Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

Сучасні підходи до інтеграційного та навантажувального тестування на базі Spring

Текстова частина до курсової роботи
за спеціальністю „Комп'ютерні науки та інформаційні технології" - 122

Керівник курсової роботи
доктор техн. наук, доцент
Глибовець А.М.

_____

(підпис)

"____"_____2020 року

Виконав студент КНІТ-4
Нікітченко Я.Ю.
"____"_____2020 року

Київ 2020

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Зав.кафедри інформатики,

доц., к.ф.-м.н.

_____С. С. Гороховський

(підпис)

“____”_____201_ р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту _Нікітченку Ярославу_____

__4_____курсу факультету інформатики

ТЕМА: __Сучасні підходи до інтеграційного та навантажувального тестування на базі Spring_____

Вихідні дані:

- Підходи до інтеграційного та навантажувального тестування на базі Spring — сучасний підхід до тестування веб-застосунків

Зміст ТЧ до курсової роботи:

Вступ
1. Загальні відомості щодо тестування програмного забезпечення
2. Основні принципи і засади інтеграційного тестування
3. Основні принципи і засади навантажувального тестування
4. Опис практичної частини роботи
Висновки
Список джерел
Додатки (за необхідністю)

Дата видачі “___”_____201_ р.

Керівник_____ Завдання отримано _____

Календарний план виконання курсової роботи

Тема: Сучасні підходи до інтеграційного та навантажувального тестування на базі Spring

№ п/п	Назва етапу курсового проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	листопад 2019р.	
2.	Огляд літератури за темою роботи	листопад-грудень 2019 р.	
3.	Опрацювання матеріалів щодо інтеграційного тестування	грудень 2019 р.	
4.	Опрацювання матеріалів щодо навантажувального тестування	січень 2020 р.	
5.	Створення супутніх проектів з використанням двох методів тестування	лютий-березень 2020 р.	
6.	Написання пояснювальної роботи	березень 2020р.	
7.	Створення слайдів для доповіді та написання доповіді	березень 2020 р.	
8.	Надання роботи керівнику для перевірки	березень 2020 р.	
9.	Корегування роботи за результатами перевірки керівником	березень 2020 р.	
10.	Остаточне оформлення пояснювальної роботи та слайдів	березень 2020 р.	
11.	Подання роботи на кафедру для перевірки на плагіат	квітень 2020 р.	
12.	Захист курсової роботи	квітень 2020 р.	

Студент Нікітченко Я. _____

Керівник Глибовець А. М. _____

“ _____” _____**р**

Зміст

Анотація

В цій роботі йдеться про сучасні методи перевірки якості програмних веб-застосунків на базі Spring. Також детально розглядається інтеграційне та навантажувальне тестування, і детально описується хід виконання кожного з них на практиці.

Вступ

Тестування — це широкий процес, який складається з декількох взаємопов’язаних процесів. Іншими словами сукупності процесів.

Часто говорячи про тестування, люди уявляють собі картинку, в якій спеціаліст перевіряє чи за всіма параметрами програма працює ідеально. Але якість не є абсолютною, це суб'єктивне поняття. Тому тестування не може повністю забезпечити коректність програмного забезпечення. Воно тільки порівнює стан і поведінку продукту зі специфікацією.

Тестування пронизує весь життєвий цикл ПЗ, починаючи від проектування і закінчуючи невизначено довгим етапом експлуатації. Ці роботи безпосередньо пов'язані із завданнями управління вимогами та змінами, адже метою тестування є якраз можливість переконатися у відповідності програм заявленим вимогам.

Отож мета і завдання моєї роботи - це розглянути різноманітні підходи до тестування програмного забезпечення, провести детальний опис кожного та продемонструвати роботу кожного з них.

Ця робота складається з 4 розділів, за допомогою яких в кінцевому результаті утвориться загальна картина особливостей інтеграційного та навантажувального тестування.

В першому розділі розглядається загальні відомості щодо тестування програмного забезпечення. В ньому детально розписано, що таке загалом тестування, навіщо воно існує та як його використовують в сучасності.

В другому розділі розповідається про особливості інтеграційного тестування. Також, в ньому розповідається про основні особливості даного виду та вказуються переваги та недоліки.

В третьому розділі відбувається детальний опис навантажувального тестування. Наводяться особливості даного виду, його переваги та недоліки.

В четвертому розділі описується приклад реалізації тестування веб-застосунку на базі Spring двома вище вказаними видами тестування.

РОЗДІЛ 1: Загальні відомості щодо тестування програмного забезпечення

1.1 Що таке тестування?

Тестування програмного забезпечення є невід'ємною частиною створення програмного продукту. Від того, наскільки досконало проведені тести, залежить те, як скоро проект буде зданий остаточно, і чи буде необхідність згодом усувати помилки.

Тестування - це процес технічного дослідження, який виконується на вимогу замовників, і призначений для вияву інформації про якість продукту відносно контексту, в якому він має використовуватись. До цього процесу входить виконання програми з метою знайдення помилок. Зазвичай, поняття якості обмежується такими поняттями, як коректність, повнота, безпечність, але може містити більше технічних вимог [2].

Тестування так само можна описати як процес верифікації та валідації того чи іншого програмного продукту, щоб дізнатися на скільки точно він задовольняє всім встановленим вимогам.

Верифікація [1] — процес оцінки системи і її компонентів з метою співставлення результатів поточного етапу розробки, початково сформованим умовам. Тобто, виконання заданих завдань, цілей, строків по розробці продукту.

Валідація [1] — це визначення відповідності ПЗ, що розробляється, очікуванням і потребам користувача, а також вимогам до системи.

Тестоване програмне забезпечення повинно проходити кожен з етапів тестування, обумовлених власниками продукту, менеджментом компанії розробника та тест-дизайнерами для того, щоб вважати програмний продукт відносно якісним або придатним до використання.

Таким чином, існує велика кількість видів тестування, якими можна перевірити якість проекту та переконатися, що все виконано згідно зазначених вимог.

1.2 Тестування “чорної та білої скриньки”

Головними методами тестування програм вважаються: тестування “білої” та “чорної скриньки”.

У випадку “білої скриньки” об'єктом тестування тут є не зовнішня, а внутрішня поведінка програми. Під час користування даним методом спеціаліст попередньо знає код і готує відповідний випадок.

Етапи роботи [10]:

1) Перший крок - зрозуміти вихідний код.
2) Другий крок – опанування навичок безпечного кодування.
3) Третій крок включає в себе тестування внутрішніх функцій програми, тому розуміння коду має велике значення.
4) Четвертий крок - написати код для перевірки програми або підготувати певні тестові випадки з відповідними даними.

Техніки тестування “білої скриньки” [10]:

1) Аналіз покритого коду. Ця техніка усуває прогалини в наборі тестових випадків шляхом визначення програми, яка не може бути досліджена жодними тест-кейсами. Крім того, можна створювати приклади для неперевіреної частини програми, які покращать якість програмного забезпечення.
2) Тестування тверджень. Ця методика перевіряє кожне твердження коду хоча б один раз протягом тестового циклу.
3) Тестування рішень / гілок. Ця методика перевіряє кожен можливий варіант у коді, цикли If-else та інші умовні цикли програмного забезпечення.

У випадку методу “чорної скриньки” функціональність програми перевіряється, не дивлячись на внутрішню структуру коду, деталі реалізації та знання внутрішніх шляхів програмного забезпечення.

Цей тип тестування повністю заснований на вимогах та технічних характеристиках програмного забезпечення. У даному методі ми просто

зосереджуємось на даних, що ми вводимо, та на результатах, які ми отримуємо, не переймаючись внутрішніми знаннями програмної програми.

Таким чином головним місцем даного методу є інтерфейс програми.

Етапи роботи [11]:

1) Спочатку вивчаються вимоги та технічні характеристики системи.
2) Тестер обирає позитивні та негативні тестові випадки, щоб переконатися, що програма їх обробляє правильно.
3) Відбувається запуск програми з визначеними сценаріями.
4) Тестер програмного забезпечення порівнює фактичні результати з очікуваними.
5) Виправлення дефектів, якщо такі наявні, та повторення попередніх кроків.

Техніки тестування чорної скриньки [11]:

1) Тестування класів еквівалентності: застосовується для мінімізації кількості можливих тестових випадків до оптимального рівня при збереженні розумного покриття тесту.
2) Тестування граничної величини: орієнтоване на значення на межах. Ця методика визначає, чи певний діапазон значень прийнятний системою чи ні. Це дуже корисно для зменшення кількості тестових випадків. Він найбільш підходить для систем, де вхід знаходиться в певних діапазонах.
3) Тестування таблиці рішень: розміщує причини та їх наслідки в матриці. У кожному стовпчику є унікальне поєднання.

Отже, підсумовуючи всю вище наведену інформацію, обидва методи є невід’ємною частиною розробки програмного продукту.

Тестування “білої скриньки”, як правило, виконується розробниками під час перевірки написаного коду, а “чорна скринька” використовується при проведенні тестування інтерфейсу та працездатності застосунку, перед передачею його кінцевим користувачам.

1.3 Необхідність тестування в розробці програмного забезпечення

Під час користування будь-якою програмою ми навіть не замислюємося про те, чого та чи інша програма працює без помилок, зависань тощо. Саме в цьому аспекті тестування відіграє далеко не останню скрипку.

Ось кілька причин важливості даного процесу [1]:

1) Тестування дозволяє перевірити, чи правильно реалізовано усі вимоги до програмного забезпечення, що розроблялось.
2) Тестування допомагає у виявленні помилок та забезпечує їх розпізнавання і вирішення до етапу розгортання програмного забезпечення.
3) Тестування пом’якшує наслідки та ризики втрат, якщо програмний продукт все ж випустили по неправильним вимогам. В такому випадку намагаються частково виправити, переоцінити та покращити програму.
4) Тестування допомагає перевірити належну інтеграцію та взаємодію програми з навколишнім середовищем.

Звідси основна мета тестування програмного забезпечення, яка полягає у намаганні виміряти рівень якості програмного продукту з точки зору основних вимог.

Люди схильні помилятися, людські помилки можуть призводити до порушення нормальної роботи програмного забезпечення на всіх стадіях розробки, причому наслідки цього можуть бути найрізноманітнішими — від незначних до катастрофічних.

Для програмного забезпечення будь-якої складності неможливо вилучити всі проблеми. Проте тестування допомагає виявити більшість помилок, з якими може зіткнутися користувач, й потенційно зменшує ризик виникнення проблем з програмою у майбутньому.

1.4 Види тестування

У залежності від переслідуваних цілей види тестування можна умовно розділити на наступні типи [2]:

- Функціональні.
- Нефункціональні.
- Пов'язані зі змінами.

Функціональні тести базуються на функціях та особливостях, а також на взаємодії з іншими системами, і можуть бути представлені на всіх рівнях тестування: компонентному або модульному, інтеграційному, системному і приймальному. Такі види перевірки розглядають зовнішню поведінку системи. Одні з найпоширеніших видів функціональних тестів [2]:

- Тестування безпеки - стратегія тестування, що використовується для перевірки безпеки системи, а також для аналізу ризиків, пов'язаних із забезпеченням цілісного підходу до захисту програми, атак хакерів, вірусів, несанкціонованого доступу до конфіденційних даних. Воно може виконуватися як автоматизовано так і в ручну, включаючи перевірку як позитивних, так і негативних тестових випадків.
- Тестування взаємодії - це функціональне тестування, що перевіряє здатність програми взаємодіяти з одним і більше компонентами або системами і включає в себе тестування сумісності та інтеграційне тестування.

Нефункціональне тестування описує тести, необхідні для визначення характеристик програмного забезпечення, які можуть бути виміряні різними величинами. У цілому, це перевірка того, як система працює. Далі перераховані основні види нефункціональних тестів [2]:

- Тестування продуктивності, яке включає в себе тестування навантаження, стресове тестування, тестування стабільності або надійності та об'ємне тестування.

- Тестування установки, яке спрямоване на перевірку успішної інсталяції та настройки, а також оновлення або видалення програмного забезпечення.
- Тестування зручності користування - це метод тестування, спрямований на встановлення ступеня зручності використання, навченості, зрозумілості та привабливості для користувачів розроблюваного продукту в контексті заданих умов.
- Тестування на відмову і відновлення, яке перевіряє тестований продукт з точки зору здатності протистояти й успішно відновлюватися після можливих збоїв, що виникли у зв'язку з помилками програмного забезпечення, відмовами обладнання або проблемами зв'язку.
- Конфігураційне тестування - ще один вид традиційного тестування продуктивності. У цьому випадку замість того, щоб тестувати продуктивність системи з точки зору навантаження, тестується ефект впливу на продуктивність змін у конфігурації.

Після проведення необхідних змін, таких як виправлення дефекту, програма повинна бути протестована для підтвердження того факту, що проблему було дійсно вирішено. Нижче перераховані види перевірок, які необхідно проводити після установки програмного забезпечення, для підтвердження працездатності або правильності здійсненого виправлення помилки [2]:

- Димове тестування. Поняття цього види перевірки пішло з інженерного середовища. При введенні в експлуатацію нового "заліза" вважалося, що тестування пройшло вдало, якщо з установки не пішов дим. В області ж програмування, воно спрямоване на поверхневу перевірку всіх модулів на предмет працездатності та наявність швидко встановлюваних критичних і блокуючих дефектів.
- Регресійне тестування - вид тестування спрямований на перевірку змін, зроблених у додатку або навколишньому

середовищі, для підтвердження того факту, що існуюча раніше функціональність працює як і раніше.

- Тестування збірки, яке спрямоване на визначення відповідності випущеної версії критеріям якості для початку тестування. За своєю метою є аналогом димового тестування, спрямованого на приймання нової версії для подальшої перевірки або експлуатації.
- Санітарне тестування - вузьконаправлене тестування, яке достатнє для доказу того, що конкретна функція працює згідно заявлених у специфікації вимог.

РОЗДІЛ 2. Основні принципи і засади інтеграційного тестування

2.1 Що таке інтеграційне тестування?

Інтеграційне тестування це такий тип тестування, коли програмні модулі інтегруються послідовно і перевіряються як група. Типовий програмний проект складається з декількох програмних модулів, кодованих різними програмістами [3].

Сенс інтеграційного тестування досить простий – об’єднати модулі, що перевіряються, в один з метою проведення тестування поведінки як комбінованого підрозділу.

Основна функція або мета цього тестування - перевірити інтерфейси між модулями.

Зазвичай інтеграційне тестування проводиться після тестування кожного блоку. Спочатку кожен модуль буде розглядатися як функціонально-самостійна частина проекту. Проте, окремий блок може викликати проблеми та помилки при взаємодії з іншими підрозділами. Після того, як модулі протестовані, вони поєднуються один з одним.

Таким чином, після того, як всі окремі модулі створені та протестовані, ми поєднуємо їх в один і починаємо виконувати інтеграційне тестування. Це робиться, щоб перевірити комбінаційну поведінку та впевнитись, що правильно виконані усі зазначені вимоги.

Ось чому тестування програмного забезпечення є таким важливим, особливо тестування підрозділів як групи.

Тут ми повинні розуміти, що інтеграційне тестування не відбувається в кінці циклу, скоріше воно проводиться одночасно з розробкою. Отже, у більшості випадків усі модулі фактично не доступні для тестування, і завдання - перевірити те, що буде розроблене.

2.2 Переваги та мета інтеграційного тестування

Розглянемо мету інтеграційного тестування. Ось кілька причин [3]:

- У реальному світі, коли розробляються програми, вони розбивається на більш дрібні модулі, а окремим програмістам присвоюється певний блок. Логіка, реалізована одним розробником, зовсім інша, ніж у іншого. Тому важливо перевірити чи все, що зазначено в вимогам, реалізовано кожним розробником і чи відповідає встановленим стандартам.
- Під час розробки обличчя або структура даних змінюються багато разів, коли вони переходять від одного модуля до іншого. Деякі значення додаються або видаляються, що спричиняє проблеми в пізніших блоках.
- Модулі також взаємодіють з деякими сторонніми інструментами або API, які також потрібно перевірити.

Тепер розглянемо кілька переваг цього тестування, і деякі з них перераховані нижче [3]:

- Це тестування забезпечує правильну роботу інтегрованих модулів або компонентів.
- Інтеграційне тестування можна розпочати, коли не всі модулі для тестування ще будуть доступні. Даний вид не вимагає заповнення кожного блоку, оскільки замість нього можна використати Stubs та драйвери.
- Він виявляє помилки, пов'язані з інтерфейсом.

2.3 Складнощі інтеграційного тестування

Нижче перераховано декілька проблем, які стосуються інтеграційного тестування [3]:

1) Інтеграційне тестування означає перевірку працездатності двох або більше інтегрованих систем, з метою забезпечення належної роботи програми. Таким чином, має бути налаштоване навколишнє середовище, в якому буде проведена перевірки інтегрованої системи.
2) Для тестування інтегрованої системи можуть бути застосовані різні шляхи та перестановки. Таким чином, кожен розробник може перевірити працездатність одного модулю відмінно від розуміння іншого співробітника.
3) Управління інтеграційним тестуванням стає складним через малу кількість факторів, які беруть в ньому участь, такі як база даних, платформа, оточення тощо
4) Інтегруючи будь-яку нову систему зі старою буде необхідно провести численні зміни та регресійне тестування.
5) Інтеграція двох різних систем, розроблених двома різними компаніями, є великою проблемою, оскільки неможливо заздалегідь бути впевненим, як одна із систем вплине на іншу, якщо відбудуться зміни в будь-якій з них.

2.4 Підходи до інтеграційного тестування

Першим розглянемо Big-Bang підхід. В цьому методі всі або більша частина розроблених модулів збираються разом, формуючи завершену програмну систему або її значну частину та використовуються для інтеграційного тестування.

Цей підхід дуже ефективний для збереження часу. Тим не менше, якщо тест-кейси та їх результати не занотовані належним чином, весь процес стане дуже складним та втратить сенс [4].

Переваги підходу Big-Bang:

- Це хороший підхід для малих систем.

Недоліки підходу Big-Bang:

- Важко виявити модуль, який викликає проблему.
- Підхід Big-Bang вимагає наявність всіх модулів для тестування, що, в свою чергу, призводить до меншого часу на тестування, оскільки на розробку та інтеграцію знадобиться більша частина часу.
- Тестування проводиться одразу, що тим самим не залишає часу для ізольованого критичного тестування модулів.

Тестування знизу-вгору — це такий підхід коли першими перевіряються елементи нижнього рівня, щоб в подальшому полегшити тести високорівневих компонентів.

Після інтеграційного тестування з груп компонентів певного рівня формується наступний рівень. Він також повинен бути перевірений. Цей підхід ефективний тільки коли всі модулі однієї частини вже розроблені. Також цей метод допомагає визначити, які рівні системи вже відповідають вимогам та виразити процес тестування в відсотках.

Таким чином компоненти вищого рівня та гілки кожного з блоків перевіряються крок за кроком, поки модуль не буде протестований повністю.

Переваги:

- Можливість простіше виокремити проблемні місця.
- Не витрачається час на очікування розробки всіх модулів на відміну від підходу Big-bang

Недоліки:

- Критичні модулі (на найвищому рівні архітектури програмного забезпечення), які керують потоком програми, тестуються останніми і можуть бути схильні до дефектів.
- Ранній прототип неможливий

Зверху-вниз - це підхід до інтеграційного тестування, коли одиниці вищого рівня тестуються першими, а одиниці нижчого рівня тестуються поетапно після цього. Цей підхід застосовується при дотриманні підходу щодо розвитку вгорі. Тестові заготовки потрібні для імітації одиниць нижчого рівня, які можуть бути недоступними протягом початкових етапів.

Переваги:

- Знаходження проблемних місць.
- Можливість отримати ранній прототип.
- Критичні модулі тестуються за пріоритетом; основні недоліки дизайну можна буде знайти та виправити спочатку.

Недоліки:

- Потрібно багато штрихів.
- Модулі на нижчому рівні тестуються не зовсім коректно.

Сендвіч-тестування — підхід, комбінований з двох попередніх.

Перший підхід дозволяє легко знаходити програмні дефекти, коли другий і третій знаходить архітектурні.

РОЗДІЛ 3. Основні принципи і засади навантажувального тестування

3.1 Основні поняття та визначення

Тестування навантаження - це вид нефункціонального тестування, яке визначає продуктивність системи в умовах навантаження в режимі реального часу. Під час нього ми перевіряємо реакцію системи в різних станах, моделюючи кілька користувачів, які одночасно отримують доступ до програми. Даний вид тестування продукту зазвичай вимірює швидкість та потужність програми.

Тестування навантаження проводиться для визначення поведінки системи як в нормальних, так і в пікових умовах. В основному він використовується для того, щоб додаток працював задовільно, коли багато користувачів намагаються отримати доступ до нього або використовувати його одночасно.

Таким чином, кожного разу, коли ми змінюємо навантаження, ми контролюємо поведінку системи.

Навантажувальне тестування можна провести різними способами [7]:

- Ручне тестування: Це одна з стратегій виконання тестування навантаження, але вона не дає повторюваних результатів, не може забезпечити вимірюваного рівня напруги для програми та унеможливлює процес координації.
- Власно-розроблені інструменти: Організація, яка усвідомлює важливість тестування навантаження, може створити власні інструменти для виконання тестів.
- Інструменти з відкритим кодом. Є кілька інструментів для перевірки навантаження, які доступні у вигляді відкритого джерела, тобто безкоштовні. Вони можуть бути не такими витонченими, як їх платні колеги, але якщо бюджет обмежений, то вони є найкращим вибором.
- Корпоративні інструменти: Вони підтримують велику кількість протоколів та можуть імітувати надзвичайно велику кількість користувачів.

Як вище зазначалося, дане тестування можна проводити як вручну, так і за допомогою утиліт. Але рекомендується проводити його за допомогою допоміжних інструментів, оскільки в іншому випадку програма буде перевірена лише на найменше навантаження.

Наприклад, коли нам потрібно протестувати завантаження системи для десяти користувачів, ми можемо досягти цього і вручну[5]. Це можна зробити, завантаживши необхідну кількість фізичних осіб з різних машин. У цьому сценарії доцільно перейти на перевірку навантаження вручну, а не інвестувати в інструмент та створювати середовище для нього.

Тоді як уявіть, що нам потрібно створити тест для 1500 користувачів. В цьому випадку нам потрібно автоматизувати навантаження за допомогою будь-якого з доступних інструментів, заснованих на технологіях, на яких створено додаток, а також на основі грошей, які ми маємо на проект.

Якщо у нас є достатньо коштів, ми можемо скористатись комерційними інструментами, такими як Load runner, але якщо у нас бюджет обмежений, ми можемо використовувати інструменти з відкритим кодом, наприклад, JMeter тощо.

При автоматизованому навантажувальному тестуванні ми замінюємо реальних користувачів на інструмент, який імітує їх дії. За допомогою цього ми можемо заощадити ресурси, а також час.

3.2 Мета навантажувального тестування

Тестування навантаження допомагає створити надійні та продуктивні системи. Якщо все виконано належним чином, ми можемо точно зрозуміти наступне [5]:

- Кількість користувачів, з якими система вміє обробляти або здатна масштабувати.
- Час відповіді кожної транзакції.
- Як поводиться кожен компонент всієї системи під навантаженням, тобто компоненти сервера додатків, компоненти веб-сервера, компоненти бази даних тощо.
- Яка конфігурація сервера найкраще обробити навантаження?
- Чи достатньо наявного обладнання, або є потреба в додатковому обладнанні.
- Ідентифікуються вузькі місця, такі як використання процесора, використання пам'яті, затримки мережі тощо.

Тестування продуктивності та тестування навантаження часто плутають між собою [8].

Тестування навантаження - це фактично підмножина тестування продуктивності, яка зосереджена на аналізі поведінки веб-додатків під певним навантаженням за попередньо визначений час.

З іншого боку, тестування продуктивності - це більш широкий термін, який включає перевірку різних аспектів системи, таких як стан програми при навантаженні вищому, ніж очікувалося (стрес-тестування); продуктивність програми з великим обсягом даних; здатність системи витримувати певне навантаження протягом тривалого часу (витривалість) тощо.

3.3 Переваги навантажувального тестування

До переваг можна віднести виявлення вузьких місць перед виробництвом, масштабованість, скорочення часу простою системи, покращення задоволеності споживачів та зменшення витрат на відмову.

Тепер розглянемо кожний з пунктів більш конкретно [6]:

- Виявлення вузьких місць перед розгортанням надасть змогу правильно оцінити програмне забезпечення або веб-сайт перед розгортанням та допоможе виділити вузькі місця. Це дозволяє вирішити їх, перш ніж вони понесуть реальні витрати у світі.
- Підвищивши масштабованість системи ми зможемо визначити ліміт працездатності програми.
- Навантажувальне тестування можна використовувати для створення сценаріїв, які можуть спричинити збій системи. Це робить його чудовим інструментом пошуку шляхів вирішення проблем з високим трафіком до їх виникнення в реальному світі.
- Якщо час відгуку веб-сайту короткий, навіть якщо він збільшується від більшої аудиторії, клієнти будуть більш схильні повернутися ще раз.
- Виявлення проблем на найбільш ранній стадії, особливо перед запуском, зменшує вартість помилок.

3.4 Недоліки навантажувального тестування

Недоліків навантажувального тестування значно менше ніж переваг. Але все ж вони наявні.

Здебільшого вони стосуються цін на інструменти автоматизації, які є достатньо великими. Також є необхідність в дуже гарній підготовці до початку тестування.

Більш детально [8]:

1) Багато інструментів для перевірки навантаження є ліцензованими та стягують велику суму грошей за ліцензію.
2) Навіть у випадку вільних та відкритих інструментів, таких як JMeter, потрібне середовище для тестування навантаження, яке повинно бути максимально наближеним до налаштування виробничого середовища. Це знову ж призводить до додаткових ресурсів і витрат.
3) Для створення тестового сценарію необхідні знання сценарію мови, що підтримується інструментом.
4) Неправильно налаштований сценарій плану тестування навантаження може призвести до помилкових проблем з продуктивністю, що знову ж може потребувати значну кількость часу та ресурсів на виправлення.

РОЗДІЛ 4. Опис практичної частини роботи

4.1 Встановлення та налаштування використаних технологій

Для того щоб провести інтеграційне тестування спочатку потрібно в pom.xml файл вставити відповідні залежності(dependencies), зображені на рисунку 1.

```
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
</dependency>
<dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
    <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
        </dependency>
</dependencies>
```

Рисунок 1 Необхідні залежності

Також для того щоб інтеграційно протестувати spring-boot застосунок необхідно використати @SpringBootTest. Spring-boot також надають інші класи такі як TestRestTemplate для тестування REST API's. RestTemplate також має методи getForObject(), postForObject(), exchange() і так далі(рисунок 2).

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class StudentControllerTests {

    @LocalServerPort
    private int port;

    TestRestTemplate restTemplate = new TestRestTemplate();
```

Рисунок 2 @SpringBootTest та TestRestTemplate

Тепер перейдемо до налаштування середовища тестування. Першим кроком буде налаштування та конфігурація бази даних для тестування. Для цього спочатку необхідно імпортувати наступні залежності(рисунок 3):

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>mysql</artifactId>
    <version>1.13.0</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.19</version>
</dependency>
```

Рисунок 3 MySql dependencies

Також у файл application.properties потрібно вставити наступні рядки, для того щоб Spring використовував правильний драйвер-клас, і щоб таблиці заповнювались і видалялись при кожному запуску тестів(рисунок 4):

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.jpa.hibernate.ddl-auto=create-drop
```

Рисунок 4 Driver configuration

Після цього необхідно створити тест-контейнер обраної бази даних(в даному випадку MySQl). Його необхідно заповнити: назва бд, ім’я користувача та пароль(рисунок 5).

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = KmauditApplication.class)
@ActiveProfiles("tc")
@ContextConfiguration(initializers = {CourseRepositoryTCLiveTest.Initializer.class})
public class CourseRepositoryTCLiveTest extends CourseRepositoryTest {
    @ClassRule
    public static MySQLContainer MySQLContainer = new MySQLContainer()
            .withDatabaseName("kmaudit")
            .withUsername("root")
            .withPassword("yarik1999n");
```

Рисунок 5 Test-container

В попередньому прикладі було використано анотацію @ClassRule. Це було зроблено для того, щоб налаштовувати контейнер бази даних перед запуском тест-методів. Також для того щоб налаштувати підключення було створено статичний внутрішній клас, який реалізує ApplicationContextInitializer. Крім

цього в анотацію @ContextConfiguration потрібно вставити Initializer.class як параметр(рисунок 6).

```
static class Initializer
        implements ApplicationContextInitializer<ConfigurableApplicationContext> {
    public void initialize(ConfigurableApplicationContext configurableApplicationContext) {
        TestPropertyValues.of(
                "spring.datasource.url=" + MySQLContainer.getJdbcUrl(),
                "spring.datasource.username=" + MySQLContainer.getUsername(),
                "spring.datasource.password=" + MySQLContainer.getPassword()
        ).applyTo(configurableApplicationContext.getEnvironment());
    }
}
```

Рисунок 6 Initializer.class

У попередньому пункті було описано, як використовувати тестові контейнери у випадку одного тесту. У реальному випадку, хотілося б повторно використовувати один і той же контейнер бази даних в декількох тестах через відносно тривалий час запуску. Тепер перейдемо до створення загального класу створення контейнерів баз даних, розширивши MySQLContainer(рисунок 7):

```
public class MySQlContainer extends MySQLContainer<MySQlContainer> {
    private static final String IMAGE_VERSION = "mysql:11.1";
    private static MySQlContainer container;

    private MySQlContainer() {
        super(IMAGE_VERSION);
    }

    public static MySQlContainer getInstance() {
        if (container == null) {
            container = new MySQlContainer();
        }
        return container;
    }

    @Override
    public void start() {
        super.start();
        System.setProperty("DB_URL", container.getJdbcUrl());
        System.setProperty("DB_USERNAME", container.getUsername());
        System.setProperty("DB_PASSWORD", container.getPassword());
    }

    @Override
    public void stop() {
    }
}
```

Рисунок 7 MySQlContainer.class

В цьому варіанті реалізовано простий паттерн singleton, в якому лише перший тест виконує запуск контейнера, а кожен наступний використовує існуючий екземпляр. У методі start () ми використовуємо System.setProperty для встановлення параметрів з'єднання як змінних середовища.

Тепер їх можна розмістити в файлі application.properties(рисунок 8):

```
spring.datasource.url=${DB_URL}
spring.datasource.username=${DB_USERNAME}
spring.datasource.password=${DB_PASSWORD}
```

Рисунок 8 Environment variables

Після цього з’являється можливість використовувати новостворений допоміжний клас при налаштуванні тесту(рисунок 9).

```
public class CourseRepositoryTCLiveTest extends CourseRepositoryTest {
    @ClassRule
    public static MySQlContainer MySQLContainer = MySQlContainer.getInstance();
```

Рисунок 9 MySQLContainer.class usage

Код конфігурації бази даних знаходиться в додатку п.4.

Наступним кроком після налаштування бази даних буде створення запитів до неї. Для цього в репозиторії застосунку(в даному випадку CourseRepository) створюємо запити, якими будемо перевіряти взаємодію з базою даних. А саме: запит пошуку курсу по id, по знаходження всіх курсів для спеціальності прикладна математика, по пошуку курсів що мають часове навантаження менше за вказане користувачем та створення нового курсу(рисунок 10).

```
@Query("SELECT u FROM Course u WHERE u.speciality = APPLIED_MATHEMATICS")
List<Course> findAllMathCourses();

@Query("SELECT u FROM Course u WHERE u.id = ?1")
Course findCourseByID(int id);

@Query("SELECT u FROM Course u WHERE u.hours_amount < ?1")
List<Course> findCourseHoursLess(int hours);

@Query(value = "INSERT INTO Course (id, course_code, course_name, " +
        "credits_amount, education_type, faculty, hours_amount, speciality) " +
        "VALUES (:id, :course_code, :course_name, :credits_amount, " +
        ":education_type, :faculty, :hours_amount, :speciality)", nativeQuery = true)
@Modifying
void insertCourse(@Param("id") Integer id, @Param("course_code") long course_code,
                  @Param("course_name") String course_name, @Param("credits_amount") BigDecimal credits_amount,
                  @Param("education_type") EducationType education_type, @Param("faculty") Faculty faculty,
                  @Param("hours_amount") BigDecimal hours_amount, @Param("speciality") Speciality speciality);
```

Рисунок 10 Запити

Для того щоб провести навантажувальне тестування було використано інструмент JMeter(рисунок 11). Він безкоштовний та є у вільному доступі. Щоб встановити JMeter спочатку потрібно встановити Java, тому надалі будемо вважати, що вона вже наявна на вашому комп'ютері.

Інструкція з встановлення на MacOS/Linux [9]:

- Завантажити файл з розширенням .tgz у списку «Binaries» з назвою «apache-jmeter-5.0.tgz» у даному випадку.
- Перейти до каталогу, де завантажено apache-jmeter-5.0.tgz та вилучити вміст файлу наступною командою: **tar -xf apache-jmeter-5.0.tgz.**
- Далі потрібно перейти до каталогу JMeter: **cd apache-jmeter-5.0**.
- Запустити JMeter наступною командою: **./bin/jmeter**.
- Також якщо у вас встановлений homebrew, то достатньо виконати команду **brew install jmeter** і надалі через консоль запускати інструмент командою **jmeter**.

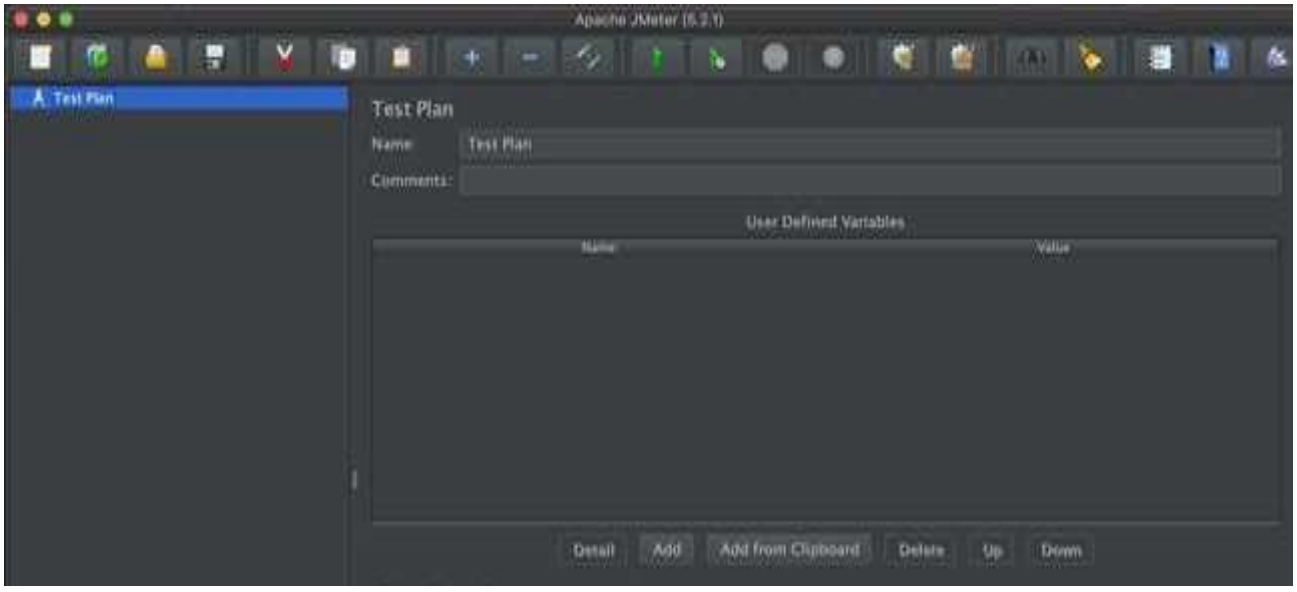

Рисунок 11 Запущений JMeter

4.2 Приклад реалізації

Для того щоб продемонструвати вище зазначені види перевірки якості у дії, був використаний проект системи збору аналітики про студентів університету. Були застосовані сучасні техніки та підходи інтеграційного та навантажувального тестування.

Спочатку розглянемо тестування вищенаведених запитів, які ми навели при конфігурації бази даних(рисунок 12).

```
@Repository
public interface CourseRepository extends IBaseRepository<Course, Integer>  {

    @Query("SELECT u FROM Course u WHERE u.speciality = APPLIED_MATHEMATICS")
    List<Course> findAllMathCourses();

    @Query("SELECT u FROM Course u WHERE u.id = ?1")
    Course findCourseByID(int id);

    @Query("SELECT u FROM Course u WHERE u.hours_amount < ?1")
    List<Course> findCourseHoursLess(int hours);

    @Query(value = "INSERT INTO Course (id, course_code, course_name, " +
            "credits_amount, education_type, faculty, hours_amount, speciality) " +
            "VALUES (:id, :course_code, :course_name, :credits_amount, " +
            ":education_type, :faculty, :hours_amount, :speciality)", nativeQuery = true)
    @Modifying
    void insertCourse(@Param("id") Integer id, @Param("course_code") long course_code,
                      @Param("course_name") String course_name, @Param("credits_amount") BigDecimal credits_amount,
                      @Param("education_type") EducationType education_type, @Param("faculty") Faculty faculty,
                      @Param("hours_amount") BigDecimal hours_amount, @Param("speciality") Speciality speciality);
}
```

Рисунок 12 CourseRepository

Першим чином, перевіримо здатність бази даних обробляти та видавати правильний результат на запит, який має повертати усі курси, що стосується спеціальності прикладна математика. Для цього було створено відповідний тест.

Він починається з створення чотирьох нових курсів. В даному випадку це МПА, ОКА, JAVA та SQL. Після створення зберігаємо їх та повертаємо необхідні нам курси за допомогою запиту,.

Оскільки серед курсів, які ми створили, лише три є для спеціальності прикладна математика, то за допомогою assertThat порівнюємо очікувану кількість з отриманою(рисунок 13).

Рисунок 13 allMathCoursesTest

Наступним ми перевіримо взаємодію бази даних з запитом, який відповідає за пошук предмета по ID. Як і в минулому випадку для цього створюємо відповідний тест.

Даний тест складається зі створення нового предмету(в даному випадку “МПА”). Зберігаємо його та за допомогою запиту, в якому ми вказуємо ID тільки що створеного предмету, видобуваємо курс.

Наприкінці, за допомогою assertThat, перевіряємо чи отриманий курс має такий же ID, як і у предмета, який ми створили на початку тесту(рисунок 14).

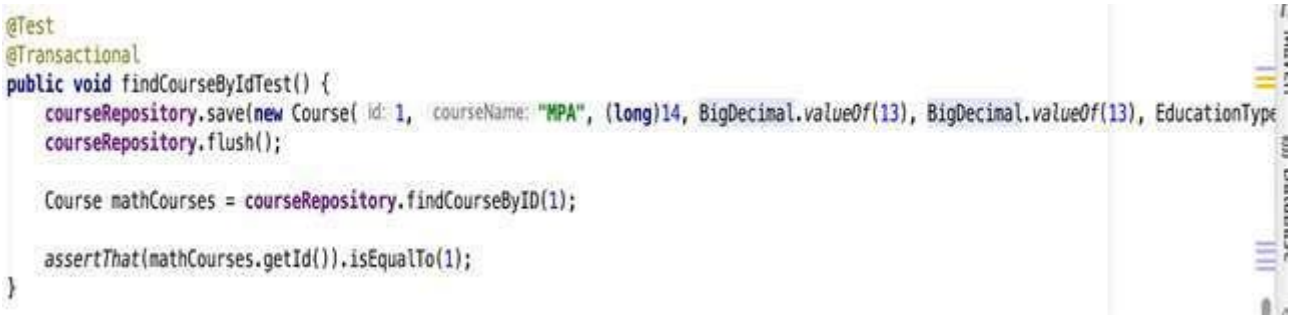

Рисунок 14 findCourseByIdTest

Далі розглянемо запит, який відповідає за знаходження курсів, часове навантаження яких менше за вказане користувачем. Розглянемо тест більш детально.

Він починається із створення чотирьох курсів з різним часовим навантаженням(22, 24, 31, 33 годин відповідно). Далі необхідно зберегти їх та виокремити лише необхідні предмети за допомогою запиту(як параметр вказати 30).

Оскільки серед створених предметів лише два із чотирьох мають часове навантаження менше 30, то за допомогою assertThat порівнюємо очікувану кількість з отриманою(рисунок 15).

Рисунок 15 findCourseHoursLessTest

Останній запит відповідає за створення нового курсу. Розглянемо його перевірку більш детально.

Спочатку виконуємо тестований запит. Як параметри передаємо id, назву, код, кількість кредитів, тип навчання, факультет, годинне навантаження та спеціальність новоствореного курсу. Наступним кроком буде створення об’єкту класу курс та заповнення його за допомогою findCoursebyId(), як аргумент вказуємо id, який ми вказали при виконанні тестованого запиту.

Наостанок, за допомогою assertThat(), перевіряємо чи не пустий новостворений об’єкт, та чи ім’я курсу рівне “SQL”(рисунок 16).

```
@Test
@Transactional
public void insertCourseTest() {
    courseRepository.insertCourse( id: 1,(long)11, course_name: "SQL",
            BigDecimal.valueOf(25), EducationType.CONTRACT,
            Faculty.FACULTY_OF_COMPUTER_SCIENCES, BigDecimal.valueOf(35),
            Speciality.COMPUTER_SCIENCE);
    Course sql = courseRepository.findCourseByID(1);

    assertThat(sql).isNotNull();
    assertThat(sql.getCourseName()).isEqualTo("SQL");
}
```

Рисунок 16 insertCourseTest

Код реалізації перевірки працездатності запитів до бази даних знаходиться в додатку п.4.

Крім цього, були протестовані Controller та Service layer даного проекту. В наступних прикладах буде розібране інтеграційне тестування CourseService та CourseController.

Дані класи мають наступні методи, які необхідно протестувати. Вони зображені на рисунках 17,18.

```
public abstract class CourseService extends BaseService<Course, Integer> {

    public CourseService() { super(Course.class); }
}
```

```
public abstract class BaseService<E extends GettableById<I>, I extends Serializable> implements IBaseService<E, I> {

    @Autowired
    private IBaseRepository<E, I> repository;

    @Autowired
    protected MessageBundleService messageBundleService;

    protected Class<E> persistentClass;

    public BaseService(final Class<E> persistentClass) { this.persistentClass = persistentClass; }

    @Override
    public I save(final E entity) { return repository.saveAndFlush(entity).getId(); }

    public I update(final E entity, final I id) {
        E entityToUpdate = getById(id);
        BeanUtils.copyProperties(entity, entityToUpdate, ObjectUtils.getNullPropertyNames(entity));
        return repository.saveAndFlush(entityToUpdate).getId();
    }

    public E getById(final I id) {
        return repository.findById(id).orElseThrow(() -> new NoSuchEntityException(
                messageBundleService.getMessage( value: "entity.not.exists", persistentClass.getSimpleName(),
                                                  String.valueOf(id))));
    }

    public void delete(final I id) {
        throw new BusinessLogicException(
                messageBundleService.getMessage( value: "entity.could.not.be.deleted", persistentClass.getSimpleName()));
    }
}
```

Рисунок 17 CourseService

```
@RestController
@RequestMapping("/courses")
public class CourseController extends BasicController<CreateUpdateCourseDto, CourseDto, Integer>  {
}
```

```
public abstract class BasicController<C, V, I extends Serializable> {

    @Autowired
    private IBaseFacade<C, V, I> basicFacade;

    @PutMapping
    @ResponseStatus(HttpStatus.CREATED)
    public IdDto<I> saveCompetitionInfo(
            @Valid
            @RequestBody
            final C requestDto) {
        return basicFacade.create(requestDto);
    }

    @PostMapping("/{id}")
    public IdDto<I> updateCompetitionInfo(
            @Valid
            @RequestBody
            final C requestDto,
            @PathVariable
            final I id) {
        return basicFacade.edit(requestDto, id);
    }

    @GetMapping("/{id}")
    public V getCompetitionById(
            @PathVariable
            final I id) {
        return basicFacade.getById(id);
    }

}
```

Рисунок 18 CourseController

Тепер перейдемо безпосередньо до інтеграційного тестування. Першим чином, потрібно імпортувати необхідні бібліотеки(рисунок 19).

```
package ukma.edu.ua.kmaudit.services;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import ukma.edu.ua.kmaudit.domains.entities.Course;
import ukma.edu.ua.kmaudit.domains.enums.Faculty;
import ukma.edu.ua.kmaudit.domains.enums.Speciality;

import java.math.BigDecimal;

import static org.assertj.core.api.Assertions.assertThat;
import static ukma.edu.ua.kmaudit.domains.enums.EducationType.CONTRACT;
```

Рисунок 19 Перелік бібліотек

Наступним кроком буде вказати, що буде протестований саме Spring-застосунок.

Це можна зробити строками коду, зображеними на рисунку 20.

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class CourseServiceTest {
```

Рисунок 20 SpringBootTest

Розглянемо інтеграційне тестування на прикладі методів save() та getById класу CourseService.

Спочатку створюємо об’єкт класу Course. Заповнюємо його необхідною інформацією та використовуємо тестований метод. За допомогою методу getById добуваємо новостворений курс.

Вкінці, за допомогою assertThat, перевіряємо чи все виконалося вірно(рисунок 21).

```
@Autowired
private CourseService courseService;

@Test
public void saveCourse() {
    Course course = new Course();
    course.setId(2);
    course.setCourseCode((long)11);
    course.setCourseName("MPA");
    BigDecimal decimal = new BigDecimal( val: 120.4);
    course.setCreditsAmount(decimal);
    course.setFaculty(Faculty.FACULTY_OF_COMPUTER_SCIENCES);
    course.setSpeciality(Speciality.APPLIED_MATHEMATICS);
    BigDecimal hours = new BigDecimal( val: 48.3);
    course.setHoursAmount(hours);
    course.setEducationType(CONTRACT);
    courseService.save(course);
    Course coursebyId = courseService.getById(2);
    assertThat(coursebyId).isNotNull();
}
```

Рисунок 21 saveCourse test

Тепер розглянемо тестування методу getById(). Для цього створюємо обєкт класу Course. Далі заповнюємо його, використовуючи метод, який тестується.

Наприкінці, за допомогою assertThat, перевіряємо чи Id отриманого предмета співпадає з очікуваним(рисунок 22).

```
@Test
public void testGetByID() {
    Course courseById = courseService.getById(2);
    assertThat(courseById.getId()).isEqualTo(2);
}
```

Рисунок 22 getById test

Реалізація тестування усіх методів даного класу знаходиться в додатку п.1.

Тепер перейдемо до CourseController класу. Для демонстрації інтеграційного тестування перевіремо працездатність методів saveCourse() та getCourseById.

Аналогічно до минулого тесту все починається з імпортування необхідних бібліотек та вказування, що буде протестований Spring-boot застосунок(рисунок 23).

```
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.boot.web.server.LocalServerPort;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import ukma.edu.ua.kmaudit.KmauditApplication;
import ukma.edu.ua.kmaudit.domains.entities.Course;
import ukma.edu.ua.kmaudit.domains.enums.*;
import java.math.BigDecimal;
import java.net.URI;
import java.net.URISyntaxException;
import static org.junit.jupiter.api.Assertions.*;
import static ukma.edu.ua.kmaudit.domains.enums.EducationType.CONTRACT;


@SpringBootTest(classes = KmauditApplication.class,
        webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
```

Рисунок 23 Імпортовані бібліотеки та SpringBootTest

Наступним кроком буде створення об’єктів класів TestRestTemplate, URI і Course(save), а також заповнення двох останніх(рисунок 24)

```
private TestRestTemplate restTemplate = new TestRestTemplate();

@Test
public void testCreateCourse() throws URISyntaxException {
    final String baseUrl = "http://localhost:" + port + "/courses/";
    URI uri = new URI(baseUrl);
    Course save = new Course();
    save.setCourseName("oki");
    save.setCourseCode((long)23344);
    save.setCreditsAmount(BigDecimal.valueOf(23));
    save.setEducationType(CONTRACT);
    save.setFaculty(Faculty.FACULTY_OF_COMPUTER_SCIENCES);
    save.setHoursAmount(BigDecimal.valueOf(22));
    save.setSpeciality(Speciality.APPLIED_MATHEMATICS);
    save.setId(3);
```

Рисунок 24 Створення об’єктів класів та їх заповнення

Далі створюємо об’єкти класів HttpHeaders(headers) та HttpEntity(request) з параметром Course і аргументами save і headers.

Останнім кроком буде використання методу exchange(RestTemplate), і як аргументи вказати вище створенні об’єкти класів, а також тип запиту(HttpMethod.PUT). Щоб перевірити правильність проробленої роботи необхідно використати assertTrue(рисунок 25).

```
HttpHeaders headers = new HttpHeaders();
HttpEntity<Course> request = new HttpEntity<>(save, headers);
restTemplate.exchange(uri, HttpMethod.PUT, request, String.class);
assertTrue(
        this.restTemplate
                .getForObject( url: "http://localhost:" + port + "/courses/3", Course.class)
                .getCourseName().equals("oki"));
```

Рисунок 25 Кінець тесту

Перейдемо до реалізації тестування методу getCourseById(). Переконатися у його працездатності можна за допомогою об’єкта класу TestRestTemplate (restTemplate). Далі необхідно зробити виклик getForObject(), вказавши як аргументи URI та клас шуканого об’єкту.

Наступним кроком буде видобування Id отриманого результату та порівняння його з очікуваним.

Наприкінці потрібно огорнути написаний код в assertTrue()(рисунок 26).

```
@Test
public void testGetById() {
    assertTrue(
            this.restTemplate
                    .getForObject( url: "http://localhost:" + port + "/courses/1", Course.class)
                    .getId().equals(1));
}
```

Рисунок 26 testGetById

Реалізація перевірки якості усіх методів даного класу знаходиться в додатку п.2.

Бувають ситуації, коли в нас немає доступу до бази даних застосунку. В такому випадку є два варіанти вирішення даної проблеми.

Перший варіант це використати Mock() на репозиторій проекту. Розглянемо його більш детально.

Для перевірки класу CourseService, екземпляр цього класу має бути створений і доступний як @Bean, щоб ми могли використати @Autowired в

нашому випадку. Дана конфігурація досягається за допомогою анотації @TestConfiguration.

Ще одна цікава річ - використання @MockBean. Дана анотація створює макет для CourseRepository, який використаний для обходу виклику фактичного репозиторію(рисунок 27):

```
public class MockServiceTest {
    @TestConfiguration
    static class CourseServiceImplTestContextConfiguration {

        @Bean
        public CourseService courseService() {
            return new CourseServiceImpl();
        }
    }
    @Autowired
    private CourseService courseService;
    @MockBean
    private CourseRepository courseRepository;
```

Рисунок 27 TestConfiguration

Далі за допомогою анотації @Before ми прописуємо дії, які будуть здійснюватись перед запуском кожного тесту. В даному випадку, за допомогою Mockito, ми прописуємо, що має виконуватись при використанні методу findById() класу CourseRepository(рисунок 28).

```
@Before
public void setUp() {
    Course alex = new Course( id: 1,  courseName: "MPA", (long)14, BigDecimal.valueOf(13),
            BigDecimal.valueOf(22), EducationType.CONTRACT, Faculty.FACULTY_OF_COMPUTER_SCIENCES,
            Speciality.APPLIED_MATHEMATICS,  courses: null);

    Mockito.when(courseRepository.findById(alex.getId()))
            .thenReturn(java.util.Optional.of(alex));
}
```

Рисунок 28 setUp()

Після виконаних дій, створюємо тест. За допомогою методу getById() заповнюємо новостворений об’єкт класу Course. А потім за допомого assertThat() порівнюємо чи очікуваний id співпадає з отриманим(рисунок 29).

```
@Test
public void testFindById() {
    int id = 1;
    Course found = courseService.getById(id);

    assertThat(found.getId())
            .isEqualTo(id);
}
```

Рисунок 29 MockTest()

Другим варіантом є використання вбудованої бази даних. Для цього спочатку потрібно додати в файл pom.xml наступні залежності(рисунок 30):

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.4.187</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>5.2.9.Final</version>
</dependency>
```

Рисунок 30 Використані залежності

Створимо файл persistent-student.properties у папці src/test/resources, який містить H2-database значення та необхідні дані користувача(рисунок 31):

```
jdbc.driverClassName=org.h2.Driver
jdbc.url=jdbc:h2:mem:myDb;DB_CLOSE_DELAY=-1
hibernate.dialect=org.hibernate.dialect.H2Dialect
hibernate.hbm2ddl.auto=create
jdbc.user=user
jdbc.pass=pass
```

Рисунок 31 persistent-student.properties

Наступним кроком буде створення класу @Configuration, який шукає попередньо створений файл, і створює місце для зберігання інформації, використовуючи визначені в ньому властивості бази даних(рисунок 32):

```
@Configuration
@EnableJpaRepositories(basePackages = "ukma.edu.ua.kmaudit.repositories")
@PropertySource("persistence-course.properties")
@EnableTransactionManagement
public class CourseJPAConfig {

    @Autowired
    private Environment env;

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(env.getProperty("jdbc.driverClassName"));
        dataSource.setUrl(env.getProperty("jdbc.url"));
        dataSource.setUsername(env.getProperty("jdbc.user"));
        dataSource.setPassword(env.getProperty("jdbc.pass"));

        return dataSource;
    }
}
```

Рисунок 32 CourseJPAConfig

Після налаштування вбудованої бази даних, перейдемо до створення тесту методу save(), класу CourseRepository. За допомогою анотації @ContextConfiguration підключаємо налаштування, створенні на попередньому кроці.

Далі створюємо новий екземпляр класу Course та заповнюємо його. Після цього використовуємо тест-метод.

Наприкінці, за допомогою findById() (аргументом виступає id курсу, що ми зберігали), заповнюємо ще один об’єкт класу та, використавши assertEquals, впевнюємося що очікувана назва така ж як і отримана(рисунок 33)

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(
        classes = { CourseJPAConfig.class },
        loader = AnnotationConfigContextLoader.class)
@Transactional
public class InMemoryTest {

    @Resource
    private CourseRepository courseRepository;

    @Test
    public void saveCourseTest() {
        Course test = new Course( id: 1,  courseName: "MPA", (long)14, BigDecimal.valueOf(13),
                BigDecimal.valueOf(22), EducationType.CONTRACT, Faculty.FACULTY_OF_COMPUTER_SCIENCES,
                Speciality.APPLIED_MATHEMATICS,  courses: null);
        courseRepository.save(test);

        Course test2 = courseRepository.findById(1).orElse( other: null);
        assert test2 != null;
        assertEquals( expected: "MPA", test2.getCourseName());
    }
}
```

Рисунок 33 InMemoryTest

Наш тест запуститься повністю автономним способом - він створить базу даних H2 в пам'яті, виконає необхідні операції, а потім закриє з’єднання та видалить базу даних.

Реалізація даних варіантів знаходяться в додатку п.5 та п.6.

Щоб повністю впевнитись у працездатності проекту було проведено навантажувальне тестування. Все починається з запуску JMeter та створення нової групи потоків(Thread Group). Тут ми вказуємо її назву, число користувачів, а також кількість повторів такої перевірки(рисунок 34).

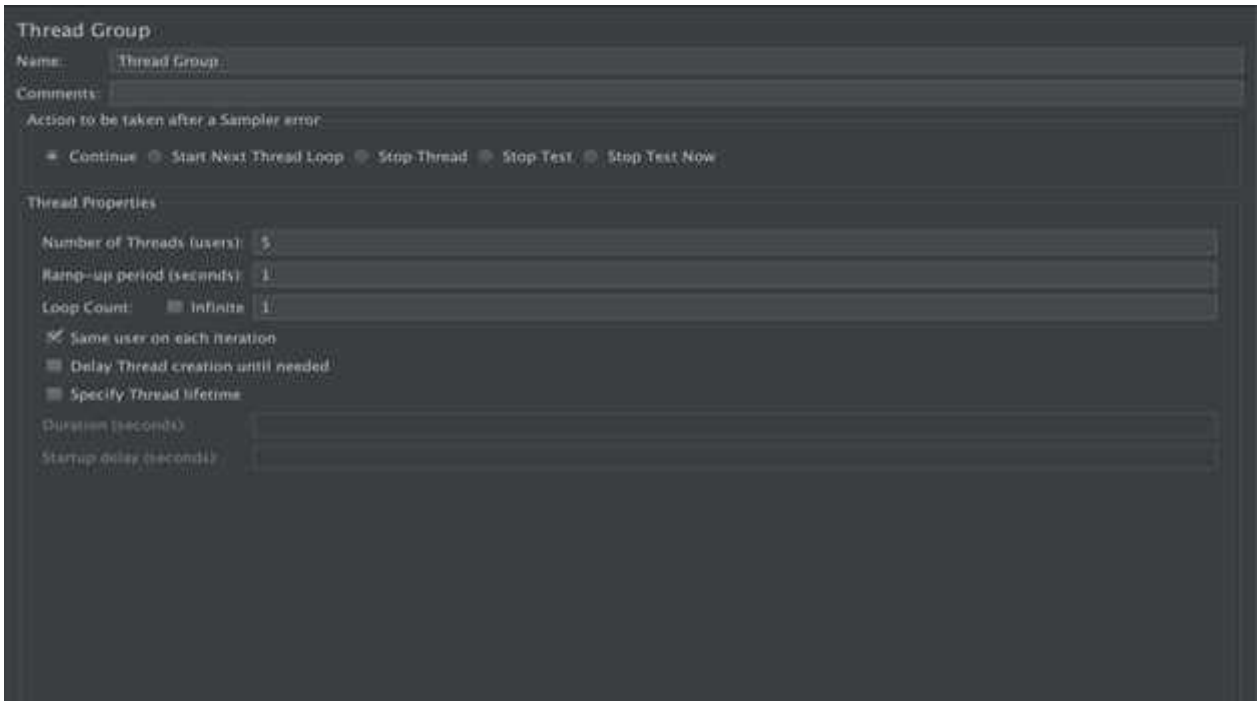

Рисунок 34 Thread Group

Наступним кроком буде додання HTTP Request. В ньому вказуємо назву серверу, номер порту. Також прописуємо шлях і вибираємо метод(GET, PUT, POST). Приклад такого HTTP Request зображено на рисунку 35.

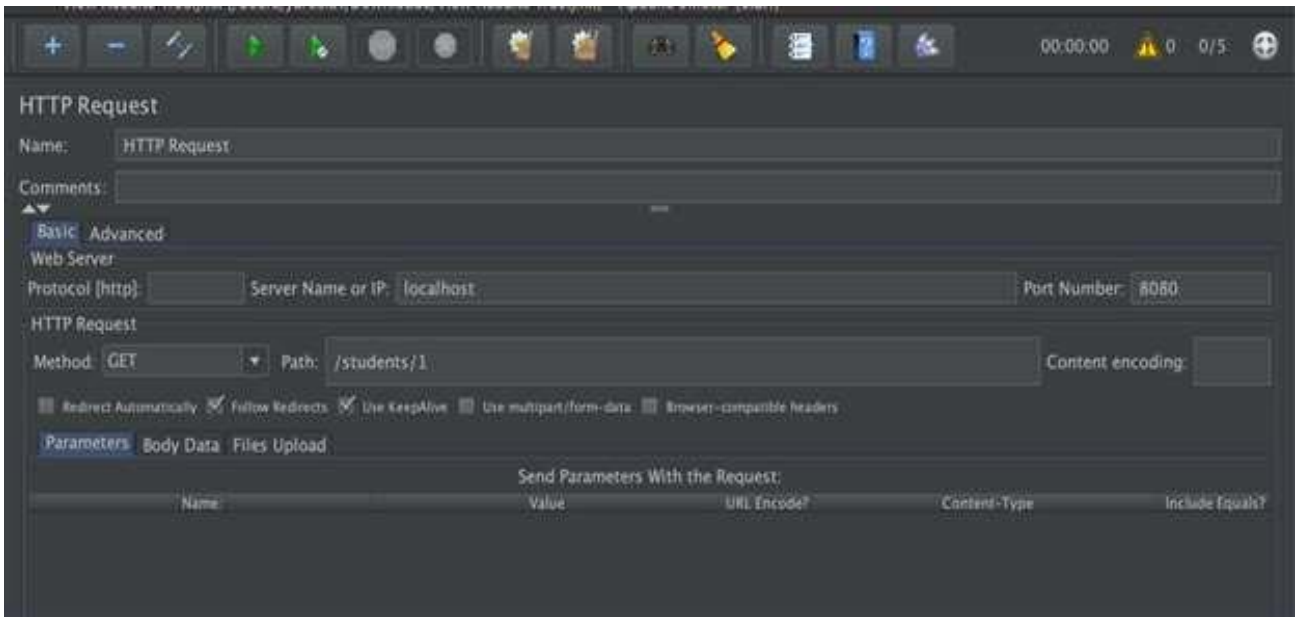

Рисунок 35 HTTP Request

У випадку якщо метод відмінний від GET, то треба заповнити Body Data необхідною інформацією у форматі JSON і вказати стандарт кодування(рисунок 36).

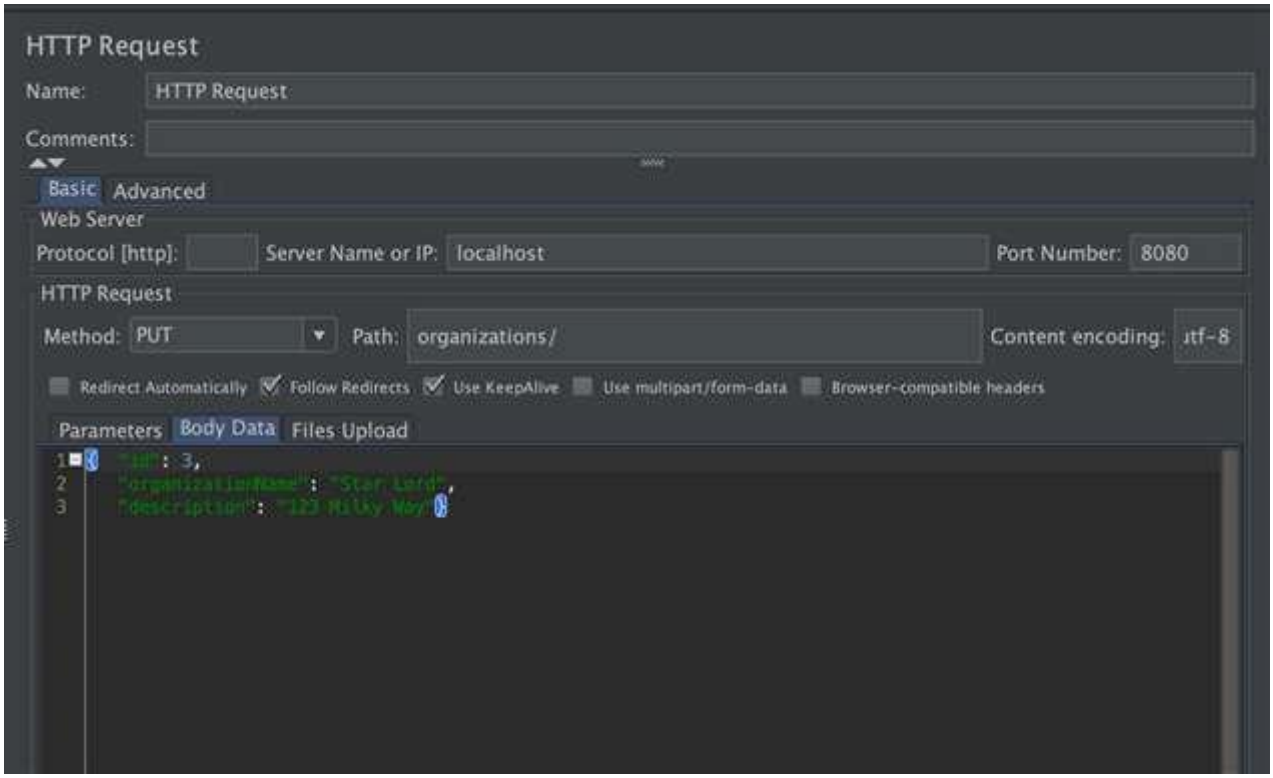

Рисунок 36 Body Data

Після цього в Http Header Manager потрібно створити новий header з ім’ям Content-Type і значенням application/json. Це зроблено для того, щоб застосунок зрозумів, в якому форматі ми подаємо дані для відправлення(рисунок 37).

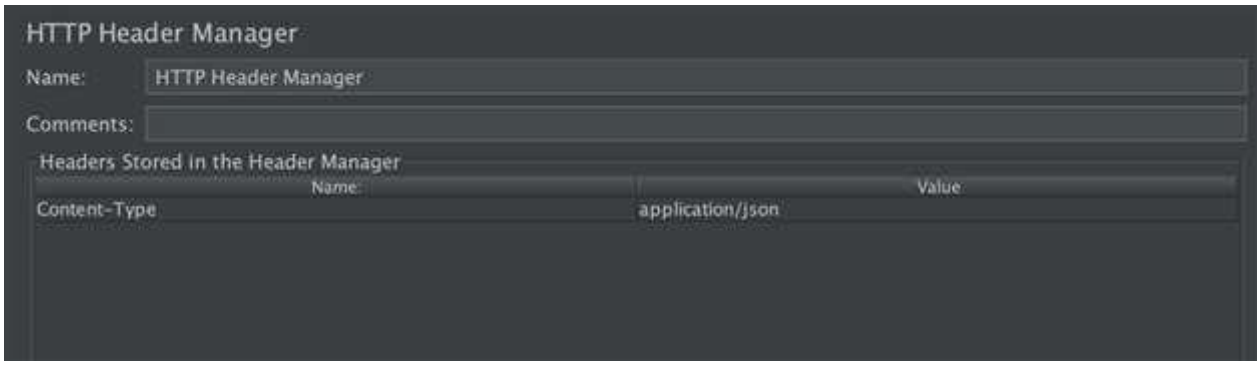

Рисунок 37 Http Header

Далі вказуємо обмеження по часу виконання тесту для кожного користувача(рисунок 38):

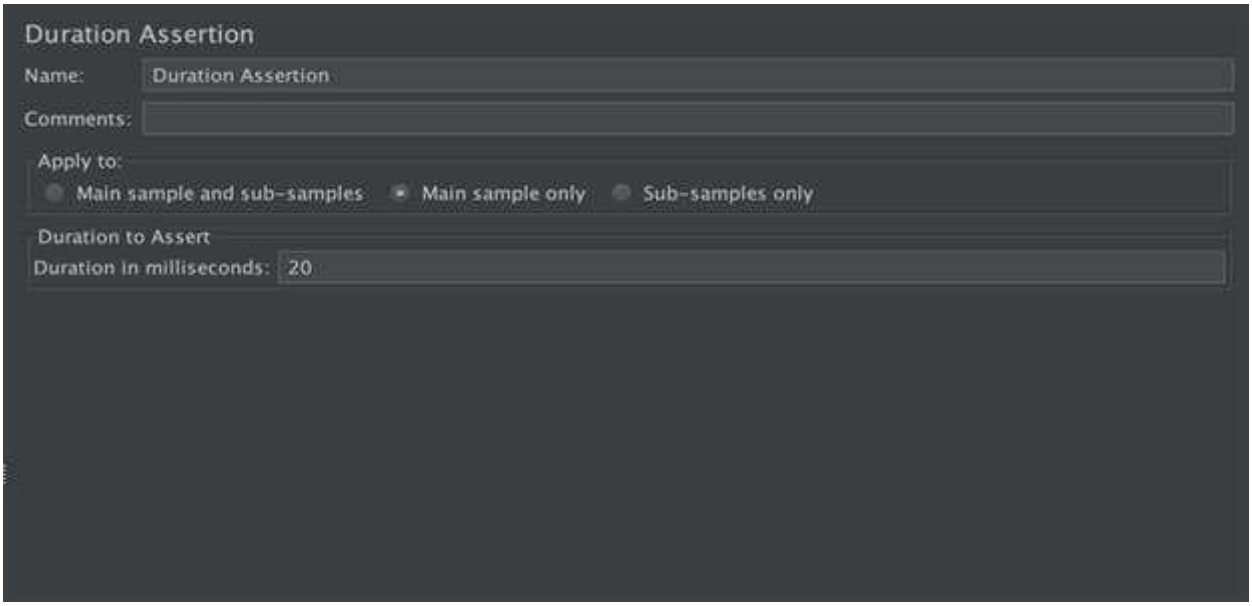

Рисунок 38 Duration Assertion

Крім обмеження по часу є можливість додати перевірку на результат виконання тесту. Тобто можна вказати рядок який буде перевірятись, а саме: тип даних, їх наповнення, URL, код відповіді тощо.

Також в цьому вікні можна вказати правило перевірки: містить, відповідає, рівне тощо. Наприкінці ми вводимо бажаний результат виконання тесту і повідомлення в разі не виконання(рисунок 39).

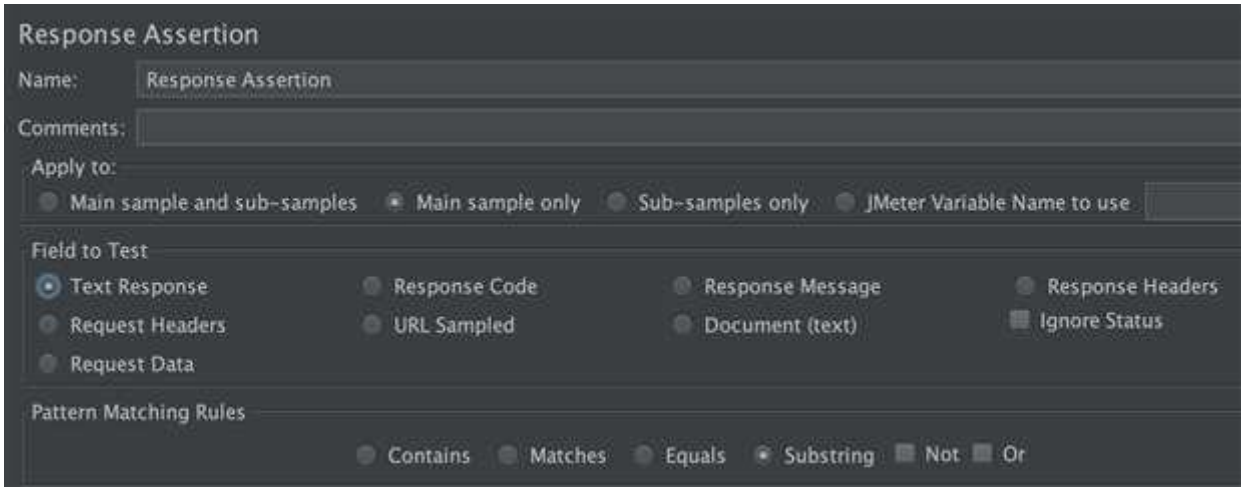

Рисунок 39 Response Assertion

Перед запуском відкриваємо View Results Tree для перегляду результату тесту. Вони можуть бути задовільні і навпаки, а також позначаються відповідними значками. Натиснувши на результат, можна отримати повну

інформацію про повний перебіг тесту для певного користувача або дізнатися про помилку, що сталася під час виконання.

Можливі варіанти результатів показано на рисунках 40, 41, 42.

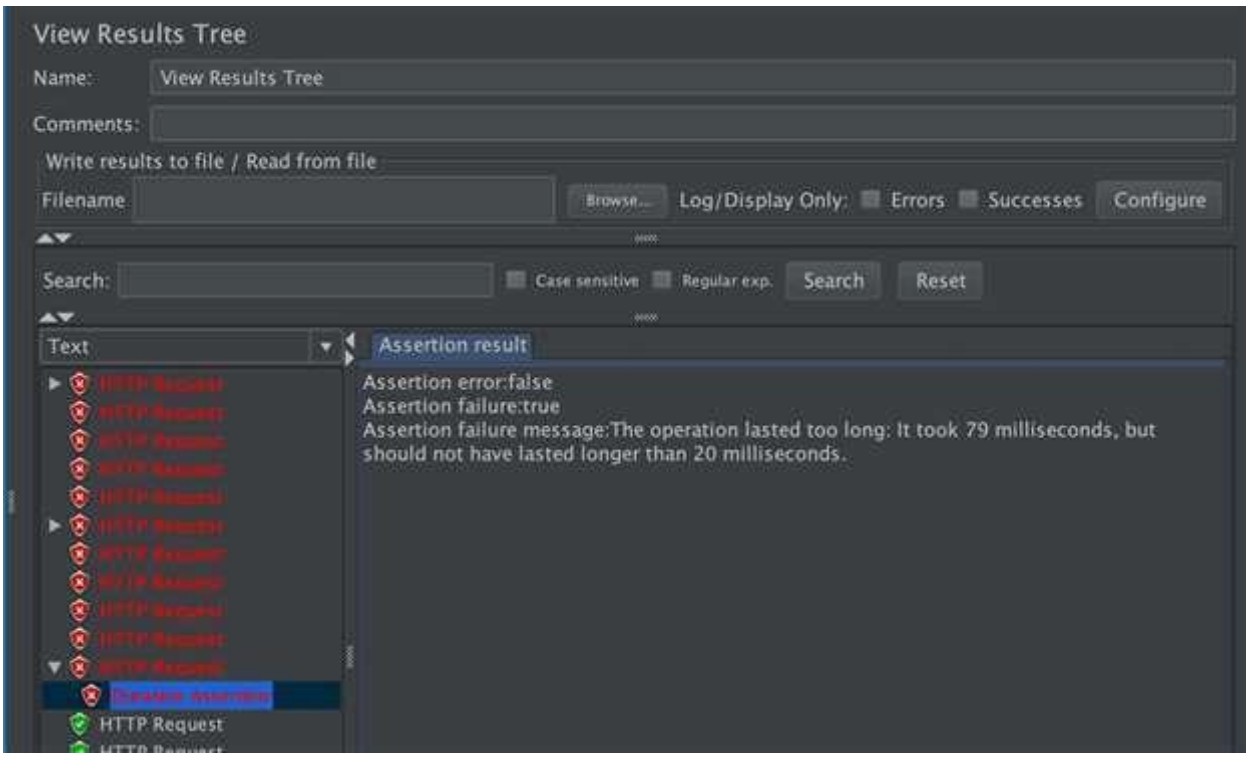

Рисунок 40 Помилка часу виконання

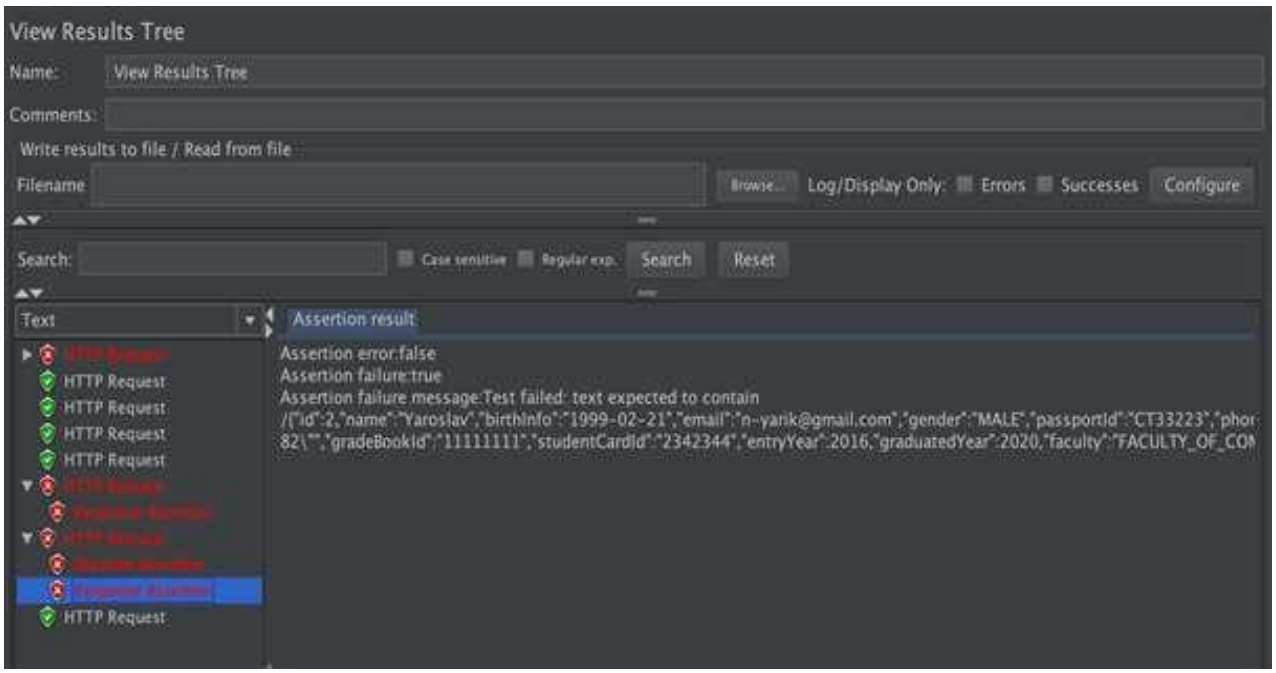

Рисунок 41 Невідповідність повернених даних

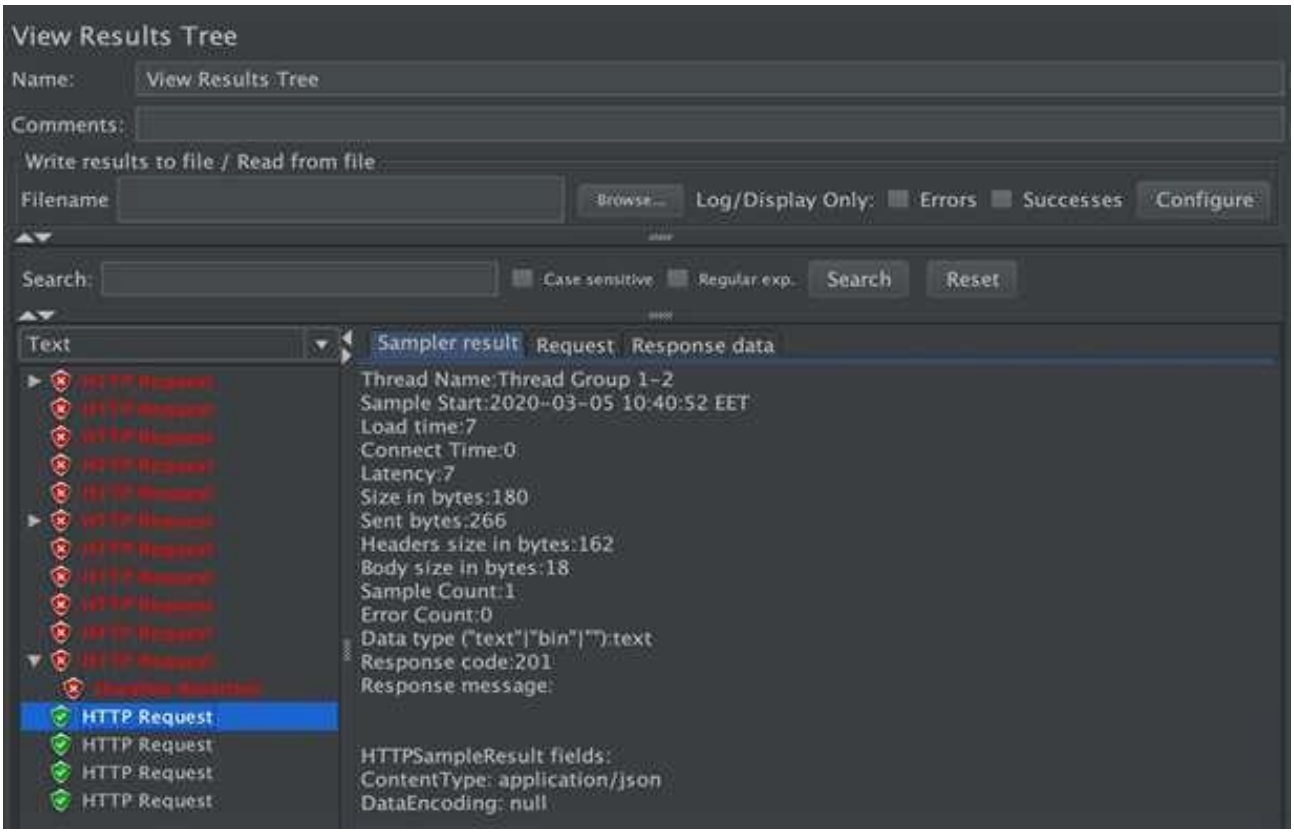

Рисунок 42 Успішний тест

Приклад скрипта, який був автоматично створений даним інструментом навантажувального тестування, знаходиться в додатку п.3.

Висновки

У даній роботі був проведений аналіз сучасних підходів до інтеграційного та навантажувального тестування; детально розписана інструкція, щодо встановлення та налаштування необхідних інструментів.

Під час дослідження була вивчена література та використані різноманітні інтернет-ресурси, проаналізовані плюси та мінуси інтеграційного та навантажувального тестування.

Крім цього, було проведено налаштування середовища для виконання роботи. А саме: продемонстровано варіант підключення бази даних для інтеграційного тестування та надана детальна інструкція щодо налаштування інструменту для навантажувального тестування.

Більш того, були наведені приклади тестування у різних умовах. Наприклад, без доступу до бази даних застосунку. Для цього використані технології створення внутрішньої бази даних та Mock().

Також в даній роботі наводиться необхідність перевірки якості при розробці програмного забезпечення. Результатом дослідження є практична робота, в якій продемонстровано тестування застосунку, з використанням інтеграційних та навантажувальних методів.

Таким чином, можна зробити висновок, що проведення перевірки якості продукту є невід’ємною частиною розробки програмного забезпечення. Тестування не тільки дозволяє перевірити чи всі вимоги до проекту виконані, а й допомагає у виявленні помилок та дефектів, що пом’якшує наслідки і зменшує ризики втрат.

Список джерел

1. [Електронний ресурс] стаття “Що таке тестування програмного забезпечення та яке його значення?” https://www.quality-assurance-group.com/shho-take-testuvannya-programnogo-zabezpechennya-ta-yake-jogo-znachennya/
2. [Електронний ресурс] лекція “Тестування програмного продукту” http://lib.mdpu.org.ua/e-book/vstup/L11.htm
3. [Електронний ресурс] стаття “Що таке інтеграційне тестування?” https://www.softwaretestinghelp.com/what-is-integration-testing/
4. [Електронний ресурс] стаття “Поняття інтеграційного тестування, його типи та приклади” https://www.guru99.com/integration-testing.html
5. [Електронний ресурс] стаття “Навантажувальне тестування для початківців” https://www.softwaretestinghelp.com/load-testing/
6. [Електронний ресурс] стаття “Що таке навантажувальне тестування?” https://stackify.com/what-is-load-testing/
7. [Електронний ресурс] стаття “Навантажувальне тестування і приклади” https://www.guru99.com/load-testing-tutorial.html#13
8. [Електронний ресурс] стаття “Навантажувальне тестування” https://artoftesting.com/load-testing
9. [Електронний ресурс] стаття “Інсталювання JMeter на Linux, Mac” https://www.quality-assurance-group.com/instalyuvannya-jmeter-na-windows-linux-mac/
10. [Електронний ресурс] стаття “Тестування білої скриньки” https://www.guru99.com/white-box-testing.html
11. [Електронний ресурс] стаття “Тестування чорної скриньки” https://www.guru99.com/black-box-testing.html

Додаток

1. Приклад інтеграційного тестування Course Service

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class CourseServiceTest {
    @Autowired
    private CourseService courseService;

    @Test
    public void saveCourse() {
        Course course = new Course();
        course.setId(2);
        course.setCourseCode((long)11);
        course.setCourseName("MPA");
        BigDecimal decimal = new BigDecimal(120.4);
        course.setCreditsAmount(decimal);
        course.setFaculty(Faculty.FACULTY_OF_COMPUTER_SCIENCES);
        course.setSpeciality(Speciality.APPLIED_MATHEMATICS);
        BigDecimal hours = new BigDecimal(48.3);
        course.setHoursAmount(hours);
        course.setEducationType(CONTRACT);
        courseService.save(course);
        Course coursebyId = courseService.getById(2);
        assertThat(coursebyId).isNotNull();
    }

    @Test
    public void testGetByID() {
        Course courseById = courseService.getById(2);
        assertThat(courseById).isNotNull();
    }
    @Test
    public void updateCourse(){
        Course ss = courseService.getById(1);
        ss.setCourseName("DB");
        courseService.update(ss, 1);
        Course refresh = courseService.getById(1);
        assertThat(refresh.getCourseName()).isEqualTo("DB");
    }

}
```

2. Приклад інтеграційного тестування Course Controller

```
@SpringBootTest(classes = KmauditApplication.class,
        webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class CourseControllerTest {

    @LocalServerPort
    private int port;

    private TestRestTemplate restTemplate = new TestRestTemplate();

    @Test
    public void testCreateCourse() throws URISyntaxException {
        final String baseUrl = "http://localhost:" + port + "/courses/";
        URI uri = new URI(baseUrl);
        Course save = new Course();
        save.setCourseName("oki");
        save.setCourseCode((long)23344);
        save.setCreditsAmount(BigDecimal.valueOf(23));
        save.setEducationType(CONTRACT);
        save.setFaculty(Faculty.FACULTY_OF_COMPUTER_SCIENCES);
        save.setHoursAmount(BigDecimal.valueOf(22));
        save.setSpeciality(Speciality.APPLIED_MATHEMATICS);
        save.setId(3);
        HttpHeaders headers = new HttpHeaders();
        HttpEntity<Course> request = new HttpEntity<>(save, headers);
        restTemplate.exchange(uri, HttpMethod.PUT, request, String.class);
        assertTrue(
                this.restTemplate
                        .getForObject("http://localhost:" + port + "/courses/3",
Course.class)
                        .getCourseName().equals("oki"));
    }
    @Test
    public void testGetById() {
        assertTrue(
                this.restTemplate
                        .getForObject("http://localhost:" + port + "/courses/1",
Course.class)
                        .getId().equals(1));
    }

    @Test
    public void testUpdateCourse() throws URISyntaxException {
```

```
        final String baseUrl = "http://localhost:" + port + "/courses/2";
        URI uri = new URI(baseUrl);
        Course save = new Course();
        save.setCourseName("oki");
        save.setCourseCode((long)23344);
        save.setCreditsAmount(BigDecimal.valueOf(23));
        save.setEducationType(CONTRACT);
        save.setFaculty(Faculty.FACULTY_OF_COMPUTER_SCIENCES);
        save.setHoursAmount(BigDecimal.valueOf(22));
        save.setSpeciality(Speciality.APPLIED_MATHEMATICS);
        save.setId(2);
        HttpHeaders headers = new HttpHeaders();
        HttpEntity<Course> request = new HttpEntity<>(save, headers);
        ResponseEntity<String> result = restTemplate.exchange(uri,
HttpMethod.POST, request, String.class);
        assertTrue(
                this.restTemplate
                        .getForObject("http://localhost:" + port + "/courses/2",
Course.class)
                        .getCourseName().equals("oki"));
    }

}
```

3. Приклад скрипта створеного JMeter (запит POST)

```
<?xml version="1.0" encoding="UTF-8"?>
<jmeterTestPlan version="1.2" properties="5.0" jmeter="5.2.1">
<hashTree>
  <TestPlan guiclass="TestPlanGui" testclass="TestPlan" testname="Test Plan" enabled="true">
<stringProp name="TestPlan.comments"></stringProp>
<boolProp name="TestPlan.functional_mode">false</boolProp>
<boolProp name="TestPlan.tearDown_on_shutdown">true</boolProp>
<boolProp name="TestPlan.serialize_threadgroups">false</boolProp>
    <elementProp name="TestPlan.user_defined_variables" elementType="Arguments" guiclass="ArgumentsPanel" testclass="Arguments" testname="User Defined Variables" enabled="true">
<collectionProp name="Arguments.arguments"/>
</elementProp>
<stringProp name="TestPlan.user_define_classpath"></stringProp>
</TestPlan>
<hashTree>
    <ThreadGroup guiclass="ThreadGroupGui" testclass="ThreadGroup" testname="Thread Group" enabled="true">
      <stringProp name="ThreadGroup.on_sample_error">continue</stringProp>
      <elementProp name="ThreadGroup.main_controller" elementType="LoopController" guiclass="LoopControlPanel" testclass="LoopController" testname="Loop Controller" enabled="true">
<boolProp name="LoopController.continue_forever">false</boolProp>
<stringProp name="LoopController.loops">1</stringProp>
</elementProp>
```

<stringProp name="ThreadGroup.num_threads">1</stringProp>
<stringProp name="ThreadGroup.ramp_time">1</stringProp>
<boolProp name="ThreadGroup.scheduler">false</boolProp>
<stringProp name="ThreadGroup.duration"></stringProp>
<stringProp name="ThreadGroup.delay"></stringProp>
<boolProp name="ThreadGroup.same_user_on_next_iteration">true</boolProp>
</ThreadGroup>
<hashTree>
<HTTPSamplerProxy guiclass="HttpTestSampleGui" testclass="HTTPSamplerProxy" testname="HTTP Request" enabled="true">
<boolProp name="HTTPSampler.postBodyRaw">true</boolProp>
<elementProp name="HTTPsampler.Arguments" elementType="Arguments">
<collectionProp name="Arguments.arguments">
<elementProp name="" elementType="HTTPArgument">
<boolProp name="HTTPArgument.always_encode">false</boolProp>
<stringProp name="Argument.value">{
"id": 2,
"organizationName": "update",
"description": "check"

}</stringProp>
<stringProp name="Argument.metadata">=</stringProp>
</elementProp>
</collectionProp>
</elementProp>
<stringProp name="HTTPSampler.domain">localhost</stringProp>
<stringProp name="HTTPSampler.port">8080</stringProp>
<stringProp name="HTTPSampler.protocol"></stringProp>

<stringProp name="HTTPSampler.contentEncoding">utf-8</stringProp>

<stringProp name="HTTPSampler.path">organizations/2</stringProp>

<stringProp name="HTTPSampler.method">POST</stringProp>

<boolProp name="HTTPSampler.follow_redirects">true</boolProp>

<boolProp name="HTTPSampler.auto_redirects">false</boolProp>

<boolProp name="HTTPSampler.use_keepalive">true</boolProp>

<boolProp name="HTTPSampler.DO_MULTIPART_POST">false</boolProp>

<stringProp name="HTTPSampler.embedded_url_re"></stringProp>

<stringProp name="HTTPSampler.connect_timeout"></stringProp>

<stringProp name="HTTPSampler.response_timeout"></stringProp>

</HTTPSamplerProxy>

<hashTree/>

<ResultCollector guiclass="ViewResultsFullVisualizer" testclass="ResultCollector" testname="View Results Tree" enabled="true">

<boolProp name="ResultCollector.error_logging">false</boolProp>

<objProp>

<name>saveConfig</name>

<value class="SampleSaveConfiguration">

<time>true</time>

<latency>true</latency>

<timestamp>true</timestamp>

<success>true</success>

<label>true</label>

<code>true</code>

<message>true</message>

<threadName>true</threadName>

<dataType>true</dataType>

<encoding>false</encoding>

<assertions>true</assertions>

<subresults>true</subresults>

<responseData>false</responseData>

<samplerData>false</samplerData>

<xml>false</xml>

<fieldNames>true</fieldNames>

<responseHeaders>false</responseHeaders>

<requestHeaders>false</requestHeaders>

<responseDataOnError>false</responseDataOnError>

<saveAssertionResultsFailureMessage>true</saveAssertionResultsFailureMessage>

<assertionsResultsToSave>0</assertionsResultsToSave>

<bytes>true</bytes>

<sentBytes>true</sentBytes>

<url>true</url>

<threadCounts>true</threadCounts>

<idleTime>true</idleTime>

<connectTime>true</connectTime>

</value>

</objProp>

<stringProp name="filename"></stringProp>

</ResultCollector>

<hashTree/>

<DurationAssertion guiclass="DurationAssertionGui" testclass="DurationAssertion" testname="Duration Assertion" enabled="true">

<stringProp name="DurationAssertion.duration">40</stringProp>

</DurationAssertion>

<hashTree/>

<HeaderManager guiclass="HeaderPanel" testclass="HeaderManager" testname="HTTP Header Manager" enabled="true">

<collectionProp name="HeaderManager.headers">

<elementProp name="" elementType="Header">

<stringProp name="Header.name">Content-Type</stringProp>

<stringProp name="Header.value">application/json</stringProp>

</elementProp>

</collectionProp>

</HeaderManager>

<hashTree/>

<ResponseAssertion guiclass="AssertionGui" testclass="ResponseAssertion" testname="Response Assertion" enabled="true">

<collectionProp name="Asserion.test_strings"/>

<stringProp name="Assertion.custom_message"></stringProp>

<stringProp name="Assertion.test_field">Assertion.response_data</stringProp>

<boolProp name="Assertion.assume_success">false</boolProp>

<intProp name="Assertion.test_type">16</intProp>

</ResponseAssertion>

<hashTree/>

</hashTree>

</hashTree>

</hashTree>

</jmeterTestPlan

4. Конфігурація бази даних та тестування запитів до неї

```
import org.junit.ClassRule;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.util.TestPropertyValues;
import org.springframework.context.ApplicationContextInitializer;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringRunner;
import org.testcontainers.containers.MySQLContainer;
import ukma.edu.ua.kmaudit.KmauditApplication;
import ukma.edu.ua.kmaudit.domains.entities.Course;
import ukma.edu.ua.kmaudit.domains.enums.EducationType;
import ukma.edu.ua.kmaudit.domains.enums.Faculty;
import ukma.edu.ua.kmaudit.domains.enums.Speciality;
import javax.transaction.Transactional;
import java.math.BigDecimal;
import static org.assertj.core.api.Assertions.assertThat;

@RunWith(SpringRunner.class)
@SpringBootTest(classes = KmauditApplication.class)
@ActiveProfiles("tc")
@ContextConfiguration(initializers =
{CourseRepositoryTCLiveTest.Initializer.class})
public class CourseRepositoryTCLiveTest extends CourseRepositoryTest {
   @ClassRule
   public static MySQLContainer MySQLContainer = new
MySQLContainer()
         .withDatabaseName("kmaudit")
         .withUsername("root")
         .withPassword("yarik1999n");

   @Test
   @Transactional
   public void allMathCoursesTest() {
      courseRepository.save(new Course(1, "MPA", (long)14,
BigDecimal.valueOf(13), BigDecimal.valueOf(13),
EducationType.CONTRACT,
Faculty.FACULTY_OF_COMPUTER_SCIENCES,
Speciality.APPLIED_MATHEMATICS, null));
      courseRepository.save(new Course(2, "OKA", (long)14,
BigDecimal.valueOf(13), BigDecimal.valueOf(13),
EducationType.CONTRACT,
```

```
Faculty.FACULTY_OF_COMPUTER_SCIENCES,
Speciality.APPLIED_MATHEMATICS, null));
        courseRepository.save(new Course(3, "JAVA", (long)14,
BigDecimal.valueOf(13), BigDecimal.valueOf(13),
EducationType.CONTRACT,
Faculty.FACULTY_OF_COMPUTER_SCIENCES,
Speciality.APPLIED_MATHEMATICS, null));
        courseRepository.save(new Course(4, "SQL", (long)14,
BigDecimal.valueOf(13), BigDecimal.valueOf(13),
EducationType.CONTRACT,
Faculty.FACULTY_OF_COMPUTER_SCIENCES,
Speciality.COMPUTER_SCIENCE, null));
        courseRepository.flush();

        int mathCourses = courseRepository.findAllMathCourses().size();

        assertThat(mathCourses).isEqualTo(3);
    }

    @Test
    @Transactional
    public void findCourseByIdTest() {
        courseRepository.save(new Course(1, "MPA", (long)14,
BigDecimal.valueOf(13), BigDecimal.valueOf(13),
EducationType.CONTRACT,
Faculty.FACULTY_OF_COMPUTER_SCIENCES,
Speciality.APPLIED_MATHEMATICS, null));
        courseRepository.flush();

        Course mathCourses = courseRepository.findCourseByID(1);

        assertThat(mathCourses.getId()).isEqualTo(1);
    }

    @Test
    @Transactional
    public void findCourseHoursLessTest() {
        courseRepository.save(new Course());
        courseRepository.save(new Course(1, "MPA", (long)14,
BigDecimal.valueOf(13), BigDecimal.valueOf(22),
EducationType.CONTRACT,
Faculty.FACULTY_OF_COMPUTER_SCIENCES,
Speciality.APPLIED_MATHEMATICS, null));
        courseRepository.save(new Course(1, "OKA", (long)14,
BigDecimal.valueOf(13), BigDecimal.valueOf(24),
EducationType.CONTRACT,
Faculty.FACULTY_OF_COMPUTER_SCIENCES,
```

```
Speciality.APPLIED_MATHEMATICS, null));
        courseRepository.save(new Course(1, "JAVA", (long)14,
BigDecimal.valueOf(13), BigDecimal.valueOf(33),
EducationType.CONTRACT,
Faculty.FACULTY_OF_COMPUTER_SCIENCES,
Speciality.APPLIED_MATHEMATICS, null));
        courseRepository.save(new Course(1, "SQL", (long)14,
BigDecimal.valueOf(13), BigDecimal.valueOf(31),
EducationType.CONTRACT,
Faculty.FACULTY_OF_COMPUTER_SCIENCES,
Speciality.APPLIED_MATHEMATICS, null));
        courseRepository.flush();

        int mathCourses = courseRepository.findCourseHoursLess(30).size();

        assertThat(mathCourses).isEqualTo(2);
    }
@Test
@Transactional
public void insertCourseTest() {
    courseRepository.insertCourse(1,(long)11,"SQL",
            BigDecimal.valueOf(25), EducationType.CONTRACT,
            Faculty.FACULTY_OF_COMPUTER_SCIENCES,
BigDecimal.valueOf(35),
            Speciality.COMPUTER_SCIENCE);
    Course sql = courseRepository.findCourseByID(1);

    assertThat(sql).isNotNull();
    assertThat(sql.getCourseName()).isEqualTo("SQL");
}
    static class Initializer
            implements
ApplicationContextInitializer<ConfigurableApplicationContext> {
        public void initialize(ConfigurableApplicationContext
configurableApplicationContext) {
            TestPropertyValues.of(
                    "spring.datasource.url=" + MySQLContainer.getJdbcUrl(),
                    "spring.datasource.username=" +
MySQLContainer.getUsername(),
                    "spring.datasource.password=" +
MySQLContainer.getPassword()
            ).applyTo(configurableApplicationContext.getEnvironment());
        }
    }
}
```

```
import org.testcontainers.containers.MySQLContainer;

public class MySQlContainer extends MySQLContainer<MySQlContainer> {
    private static final String IMAGE_VERSION = "mysql:11.1";
    private static MySQlContainer container;

    private MySQlContainer() {
        super(IMAGE_VERSION);
    }

    public static MySQlContainer getInstance() {
        if (container == null) {
            container = new MySQlContainer();
        }
        return container;
    }

    @Override
    public void start() {
        super.start();
        System.setProperty("DB_URL", container.getJdbcUrl());
        System.setProperty("DB_USERNAME", container.getUsername());
        System.setProperty("DB_PASSWORD", container.getPassword());
    }

    @Override
    public void stop() {
    }
}
```

5. Тестування з використанням Mock()

```
import org.junit.Before;
import org.junit.Test;
import org.mockito.Mockito;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.context.annotation.Bean;
import java.math.BigDecimal;

import static org.assertj.core.api.Assertions.assertThat;

public class MockServiceTest {
    @TestConfiguration
    static class CourseServiceImplTestContextConfiguration {
        @Bean
        public CourseService courseService() {
            return new CourseServiceImpl();
        }
    }
    @Autowired
    private CourseService courseService;
    @MockBean
    private CourseRepository courseRepository;
    @Before
    public void setUp() {
        Course alex = new Course(1, "MPA", (long)14,
BigDecimal.valueOf(13),
                BigDecimal.valueOf(22), EducationType.CONTRACT,
Faculty.FACULTY_OF_COMPUTER_SCIENCES,
                Speciality.APPLIED_MATHEMATICS, null);

        Mockito.when(courseRepository.findById(alex.getId()))
                .thenReturn(java.util.Optional.of(alex));
    }
    @Test
    public void testFindById() {
        int id = 1;
        Course found = courseService.getById(id);

        assertThat(found.getId())
                .isEqualTo(id);
    }
}
```

6. Тестування з використанням вбудованої бази даних

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.core.env.Environment;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import
org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.sql.DataSource;

@Configuration
@EnableJpaRepositories(basePackages =
"ukma.edu.ua.kmaudit.repositories")
@PropertySource("persistence-course.properties")
@EnableTransactionManagement
public class CourseJPAConfig {

   @Autowired
   private Environment env;

   @Bean
   public DataSource dataSource() {
      DriverManagerDataSource dataSource = new
DriverManagerDataSource();

dataSource.setDriverClassName(env.getProperty("jdbc.driverClassName"));
      dataSource.setUrl(env.getProperty("jdbc.url"));
      dataSource.setUsername(env.getProperty("jdbc.user"));
      dataSource.setPassword(env.getProperty("jdbc.pass"));

      return dataSource;
   }
}
```

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import
org.springframework.test.context.support.AnnotationConfigContextLoader;
import org.springframework.transaction.annotation.Transactional;
import ukma.edu.ua.kmaudit.configs.CourseJPAConfig;
import ukma.edu.ua.kmaudit.domains.entities.Course;
import ukma.edu.ua.kmaudit.domains.enums.EducationType;
import ukma.edu.ua.kmaudit.domains.enums.Faculty;
import ukma.edu.ua.kmaudit.domains.enums.Speciality;
import ukma.edu.ua.kmaudit.repositories.CourseRepository;

import javax.annotation.Resource;

import java.math.BigDecimal;

import static org.junit.Assert.assertEquals;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(
        classes = { CourseJPAConfig.class },
        loader = AnnotationConfigContextLoader.class)
@Transactional
public class InMemoryTest {

    @Resource
    private CourseRepository courseRepository;

    @Test
    public void saveCourseTest() {
        Course test = new Course(1, "MPA", (long)14, BigDecimal.valueOf(13),
                BigDecimal.valueOf(22), EducationType.CONTRACT,
Faculty.FACULTY_OF_COMPUTER_SCIENCES,
                Speciality.APPLIED_MATHEMATICS, null);
        courseRepository.save(test);

        Course test2 = courseRepository.findById(1).orElse(null);
        assert test2 != null;
        assertEquals("MPA", test2.getCourseName());
    }
}
```