КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра теорії та технології програмування

Кваліфікаційна робота
на здобуття ступеня магістра

з освітньо-наукової програми «Інформатика»
за спеціальністю 122 Комп'ютерні науки
на тему:

МЕТОДИ АТЕСТАЦІЇ РОЗПОДІЛЕНИХ СИСТЕМ НА БАЗІ СЕРВІСНО-ОРІЄНТОВАНОЇ АРХІТЕКТУРИ

Виконала студентка 2-го курсу
Анастасія ХАРЧЕНКО

(підпис)

Науковий керівник:
доцент, кандидат технічних наук
Олексій ТКАЧЕНКО

(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студентка
(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри теорії та технології програмування
«08» травня 2023 р.,
протокол № 16

Завідувач кафедри
Микола НІКІТЧЕНКО
(підпис)

Київ-2023

РЕФЕРАТ

Обсяг роботи 72 сторінки, 34 ілюстрації, 8 таблиць, 17 джерел посилань.

КЛЮЧОВІ СЛОВА: СЕРВІС, СИСТЕМА, API, JAVA, SWAGGER UI, UML, ТЕСТУВАННЯ, АВТОМАТИЗАЦІЯ, CYPRESS, ІНТЕРНЕТ-МАГАЗИН.

У ролі об’єкта дослідження виступають інформаційні системи на базі сервісно-орієнтованої архітектури. А у ролі предмета – методи атестації систем із сервісно-орієнтованою архітектурою.

Мета роботи полягає у тому, щоб застосувати різні види атестації для розробленої системи із сервісно-орієнтованою архітектурою. Для досягнення поставленої мети визначено такі завдання:

- Дослідити сервісно-орієнтовану архітектуру
- Дослідити методи атестації, що існують
- Розробити серверну частину інформаційної системи «Інтернет-магазин» з використанням SOA
- Спроєктувати множину тестів різних типів для розробленої системи та провести тестування
- Зробити порівняльний аналіз методів

Методами дослідження є загальнонаукові (аналіз, синтез, порівняння), методи тестування програмного забезпечення. Засобами розробки сервісно-орієнтованої системи є мова програмування Java, як рушійна машина для API був використаний фреймворк Spring boot, задля UI оформлення API додано Swagger-UI, база даних PostgreSQL, середовище для створення тестів Visual Studio Code та інструментом для тестування було обрано Cypress.

Результати роботи: досліджено архітектуру сервісно-орієнтованих систем та методи атестації, розроблено серверну частину інформаційної системи «Інтернет-магазин», спроєктовано множину тестів та проведено тестування, зроблено порівняльний аналіз застосованих методів тестування.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

SOA – Service-oriented architecture, сервісно-орієнтована архітектура;

JAVA – мова програмування;

API – Application Programming Interface, інтерфейс прикладного програмування;

REST – REpresentational State Transfer, «передача репрезентативного стану»

JSON – JavaScript Object Notation, текстовий формат обміну даними між комп’ютерами;

SOAP – Simple Object Access Protocol;

HTTP – HyperText Transfer Protocol, протокол передачі даних;

СКБД – Система керування базами даних

БД – База даних;;

SQL – Structured Query Language;

SDLC – Software Development Lifecycle, життєвий цикл програмного забезпечення;

UML – Unified Modeling Language, уніфікована мова моделювання, використовується в парадигмі об’єктно-орієнтованого програмування;

ВСТУП

Оцінка сучасного стану об'єкта розробки. Розробка програмного забезпечення стала невід'ємною частиною сучасного суспільства. Зі зростанням попиту на програмне забезпечення важливо, щоб розробники гарантували, що їхні продукти мають найвищу якість. Одним зі способів досягти цього є атестація програмного забезпечення.

Часто через наявність критичних помилок, які не були виявлені на початку розробки приводить до того, що програми закінчують своє існування. Можливі ситуації, коли такі помилки коштують набагато дорожче ніж програмне забезпечення в цілому. Ще гірше, коли помилки знаходяться вже у використовуваних замовниками, програмах, а якщо вони ще й пов'язані з безпекою підприємства, то збитки можуть бути шалені. Таке неякісне програмне забезпечення не буде конкурентно спроможним на ринку і не принесе користі та очікуваного прибутку. Тож, дуже важливо забезпечити високий рівень якості ПЗ та спроможність програми працювати швидко, навіть при високому навантаженні на систему.

Актуальність роботи та підстави для її виконання. Для того, щоб визначити якість програмного продукту зазвичай використовують набір властивостей, що допомагають показати на скільки готове рішення відповідає вимогам замовників. Але це не означає, що атестація системи завершується на цьому етапі. Однак, найефективніших методів для забезпечення якості продукту на разі немає. Тому при підготовці будь-якого програмного забезпечення важливо знайти та застосовувати методи та підходи, які приноситимуть користь для відображення якості, вартості та часу розробки. Але перед тим, як почати верифікувати програмне забезпечення, потрібно детально розглянути властивості та побудову системи. На сьогодні однією з найпопулярніших моделей розподілених систем є сервісно-орієнтована архітектура.

Сервісно-орієнтована архітектура (SOA) стала важливою складовою сучасної бізнес-практики. Це стиль архітектури програмного забезпечення, який дозволяє організаціям створювати програми, поєднуючи слабозв'язані та незалежно розгорнуті служби.

SOA має важливе значення у сучасному бізнесі, оскільки дозволяє організаціям досягти гнучкості та інновацій. Розбиваючи додатки на менші багаторазові компоненти, організації можуть швидко реагувати на зміну ринкових умов і вимог споживачів. SOA також дозволяє організаціям інтегрувати різні системи та програми, покращуючи обмін даними та співпрацю між відділами. [1]

Мета й завдання роботи.

Мета роботи полягає у тому, щоб визначити найкращу стратегію для атестації систем із сервісно-орієнтованою архітектурою.

Для досягнення поставленої мети визначено такі завдання:

- Дослідити сервісно-орієнтовану архітектуру
- Дослідити методи атестації, що існують
- Розробити серверну частину інформаційної системи «Інтернет-магазин» з використанням SOA
- Спроєктувати множину тестів різних типів для розробленої системи та провести тестування
- Зробити порівняльний аналіз методів

Об'єкт, методи й засоби дослідження або розроблення. У ролі об'єкта дослідження виступають інформаційні системи на базі сервісно-орієнтованої архітектури. А у ролі предмета – методи атестації систем із сервісно-орієнтованою архітектурою. Методами дослідження є загальнонаукові (аналіз, синтез, порівняння), методи тестування. Засобами розробки сервісно-орієнтованої системи є мова програмування Java, як рушійна машина для API був використаний фреймворк Spring boot, задля UI оформлення API додано Swagger-UI, база даних PostgreSQL, середовище для створення тестів Visual Studio Code та інструментом для тестування було обрано Cypress.

Можливі сфери використання. Робота може бути використана, як приклад можливого атестування сервісно-орієнтованих систем та виконаний порівняльний аналіз застосованих методів тестування може допомогти фахівцям у розробці сценаріїв тестування.

Апробація роботи та публікації з теми роботи. За результатами роботи були опубліковані тези з конференції «Шевченківська весна» з 14.04.2023 року.

РОЗДІЛ 1 ХАРАКТЕРИСТИКА ТА АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Розподілені системи та специфіка перевірки їх атестації

Різниця між монолітною та розподіленою архітектурою.

Монолітна архітектура – модель, що зайшла у традицію програмного забезпечення, яка містить у собі єдиний модуль, що працює автономно та незалежно від інших програм. Архітектуру моноліту зручно використовувати на перших етапах розробки системи, щоб не ускладнювати процес розгортання. Але така система складно піддається масштабуванню і у разі розширення системи потребує повного рефакторигу. Тому все далі стає популярними системами із розподіленою архітектурою. [2]

Розподілені системи – це система з кількома компонентами, розташованими на різних машинах, які обмінюються даними та координують дії, щоб кінцевий користувач виглядав як єдина узгоджена система. [3]

Атестація розподілених систем має де які особливості. Потрібно враховувати як працюють всі частини системи як окремо, так і в цілому. Тобто в системах з такою архітектурою додатково треба перевіряти окремо серверну та клієнтську частини, їх взаємодію, роботу в мережі та всі аспекти, пов’язані із цілісністю даних, стійкістю тощо.

В даній роботі під словом атестація розглядається всі можливі методи перевірки систем:

- Верифікація та валідація
- Тестування
- Доведення програм

Основний фокус зосереджений на усіх видах тестування систем із розподіленою архітектурою, а саме сервісно-орієнтованою архітектурою.

1.2 Огляд методів атестації програмного забезпечення

Основні напрями перевірки правильності прорамного забезпечення – це формальні специфікації, верифікація, методи доведення та тестування. [4]

Під атестацією програмного забезпечення будемо розуміти сукупність процесів перевірки системи, метою яких є отримання свідчень того, що розробка досягла поставлених цілей.

До основних методів атестації програмного забезпечення віднесемо:

Верифікація (Verification) – перевірка на відповідність специфікаціям (вимогам), оцінка правильності написання програмного коду.

Доведення програм відбувається з використанням тверджень, що складаються у формальній мові і допомагають перевіряти правильность програми в заданих для неї точках. Набір тверджень, перед- і постумов застосовується для перевірки отриманого результату у одній із точок програми. Якщо твердження відповідає скінченому оператору програми, то за допомогою постумови робиться висновок про повну або часткову правильність роботи програм. [4]

Валідація (Validation) – перевірка на придатність використання в конкретній предметній області, на відповідність очікуванням. У деяких випадках верифікація та валідація вважаються синонімами.

Тестування (Testing) - це процес визначення, чи виконує програмне забезпечення очікувані функції та вимоги. Цей метод охоплює ручне та автоматизоване тестування. Виходячи зі стандарту ANSI/IEEE 1059, тестування визначається як - процес аналізування елементів програмного забезпечення для виявлення відмінностей між чинними та необхідними умовами (тобто дефекти/помилки/баги) і для оцінки функцій елемента програмного забезпечення.

Інспектування, огляди (Review) – процес перевірки документації різного типу (технічного завдання, вимог, моделей, програмного коду, інструкцій користувача тощо).

1.2.1 Опис вимог та якість програмного забезпечення

Специфікація програми - це документ, який описує функціональність, поведінку та вимоги до програмного забезпечення. Основна ціль специфікації це не тільки описання як програма повинна працювати, а й виконання вимог для відображення коректності та ефективності програми.

Вимога – це те, що продукт повинен виконувати, або якість, яку він повинен мати.

Специфікація вимог – набір усіх вимог, які мають бути сформовані та висунуті до проєктування та тестування продукту. Специфікація також повинна містити відповідну інформацію необхідну для розробки, тестування або обслуговування програми. [5]

Формальна специфікація є повним описом моделі системи та вимог до її поведінки, який виражений у формальних методах. Формальний метод розробки ПЗ використовує математичні засади, такі як моделювання, математична логіка, теорія множин, теорія скінченних автоматів, алгебра та інші, для створення формальної специфікації, верифікації та аналізу вимог до ПЗ та предметних областей. Ці методи та інструментальні засоби іноді називають системою формальних міркувань.

Як правило, крім системи формальних міркувань формальні методи включають стандартизовані мови (мови специфікацій, формальні нотації). Приклади формальних методів: CSP, OBJ, CCS, VDM, B-метод, Z-метод, RAISE.
Формальна мова (мова специфікацій, формальна нотація) – це мова з точно визначеним синтаксисом та семантикою.

Мови формальних специфікацій використовуються для написання специфікації, тобто для опису властивостей певної предметної області. Це, наприклад, мови Z, B, CLEAR, LARCH.

Формальна розробка програм - це метод, що передбачає систематичне перетворення специфікації програми на виконуваний код за допомогою

формальних методів. Ці методи базуються на використанні мови специфікацій та формальних міркувань для створення програмного забезпечення.

Застосування формальних міркувань для аналізу моделі та системи ґрунтується на математичних засадах та мовах формальної специфікації. Останні використовують математичні концепції, такі як числення предикатів, алгебра та теорія скінченних автоматів, щоб створити модель системи. Далі ці інструментальні засоби застосовуються для аналізу самої системи.

При формальній розробці програми проводяться кроки, які включають аналіз системи з метою перевірки її властивостей. Для цього використовуються засоби формальних міркувань, які застосовуються за допомогою двох основних підходів.

Перший підхід - це автоматичне доведення теорем, яке полягає у доведенні логічних тверджень про властивості системи за допомогою програмного забезпечення, що базується на математичній логіці.

Другий підхід - це перевірка моделі, що полягає у перевірці, чи є побудована структура моделлю заданої предметної області. Для цього використовуються спеціальні програмні засоби. [6]

Концепція формальних методів вводить інструменти для математичного опису системи (або частин системи) у специфікації та доведення того, що отримана програма відповідає вимогам, описаним у специфікації. Формальна специфікація є точною, і немає ризику неправильного тлумачення. Крім того, якщо є доказ того, що реалізація відповідає специфікації, тоді можна бути впевненим, що програмісти реалізували те, що описано в специфікації.

На практиці неможливо повністю гарантувати, що кінцева реалізація безпомилкова, оскільки використовуваний формальний метод може мати дефекти або може бути якась помилка в доказі. Проте, ширше використання формальних методів і інструментів призведе до кращих і надійніших методів і інструментів.
Підсумовуючи: за допомогою формальних методів у розробці системи можна виявити помилки раніше, а деякі класи помилок можна майже усунути. [7]

Проблема інтегрування формальної специфікації при розробці програм

Створити програмне забезпечення з формальної специфікації непросто, навіть якщо специфікація є виконуваною: конструкції, які використовуються у формальній специфікації, можуть погано перекладатися на цільову мову, що призводить або до неефективної реалізації, або до значної кількості виправлень для оптимізації дизайну коду. [8]

Використання формальної специфікації є одним із найперспективних напрямів в галузі програмування.

1.2.2 Верифікація та валідація

Верифікація програмного забезпечення – це перевірення того, що програмне забезпечення виконує поставлені цілі без помилок. Це також включає процес перевірки правильності та коректності розробленого програмного забезпечення. Він перевіряє, чи відповідає розроблений продукт поставленим вимогам. Тобто верифікація – це про статичне тестування. Якщо ставити питання, що саме робить верифікація, то воно звучатиме так:

«Перевірка означає, чи правильно ми створюємо продукт?»

Порівняння основних методів верифікації

Основні методи верифікації зображені на рисунку 1.1

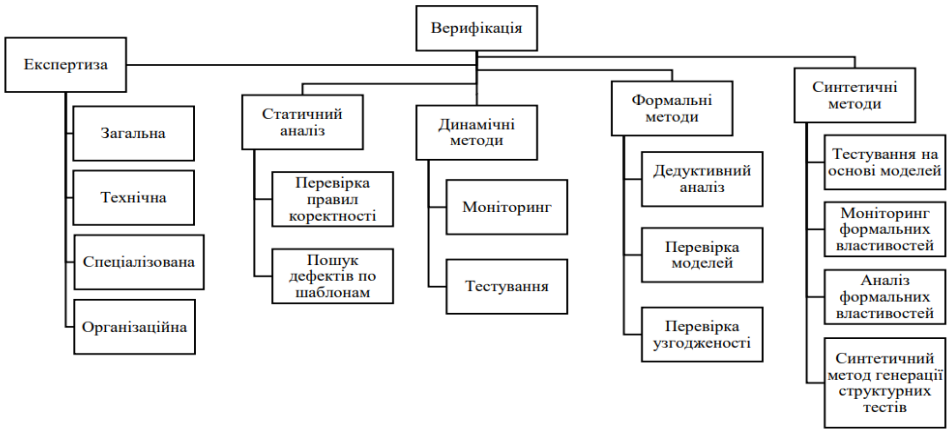

Рисунок 1.1 – Класифікація основних методів верифікації ПЗ [9]

Методи верифікації складаються з:

- експертизи,
- статичного аналізу,
- динамічного аналізу,
- формальних та синтетичних методів.

Експертиза – містить усі можливі методи верифікації, в яких оцінювання програмного забезпечення проводиться людьми, що виконують верифікацію. Перевага такого методу полягає у тому, що він допомагає знайти велику кількість помилок у програмі.

Статичний аналіз - це метод аналізу програмного коду без його фактичного запуску. Статичний аналіз має два напрямки: перевірка відповідності всіх формалізованих правил побудови артефактів, та знаходження типових помилок та багів, використовуючи визначені шаблони. Зазвичай інструменти для проведення статичного аналізу використовують одразу разом ці типи перевірок. Статичний аналіз є одним з найпопулярніших для застосування методів для проведення верифікації.

Правила для перевірки коректності коду та шаблони для знаходження типових помилок, перевірені на практиці, можуть бути інтегровані в середовища розробки.

Виявлені переваги статичного аналізу:

– Можливість автоматичного аналізу багатьох шляхів запуску одночасно.

– Виявлення погрішностей, що проявляються тільки на одиничних виконаннях або на незвичних вхідних даних.

– Можливість аналізування неповного набору вхідних файлів.

Виявлені негативні риси даного методу:

– Можлива велика кількість хибних спрацювань.

– Після отримання результатів роботи – необхідна додаткова людська перевірка, що вимагає додаткових часових та матеріальних ресурсів.

Для проведення верифікації використовуючи динамічні методи потрібно порівняти результати роботи програми із завчасно поставленими до неї вимогами. Існують два основних методи динамічної верифікації програмного забезпечення: моніторинг та тестування. Моніторинг включає спостереження, запис та оцінку результатів роботи програми в процесі її звичайного використання. Тестування передбачає виконання заздалегідь підготовлених сценаріїв з метою виявлення помилок в програмі. Перевагою даного методу є висока точність виявлення помилок.

А негативні риси – необхідність мати набір вхідних даних та середовище виконання, а також високі вимоги до ресурсів.

Формальні методи верифікації.

Сенс формальних методів верифікації полягає у тому, щоб провести пошук помилок на математичній моделі, без використання фізичної реалізації, що є іноді зручно та економічно.

Для того, щоб провести аналіз формальних моделей необхідно застосувати специфічні техніки, відомі як дедуктивний аналіз, перевірка узгодженості, перевірка моделей.

З недоліків цього методу: для того, щоб побудувати такі моделі потрібно зважати на коректність моделі програмного забезпечення. Адже, тільки після правильної побудови даної моделі можна правильно проаналізувати її властивості.

Проте, щоб застосувати даний метод, потрібні глибокі знання математичної логіки та алгебри та мати набір навичок для роботи з цим апаратом.

Завершальним методом верифікації є синтетичний метод. Як правило, синтетичний метод поєднує у собі підході деяких типів – наприклад, статичний метод у поєднанні з формальним аналізом та тестуванням. [9]

Валідація

Валідація — це процес перевірки відповідності програмного продукту до вимог, іншими словами, вимог високого рівня. Цей процес перевіряє чи те що, ми

розробляємо, є правильним продуктом. Це перевірка фактичного та очікуваного продукту.

Валідація – це про динамічне тестування. Якщо ставити питання, що саме робить валідація, то воно звучатиме так:

«Перевірка означає чи створюємо ми правильний продукт?»

Розглянемо невеликий порівняльний аналіз цих понять у таблиці 1.1

Таблиця 1.1 - Порівняльний аналіз понять верифікація та валідація

Верифікація	Валідація
Містить перевірку коду, дизайну, документів	Тестування фактичного, готового, розробленого продукту
Під час верифікації використовуються покрокові інструкції та інспекції	Під час валідації перевіряється чи відповідає програмне забезпечення очікуванням клієнта
Не містить перевірку коду	Включає перевірку коду
Більше опирається на специфікацію	Більше опирається на забезпечення відповідності вимогам та очікуванням клієнта
Можливо знаходити помилки на ранніх стадіях розробки	Помилки, зазвичай, виявляються на етапі тестування
Метою перевірки є архітектура та специфікація програмного забезпечення	Метою валідації є реальний продукт
Перевірка призначення для запобігання помилкам	Перевірка призначена для виявлення помилок
Також має назву тестування методом «білого ящика» або статичне тестування	Валідацію можна віднести до тестування методом «чорної скриньки» або динамічне тестування

1.2.3 Тестування програмного забезпечення

Тестування – це процес оцінювання системи або її компонентів з метою визначення того, задовольняє вона встановленим вимогам чи ні. Простими словами, тестування — це виконання системи з метою виявлення будь-яких прогалин, помилок або присутність невірних вимог, що суперечать замовленим вимогам.

Виходячи зі стандарту ANSI/IEEE 1059, тестування визначається як - процес аналізування елементів програмного забезпечення для виявлення відмінностей між чинними та необхідними умовами (тобто дефекти/помилки/баги) і для оцінки функцій елемента програмного забезпечення.

Одна із важливих переваг тестування полягає в тому, що воно визначає помилки та баги в програмному забезпеченні. Ці помилки можуть варіюватися від незначних до серйозних проблем, які можуть спричинити зупинку у роботі системи.

Тестування допомагає розробникам виявити та виправити ці помилки, перш ніж вони стануть проблемою для користувачів. Тестуючи програмне забезпечення, розробники можуть переконатися, що воно є стабільним і надійним, що важливо для успіху будь-якого програмного продукту. [10]

Нарешті, тестування може призвести до покращення продуктивності програмного забезпечення та взаємодії з користувачем. Тестуючи програмне забезпечення, розробники можуть визначити області, які потребують вдосконалення. Наприклад, тестування може виявити проблеми з продуктивністю, які можуть спричиняти повільну роботу програмного забезпечення.

Потім розробники можуть працювати над оптимізацією програмного забезпечення для збільшення продуктивності системи. Тестування також може виявити проблеми зручності використання, які можуть викликати розчарування у користувачів. Розв’язуючи ці проблеми, розробники можуть покращити взаємодію з користувачем і зробити програмне забезпечення більш зручним.

Ранній початок тестування зменшує витрати та час на перероблення та створення якісного програмного забезпечення, яке пропонується клієнту. Проте у життєвому циклі розробки програмного забезпечення (SDLC) тестування може розпочинатись з фази збору вимог і продовжити до розгортання програмного забезпечення.

Це також залежить від різновиду розробки, яка використовується. Наприклад, у загальновідомій моделі Waterfall формальне випробування системи проводиться на етапі тестування; але в інкрементній моделі верифікація та валідація виконується в кінці кожної ітерації, а в кінці тестується повністю вся програма. [10]

Важко визначити, коли припинити тестування, оскільки тестування — це нескінченний процес, і ніхто не може стверджувати, що програмне забезпечення перевірено на 100%. Щоб зупинити процес тестування, необхідно врахувати наступні аспекти:

- Терміни тестування;
- Завершення виконання тесту;
- Завершення функціонального та кодового покриття до певної точки;
- Рівень помилок падає нижче певного рівня, і жодних високопріоритетних помилок не виявлено;
- Управлінське рішення

Види тестування програмного забезпечення

Класифікація за ознаками наведена у таблиці 1.2.

Таблиця 1.2 - Види тестування програмного забезпечення

Тестування за мірою автоматизації	1. Ручне тестування 2. Автоматизоване тестування 3. Напівавтоматизоване тестування
За мірою підготовленості до тестування	1. Формальне тестування (тестування за документацією) 2. Тестування типу ad hoc або інтуїтивне тестування
За рівнем знання системи	1. Black box testing 2. White box testing 3. Grey box testing
За мірою ізольованості модулів	1. Component/unit testing 2. Integration testing 3. Системне тестування
За кількістю витраченого часу для проведення тестування	1. Альфа-тестування: • Smoke testing • Тестування нових функцій • Регресійне тестування • Acceptance testing 2. Бета-тестування
За визначеним об’єктом тестування	1. Тестування функцій 2. Тестування продуктивності • Load testing • Stress testing • Stability testing 3. Тестування зручності інтерфейсу за функціями 4. Тестування інтерфейсу 5. Тестування безпечності 6. Тестування локалізації 7. Тестування сумісності
За рівнем позитивності сценаріїв	1. Тестування позитивних сценаріїв 2. Тестування негативних сценаріїв

1.2.4 Історія тестування програмного забезпечення

Тестування програмного забезпечення з’явилося в один час із розробкою програмного забезпечення, яка розпочалася одразу після Другої світової війни. Авторство першої частини програмного забезпечення приписують вченому Тому Кілберну, яка була видана 21 червня 1948 року в Манчестерському університеті в Англії. Він виконував математичні розрахунки використовуючи інструкції машинного коду.

У 1950 році було презентовано тест Тьюринга, який обстежує інтелект машин.

У 1957 році методом тестування програмного забезпечення вважалося налагодження. Налагодження було основним методом тестування в той час і залишалося ним протягом наступних двох десятиліть. До 1980-х років команди розробників не лише виявляли та виправляли помилки програмного забезпечення, а й тестували програми в реальних умовах. Це заклало основу для ширшого погляду на тестування, яке охоплювало процес забезпечення якості, який був частиною життєвого циклу розробки програмного забезпечення.

Чарльз Л. Бейкер (RAND Corporation) розрізняє тестування програми від налагодження у своїй рецензії на книгу Дена МакКракена «Цифрове комп’ютерне програмування».

У 1958 році Джеральд М. Вайнберг сформував першу команду, яка спеціалізувалася на тестуванні.

У 1961 році з’являються перші згадки про тестування у книзі «Основи комп’ютерного програмування» Джеральда Вайнберга та Герберта Лідса.

У 1968 році тестування програмного забезпечення згадується у звіті НАТО.

У 1970 році Вінстон Ройс описав водоспадну модель у статті «Управління розробкою великих програмних систем».

У 1971 Річард Ліптон розробив концепцію мутації, що являли собою тип тестування програмного забезпечення, під час якого певні оператори вихідного

коду змінюються/мутуються, щоб перевірити, чи здатні тестові випадки знайти помилки у вихідному коді.

1973 рік - було розроблено техніку, основою якої був графік причинно-наслідкових зв'язків.

У 1978 році було запропоновано функціональне тестування.

Епоху 1979 року називають руйнівною, тому що у цей період тестування опиралося на ламання коду для того, щоб знайти якісь помилки у системах. Тоді ж і з'явився термін «тестування на злам».

У 1983 році була надрукована перша версія стандарту IEEE 829 для тестової документації ПЗ.

1985 рік компанія AutoTester розробила Autotester, перша комерційна програма для тестування ПК.

1986 рік Девід Гелперін і Вільям Гетцел заснували програмне забезпечення якості інженерії.

1988-2000 рр. Тестування вийшло на якісно новий рівень, що призвело до подальшого розвитку методологій і потужних інструментів для управління процесом тестування.

У 2004 році було розроблено Selenium, інструмент для тестування веб-додатків.

2005 рік став відкриттям для SoapUI який був випущений у вересні та став досить популярним.

2008 рік була заснована компанія з тестування програмного забезпечення Altom, також цього року було запущено службу тестування Applause. [11]

Сучасність: зараз існують різні типи тестування програмного забезпечення, які перевіряють різні аспекти програмного забезпечення в різних сценаріях. Штучний інтелект також використовується для тестування програмного забезпечення на основі поведінки кінцевих користувачів. Тепер тестувальники виконують різні типи тестування, наприклад модульне тестування, інтеграційне тестування, приймальне тестування, тестування продуктивності тощо.

1.3 Атестація систем на базі сервісно-орієнтованої архітектури

1.3.1 Сервісно-орієнтована архітектура

Сервісно-орієнтована архітектура або SOA — це гнучкий/результативний засіб інтеграції різноманітних програм/сервісів ціль якої є досягнення потреб які актуальні для бізнесу.

Відмінність від інших нелегких систем полягає у тому, що SOA містить в собі менші компоненти/сервіси/мікросервіси, які об’єднані між собою, щоб задовольнити потреби бізнесу.

Сервіси — це частини, які складають систему в єдине ціле, і коли є потреба в невеликій модифікації, а не в системі в цілому, змінюється конкретний сервіс. Сервіс – це складова, яка здатна виконувати поставлене завдання. На рисунку 1.3.1 зображено ієрархію додатків SOA [12].

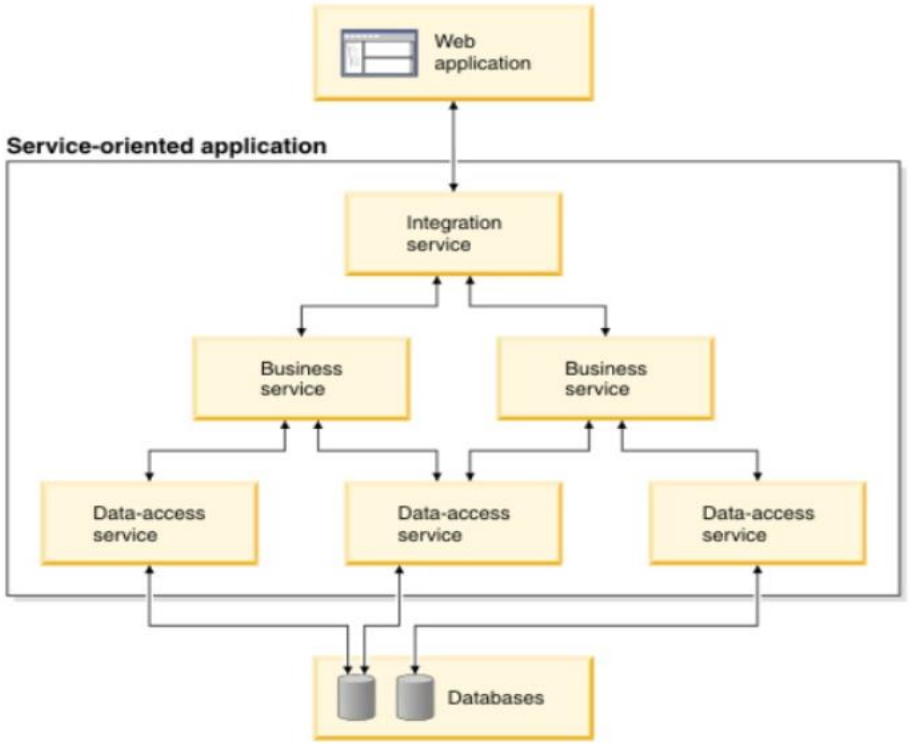

Рисунок 1.2 - Побудова сервісно-орієнтованої архітектури [13]

Система з такою архітектурою більш доступна для продажу, ніж доісторична система, оскільки замовник отримує необхідний результат для задоволення своїх потреб. Йому не обов'язково одержувати повністю всю систему.

Сервіси/мікросервіси є слабко пов'язаними, багаторазовими та без збереження стану, що допомагає системі ставати ще надійнішою. Ця модель є високорентабельною з боку розгортання, розроблення та обслуговування. На наступному рисунку можемо побачити порівняння архітектури системи від моноліту до сервісно-орієнтованої.

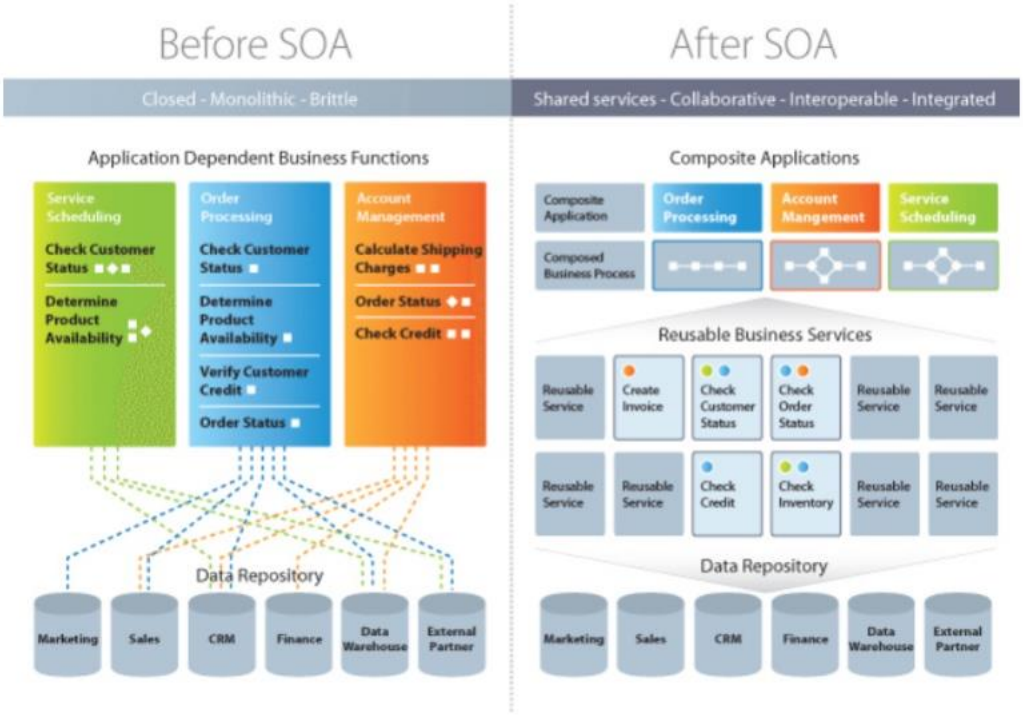

Рисунок 1.3 – Порівняння монолітної архітектури із сервісно-орієнтованою [13]

Кожен продукт, інфраструктура, модель, проходять етап тестування, щоб задовольнити людей, які в майбутньому будуть користуватись системою, з точки зору продукту високої якості.

1.3.2 Основні етапи атестації сервісно-орієнтованої системи

Тестування SOA – це процес перевірки того, що кожен компонент SOA правильно функціонує. Для того, щоб провести тестування SOA системи програмні інженери застосовують різні підходи та методи. Вони також під час тестування розглядають кожен рівень SOA, щоб бути впевненими, що кожен процес у системі працює. Тестування SOA, як і інше тестування в розробці ПЗ, дозволяє інженерам порівняти власний продукт із запитами клієнта та переконатися, що всі вимоги збережені. [13]

Тестування SOA не обмежується лише тестуванням рівня/тестуванням протоколу вебсервісу. Це загальне тестування архітектури та кожної її частини.

Планування тестування може бути схожим до звиклого процесу тестування. Тобто:

- огляд вимог
- розробка плану для тестування
- обрання методу та дизайну тестів
- налаштування середовища в якому буде відбуватись тестування
- фаза виконання тестування
- фаза результату тестування

Тестування SOA-системи складається з 3 рівнів архітектури:

- Service layers – рівні сервісу
- Process Layers – процесовий рівень
- Service consumers – зі сторони користувача

Рівень послуг: як уточнюється, він включає послуги, які пропонуються користувачеві згідно з наданим документом SRS. Наприклад, програма соціальних медіа містить служби входу/реєстрації, публікації, видалення, редагування тощо. Усі ці служби взаємодіють із доданою базою даних, отримують необхідну інформацію та виконують потрібну послугу.

Рівень процесу: цей рівень складається з процесів і служб, які складають єдину функціональність. Інтерфейс користувача та процедура наголошуються в основному на цьому рівні. Згідно з нашим прикладом, програма соціальних медіа вимагає значної інтеграції інтерфейсу користувача та бази даних, оскільки тут кожен клац призводить до абсолютно нового процесу. Ми враховуємо такі операції: видалення даних, створення нових трекерів, оновлення даних, додавання нових даних.

Рівень споживача: як випливає з назви, це рівень споживача, тобто це середовище, за допомогою якого споживач і постачальник взаємодіють. Таким чином, цей рівень містить весь інтерфейс користувача.

Відповідно до архітектури SOA, підкласифікація на рівні споживача описана таким чином:

- Наскрізний рівень.
- Рівень інтерфейсу.
- Рівень обслуговування.

Для проєктування тестів використовується низхідний підхід, оскільки під час проєктування служби множаться на кожному рівні, починаючи лише з інтерфейсу користувача та бази даних. Метод «знизу вгору» використовується для виконання тесту, тому що під час виконання тесту найнижчий рівень має найбільшу кількість незалежних служб, які необхідно перевірити в першу чергу, щоб перевірити, чи працює їх незалежна функціональність.

1.4 Тестування серверної частини (backend) завдяки API

API (програмний інтерфейс застосунку) — це набір усіх процедур і функцій, які дозволяють нам створювати програму за допомогою доступу до даних або функцій операційної системи чи платформ. Тестування такого програмного інтерфейсу відоме як тестування API.

REST API тестування – це відома техніка тестування вебдодатків із відкритим кодом, яка застосовується для тестування RESTful API. Метою такого тестування є надсилання різних запитів через протокол HTTP/S, а отримані відповіді відповідають на питання, чи працює REST API правильно, чи ні.Для того, щоб протестувати REST API використовують різні методи, але найвідоміші із них це GET, POST, PUT і DELETE.

REST розшифровується як Representational State Transfer. Це стиль архітектури та метод комунікації, який застосовується при розробці веб-сервісів або вебслужб. Цей метод комунікації став логічним вибором при створенні API. Він дає можливість для користувачів підключатися та результативно працювати з хмарними додатками.

Також API можна характеризувати як набір програмних інструкцій для доступу до вебдодатку. Тобто, це набір команд, які застосовують як окрему програму для прямого зв’язку між сервісами та використовування функцій для отримання інформації з цих сервісів. [14]

Різні способи для тестування REST API

Підтверджувальне тестування: воно вважається гарантією правильної розробки та відбувається на останніх етапах перевірки поведінки та аспектів ефективності продукту.

Тестування безпеки: це робиться для захисту реалізації API від зовнішніх загроз. Він також включає розробку контролю доступу API, перевірку методологій шифрування та керування правами користувачів.

Тестування інтерфейсу користувача: фокусується на інтерфейсі користувача для API, а не на тестуванні самого API.

Функціональне тестування: окремі функції в кодовій базі включені до функціонального тестування та обробляють функції API запланованим чином.

Навантажувальне тестування: перевіряє, чи працює надане рішення за планом, і зазвичай виконується після завершення всієї кодової бази.

Виявлення під час виконання та помилок : головним чином зосереджено на виявленні помилок, помилках виконання та моніторингу та має справу з

універсальним результатом кодової бази API (оскільки це пов’язано з фактичним запуском API).

Тестування на проникнення: бере участь у процесі аудиту як другий тест. [14]

Підтвердження автентичності результатів.

При тестуванні API виникає питання: як підтвердити автентичність результатів?

Щоб дати відповідь на це запитання потрібно розглянути коректні HTTP статус кодів. [15]

Знайти їх можна у таблиці 1.3

Таблиця 1.3 – Можливі статуси кодів HTTP

1* Інформаційний	2* Успішний	3* Перенаправлення	4* Помилка вводу	5* Помилка сервера
100 Продовження	200 ОК	300 Множинний вибір	400 Неправильний запит	500 Внутрішня помилка сервера
101 Перемикання протоколів	201 Створено	301 Переміщено	401 Несанкціонований доступ	501 Метод не підтримується
102 Обробка	202 Прийнято	302 Тимчасово переміщено	402 Потрібна оплата	502 Помилка шлюзу
	203 Інформація не авторитетна	303 Розглянути інший ресурс	403 Заборонено	503 Сервіс недоступний

	204 Немає вмісту	304 Ресурс не змінювався	404 Не знайдено	504 Шлюз не відповідає
	205 Повтор посилання вмісту	305 Потрібно використати проксі-сервер	405 Не припустимий метод	505 Версія HTTP не підтримується

Проблеми, які зустрічаються при тестуванні API

Початкове налаштування тестування API: підготовка до тестування API та запуск його середовища вимагає певних технічних навичок, що є однією з найскладніших частин процесу. На цьому етапі проблеми будуть виникати часто і у великих кількостях.

Підтримка форматування даних (оновлення схеми тестування API): схема діє як план для опису синтаксису API та граматики тексту. Він визначає, як дані форматуються в коді, обробляє всі запити та відповіді, а також містить формат даних. Це необхідно підтримувати протягом усього процесу. Хоча цей виклик, ми можемо подолати. Це робиться шляхом регулярного обслуговування та оновлення схеми, щоб переконатися, що щойно додані параметри включені в схему.

Послідовність викликів API: під час роботи з багатопотоковими програмами користувач може надсилати кілька запитів API одночасно, що може стати проблемою для тестування, якщо вони не надіслані в правильному порядку. Щоб подолати цю проблему, виклики API мають бути в правильному порядку, щоб програма видавала помилку.

Перевірка параметрів: запитуючи запити API, команда тестування також може виявити, що перевірка параметрів буде складною. Велика кількість параметрів і варіантів їх використання робить це непростим завданням. Ми повинні бути впевнені, що дані кожного важливого параметра використовують правильний рядковий або числовий тип даних, який відповідає призначеному діапазону значень, обмеженню довжини та критерію перевірки. Цю проблему можна подолати шляхом постійного синтетичного моніторингу API, щоб якомога раніше виявляти проблеми.

Тестування всіх можливих комбінацій запитів параметрів: зв’язок між системами (обробляється API) здійснюється шляхом призначення значень даних параметрам і передачі цих параметрів через запити даних. Тут необхідно перевірити всі комбінації запитів параметрів в API, щоб перевірити наявність недоліків у конкретних конфігураціях. В іншому випадку проєкт може мати два значення для одного параметра. Тому спробуйте додати менше додаткових

параметрів, щоб зменшити ймовірність поєднання. Також має бути правильний вибір програм, які не є складними для повсякденних операцій.

Залишаючи поза увагою твердження про час відповіді: API, як правило, займають менше часу під час виклику. Але що, коли це зайняло понад 10 секунд, чи буде це ефективним? Зовсім ні! і стає складнішим для тестувальників програмного забезпечення. Отже, налаштуйте твердження про час відповіді, які мають бути обґрунтованими та зможуть відобразити час відповіді. Затвердження великого порогового значення часу відгуку набагато краще, ніж нічого, головним чином під час тестування кінцевих точок виробництва.

Інтеграція системи відстеження. Завжди переконатися, що система API працює належним чином із системою відстеження даних чи ні, також є великою проблемою. Отже, нам потрібно повернути правильні відповіді щодо того, чи правильно працює виклик. Оскільки це останній крок процесу тестування, команда може бути надто розчарована, тому не приділить йому належної уваги. Щоб подолати цю проблему, потрібно зосередитися на проєктній частині. Також перевірте його інтеграцію з іншими системами.

РОЗДІЛ 2 РОЗРОБКА ВИМОГ І МОДЕЛЮВАННЯ СЕВІСНО-ОРІЄНТОВАНОЇ СИСТЕМИ

2.1 Аналіз і специфікація вимог до системи «Інтернет-магазин»

Для того, щоб створити необхідну систему – потрібно скласти детальний план проєктування та поставити задачу перед виконанням розробки. Тож, почати проєктування слід з виявлення вимог.

Бізнес-вимоги – це кінцева ціль (або цілі) системи. В цьому випадку треба описати бізнес вимоги для замовників системи на базі сервісно-орієнтованої архітектури. Для того, щоб з'ясувати основні бізнес вимоги потрібно дати відповідь на такі питання:

- Хто є основними стейкхолдерами для системи?
- Які вони мають вимоги, або побажання для системи?
- Як можна класифікувати дані вимоги?

Стейкхолдери – фахівці, які є зацікавлені у створені даного продукту та тим чи іншим способом вони можуть впливати на бізнес.

Стейкхолдерами для даної системи є:

- Майбутній власник інтернет-магазину
- Менеджери з продажів

Цілі, яких вимагають стейкхолдери:

1. Створити систему, яка буде розподіленою на сервіси
2. Розробити можливість, щоб сервіси могли звертатись один до одного
3. Можливість швидкого виправлення даних в одному сервісі без негативного впливу для інших сервісів.

Отримавши дані від стейкхолдерів, можемо висунути такі бізнес-вимоги:

1. Наявність окремих сервісів, але з можливістю спілкування між собою
2. Безперебійне оновлення та виправлення помилок

3. Можливість маніпулювання даними у режимі реального часу без шкоди для інших сервісів.

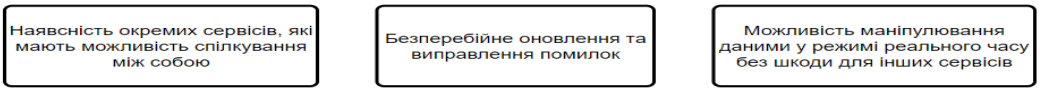

Рисунок 2.1 - Бізнес вимоги для системи на базі SOA

Наступний крок в проєктуванні - це з’ясування функціональних та нефункціональних вимог до системи.

Тож, маємо такі функціональні вимоги до системи:

Для Products сервісу:

- Створення товару
- Редагування товару
- Видалення товару

Для Cart сервісу:

- Створення кошику та додавання в неї товару
- Редагування кошику
- Видалення кошику

Рисунок 2.2 - Функціональні вимоги для сервісно-орієнтованої системи «Інтернет-магазин»

На рисунку 2.3 показано типи нефункціональних вимог до інформаційної системи «Інтернет-магазин», побудованої на базі сервісно-орієнтованої архітектури.

Рисунок 2.3 - Типи нефункціональних вимог до сервісно-орієнтованої системи «Інтернет-магазин»

Масштабованість повинна бути, бо інтернет-магазин може бути розширений за кількістю сервісів та додаткових функцій, тож, система повинна бути масштабованою.

Надійність – це ступінь захищеності системи від збоїв та кібератак.

Надійність, продуктивність, експлуатаційну придатність відносять до атрибутів якості.

Обмеження – формулювання умов, які модифікують вимоги, звужуючи вибір можливих рішень щодо їх реалізації.

2.2 Постановка задачі та алгоритм розв’язання задачі

2.2.1 Постановка задачі

Для написання кваліфікаційної роботи розглядався принцип створення системи на базі SOA для проведення тестування даної системи.

Як тематику створення системи було обрано «Інтернет-магазин».

Інтернет-магазин на базі сервісно-орієнтованої архітектури значно полегшить роботу для користувачів системи, тому що дана архітектура має багато переваг і дозволяє постійно змінювати та виправляти помилки в різних сервісах без шкоди для системи в цілому. В тому ж числі сервісно-орієнтована архітектура допомагає власникам системи бути гнучкими та легко підлаштовуватись до умов ринку.

Для того, щоб створити Інтернет-магазин потрібно зосередитись на тому, яким чином розбити систему на модулі. На даному етапі було прийнято рішення розбити систему на два сервіси:

- Товари (Products)
- Кошик (Cart)

Завдяки сервісу «Товари», власник інтернет-магазину або менеджер магазину, може з легкістю створювати, редагувати та видаляти необхідні об’єкти.

А сервіс «Кошик» відповідає за збереження вибору користувача інтернет-магазину. Тож, завдяки, даному сервісу клієнт зможе додати товар в кошик, редагувати свій вибір та за потреби видалити.

Через те, що система буде розглядатись зі сторони бекенду (API), треба не забувати про необхідність спілкування сервісів між собою. Щоб це перевірити було вирішено використовувати Swagger UI.

Swagger – фреймворк для специфікації RESTful API, який дає можливість не тільки дивитися специфікацію систем, але й відправляти запити завдяки Swagger UI.

Тож, можна представити інформаційну модель системи «Інтернет-магазин».

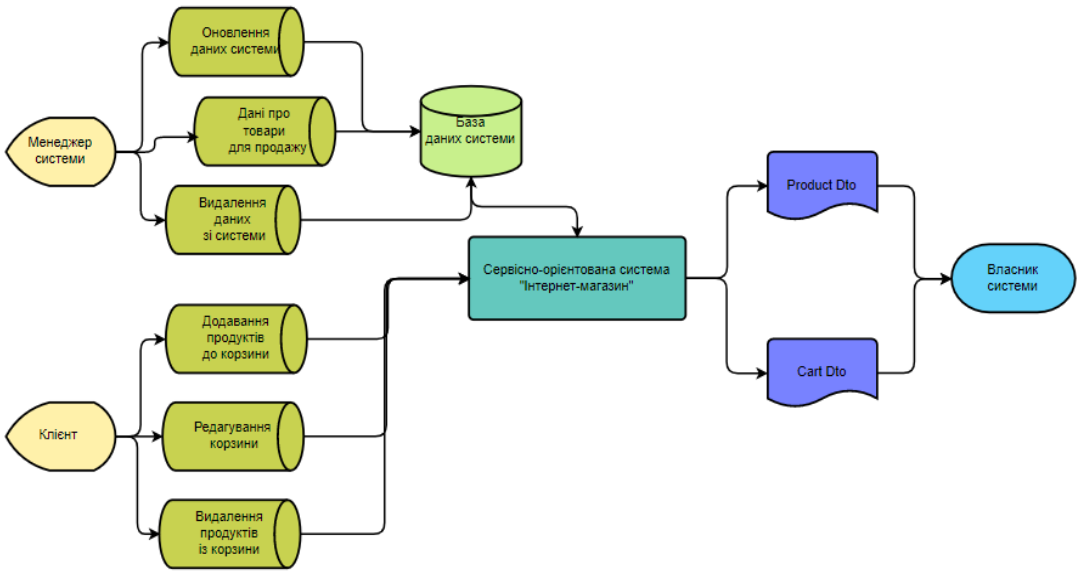

Рисунок 2.4 - Інформаційна модель сервісно-орієнтованої системи «Інтернет-магазин»

Вихідна інформація

Вихідна інформація – це результат роботи системи. Для даної системи вихідною інформацією виступають Product Dto і Cart Dto.

Product Dto і Cart Dto це відповідь на запит до системи. Опис вихідних повідомлень можна знайти у таблиці 2.1

Таблиця 2.1 – Перелік та опис вихідних повідомлень

Назва вихідного повідомлення	Форма подання	Періодичність	Допустимий час затримки	Користувачі
Схема результату створення товару	Машинограмма/ екранограмма	За запитом	-	Менеджери або власник магазину
Схема результату модифікації товару	Машинограмма/ екранограмма	За запитом	-	Менеджери або власник магазину
Схема результату видалення товару	Машинограмма/ екранограмма	За запитом	-	Менеджери або власник магазину
Схема результату додавання товару в корзину	Машинограмма/ екранограмма	За запитом	-	Клієнт магазину
Схема результату модифікування товарів в корзині	Машинограмма/ екранограмма	За запитом	-	Клієнт магазину
Схема результату видалення товарів із корзини.	Машинограмма/ екранограмма	За запитом	-	Клієнт магазину

До вихідних повідомлень належать:

- Схема результату створення товару;
- Схема результату модифікації товару;
- Схема результату видалення товару;
- Схема результату додавання товару в корзину;
- Схема результату модифікування товарів в корзині;
- Схема результату видалення товарів із корзини.

Вхідна інформація

Вхідна інформація до системи – це створення позицій товарів для магазину, їхні ознаки: кількість, ціна, залишок на складі тощо. Також вхідною інформацією є дані про замовлення, які надає клієнт магазину. Це може бути: найменування товару, замовлена кількість, розрахункова ціна, тощо.

Перелік і опис вхідних даних наведено в таблиці 2.2.

Таблиця 2.2 – Перелік і опис вхідних даних

№ з/п	Назва вхідного повідомлення	Форма подання	Термін і частота надходження	Джерело
1	Запит на створення або редагування товару	Масив даних	Щоразу за запитом	Менеджери або власник магазину
2	Запит на видалення товару	Масив даних	Щоразу за запитом	Менеджери або власник магазину
3	Запит на додавання або редагування корзини	Масив даних	Щоразу за запитом	Клієнт магазину
4	Запит на видалення товарів з корзини	Масив даних	Щоразу за запитом	Клієнт магазину

2.2.2 Алгоритм розв’язання задачі

Для розв’язку задачі було побудовано два алгоритми дій. Перший алгоритм відноситься до сервісу товарів. Слід відзначити, що даний алгоритм відбувається від лиця менеджера магазину.

На рисунках зображено алгоритм розв’язання задачі у вигляді блок-схем.

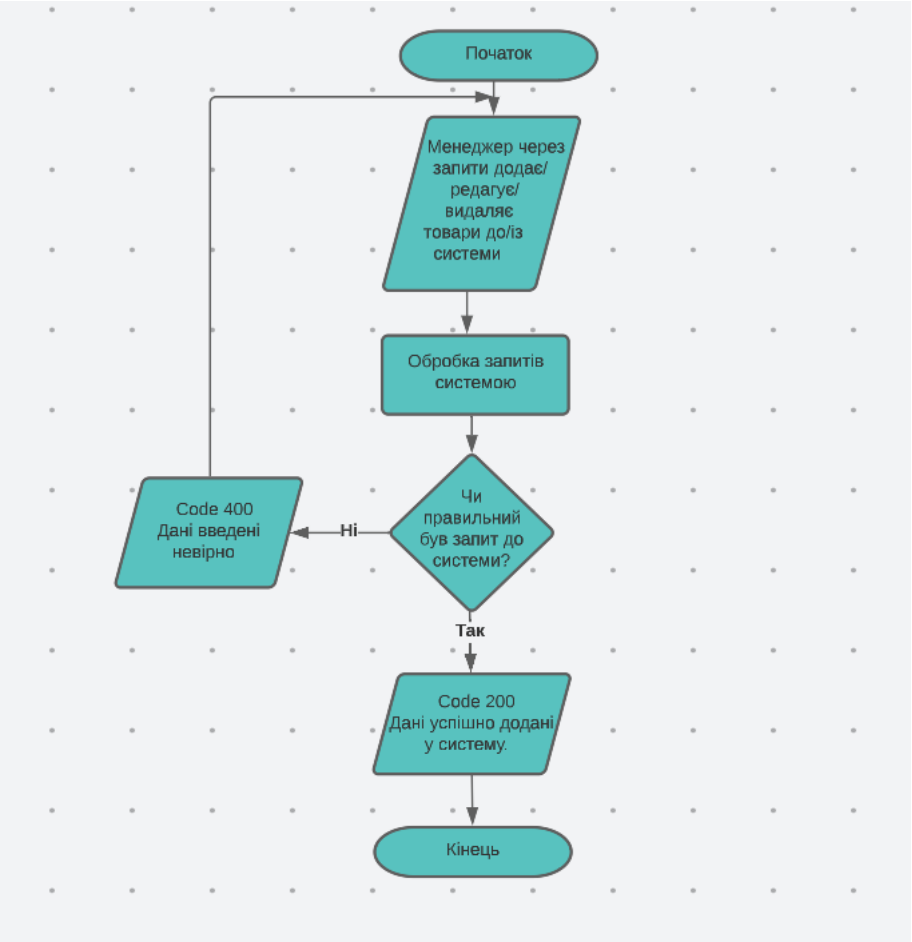

Рисунок 2.5 - Алгоритм розв’язання задачі для створення/модифікування/видалення товару

Другий алгоритм був створений для сервісу корзини. Цей алгоритм відбувається від лиця клієнту магазину.

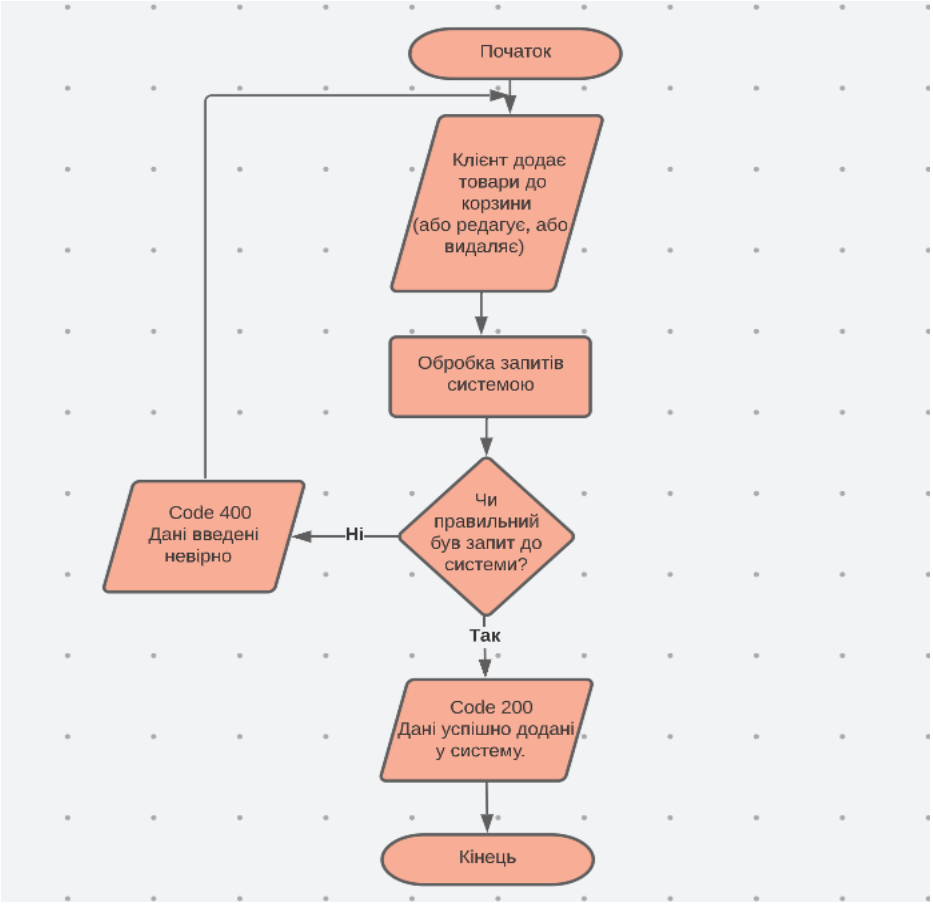

Рисунок 2.6 - Алгоритм розв’язання задачі для створення/модифікування/видалення кошику

Використовувана інформація

У таблиці 2.3 представлена інформація про використовувані масиви.

Таблиця 2.3 – Перелік масивів використовуваної інформації

Масив	Максимальна кількість записів
Масив даних менеджера	100000
Масив даних клієнта	100000

2.3 Моделювання розподіленої системи «Інтернет-магазин» на базі SOA

2.3.1 Моделювання поведінки системи

Для того, щоб змоделювати поведінку системи було використано уніфіковану мову моделювання, яка використовується у парадигмі ООП.

Діаграма прецедентів – описує можливі маніпуляції акторів із прецедентами в системі. Основані ці дії на функціональних можливостях системи.

Тож, складові частини діаграми прецедентів:

- Актор – всі можливі користувачі системи;
- Прецедент – якась функція, яку виконує актор із системою.

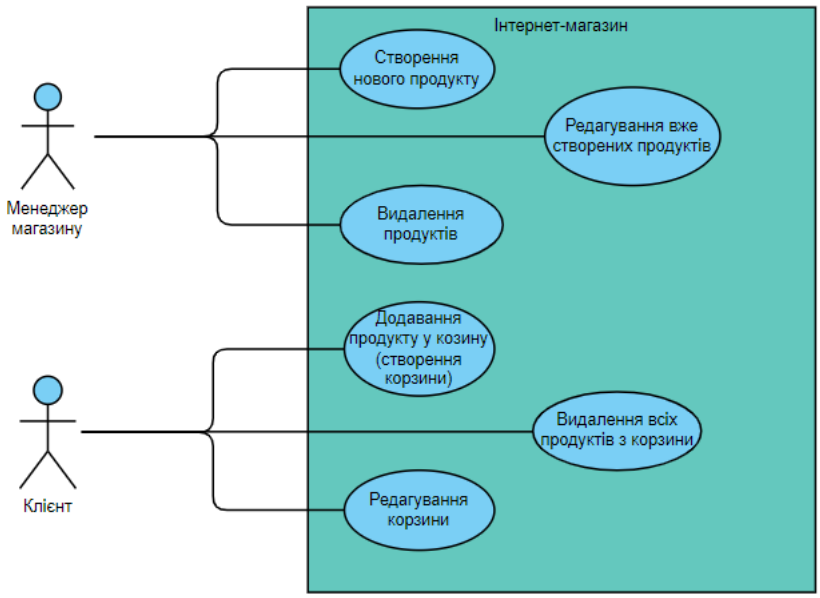

Рисунок 2.7 - Діаграма прецедентів системи «Інтернет-магазин»

Діаграма послідовності — це форма діаграми взаємодії, яка показує об’єкти як лінії життя, що проходять по сторінці, а їх взаємодія з часом представлена у вигляді повідомлень, намальованих у вигляді стрілок від вихідної лінії життя до цільової лінії життя.

Схеми послідовностей добре показують, які об’єкти взаємодіють з якими іншими об’єктами; і які повідомлення викликають цю комунікацію.

Схеми послідовності не призначені для показу складної процедурної логіки.

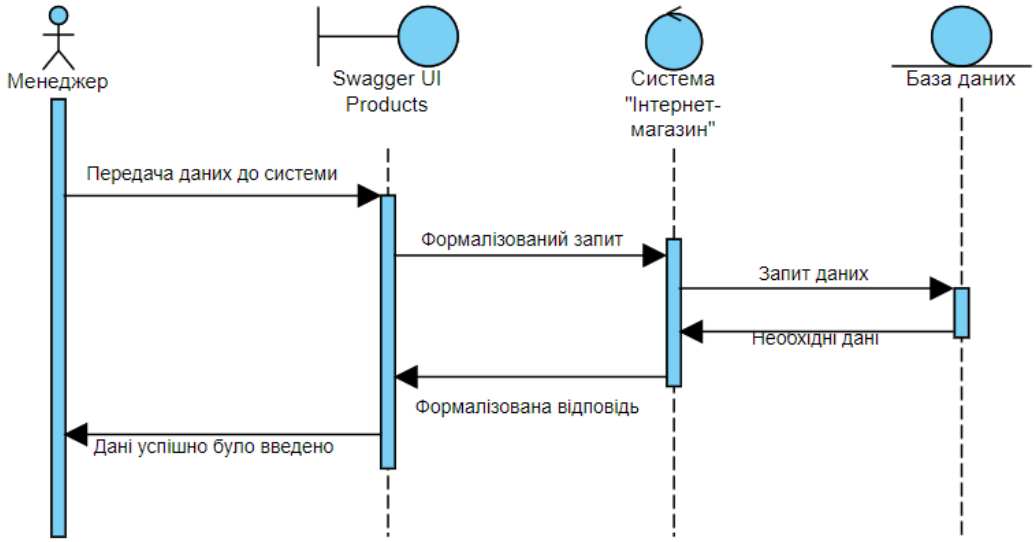

Рисунок 2.8 - Схема послідовності системи «Інтернет-магазин» зі сторони менеджера магазину

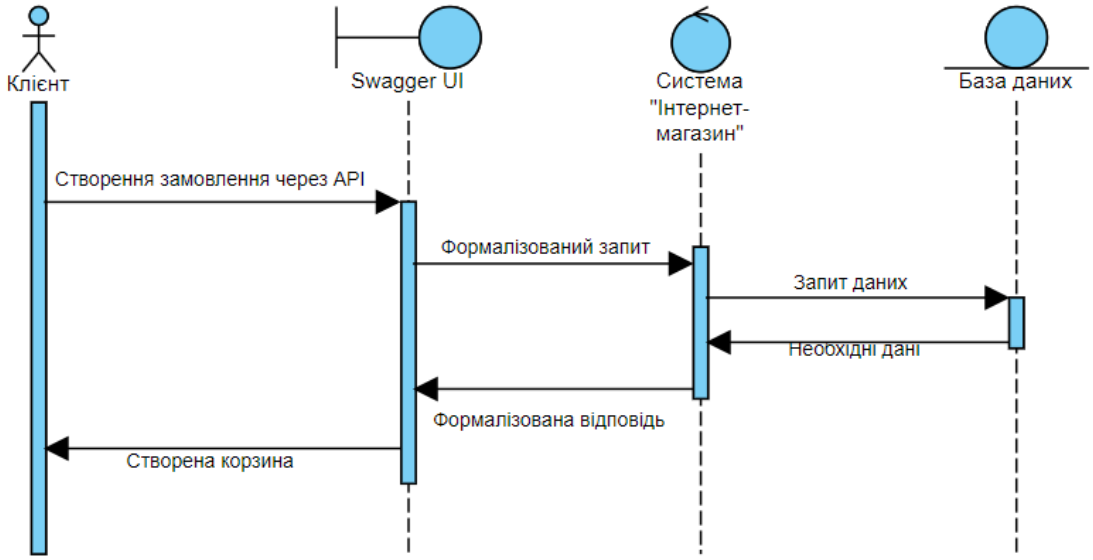

Рисунок 2.9 - Схема послідовності системи «Інтернет-магазин» зі сторони клієнта магазину

Діаграма визначень подана на наступному рисунку

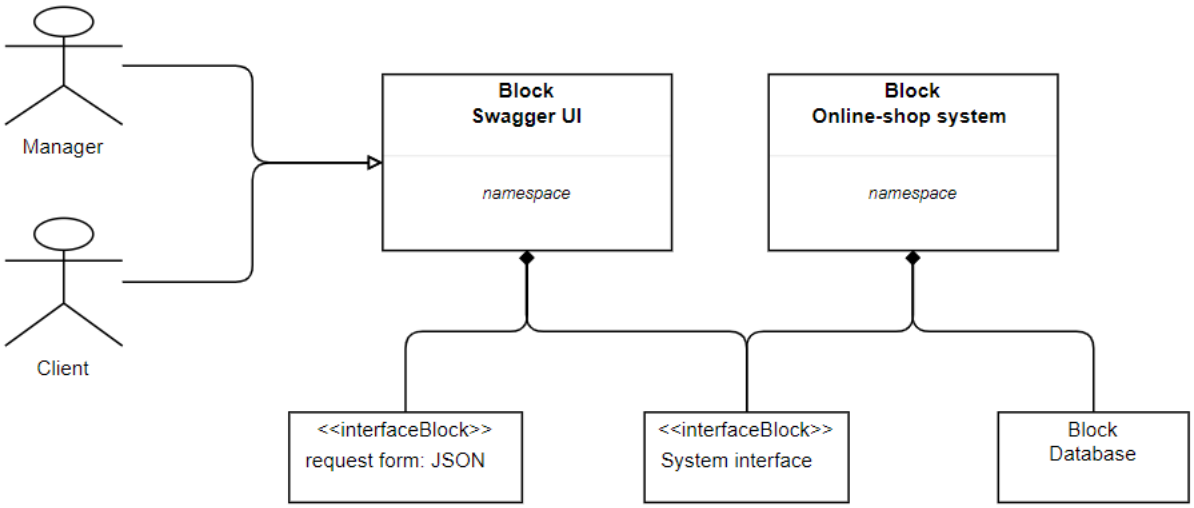

Рисунок 2.10 - Діаграма визначень системи «Інтернет-магазин»

2.3.2 Інтерфейс готового API для системи «Інтернет-магазин»

У цьому підрозділі буде наведено реалізований інтерфейс для API товарів та корзини. В першу чергу розглянемо товари:

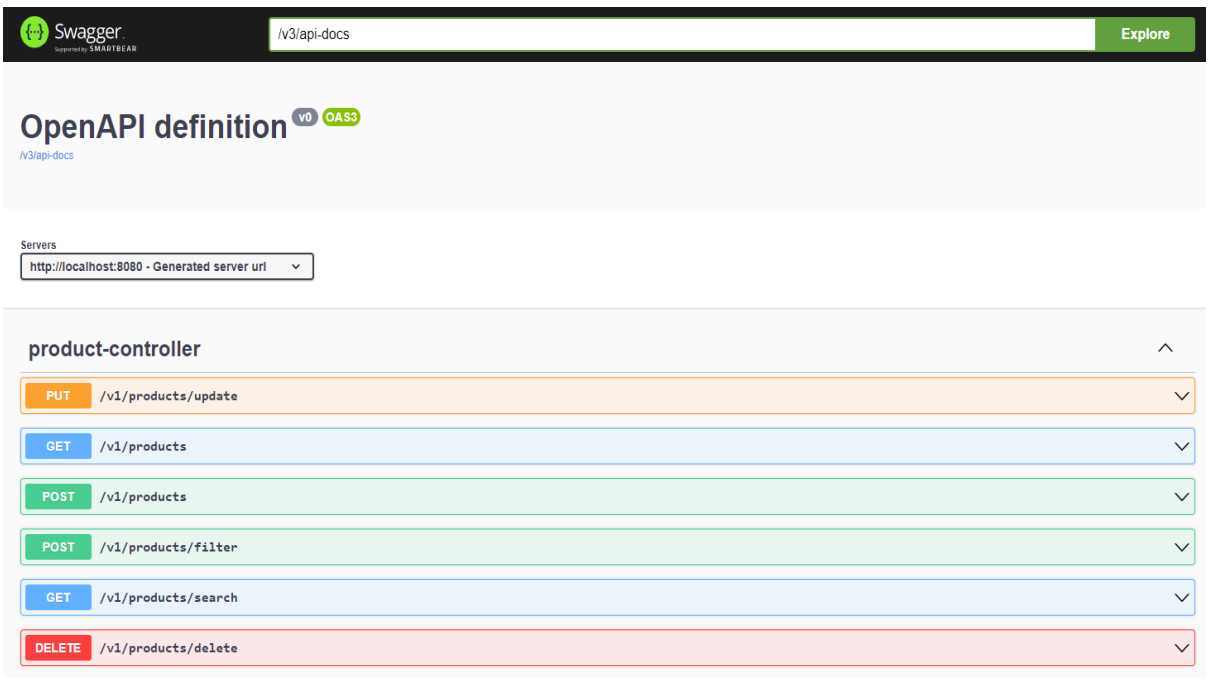

Рисунок 2.11 - Головна сторінка API для товарів

На наступних рисунках зображено приклад інтерфейсу запиту PUT і GET для сервісу товарів

Рисунок 2.12 - Запит PUT для товарів

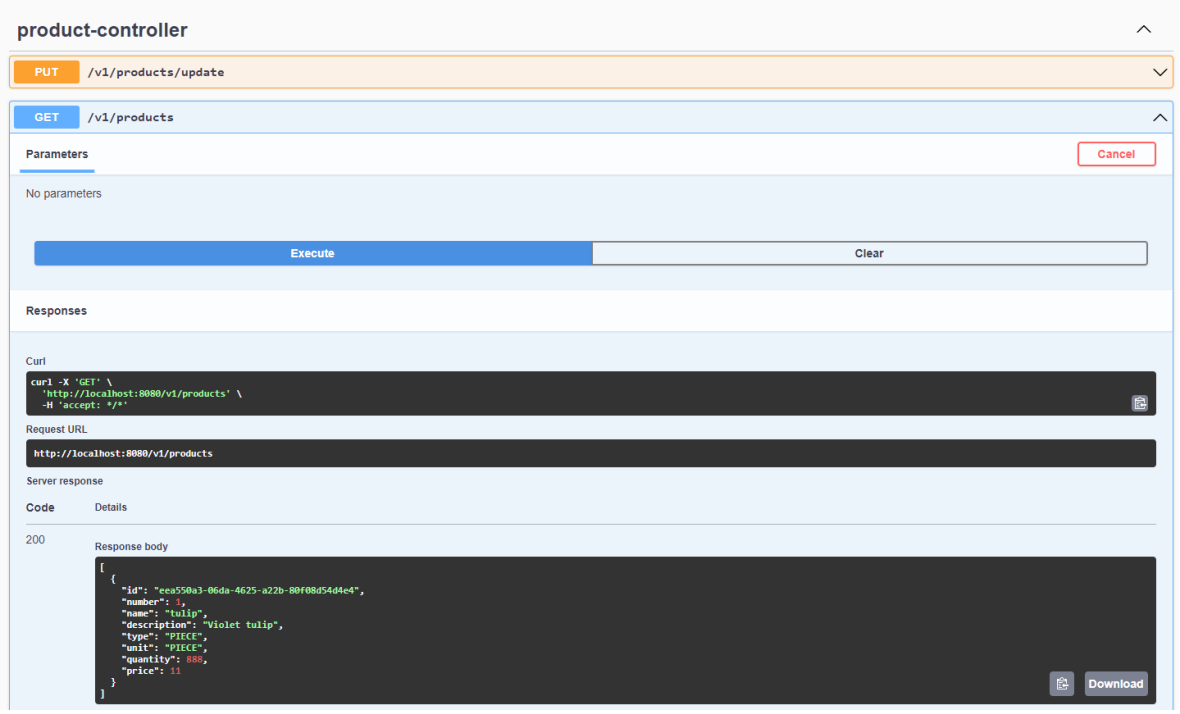

Рисунок 2.13 - Запит GET для товарів

Наступним буде інтерфейс другого сервісу – кошик.

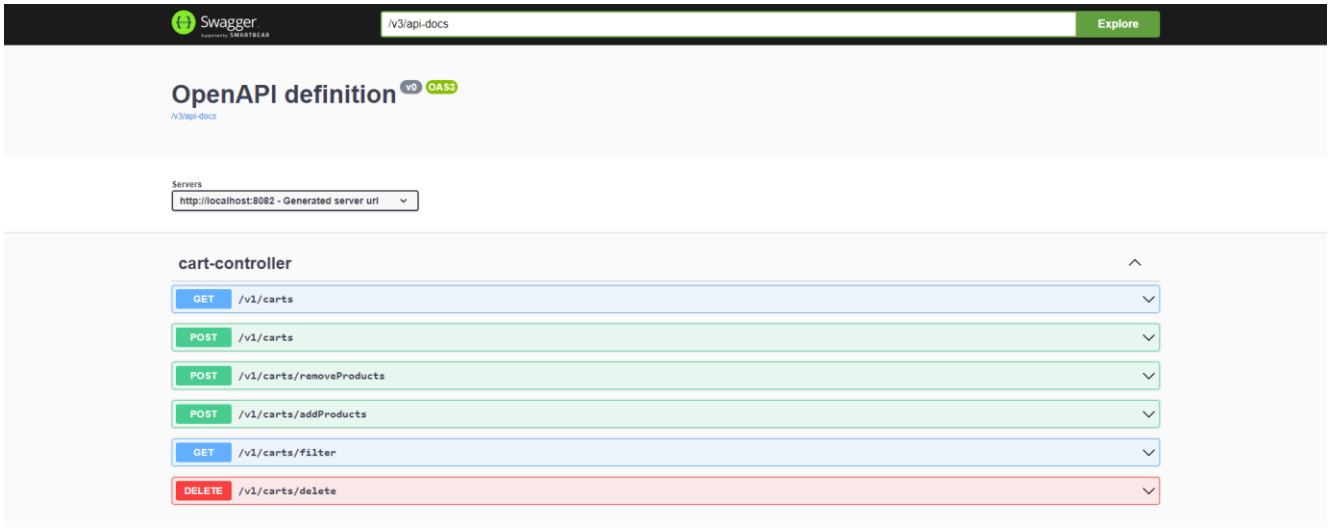

Рисунок 2.14 - Головна сторінка API для корзини

Нижче наведено інтерфейс для запиту POST для сервісу кошика

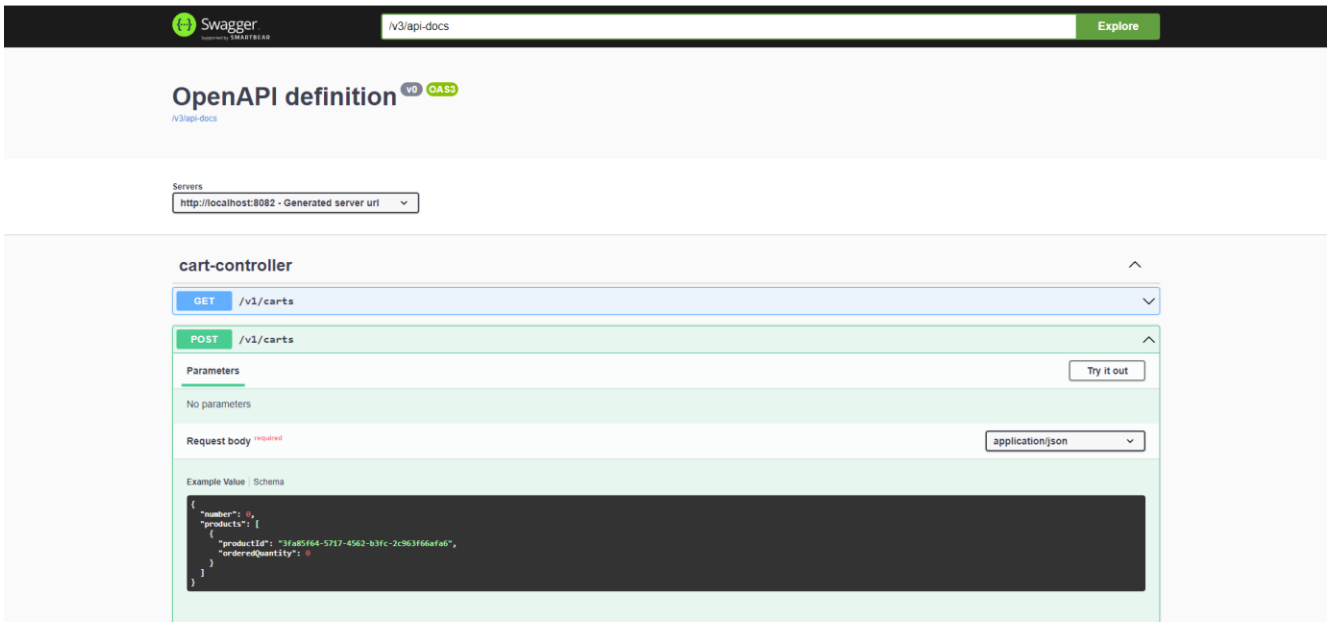

Рисунок 2.15 - Запит POST для створення кошику

РОЗДІЛ 3 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ТЕСТУВАННЯ СИСТЕМИ

3.1 Проєктування за стандартом IDEF0

IDEF (Integrated Definition) — це графічна методологія зображення процесів, яка популярна тим, що використовується для розробки систем та пз. Дані методи характерні для моделювання функціональних даних, симуляції, аналізу та отримання результативних знань. IDEF був задуманий і вперше описаний ВПС США в на початку 1980-х років.

Він був розроблений для стандартного документування та бізнес-аналізу процесів. Зараз ця методологія використовується як регламентований підхід до аналізу підприємства, охоплення моделей процесів «як є» та для симуляції діяльності в бізнес-групі.

Незважаючи на те, що IDEF спочатку був винайдений для виробничого середовища, тепер ця методологія моделювання процесу є популярною для більш широкого використання та для розроблення програмного забезпечення в цілому.

Контекстна діаграма – це діаграма початкового (верхнього) рівня, що представляє логіку системи вцілому, у вигляді «чорного ящика», і поєднує її із зовнішнім світом за допомогою використання інтерфейсних дуг.

Функціональний блок є основною складовою контекстної діаграми. Також можуть бути наявні певна, необмежена кількість стрілок для зображення мети моделювання.

Наступний рисунок зображує моделювання процесу тестування сервісно-орієнтованої системи.

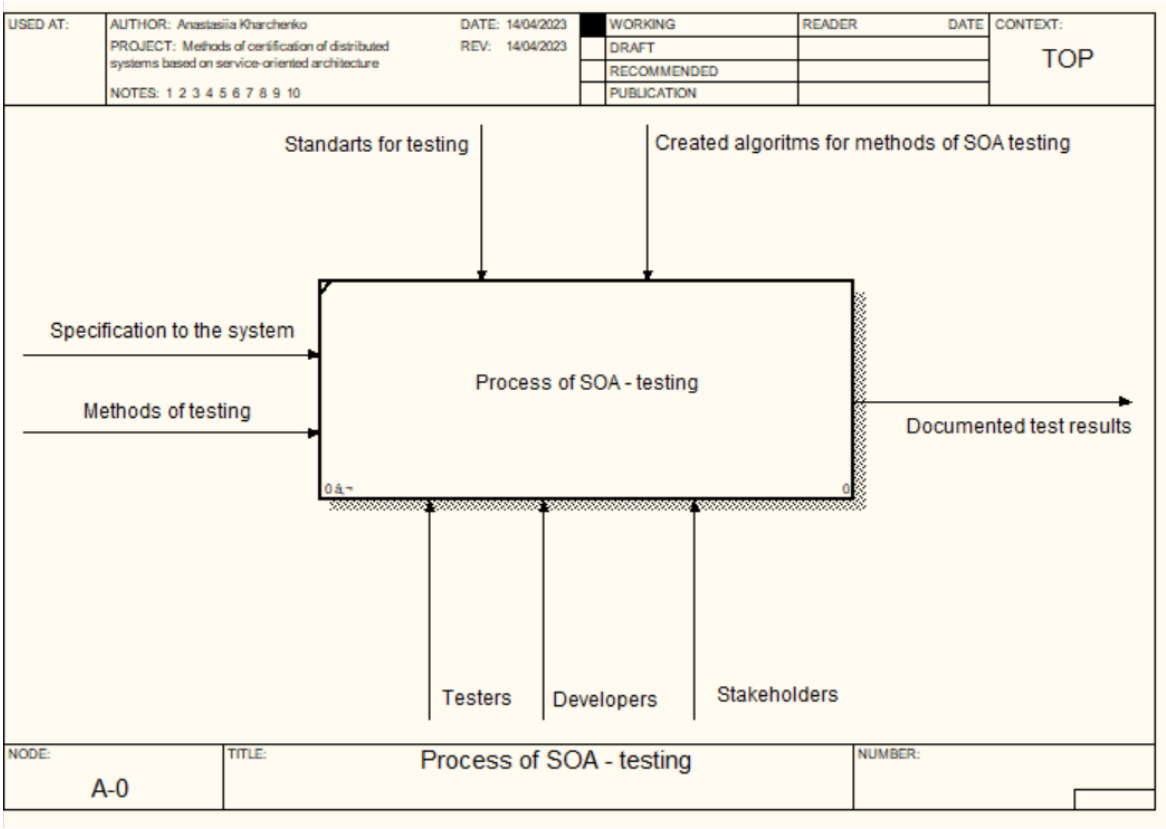

Рисунок 3.1 - Контекстна діаграма для процесу тестування системи на базі SOA

На діаграмі можна побачити вхідні дані, вихідні, механізм та способи управляння процесом тестування.

Вхідні дані:

- Специфікація системи
- Існуючі методи тестування

Механізми які необхідний для роботи:

- Тестувальники
- Розробники
- Стейкхолдери

Способи управління, які задіяні у роботі:

- Загально відомі стандарти тестування
- Методи тестування SOA-систем

Вихідні дані виступають у вигляді задокументованих тест-результатів.

Для подальшого проєктування є необхідність зануритись у більш деталізований ступінь процесу. Це досягається завдяки діаграмі декомпозиції. Так як контекстна діаграма не дає повного бачення процесу, а лише виділяє головний процес.

Діаграма декомпозиції першого рівня – це деталізований розгляд опису процесу. Для розуміння можна сказати, що це те саме, що розбити складну задачу на декілька малих. Ці процеси, які відбуваються у діаграмах декомпозиції, вони можуть бути як одночасними так і паралельними.

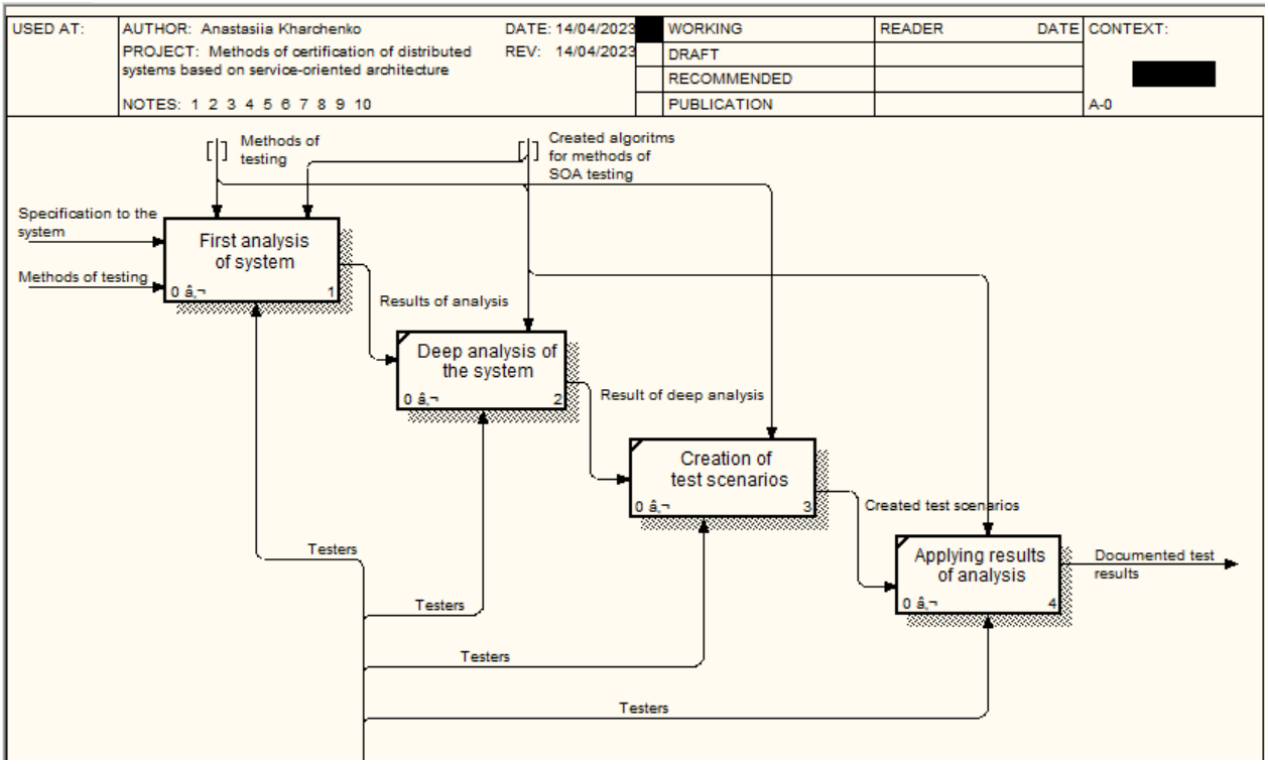

Рисунок 3.2 - Діаграма декомпозиції першого рівня для процесу тестування системи

Першою задачею стоїть поверхневий огляд системи, для якої буде проведено тестування. Потрібно це для того, щоб зрозуміти принцип роботи системи та підготуватися до наступного процесу. Отримуючи результат поверхневого аналізу системи, ми переходимо до глибокого аналізу.

Це означає, що ми повинні проаналізувати всі взаємодії системи, знайти закономірності, освоїти всі бізнес-процеси системи та зрозуміти потреби стейкхолдерів.

Наступним кроком буде продумання можливих сценаріїв, які в майбутньому будуть використані для тестування. І завершальний процес це закріплення результатів аналізу, що при виході дає задокументовані тест результати.

Наступна діаграма декомпозиції відображає другий рівень тестування сервісно-орієнтованої системи.

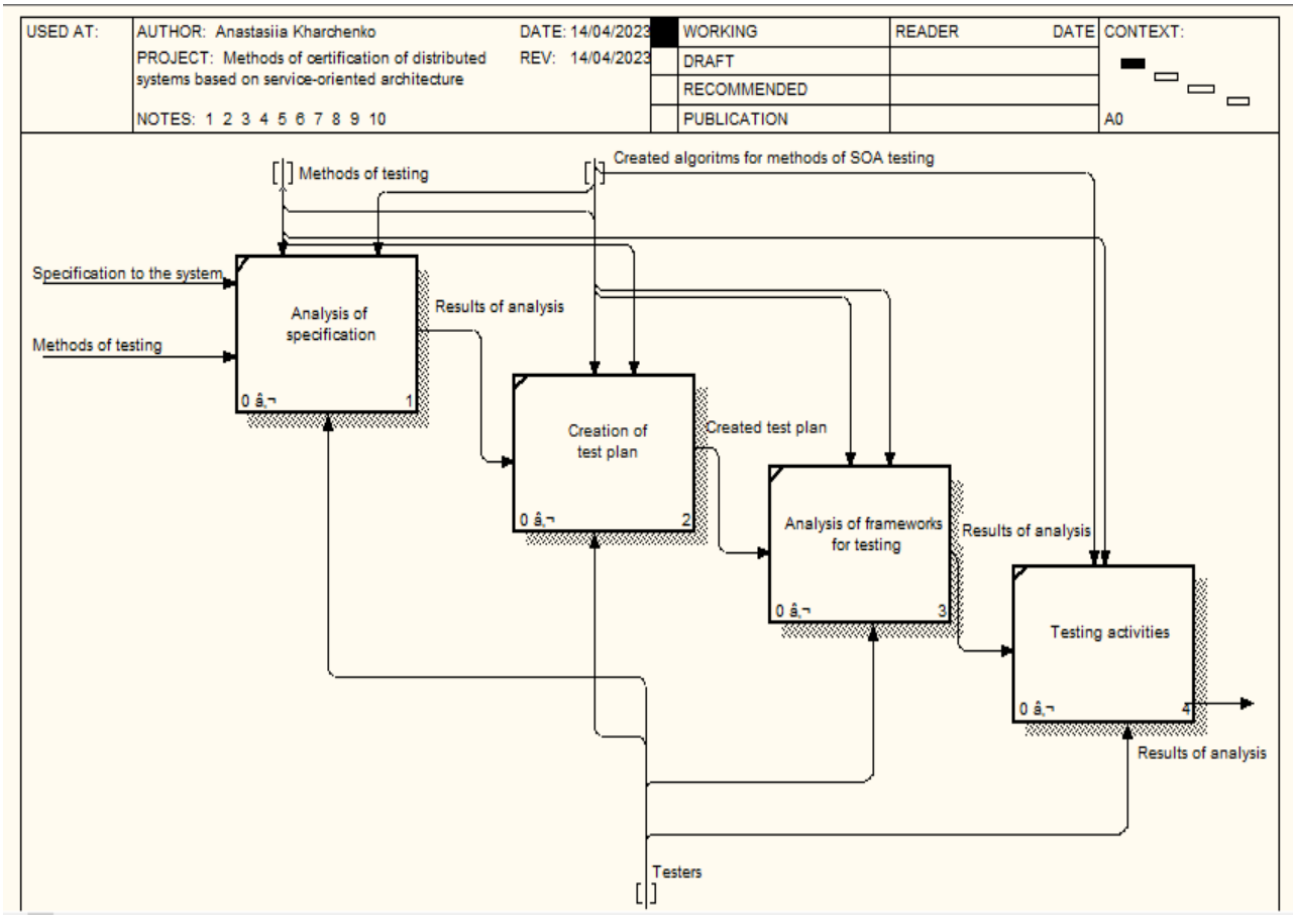

Рисунок 3.3 - Діаграма декомпозиції другого рівня для процесу тестування системи

Першим етапом для декомпозиції другого рівня буде аналіз та перевірка специфікації до системи. Специфікація системи – важливий етап, адже, відсутність аналізу специфікації перед розробкою або тестуванням дає можливість розмноженню кількості помилок, які в майбутньому можуть дорого коштувати.

Другий етап – це створення тест-плану. Якщо завдяки першому рівню діаграми ми змогли продумати можливі сценарії для тестування, то очікується, що на другому рівні ми матимемо змогу скласти органічний тест-план.

Третій етап – це вибір середовища завдяки якому буде зроблене тестування. На цьому етапі потрібно проаналізувати існуючі програми, або фреймворки для тестування і обрати один з найкращих по відгукам спеціалістів. І заключним етапом на другому рівні буде початок написання тестів.

3.2 Вибір програмного середовища для тестування

Cypress

Cypress — це інтерфейсний інструмент тестування, створений виключно для сучасної мережі на основі JavaScript. Він спрямований на вирішення проблемних моментів, з якими стикаються розробники та інженери з контролю якості під час тестування програми. Cypress є більш зручним для розробників інструментом, який використовує унікальну техніку маніпулювання DOM і працює безпосередньо у браузері. Cypress також надає унікальну інтерактивну програму тестування, у якій він виконує всі команди.

В офіційній документації зазначено, що Cypress принципово та архітектурно відрізняється від Selenium.

Використовуючи Cypress, тестувальники або розробники можуть створювати:

1. Модульні тести
2. Інтеграційні тести
3. End to End тести

Переваги:

- Фреймворк Cypress робить знімки під час виконання тесту. Це дозволяє QA або розробникам наводити курсор миші на певну команду в журналі команд, щоб точно побачити, що сталося на цьому конкретному кроці.

- Не потрібно додавати явні чи неявні команди очікування в тестові сценарії, на відміну від Selenium. Cypress автоматично очікує на команди та твердження.
- Розробники або тестувальники можуть використовувати шпигуни, заглушки та годинники для перевірки та контролю поведінки відповідей сервера, функцій або таймерів.
- Операція автоматичного прокручування гарантує, що елемент у полі зору перед виконанням будь-якої дії (наприклад, натискання кнопки)
- Раніше Cypress підтримував лише тестування Chrome. Однак завдяки останнім оновленням Cypress тепер підтримує браузери Firefox і Edge.
- Коли програміст пише команди, Cypress виконує їх у реальному часі, забезпечуючи візуальний зворотний зв'язок під час виконання.
- Cypress має чудову документацію.

Недоліки:

- Не можна використовувати Cypress для керування двох браузерів одночасно
- Він не підтримує кілька веб-сторінок одночасно
- Cypress підтримує лише JavaScript для створення тестів
- На даний момент Cypress не підтримує такі браузери, як Safari та IE.
- Обмежена підтримка iFrames

Selenium

Selenium це популярний інструмент для автоматизації тестування, який допомагає автоматизувати веб-браузери. Цей інструмент містить у собі відкритий вихідним код і є провідним вибором для тестувальників вже більше десяти років.

Це дозволяє тестувальникам автоматизувати тестові сценарії для потрібного браузера за допомогою бібліотеки Selenium WebDriver разом із мовною структурою.

Розробники та спеціалісти з контролю якості також мають можливість вибирати мову програмування на свій смак. Розробники Selenium розробили мовні прив'язки для кількох мов, таких як Ruby, Python, Java тощо. [16]

WebDriver використовує дротовий протокол JSON для виконання тестів. На високому рівні виконання включає три основні етапи:

1. Тестові команди транслюються в URL
2. Драйвери веб-переглядача отримують ці URL-адреси за допомогою HTTP-сервера
3. URL-адреси пересилаються як запит до фактичних браузерів, і всі команди в тестових сценаріях виконуються.

Переваги:

- Сумісний з кількома ОС, такими як Windows, Linux, Unix, Mac
- Надає тестувальникам гнучкість у виборі мови програмування за власним вибором, наприклад Java, Ruby, Python тощо.
- Сумісний із сучасними браузерами, такими як Safari, Chrome, Firefox тощо.
- Надає стислі API

Недоліки:

- Немає вбудованої команди для автоматичної генерації результатів тесту
- Важко впоратися із завантаженням сторінки або елемента
- Обмежена підтримка тестування зображень
- Створення тестових випадків займає багато часу
- Складно налаштувати тестове середовище порівняно з Cypress

3.3 Реалізація тестування сервісно-орієнтованої системи «Інтернет-магазин»

3.3.1 Модульне тестування

Почати тестування сервісно-орієнтованої системи, в першу чергу, потрібно з модульного типу тестування. Розроблена система має два сервіси:

- Товар
- Кошик

Перш за все, слід почати тестування з модулю Товари. Ідея першого сценарію закладається в тому, щоб створити товар.

Приклад тесту:

```
it.only('Module testing - Product create', () => {
    cy.visit('http://localhost:8080/swagger-
ui/index.html#/product-controller/createProduct')
    cy.get(".try-out").click()
    cy.get(".body-param").clear().type('{"number": 3,"name":
"tulip","description": "red tulip","type": "PIECE","unit":
"PIECE","quantity": 100,"price": 10}', {
parseSpecialCharSequences: false })
    cy.get(".opblock-control__btn").click()
    cy.get(".highlight-code").should("contain", '"name":
"tulip"')
```

Результатом такого тесту є вдало створений товар.

Рисунок 3.4 - Результат проведення тесту «Створення товару»

Наступний сценарій – перевірити можливість редагування вже створеного товару. В попередньому випадку у нас був товар із назвою тюльпан, але ми хочемо конкретизувати, який це тюльпан, наступний приклад коду:

```
 it.only('Module testing - Product update', () => {
    cy.visit('http://localhost:8080/swagger-
ui/index.html#/product-controller/updateProduct')
    cy.get(".try-out").click()
    cy.get(".highlight-code").should("contain", '"name":
"string"')
    cy.get(".parameters-
col_description").find("input").clear().type("3d69031a-ae71-
4644-9a59-5d0a0c14071b")
    cy.get(".body-param").clear().type('{"number": 3,"name":
"pink tulip","description": "tulip","type": "PIECE","unit":
"PIECE","quantity": 100,"price": 10}', {
parseSpecialCharSequences: false })
    cy.get(".opblock-control__btn").click()
    cy.get(".highlight-code").should("contain", '"name": "pink
tulip"')
  })
```

І результатом цього сценарію буде змінена назва у товарі.

Рисунок 3.5 - Результат проведення тесту «Модифікація товару»

Наступний сценарій пропонує нам подивитися, які взагалі є товари у системі і в якій кількості.

```
it.only('Module testing - Product get', () => {
    cy.visit('http://localhost:8080/swagger-
ui/index.html#/product-controller/getAllProducts')
    cy.get(".try-out").click()
    cy.get(".opblock-control__btn").click()

  })
```

І результат роботи тесту:

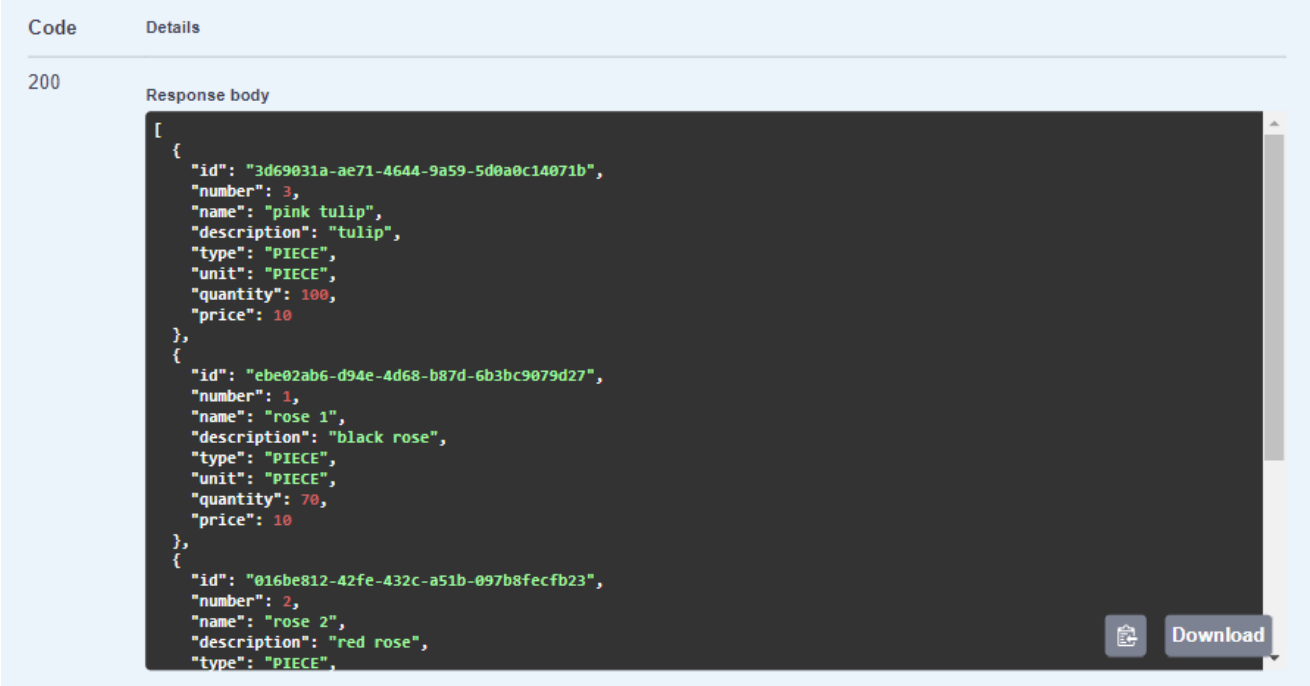

Рисунок 3.6 - Результат проведення тесту «Витяг усіх існуючих товарів»

Останній сценарій для завершення модульного тестування – це видалення товару.

```
it.only('Module testing - Product delete', () => {
  cy.visit('http://localhost:8080/swagger-
ui/index.html#/product-controller/deleteProduct')
    cy.get(".try-out").click()
    cy.get(".parameters-
col_description").find("input").clear().type("3d69031a-ae71-
4644-9a59-5d0a0c14071b")
    cy.get(".opblock-control__btn").click()

  })
```

І результат:

Рисунок 3.7 - Результат проведення тесту «Видалення товару»

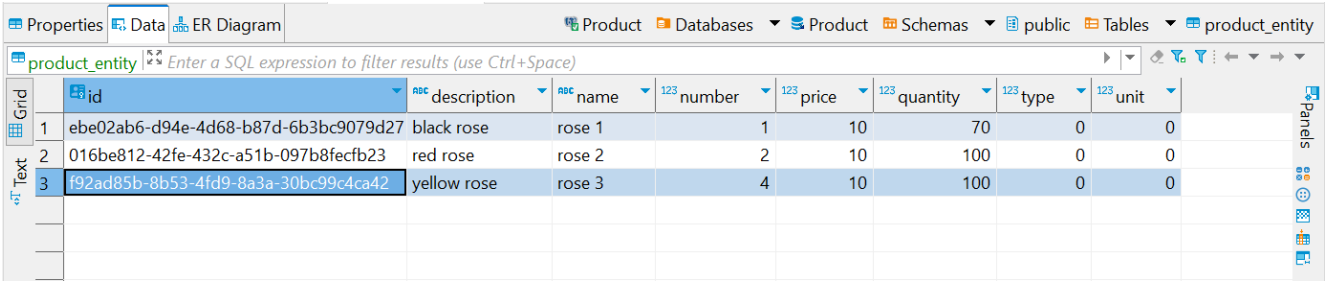

Рисунок 3.8 - Зображення з бази даних для перевірки того, що товар дійсно видалився

Закінчивши модульне тестування, можна зробити деякі висновки.

Модульне тестування спеціалізується на перевірці ключових функцій сервісу. Авжеж, можна розробити безліч сценаріїв для модульного тестування, але мета цієї кваліфікаційної роботи проаналізувати існуючі методи тестування, в даному випадку це : модульне, інтеграційне та регресійне тестування та зробити висновок, які методи більше підходять для тестування сервісно-орієнтованої архітектури і як саме потрібно їх використовувати. [17]

Після атестації першої частини системи – помилок в програмному забезпечені було не знайдено, але це не означає, що їх немає, адже один із принципів тестування каже про те що «Вичерпне тестування неможливе» і «Тестування показує наявність помилок, а не їх відсутність».

Отже, модульне тестування корисне для того, щоб протестувати основні функції окремого сервісу. Тому атестація сервісно-орієнтованих систем не може бути повною без цього типу тестування. Якщо розглядати час та складність розробки сценаріїв, то можна прийти до висновку, що модульне тестування – найлегше серед запропонованих методів.

3.3.2 Інтеграційне тестування

Інтеграційне тестування відбувається з метою перевірити, як взаємодіють сервіси між собою. У випадку системи «Інтернет-магазин», сервіс товарів був протестований завдяки модульному тестуванню, тому атестувати другий сервіс допоможе інтеграційне тестування. Тож, перший сценарій для цього виду тестування полягає у тому, що треба раніше створений товар покласти у корзину. Маємо наступні кроки:

```
it('Integration testing - Adding product to Cart', () => {
    cy.visit('http://localhost:8082/swagger-ui/index.html#/cart-
controller/createCart')
    cy.get(".try-out").click()
    cy.get(".body-param").clear().type('{"number": 4,"products":
[{"productId": "f92ad85b-8b53-4fd9-8a3a-
30bc99c4ca42","orderedQuantity": 10}]}', {
parseSpecialCharSequences: false })
    cy.get(".opblock-control__btn").click()
    cy.get(".highlight-code").should("contain", '"productId":
"f92ad85b-8b53-4fd9-8a3a-30bc99c4ca42"')
  })
```

І результат цього сценарію є на наступному рисунку.

Рисунок 3.9 - Результат проведення тесту «Додавання товару в корзину»

Наступний сценарій дозволяє перевірити можливість додання у вже створену корзину нових товарів.

```
it.only('Integration testing - Updating product in the Cart', ()
=> {
    cy.visit('http://localhost:8082/swagger-ui/index.html#/cart-
controller/addProductsToCart')
    cy.get(".try-out").click()
    cy.get(".parameters-
col_description").find("input").clear().type("0193e10f-2249-
4f27-b57d-d2405ac64aaf")
    cy.get(".body-param").clear().type('{"number": 4,"products":
[{"productId": "6cd05436-42e9-4f82-a4ab-
0ad847a97b69","orderedQuantity": 10}]}', {
parseSpecialCharSequences: false })
    cy.get(".opblock-control__btn").click()
    cy.get(".highlight-code").should("contain", '"productId":
"6cd05436-42e9-4f82-a4ab-0ad847a97b69"')
  })
```

І маємо результат:

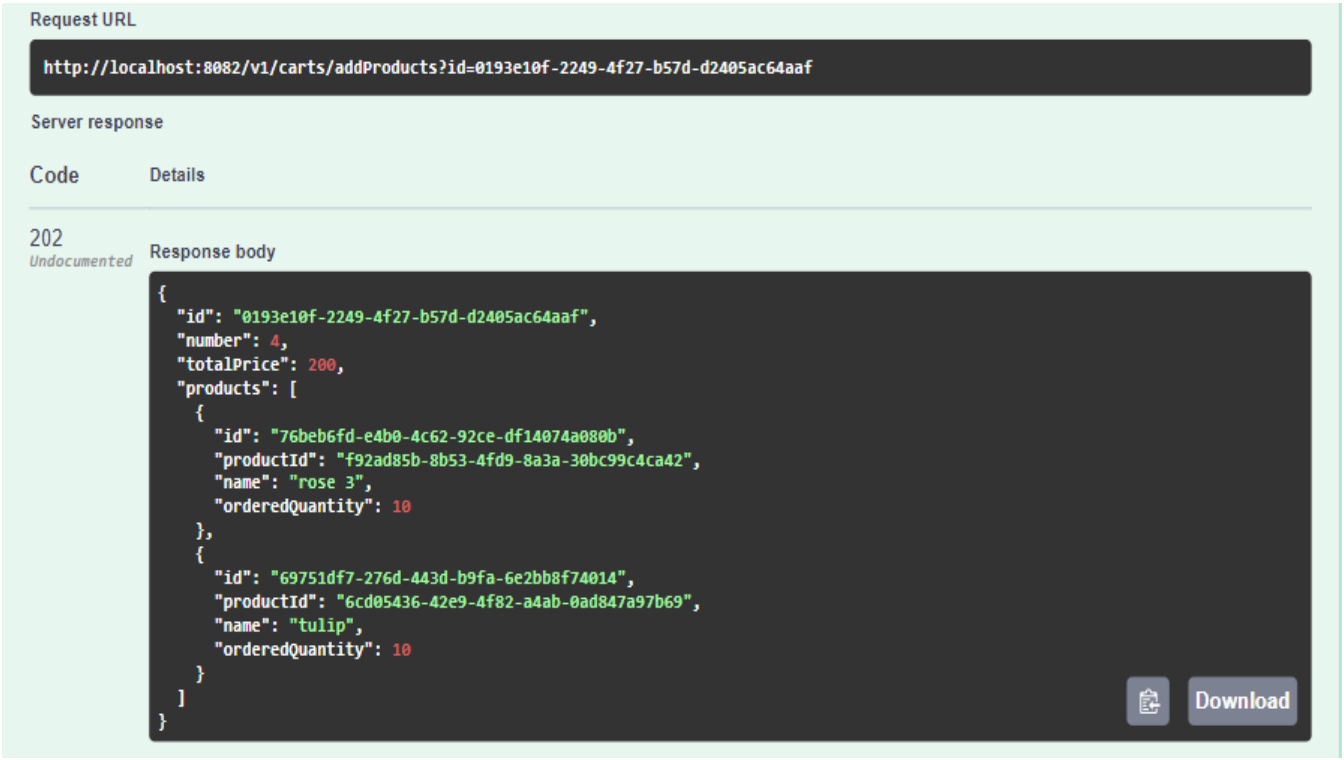

Рисунок 3.10 - Результат проведення тесту «Додавання у вже створену корзину нових товарів»

Як видно з малюнку результатом є два додані товари у раніше створену корзину.

Наступний тестовий сценарій це остання функція, яка завершує цикл – видалення корзини. Тому маємо:

```
it.only('Integration testing - Delete the Cart', () => {
    cy.visit('http://localhost:8082/swagger-ui/index.html#/cart-controller/deleteProduct')
    cy.get(".try-out").last().click()
    cy.get(".parameters-col_description").find("input").clear().type("0193e10f-2249-4f27-b57d-d2405ac64aaf")
    cy.get(".opblock-control__btn").click()
```

І відповідь на тест на наступному рисунку:

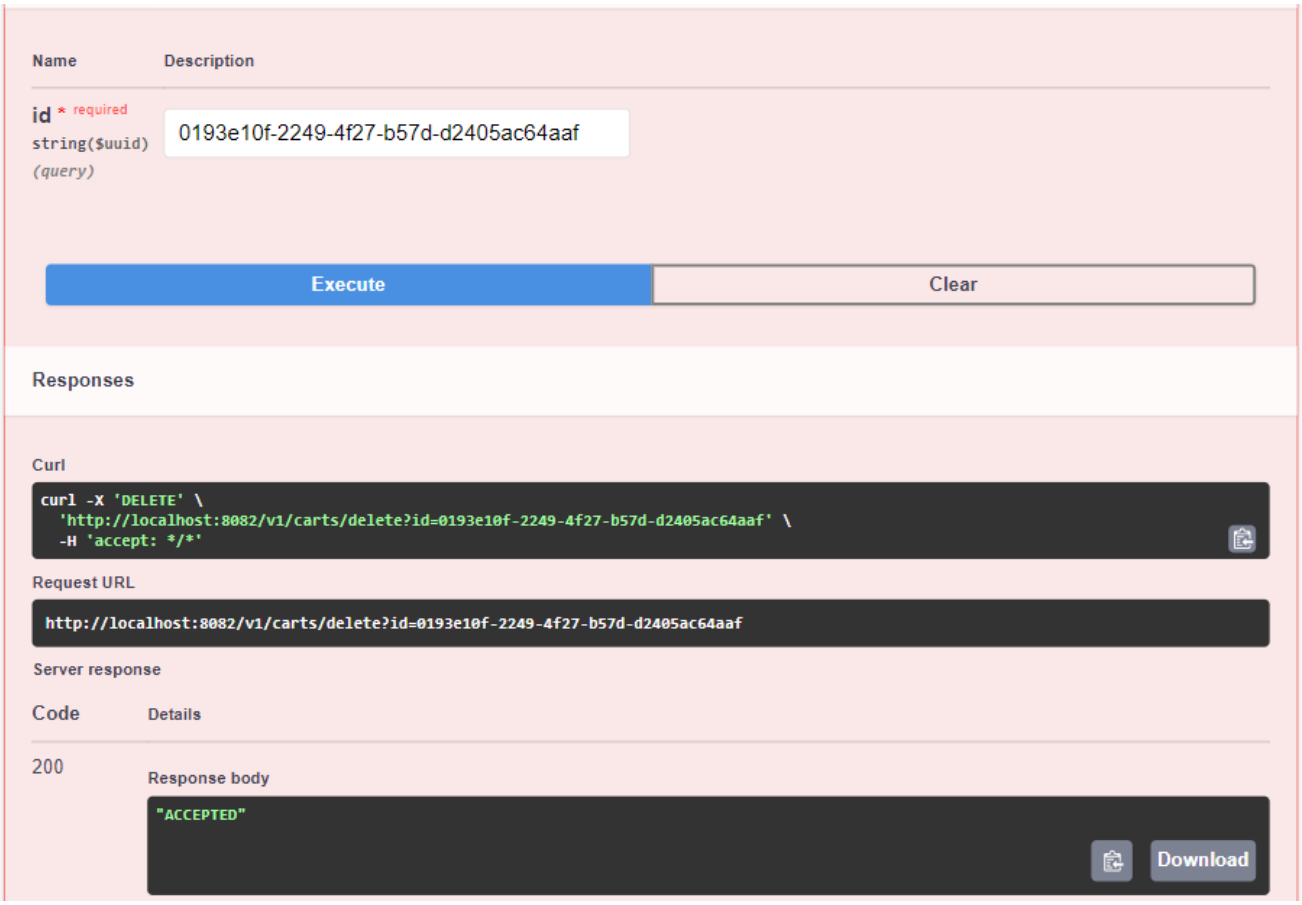

Рисунок 3.11 - Результат проведення тесту «Видалення створеної корзини»

Після інтеграційного тестування можна зробити деякий висновок.

Інтеграційне тестування є складнішим за модульне, тому що в ньому тестується взаємодія одразу двох сервісів. Але в даному випадку, завдяки інтеграційному тестуванню ми перевірили весь функціонал сервісу кошик. Інтеграційне тестування потребує більше часу для створення сценаріїв та самого прогону тестів. Також слід зазначити, що в даних прикладах також не було помічено помилок, але це не означає, що їх немає, це означає, що треба розробляти більше та складніші сценарії.

Отже, інтеграційне тестування корисно проводити після модульного, саме тому, я обрала таку послідовність дій: модульне −> інтеграційне −> регресійне тестування.

Інтеграційне тестування потребує більше часу, ніж модульне, але воно допомагає атестувати і другий сервіс і їх взаємодію.

3.3.3 Регресійне тестування

Регресійне тестування проводиться в вже протестованих модулях системи. Але, як правило, саме регресійне тестування допомагає знайти найбільшу кількість помилок.

Тож, маємо такий сценарій:

1. Створюємо новий товар
2. Переходимо на сервіс Корзини
3. Додаєм раніше створений товар у корзину (Що дозволяє нам взагалі створити корзину)
4. Видаляємо товар

Додаємо до цього тест:

```
it.only('Regression testing - All product flow', () => {
    cy.visit('http://localhost:8080/swagger-
ui/index.html#/product-controller/createProduct')
    cy.get(".try-out").click()
    cy.get(".body-param").clear().type('{"number": 1,"name":
"tulip","description": "Violet tulip","type": "PIECE","unit":
"PIECE","quantity": 988,"price": 11}', {
parseSpecialCharSequences: false })
    cy.get(".opblock-control__btn").click()
    cy.get(".highlight-code").should("contain", '"name":
"tulip"')

    cy.get(".language-
json").find("span").eq(4).invoke('text').then((text) =>{
      cy.visit('http://localhost:8082/swagger-
ui/index.html#/cart-controller/createCart')
      cy.get(".try-out").click()
      cy.get(".body-param").clear().type('{"number":
1,"products": [{"productId": ' + text + ',"orderedQuantity":
100}]}', { parseSpecialCharSequences: false })
      cy.get(".opblock-control__btn").click()
      cy.get(".highlight-code").should("contain", text )
      cy.get("button").contains("/v1/carts/delete").click()
      const id = text.substring(1, 37)
      cy.get(".try-out").last().click()
```

```
      cy.get(".parameters-
col_description").find("input").clear().type(id)
      cy.get(".opblock-control__btn").last().click()
    } )
  })})
```

І маємо такий результат:

Рисунок 3.12 - Створено новий товар

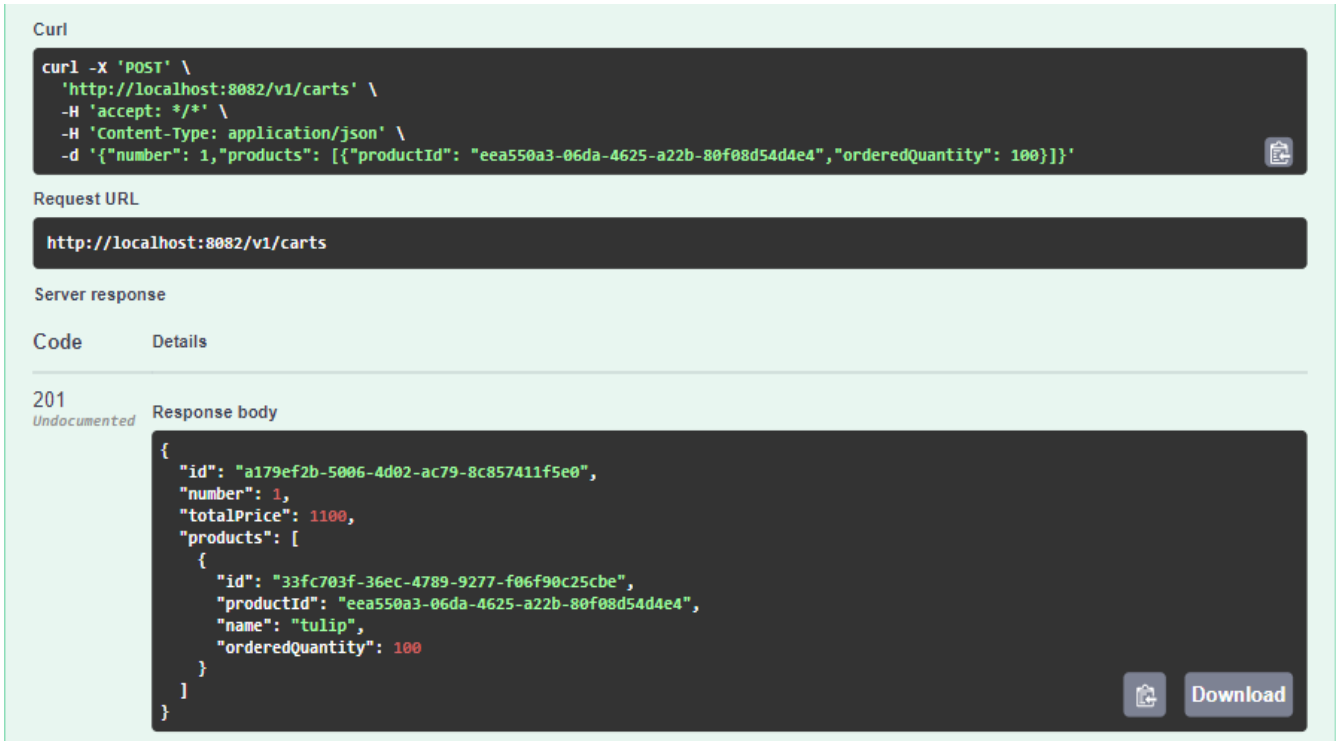

Рисунок 3.13 - Створено нова корзина і додано товар

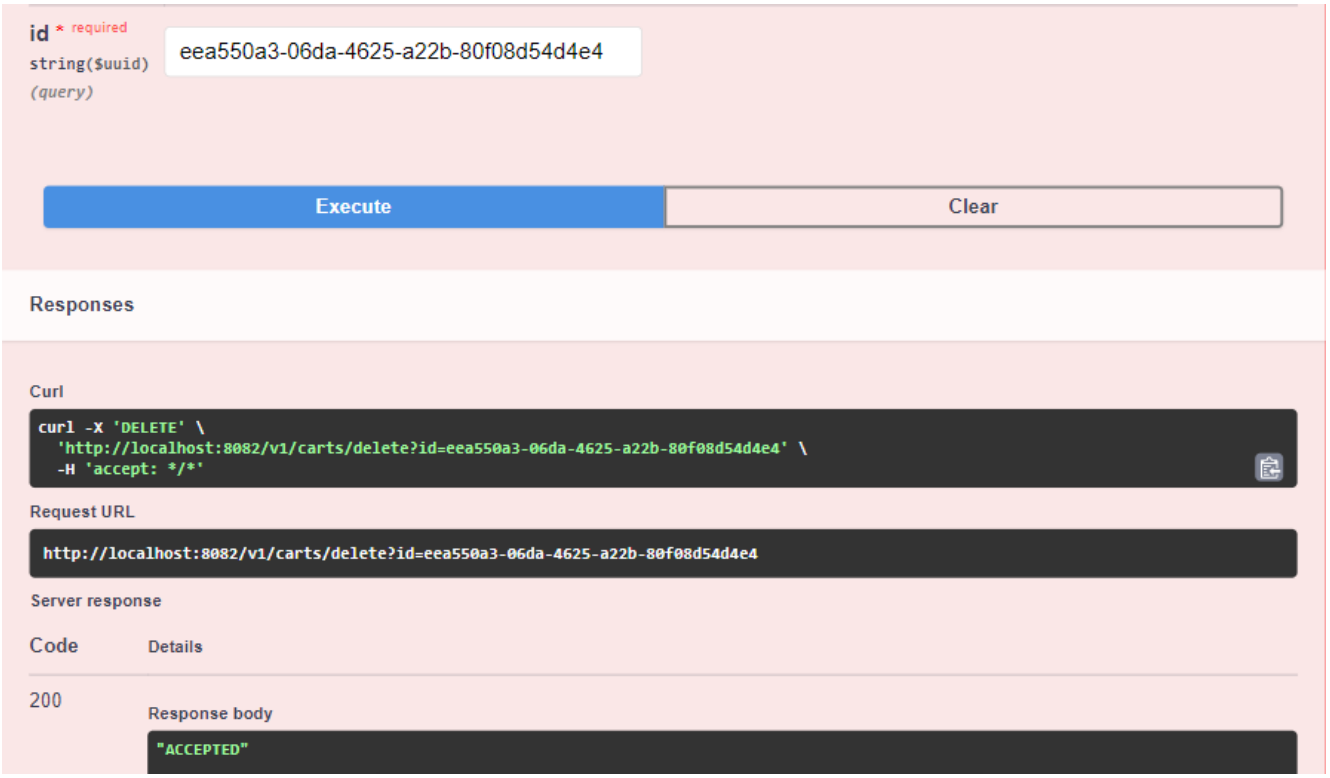

Рисунок 3.14 - Створена корзина видалена

Але слід зауважити, що у поле було введено неправильний ідентифікатор, якщо нам треба, щоб кошик було видалено, то в це поле ми маємо ввести ідентифікатор кошика, а не товару. Тож, виходить, що у поле, де видаляється корзина, було введено ідентифікатор товару і результатом запиту був код 200, що означає вдале видалення. Якщо звернутись до бази даних, то можемо прийти до висновку, що нічого не було видалено.

Тому зараз була знайдена перша помилка у системі. Для того, щоб зафіксувати помилку – потрібно написати невеликий звіт «Bug report», який наведений у таблиці 3.1

Таблиця 3.1 - Bug report

Title	The user recieves the 200 code when tries to delete a cart with wrong id
Priority	High
Issue status	Open
Author	Kharchenko Anastasiia
Steps to reproduce	1. Create a product 2. Create a cart with early created product 3. Specify a wrong id for delete request and try to delete a cart
Expected result	Impossible to delete a cart. Expected code 400 or 500
Actual result	Code 200 but nothing deleted

Після першого тесту, переходимо до другого у якому будемо видаляти товар в правильний спосіб.

Маємо тест:

```
it.only('Regression testing - All product flow (right way)', ()
=> {
  cy.visit('http://localhost:8080/swagger-
ui/index.html#/product-controller/createProduct')
  cy.get(".try-out").click()
  cy.get(".body-param").clear().type('{"number": 2,"name":
"rose","description": "Red rose","type": "PIECE","unit":
"PIECE","quantity": 1000,"price": 10}', {
parseSpecialCharSequences: false })
  cy.get(".opblock-control__btn").click()
  cy.get(".highlight-code").should("contain", '"name": "rose"')

  cy.get(".language-
json").find("span").eq(4).invoke('text').then((text) =>{
    cy.visit('http://localhost:8082/swagger-ui/index.html#/cart-
controller/createCart')
    cy.get(".try-out").click()
    cy.get(".body-param").clear().type('{"number": 2,"products":
[{"productId": ' + text + ',"orderedQuantity": 100}]}', {
parseSpecialCharSequences: false })
    cy.get(".opblock-control__btn").click()
```

```
    cy.get(".highlight-code").should("contain", text )
    cy.visit('http://localhost:8080/swagger-
ui/index.html#/product-controller/deleteProduct')
    const id = text.substring(1, 37)
    cy.get(".try-out").last().click()
    cy.get(".parameters-
col_description").find("input").clear().type(id)
    cy.get(".opblock-control__btn").last().click()
     }) })
```

Отримуємо результат:

Рисунок 3.15 - Товар був створений успішно

Рисунок 3.16 - Створена корзина і в неї покладений товар

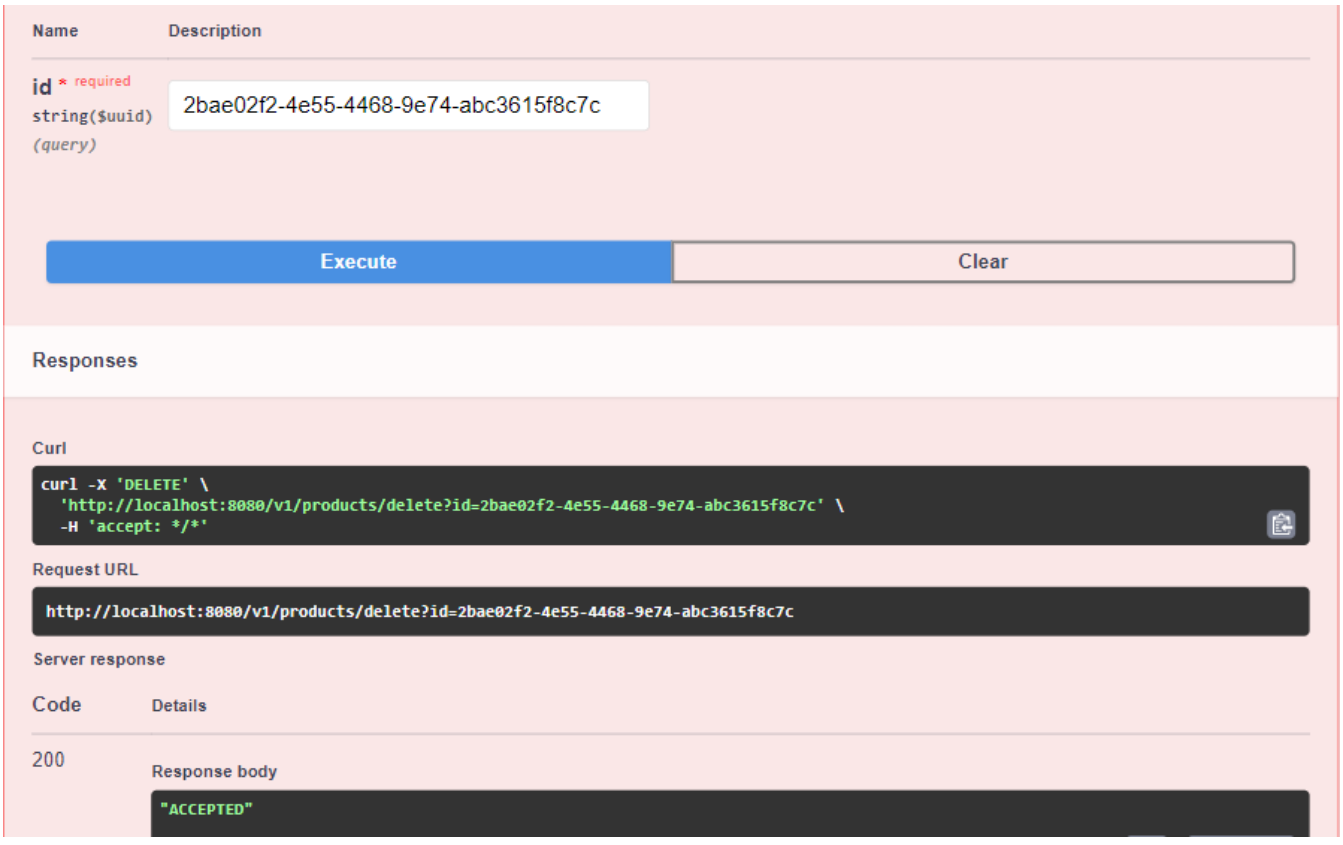

Рисунок 3.17 - Товар був видалений

Наступний тест полягає у тому, щоб перевірити можливість замовлення більшої кількості товару чим є в наявності. За даним сценарієм очікується, що на момент оформлення кошику ми отримаємо негативну відповідь від серверу. Але зможемо видалити товар. Маємо тест:

```
 it.only('Regression testing - All product flow (with high quantity)', () => {
      cy.visit('http://localhost:8080/swagger-ui/index.html#/product-
controller/createProduct')
      cy.get(".try-out").click()
      cy.get(".body-param").clear().type('{"number": 1,"name": "lilia","description":
"white lilia","type": "PIECE","unit": "PIECE","quantity": 10,"price": 10}', {
parseSpecialCharSequences: false })
      cy.get(".opblock-control__btn").click()
      cy.get(".highlight-code").should("contain", '"name": "lilia"')

      cy.get(".language-json").find("span").eq(4).invoke('text').then((text) =>{
        cy.visit('http://localhost:8082/swagger-ui/index.html#/cart-
controller/createCart')
        cy.get(".try-out").click()
        cy.get(".body-param").clear().type('{"number": 41,"products": [{"productId": ' +
text + ',"orderedQuantity": 100}]}', { parseSpecialCharSequences: false })
        cy.get(".opblock-control__btn").click()
        cy.get(".highlight-code").should("contain", text )
        cy.visit('http://localhost:8080/swagger-ui/index.html#/product-
controller/deleteProduct')
        const id = text.substring(1, 37)
        cy.get(".try-out").last().click()
        cy.get(".parameters-col_description").find("input").clear().type(id)
        cy.get(".opblock-control__btn").last().click()
         }) })
```

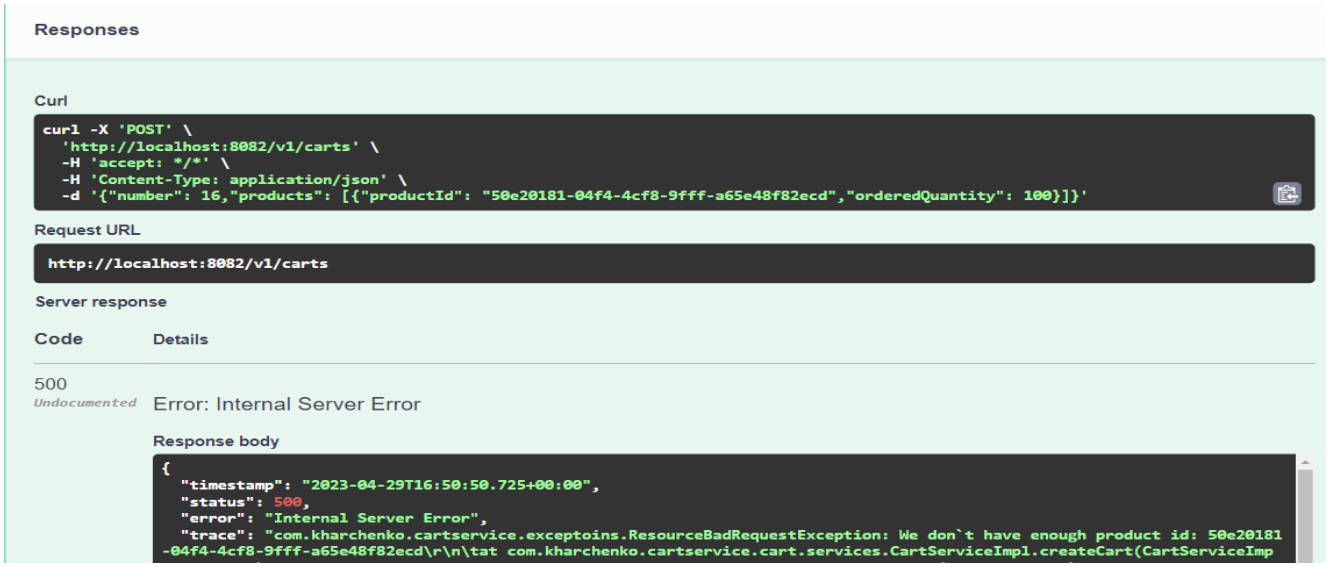

Рисунок 3.18 – Повідомлення про нестачу товару

Отже, коли система готова - виникає необхідність в атестуванні всього можливого функціоналу тому регресійне тестування корисно виконувати після модульного і інтеграційного тестування. Користуючись методом регресійного тестування можна виявити велику кількість помилок, які не були помічені до цього.

Але даний вид тестування найскладніший як з точки зору написання сценаріїв, так і з точки зору виконання тестів. Це пов’язано із тим, що регресійне тестування може зачіпати різні сервіси в одному сценарії.

Тож, можна зробити висновок, що регресійне тестування дуже корисне для атестації таких систем, як сервісно-орієнтовані, і рекомендовано його проводити після значного відрізку часу розробки.

3.3.4 Порівняльний аналіз методів

Модульне тестування використовується для перевірки правильності функціонування модулю системи. В даному випадку, тестування було розбите на три основні сценарії для того, щоб перевірити коректність відпрацювання функцій системи.

Інтеграційне тестування було виконано з метою перевірити взаємозв’язок між сервісами у системі «Інтернет-магазин». Описано три основні сценарії для інтеграційного тестування.

Регресійне тестування було виконано з ціллю об’єднання перевірки виконання функцій модульного тестування та інтеграційного, щоб протестувати систему в цілому. У роботі наведено три сценарії регресійного тестування.

Випробування проводились на комп’ютері з такими технічними характеристиками:

- Windows version 10 Pro
- Processor: Intel(R) Core(TM) i7-8665U CPU @ 1.90GHz 2.11 GHz
- System type: 64-bit Operating System, x64-based processor
- Installed memory (RAM): 16,0 GB

Для порівняльного аналізу було наведено такі критерії:

Порівняльний аналіз наведений у таблиці 3.1

Таблиця 3.1 – Порівняльний аналіз методів тестування

№	Модульне тестування	Т (сек)	С*
1	Створення товару	4	3
2	Модифікація товару	4	4
4	Видалення товару	2	3
№	Інтеграційне тестування	Т	С*
1	Створення кошику	5	3
2	Додавання товару корзину	4	3
3	Видалення корзини	2	2
№	Регресійне тестування	Т	С*
1	Створення товару для інтернет-магазину, додавання товару в кошик, видалення товару	11	5
2	Створення товару для інтернет-магазину, додавання товару в кошик, видалення кошику	10	5
3	Створення товару, додавання товару у кошик із замовленою кількістю більшою ніж є у наявності	8	4

*оцінка здійснюється за 5-ти бальною шкалою: 1 – найменша одиниця (найлегше); 5 – найбільша одиниця (найскладніше)

Отримані дані проаналізовано і можна підвести підсумок у вигляді таблиці зі середнім значенням по кожному методу тестування

Таблиця 3.2 – Середня оцінка критеріїв для кожного виду тестування

	Модульне тестування	Інтеграційне тестування	Регресійне тестування
Час виконання (сек)	10	11	29
Трудоємність написання сценарію	10	8	12

Результати аналізу можна візуально зобразити у вигляді таких діаграм:

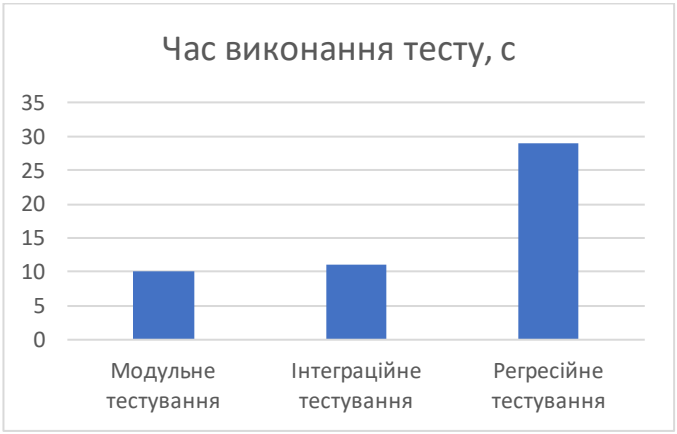

Рисунок 3.19 – Розрахований середній час виконання тесту на кожен вид тестування

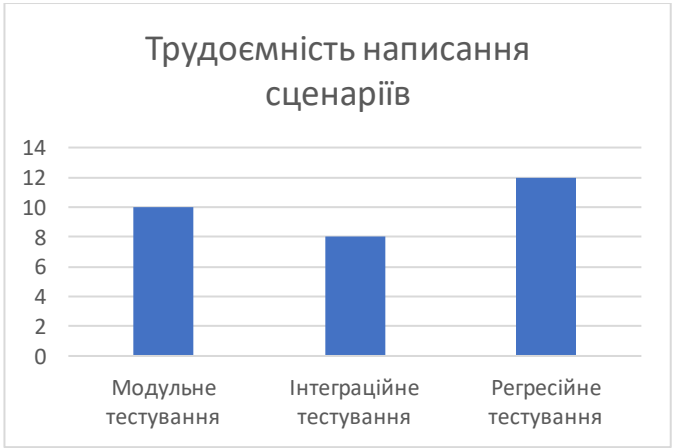

Рисунок 3.20 – Розрахована середня трудоємність написання сценаріїв

ВИСНОВКИ

У даній роботі досліджено: розподілені системи та специфіка їх перевірки, методи атестації програмного забезпечення, етапи атестації систем на базі сервісно-орієнтованої архітектури, методи тестування серверної частини системи, інструменти для виконання автоматизованих тестів.

Виявлено такі переваги сервісно-орієнтованої архітектури як гнучкість та клієнтоорієнтованість, надійність, можливість масштабування і доступності, можливість інтегрування різних систем та програм для покращення обміну даними. Також виявлено недоліки такої архітектури: складність спілкування сервісів між собою, для кожного сервісу потрібна окрема база даних, складність тестування.

У ході роботи змодельована і розроблена серверна частина системи «Інтернет-магазин» задля UI оформлення API було додано Swagger-UI. Спроєктована множина тестів різних типів для розробленої системи та проведено тестування. Здійснено порівняння різних типів тестування для серверної частини клієнт-серверної системи за критеріями: час виконання тесту, трудомісткість проєктування тестів.

До перспектив подальшої розробки варто віднести розширення функціональності системи додатковими сервісами та вдосконалення методики атестації для сервісно-орієнтованих систем.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Харченко А. Методи атестації розподілених систем на базі сервісно-орієнтованої архітектури : матеріали XXI міжнар. наук.-прак. конф. "Шевченківська весна - 2023", м. Київ, 14 квітня 2023 р. С. 113..
2. Порівняння мікросервісної та монолітної архітектур [Електронний ресурс] Режим доступу до ресурсу: https://www.atlassian.com/ru/microservices/microservices-architecture/microservices-vs-monolith.
3. Andrew S. Tanenbaum Distributed Operating Systems 1994, 588 с.
4. Лавріщева К.М. ПРОГРАМНА ІНЖЕНЕРІЯ.–К.– 2008, 319 с.
5. Специфікація вимог [Електронний ресурс] Режим доступу до ресурсу: https://argondigital.com/blog/product-management/requirements-vs-specifications.
6. Омельчук Л.Л. Формальні методи специфікації програм. – К.: УкрІНТЕІ, 78 с.
7. Ioana Rodhe, Martin Karresand - Overview of formal methods in software engineering December 2015, 45 p.
8. Formal and Informal Specifications of a Secure System Component: Final Results in a Comparative Study T. M. Brookes 1, J. S. Fitzgerald ~, P. G. Larsen 227 p.
9. А. В. Лучкова - Аналіз методів верифікації програмного забезпечення 3 с.
10. Тестування програмного забезпечення [Електронний ресурс]. Режим доступу до ресурсу: https://www.wiki-data.uk-ua.nina.az.
11. History of software testing [Електронний ресурс]. Режим доступу до ресурсу: https://www.geeksforgeeks.org/history-of-software-testing/.

12. Thomas Erl. Service-Oriented Architecture: Analysis and Design for Services and Microservices. December 2016, 416 p.
13. Test Strategies for SOA [Електронний ресурс] Режим доступу до ресурсу: https://www.browserstack.com/guide/test-strategies-for-soa-applications.
14. Rest API testing [Електронний ресурс]. Режим доступу до ресурсу: https://www.geeksforgeeks.org/rest-api-testing-and-manual-test-cases/.
15. HTTP status code [Електронний ресурс]. Режим доступу до ресурсу: https://moz.com/learn/seo/http-status-codes.
16. Selenium vs Cypress [Електронний ресурс]. Режим доступу до ресурсу: https://www.browserstack.com/guide/cypress-vs-selenium.
17. Types of software testing [Електронний ресурс]. Режим доступу до ресурсу: https://hackr.io/blog/types-of-software-testing.