Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя
(повне найменування вищого навчального закладу)
Факультет комп'ютерно-інформаційних систем і програмної інженерії
(назва факультету)
Кафедра комп'ютерних систем та мереж
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістра
(освітній ступінь))

на тему: **Методи і засоби імплементації компонентів розумних комп'ютерних систем з використанням прикладних програмних інтерфейсів**

Виконав: студент (ка) 6 курсу, групи СІм-61
спеціальності 123 «Комп'ютерна інженерія»
(шифр і назва спеціальності)

	(підпис)	Денисов Д.В. (прізвище та ініціали)
Керівник	(підпис)	Яцишин В.В. (прізвище та ініціали)
Нормоконтроль	(підпис)	Луцик Н.С. (прізвище та ініціали)
Завідувач кафедри	(підпис)	Осухівська Г.М. (прізвище та ініціали)
Рецензент	(підпис)	Марценко С.В. (прізвище та ініціали)

Тернопіль
2022

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя
(повне найменування вищого навчального закладу)
Факультет комп'ютерно-інформаційних систем і програмної інженерії
Кафедра комп'ютерних систем та мереж

ЗАТВЕРДЖУЮ
Завідувач кафедри *Осухівська Г.М.*
«_____» 2022 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня *магістр*
(назва освітнього ступеня)

за спеціальністю *123 «Комп'ютерна інженерія»*
(шифр і назва спеціальності)

студенту *Денисову Денису Васильовичу*
(прізвище, ім'я, по-батькові)

1. Тема проекту (роботи) *Методи і засоби імплементації компонентів розумних комп'ютерних систем з використанням прикладних програмних інтерфейсів*
Керівник проекту (роботи) *Яцишин Василь Володимирович, к.т.н., доц.*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
Затверджені наказом ректора від «*06*» *грудня* 2022 року №*4/7-986*
2. Термін подання студентом завершеної роботи
3. Вихідні дані до роботи *Особливості реалізації розумних комп'ютерних систем, типи і принципи побудови прикладних програмних інтерфейсів, опис технології PaaS*
4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ. 1. Прикладні програмні інтерфейси в комп'ютерних системах
2. Імплементація методу побудови прикладних програмних інтерфейсів в комп'ютерних системах
3. Реалізація та експериментальні дослідження API комп'ютерних систем
4. Охорона праці та безпека в надзвичайних ситуаціях. Висновки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1. Актуальність і мета дослідження. 2. Задачі дослідження, , об'єкт і предмет, наукова новизна і практична цінність дослідження. 3. Базові архітектури і протоколи взаємодії при проектуванні API комп'ютерних систем 4. API як сервіси. 5Метод проектування прикладних програмних інтерфейсів. 6. Формалізація програмних інтерфейсів. 7. Типи серверів при використанні API. 8. Висновки

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Охорона праці та	*Осухівська Г.М.*		
безпека в надзвичайних			
ситуаціях			

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	*Прикладні програмні інтерфейси в комп'ютерних*		*виконано*
	системах		
2.	*Імплементація методу побудови прикладних програмних*		*виконано*
	інтерфейсів в комп'ютерних системах		
3.	*Реалізація та експериментальні дослідження API*		*виконано*
	комп'ютерних систем		
4.	*Охорона праці та безпека в надзвичайних ситуаціях*		*виконано*
5.	*Оформлення пояснювальної записки*		*виконано*
6.	*Оформлення графічного матеріалу*		*виконано*
7.	*Попередній захист кваліфікаційної роботи магістра*		*виконано*
8.	*Захист кваліфікаційної роботи магістра*		

Студент _____ *Денисов Д.В.*
(підпис) (прізвище та ініціали)

Керівник проекту (роботи) _____ *Яцишин В.В.*
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Методи і засоби імплементації компонентів розумних комп’ютерних систем з використанням прикладних програмних інтерфейсів // Кваліфікаційна робота// Денисов Денис Васильович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп’ютерно-інформаційних систем та програмної інженерії, група СІм-61 // Тернопіль, 2022 // с. – 90 , рис. – 38 , табл. –10 , аркушів А1 –8 , додат. – 1, бібліогр. – 23 .

КЛЮЧОВІ СЛОВА: МЕТОД, ЗАСІБ, ПРИКЛАДНИЙ ПРОГРАМНИЙ ІНТЕРФЕЙС, КОМПОНЕНТ, КОМП’ЮТЕРНА СИСТЕМА.

У кваліфікаційній роботі магістра проведено аналіз існуючих технологій проектування комп’ютерних систем, розроблено метод побудови API на основі принципів компонентного підходу, що дає змогу врахувати вимоги архітектури SOA та представити в узагальненому, уніфікованому вигляді конструкцію прикладних програмних інтерфейсів.

Запропоновано математичне представлення структури програмного компоненту комп’ютерної системи, що використовує парадигму об’єктно-орієнтованого підходу та описується за допомогою елементів теорії множин, що дало змогу використовувати їх у процесі імплементації API за допомогою мікросервісів. Обґрунтовано метод перевірки взаємозв’язків між функціональними сервісами комп’ютерних систем у хмарному середовищі із застосуванням методів тестування API.

Інструментами системи Onlizer реалізовано прикладний програмний інтерфейс пошуку і зберігання даних, що дає змогу знизити поріг входу при розробці комп’ютерних систем за рахунок імплементованої в платформі drag&drop процедури.

ABSTRACT

Methods and means for implementing smart computer systems' components using application programming interfaces /Master thesis / Denysov Denys Vasylovych / Ternopil Ivan Pul'uj National Technical University, Faculty of Computer Information Systems and software engineering, group CIm -61 // Ternopil, 2022// p. - 90, fig. – 38, table. – 10, Sheets A1 – 8, Add – 1, Ref. – 23.

KEYWORDS: METHOD, TOOL, APPLICATION PROGRAMMING INTERFACE, COMPONENT, COMPUTER SYSTEM,.

In the master's qualification work, an analysis of existing computer system design technologies was carried out and it was established that the most popular way of implementing complex systems is the integration and aggregation of components using application software interfaces. A method of building an API based on the principles of the component approach has been developed, which makes it possible to take into account the requirements of the SOA architecture and present the design of application software interfaces in a generalized, unified form.

A mathematical representation of the structure of a software component of a computer system is proposed, which uses the paradigm of an object-oriented approach and is described using the elements of set theory, which made it possible to use them in the process of implementing APIs using microservices.

The method of checking the interrelationships between functional services of computer systems in the cloud environment with the application of ARI testing methods is substantiated.

The tools of the Onlizer system implement an applied software interface for searching and storing data, which makes it possible to lower the entry threshold when developing computer systems due to the drag&drop procedure implemented in the platform.

ЗМІСТ

ПЕРЕЛІК ОСНОВНИХ УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ І СКОРОЧЕНЬ

БД	База Даних
ПС	Програмні Системи
ACID	Atomicity Consistency Isolation Durability
CASE	Computer Aided Software Engineering
DDL	Data Definition Language
DFD	Data Flow Diagram
ER	Entity Relations
MSF	Microsoft Solution Framework
RDD	Requirement Driven Design
UML	Unified Modeling Language
XML	Extended Markup Language

ВСТУП

Актуальність теми. Технологічний розвиток в IT сфері сприяє вдосконаленню існуючих комп'ютерних систем шляхом впровадження інноваційних рішень, орієнтованих на підвищення ефективності та функціональності як апаратного, так і програмного забезпечення. Зараз спостерігається тренд щодо інтенсифікації впровадження «розумних» пристроїв як для побутових потреб, так і для промислових, аграрних та інших типів підприємств, що в комплексі становлять складні і водночас розумні комп'ютерні системи.

Важливими аспектами і складовими частинами при імплементації комп'ютерних систем є програмні засоби, які забезпечують керування бізнес-процесами і технічними пристроями. Беручи до уваги різноманітність та неоднорідність існуючих програмних систем, і потребу у їхній агрегації для підвищення ефективності функціонування комп'ютерних систем, постає необхідність в інтеграції готових рішень. Для цього доцільно скористатися підходом прикладних програмних інтерфейсів.

В основі такого підходу лежать набір правил, що використовується для забезпечення взаємодії однієї програмної системи з іншою. Це в перспективі дає змогу підвищити масштабованість і гнучкість більш складних комп'ютерних систем та сприяє розширенню архітектурного рішення без порушення його цілісності. Враховуючи аспект міграції багатьох програмних комплексів у хмарне середовище, то доцільно проектувати розумні комп'ютерні системи, опираючись на особливості хмарних сервісів. Це дозволить підвищити надійність функціонування як програмного забезпечення комп'ютерної системи, так і системи в цілому, а також забезпечити необхідний рівень продуктивності.

Архітектура, що є базовою для організації веб-орієнтованих сервісів представляє собою клієнт-сервер на найвищому рівні, а на прикладному – це багаторівнева структура або мікросервіси. Принципи побудови багатошарової архітектури досліджувались багатьма вченими, зокрема, Р. Джонсона, А. Фрімена, Р. Хелма та іншими. Однак незважаючи на розвинутість інструментів реалізації та

формалізацію основних компонентів при використанні таких архітектур, при переході до більш комплексних систем можуть виникати проблеми з гнучкістю і масштабованістю рішень. Тому для розв’язку такої задачі, пропонується провести дослідження щодо можливості застосування компонентного підходу разом з підходом до реалізації сервіс-орієнтованих систем.

Актуальність такого дослідження підтверджується відсутністю стандартизованого підходу до проектування API, що вимагає додаткових досліджень методів і засобів імплементації компонентів розумних комп’ютерних систем з використанням прикладних програмних інтерфейсів.

Мета роботи полягає у дослідженні методів і засобів імплементації компонентів розумних комп’ютерних систем з використанням прикладних програмних інтерфейсів.

Основні **задачі дослідження**:

– провести аналіз наукових публікацій для оцінювання існуючих методів і засобів імплементації компонентів розумних комп’ютерних систем;

– проаналізувати переваги і недоліки технологій проектування прикладних програмних інтерфейсів та визначити можливі способи зменшення або повного усунення;

– розробити метод проектування прикладних програмних інтерфейсів для подальшого його використання у хмарних сервісах;

– реалізувати та експериментально дослідити API при інтеграції розумних комп’ютерних систем за допомогою платформи як сервісу.

Об’єкт дослідження – процеси імплементації прикладних програмних інтерфейсів в комп’ютерних системах.

Предмет дослідження — моделі, методи і засоби створення прикладних програмних інтерфейсів при інтеграції комп’ютерних систем.

Методи дослідження: Для вирішення поставлених задач використано наступні методи: аналіз та узагальнення – при проведенні аналізу існуючих підходів до імплементації прикладних програмних інтерфейсів; формалізації – при

обґрунтуванні компонентного підходу і розробці методу проектування програмних інтерфейсів; проектування та програмування – при використанні платформи розробки програмних інтерфейсів та апробації запропонованого методу; експеримент та вимірювання – для розробки програмних інтерфейсів при інтеграції комп'ютерних систем.

Наукова новизна отриманих результатів. Наукова новизна полягає у вирішенні науково-практичної задачі проектування та імплементації прикладних програмних інтерфейсів у «розумних» комп'ютерних системах, при цьому одержано наступні результати:

– уперше, запропоновано метод побудови API на основі принципів компонентного підходу, що дає змогу врахувати вимоги архітектури SOA та представити в узагальненому, уніфікованому вигляді конструкцію прикладних програмних інтерфейсів;

– уперше запропоновано математичне представлення структури програмного компоненту комп'ютерної системи, що використовує парадигму об'єктно-орієнтованого підходу та описується за допомогою елементів теорії множин, що дало змогу використовувати їх у процесі імплементації API за допомогою мікросервісів.

Практичне значення одержаних результатів. На основі запропонованого методу та з допомогою інструментів платформи Onlizer спроектовано інтерфейс, який дає змогу забезпечувати інтеграцію різних сервісів і дозволяє знизити поріг входу розробника комп'ютерних систем без особливих знань та особливостей програмування.

Публікації. Результати кваліфікаційної роботи апробовані на X науково-технічній конференції Тернопільського національного технічного університету імені Івана Пулюя «Інформаційні моделі, системи та технології» (8-9 грудня 2022 року) як тези конференцій.

1. Лупенко А.М., Куліков С.О., Денисов Д.В. Класифікація та особливості застосування прикладних програмних інтерфейсів при реалізації комп'ютерних систем. Матеріали X науково-технічної конференції Тернопільського

національного технічного університету імені Івана Пулюя «Інформаційні моделі, системи та технології» (8-9 грудня 2022 року). Тернопіль: ТНТУ. 2022. С. 79.

2. Яцишин В.В., Шаблій Н.Р., Денисов Д.В. Призначення і доцільність використання API Gateway у комп'ютерних системах. Матеріали X науково-технічної конференції Тернопільського національного технічного університету імені Івана Пулюя «Інформаційні моделі, системи та технології» (8-9 грудня 2022 року). Тернопіль: ТНТУ. 2022. С. 80.

Структура роботи. Кваліфікаційна робота містить розрахунково-пояснювальну записку та графічний матеріал. До складу записки входить вступу, 4 розділи, загальні висновки, список використаних джерел і додатки. Обсяг роботи: розрахунково-пояснювальна записка – 90 арк. формату А4, графічна частина – 8 аркушів формату А1.

РОЗДІЛ 1
ПРИКЛАДНІ ПРОГРАМНІ ІНТЕРФЕЙСИ В КОМП'ЮТЕРНИХ СИСТЕМАХ

1.1. Аналіз базових понять і підходів до проектування програмних інтерфейсів

Ключовим поняттям кваліфікаційної роботи магістра є програмний інтерфейс. Цей термін є надзвичайно широким і в залежності від завдань, що потрібно виконати може мати різноманітні значення. З точки зору цього дослідження програмний інтерфейс можна описати, як метод чи інтерфейс використання або імплементація якого надає можливість отримувати чи змінювати стан системи через використання різного роду application programming interface (API). В рідкісних випадках під цим визначенням буде розумітись зовнішнє представлення програмної системи.

API представляє собою сукупність правил і відповідних реалізацій, що визначають яким чином програмна система чи її компонент буде взаємодіти із зовншінми по відношенню до неї засобами.

Властивості, якими повинен володіти API:

– простота застосування – передбачає, що API не є складним у використанні та підтримці;

– висока конверсія серед розробників ПЗ – стимулює кількість клієнтів сервісу;

– високий рівень поширеності сервісу – вимірюється кількістю користувачів, які застосовують API;

– ізольованість компонентів – характеризує ступінь структурованості програмного інтерфейсу і чим він вищий, тим ефективніша ізольованість компонентів;

– зручність використання API – програмний інтерфейс є своєрідним інтерфейсом для розробників на який вони одразу звертають увагу при аналізі

продукту. У випадку, коли прикладний програмний інтерфейс не відповідає заявленим вище властивостям, то відповідно IT-експерт не рекомендуватиме його застосування для компанії.

За способом реалізації програмні інтерфейси поділяють на наступні групи: за способом реалізації:

– програмні інтерфейси веб-сервісів – найбільш поширеними серед таких видів інтерфейсів є REST, SOAP та віддаленого виклику XML та JSON. Окрім цього варто виділити інтерфейси, які використовують веб-сокети;

– програмні інтерфейси бібліотек – передбачають доступ до бібліотек або їх частин у мовах програмування Java Script, а також до класів, наприклад, у Java та C#

Класифікація прикладних програмних інтерфейсів за сферою застосування передбачає наявність наступних груп:

а) функції і процедури операційної системи:

– доступ до файлової системи;

– доступ до інтерфейсу користувача;

– прикладні програмні інтерфейси віддаленого доступу до об’єктів, для прикладу, .Net та CORBA;

б) прикладні програмні інтерфейси апаратного забезпечення:

– інтерфейси до апаратного прискорення відео (OpenCL...);

– інтерфейси жорстких дисків;

– інтерфейси PCI шини;

– …

До Web API відносяться XML-RPC і JSON-RPC, SOAP і REST.

Віддалений виклик процедур (RPC) – технологія, яка може поєднувати різні протоколи, що забезпечують виклик методу у зовнішньому додатку.

XML– протокол віддаленого виклику, який почав використовуватися після створення альтернативної мови розмітки XML і довгий час просувався й підтримувався компанією Microsoft. Після того, як Microsoft перейшов до

використання SOAP, даний протокол втратив цінність для цієї компанії. Однак варто відмітити, що він ще досі може використовуватися в інших мовах програмування, наприклад PHP.

SOAP – це протокол, який був проміжним між XML-RPC і REST. Однак він характеризувався високою складністю використання і у результаті не набув значного поширення та не був визнаний як стандарт.

Більш простий та ефективний підхід реалізації прикладних програмних інтерфейсів – REST («передача стану представлення»), який запропонований у 2000 р. Суть та ідея передачі стану представлення полягає у тому, що кожен запит до сервісу змінює стан застосунку. Потрібно відмітити, що REST не є протоколом чи стандартом, а є підходом для проектування прикладних програмних інтерфейсів.

Принципи за якими побудований і використовується REST базується на використанні клієнт-серверної архітектури, а ресурси, при цьому, представляють собою будь-які дані. Кожен ресурс повинен містити унікальний код-ідентифікатор за значенням якого можна одержати потрібні дані, а також зв’язок між ресурсами здійснюється за допомогою наявного у response значення ідентифікатора або посилання.

Варто відмітити і той факт, що REST використовує типові HTTP-методи, а сервер не організований таким чином, що не зберігає стан, тобто він не виділяє викликів та немає можливості зберігати в пам’яті усі сесії.

У випадку використання REST при організації сервісу на хмарних ресурсах чи на кластері серверів, відпадає потреба забезпечення узгодженості станів сервісів за наявними хостами. Це дозволяє значно спростити процес організації масштабування рішення. Підхід REST передбачає використання сукупності правил і функціональності протоколу HTTP і може використовувати всі його переваги, зокрема, це стосується кешування даних, масштабованості при передачі інформації, зменшення та робота з стандартними кодами помилок.

З позиції практики при проектуванні API необхідно враховувати наступні рекомендації:

– використання SSL для забезпечення процесів ідентифікації та аутентифікації;

– формування документації і версії сервісу;

– для забезпечення часової продуктивності опрацювання запитів, створений або змінений об’єкт за допомогою методів PUT та POST, мають повертатися;

– підтримка вибору за вказаними критеріями, впорядкування і посторінкове відображення;

– підтримка MediaType – шлях формування правила для сервера щодо отримання вмісту. При зверненні з браузера до стандартного інтерфейсу веб сервісу, сервер повертає XML, а з Postman – JSON.

– використання тільки стандартного механізму кешування і дати останньої зміни – ці два параметри, які потрібні для розуміння клієнтом того, що вміст не потребує оновлення.

– орієнтація на стандартні коди помилок HTTP.

Властивості HTTP методів наведено у табл. 1.1

Таблиця 1.1

Властивості HTTP

HTTP метод	Ідемпотентність	Безпечність
OPTIONS	+	+
GET	+	+
HEAD	+	+
PUT	+	-
POST	-	-
DELETE	+	-
PATCH	+	-

Аналізуючи методи, які наведено у табл. 1.1, то наприклад «OPTIONS» використовується для одержання параметрів безпеки, HEAD – повертає лише заголовки повідомлення, PATCH використовується для часткової зміни вмістимого.

Як видно з табл. 1.1, усі методи, окрім методу POST відповідають властивості ідемпотентності, що передбачає здатність виконання одного і того самого запиту до сервісу декілька разів, а відповідь кожен раз буде незмінною. Припустимо, що виконалась операція PUT, що змінює об’єкт і у результаті її виконання одержано некоректну відповідь. При цьому невідомо, що спровокувало її виникнення і в який проміжок час. Однак, завдячуючи властивості ідемпотентності, клієнти можуть бути впевненими у цілісності даних, оскільки цю дію можна поворити ще кількка разів.

Властивість безпечності («Safe») вказує на те, що запит до сервера не передбачає внесення змін в об’єкт, тобто метод GET може викликатися довільну кількість разів, однак він не буде змінювати вмістимого об’єкту.

При роботі з web-інтерфейсами будуть використовуватись принципи сервіс орієнтованої архітектури (СОА). СОА це принципи побудови та використання розподілених можливостей, які в свою чергу можуть бути керованими власними зонами, в яких вони розташовуються [].

СОА базується на тому, що певні об’єкти можуть мати можливості вирішення потреб інших об’єктів. При чому зв’язок між потребами та можливостями не обов’язково повинен бути один-до-одного, оскільки можуть існувати такого роду потреби, для вирішення яких потрібно використати комбінацію можливостей різних об’єктів, і навпаки.

Ключовими концепціями для опису СОА парадигми є: видимість, взаємодія і ефект. Видимість є здатністю потреби і відповідної можливості бачити один одного. зазвичай це надається шляхом описів таких аспектів як функції і технічні вимоги, пов'язані обмеження і політики, а також механізми для доступу або отримання відповіді. Описи повинні бути форматовані з використанням загальнодоступного і зрозумілого синтаксису і семантики [].

Під взаємодією розуміється використання певної можливості. Наприклад, при проміжному обміні повідомленнями, взаємодія продовжується через серії обміну інформацією та викликами дій. Тут багато різних взаємодій, але вони всі закріплені до певного контексту виконання — множини технічних і бізнес елементів, які створюють зв'язок між потребами і можливостями, які в свою чергу складаються з цих елементів. Все це дозволяє постачальникам і споживачам сервісів взаємодіяти і надає точки прийняття рішення для використання різних політик і контрактів [].

Ціллю виконання певної взаємодії є необхідність отримання певного ефекту в реальному світі, при чому цей ефект може бути представлений у вигляді отримання певної інформації чи зміни стану об’єкта.

Центральним поняттям СОА є сервіс — це механізм, за допомогою якого потреби та можливості з’єднуються разом.

До сервісу застосовуються описані вище парадигми СОА: видимість, взаємодія та ефект. Видимість проявляється через опис сервісу, який містить інформацію необхідну для взаємодії з ним і описує його в таких термінах, як, вхідна інформація, вихідна інформація і зіставлені семантики.

Об’єкти, що надають можливості виступають постачальниками сервісу, а об’єкти, які використовують надані можливості для вирішення власних потреб — користувачами сервісу. Завдяки опису сервісу користувач може вирішувати сервіс, якого постачальника йому потрібно використати.

1.2. Аналіз сучасних архітектур при проектуванні програмних інтерфейсів

1.2.1. Аналіз сервіс орієнтованої архітектури

Основне призначення СОА архітектури це зменшити затрати на інтеграцію системи та збільшити можливості для її масштабування. Масштабованість систем, які побудовані з використанням принципів СОА досягається за рахунок того, що обмеження на мережі взаємодії є мінімальними. Таким чином, ці системи стають

більш гнучкими та адаптивними і можуть швидше пристосовуватись до різних змін в бізнес-процесах.

Враховуючи те, які переваги надає використання СОА архітектури, варто розглянути основні етапи та принципи, яких потрібно дотримуватись, щоб в повній мірі використати можливості сервіс орієнтованої архітектури.

Як було сказано в пункті 1.1 ключовим елементом в СОА є сервіс. Сервіс це механізм, який здійснює доступ до однієї або більше можливостей, доступ надається через описаний інтерфейс і застосований згідно обмежень і політик, визначених в описі сервісу. Як раніше згадувалось, сервіс може мати як постачальника так і користувача при чому вони можуть не знати про існування одне одного. Також, користувач сервісу не знає внутрішньої структури сервісу і які операції в середині нього виконуються та може взаємодіяти з ним лише через описані публічні інтерфейси.

До сервісів застосовуються основні поняття СОА, такі як: видимість, взаємодія та ефект.

Видимість сервісу проявляється в тому, що користувач та постачальник сервісу повинні бути видимі одне одному. Щоб задовольнялись правила видимості потрібно виконати умови інформованості, готовності та досяжності (рис. 1.1).

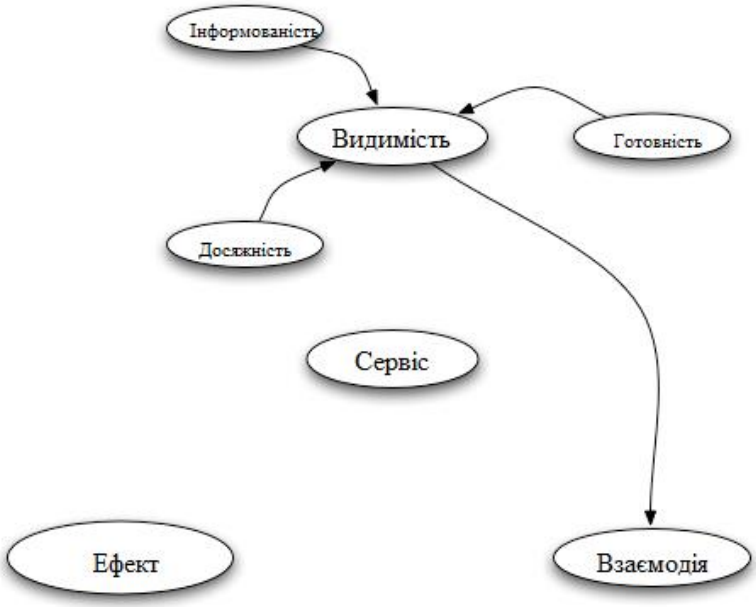

Рис. 1.1. Концепції, що відносяться до видимості

Концепція інформованості передбачає, що користувачі та постачальники сервісу повинні знати про існування одне одного. Це означає, що опис сервісу повинен бути зроблений таким чином, щоб потенційні користувачі знали про існування сервісу та його можливості.

З концепції інформованості плавно випливає концепція готовності. Вона передбачає, що коли користувач обрав певний сервіс, сам сервіс має бути готовий до взаємодії, а користувач до отримання послуги заради якої використовує його.

Досяжність це зв'язок між сервісними учасниками в місці взаємодії, можливо, шляхом обміну інформацією, тобто вони повинні володіти можливість спілкування одне з одним.

Споживач сервісу може мати намір взаємодіяти з сервісом і може отримати всю інформацію необхідну для з'єднання з сервісом. Однак, якщо сервіс недосяжний, наприклад, якщо не існує каналу зв'язку між споживачем і постачальником, тоді, дійсно, сервіс не є доступним для споживача [].

Взаємодія з сервісом викликає певну дію сервісом, що в основному полягає в відправці повідомлень чи зміні стану. На рис. 1.2 показана модель взаємодії з сервісом.

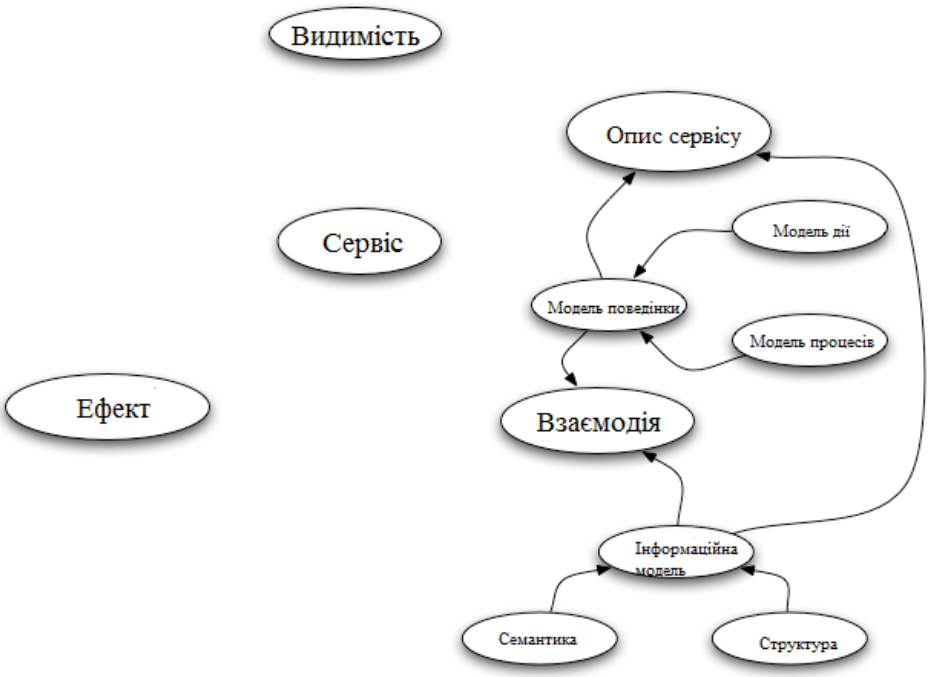

Рис 1.2. Модель взаємодії з сервісом

Як видно з рис. 1.2 основними концепціями при взаємодії є інформаційна модель та модель поведінки, які випливають з опису сервісу.

Інформаційна модель характеризує інформацію якою інформацією можна обмінюватися з сервісом. Вона включає в себе формат інформації, структурні зв’язки, а також визначення використовуваних елементів.

Розширення, за допомогою яких одна система може ефективно інтерпретувати інформацію, віддану від іншої системи визначається семантикою зобов'язань. Семантика зобов'язань системи є зв'язком між системою та інформацією, яку вона може прийняти. Що досить мінливо і залежить від додатків, наприклад, сервіс шифрування розуміє всю інформацію як потік байтів для шифрування або дешифрування, тоді як сервіс бази даних може сприймати ту саму інформацію як потік елементів запиту на вибірку і/або модифікацію даних.

У будь-якому випадку можливий поділ інтерпретації інформаційного блоку на структуру і семантики, при цьому ці дві частини є частинами інформаційної моделі.

Знання подання, структури і форми запитуваної інформації є ключовими на початковому етапі для гарантування ефективної взаємодії з сервісом. Існує кілька рівнів такої структурної інформації, що включає кодування символьних даних, формат даних структурних типів зіставлених з інформаційними елементами [].

Семантика показує в якій формі потрібно передати інформацію, щоб вона могла бути правильно про інтерпретована із збереженням закладеного в неї значенням.

Наступною вимогою для взаємодії є модель поведінки. Вона визначає дії які може виконати сервіс та певні тимчасові аспекти, які може використовувати сервіс. В моделі поведінки виділяють: модель дій — характеризує дії, які можуть виконуватися сервісом; модель процесів — тимчасові зв'язки і тимчасові властивості дій та подій, зіставлених при взаємодії з сервісом.

Взаємодія користувача з сервісом передбачає, що сервіс повинен повернути користувачу певну інформацію або якось модифікувати стан системи це і буде називатися ефектом в реальному світі.

Варто більш детально зупинитись на таких концепціях пов’язаних із сервісами, як: опис сервісу, контекст виконання, контракт і політики, які відносяться до сервісу і елементів взаємодії з ним.

Одна з ознак сервіс орієнтованої архітектури є велика кількість пов'язаних документів та описів.

Опис сервісу представляє інформацію необхідну для того, щоб використовувати сервіс. У більшості випадків, не існує жодного «правильного» опису, а існують необхідні елементи опису, що залежать від контексту і потреби сторін, що використовують взаємодіє об'єкт.

У той час як існують певні елементи, які ймовірно є частиною будь-якого сервісного опису, а саме інформаційна модель, при цьому багато інших елементів, такі як функції і політики можуть варіюватися.

Метою опису є спрощення взаємодії й видимості між учасниками сервісної взаємодії, особливо коли учасники існують в різних областях. Існування описів, робить можливим для потенційних учасників створювати системи з використанням сервісів і досить просто отримувати необхідні сервіси.

На практиці рекомендовано представляти опис сервісу з урахуванням стандарту, затвердженого формату. Такий формат спрощує використання загальних інструментів розробки процесів (такі як механізми виявлення), що може бути отримано з опису сервісу.

У той час як концепція СОА підтримує використання сервісу без необхідності знати споживачеві деталі сервісної взаємодії, опис сервісу уможливлює критичну інформацію, яку потребує споживач для прийняття рішення про використання або невикористання сервісу. Зокрема, споживач сервісу потребує отримання такої інформації:

– що сервіс існує і досяжний;
– що сервіс виконує деяку функцію або набір функцій;
– що сервіс функціонує з дозволу певного набору обмежень і політик;

– що сервіс буде (в деяких явним або неявним рамкам) взаємодіяти з політиками, які прийняті споживачем сервісу;

– як взаємодіяти з сервісом для досягнення необхідних цілей, включаючи формат і вміст інформаційного потоку між сервісом і споживачем та очікуваної послідовності обмінюваної інформації [].

Політика представляє деякі обмеження або умови використання, застосування чи опису об'єкта яким володіє будь-який учасник. Контракт, з іншого боку, представляє договір двох або більше сторін. Як і політики, контракти, також відносяться до умов використання сервісу; вони також можуть обмежувати передбачувані ефекти в реальному світі від використання сервісу.

Контекст виконання сервісної взаємодії це множина інфраструктурних елементів, об'єктів процесу, політик тверджень і угод, які визначені як частина сервісної взаємодії, і різних видів шляхів між потребами і можливостями.

Дотримання вище описаних принципів СОА дозволяє створювати високо масштабовані системи із слабкими зв’язками між елементами, що також покращує можливості для тестування та забезпечення якості як системи в цілому так і кожного окремого сервісу.

Наступним кроком є аналіз підходів, що використовуються в СОА для виконання взаємодії компонентів між собою чи користувачами.

1.2.2. Remote procedure call (RPC)

RPC це концепція при якій виклики методів в програмі передаються на виконання через мережу. Найбільшу продуктивність цей підхід показує коли зв’язок між взаємодіючими комп’ютерами є досить швидким, а об’єми даних, що передаються є невеликими.

Для виконання RPC потрібна наявність stub генератора та бібліотеки виконання.

Stub генератор призначений спростити процес упаковки аргументів функції та автоматизувати отримання результатів. Також, на етапі його компіляції можуть

бути виявлені допущені помилки, а код може бути оптимізовано, що призведе до підвищення продуктивності такої системи.

Входом для stub-компілятора буде лише набір викликів сервера, які бажає використовувати клієнт. Stub генератор приймає на вхід певний інтерфейс з якого генеруються методи, які може використовувати на своїй стороні клієнт, щоб створити RPC виклик.

Зовнішньо це виглядає ніби виклик звичайної функції, хоча після її виклику клієнтський stub повинен виконати наступні кроки:

– створити буфер повідомлень, який, зазвичай, є масивом байтів певного розміру;
– запакувати необхідну інформацію в буфер повідомлень;
– відправити повідомлення на RPC сервер;
– зачекати на отримання результатів;
– розпакувати код повернення та інші аргументи;
– повернутися до виклику.

На стороні сервера відбуваються наступні кроки:

– розпакування повідомлення;
– виклик необхідної функції;
– запакування результатів;
– відправлення відповіді.

Бібліотека виконання бере на себе більшу частину роботи в RPC пов’язану із продуктивністю та надійністю роботи системи.

Однією із найважливіших проблем, яку повинна вирішувати бібліотека виконання є так звана проблема іменування. Тобто, потрібно знайти саме той вузол системи на якому знаходиться процедура, що викликається через RPC. Найпростішим її вирішенням може слугувати використання імені хоста та порта конкретної обчислювальної машини.

Ще однією проблемою яка постає при використанні RPC є вибір протоколу транспортного рівня. З однієї сторони, протокол TCP забезпечує надійну доставку

даних, але при цьому виконується багато додаткових запитів, які можуть суттєво сповільнити час виконання програми, з іншого боку, UDP не потрібно виконувати такої кількості викликів, але він не гарантує надійності доставки даних. Враховуючи ці аспекти потрібно певним чином комбінувати використання цих протоколі для того, щоб забезпечити достатню швидкодію та надійність програмного продукту [].

1.2.3. CORBA

CORBA представляє собою стандарт щодо рекомендацій написання розподілених додатків. На рис. 1.3 наведені основні компоненти CORBA архітектури:

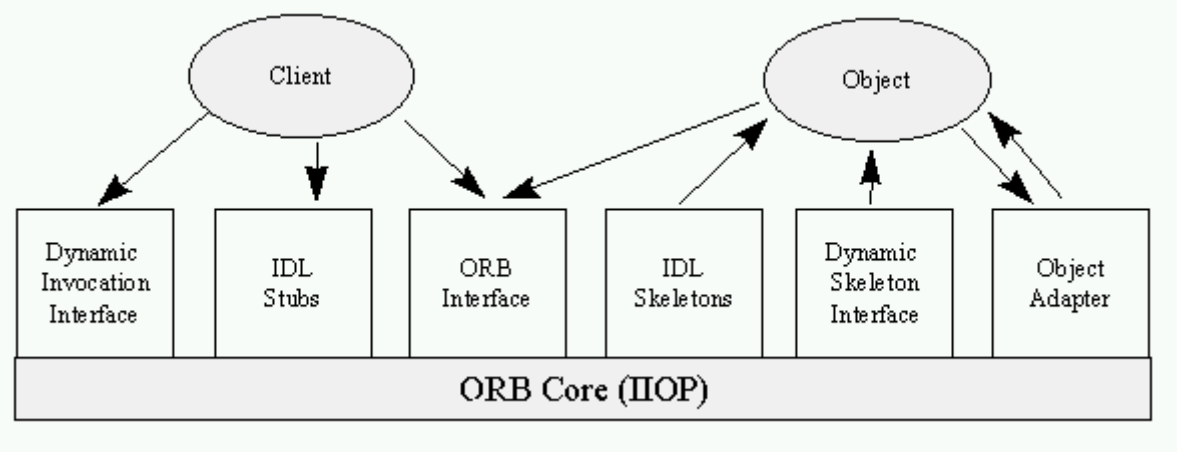

Рис 1.3. Основні компоненти CORBA архітектури

CORBA може використовуватися в різних ситуаціях. Так як CORBA забезпечує інтеграцію між серверами різних виробників, від мейнфремів, серверів до десктопів, портативних пристроїв і вбудованих систем, це програмне забезпечення середнього рівня для великих (і не дуже) систем. Одна з найпоширеніших областей застосування CORBA — сервіси, які обслуговують велику кількість клієнтів, з великим числом запитів і володіють високою надійністю. CORBA працює за лаштунками багатьох найбільших в світі сайтів. Специфікації масштабованості і стійкості до збоїв забезпечують життєздатність

цих систем. Але CORBA використовується не тільки для великих додатків, вона використовується також в системах реального часу і маленьких вбудованих системах [6].

Характеристика основних компонентів CORBA:

– Object, об'єкт — це сутність CORBA, що складається з індивідуальності, інтерфейсу і реалізації, яку також називають сервером (servant).

– Servant, сервер, серверний об'єкт — це реалізація об'єкта, що визначає операції, описані в IDL інтерфейсі CORBA. Сервери можуть бути написані на різних мовах програмування, включаючи C, C ++, Java, Smalltalk, Ada і Perl.

– Client, клієнт — програмна сутність, що викликає операції реалізації об'єкта. Доступ до сервісів віддаленого об'єкта повинен бути прозорий для того хто їх викликає. В ідеалі виклик не повинен відрізнятися від звичайного виклику об'єкта. Інші компоненти на рис. 1.3 забезпечують цю прозорість.

– Object Request Broker, Брокер об'єктних Запитів (ORB) — ORB забезпечує механізм для прозорої доставки запитів клієнта реалізації об'єкта. ORB спрощує розподілене програмування, приховуючи від клієнта деталі виклику методів. Запити клієнта виглядають як виклик локальних функцій. Коли клієнт викликає операцію, ORB відповідає за пошук реалізації об'єкта, прозору його активацію в разі необхідності, доставку запиту серверу і повернення відповіді.

– CORBA IDL stubs and skeletons, стаб і скелетон CORBA IDL — ці артефакти виконують роль «клею» між клієнтським і серверним додатками і ORB. Трансформація оголошень CORBA IDL в оголошення на цільовій мові програмування виконується компілятором CORBA IDL. Використання компілятора зменшує можливість неузгодженості клієнтських стабів і серверних скелетонів і збільшує можливості для автоматизованої оптимізації компілятором.

– Dynamic Invocation Interface, Інтерфейс динамічного Виклику (DII) — інтерфейс дозволяє клієнту звертатися безпосередньо до механізму запитів, що надається ORB. Додатки використовують DII для динамічного формування запитів до серверів без необхідності компіляції із стаб інтерфесом IDL. На відміну від стаб

IDL (які дозволяють запити тільки в стилі RPC), DII дозволяє клієнтам робити неблокуючі відкладені синхронні (окремі операції відправки та отримання) і односторонні (тільки відправка) виклики.

– Dynamic Skeleton Interface, Інтерфейс динамічних скелетонів (DSI) — це серверний аналог клієнтського DII. DSI дозволяє ORB доставляти запити реалізації об'єкта, для якої на момент компіляції не відомий тип реалізованого об'єкта. Для клієнта не відрізняються виклики через типізовані IDL скелетони і динамічні скелетони.

– Object Adapter, Об'єктний Адаптер — допомагає ORB в доставці повідомлень об'єкту і в його активації. Об'єктний адаптер асоціює реалізацію об'єкта з ORB. Об'єктні адаптери можуть бути спеціалізовані для забезпечення підтримки певних стилів реалізації (таким як об'єктний адаптер OODB для сховищ і бібліотечний адаптер для локальних об'єктів).

1.2.4. Simple Object Access Protocol

SOAP представляє собою протокол для передачі повідомлень у середовищі, яке є децентралізованим і розподіленим та використовує XML. Даний протокол був розроблений компанією Microsoft, але згодом був стандартизований консорціумом W3C.

Повідомлення, відправляються SOAP в форматі XML, і складаються з трьох частин – оболонка, заголовок і тіло, що можна побачити на рис 1.4. Оболонка включає заголовок повідомлення і тіло, та містить багато інформації, необхідної для обробки повідомлення, в тому числі опис виду даних, які будуть знайдені всередині оболонки, а також інформацію про те, як слід обробляти ці дані. Вона також містить інформацію про відправника і одержувача повідомлення.

SOAP не вимагає того, щоб повідомлення містили заголовок, хоча на практиці, повідомлення будуть включати їх, коли SOAP буде використовуватись в веб-службах. Інформація знайдена в заголовках може виконувати різноманітні

функції, такі як забезпечення автентифікації. Дані, що знаходяться в заголовках організовані в блоки заголовку. Може бути один або кілька блоків в заголовку.

Тіло повідомлення містить в собі дані. Дані можуть бути запитом про надання інформації — наприклад, коли запитуюча служба шукає реєстр сервісів для web-служби. Або це може бути відповіддю на запит про надання інформації, наприклад, коли реєстр відправляє назад дескриптор служби. Дані, що знаходяться в тілі організовані в під елементи. Може бути один або більше проміжних елементів в тілі запиту.

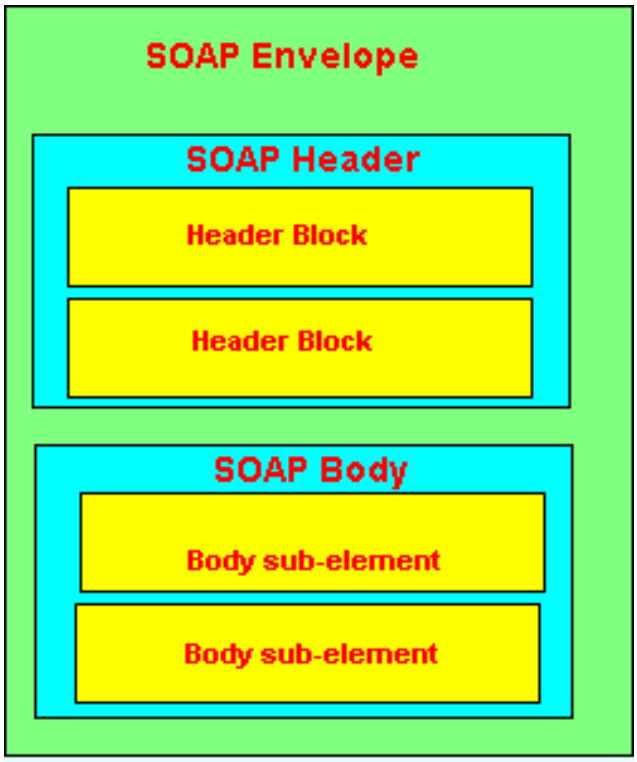

Рис. 1.4. Структура SOAP повідомлення

Найбільша потенційна проблема з протоколом в тому, що різні реалізації SOAP не можуть працювати один з одним. Розробники рідко працюють безпосередньо з самим SOAP — замість цього, використовується SOAP-інструментарій, який створює SOAP-повідомлення, або інструменти web-сервісів, які створюють їх. Часто це набори інструментів перекладу з мови на SOAP — наприклад, можна використовувати інструментарій, щоб перевести функцію COM-

виклику до SOAP і інший інструментарій, щоб перевести функції JAVA в виклики SOAP. Проблема полягає в тому, що кожен набір інструментів може реалізувати SOAP по-різному, і вони можуть в кінцевому підсумку працювати не належним чином один з одним []. SOAP використовує протоколи транспортного рівня, такі як HTTP, SMTP та інші.

1.2.5. Representational State Transfer (REST)

REST — це стиль побудови розподілених систем, який призначений спростити виконання запитів до web-серверів, які були в SOAP.

Як було сказано в пункті 1.2.4, SOAP використовує для своєї роботи повідомлення в форматі XML, що встановлює певні обмеження для сервісів, які взаємодіють за його допомогою, оскільки, потрібно додатково конвертувати інформацію. REST призначений позбавитись від цієї проблеми, оскільки, може використовувати, як XML так і JSON формати для передачі повідомлень, а також їхнє поєднання.

Також, REST підтримує явне застосування HTTP методів при імплементації певних операцій. Такий підхід до проектування REST забезпечує однозначність операцій CRUD і методів HTTP, тобто:

- формування об’єкта на сервері виконується методом POST;
- одержання об’єкта передбачає застосування методу GET;
- зміна або оновлення стану об’єкта забезпечується методом PUT;
- видалення об’єкту забезпечується методом DELETE.

Ключовим елементом в REST є те, що стан не зберігається, тобто, серверу і клієнту не має необхідності відслідковувати його. Коли клієнт не комунікує з сервером, то останній не має уявлення про його існування. Сервер також не веде облік минулих запитів. Кожен запит розглядається як самостійний [].

Сервер, у якому не передбачено зберігання стану працює набагато краще, оскільки вся відповідальність покладається на сторону клієнта. Усі вище описані

підходи так чи інакше використовуються при побудові web-сервісів та дозволяють реалізовувати архітектуру СОА.

1.3. Cloud-сервіси та програмні інтерфейси

Важливе місце в розробці сучасного програмного забезпечення посідають cloud-сервіси. Вони можуть суттєво прискорити розробку web-інтерфейсів, їх розгортання та підтримку.

При проектування web-сервісів найкраще використовувати можливості платформи як послуги (PaaS) — це повноцінна сфера розробки і розгортання в cloud з ресурсами, які дозволяють створювати будь-які додатки, від простих хмарних додатків до просунутих хмарних додатків промислового класу. При цьому купуються лише необхідні ресурси у постачальника хмарних служб, та оплачуються в міру їхнього використання.

PaaS також дає можливість знизити або взагалі уникнути затрат і труднощів, які мають відношення до придбанням ліцензій програмного забезпечення і управлінням ними, базової інфраструктури додатків, ПЗ проміжного рівня, засобів розробки і інших ресурсів. Розробник керує додатками і службами, які розробляє, а постачальник хмарних служб зазвичай керує всім іншим [].

Для проектування web-інтерфейсів найкраще застосовувати можливості служби додатків, від Microsoft Azure.

Служба додатків — це рішення на основі моделі платформа як послуга (PaaS) від Microsoft Azure, яке дозволяє створювати web і мобільні додатки для будь-якої платформи або пристрою. Крім того, з його допомогою можна інтегрувати програми з рішеннями SaaS, підключатися до локальних додатків і автоматизувати бізнес-процеси. Azure запускає програми на повністю керованих віртуальних машинах. При цьому можна вибрати тип віртуальних машин, які необхідно використовувати для конкретного навантаження (загальні або виділені).

Служба додатків має можливості для роботи в мережевому і мобільного середовищі, які раніше пропонувалися в складі окремих рішень web-сайти Azure і

мобільні служби Azure. Вона також передбачає нові можливості для автоматизації бізнес-процесів і розміщення хмарних API. Інтегрована служба додатків дозволяє легко створити єдине рішення, що поєднує кілька компонентів (наприклад, web-сайти, серверні частини мобільних додатків, інтерфейси API RESTful і бізнес-процеси) [].

Служба додатків має кілька типів додатків, які призначені для розміщення конкретних робочих навантажень:

– web-додатки — для розміщення веб-сайтів і веб-додатків;

– мобільні додатки — для розміщення серверної частини мобільних додатків;

– додатки API — для розміщення API RESTful;

– додатки Logic Apps — для автоматизації бізнес-процесів та інтеграції систем і даних в cloud без необхідності створювати код.

Програма може складатися з декількох різних додатків служби додатків. Наприклад, якщо додаток складається з web-інтерфейсу і серверної частини API RESTful, можна зробити наступне:

– розгорнути обидва цих компонента в web-додатку;

– розгорнути код зовнішнього інтерфейсу в web-додатку, а код серверної частини — в додатку API [7].

При проектуванні web-інтерфейсів можна використовувати API додатки Azure, які мають наступні особливості:

– Використання існуючого API "як є". Не потрібно змінювати код в існуючих інтерфейсів API, щоб скористатися перевагами додатків API. API може використовувати будь-яку мову або платформу, підтримувані службою додатків, включаючи ASP.NET і C #, Java, PHP, Node.js і Python.

– Просте використання. Інтегрована підтримка метаданих API Swagger спрощує використання API безліччю різних клієнтів. Дозволяє автоматично створювати код клієнта для API на різних мовах, включаючи C #, Java і Javascript.

– Простий контроль доступу. Забезпечує захист додатків API від доступу без перевірки автентичності, не вносячи змін в код. Служби вбудованої перевірки автентичності захищають API для доступу інших служб і клієнтів, що представляють користувачів. У число підтримуваних постачальників посвідчень входять: Azure Active Directory, Facebook, Twitter, Google і обліковий запис Microsoft. Клієнти можуть використовувати бібліотеку перевірки автентичності Active Directory (ADAL) або пакет SDK для мобільних додатків.

– Інтеграція з Visual Studio. Виділені інструменти в Visual Studio спрощують роботу по створенню, розгортання, налагодженні додатків API і управління ними.

– Інтеграція з додатками логіки. Створені додатки API можуть використовуватися додатками логіки служби додатків.

Крім того, додаток API може використовувати переваги функцій, що надаються web-додатками і мобільними додатками. Вірно і зворотне: якщо використовувати web-додаток або мобільний додаток для розміщення API, воно може використовувати такі переваги додатків API, як метадані Swagger для створення коду клієнта і CORS для між доменного доступу в браузері [8].

1.4. Висновки до розділу

У даному розділі одержано наступні результати:

1 Проведено аналіз існуючих технологій проектування комп’ютерних систем і встановлено, що найбільш популярним шляхом імплементації складних систем є інтеграція та агрегація компонентів з використанням прикладних програмних інтерфейсів, що дає змогу підключати різні сервіси та ефективно керувати ними.

2 Проведено дослідження щодо процесу проектування архітектури комп’ютерних систем та її структурних елементів на рівні програмних інтерфейсів, встановлено принципи організації та імплементації архітектури SOA, що дало

можливість визначити вузькі місця застосування API і сформувати пропозицію зниження впливу негативних факторів із застосуванням математичного апарату представлення процесу проектування прикладних програмних інтерфейсів.

З Проаналізовано особливості застосування хмарних сервісів при проектуванні, реалізації і підтримці комп’ютерних систем, що дало змогу обґрунтувати використання підходу платформи як сервісу і забезпечити ефективність використання ресурсів, масштабованості та гнучкості налаштування параметрів системи.

РОЗДІЛ 2
ІМПЛЕМЕНТАЦІЯ МЕТОДУ ПОБУДОВИ ПРИКЛАДНИХ ПРОГРАМНИХ ІНТЕРФЕЙСІВ В КОМП'ЮТЕРНИХ СИСТЕМАХ

2.1. Розробка методу проектування програмних інтерфейсів з врахуванням особливостей компонентного підходу

На сьогоднішній день програмне забезпечення стає все більш багатофункціональним, а програмний код збільшується у розмірах, що ускладнює його підтримку та масштабування. Щоб полегшити подальшу підтримку та розробку таких великих проектів Мартіном Фаулером та Джеймсом Льюісом було запропоновано поняття мікросервісного чи компонентного підходу.

Стиль архітектури на основі мікросервісів є підходом, який забезпечує можливість функціонування цілісного додатка, що побудований з набору простих підпрограм (сервісів). Кожен з мікосервісів виконується у власному контейнері і взаємодіє з іншими за допомогою простих механізмів, наприклад HTTP. Такі сервіси реалізують визначені бізнес-вимоги, а їх розгортання відбувається за допомогою спеціалізованого середовища автоматизації незалежно один від одного. Характерною особливістю цього підходу є те, що централізоване керування практично відсутнє, або зведене до мінімуму. Окрім цього, мікросервіси можуть бути реалізованими за допомогою неоднорідних мов програмування та з використанням різних СКБД для зберігання даних [4].

Компонентом, прийнято вважати, таку одиницю програмного забезпечення, яка може бути не залежно замінена чи оновлена.

Компонент повинен бути:

– малим, тобто розроблятись однією командою розробників таким чином, щоб бути в подальшому зрозумілим одною людиною;

– сфокусованим — компонент повинен вирішувати лише одну бізнес-задачу;

– слабо зв’язаним, тобто зміни в одному компонентів не потребують змін в іншому;

– високо узгодженим — компонент містить всі необхідні методи вирішення поставленої задачі.

Для передачі даних між сервісами використовується підхід «смарт сервіси і проста взаємодія». Він означає, що шині передачі даних все-одно, що передається і яка логіка виконується, а всю логіку беруть на себе сервіси. В іншому ж випадку, через велику кількість операцій з оброки даних на шині буде значно сповільнюватись процес комунікації.

Великою перевагою, яку надає компонентний підхід є використання децентралізованого зберігання даних. Децентралізоване зберігання передбачає використання компонентом своєї і лише своєї бази даних. Бази даних при цьому не взаємодіють(рис. 2.1).

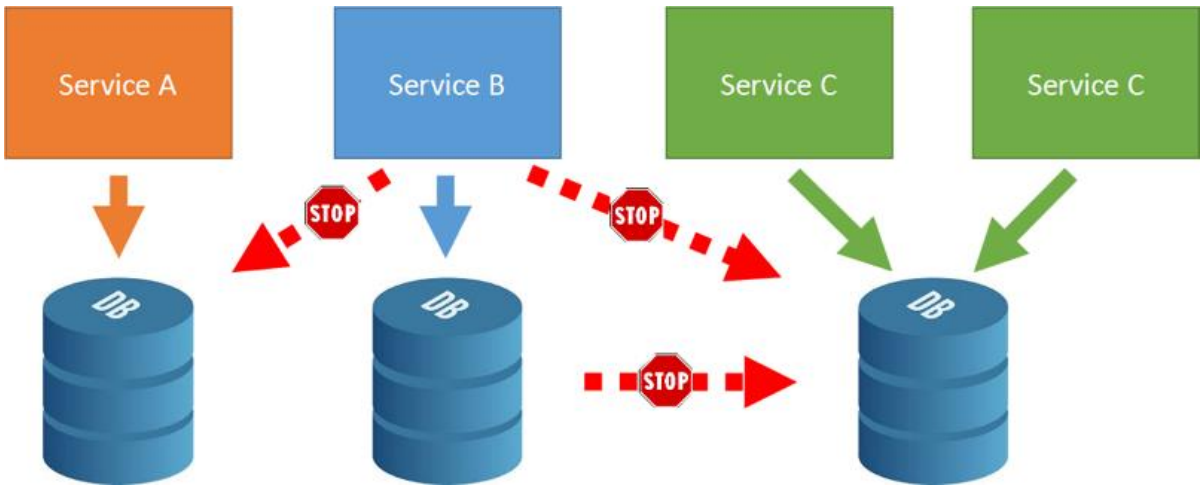

Рис. 2.1. Взаємодія сервісів з базами даних

Взаємодія сервісів між собою можлива лише через мережу. Розробляючи програму за таким підходом, варто виходити з того, що компоненти можуть не працювати і тому цей момент потрібно враховувати при розробці нових компонентів. Також, варто передбачити системи автоматичного розгортання та моніторингу сервісів, оскільки в системі їх може бути дуже багато і фізично прослідкувати за роботою кожного є, часто, неможливим.

Для виконання взаємодії компонентів між собою використовується API Gateway (рис. 2.2).

Шлюз API є ефективним інструментом при реалізації додатку на основі архітектури мікросервісів. Якщо в backend передбачено декілька сервісів, то для доступу до них доцільно додати елементарний компонент, основне призначення якого буде полягати у збиранні бізнес-запитів у відповідності до функцій сервісів. Це дозволить забезпечити маршрутизацію повідомлень. Шлюз API доставляє повідомлення у вигляді, який передбачений специфікацією конкретного користувача. Для прикладу, якщо існує дві версії веб-орієнтованого додатку, що працює як на звичайному комп'ютері, так і на мобільній платформі. У такому випадку варто використовувати два різних шлюзи API, що забезпечать можливість збору даних з сервісів і різного їх представлення. Логіка функціонування такого Gateway повинна бути мінімальною і зводитися лише до передачі даних, а не до їх опрацювання. Це зумовлено тим, що при зростанні функціональної складності виникають проблеми з дублюванням та підтримкою працездатності.[5]

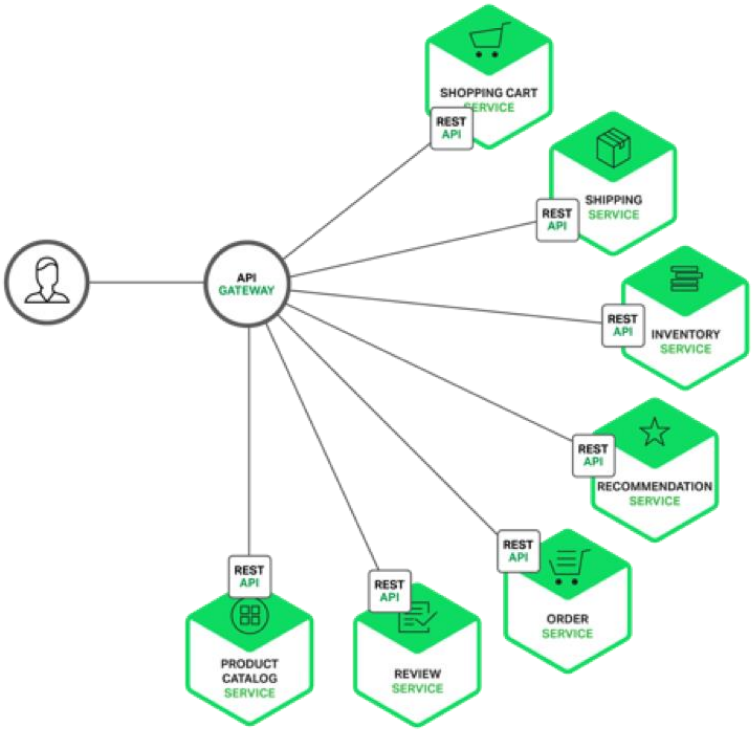

Рис. 2.2. Використання шлюза API

У компонентному підході виділяють три види зв’язків між компонентами:

– виявлення сервісу – випадок, коли компоненти знають про існування один одного, а взаємодія відбувається на пряму;

– шина повідомлень – випадок, коли використовується патерн «publisher-subscriber», при цьому ні «publisher» не володіє інформацією про підписників, ні «subscriber» не знає про джерела надходження даних;

– гібридний.

Розрізняють виявлення сервісу на стороні сервера та виявлення сервісу збоку клієнта. У першому випадку, клієнт комунікує з деяким сервісом не напряму, а через балансувальника навантаження.

Сьогодні існує велика кількість балансувальників навантаження, зокрема, це стосується провайдерів хмарних сервісів від Amazon, Azure і т.п. Кожен такий балансувальник керується власними правилами при передачі запиту ч ивиклику сервісу(рис. 2.3).

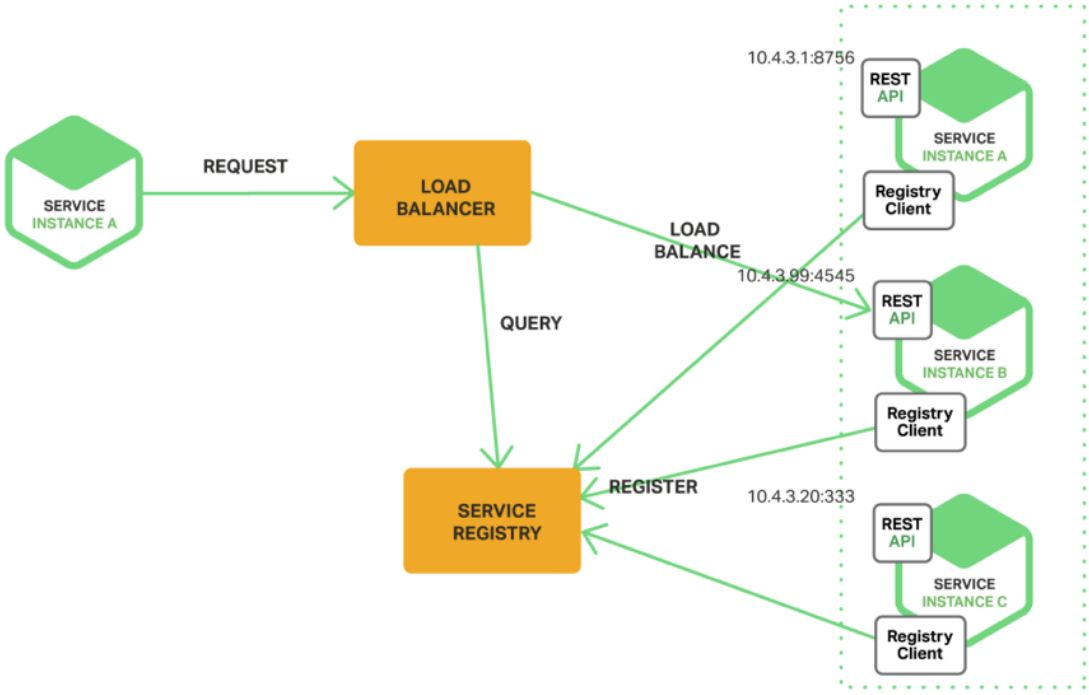

Рис. 2.3. Зв’язок виявлення сервісу на стороні сервера

Як видно з рис. 2.3, окрім балансувальника навантаження існує ще один сервіс реєстрів. Такий сервіс може бути організований по-різному з врахування особливостей того, хто, яким чином і з якого місця реєструється. Окрім цьогго, можна забезпечити таку функціональність, що сервіс реєстрів записуватиме усі види компонентів. Балансувальник навантаження одержує усі дані від сервісу реєстрів. У такому випадку, основна задача балансувальника зводиться до простого одержання інформації від сервісу реєстрів про те, де знаходяться користувацькі сервіси з подальшим формуванням запитів до них. Задача сервісу реєстрів полягає у зберіганні даних про компоненти. Такі дані він може отримувати внаслідок самостійного опитування компонентів, або з файлу конфігурації або іншими шляхами [5].

Виявлення сервісу на стороні клієнта представляє собою кардинально інший підхід до взаємодії мікросервісів, який показано на рис. 2.4.

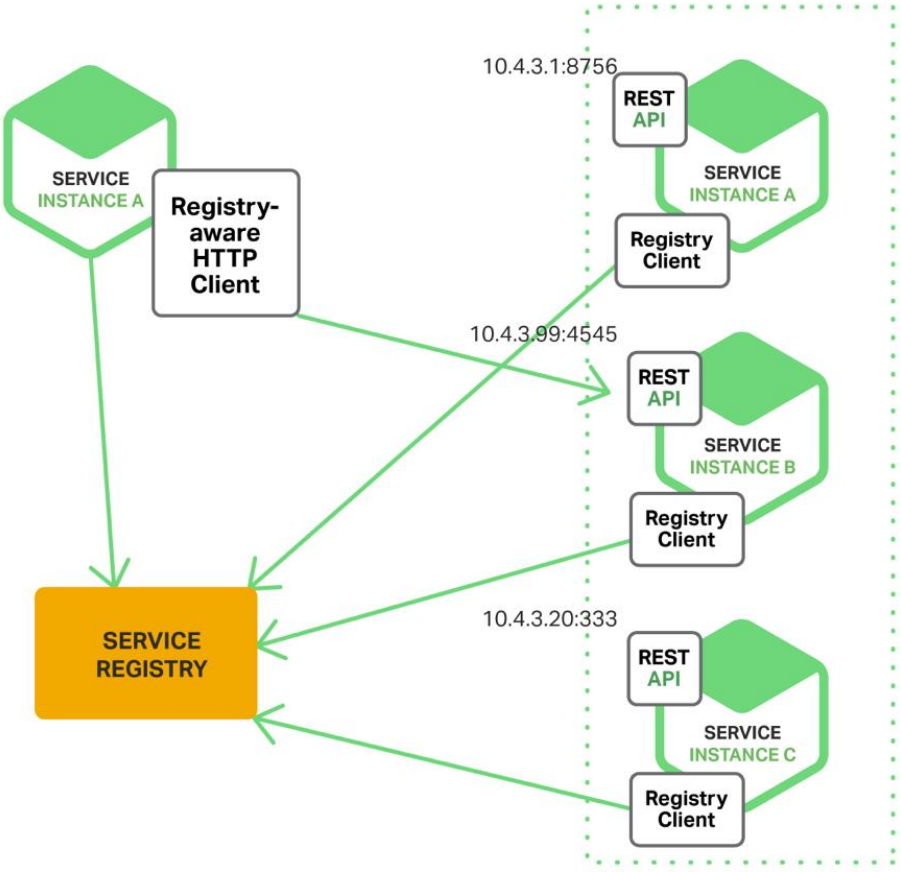

Рис. 2.4. Виявлення сервісів на стороні клієнта

Виходячи з представлення балансувальника навантаження проілюстрованого на рис.2.4, мікросервіси звертаються безпосередньо до сервісу реєстрів для одержання адреси компонентів. Такий підхід є дещо кращим за той, який показано на рис. 2.3 при відповідності умові, що клієнт є внутрішнім і не комунікує із зовнішнім середовищем з якого можуть вестись наприклад атаки DDoS [5].

Шина повідомлень застосовується для вирішення спеціалізованих задач і не призначена для виконання повторних запитів чи передачі значних обсягів даних. Патерн шини повідомлень забезпечує ізольованість постачальників та одержувачів даних. Видавці («publisher») не володіють інформацією про те, кому вона необхідна, як і підписники («subscriber») не знають про джерело походження даних. При цьому можливий варіант, коли одні і ті ж дані опрацьовуються різними видавцями та різними одержувачами.

При такій організації системи необхідно передбачити ще два функціональні модулі або сервіси: до функцій першого брокера належить збір повідомлень, а другий компонент відповідає за момент часу їх доставки. Не доцільно використовувати шину повідомлень для передачі інформації, що вимірюється у мегабайтах.

Шина повідомлень функціонує за принципом командного патерну і використовується лише в тих випадках, коли необхідно надіслати сповіщення від одного мікросервісу до іншого щодо зміни стану першого. В залежності від сповіщення відбудеться реакція взаємопов’язаних сервісів.

Для забезпечення описаного вище сценарію, найбільш доцільним є використання гібридного підходу, тобто поєднання шини повідомлень з підходом виявлення сервісів на стороні клієнта або сервера. До шини повідомлень надходить повідомлення про зміну певних даних. Далі «subscriber» звертається до сервісу реєстрів одержує за значенням ідентифікатора відправника (publisher) місце, звідки можна одержати необхідні дані. При такій організації, шина повідомлень не є надто завантаженою [5].

2.2. Формалізація структури програмних інтерфейсів

2.2.1. Математична модель компоненту та інтерфейсу

Математична модель описує абстрактне відображення того, яким чином представляються важливі атрибути, параметри, достеменна структуру і можливість до комунікації елементів одного середовища компонентів.

В загальному випадку, для представлення моделі будь-якого компоненту можна застосувати нступний вираз:

$$Comp = < CName, CInt, CFact, CInp, CServ > \quad (2.1)$$

де $CName$ – унікальна назва сервісу;

$CInt = \{ CInt^i \}$ – сукупність інтерфейсів, які мають відношення до компонента;

$CFact$ – інтерфейс управління об’єктами;

$CInp = \{ CInp^i \}$ – сукупність визначених програмних реалізацій компонента;

$CServ$ – інтерфейс, що задає множину системного програмного забезпечення, потрібного для забезпечення працездатності компонента і його комунікації у відповідному середовищі [6].

На множині $CInt = CInt_1 \cup CInt_2$ передбачено визначення двох видів компонентів. Перший вид забезпечує опис внутрішніх компонентних інтерфейсів, а другий – інтерфейси комунікації між зовнішніми компонентами.

Для представлення моделі будь-якого інтерфейсу використовується наступний вираз:

$$CInt^i = (IntName^i, IntFunc^i, IntSpec^i) \quad (2.2)$$

де $CInt^i$ – інтерфейс управління об'єктами, які належать до конкретного компоненту;

$IntName^i$ – назва інтерфейсу;

$IntFunc^i$ – множина програмних функцій, які імплементує інтерфейс;

$IntSpec^i$ – опис інтерфейсу.

Вимога цілісності інтерфейсу є необхідною вимогою для його існування і описується таким чином:

$$\forall CInt^i \in CInt \, \exists CImp^j \in CImp [\, Provide(CInt^i) \subseteq CImp^j \,] \qquad (2.3)$$

де $Provide(CInt^i)$ – функціонал, що гарантує виконання програмних функцій $CInt^i$.

Для кожного з таких інтерфейсів може існувати кілька реалізацій, які розрізняються особливостями функціонування (наприклад, операційним середовищем, засобами збереження даних і т.д.) [6].

Взаємодія двох компонентів $Comp_1$ і $Comp_2$ визначається наступною необхідною умовою: якщо $CInt_1^i \in CInt_1$, то повинен існувати $CInt_2^k \in CInt_2$ такий, що:

$$Sign(CInt_1^i) = Sign(CInt_2^k) \,\&\, Provide(CInt_1^i) \in CImp_2^j \qquad (2.4)$$

де $Sign(...)$ – сигнатура відповідного інтерфейсу.

Інтерфейс $CInt^i$ з формули 2.2 визначає необхідні методи екземплярами компоненту, до яких відносяться:

- пошук та визначення необхідного екземпляру компоненту – Locate;
- створення екземпляру компоненту – Create;
- видалення екземпляру компоненту – Remote.

Ці методи складають основу для будь-яких інтерфейсів управління екземплярами в межах будь-якої компонентної моделі.

Будь-яку реалізацію компоненту можна представити у наступному виді:

$$CImp^j = (ImpName^j, ImpFunc^j, ImpSpec^j) \quad (2.5)$$

де $ImpName^j$ – ідентифікатор імені реалізації компонента;

$ImpFunc^j$ – сукупність реалізацій програмних функцій;

$ImpSpec^j$ – специфікація реалізації (опис умов виконання, параметрів налаштувань реалізації і т. д.).

Реалізація компоненту – це сукупність програмних функцій певної сигнатури і типи даних для переданих параметрів або параметрів, які повертаються. За цими описами порівнюють імплементацію компонента та інтерфейсів на основі нотацій власних програмних функцій, які реалізують зв'язування інтерфейсу і компоненту. Відмінність у зв'язуванні при використанні компонентного підходу та, наприклад, об'єктно-орієнтованого, полягає в тому, що процес формування зв'язків виконується на пізніших, а іноді на завершальних стадіях чи в режимі run-time виконання програми.

Взаємозв'язок між компонентною та об'єктною моделлю можна відслідкувати за допомогою наступних тверджень.

В процесі свого функціонування компонент за допомогою методу Create з інтерфейсу $CFact$ породжує екземпляри:

$$CFact.Create : Comp \rightarrow (Cins_k^{ij}) \quad (2.6)$$

$$Cins_k^{ij} = (Iins_k^{ij}, IntFunc^i, ImpFunc^j) \quad (2.7)$$

де $Cins_k^{ij}$ – екземпляр *k* компоненту, який надає свій функціонал за допомогою інтерфейсу $IntFunc^i$ і забезпечує реалізацію цього інтерфейсу за допомогою $ImpFunc^j$;

$Iins_k^{ij}$ – унікальний ідентифікатор екземпляру компоненту.

Нехай існує де-яка об'єктна система подана у вигляді діаграми класів:

$$Osyst = (Oclass, G) \tag{2.8}$$

де $Oclass = \{ Oclass^i \}$ – множина класів;

G – об'єктний граф, який відображає зв'язки і відношення між класами та екземплярами.

Кожен клас поданий у вигляді:

$$Oclass^i = (ClassName^i, Method^i, Field^i) \tag{2.9}$$

де $ClassName^i$ – ім'я класу;

$Method^i = \{ Method_j^i \}$ – множина методів;

$Field^i = \{ Field_n^i \}$ – множина змінних, що визначають стан екземплярів класу.

Нехай $Pfield^i \subset Field^N$ – множинна внутрішніх змінних, які доступні ззовні. Кожній $Pfield^i \subset Field^N$ ставлять у відповідність методи $get < Pfield_n^i >$ та $set < Pfield_n^i >$ для присвоєння та вибірки відповідного значення, тобто ці змінні стають властивостями у сучасній компонентній моделі. Це означає, що для інших класів при зверненні до цих змінних будуть застосовуватися властивості, представлені у вигляді методів.

Введемо нову множину методів:

$$Im ethod^i = Method^i \cup \{ get < Pfield_n^i > \cup set < Pfield_n^i > \} \tag{2.10}$$

яким прирівняєм інтерфейс $Ifunc^i$, що складається з прототипів методів, які входять в $Imethod^N$.

Паралельно з $Osyst$ буде розглядатись система:

$$Isyst = (Ifunc, IG) \tag{2.11}$$

де $Ifunc = \{ Ifunc^i \}$ – множина інтерфейсів;

IG – інтерфейсний граф, ідентичний графу G.

Клас $Oclass^i$ породжує свої екземпляри(об'єкти):

$$Obj_k^i = \{ ObjName_k^i, Method^i, Field^i \} \tag{2.12}$$

яким в системі $Isyst$ будуть відповідати елементи інтерфейсу

$$Iobj_k^i = \{ Iname_k^i, Ifunc^i \} \tag{2.13}$$

При цьому характерною особливістю є те, що для будь-якого елементу інтерфейсу не визначеною є його реалізація. Поставивши конкретному інтерфейсу у відповідність реалізацію $ImpFunc^j$ формується елемент, який аналітично можна записати таким чином

$$Iobj_k^{ij} = \{ Iname_k^i, Ifunc^i, ImpFunc^j \} \tag{2.14}$$

який еквівалентний екземпляру компонента

$$Cins_k^{ij} = (Iins_k^{ij}, IntFunc^i, ImpFunc^j) \tag{2.15}$$

Основні розбіжності визначають наступні чинники. По-перше, вибір придатної реалізації відбувається у процесі розгортання, а не на ранніх процесах, як це потрібно при об’єктно-орієнтованому підході. Наступне, екземпляр класу володіє визначеними у класі властивостями і методами і не може використовувати більшу їх кількість, ніж це передбачено у базовому або батьківському класі. На противагу цьому, реалізація компонента може підтримувати декілька не пов'язаних між собою інтерфейсів. Об’єктна та інтерфейсна системи є еквівалентними.

Отже в процесі проектування програмного забезпечення на основі компонентного підходу доцільно використовувати властивості об’єктно-орієнтованої парадигми з відповідними інструментальними засобами їх підтримки. Використовуючи об’єктно-орієнтовані методи проектування, паралельно створюють об’єктну систему і відповідну їй систему інтерфейсів без конкретизації їх реалізації. У результаті проектування одержують множину інтерфейсів. Після цього потрібно розв’язати задачу щодо забезпечення відповідного покриття інтерфейсів реалізацією компонентів. У даному випадку немає необхідності враховувати реалізацію функціоналу програмної системи, яка виконується компонентами шляхом інтеграції (збірки) і розгортання компонентної програмної системи [6].

2.3. Обґрунтування методу інтеграційного тестування програмних інтерфейсів

Традиційні програмні комплекси стають все більшими в розмірах, більш складними і не контрольованими, що призводить до збільшення витрат на технічне обслуговування та розробку, а також зниження продуктивності праці в плані розробки, щоб забезпечити необхідний рівень якості і знижує швидкість переходу на нові технології. Тому зростає попит на розробку нових, більш економічно та часово вигідних стандартів на розробку ПЗ. Одним з потенційно можливих рішень на сьогодні є розробка на базі компонентів. Він створений на ідеї того, що програма повинна являти собою сукупність компонентів поєднаних між собою, як в області

електроніки чи механіки. Такі інженерні області як будівництво, хімія, електроніка є більш старшими тому в них вже є свої стандарти та правила. Швидкість змін в ПЗ та реалізація різних точок зору є складним та тривалим процесом для стандартизації поняття компоненту в сфері розробки ПЗ. Оскільки процес розробки із застосуванням КОП є швидшим та економнішим, то він зараз широко використовується. Дотримання вимог якості є хорошою практикою, тому на рівні компонентів забезпечується завдяки використанню автономного тестування. Але при інтеграції компонентів непередбачувані ситуації можуть виникнути на різних рівнях. Тому необхідно виконувати тестування на рівні інтеграції компонентів.

Коли обговорюється інтеграція компонентів перш за все звертається увага на інтерфейс, тому що інтерфейс поєднує два і більше компоненти для того, щоб програма була придатною для використання. Інтерфейс є точкою для взаємодії компонентів, за допомогою яких клієнтський компонент може користуватися послугами оголошені в інтерфейсі наданому провайдером. Кожен інтерфейс ідентифікується за своїм іменем та унікальним ідентифікатором. Кожен інтерфейс може включати в себе множину операції, де кожна операція виконує конкретне завдання. Компоненти ПЗ можуть бути включені в систему, як блоки, так як показано на рис. 2.5. на прикладі системи бронювання квитків в готелі, де кожен компонент є протестованим та інтегрованим в нове середовище для виконання необхідних завдань. Компоненти поєднані одне з одним в новому середовище не тільки розуміють це, але вимагають додаткових зусиль для інтеграції. Інтеграція компонентів є однією з найважливіших стадій, які повинен реалізувати користувач, якщо інтерфейс компоненту підключено не правильно це може викликати помилку, або навіть з’єднання правильне, але результат не є таким на який очікували, то потрібно проводити повне тестування інтеграції.

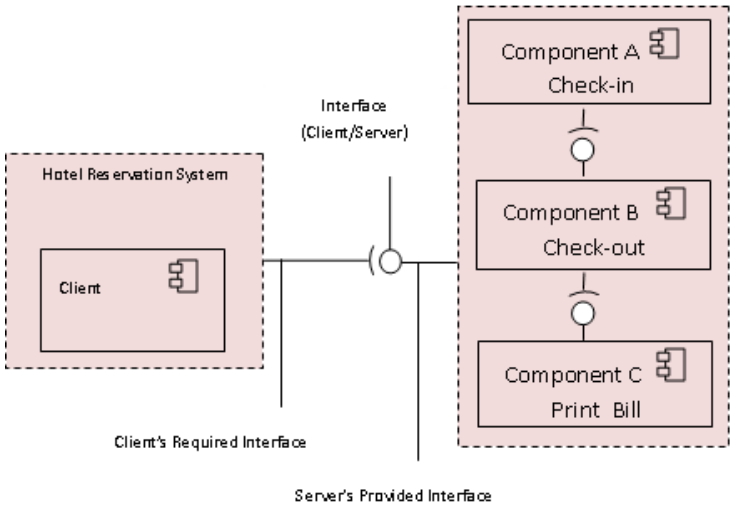

Рис. 2.5. Компонент системи бронювання готелю з інтерфейсом клієнта та провайдера

На рис. 2.6 наведено UML 2.0 діаграму на якій показано взаємодію двох компонентів: компонент для обробки замовлення клієнта звертається до компонента, що відповідає за процес зміни рахунку на клієнтській карті, функціонал якої останній і надає.

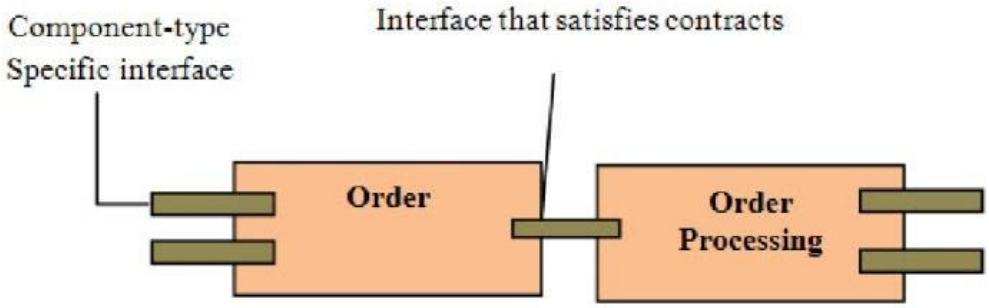

Рис. 2.6. Відображення компонентів на UML 2.0 діаграмі

Перед тим як перейти до питань пов’язаних з процесом інтеграції компонентів важливо розглянути процес створення компоненту та рівень автономного тестування. Створення компоненту є складним завданням через відсутність стандартів, які б забезпечували підтримку їх на різних мовах та платформах. Нижче буде наведено процес створення компонента та їх автономне тестування.

Для створення компоненту, який зможе взаємодіяти з різними мовами на різних платформах потрібно, щоб він міг бути скомпільований IDL компілятором. Існує багато різних IDL компіляторів створених різними фірмами, які забезпечують адаптивне середовище для багатьох мов.

Наприклад, для створення компоненту використовуючи OmniORB IDL компілятор в IDL мови C++ потрібно виконати наступні кроки:

– скомпілювати IDL (для відображення IDL в C++);

– компілятор IDL створить файли (скелет та інші файли необхідні для CORBA та мережевої взаємодії);

– реалізація методів клієнтської та серверної частини;

– компіляції методів реалізації, сервер, клієнт та інших файлів програми за допомогою компілятора C++;

– пересилання файлів до клієнта та сервера;

– запуск служб іменування, клієнтської та серверних програм на відповідному порті.

Наступним етапом є автономне тестування компонентів.

Автономне тестування проводиться з метою знаходження помилок в програмі. Воно використовується для перевірки специфікації вхідних та вихідних параметрів. Розробники знають, що автономне тестування є важливим кроком для створення надійних додатків. Додаток, що використовує компонент може розроблятись сторонніми розробниками, тому необхідно проводити автономне тестування різних компонентів. Різноманітні фреймворки для автономного тестування можуть пришвидшити процес розробки тестів(наприклад: Junit, Nunit). Процедура тестування може відрізнятись в залежності від фреймворку, але все одно буде перевіряти чи тестована процедура придатна для використання. Автономне тестування полегшує процес рефакторингу, коли процедура змінюється на рівні коду, легко перевірити чи вона досі працює вірно. Тестування може бути проведене в будь-який час, для перевірки функціональних можливостей. Автономні тести дозволяють колективну взаємодію, оскільки створюються за

методологією чорного ящика, тому зміни в коді не повинні впливати на правильність роботи самого тесту.

На даний момент існує багато досліджень, що описують як має бути правильно створений компонент та як забезпечити його надійність, але лише декілька дослідників звернуло увагу на те, як правильно та ефективно організувати інтеграційне тестування компонентів.

Багато компонентів є гарно протестованими, але багато з них було розгорнуто окремо, тому хороше інтеграційне тестування є необхідною умовою для забезпечення якості. Але ця проблема є досить складною тому, що з одного боку необхідно протестувати конфігурацію IDL, а з іншого компоненту на конкретній мові програмування. Оскільки компоненти розробляються різними виробниками, то навіть ідеально протестована система потребує додаткових зусиль для тестування при інтеграції нового компоненту.

Оскільки природа компоненту є неоднорідною, то він може бути реалізований для різних платформ на різних мовах програмування. Операційні системи розроблені на різних платформах є одним із найбільш складних питань. Потрібно, щоб компонент реалізований на будь-якій мові мав змогу підтримувати різні операційні системи, бази даних різноманітних постачальників та проміжне ПЗ для здійснення взаємодії. Існує багато компонентів придатних для створення ПЗ, але в них є обмеження щодо використаних технологій. Коли здійснюється вихід за межі цих технологій, то вони стають не сумісними з іншими середовищами. Наприклад, Microsoft розробила платформу .NET та включила в неї інструмент проміжної мови - Microsoft Intermediate Language (MSIL). MSIL підтримує пакети власної розробки та обмежені можливості Java та COBOL. Він легко компілюється та генерує файли, які підтримують мови Visual C++, Visual Basic, Visual J++, Java, і COBOL. Тому на платформі .NET можна легко використовувати компоненти власної розробки та розроблені на інших мовах не турбуючись про сумісність.

Інше питання яке виникає при інтеграції компонентів є їхня взаємодія. Багато людей вважає, що інтеграція компоненту є лише процесом підключення до правильного шляху. І в багатьох простив випадках компоненти можна дійсно

розглядати як точки взаємодії інтерфейсів. Іншими словами, компонент ПЗ може відправити запит на сервер і отримати від нього відповідь, тільки якщо запит клієнта мав правильний формат. Для більш складних компонентів ПЗ до уваги потрібно взяти проміжне ПЗ, найменування і моделі даних. Проміжний зв’язок показує як різні компоненти взаємодіють одне з одним. Їхня взаємодія може використовувати RMI, ORB чи інший спосіб. Різні структури даних можуть вимагати різного формату проміжного зв’язку. Як правило, програмні компоненти розробляються різними постачальниками, які можуть зробити різні припущення про те, як компоненти взаємодіють і які деталі беруть участь у взаємодії. Крім того, компонент може надавати різні інтерфейси, які можуть мати різні обмеження і різні типи взаємодії одне з одним. Виклик різних послідовностей цих компонентів можуть давати різні результати.

При зростаючому попиті систем багато дослідників почали приділяти велику увагу тестуванню компонентів системи базуючись на тестуванні на автономному та системному рівні. Зараз існує багато публікацій в яких описується наскільки важливим є вдосконалення тестування КОП. Було надано багато технік інтеграційного тестування, нижче буде розглянуто основні з них.

Уніфікована мова моделювання (UML) – універсальна мова моделювання, яка використовується для визначення, візуалізації, конструювання і документування артефактів програмної системи. В UML діаграми станів можуть бути використані для опису статичної і динамічної поведінки компонента або об'єкта з плином часу, моделюючи його життя. UML має багато корисних інструментів, такі як діаграми взаємодії, діаграми станів, діаграми співпраці, діаграми компонентів, які дозволяють диференціювати діяльність компонента в різних фазах, і, отже, можуть бути використані в тестуванні компонентної системи. Всі елементи UML аналізують і застосовують різні критерії тестування для перевірки системи на основі компонентів. В діаграмі станів основними елементами є стани, переходи, події та дії. Стани та переходи визначають усі можливі стани і зміни стану об'єкта під час його життя. Зміни стану відбуваються як реакції на події, отримані з інтерфейсів об'єкта. Ця тестова модель використовує діаграми

станів та співпраці, щоб виявити недоліки існуючі між інтерфейсами, які поєднані між собою в системі. Цей метод включає в себе розробку на базі UML для процесів, що тестуються. Наразі, цей метод тестування ще не реалізований, але в перспективі це може бути автоматичним методом тестування. Граф взаємодії компонентів (CIG) – показує взаємодії і різні залежності між компонентами. CIG виявляє помилки інтерфейсу, що виникають в процесі інтеграції різних компонентів для побудови програми. Підхід використовує статичну і динамічну інформацію для розробки тестів і визначення придатності тесту. Приклад CIG для побудови додатку показано на рис. 2.7. Граф взаємодії компонентів це сформовані набори послуг та необхідних інтерфейсів, в якому кожна вершина являє собою метод інтерфейсу. Ребра представлені з вершин відповідних необхідних інтерфейсів до вершин інтерфейсів для компонентів залежностей, а також надає необхідні інтерфейси для компонента залежності. Запропонований підхід може бути застосований на всіх типах компонентів і він базується на результатах тестування чорного ящика.

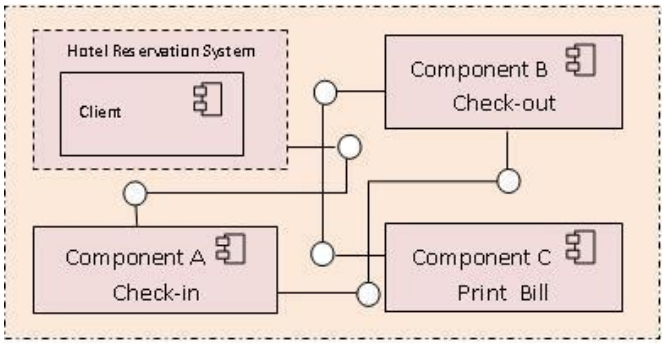

Рис. 2.7. Приклад графу взаємодії компонентів

Сертифікація компонентів є достовірною демонстрацією того, що компонент відповідає вказаним вимогам і є прийнятним для використання, як зазначено. Хороша методологія сертифікації показує надійність на яку заслуговує компонент. Підхід до тестування заснований на методі сертифікації компонентів. Він охоплює сертифікацію всього додатку, а також його складових частин. Сертифікація відноситься до безпеки критично важливих частин компонентів, які були виявлені

в ході аналізу. Для сертифікації компонентів пропонується тестування на базі чорного ящика для рівня компоненту, а на системному рівні технологія бустера помилок, перевірка відсутності оперативного рівня і здійснення заходів щодо захисту етап будівництва.

Тестування взаємодії компонентів базується на припущенні, щодо того, як компоненти взаємодіють один з одним. Ці припущення обмежені як формальні вимоги до випробувань, які визначають вибір тестових прикладів. Методика тестування в даний час орієнтована на взаємодію потоку управління подій; інші види взаємодії невідомі. Таким чином, не можна . Масштабованість є ще одним питанням пов’язаним з цією технікою. У теорії ця модель може масштабуватися від автономного тестування до тестування системи, але при збільшенні розміру моделі програми, вона стає все більш важкою.

Метод на основі метаданих компоненту базується на тому, що інформація про метадані компонента використовується для аналізу і тестування компонентів. При інтеграції добре розробленого компоненту для побудови програми, можливо, буде потрібно виконати ряд завдань, які включають вимоги третіх сторін, щодо інформації про сертифікацію компоненту, аналіз і тестування системи, а також оцінку якості отриману на основі станів до та після інтеграції. Метадані базуються на різних видах інформації в залежності від певного контексту і потреб. Існує унікальне розміщення та унікальну мітку для кожного виду метаданих. Вихідний код для компонента, як правило, не доступний і тому розробник має тільки формальну специфікацію компонентів. Розробник компонента вбудовує цю інформацію у опис програмного забезпечення. Метадані ілюструють як статичний, так і динамічний аспекти компонента. Метадані підвищують точність аналізів в залежності від конкретних функцій, необхідних користувачеві компоненту. Ідея, що лежить в основі концепції метаданих, щоб визначити інфраструктуру, яка дозволяє розробнику компонента додати різні типи даних, які будуть корисні в певному контексті.

Очевидно, що метадані також можуть бути зроблені для внутрішньо використовуваних компонентів, так що всі компоненти, які використовуються для

створення додатку можуть бути оброблені в уніфікований спосіб. Поняття надання метаданих з програмними компонентами вельми пов'язана з тим, що інженери-механіки роблять з апаратними компонентами: так само, як при побудові автомобілів, компонент програмного забезпечення повинен надати деяку інформацію про себе, щоб використовувати в іншому контексті. Чим більше метаданих які можна отримати з/про компоненти, тим менше буде обмеження на завдання, які можуть бути виконані користувачем компонента, таких як прикладні методики аналізу програм, перевірка моделі і т.д. У цьому сенсі, наявність метаданих компонента може сприйматися як гарантія для розробника програми, який вибирає компоненти для розгортання в його системі. Досі цей метод використовується і протестований в невеликих додатках.

Пропонується також наступний метод, який є поєднанням діаграм послідовностей та UML діаграмі станів. Крім того, семантична інформація виражається в об’єктній мові обмежень. Процес побудови використовує стандартні нотації, які здобули широке використання. Це долає проблему вивчення нових позначень та мов при використані пропонованого підходу. Такий підхід передбачає, що всі компоненти в рівні інтеграції вже були попередньо випробувані окремо, і, таким чином, розглядає їх як чорні скриньки. Тестування інтеграції компонентів проводиться з виконанням наступних кроків:

– будується тестова модель на основі UML, заснованого на одному потоці подій. Спочатку створюється діаграма послідовностей для нормального потоку подій. Ця інформація потім використовується для моделювання станів взаємодії компонентів та переходів з використанням UML діаграми станів.

– UML діаграма станів використовується для генерації тестових послідовностей для нормальної і виняткової поведінки.

– тестові випадки вибираються шляхом застосування критеріїв відбору тестових прикладів на даній тестовій моделі UML.

– методика тестування має потенціал для автоматизації, де тестові послідовності можуть бути генеровані автоматично з діаграм станів.

Як підсумок, було розглянуто різні методи інтеграційного тестування для систем, що базуються на компонентах. Також запропонований метод інтеграційного тестування компонентів на базі UML та об'єктній мові обмежень, який долає деякі з існуючих обмежень при інтеграційному тестуванні. Велика частина інтеграційного тестування компонентів може бути автоматизовано. UML є напів завершеною мовою для індикації, візуалізації, побудови і документування артефактів програмних систем і часом не в змозі описати або змоделювати задовільну перспективу. Пропонований підхід не передбачає наявності вихідного коду компонентів тому може бути використаний для інтеграційного тестування придбаних компонентів [11].

2.4. Висновки до розділу

Основні результати даного розділу полягають в наступному:

1. Запропоновано метод побудови API на основі принципів компонентного підходу, що дає змогу врахувати вимоги архітектури SOA та представити в узагальненому, уніфікованому вигляді конструкцію прикладних програмних інтерфейсів.

2. Запропоновано математичне представлення структури програмного компоненту комп'ютерної системи, що використовує парадигму об'єктно-орієнтованого підходу та описується за допомогою елементів теорії множин, що дало змогу використовувати їх у процесі імплементації API за допомогою мікросервісів.

3. Забезпечено формалізоване представлення процесу побудови прикладних програмних інтерфейсів, що дало змогу виявити і забезпечити оптимальність їхнього створення на основі хмарних сервісів.

4. Обґрунтовано метод перевірки взаємозв'язків між функціональними сервісами комп'ютерних систем у хмарному середовищі із застосуванням методів тестування API, що дало змогу мінімізувати зчепленість сервісів і максимізувати зв'язність всередині компонентів.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ API КОМП'ЮТЕРНИХ СИСТЕМ

3.1. Засоби проектування REST API

Використання REST є найбільш ефективним способом побудови програмних інтерфейсів для web-додатків. Розглянемо проектування REST API сервісу, що відповідає запропонованому методу проектування програмних інтерфейсів і принципам самого REST.

Для початку необхідно вибрати технології реалізації сервісу. Перший варіант, який можна використати – технологія WCF Services. Дана технологія має ряд переваг і недоліків з точки зору практики. Перевагою цієї технології є підтримка різних форматів у яких представляються дані: XML, JSON, ATOM. До недоліків належить використання лише webHttpBinding і жодних інших, а також підтримка HTTP Get і POST, інші операції не підтримуються. Враховуючи недоліки WCF Services, що не забезпечують гнучкості опрацювання даних і масштабованості сервісів, перейдемо до аналізу іншої технології Web API. Перевагами даної технології є:

- простота реалізації;
- відкритий вихідний код;
- підтримка усіх можливостей HTTP;
- здатність реалізовувати MVC підхід;
- підтримка багатьох форматів даних.

Враховуючи переваги Web API обґрунтованим є його вибір для проектування і реалізації програмних інтерфейсів як сервісів. На наступному етапі необхідно обрати хостинг і технологію реалізації. Серед найбільш популярних хостингів і технологій реалізації є:

- ASP.NET MVC;
- Azure;

– OWIN — Open Web Interface for .NET:
  - IIS;
  - Self-hosted.

OWIN – це не платформа і не бібліотека, а специфікація, яка усуває високу залежність веб-додатків з реалізацією сервера. Вона дозволяє запускати додатки на будь-якій платформі з підтримкою OWIN. Фактичного специфікація представляє собою структуру даних типу словника і містить назви параметрів і відповідні їх значення. Основні параметри описують у специфікації. На рис. 3.1 наведено структуру з використанням OWIN для функціонування web-сервісів.

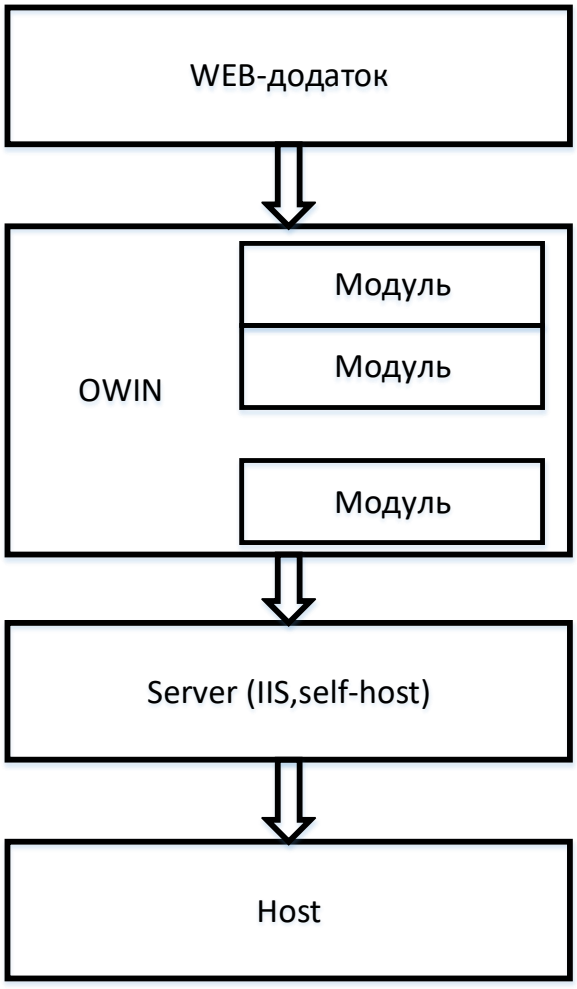

Рис. 3.1. Структура OWIN

Як видно зі схеми, при такій архітектурі на хості розміщується сервер, що підтримує дуже простий словник, який містить сукупність «ключ-значення». Усі

модулі, що підключаються до сервера, конфігуруються саме таким чином. Сервер, що підтримує цей контракт, прив'язаний до певної платформи, вміє розпізнавати всі ці параметри та ініціалізувати інфраструктуру відповідним чином. З цього виходить, що сервіс реалізований за допомогою OWIN може бути вільно, без змін коду, перенесений його між платформами і використовувати те ж саме на інших ОС.

Katana – реалізація OWIN від Microsoft. Вона дозволяє розміщувати OWIN на IIS. У лістингу на рис. 3.2 наведено код конфігурації OWIN на IIS

```
[assembly: OwinStartup(typeof (Startup))]
namespace RestApiDemo
{
public class Startup
{
public void Configuration(IAppBuilder app)
{
var config = new HttpConfiguration();
config.MapHttpAttributeRoutes();
app.UseWebApi(config);
}
}
}
```

Рис. 3.2. Лістинг конфігурації OWIN на IIS

Аналізуючи код лістингу рис. 3.2, для функціонування сервісу необхідно вказати клас Startup. Це простий dll, який запускає IIS. Далі викликається конфігуратор і цього коду достатньо, щоб сервіс запрацював.

Далі перейдемо, власне, до проектування інтерфейсу і проаналізуємо, як все має виглядати і яким правилам відповідати. Всі ресурси у REST іменовані.

Для прикладу розглянемо просту модель з розкладом руху поїздів на станціях. Приклади найпростіших запитів REST:

– кореневі (незалежні) сутності API:

– GET /stations – отримати всі станції;
– GET /stations/123 – отримати інформацію по станції з ID = 123;
– GET /trains – розклад всіх потягів;
– залежні (від кореневої) сутності:
– GET /stations/555/departures – потяги, що проходять через станцію 555.

Отже, у сервісі є станції, і тепер потрібно написати найпростіший контролер (рис. 3.3).

```
[RoutePrefix("stations")]
public class RailwayStationsController : ApiController
{
[HttpGet]
[Route]
public IEnumerable<RailwayStationModel> GetAll()
{
return testData;
}

RailwayStationModel[] testData = /*initialization here*/
}
```

Рис. 3.3. Лістинг контролера

На рис. 3.3 наведено опис контролера, який побудований на атрибутах. Контролер містить назву і повертає список (в даному випадку – випадкові тестові дані).

Однак, бувають ситуації, коли результат запиту містить дуже багато результатів, яких не потребує клієнт. При цьому необхідно реалізувати розбиття результатів на сторінки. Замість цього є простий спосіб – використовувати легку версію OData, яка підтримується Web API (лістинг 3.3)

```
[RoutePrefix("stations")]
public class RailwayStationsController : ApiController
{
[HttpGet]
[Route]
[EnableQuery]
public IQueryable<RailwayStationModel> GetAll()
{
return testData.AsQueryable();
}

RailwayStationModel[] testData = /*initialization here*/
}
```

Рис. 3.3. Лістинг застосування ODATA

IQueryable дозволяє використовувати кілька простих, але ефективних механізмів фільтрації та управління даними на клієнтській стороні. Єдине, що потрібно зробити, це підключити OData пакет з NuGet, вказати EnableQuery і повертати інтерфейс iQueryable. Різниця даної версії пакету у порівнянні із стандартним полягає у відсутності контролера повернення метаданих. Стандартна версія OData повертає відповідь, виконавши обгортку над моделлю, а також здатна також повернути зв’язне дерево ресурсів. Окрім цього, обмежена версія не забезпечує виконання функцій агрегації. Параметри запитів, які можуть використовуватись наведені у табл. 3.1

Таблиця 3.1

Параметри запитів

Параметри запитів	Приклад застосування
$filter	*Stations?$filter=Name eq'Тернопільський вокзал' Stations?$filter=contains (Name,'Терн')*
$select	*Stations?$select=Name,Id*
$orderby	*Stations?$orderby=Name desc*
$top	*Trains?$top=40*
$skip	*Trains?$skip=1000&$top=40*

Призначення та опис параметрів наведено у табл. 3.2.

Таблиця 3.2

Призначення параметрів у запитах

№	Назва параметра	Призначення
1.	$filter	Синтаксично містить фільтр і назву поля для якого застосовується фільтрація
2.	$select	Функція, яка дає змогу спростити і прискорити взаємодію з сервером
3.	$orderby	сортування елементів
4.	$top і $skip	обмеження на вибірку

Атрибут EnableQuery містить сукупність параметрів, які дозволяють накласти обмеження на результати: повертати вказану кількість рядків, не реалізовувати з'єднання, математичні операнди і т. д. До основних параметрів належать атрибути представлені у табл. 3.3.

Таблиця 3.3

Параметри атрибута EnableQuery

Параметр	Атрибут
EnableQuery	AllowedArithmeticOperators
	AllowedFunctions
	AllowedLogicalOperators
	AllowedOrderByProperties
	AllowedQueryOptions
	EnableConstantParameterization
	EnsureStableOrdering
	HandleNullPropagation
	MaxAnyAllExpressionDepth
EnableQuery	MaxExpansionDepth
	MaxNodeCount
	MaxOrderByNodeCount
	MaxSkip
	MaxTop
	PageSize

Як приклад наведемо застосування запиту GET при використанні REST (табл. 3.4).

Таблиця 3.4

Приклад використання GET-запиту

Запит	Опис
GET /stations	отримати всі вокзали
GET /trains	отримати розклад всіх потягів
GET /stations/555/	отримати дані про прибуття з вокзалу 555
GET /stations/555/departures	отримати дані про відправлення з вокзалу 555

Як приклад, розглянемо запис про вокзал з ідентифікатором 555 і сформулюємо задачу щодо необхідності одержання інформації про рух поїздів по ньому. У даному випаду, можливі два варіанти розв’язку, однак тут доцільно використовувати залежну від вокзалу сутність.

При першому варіанті можна виконувати фільтрацію атрибутів контролера з подальшим їх об’єднанням в єдиний клас і жодних проблем при цьому не виникає.

Однак у випадку існування вкладених сутностей, можливе значне розростання в глибину, а це може призвести до виникнення проблем з підтримкою формату.

Для вирішення цієї задачі можна використовувати вбудовані у контролер змінні, які описано у лістингу, що наведений на рис. 3.4.

```
[RoutePrefix("stations/{station}/departures")]
public class TrainsFromController : TrainsController
{
[HttpGet]
[Route]
[EnableQuery]
public IQueryable<TrainTripModel> GetAll(int station)
{
return GetAllTrips().Where(x => x.OriginRailwayStationId == station);
}}
```

Рис. 3.4. Лістинг опису змінних всередині атрибутів контролера

Згідно лістингу, приведеного на рис. 3.4 усі залежні сутності визначено за допомогою окремого контролера і їх будуть розпізнавати усі інші контролери. Це обумовлено тим, що сервісу невідомо про існуючі залежності, незважаючи на те що сутності є вкладеними згідно відображення URL.

Однак можливе виникнення проблеми, пов’язаної з дублюванням сутності «stations», яка полягає в тому, що при її зміні система не буде функціонувати. Для вирішення цього питання доцільно створити в контролері константи, як показано на рис. 3.5.

```
public static class TrainsFromControllerRoutes
{

public const string BasePrefix =
RailwayStationsControllerRoutes.BasePrefix +
"/{station:int}/departures";

public const string GetById = "{id:int}";
}
```

Рис. 3.5. Лістинг використання констант у контролері

Враховуючи попередній лістинг, залежний контролер матиме наступний вигляд (рис. 3.6).

```
[RoutePrefix(TrainsFromControllerRoutes.BasePrefix)]
public class TrainsFromController : TrainsController
{
[HttpGet]
[Route]
[EnableQuery]
public IQueryable<TrainTripModel> GetAll(int station)
{
return GetAll().Where(x => x.OriginRailwayStationId == station);
}
}
```

Рис. 3.6. Лістинг програмного коду залежного контролера

Вище у табл.3.4 проаналізовано можливості застосування запиту GET, однак для реалізації повноцінного сервісу з API необхідно дослідити й інші операції. У табл. 3.5 наведено приклади застосування HTTP-запитів.

Таблиця 3.5

Застосування HTTP-запитів

Операція	Опис	Приклад
POST	Дозволяє створити нову сутність	POST /Stations – додавання нової сутності в колекцію станцій
PUT	Зміна сутності	PUT /Stations/12 змінює сутність ID = 12. JSON, який передано в параметрі, буде записаний замість сутності з ID = 12
DELETE	Видалення сутності	DELETE /Stations/12 — видалити сутність ID = 12

Припустимо, що є API, який варто запропонувати людям, і є доменна модель. Як пов'язані сутності API з доменної моделлю? Насправді вони між собою не пов’язані.

Розглянемо приклад автоматизації готельного бізнесу. Для прикладу існують сутності, які описують готель, резервування номера, номер і прилади, які з ними пов’язані. У даному проекті це дозволяє забезпечувати управління розумними пристроями в кімнатах (рис. 3.7).

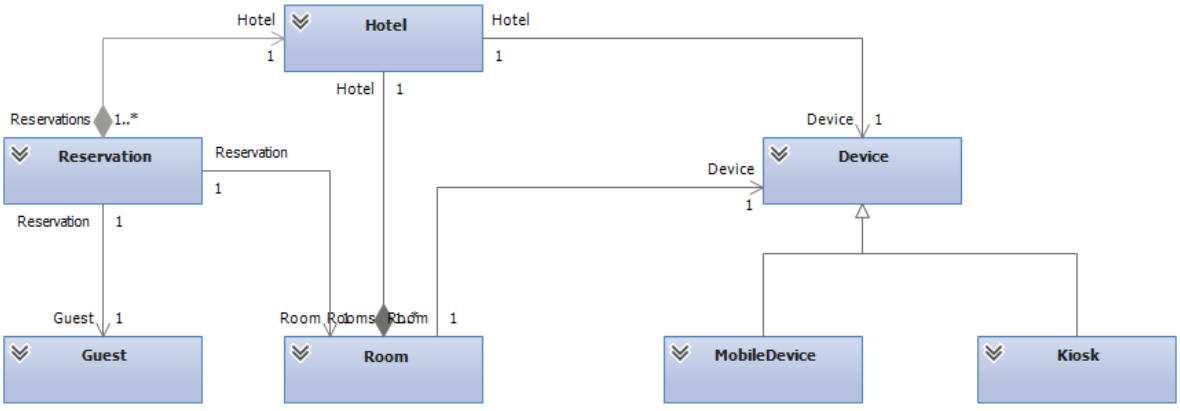

Рис. 3.7. Опис доменної моделі «Готель»

Аналізуючи рис. 3.7 можна побачити, що сутність, яка описує пристрої є окремою і не до кінця відомо яким чином її відділити від готелю Найбільш

ефективним рішенням при цьому є застосування предметно-орієнтованого підходу Domain-driven design. Для цього варто встановити межі предметних областей і границь сутностей, що забезпечують узгодженість системи. Для цього проведемо аналіз поняття ізольований контекст або ізольований домен. Ізольований домен представляє собою набори об’єктів, які є незалежними і описуються кардинально різними моделями. Для прикладу розглянемо готелі та пристрої керування кімнатами як різні контексти, що не є пов’язаними однак існує певне дублювання інформації. У даному випадку, створюється допоміжна сутність, яка формує зв’язок між кімнатами і пристроями керування (рис. 3.8).

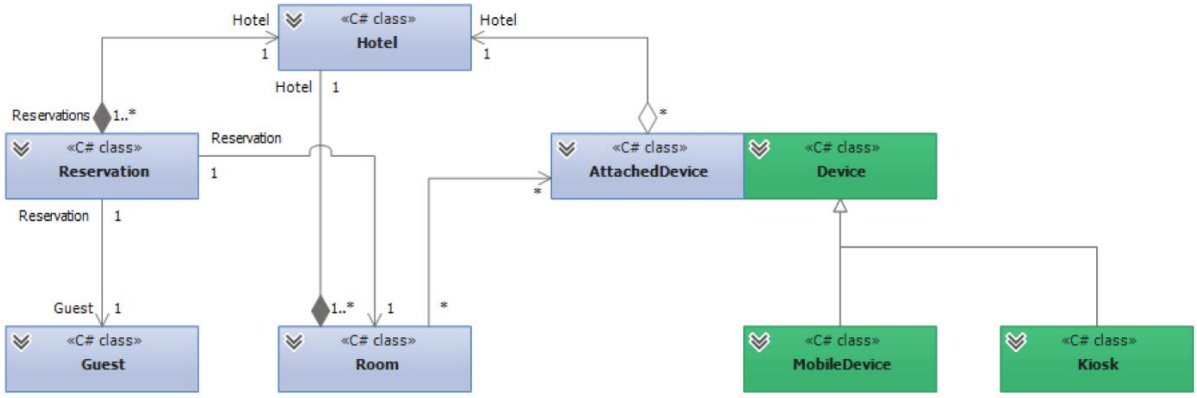

Рис. 3.8. Доменна модель з використанням ізольованого домену

При використанні Domain-driven design aggregate route – сутність, яка має доступ до всіх нащадків предметної області. Це вершина дерева Hotel, звертаючись до якої можна одержати доступ до інших сутностей, а доступу до AttachedDevice таким чином одержати не можливо. Аналогічно, класи, які описують готельний номер і бронювання не матимуть жодного змісту, якщо не матимуть зв’язку з головною сутністю готель. У зв’язку з цим доступ до всіх дочірніх класів виконується через корінь дерева – готель.

Клас, який інтерпретує пристрої, наявні у готельних номерах представляє собою окрему сутність, що представляє вершину зовсім іншого дерева, що містить відмінний набір полів.

Виходячи з проведених процедур аналізу предметної області, можна зробити висновок про те, що у випадку входження сутності до двох Отже, якщо одна сутність входить одночасно до складу двох різних доменів, то необхідно виконати декомпозицію.

Приклади використання операцій в описаній вище предметній області показано у табл. 3.6.

Таблиця 3.6

Застосування запитів у домені

Тип запиту	Значення запиту	Опис результату
PUT	/hotels/555/rooms/105/attachedDevices	Замінити всю колекцію прив’язаних пристроїв на нову
POST	/hotels/555/rooms/105/attachedDevices	Прив’язати ще один пристрій
DELETE	/hotels/12	Видалити опис готелю ID=12
POST	/hotels/123/reservations	Створити нову резервацію в готелі ID=123

Ефективним також є застосування архітектури Command Query Responsibility Segregation при реалізації програмних інтерфейсів. В основі архітектура CQRS лежить підхід, заснований на декомпозиції потоків даних, як показано на рис. 3.9.

При такому представленні архітектури передбачено потік, що забезпечує можливість відправлення певної команди до сервера для зміни стану предметної області. Оскільки, користувач безпосередньо не виконує операції над даними, то ймовірність їх зміни не завжди буде мати позитивний результат.

Таким чином, у випадку відправлення користувачем команди щодо зміни стану сутності, сервер спочатку її опрацьовує і звертається до деякої моделі. Після цього виконується оптимізація для представлення інформації на стороні користувацького інтерфейсу.

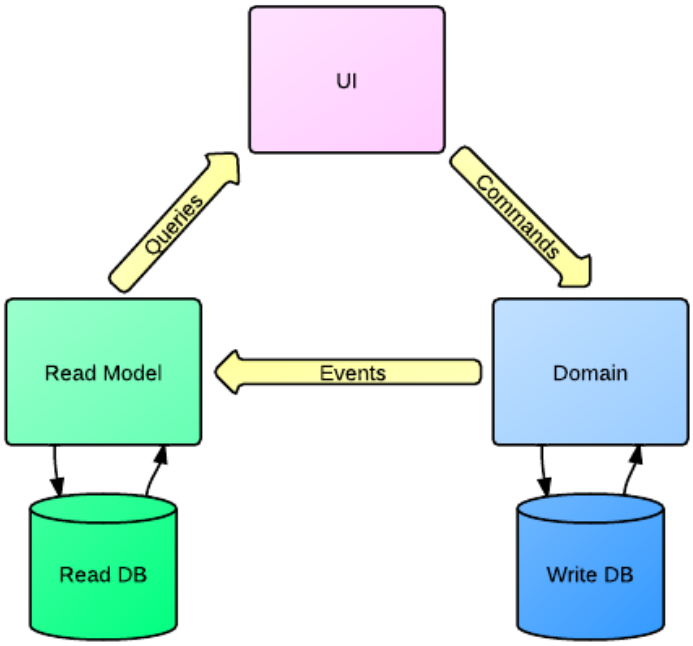

Рис. 3.9. Архітектура CQRS

Підхід, заснований на архітектурі CQRS забезпечує відповідність вимогам до проектування прикладних програмних інтерфейсів REST, тобто у випадку існування сутності «команда», то це означає, що існує інша сутність «список команд».

3.2. Створення додатку в системі Onlizer

Onlizer представляє собою платформу, що містить засоби автоматизації бізнес-процесів без необхідності написання програмного коду і дає змогу забезпечити ефективну інтеграцію як програмних додатків, так і опрацювання потоків даних та і IoT-рішень при побудові комп'ютерних систем.

Початок роботи з платформою Onlizer, як і з іншими системами, передбачає необхідність попередньої реєстрації. Після цього потрібно сформувтаи або авторизуватись у власному кабінеті користувача. Створити кабінет користувача можна дуже простим шляхом, покроково виконавши наступні інструкції.

Перший крок полягає у відкритті сторінки за адресою «https://onlizer.com» у веб- браузері і натисненні елементу керування «Безкоштовна пробна версія», що

розташована на верхній панелі навігації. Окрім цього, можна безпосередньо відкрити сторінку реєстрації в браузері за прямим посиланням.

Для реєстрації необхідно заповнити відповідні поля форми: ім'я; прізвище; надійний пароль для захисту облікового запису; дійсна електронна пошта – рекомендується використання Outlook.com, GMail або сервер електронної пошти організації для гарантованої доставки пошти; дійсний особистий або робочий номер телефону, що можна використовувати для двофакторної авторизації та з інших причин безпеки, номер телефону є необов'язковою інформацією, і його можна залишити порожнім.

Персональні дані клієнтів не надаються стороннім службам. Після заповнення необхідних полів потрібно натисну кнопку «Зареєструватися» і дочекайтеся, поки платформа виконає всі необхідні дії. Після налаштування облікового запису користувач буде перенаправлений на особистий портал і зможе почати створювати інтеграції безкоштовно. Після успішної авторизації користувач переходить на сторінку, як показано на рис. 3.10.

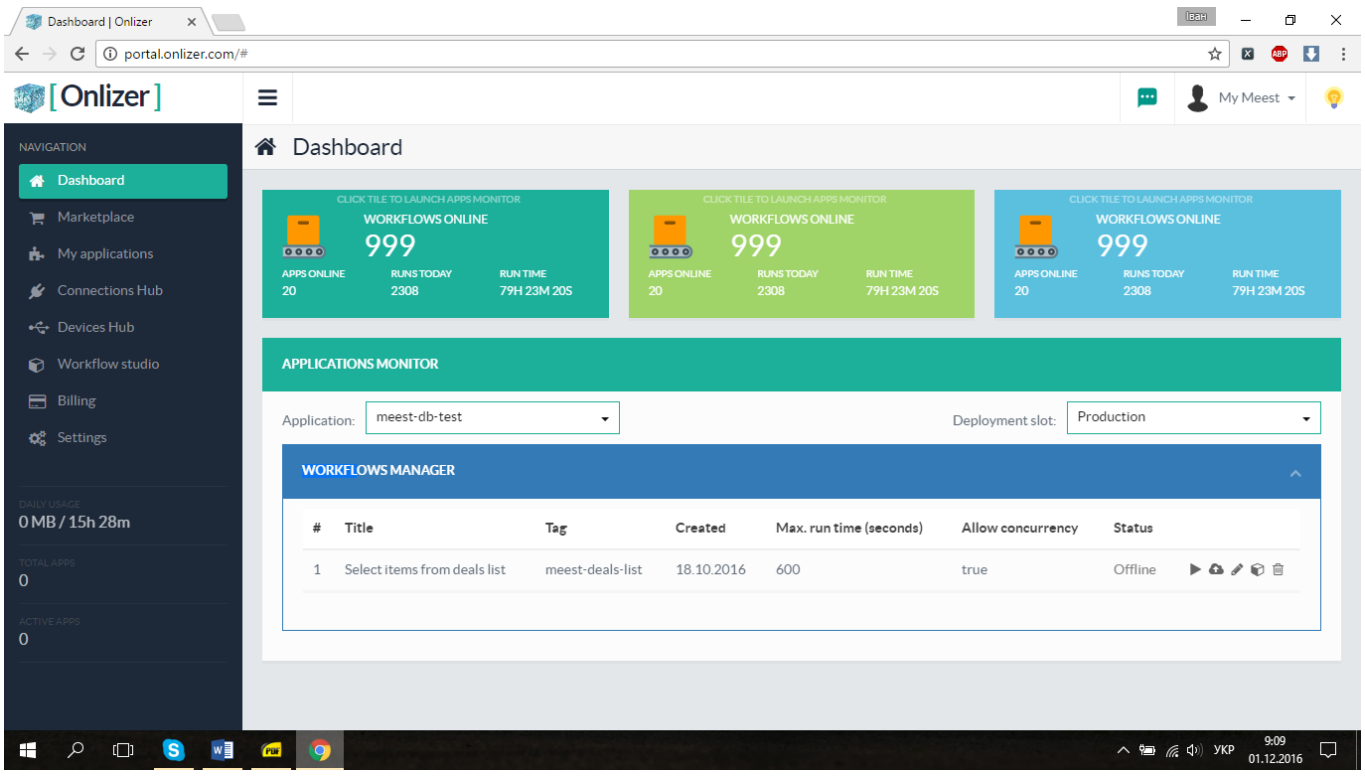

Рис. 3.10. Інтерфейс особистого кабінету Onlizer

На цій сторінці можна побачити існуючі Application, а також які Workflows є в цьому Application та їхній статус.

Application – це об’єкт, що представляє собою головну системну одиницю в платформі інтеграції, яка містить Workflows, що безпосередньо виконують певний функціонал. Application нагадує такий собі міні-проект, що може містити в собі кілька воркфловів, які забезпечують деяку визначену функціональність та інтеграцію заданих сервісів.

Для створення проекту за допомогою Application необхідно перейти до пункту меню та обрати «My applications». Після цього виконується перехід до вказаної сторінки на якій треба натиснути на «New application». Далі з’являється веб-форма, показана на рис. 3.11, у якій необхідно вказати назву додатку, яка має бути унікальною у межах системи.

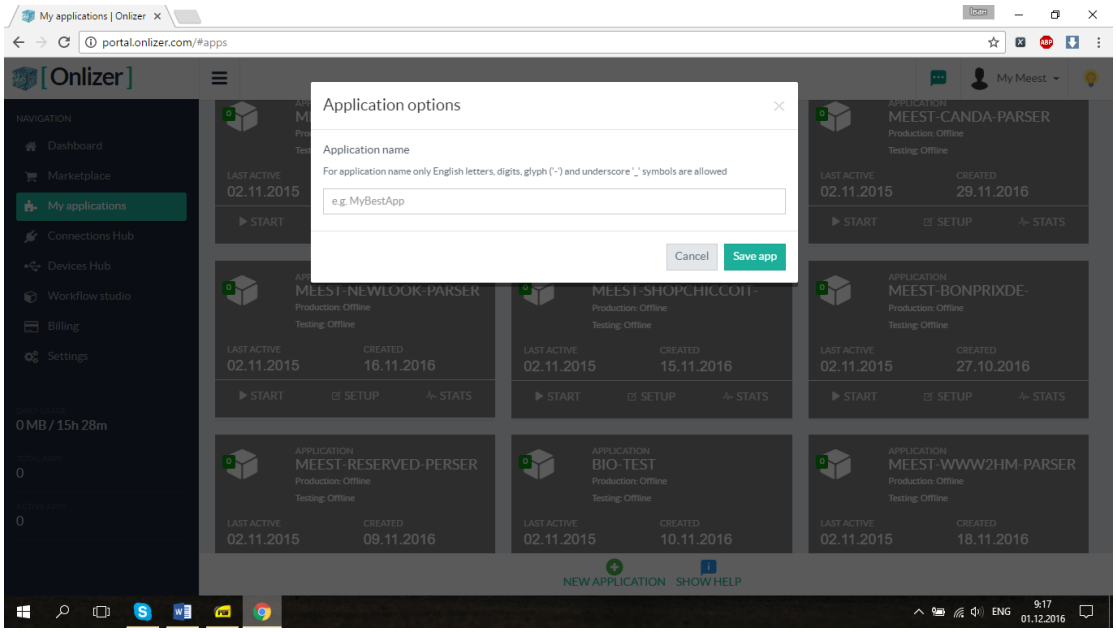

Рис. 3.11. Інтерфейс створення проекту

Після створення проекту виконується автоматичне перенаправлення до сторінки з налаштуваннями, що забезпечує функціональність створення нових потоків виконання, або запустити на виконання існуючі воркфлов. Робочий процес (workflow) представляє собою компонент до складу якого входять інші менш елементарні частини, що разом формують потік для виконання деякої задачі. Для

того, щоб побудувати робочий потік необхідно в параметрах «Application» натиснути «Add workflow» і у вікні, показаному на рис. 3.12 заповнити поля щодо його назви, тег пришвидшеного пошуку, максимально допутиму затримку виконання в секундах і т.п.

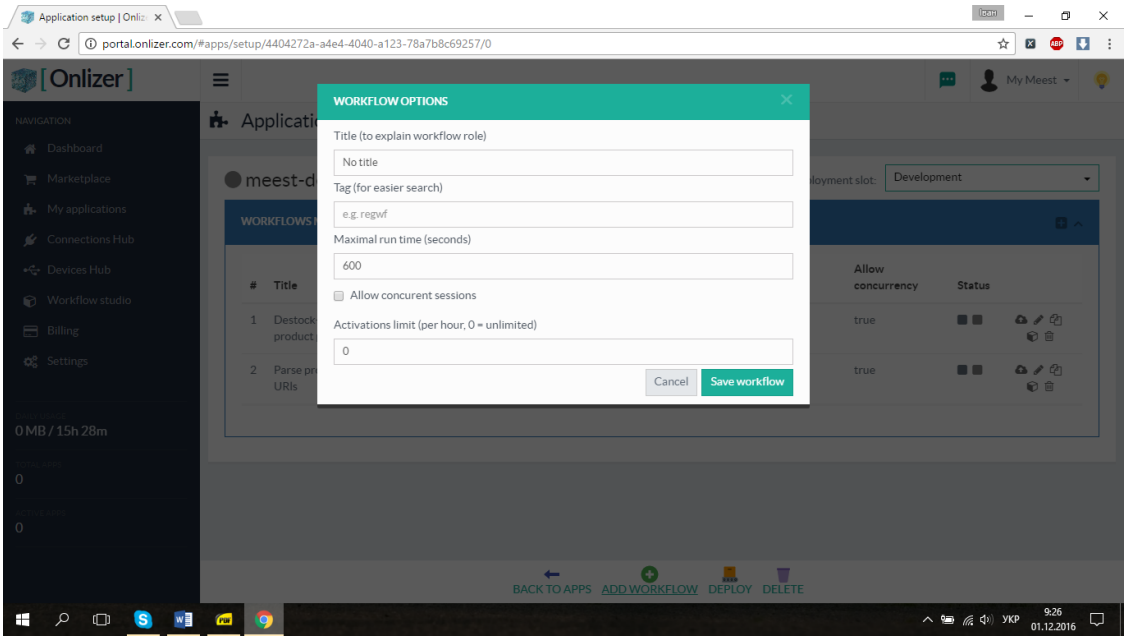

Рис. 3.12. Інтерфейс створення робочого потоку

Після створення робочого потоку з’являється можливість можливість до переходу у середовище-редактор, яке проілюстровано на рис. 3.13. У даному вікні за допоомогою технології drag&drop проектується алгоритм виконання операцій з готових компонентів, які реалізують бізнес-логіку деякого сценарію.

Перейдемо до експериментальної частини при роботі з платформою Onlizer. Як приклад, розглянемо робочий потік для одержання інформації про товари, які наявні в електронному магазині за адресою «http://www.destock-sport-et-mode.com».

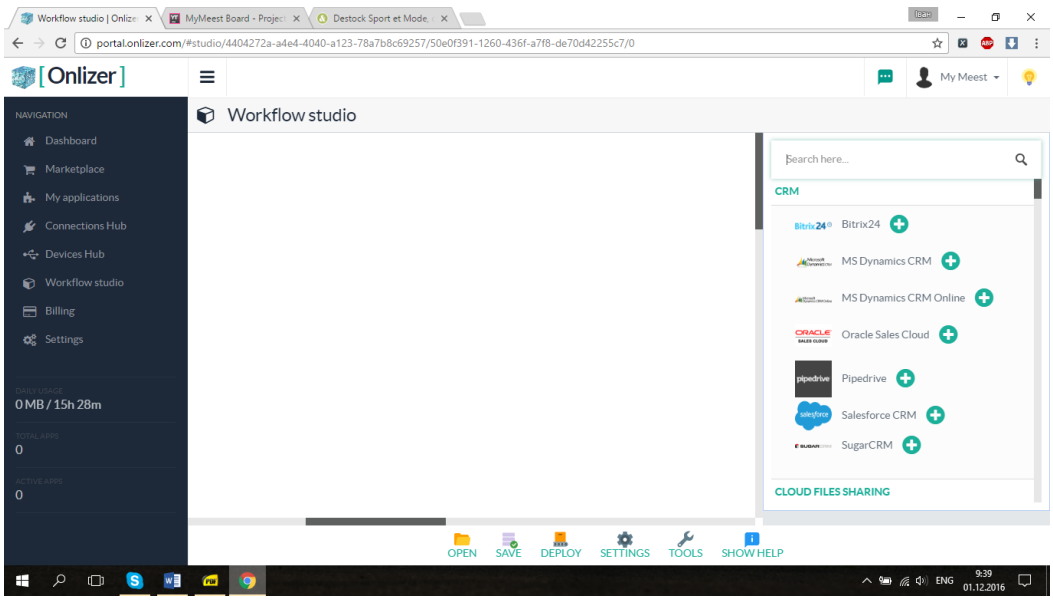

Рис. 3.13. Інтерфейс середовища редагування робочих потоків

Перший крок імплементації поставленої задачі полягає у створенні проекту та робочого потоку у відповідності до вище описаних кроків. Після успішного їх створення на панелі справа необхідно знайти та обрати конектор, який відповідає за кінцеву або початкову точку одержання результату («HTTP_Endpoint»), а для забезпечення знаходження товарів використовується спеціальний парсер – «HTML_Parser». Перший конектор забезпечує перехід до вказаної сторінки з товарами і передачі результатів виконання іншому конектору, який орієнтований на збір інформації про об’єкт.

Для зручності використання конектора передбачено кнопку «+» після натискання на яку, конектор відображається на робочому полі у редакторі робочого потоку. Зовнішній вигляд середовища побудови воркфловів з уже наявними в полі конекторами продемонстровано на рис. 3.14.

Коли на робоче поле додано усі необхідні конектори, які формують робочий потік, наступний крок полягає у налаштуванні відношень між ними. Для цього треба натиснути на кнопці у вигляді трикутника, який розміщений у правому верхньому куті відображення конектора і обрати пункт «Connection». Пункт формування зв’язку між конекторами поміщений у відповідному меню, як показано на рис. 3.16. Коли виконані попередні дії, то поява відношення між

конекторами відобразиться після того, коли буде обраний інший компонент з яким встановлюється зв’язок..

Для реалізації поставленої задачі необхідно виконати з’єднання «HTTP_Endpoint» з «HTML_Parser», а після цього «HTML_Parser» із ще одним «HTTP_Endpoint».

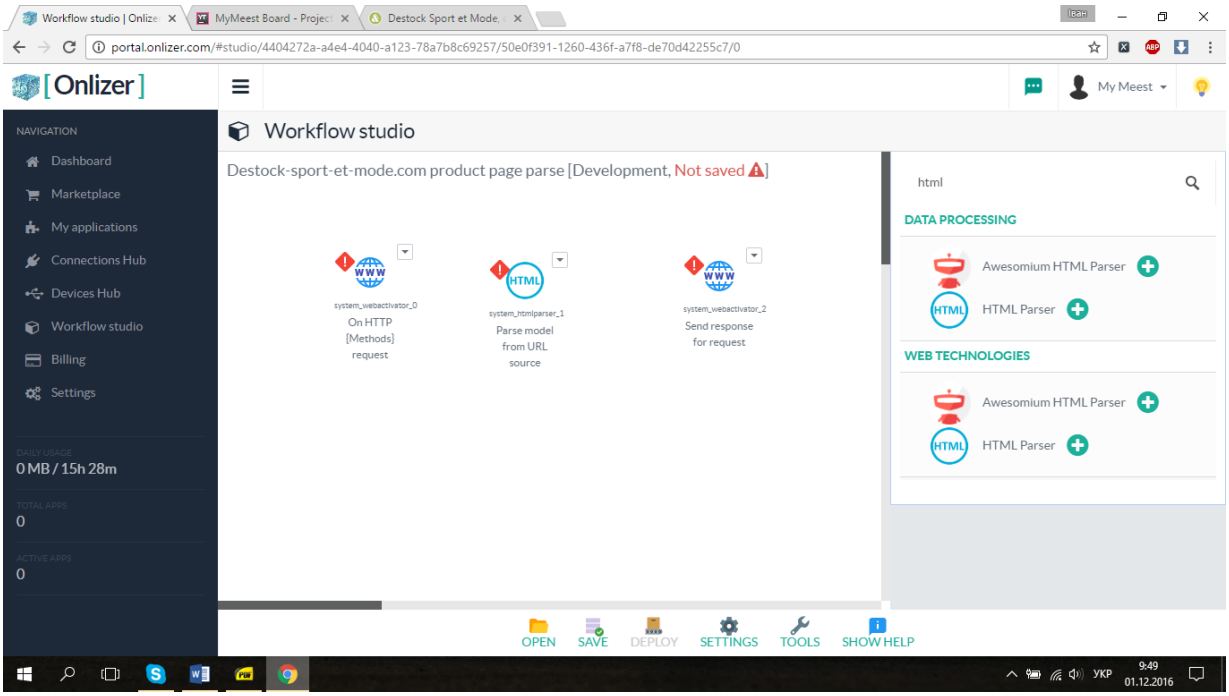

Рис. 3.14. Коненктори у середовищі проектування

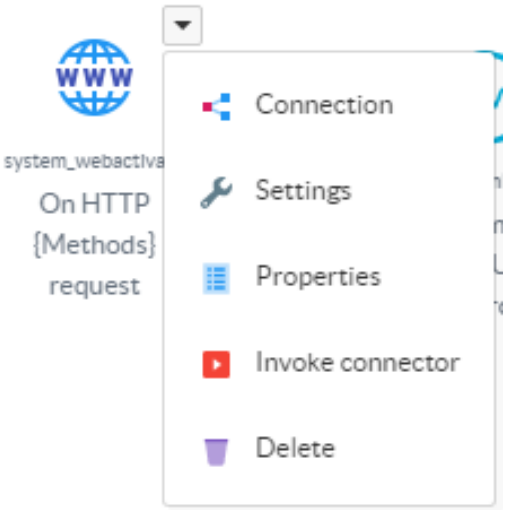

Рис. 3.15. Параметри конектора

Таким чином, з’єднавши усі компоненти утворюється робочий потік, однак для забезпечення логіки опрацювання даних потрібно налаштувати параметри конекторів. Для цього потрібно викликати контекстне меню та обрати «Settings».

У процесі початкового налаштування необхідно вказати метод, який буде опрацьовуватись конектором. Далі у вікні настройок, показаному на рис. 3.16 вказати необхідні параметри методу.

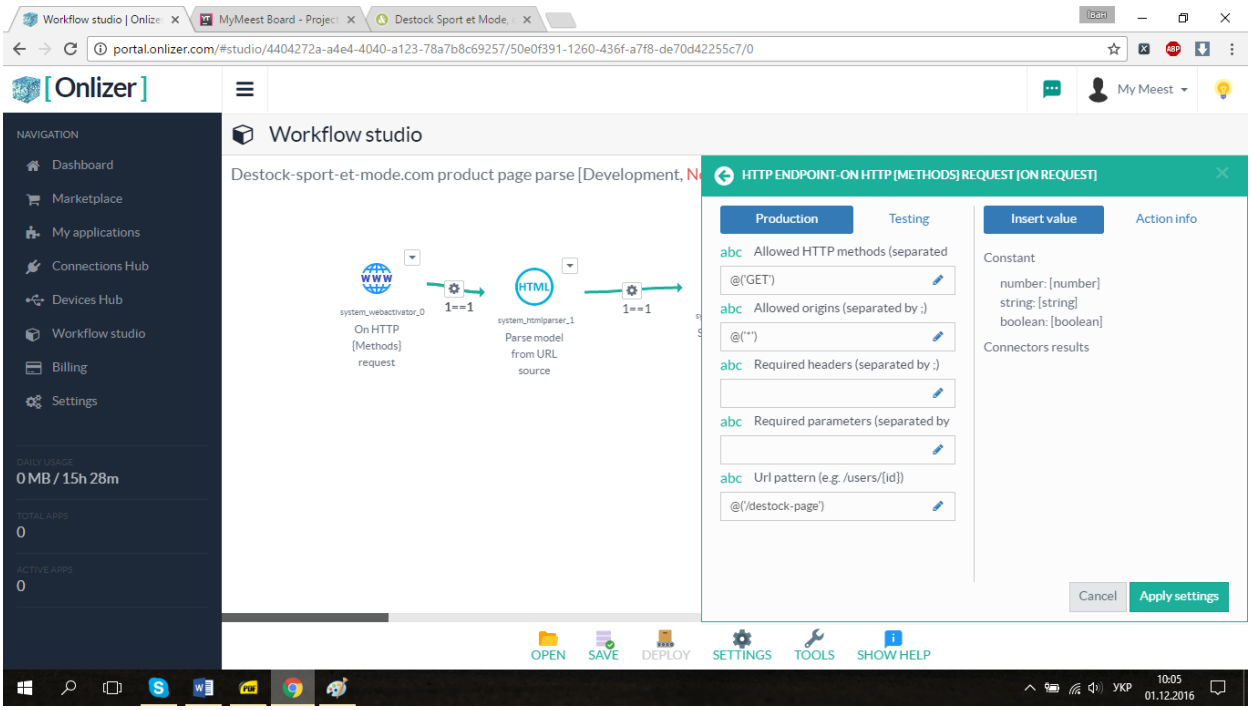

Рис. 3.16. Інтерфейс параметрів конектора

Після завершення налаштувань усіх компонентів і зв’язків між ними потрібно зберегти зміни робочого потоку, натиснувши відповідну кнопку. Після цього, його можна запустити на виконання, натиснувши на «Start» у вікні налаштування проекту.

Приклад результату виконання завдання щодо збору інформації проілюстровано на рис. 3.17.

Рис. 3.17. Результат виконання робочого потоку

На основі платформи Onlizer можна ефективно будувати проекти інтеграції з використанням прикладних програмних інтерфейсів і при цьому не володіючи особливими навиками програмування. Це дає змогу забезпечити інтеграцію різних програмних компонентів управління комп'ютерних систем і тим самим скоротити час розробки проекту.

3.3. Висновки до розділу

Основні результати даного розділу полягають в наступному:

1. Проаналізовано та імплементовано прикладний програмний інтерфейс за допомогою запропонованого методу, який підтримує і відповідає вимогам до проектування REST API, використовує архітектурний патерн OWIN, а також концепцію технології OData, що дало змогу на практиці забезпечити ефективність проектування веб-інтерфейсів.

2. Обґрунтовано використання системи Onlizer при проектуванні API на засадах компонентного підходу, що забезпечує можливість зменшення часу при його імплементації з врахуванням компонентів повторного використання.

3. Інструментами системи Onlizer реалізовано прикладний програмний інтерфейс пошуку і зберігання даних, що дає змогу знизити поріг входу при розробці комп'ютерних систем за рахунок імплементованої в платформі drag&drop процедури.

РОЗДІЛ 4
ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1. Охорона праці

Тема дипломної роботи магістра присвячена дослідженню методів і засобів імплементації прикладних програмних інтерфейсів при проектуванні розумних комп’ютерних систем. Проаналізуємо основні правила і норми, яких необхідно дотримуватись при експлуатації комп’ютерів та периферійних пристроїв під час проведення дослідження.

В загальному, поняття охорона праці в комп’ютерних системах являє собою дотримання всіх вимог і нормативів, що присутні в законодавчих актах про охорону праці. Закони цієї області спрямовані на якісну і безпечну експлуатацію робочих приладів і приміщень, дотримання санітарно-гігієнічних умов праці і захист від інших небезпечних чинників на підприємстві. Ці засоби є складовими дослідження математичного і програмного забезпечення автоматизованої системи підбору команди розробників комп’ютерних систем. В основних законодавчих актах про охорону праці приділяється велика увага поліпшенню умов праці в усіх галузях господарства, впровадженню сучасних засобів техніки безпеки і забезпечення санітарно-гігієнічних умов, що запобігають виробничому травматизму і професійним захворюванням.

Охорона життя і здоров’я людини є пріоритетним напрямком соціальної політики держави. В Україні прийнято закон прямої дії «Про охорону праці», який регламентує захист конституційного права працівників на безпечні умови праці. Законодавство України про охорону праці складається із загальних законів України та спеціальних законодавчих актів. Загальними законами України, що визначають основні положення з охорони праці є Конституція України, Закон України «Про охорону праці», Кодекс законів про працю (КЗпП), Закон України «Про загальнообов'язкове державне соціальне страхування від нещасного випадку на

виробництві та професійного захворювання, які спричинили втрату працездатності».

При виконанні досліджень кваліфікаційної роботи, які передбачали використання ПК, площа та об'єм для одного робочого місця оператора визначається згідно вимог НПАОП 0.00-7.15-18 «Вимоги щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями», зокрема площа повинна становити не менше 6,0 квадратних метрів, об'єм - не менше 20,0 кубічних метрів.

Згідно вимог охорони праці та державних санітарних правил, стіни, стеля та підлога приміщень, в яких розміщені ЕОМ, повинні бути виготовлені з матеріалів, дозволених для оформлення приміщень органами державного санітарно-епідеміологічного нагляду.

Заземлені конструкції, що знаходяться в приміщеннях, де розміщені робочі місця операторів (батареї опалення, водопровідні труби, кабелі із заземленим відкритим екраном), повинні бути надійно захищені діелектричними щитками та сітками з метою недопущення потрапляння працівника під напругу.

Організація робочого місця оператора повинна забезпечувати відповідність усіх елементів робочого місця та їх розташування ергономічним вимогам.

У приміщенні, де одночасно експлуатуються понад п'ять електронно-обчислювальних машин (ЕОМ), на помітному та доступному місці мають бути встановленні аварійні резервні вимикачі, які можуть повністю вимкнути електричне живлення приміщення, крім освітлення [26].

Дотримання правил значно знижує наслідки несприятливої дії на працівників шкідливих та небезпечних факторів, які супроводжують роботу з відео-дисплейними терміналами, зокрема можливість зорових, нервово-емоційних переживань, серцево-судинних захворювань. Виходячи з цього, роботодавець повинен забезпечити гігієнічні й ергономічні вимоги щодо організації робочих приміщень для експлуатації електронно-обчислювальних машин (ЕОМ) з ВДТ, робочого середовища, робочих місць з ЕОМ, режиму праці і відпочинку при роботі з ЕОМ тощо, які викладені у нормах НПАОП 0.00-7.15-18.

Відповідно до встановлених гігієнічно-санітарних вимог роботодавець зобов'язаний забезпечити в приміщеннях з ЕОМ оптимальні параметри виробничого середовища [26].

Для захисту від прямих сонячних променів, які створюють прямі та відбиті відблиски з поверхні екранів персонального комп’ютера і клавіатури повинні бути передбачені сонцезахисні пристрої, вікна повинні мати жалюзі або штори.

Основні задачі охорони праці при використанні комп'ютерної техніки:

– аналіз впливу факторів виробничого середовища на здоров'я і працездатність користувачів персональних комп’ютерів;

– вдосконалювання методів оцінки працездатності і стану здоров'я користувачів ПК;

– розробка і впровадження організаційно-технічних, гігієнічних і соціально-економічних заходів щодо раціоналізації виробничого середовища;

– розробка і впровадження профілактичних і оздоровчих заходів, що дозволяють зберігати здоров'я людини і підвищувати її працездатність;

– вдосконалення методик навчання користувачів ПК питанням охорони праці.

Вимогам нормативних актів з охорони праці мають відповідати:

– умови праці на кожному робочому місці;

– безпека технологічних процесів, машин, механізмів, обладнання й інших засобів виробництва;

– стан засобів колективного та індивідуального захисту;

– санітарно-побутові умови.

Отже, при дослідженні методів і засобів імплементації прикладних програмних інтерфейсів при проектуванні розумних комп’ютерних систем, проаналізовано та враховано необхідні вимоги щодо охорони праці при використанні електронно-обчислювальної техніки і забезпечено умови для зручної та ефективної роботи працівників.

4.2. Засоби захисту персоналу від уражень радіації

При виникненні надзвичайної ситуації, зокрема, при аварійному викиданні в атмосферу радіоактивних речовин можливі такі види радіоактивного впливу на населення:

– зовнішнє опромінення при проходженні радіоактивної хмари;

– внутрішнє опромінення при вдиханні радіоактивних аерозолів (інгаляційна небезпека);

– контактне опромінення внаслідок радіоактивного забруднення шкіри і одягу;

– зовнішнє опромінення, зумовлене радіоактивним забрудненням поверхні землі, будівель, споруд та ін.;

– внутрішнє опромінення при використанні забруднених продуктів харчування і води.

Розрахункові дані та результати прямих вимірювань рівня радіації і дози опромінення мають бути основою для вжиття заходів захисту населення від зовнішнього і внутрішнього опромінення, в тому числі й профілактичне застосування стабільного йоду.

Основою розробки заходів захисту населення в умовах радіоактивного забруднення при ядерній аварії є рекомендації Міжнародного агентства з атомної енергії (МАГАТЕ) 1988 р., а також норми радіаційної безпеки України (НРБУ—1997).

Враховуючи рівень радіації, а також прогноз можливих аварійних викидів радіоактивних речовин та метеорологічні дані, приймається рішення про проведення таких термінових і невідкладних заходів захисту в умовах ранньої фази радіаційної аварії:

– укриття населення;

– обмеження перебування населення на відкритій місцевості;

– евакуація у разі загрози здоров'ю;

– проведення йодової профілактики;

– тимчасова заборона вживання продуктів харчування і води із зони радіоактивного забруднення.

Крім цих заходів у період ранньої і пізньої фази проводяться довгострокові заходи:

– тимчасове відселення;

– евакуація — переселення на постійне місце проживання;

– обмеження вживання води і продуктів харчування забруднених радіоактивними речовинами;

– заходи захисту при виробництві продукції тваринництва, рослинництва і лісогосподарської діяльності;

– дезактивація території і будівель;

– інші заходи: гідрологічні, протиповіневі, обмеження лісокористування, полювання, рибної ловлі, перебування у полі при проведенні сільськогосподарських робіт.

Критерієм для прийняття рішення про заходи захисту населення на ранній і середніх фазах після аварії є дози зовнішнього і внутрішнього опромінення (табл. 5.1) з установленими двома рівнями радіаційного впливу — нижнім і верхнім — згідно з рекомендацією МАГАТЕ і НРБУ—1997.

При прогнозованому опроміненні, що не перевершує нижнього рівня, заходи, перелічені в табл. 5.1 не проводяться. Якщо прогнозоване опромінення перевищує нижній рівень, але не досягає верхнього рівня, то проведення вказаних заходів може бути відкладене.

Якщо прогнозоване опромінення досягає або перевищує верхній рівень, то обов'язково необхідно проводити заходи, наведені в табл. 4.1, навіть якщо вони пов'язані з порушенням нормальної життєдіяльності населення і об'єктів.

Таблиця 4.1

Критерії для прийняття рішень на ранній фазі розвитку аварії

Захисні заходи	Дозові критерії (прогнозована доза за перші 10 діб), мЗв			
	Все тіло		Окремі органи (легені, щитовидна залоза, шкіра)	
	Нижній рівень	Верхній рівень	Нижній рівень	Верхній рівень
Укриття, захист органів дихання і шкіри	5	50	50	500
Йодова профілактика:				
дорослі	—	—	50*	500*
діти, вагітні жінки	—	—	50*	250*
Евакуація:				
дорослі діти, вагітні жінки	50 10	500 50	500 200*	5000 500*

Радіаційний захист населення включає в себе:

– організацію безперервного контролю, виявлення та оцінку радіаційної та хімічної обстановки в районах розміщення радіаційно-небезпечних об'єктів;

– завчасне накопичення, підтримання в готовності і використання при необхідності засобів індивідуального захисту, приладів радіаційної розвідки і контролю;

– створення, виробництво та застосування уніфікованих засобів захисту, приладів і комплектів радіаційної розвідки і дозиметричного контролю;

– придбання населенням у встановленому порядку в особисте користування засобів індивідуального захисту та контролю за використанням їх за призначенням;

– своєчасне впровадження і застосування засобів і методів виявлення та оцінки масштабів і наслідків аварій на радіаційно-небезпечних об'єктах;

– створення і використання на радіаційно-небезпечних об'єктах систем (переважно автоматизованих) контролю обстановки і локальних систем оповіщення;

– розробку і застосування, за необхідності, режимів радіаційного захисту населення і функціонування об'єктів економіки та інфраструктури в умовах забрудненості (зараженості) місцевості;

– завчасне пристосування об'єктів комунально-побутового обслуговування і транспортних підприємств для проведення спеціальної обробки одягу, майна і транспорту, проведенням цієї обробки в умовах аварій;

– навчання населення використання засобів індивідуального захисту і правилам поведінки на забрудненій (зараженої) території.

До числа основних заходів щодо захисту населення від радіаційного впливу під час радіаційної аварії, належать:

– виявлення факту радіаційної аварії та оповіщення про неї;

– виявлення радіаційної обстановки в районі аварії;

– організація радіаційного контролю;

– встановлення та підтримання режиму радіаційної безпеки;

– проведення, при необхідності, на ранній стадії аварії йодної профілактики населення, персоналу аварійного об'єкта, учасників ліквідації наслідків аварії;

– забезпечення населення, персоналу аварійного об'єкта, учасників ліквідації наслідків аварії засобами індивідуального захисту та використання цих коштів;

– укриття населення, яке опинилося в зоні аварії, в притулках і укриттях, що забезпечують зниження рівня зовнішнього опромінення і захист органів дихання від проникнення в них радіонуклідів, які опинилися в атмосферному повітрі;

– санітарна обробка населення, персоналу аварійного об'єкта, учасників ліквідації наслідків аварії;

– дезактивація аварійного об'єкта, об'єктів виробничого, соціального, житлового призначення, території, сільськогосподарських угідь, транспорту, інших технічних засобів, засобів захисту, одягу, майна, продовольства і води;

– евакуація або відселення громадян із зон, в яких рівень забруднення перевищує допустимий для проживання населення.

Висновки

Радіація на сьогодні є чи не найнебезпечнішим фактором впливу не тільки на людину, але й на усі живі організми на планеті. Неконтрольовані ядерні реакції, ядерні війни та ряд інших можливих результатів людської діяльності, пов’язаних із радіацією негативно впливають на здоров’я людини та навколишнє середовище. Підтвердженням цього є аварія на Чорнобильській АЕС, наслідки якої відчувають до сьогодні як в Україні, так і за її межами. Дотримання рекомендацій щодо захисту населення від впливу радіації, які проаналізовано вище, дає змогу мінімізувати ризики, пов’язані із загибеллю великої кількості людей, а також зберегти їхнє здоров’я.

ВИСНОВКИ

Основні наукові та практичні результати полягають в наступному.

1. Проведено аналіз існуючих технологій проектування комп’ютерних систем і встановлено, що найбільш популярним шляхом імплементації складних систем є інтеграція та агрегація компонентів з використанням прикладних програмних інтерфейсів, що дає змогу підключати різні сервіси та ефективно керувати ними.

2. Проведено дослідження щодо процесу проектування архітектури комп’ютерних систем та її структурних елементів на рівні програмних інтерфейсів, встановлено принципи організації та імплементації архітектури SOA, що дало можливість визначити вузькі місця застосування API і сформувати пропозицію зниження впливу негативних факторів із застосуванням математичного апарату представлення процесу проектування прикладних програмних інтерфейсів.

3. Проаналізовано особливості застосування хмарних сервісів при проектуванні, реалізації і підтримці комп’ютерних систем, що дало змогу обґрунтувати використання підходу платформи як сервісу і забезпечити ефективність використання ресурсів, масштабованості та гнучкості налаштування параметрів системи.

4. Запропоновано метод побудови API на основі принципів компонентного підходу, що дає змогу врахувати вимоги архітектури SOA та представити в узагальненому, уніфікованому вигляді конструкцію прикладних програмних інтерфейсів.

5. Запропоновано математичне представлення структури програмного компоненту комп’ютерної системи, що використовує парадигму об’єктно-орієнтованого підходу та описується за допомогою елементів теорії множин, що дало змогу використовувати їх у процесі імплементації API за допомогою мікросервісів.

6. Забезпечено формалізоване представлення процесу побудови прикладних програмних інтерфейсів, що дало змогу виявити і забезпечити оптимальність їхнього створення на основі хмарних сервісів.

7. Обґрунтовано метод перевірки взаємозв'язків між функціональними сервісами комп'ютерних систем у хмарному середовищі із застосуванням методів тестування API, що дало змогу мінімізувати зчепленість сервісів і максимізувати зв'язність всередині компонентів.

8. Проаналізовано та імплементовано прикладний програмний інтерфейс за допомогою запропонованого методу, який підтримує і відповідає вимогам до проектування REST API, використовує архітектурний патерн OWIN, а також концепцію технології OData, що дало змогу на практиці забезпечити ефективність проектування веб-інтерфейсів.

9. Обґрунтовано використання системи Onlizer при проектуванні API на засадах компонентного підходу, що забезпечує можливість зменшення часу при його імплементації з врахуванням компонентів повторного використання.

10. Інструментами системи Onlizer реалізовано прикладний програмний інтерфейс пошуку і зберігання даних, що дає змогу знизити поріг входу при розробці комп'ютерних систем за рахунок імплементованої в платформі drag&drop процедури.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Технології проектування програмного забезпечення URL: http://www.iasa.com.ua/studentam/study-materials-ua/testing/d456dkovska-m-v-lek c456ya-1-zhitt454v456-cikli-rozrobki-programnogo-zabezpechennya (дата звернення 13.11.2022 р).

2. Моделі життєвого циклу у сучасних методологіях розробки ПЗ URL:: http://kpi.km.ua/teachers/radelchukgi.html?id=130 (дата звернення 15.11.2022 р).

3. Брауде Е. Технология разработки программного обеспечения. К. : Изд-во "Питер", 2014. 655 с.

4. Буч Г. UML: специальный справочник. 2012. 656 с.

5. Введение в Web API. 2015. URL: https://metanit.com/sharp/mvc/12.1.php (дата звернення 25.11.2022 р.).

6. Вигерс К. Разработка требований к программному обеспечению. 2014. 576 с.

7. Грабовский М. Современные технологии и стандарты разработки программного обеспечения. Корпоративные системы. 2010. с.71 – 76.

8. Бабак В.П. Основи теорії ймовірностей та математичної статистики: Навчальний посібник. К.: КВІШ, 2013. 432 с.

9. Ларман К. Применение UML и шаблонов. К.: Вильямс, 2011. 496 с.

10. Палермо Д. ASP.NET MVC 4 в действии. URL: https://smarly.net/asp-net-mvc-4-in-action/mastering-asp-net-mvc/asp-net-web-api/what-is-web-api. (дата звернення 28.11.2022 р.).

11. Підходи до проектування REST FULL API. URL: http://it-ua.info/news/2016/02/17/pdhodi-do-proektuvannya-restful-api.html. (дата звернення 28.11.2022 р.).

12. Створення API за допомогою Ruby on Rails та GraphQL. URL: https://codeguida.com/post/826. (дата звернення 28.11.2022 р.).

13. Фаулер М. Архитектура корпоративных программных приложений. К.: Издательский дом "Вильямс". 2016. 544 с.

14. Яцишин В.В., Шаблій Н.Р., Денисов Д.В. Призначення і доцільність використання API Gateway у комп'ютерних системах. Матеріали X науково-технічної конференції Тернопільського національного технічного університету імені Івана Пулюя «Інформаційні моделі, системи та технології» (8-9 грудня 2022 року). Тернопіль: ТНТУ. 2022. С. ХХХ.

15. Лупенко А.М., Куліков С.О., Денисов Д.В. Класифікація та особливості застосування прикладних програмних інтерфейсів при реалізації комп'ютерних систем. Матеріали X науково-технічної конференції Тернопільського національного технічного університету імені Івана Пулюя «Інформаційні моделі, системи та технології» (8-9 грудня 2022 року). Тернопіль: ТНТУ. 2022. С. ХХХ.

16. Hull E. Requirements Engineerig/ E. Hull, Ken Jackson, Jeremy Dick. // Springer Science+Business Media. 2015. 240 c.

17. IEEE Std 830-1993, IEEE Recommended Practice for Software Requirements Specifications (ANSI).

18. ISO/IEC 12207:2008 Systems and software engineering – Software life cycle processes.

19. Sommerville I. Deriving Information Requirements from Responsibility Models / I. Sommerville, R. Lock, T. Storer, J. Dobson// Proc. CAiSE 2009. 21st International Conference on Advanced Information Systems Engineering, Amsterdam, June 2013, pp.515- 529.

20. Musa J.D. Operational Profiles in Software Reliability Engineering // IEEE Software.-V.10.- N.2.- 2003.- P. 14 - 32.

21. Методы оценки эффективности информационных систем URL: https://sites.google.com/site/isefficiency/metody-ocenki-effektivnosti-informacionnyh-sistem (дата звер-нення 05.12.2022 р.).

22. Жидецький В.Ц. Охорона праці користувачів комп´ютерів. Львів: Афіша, 2011. 176 с.

23. Желібо Е.Н. Безпека життєдіяльності: Навчальний посібник/ За редакцією Е.П. Желібо, В.М. Пічі. – Київ: «Караве-ла», Львів: «Новий світ - 2000», 2011. 320с.

Додаток А

Текст наукових публікацій дипломної роботи магістра

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ТЕРНОПІЛЬСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ІМЕНІ ІВАНА ПУЛЮЯ

М А Т Е Р І А Л И

X НАУКОВО-ТЕХНІЧНОЇ КОНФЕРЕНЦІЇ

«ІНФОРМАЦІЙНІ МОДЕЛІ, СИСТЕМИ ТА ТЕХНОЛОГІЇ»

7–8 грудня 2022 року

ТЕРНОПІЛЬ
2022

УДК 004.415.25
А. Лупенко, С. Куліков, Д. Денисов
(Тернопільський національний технічний університет імені Івана Пулюя, Україна)

КЛАСИФІКАЦІЯ ТА ОСОБЛИВОСТІ ЗАСТОСУВАННЯ ПРИКЛАДНИХ ПРОГРАМНИХ ІНТЕРФЕЙСІВ ПРИ РЕАЛІЗАЦІЇ КОМП'ЮТЕРНИХ СИСТЕМ

UDC 004.415.25
A. Lupenko, S. Kulikov, D. Denysov

CLASSIFICATION AND FEATURES OF THE APPLICATION PROGRAMMING INTERFACES IN COMPUTER SYSTEMS IMPLEMENTATION

API (Application Programming Interface) – набір правил і механізмів, за допомогою яких один додаток або компонент взаємодіє з іншими.

Властивості, якими повинен володіти API:

– простота використання і підтримки – хороший API просто використовувати і підтримувати;

– хороша конверсія в середовищі розробників – стимулює кількість клієнтів і користувачів сервісу;

– популярність сервісу – чим більше користувачів API, тим вища популярність сервісу;

– ізоляція компонентів – чим краща структура API, тим краще ізоляція компонентів;

– зручність використання API – програмний інтерфейс є своєрідним інтерфейсом для розробників на який вони звертають увагу в першу чергу при зустрічі з продуктом. Якщо API не забезпечує попередніх властивостей, то технічний експерт не буде рекомендувати компаніям використовувати такий продукт, купуючи щось стороннє.

Класифікація API за способом реалізації:

– Web service APIs: XML-RPC and JSON-RPC, SOAP, REST, WebSockets APIs, Library-based APIs,

– Java Script;

– Class-based APIs: C# API, Java.

Види API за категоріями застосування:

– OS function and routines – Access to file system; Access to user interface;

– Object remoting APIs – CORBA, .Net Remoting;

– Hardware APIs: Video acceleration (OpenCL...), Hard disk drives; PCI bus та ін.

RPC (remote procedure call – віддалений виклик процедур») – поняття, що поєднує давні, середні і сучасні протоколи, які дозволяють викликати метод в іншому додатку.

XML-RPC – протокол, що з'явився в 1998 р. незабаром після появи XML. Спочатку він підтримувався Microsoft до переходу на SOAP, тому .Net Framework не містить класів для підтримки цього протоколу. Незважаючи на це, XML-RPC продовжує існувати в різних мовах (особливо в PHP).

SOAP з'явився в 1998 р. стараннями Microsoft. Він був анонсований як революція в світі. Не можна сказати, що все пішло за планом Microsoft: була величезна кількість критики через складність протоколу. У той же час, були і ті, хто вважав SOAP справжнім проривом. Протокол продовжував розвиватися десятками нових і нових специфікацій, поки в 2003 р. W3C не затвердила в якості рекомендації SOAP 1.2.

УДК 004.415.25
В. Яцишин, Н. Шаблій, Д. Денисов
(Тернопільський національний технічний університет імені Івана Пулюя, Україна)

ПРИЗНАЧЕННЯ І ДОЦІЛЬНІСТЬ ВИКОРИСТАННЯ API GATEWAY У КОМП'ЮТЕРНИХ СИСТЕМАХ

UDC 004.415.25
V. Yatsyshyn, N. Shabliy, D. Denysov

PURPOSE AND FEASIBILITY OF USING API GATEWAY IN COMPUTER SYSTEMS

Взаємодія сервісів між собою можлива лише через мережу. Розробляючи програму за таким підходом, варто виходити з того, що компоненти можуть не працювати і тому цей момент потрібно враховувати при розробці нових компонентів. Також, варто передбачити системи автоматичного розгортання та моніторингу сервісів, оскільки в системі їх може бути дуже багато і фізично прослідкувати за роботою кожного є, часто, неможливим.

Для виконання взаємодії компонентів між собою використовується API Gateway (рис. 1).

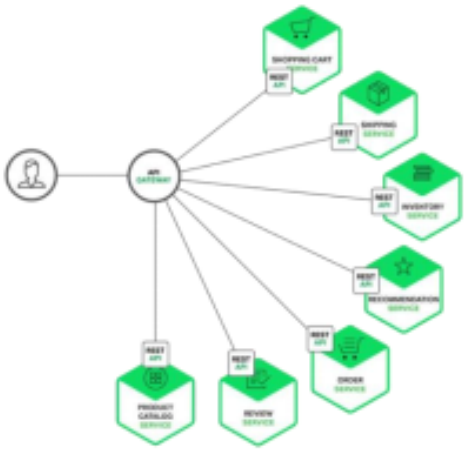

Рисунок 1. Використання API Gateway

Gateway прикладних програмних інтерфейсів доцільно використовувати у випадку проектування мікросервісної архітектури. Функції, які він буде виконувати, полягають у тому, що за наявності в backend декількох сервісів, більш ефективно застосовувати для доступу до них елементарний компонент. Задача API Gateway – забезпечувати збір бізнес-викликів до відповідних сервісів, що дозволить підвищити ефективність процесу маршрутизації повідомлень. Ще однією функцією API Gateway є забезпечення представлення даних для користувача у тому вигляді, в якому він цього потребує і який є наперед визначеним. Для прикладу, за наявності двох версій додатку (для веб і мобільного використання) доцільно використовувати 2 API Gateway, що забезпечать збір даних від однакових сервісів, але формувати їх вигляд будуть по-різному. Вимогою при імплементації API Gateway є задоволення простоти функціональності та мінімальної бізнес-логіки для уникнення надмірного дубляжу та спрощення процесу підтримки. Простота зводиться до того, що API Gateway не виконує жодних функцій окрім передачі даних.