Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

Всебічне тестування проекта з мікросервісною архітектурою на основі фреймворку Spring

Текстова частина до курсової роботи
за спеціальністю „Прикладна математика” - 113

Керівник курсової роботи
доктор техн. наук, доцент
Глибовець А.М.
_____
(підпис)
“____”_____202_ р.

Виконав студент
ПМ-3 Катрич К. В.
“____” _____ 202_ р.

Київ 2021

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав.кафедри інформатики,
проф., д.ф.-м.н.
_____ С. С. Гороховский
(підпис)
“____”_____202_ р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту Катричу Костянтину Вячеславовичу факультету інформатики 3 курсу
ТЕМА: Всебічне тестування проекта з мікросервісною архітектурою на основі фреймворку Spring
Вихідні дані:

- Підходи до тестування проектів з мікросервісною архітектурою

Зміст ТЧ до курсової роботи:

Індивідуальне завдання
Вступ
1. Огляд проекту для ознайомлення
2. Розробка схеми тестування
3. Написання тестів
4. Опис написаних тестів та практичної частини

Висновки
Список джерел
Додатки

Дата видачі “___” _____ 202_ р. Керівник _____
(підпис)

Завдання отримав _____
(підпис)

Календарний план виконання курсової роботи
Тема: Всебічне тестування проекта з мікросервісною архітектурою на основі фреймворку Spring

№ п.п	Назва етапу курсового проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	Листопад 2020 р.	
2.	Огляд літератури за темою роботи	Листопад-грудень 2020 р.	
3.	Опрацювання матеріалів щодо інтеграційного тестування	Грудень 2020 р.	
4.	Опрацювання матеріалів щодо навантажувального тестування	Січень 2021 р.	
5.	Створення супутніх проектів з використанням двох методів тестування	Лютий-березень 2021 р.	
6.	Написання пояснювальної роботи	Березень 2021 р.	
7.	Створення слайдів для доповіді та написання доповіді	Квітень 2021 р.	
8.	Надання роботи керівнику для перевірки	Квітень 2021 р.	
9.	Корегування роботи за результатами перевірки керівником	Травень 2021 р.	
10.	Остаточне оформлення пояснювальної роботи та слайдів	Травень 2021 р.	
11.	Подання роботи на кафедру для перевірки на плагіат	Травень 2021 р.	
12.	Захист курсової роботи	Травень 2021 р.	

Студент _____Катрич К. В._____ _____

Керівник _____Глибовець А. М._____ _____

“_____”_____202_р

Зміст

Анотація.....5
Вступ.....6
Розділ 1: Що таке тестування?
1.1Для чого потрібне тестування?.....7
1.2Різновиди тестування.....7
1.3Підходи до тестування.....10
Розділ 2: Аналіз програми перед написанням тестів до неї
2.1 Особливості мікросервісної архітектури.....13
2.2 MindMap проекта.....14
2.3 Framework Spring.....15
Розділ 3: Рівні тестування
3.1 Unit тести.....16
3.2 Component тести.....18
3.3 End-to-end тести.....22
Висновки.....24
Список використаної літератури.....25

Анотація

В цій роботі йдеться про різноманітні типи та підходи до тестування проекті з мікросервісною архітектурою на основі фреймворку Spring. Тут детально розглянуто рівні такого тестування та можливі корисні анотації до кожного окремого розділу.

Вступ

За останні 20 років професія програміста пройшла довгий шлях. Кожен день з'являються нові програмні забезпечення та комп’ютерні ігри, мови програмування та фреймворки, сайти та блокчейн валюта, перерахувати все майже неможливо. Різні програмні застосування почали займати значно більше місця в повсякденному житті людини. Звичайно разом з кількістю коду збільшилися і вимоги до нього. В першу чергу вимоги до надійності коду, впевненості у тому що в важливий момент часу програма не зламається та завжди буде робити те, що від неї очікуються.

Бути впевненим у надійності навіть простої програми майже неможливо. Кожен може щось забути чи просто трішки помилитись, та вся система може впасти коли в середині важливого процесу. Неявні помилки можуть не помічати навіть декілька рокі, для їх активації будуть потрібні спеціальні умови.

Кількість будь-яких помилок зростає разом з кожним рядком нового коду, особливо в великому проекті. Якщо відтермінувати тестування, то помилки будуть все частіше з’являтися. Починаючи з найменших ділянок коду помилки будуть розростатися і переходити все далі й далі, бо програміст не буде знати, що там щось не так, як треба.

Ще складніше провести тестування не одного великого, а декількох окремих проектів, які повинні працювати разом, як один налагоджений організм. Гарним прикладом може слугувати проекти з мікросервісною архітектурою. Окремі частини зазвичай пишуть різни люди, можливо навіть на різних мовах програмування. Сервіси весь час повинні синхронізуватися та за цим треба слідкувати. Помилка може статися на будь-якому моменті в будь-якій частині окремого сервісу. Загалом все дуже складно та заплутано.

Цією роботою я хочу провести дослідження того, як треба тестувати системи на основі такої архітектури, які є підводні каміння, корисні прийоми та в

загалом вирішення проблем з якими зустрінеться кожен.

Розділ 1: Що таке тестування?

1.1 Для чого потрібне тестування?

Тестування програмного забезпечення [8] - це процес перевірки системи з метою виявлення будь-яких помилок, прогалин або відсутніх вимог у порівнянні з фактичними вимогами.

Навіть простий додаток може бути піддано під велику кількість і різноманітність тестів. План управління тестуванням допомагає розставити пріоритети, які типи тестування становлять найбільшу цінність з урахуванням доступного часу і ресурсів. Ефективність тестування оптимізується за рахунок виконання найменшого числа тестів для виявлення найбільшого числа дефектів.

1.2 Різновиди тестування

Розглянемо види тестування [3]. Існує багато різних класифікацій з своїми особливостями. Зазвичай всі види класифікують за наступними категоріями:

1) По об’єктах які піддаються тестуванню.
2) За ступенем знання тестованої системи.
3) За ступенем ізольованості частини компонентів тестованого ПЗ.
4) За ступенем глибини тестування.
5) За часом коли проводиться тестування.
6) За зовнішніми ознаками позитивності сценаріїв.
7) За критеріями запуску програми або програмного коду.
8) За ступенем підготовки до тестування.
9) Інтуїтивне тестування.

Самою великою категорією вважається «По об’єктах які піддаються тестуванню». Складається вона що най менше з 9 видів тестування. Проаналізуємо декілька з них.

Функціональне тестування

Процес заснований на заздалегідь відомій поведінки користувача, заснований в першу чергу на детальному аналізі та вивченні функціональної специфікації додатку, системи або невеликого модуля.

Тестувальник забезпечення проводячи функціональне тестування виходять зі свого особистого досвіду, документації до проекту. Зі спілкування з представниками замовника і своєю командою передбачає, як буде себе вести користувач й на основі цього робить ті ж дії, перевіряючи додаток.

Тестування локалізації

Локалізація - процес адаптації програмного продукту до мови і культури клієнта. Для успішної реалізації продукту в усі необхідні країни, потрібно приділяти увагу не тільки технічного перекладу елементів інтерфейсу на мову країни-покупця, а й багатьом іншим моментам. Саме значення слів, технологій, розміщення кнопок, текстових полів, зображень повинно виглядати гарно. Не повинно виникнути проблем зі сприйняттям кінцевим клієнтом продукту.

Проаналізуємо й іншу категорію тестування, а саме про «За ступенем знання тестованої системи». Поділяється вона на тестування чорної, білої та сірої коробки.

Чорна коробка - ми не знаємо, як влаштована тестована система.

Тестування методом «чорноної коробки», також відоме як тестування яке засноване на документації або тестування поведінки. Техніка тестування заснована на роботі виключно з зовнішніми інтерфейсами системи.

Переваги:

- Тестування проводиться з позиції кінцевого користувача і може допомогти виявити неточності і суперечність в специфікації.
- Тестувальника нема потреби знати мови програмування і заглиблюватися в особливості реалізації програми.
- Тестування може проводитися фахівцями, незалежними від відділу розробки, що допомагає уникнути упередженого ставлення.
- Можна починати писати тест-кейси, як тільки готова специфікація.

Недоліки:

- Тестується тільки дуже обмежена кількість шляхів виконання програми.
- Без чіткої специфікації досить важко скласти ефективні тест-кейси.
- Деякі тести можуть виявитися надмірними, якщо вони вже були проведені розробником на рівні модульного тестування.

Біла коробка - нам відомі вся реалізації системи що тестується.

Тестування методом білої коробки - тестування програмного забезпечення, який передбачає, що реалізація системи повністю відома. Оскільки нам відомий весь код, ми можемо обирати вхідні дані для тестів, які будуть оброблювати критичні точки, а також ми будемо знати який повинен бути результат таких тестів.

Переваги:

- Тестування може проводитися на ранніх етапах. Нема потреби чекати

створення призначеного для користувача інтерфейсу.

- Можна провести більш ретельне тестування, з покриттям великої кількості шляхів виконання програми.

Недоліки:

- Для виконання тестування необхідна велика кількість знань з області.
- При використанні автоматизації тестування на цьому рівні, підтримка тестових скриптів може виявитися досить накладної, якщо програма часто змінюється.

Сіра коробка - нам відомо тільки деякі особливості реалізації тестової системи.

Тестування методом сірої коробки - метод тестування програмного забезпечення, який передбачає комбінацію Білої та Чорної коробки. Тобто внутрішній устрій програми нам відомо лише частково. Передбачається що буде наданий весь доступ до внутрішньої структури коду задля написання важливих тестів, але саме тестування буде проводитися так само, як в методі чорної коробки, з позиції користувача.

1.3 Підходи до тестування.

З різної літератури я дізнався, що існує багато різноманітних підходів до тестування. Але деякі методологій зацікавили мене більше. Останнім часом все частіше можна почути абревіатуру TDD[2]. TDD або Test Driven Development в перекладі означає «Розробка на основі тестів». Підхід TDD полягає в тому, що перш ніж написати програмний код, програміст спочатку пишете тест, який буде служити специфікацією, тобто визначати, що має робити цей код.

Це надзвичайно потужна концепція розробки програмного забезпечення, але часто її неправильно використовують. Зазвичай застосування концепції «розробка

через тестування» означає використання модульних тестів для управління виробництвом коду програми. Але насправді цей підхід можна застосовувати на будь-якому рівні.

Підхід TDD перевертає все з ніг на голову, та замість того, щоб спочатку писати код, а потім писати модульні тести для перевірки цього коду, спочатку пишуться модульні тести, а вже потім - код, щоб цей тест пройшов. Таким чином, модульне тестування «управляє» розробкою коду.

Цей процес повторюється знову і знову. Ви пишете ще один тест, який визначає більше функціональності того, що повинен робити код. Потім ви пишете і вносите сам код, домагаючись успішного завершення тесту.

Після того, як ви тест пройшов, ви приступаєте до рефакторингу коду, тобто організовуєте або очищаєте його, щоб зробити коротшім та більш зрозумілим.

Переваги TDD перед звичайними методологіями:

- Гарна архітектура коду. Якщо ви хочете, щоб ваш код перевірявся, він повинен мати належну структуру.
- Якщо ви хочете щось змінити у своєму коді, це дуже просто. TDD забезпечує надійність.
- Це вимагає чіткого розуміння коду, щоб програмісти могли зрозуміти його та додати необхідні зміни.

Недоліки:

- Розробка коду йде повільно. Швидкий розвиток коду неможливий, оскільки спочатку нам доводиться писати тестові кейси, але довгострокова розробка дуже швидка.
- Тести важко написати, оскільки код складніший для написання та розуміння.
- Іноді нам потрібна вся команда розробників, для написання тестових кейсів.

- Може бути багато непорозумінь.

Існує схожа за значенням концепція BDD. BDD або Behaviour Driven Development - «Розробка через поведінку». Це гнучка методологія, тісно пов'язана з TDD[1]. На перший погляд BDD важко виокремити від TDD: обидва підходи припускають написання документації і тестів до старту етапу розробки. Відмінність тільки у тому, що в BDD для опису тестів використовують природню мову, зрозумілу кожному учаснику проекту, що, фактично, об'єднати постановку задачі, тести і документацію воєдино.

Після детального аналізу, я виокремив переваги та недоліки BDD перед TDD

Переваги:

- Тести читаються не тільки програмістами.
- Їх легко змінювати. Вони часто пишуться майже чистою англійською.
- Результати виконання тестів більш "людяні".
- Тести не залежать від цільової мови програмування. Міграція на іншу мову сильно спрощується.

Але недоліків теж багато:

- Створення сценаріїв та ведення файлів потребує великих зусиль та часу.
- Потрібне гарне спілкування всередині компанії.
- Інтерпретування тесту у кожного може бути своє, а для тесту який буде максимально точним потрібно багато зусиль.

Щоб позбутися недоліків, опис можна нескінченно уточнювати, але це не вихід. Найбільш однозначним і точним описом тесту, як це не дивно, буде його програмний код, що приводить нас знову до TDD.

Розділ 2: Аналіз програми перед написанням тестів до неї

2.1Особливості мікросервісної архітектури

Прочитавши багато літератури по темі мікросервісної архітектури проектів, я визначив наступне позначення для цього [4]. Мікросервісна архітектура - це спосіб структурування системи, при якій кілька незалежних сервісів спілкуються один з одним певним чином (зазвичай допомогою вебсервісів). Ключова особливість полягає в тому, що кожен мікросервіс здатний оновлюватися і розвиватися незалежно від інших.

Як можна здогадатися з назви мікросервісна архітектура складається з багатьох окремих мікросервісів. Для кращого розуміння я порівняти мікросервіс з повною його протилежністю - монолітом.

Особливості мікросервіса:

- Він не великий.
- Він незалежний.
- Він будується навколо бізнес-потреби і використовує обмежений контекст.
- Він взаємодіє з іншими мікросервісами через мережу.
- Мінімальна централізація зверху.
- Процеси його розробки та підтримки потребують автоматизації.
- Ітераційний розвиток.

У мікросервісной архітектури повністю відсутня загальної бази даних. Вони заснований на системі персистентності, яка буде єдиною для управління і використання.

Переваг у такого підходу однозначно багато. Основними звичайно є відмовостійкість. Якщо якась база випадково впаде, то буде втрачено тільки частина даних, замість того щоб втратити відразу все. Також не треба забувати про куди кращу масштабованість. Згодом в БД монолітної архітектури, в силу нових змін і доповнень, з'являються якісь милиці і хаос. Все через те, що основну структуру БД поміняти майже нереально, і якщо знадобитися щось інше, то прийдеться вигадувати як ці зміни. У той час як у децентралізованої бази даних немає таких гострих проблем.

2.2 MindMap проекта.

Задля підготовки перед написанням тестів, я знайшов та опрацював спосіб кращого розуміння тестової системи названий Mind Map [5]. Mind Map або інтелектуальна карта - це інструмент для візуального відображення інформації, який допомагає ефективно її структурувати. Така форма викладу інформації простіше для розуміння людським мозком, ніж рядковий текст, і від того простіше для застосування в роботі.

Переваги Mind Map:

- Наочність і візуалізація. Головною гідністю Mind Map для тестувальника є наочне бачення тестованого продукту, його функцій і залежностей між собою.

- Відмінна альтернатива документації. Таку карту дуже добре демонструвати новим співробітникам як альтернативу або доповнення до документації.

- Легко підтримувати. З виходом нових функцій її нескладно доповнити і знову відстежити взаємозв'язки нових частин додатка, можливо навіть виявити де продукт можна зробити простіше і зрозуміліше користувачеві.

- З виходом нових функцій її нескладно доповнити і знову відстежити взаємозв'язки нових частин додатка, можливо навіть виявити де продукт

можна зробити простіше і зрозуміліше користувачеві.

2.3 Spring Framework

Головною задачею в цій роботі було ознайомлення з фреймворком Spring. Цей фреймворк займає передові позиції у сфері програмування на Java вже дуже значний час. Перша стабільна версія була видана в 2004-му році. Він надає добре документовані і легкі у використанні засоби вирішення проблем, що виникають при створенні додатків корпоративного масштабу.

Для себе я склад наступне означення цього фреймворка [6]: Spring Framework являє собою просто контейнер впровадження залежностей, з декількома зручними шарами (доступ до бази даних, проксі, аспектно-орієнтоване програмування, RPC, вебінфраструктура MVC). Це все дозволяє вам швидше і зручніше створювати Java-додатки.

Spring Framefork працює за принципом IoC (Inversion of Control, з англійської - Інверсія контролю), також відоме як Dependency Injection [7]. Це є процесом, згідно з яким об'єкти визначають свої залежності з іншими об'єктами з якими вони працюють через аргументи конструктора, фабричного методу або властивості. Потім контейнер впроваджує ці залежності при створенні Bean. Цей процес досить незвичайний, названий він Inversion of Control, через те що Bean сам контролює реалізацію і розташування своїх залежностей.

Для правильної роботи треба налаштовувати ApplicationContext - об'єкт який успадковує інтерфейс BeanFactory, який займається всіма залежностями, і додає більше специфічну функціональність. Налаштувати його треба через спеціальну конфігурацію, яка має багато різних параметрів.

Спочатку конфігурація в Spring була доступна виключно за допомогою XML-файла, але, починаючи з версії Spring 2.5, стала можлива конфігурація за допомогою анотацій. Завдяки цьому, ми можемо пов'язувати між собою модулі вставивши анотації безпосередньо в Java-клас. Це значно полегшує роботу з

конфігурацією і збільшує її читабельність.

Функціонал Spring анотацій дуже широкий, завдяки одній анотації можна під'єднати будь-яку базу даних в проект, налаштувати API або створити сервіс. Творці цього фреймворку розуміють що в ньому повинне бути зручне налагодження тестів до коду. Spring надає широкий спектр інструментів для створення тестів.

Розділ 3: Рівні тестування

3.1 Unit тести

Ознайомлення з тестами правильно починати з найнижчого рівня тестування – модульного тестування [9]. Зона відповідальності у таких тестах дуже маленька, охоплюють вони зазвичай один або декілька методів декількох класів і перевіряються в них досить низькорівневі алгоритми.

Для таких тестів не треба завантажувати весь Spring Context. Зазвичай це дуже довгий процес, поки фреймворк знайде потрібну конфігурацію, поки впровадить все залежності пройде багато часу, а таких модульних тестів може бути дуже багато.

Замість використання Spring ви можете створювати екземпляр класу для тестування і, при необхідності, скористатися імітацією бібліотеки, щоб ізолювати тестований екземпляр від його залежностей. Наприклад, якщо у нас є сервіс, анотований як Spring сервіс, який виконує деякі обчислення і звертається на якийсь репозиторій для отримання даних, то це може бути протестовано без підняття всього фреймворку.

Мною був досліджена зручна бібліотка Mockito, завдяки якій можна досить зручно налаштовувати імітацію деяких репозиторіїв. Можна налаштовувати умови і результати імітації.

Приклад:

```
@Test
public void exampleTest() {
    Mockito.when(dataService.getDataItemById("idValue"))
            .thenReturn("dataItem");
}
```

Також це можна зробити через анотації:

```
@MockBean
private DataService dataService;

@Test
public void exampleTest() {
    given(this.dataService.someCall()).willReturn("dataItem");
}
```

Крім того, можна використовувати @SpyBean для імітації сервісу. У той час як Mock повністю переписує логіку Bean, Spy – за замовчуванням буде виконувати оригінальну поведінку методів об'єкта, але їх поведінку можна перевизначити.

Також Mockito може працювати з JNDI. JNDI, або ж Java Naming and Directory Interface, являє собою Java API для доступу до служб імен та каталогів. Пакет містить часткову реалізацію JNDI SPI, який ви можете використовувати, щоб створити просту JNDI середу для тестових наборів або автономних додатків. Ми можемо імітувати об'єкти Servlet API, призначені для використання з інфраструктурою Spring Web MVC, які корисні для тестування вебконтекстів і контролерів. Ці фіктивні об'єкти зазвичай більш зручні у використанні, ніж динамічні фіктивні об'єкти або присутні фіктивні об'єкти API.

```
@Autowired
 private WebApplicationContext wac;

 private MockMvc mockMvc;

 @Before
 public void setup() {
   this.mockMvc = webAppContextSetup(this.wac).build();
 }

 @Test
 public void getFoo() throws Exception {
   this.mockMvc.perform(get("/get").accept("application/json"))
       .andExpect(status().isOk())
       .andExpect(content().mimeType("application/json"))
       .andExpect(jsonPath("$.name").value("data"));
 }
```

З літератури я дізнався про ReflectionTestUtils. Це набір корисних методів, завдяки якому ви можете змінити значення константи, встановити непублічне поле, викликати метод, що не є загальнодоступним, або викликати метод, який не є загальнодоступним, при тестуванні коду програми на такі випадки використання, як такі:

- Фреймворки ORM, що попускають приватний або захищений доступ до полів, на відміну від методів загальнодоступного set() для властивостей у сутності домену.
- Підтримка Spring для анотацій (таких як @Autowired, @Inject та @Resource), які забезпечують введення залежностей для приватних або захищених полів, методів встановлення та методів конфігурації.
- Використання таких анотацій, як @PostConstruct та @PreDestroy для методів зворотного виклику життєвого циклу.

3.2 Component тести

Далі я зайнявся інтеграційними тестами. Як визначає Вікіпедія, інтеграційне тестування - це фаза тестування, при якому окремі програмні модулі об'єднуються та тестуються в групах. Інтеграційне тестування як вхідних даних використовує модулі, за допомогою яких було проведено модульне тестування,

групує їх у більші множини, виконує тести, визначених в плані тестування для цих багатьох множин та представляє їх для подальшого системного тестування.

Spring Framework забезпечує відмінну підтримку інтеграційного тестування. У цьому нам допоможе бібліотека org.springframework.test.

Вона включає пакет, який містить цінні класи для інтеграційного тестування з контейнером Spring. Це тестування не залежить від сервера додатків або іншого середовища розгортання.

Підтримка інтеграційного тестування Spring переслідує наступні основні цілі:

- Управління кешуванням контейнерів Spring IoC між тестами.
- Забезпечення впровадження залежностей примірників тестових пристосувань.
- Забезпечення управління транзакціями, відповідне інтеграційного тестування.

Найчастіше інтеграційні тести представляють зв'язку декількох сервісів, сервісу з API або сервісу з базою даних. Для кожного типу існує багато допоміжних анотацій. Про пару сервіс-API було згадано в модульному тестуванні. З літератури я дізнався що при тестуванні декількох сервісів найскладніше це правильно задати Spring Context, тому я вирішив зануритися в це питання.

Spring TestContext Framework забезпечує послідовне завантаження ApplicationContext примірників і WebApplicationContext примірників Spring, а також кешування цих контекстів. Підтримка кешування завантажених контекстів важлива, тому що час запуску може стати проблемою - не через накладні витрати самого Spring, а тому, що об'єкти, екземпляри яких створюються контейнером Spring, вимагають часу для створення екземплярів. Наприклад, для проекту з 50-100 файлами може знадобитися від 10 до 20 секунд для завантаження файлів зіставлення, і ці витрати перед запуском кожного тесту.

За замовчуванням після завантаження налаштоване значення ApplicationContext повторно використовується для кожного тесту. Таким чином, витрати на установку оплачуються тільки один раз для кожного набору тестів, і подальше виконання тестів відбувається набагато швидше.

Але навіть тут можна прискорити процес завантаження контексту. Завдяки анотації @ContextHierarchy в нас може бути кілька контекстів, які поділяють відносини батько-нащадок. Ієрархія контексту дозволяє декільком дочірнім контекстам спільно використовувати компоненти, які знаходяться в батьківському контексті. Кожен дочірній контекст може перевизначити конфігурацію, успадковану від батьківського контексту.

Тут варто відзначити що у контексту може бути тільки один батьківський елемент і кілька дочірніх. Крім того, дочірній контекст може звертатися до Bean-компонентів в батьківському контексті, але не навпаки.

Приклад:

```
@ExtendWith(SpringExtension.class)
@WebAppConfiguration
@ContextHierarchy({
    @ContextConfiguration(classes = RootConfig.class),
    @ContextConfiguration(classes = ChildConfig.class)
})
class ControllerIntegrationTests {}
```

Для пар сервіс-БД використовують анотацію @DataJpaTest. За замовчуванням він налаштує вбудовану базу даних в пам'яті, сканує @Entity класи і налаштовує репозиторії Spring Data JPA. Звичайні @Component Bean не завантажуються в ApplicationContext.

Тести JPA даних за замовчуванням є транзакційними і відкочуються в кінці кожного тесту. Якщо це ви цього не хочете, то можете відключити управління транзакціями для тесту або для всього класу наступним чином:

```
@RunWith(SpringRunner.class)
@DataJpaTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public class ExampleTests {}
```

Вбудовані бази даних в пам'яті зазвичай добре підходять для тестів, оскільки вони швидкі і не вимагають установки розробника. Однак, коли необхідно працювати з реальною базою даних, тоді треба використовувати @AutoConfigureTestDatabase анотацію:

```
@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace=Replace.NONE)
public class ExampleTests {}
```

Існує аналогічна анотація тільки для взаємодії з JDBC (Java DataBase Connectivity) - @JdbcTest

Досить зручно можна тестувати й не реляційною базу даних MongoDB завдяки анотації @DataMongoTest

```
@RunWith(SpringRunner.class)
@DataMongoTest
public class ExampleDataMongoTests {

    @Autowired
    private MongoTemplate mongoTemplate;

}
```

Для того, щоб запустити частину додаток треба використовувати @SpringBootTest (webEnvironment = WebEnvironment.MOCK). Вона завантажує Context вебдодатка і надає імітацію вебсередовища. Він не завантажує справжній http-сервер, просто імітує поведінку всього веб-сервера.

WebEnvironment.MOCK дає вам деякі переваги, такі як простота використання або ізоляція інших факторів, але це може бути не дуже хорошою практикою тестування інтеграції.

3.3 End-to-end тести

В останню чергу мною були розглянуті End-to-End тести. E2E - це метод тестування, за допомогою якого тестується весь програмний продукт від API до самої бази даних, щоб гарантувати, що всі додаток працює правильно. Він визначає системні залежності продукту і гарантує, що всі інтегровані компоненти, які вже були протестовані раніше, працюють разом, як очікувалося. У такому типі тестування ми імітуємо тільки якісь зовнішні сервери, якщо такі приєднай до нашого проекту.

Основна мета цього тестування полягає в тому, щоб протестувати досвід кінцевого користувача шляхом моделювання реального автоматизоване і перевірки тестованої системи і її компонентів на предмет інтеграції і цілісності даних.

Є два види такого тестування:

- Горизонтальне тестування E2E
  Часто використовується метод, застосовуваний горизонтально в контексті декількох додатків і легко реалізується в одному додатку. Приклад: вебдодаток системи електронної комерції включає облікові записи, стан товарних запасів і відомості про доставлення.
- Вертикальне тестування E2E
  Використовується для тестування критичних компонентів складної обчислювальної системи, в якій зазвичай не беруть участь користувачі або інтерфейси. Цей метод належати до пошарового тестування, що означає, що тести виконуються в послідовному ієрархічному порядку. Щоб гарантувати якість, кожен компонент системи або продукту тестується від початку до кінця.

E2E тести повинні бути максимально схожі на виробниче середовище. Для того, щоб визначити порт для нашого застосування можна використовувати @SpringBootTest (webEnvironment = WebEnvironment.RANDOM_PORT) або

WebEnvironment.DEFINED_PORT. При використані цього тестується реальний http-сервер зі своїм певним портом, для налагодження якого треба використовувати TestRestTemplate.

В тестах типу E2E дуже відповідальна роль, вони повинні знайти глобальні проблеми перед тим як запустити на виробництві всю систему разом.

Мною також були розглянуті Альфе та Бета тестування, які йдуть після E2E тестів. Полягати вони в тому. щоб тестувати кінцевий продукт на справжніх клієнтах.

Висновки

В цій роботі було проаналізовано кілька видів тестування програмного забезпечення, програми на основі мікросервісной архітектури, фреймворку Spring , а також безліч його анотації. Було порівняно різновиди тестування за ступенем знань внутрішньої тестованої системи. Так само було проаналізовано різні рівні тестування системи, кожен з них був зрозуміло описаний. Разом з рівнями тестування були надані і описані корисні анотації фреймворку. Більш того були розглянуті різні бази даних, які треба було протестувати.

Під час дослідження було прочитана необхідна література з різноманітних сайтів, форумів, та документація фреймворку Spring. Так само було проведено дослідження безпосередньо під час виконання практичної частини курсової роботи.

В роботі наведені причини, чому тестування всього програмного забезпечення є обов'язковими.

Отже, зі збільшенням кількості програм на мікросервісній архітектурі, їх тестування стає все більш обов'язковим. Відсутність помилок, як на найнижчих рівнях, так й на рівні всієї системи є обов'язковою умовою для стабільної і продуктивної роботи коду. Оскільки кожен програміст відповідальний за продукт який виробляє, то вважаю що мінімальні знання з тестування повинні бути у кожного.

Список джерел

1. [Електронний ресурс] стаття “BDD vs TDD: Pros and Cons” https://www.fleekitsolutions.com/bdd-vs-tdd-pros-cons/

2. [Електронний ресурс] стаття “Introduction to Test Driven Development (TDD)” http://www.agiledata.org/essays/tdd.html
3. [Електронний ресурс] стаття “Види тестування ПЗ стаття” https://qaevolution.ru/testirovanie-po/vidy-testirovaniya-po/

4. [Електронний ресурс] сторінка “Мікросервісна архітектура” https://ru.wikipedia.org/wiki/%D0%9C%D0%B8%D0%BA%D1%80%D0%BE%D1%81%D0%B5%D1%80%D0%B2%D0%B8%D1%81%D0%BD%D0%B0%D1%8F_%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0
5. [Електронний ресурс] стаття “Карта MindMap” https://blog.checkiant.com/ru/blog-o-produktivnosti/166-tekhnologiya-mind-mapping
6. [Електронний ресурс] стаття “Що таке Spring Framework” https://habr.com/ru/post/490586/
7. [Електронний ресурс] сторінка “Spring Framework” https://ru.wikipedia.org/wiki/Spring_Framework
8. [Електронний ресурс] сторінка “Software testing” https://en.wikipedia.org/wiki/Software_testing
9. [Електронний ресурс] сторінка “Spring Boot features” https://docs.spring.io/spring-boot/docs/1.5.2.RELEASE/reference/html/boot-features-testing.html