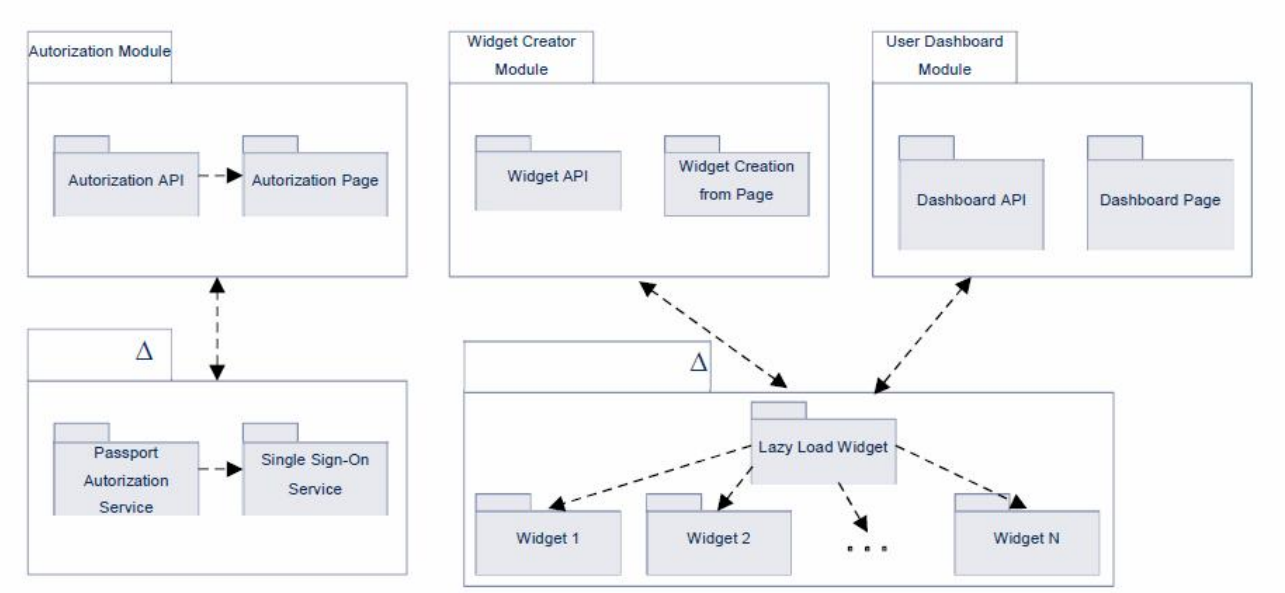

Рис. 3.13. UML діаграма архітектури системи

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.

Висновки до розділу

Підводячи підсумки слід зазначити, що програмну реалізацію сервіс-інтегратора з використанням мікросервісних контейнерів було виконано у відповідності до мети дослідження. Виконано реалізація методології побудови архітектури великих та розподілених корпоративних web-додатків, наведено архітектуру пропонованого рішення.

ВИСНОВКИ

В магістерській роботі виконано дослідження моделей та методів побудови великих та розподілених WEB-додатків. 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 сервісів;

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

– здійснено алгоритмічну реалізацію пропонованої ситстеми;

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

В магістерській роботі розкрито питання побудови архітектури сучасних корпоративних додатків на Node.js. Запропоновано архітектуру програмного забезпечення та рекомендації щодо впровадження для розробки хмарних додатків Node.js.

Розроблена архітектура сучасних корпоративних додатків на Node.js використовує підходи «зверху-вниз», «розділяй і володарюй», щоб ефективно структурувати дизайн додатків для кращої ремонтопридатності та розширюваності з часом. З іншого боку, описані принципи та особливості впровадження, які розкривають процедури встановлення та використання. Також представлено практичні сценарії того, як розробники можуть використовувати запропоновану архітектуру сучасних корпоративних додатків та рекомендації щодо впровадження для різних типів додатків.

Перспективи подальших досліджень базуються на розширенні сфери дій пропонованої архітектури до різних типів додатків, щоб показати ефективність розробки.

СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА

1. Козак Є. Б. Програмні методи організації транспортних потоків у рамках концепції Internet of Vehicles. Науковий журнал «Комп'ютерно-інтегрована технології: освіта, наука, виробництво», 2021. № 44. С. 94-100. https://doi.org/10.36910/6775-2524-0560-2021-44-15.
2. About Node.js, and why you should add Node.js to your skill set? – Режим доступу. – http://blog.training.com/2016/09/about-nodejs-and-why-you-should-add.html
3. Todd Veldhuizen «Techniques for scientific C++» – Режим доступу. – https://web.archive.org/web/20061013134847/http://osl.iu.edu/~tveldhui/papers/techniques/
4. Порівняння фреймворків express та react на платформі Node.js / А. В. Єрукаєв, О. С. Копча, Д. С. Лукенів // Тези доповідей восьмої міжнародної науково-практичної конференції «Управління розвитком технологій». Тема: Інформаційні технології розвитку змісту освіти. // Відповідальна за випуск завідувач кафедри ІТ С.В. Цюцюра, – К. : КНУБА, 2021. С. 25-26.
5. Розробка мобільного застосування з автоматичним налаштуванням функцій клієнтської частини / Л. Є. Кузьмиченко, О. О. Арсірій // Матеріали Десятої Міжнародної наукової конференції студентів та молодих вчених «Сучасні інформаційні технології – 2020» «Modern Information Technology - 2020» (14-15 травня 2020 р., м.Одеса) / МОН України; Одес. Нац. політех. ун-т ; Ін-т комп'ют. систем. – Одеса : Наука і техніка, 2020. С. 154-155.
6. Використання технології віртуальних інтерфейсів для підтримки онлайн гри / В. О. Кривенька, О. С. Городецька, Л. А. Савицька // Науково-технічна конференція факультету інформаційних технологій

та комп`ютерної інженерії. ВНТУ – 2021. – https://conferences.vntu.edu.ua.

7. Maxim Bartkov (2021). GRAAL AS A MULTILINGUAL PLATFORM. New York. TK Meganom LLC. Innovative Solutions in Modern Science. 3(47). doi: 10.26886/2414-634X.3(47)2021.9
8. Управління якістю програмних веб-систем засобами розробки / Шинкарук, О.М., Яшина, О.М., Онишко, О.Г. // Вісник Хмельницького національного університету, №6, 2020 (291). С. 39-44.
9. Schreck, Hanna-Reetta. (2021). Modern Corporeality. 10.4324/9780367823672-10.
10. Panetta, Daniele & Steppert, Michael & Ross, Tobias & Szidat, Sönke & Cutler, Cathy & Del Guerra, A. & Düllmann, Christoph & Eberhardt, Klaus & Edelstein, Norman & Gaeggeler, Heinz & Langrock, Gert & Moenius, Thomas & Morss, Lester & Rösch, Frank & Ruehm, Werner & Trautmann, Norbert & Walther, Clemens & Wendt, Klaus & Zeh, Peter. (2016). Modern Applications. 10.1515/9783110221862.
11. Gonzalez-Morón, Dolores & Kauffman, Marcelo. (2020). Modern applications of neurogenetics. 10.1016/B978-0-12-819178-1.00013-7.
12. Zverovich, Vadim. (2021). Modern Applications of Graph Theory. 10.1093/oso/9780198856740.001.0001.
13. Beaumont, Perry. (2019). Theoretical foundations and modern applications. 10.4324/9780429053047-4.
14. Sutanto, Sutanto & Gunawan, Waliadi & Faeshal, Faeshal. (2021). ARSITEKTUR CONTAINER DOCKER PADA APLIKASI EXPERT ASSIST DENGAN TEKNOLOGI NODE.JS, EXPRESS FRAMEWORK & CLOUD DATABASE NoSQL MONGODB ATLAS. Jurnal Sistem Informasi dan Informatika (Simika). 4. 73-89. 10.47080/simika.v4i1.1189.
15. Bhardwaj, Harsh. (2021). Challenges with Implementation of Node.Js. International Journal for Research in Applied Science and Engineering Technology. 9. 1086-1090. 10.22214/ijraset.2021.37464.

16. Shcherbakov, E. & Shcherbakova, M.. (2021). Event dispatching algorithms in Node.js. Scientific news of Dahl university. 10.33216/2222-3428-2021-20-14.
17. Mardan, Azat. (2018). Intro to Node.js: Learn Backbone.js, Node.js, and MongoDB. 10.1007/978-1-4842-3718-2_6.
18. Ахмед А., Ахмад С., Ехсан Н., Мірза Е., Сарвар С. З. Розробка програмного забезпечення Agile: вплив на продуктивність і якість. Управління інноваціями та технологіями (ICMIT), 2010 Міжнародна конференція IEEE з питань; 2010. 287–291.
19. Амсел Н., Ібрагім З., Малік А., Томліnсон Б. На шляху до стійкої розробки програмного забезпечення (Трек NIER). Матеріали 33-ї міжнародної конференції з інженерії програмного забезпечення;2011. 976–979.
20. Баджпай В., Горті Р. П. Про нефункціональні вимоги: опитування. Електротехніка, електроніка та інформатика (SCEECS), 2012 IEEE Students' Conference on; 2012. 1–4.
21. Бек К., Бідл М. та ін. Маніфест гнучкої розробки ПЗ. [веб-сайт]. 13 лютого 2001. [цитовано 26 квітня 2016]. Доступно з: http://agilemanifesto.org.
22. Белл Е., Брайман А. Етика дослідження менеджменту: дослідницький аналіз вмісту. Британський журнал менеджменту. 2007 рік; 18 (1): 63–77.
23. Бетц С., Капорале Т. Стійке проектування програмного забезпечення. Великі дані та хмарні обчислення (BdCloud), Четверта міжнародна конференція IEEE 2014; 2014. 612–619.
24. Бем Б., Тернер Р. Проблеми управління впровадженням гнучких процесів у традиційному організації розвитку. Програмне забезпечення IEEE. 2005 вер.; 22 (5): 30–39.
25. 1. Sergii Telenyk, Grzegosz Nowakowski, Kostyantyn Yefremov, Volodymyr Khmelyuk. Logics based application integration for

interdisciplinary scientific investigations // 2017 9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), 21-23 Sept. 2017, Bucharest, Romania. DOI: 10.1109/IDAACS.2017.8095241

26. Шаховська Н.Б. Методи опрацювання консолідованих даних за допомогою просторів даних / Н.Б.Шаховська // Проблеми програмування. – 2011. – № 4. – С. 72-84.
27. Теленик С.Ф. Семантична інтеграція різнорідних інформаційних ресурсів / С.Ф.Теленик, О.А.Амонс, К.В.Ефремов, С.В.Жук // Вісник НТУУ «КПІ», Інформатика, управління та обчислювальна техніка». – №58. –2013. – С.29 – 45.
28. Теленик С.Ф. Логічний підхід до інтеграції програмних застосувань підтримки міждисциплінарних наукових досліджень // С.Ф.Теленик, О.А.Амонс, К.В.Єфремов, В.Т.Лиско // Наукові вісті НТУУ «КПІ». – №5 (91). –2013. – С.53–72.
29. Теленик С.Ф. Управління високопродуктивними IT-інфраструктурами / Ю.В.Бойко, М.М.Глибовець, С.В.Єршов, С.Л.Кривий, С.Д.Погорілий, О.І.Ролік, С.Ф.Теленик, М.В.Ясочка // Вісник НТУУ «КПІ», Інформатика, управління та обчислювальна техніка». – №61. –2014 . – С.120 – 141.
30. Cacti - The Complete RRDTool-based Graphing Solution [Електронний ресурс] – Режим доступу до ресурсу: http://www.cacti.net.
31. Cacti (software) [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Cacti_(software).
32. Let’s Encrypt [Електронний ресурс] – Режим доступу до ресурсу: https://letsencrypt.org/.
33. Oracle. Siebel Business Process Framework: Workflow Guide Version [Електронний ресурс] – Режим доступу до ресурсу: http://docs.oracle.com/cd/B40099_02/books/PDF/BPFWorkflow.pdf.

34. HP Service Activator (HPSA) solution development training [Електронний ресурс] – Режим доступу до ресурсу: http://www8.hp.com/h20195/v2/GetPDF.aspx%2F4AA4-9576ENW.pdf.
35. NetCracker. The Solution Products [Електронний ресурс] – Режим доступу до ресурсу: http://www.nec.com/en/global/techrep/journal/g10/n02/pdf/100221.pdf.
36. Information technology – Object Management Group Business Process Model and Notation / ISO/IEC 19510, 2013.
37. K. Heather. Navigating the SOA Open Standards Landscape Around / The Open Group, 2009. – 27p.
38. Web Services Architecture – W3C Working Group Note [Електронний ресурс]– Режим доступу до ресурсу: http://www.w3.org/TR/ws-arch/.
39. SOA Reference Architecture. – The Open Group / ISBN: 1-937218-01-0, 2011. – 192 p.
40. Reference Architecture Foundation for Service Oriented Architecture Version.— The Organization for the Advancement of Structured Information Standards (OASIS), 2012 [Електронний ресурс] – Режим доступу до ресурсу: http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/soa-ra.html.
41. Service-Oriented Architecture Ontology. – The Open Group / ISBN: 1-931624-88-7, 2010.
42. Calero C., Bertoa M. F. 25010+ S: модель якості програмного забезпечення зі стійкими характеристиками. Стійкість як елемент якості ПЗ. Green в розробці програмного забезпечення Green by Розробка програмного забезпечення (GIBSE 2013) спільно з AOSD. Березень 2013 р.;
43. Calero C., Bertoa M. F., Moraga M. Á. Систематичний огляд літератури щодо заходів стійкості програмного забезпечення. Матеріали 2-го Міжнародного семінару з екології та сталого розвитку програмне забезпечення; 2013. 46–53.

44. Гао Ю. Дослідження правила еволюції моделі процесу розробки програмного забезпечення. Управління інформацією та інженерія (ICIME), 2010 е 2-га міжнародна конференція IEEE на; 2010. 466–470.
45. Гібберт М., Руйгрок В., Вікі Б. Що вважається ретельним практичним дослідженням? Журнал стратегічного менеджменту. 2008 рік; 29 (13): 1465–1474.