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

Розробка нотифікаційного сервісу

Текстова частина до курсової роботи
за спеціальністю «Інженерія програмного забезпечення» 121

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

_____

(підпис)

“____” _____ 2021 р.

Виконав студент 4-го курсу
Смакула Р.В.
“____” _____ 2021 р.

Київ 2021

Календарний план виконання курсової роботи

Тема: Створення нотифікаційного сервісу

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Розгляд та аналіз проблеми та перегляд існуючої архітектури застосунку	Вересень-жовтень 2020р.	
2.	Розробка архітектури та бізнес логіки сервісу	Жовтень-грудень 2020р.	
3.	Підбір технологій та розробка back-end частини	Грудень-лютий 2020-2021р.	
4.	Підбір технологій та розробка front-end частини	Лютий-березень 2021р.	
5.	Написання текстової частини	Березень 2021р.	
6.	Перегляд праці науковим керівником	Березень 2021р.	
7.	Підготовка до презентації роботи	Квітень 2021р.	
8.	Презентація курсової роботи		

Студент Смакула Р.В _____

Керівник Глибовець А.М. _____

“_____” _____ 2021 р.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ТЗ – Технічне Завдання

QA – Quality assurance

SRP – Single-responsibility principle

SOLID – S (Single-responsibility principle), O (Open-closed principle), L (Liskov substitution principle), I (Interface segregation principle), D (Dependency inversion principle)

STOMP – Simple Text Oriented Messaging Protocol

JMS – Java Message Service

TDD – Test-driven development

ATDD - Acceptance test driven development

BDD - Behavior driven development

HTML – HyperText Markup Language

API – Application programming interface

АНОТАЦІЯ

Робота з створення нотифікаційного сервісу має на меті надати інструмент для надсилання повідомлень усім типам користувачів хмарного застосунку для запису студентів на курсові роботи.

Для будь-якого програмного застосунку надзвичайно важливими є його незалежність від сторонніх змін, легкість у розробці, публікуванні, а також у найбільш затратному процесі – підтримці. На всіх цих критеріях був поставлений акцент під час розробки сервісу. Також бралося до уваги можливість витримувати великі навантаження і здатність до розширення.

Для розробки розглядалися такі технології як SpringBoot, ActiveMQ, Thymeleaf.

ВСТУП

У сучасному світі все більше і більше процесів стають цифровими. Це робиться з багатьох причин. По-перше, такі процеси надають нам доступ до інформації незалежно від місцезнаходження. Така перевага дає можливість системі залучати більше користувачів. По-друге, швидкість наших дій в цифровій системі зростає, що економить нам час. По-третє, всі правила організації стають прозорими і однаковими для всіх. Цей чинник зменшує ймовірність порушення прав користувачів до надзвичайно низького рівня.

У Києво-Могилянській академії існує декілька хмарних сервісів. Найбільш поширеним і відомим серед студентів є DistEdu і САЗ НаУКМА. Неможливо переоцінити важливість, яку вони мають для університету. DistEdu зробив поширення матеріалів, оцінення і завантаження робіт, складання контрольних та іспитів простою буденною справою. САЗ НаУКМА надав змогу студентам зменшити кількість документів у паперовому вигляді і час проведений для запису на дисципліни в рази.

Виходячи з відгуків студентів та викладачів НаУКМА, можна сказати, що процес запису на курсові роботи завдає багато труднощів. Зараз це відбувається наступним чином: викладач подає на кафедру список з темами робіт, які він пропонує, далі студент звертається до викладача з пропозицію написання курсової роботи. Викладач або погоджується, або відмовляє студентові. Проте цей процес дуже важко контролювати, оскільки записи відбуваються віддалено або безпосередньо після очної зустрічі. Також складність додає той факт, що в залежності від ступеня викладача, він може бути науковим керівником у різній кількості проектів. Важливим аспектом, який має бути реалізованим є сприяння конкуренції серед зацікавлених у темі студентів.

Як вирішити проблему запису на курсові роботи? Одним з найбільш очевидних рішень є створення хмарного сервісу запису та оцінювання курсових робіт, частиною якого є нотифікаційний сервіс.

Для чого потрібний нотифікаційний сервіс? Сервіс розсилання повідомлення має на меті інформування студентів і керівників про події, які мають відношення до їхнього проекту. Такими повідомленнями можуть бути: запит на написання курсової роботи, повідомлення про схвалення або відхилення запиту на написання курсової роботи, а також багато інших.

Дана робота складається з чотирьох частин.

Перша частина містить дослідження шаблонізаторів у SpringBoot. В ній розглянуто способи застосування інструменту Thymeleaf для створення шаблонів повідомлень.

Друга частина описує методи створення Unit і Integration тестів. У ній наведено приклади розв’язання проблем, які можуть виникнути при роботі над тестуванням застосування.

У третій частині описується методи комунікації сервісів у хмарному середовищі. Зокрема, розглядається популярний брокер повідомлень ActiveMQ.

Четверта частина містить опис практичного завдання.

ШАБЛОНІЗАТОРИ У SPRING BOOT

1.1 Базовий опис шаблонізаторів

Шаблонізатор – інструмент, який дозволяє використовувати статичні шаблонні файли вашого застосунку. В процесі виконання програми шаблонізатор підставляє значення змінних в шаблонному файлі і перетворює його на HTML файл. Цей підхід полегшує дизайн HTML сторінки.

На рисунку 1 вказано принцип роботи шаблонізатора у застосунку розробленого за допомогою мови програмування Java.

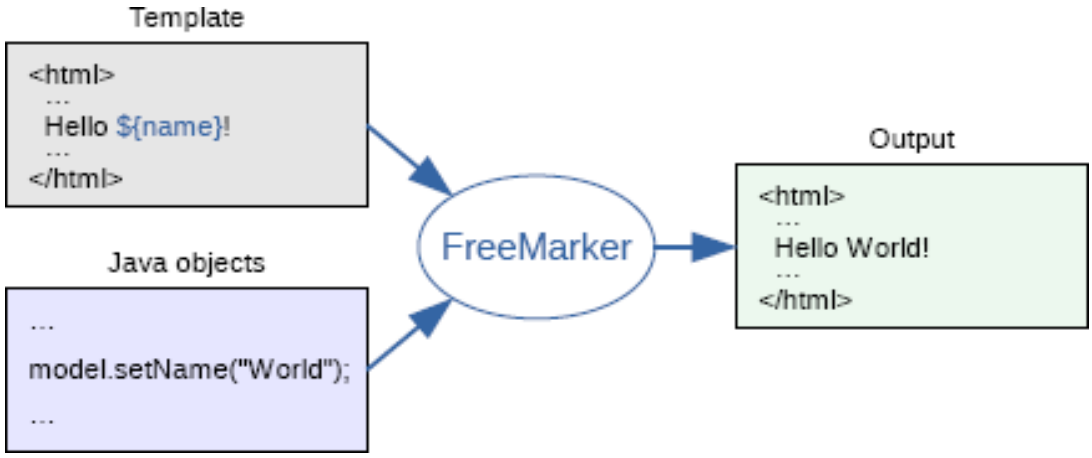

Рисунок 1 [1]

1.2 Різниця між шаблонізаторами

Розробник має можливість обирати серед великої кількості шаблонізаторів. Здебільшого вони всі відрізняються тим, на яких мовах програмування вони підтримуються. Також варіанти відрізняються між собою функціоналом, який вони надають, проте практично у кожному наявна підтримка змінних, функцій, включень, включень при умові, циклів, порівнянь і присвоєнь. Лише в деяких шаблонізаторах реалізовано наслідування.

На момент виконання курсової роботи найпопулярнішими серед програмістів Java були такі шаблонізатори як Thymeleaf, JavaServer Pages, Groovy, FreeMarker, Jade.

Engine	License	Variables	Functions	Includes	Conditional includes	Looping	Evaluation	Assignment	Inheritance
Thymeleaf	Apache	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
JavaServer Pages	CDDL + GNU GPL	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
FreeMarker	Apache	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Groovy	Apache	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Jade	-	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Рисунок 2 [2]

Для створення нотифікаційного сервісу використовувався фреймворк SpringBoot. Тому під час обирання шаблонізатора бралося до уваги кількість документації і прикладів у вільному доступі для інтеграції з ним. Практично кожне джерело вказувало на доцільність використання Thymeleaf у проекті, оскільки для його підключення і використання потрібно зробити лише декілька кроків.

1.3 Шаблонізатор Thymeleaf

1.3.1 Визначення

Thymeleaf – новітній Java шаблонізатор на стороні сервера, який можна використовувати як для веб-застосунків, так і для автономних середовищ. [4]

1.3.2 Опис

Основною метою Thymeleaf є створення можливості використання шаблонів, які написані на схожій до розмовної мові (natural templates), з подальшою конвертацією їх у HTML, який можна відображати у браузерах, а також переглядати як статичний прототип, що дозволяє посилити співпрацю в командах розробників.

Завдяки модулям для фреймворку Spring, безлічі інтеграцій з інструментами для розробки та можливості написання власного функціоналу, Thymeleaf ідеально підходить для сучасної веб-розробки HTML5 в середовищі з JVM.

Розширення для покращення умов використання Thymeleaf наявні у всіх популярних середовищах розробки, таких як IntelliJ IDEA, Eclipse та інших.

1.3.3 Базовий синтаксис

Синтаксис шаблонізатора Thymeleaf надзвичайно схожий до звичайного HTML синтаксису з додатковими тегами, атрибутами та інструкціями. Тому не потрібно багато часу для вивчення засобу його використання і початку його застосування.

1.3.3.1 Діалекти

Thymeleaf - надзвичайно розширюваний шаблонізатор, який дозволяє визначити та налаштувати спосіб обробки шаблонів до тонких деталей, наприклад, визначення власних тегів і атрибутів.

Об'єкт, який застосовує певну логіку до артефакту розмітки (тег, текст, коментар), називається процесором, і набір цих процесорів разом з деякими додатковими артефактами - це те, з чого діалект зазвичай складається. Основна бібліотека Thymeleaf надає діалект, який називається Standard Dialect, функціонал якого задовольняє більшість користувачів.

Також можливим є використання SpringStandard Dialect, який має адаптації для кращого використання деяких функцій Spring (наприклад, використання Spring Expression Language або SpringEL замість OGNL).

Однією з переваг Thymeleaf є можливість переглядання статичного контенту у веб-браузері, який просто ігнорує конструкції, які не є частиною HTML.

1.3.3.2 Використання текстових ресурсів і локалізація

Використовувати багатонаціональні текстові ресурси в Thymeleaf дуже просто. Все, що потрібно зробити, це написати назву ресурсу замість ? в наступному виразі #{?}.

```
<p th:text="#{hello.world}">Hello world!</p>
```

Значення за замовчуванням для тегу <p/> служить для відображення статичного контенту.

1.3.3.3 Використання змінних

Використання змінних дуже схоже до використання текстових ресурсів. Тепер лише потрібно замість # використати $. Тобто підставити назву змінної замість ? у наступний вираз ${?}.

```
<p th:text="${language}">Українська</p>
```

Значення за замовчуванням знову використовується для відображення статичного контенту.

1.3.3.4 Інший функціонал

Описувати функціонал Thymeleaf можна надзвичайно довго. Даний шаблонізатор також дозволяє застосовувати вставлення фрагментів, порівняння, вставлення фрагментів в залежності від результату порівняння, ітерації, оптимізацію через пізній доступ до даних, наслідування шаблонів, арифметичні операції та багато іншого.

1.3.4 Спосіб використання у SpringBoot програмі

У даному прикладі використовується мова програмування Java. Maven обрано як система автоматичного збирання.

Після того, як було підключено SpringBoot залежності до проекту, потрібно також вказати залежність бібліотеки Thymeleaf.

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

Налаштуємо використання локалізованих текстових ресурсів. Для цього потрібно створити клас, додати до нього анотації @Configuration і написати метод з анотацією @Bean, який буде вказувати як саме SpingBoot має створювати об’єкти типу MessageSource. Для цього використаємо його реалізацію ResourceBundleMessageSource. Також вкажемо додаткові конфігурації текстових ресурсів: префікс назви файлу, формат кодування символів і використання коду ресурсу як його значення у разі його відсутності. У прикладі використано дві локалізації: українську і англійську.

```
@Configuration
public class LocaleConfiguration {

    public static final Locale[] locales = new Locale[]{new Locale("us"),
new Locale("uk")};

    @Bean
    public ResourceBundleMessageSource messageSource() {
        ResourceBundleMessageSource messageSource = new
ResourceBundleMessageSource();
        messageSource.setBasename("messages");
        messageSource.setDefaultEncoding("UTF-8");
        messageSource.setUseCodeAsDefaultMessage(true);
        return messageSource;
    }

}
```

Створимо два файли з розширенням .properties для текстових ресурсів. Перший файл з назвою messages для англомовних ресурсів, другий – messages_uk для україномовних ресурсів. Для додавання підтримки інших мов потрібно створити файл з назвою messages_?, де замість ? підставити код локалізації.

```
hello.world="Hello World"
```

```
hello.world="Привіт світ"
```

Для використання шаблонізатора у коді потрібно використати об’єкт класу, який задовольняє інтерфейс ITemplateEngine. Можна використати клас SpringTemplateEngine, який наявний у компоненті, який ми підключали через Maven. В інтерфейса ITemplateEngine є метод process, який приймає параметри String template, IContext context та повертає оброблений, готовий до відображення в браузері HTML код.

```
Locale locale = LocaleConfiguration.locales[1];
Map<String, Object> variablesMap = new HashMap<>();
variablesMap.put("language", locale.getLanguage());
Context context = new Context(locale, variablesMap);
String html = templateEngine.process(templatePath, context);
```

SpringTemplateEngine за замовчуванням використовує такий шлях для збереження шаблонів: classpath:templates, проте це можливо налаштувати.

Далі потрібно створити шаблон для відображення. В батьківському тезі html вказуємо простір імен тегів Standard Dialect.

```
<html lang="en" xmlns:th="http://www.thymeleaf.org">
```

Створюємо два теги <p/>, в одному задаємо назву мову, значення якої буде братися із змінної, в другому задаємо назву локалізованого текстового ресурсу.

```
<p th:text="${language}">English</p>
<p th:text="#{hello.world}">Default text!</p>
```

При статичному перегляді сторінка буде виглядати наступним чином.

English

Default text!

Рисунок 3

При обробленому шаблонізатором перегляді сторінка буде виглядати ось так.

uk

Привіт світ

Рисунок 4

1.4 Результат дослідження

Дослідивши технології шаблонізаторів, можна сказати, що вони полегшують процес розробки розмітки, її читабельності, їхнє освоєння займає невелику кількість часу, а також вони дають змогу повторно використовувати написані компоненти, що є дуже важливим фактором.

НАПИСАННЯ UNIT І INTEGRATION ТЕСТІВ

2.1 Визначення

Unit testing — це метод тестування програмного забезпечення, який полягає в окремому тестуванні кожного модуля коду програми. Модулем називають найменшу частину програми, яка може бути протестованою. У процедурному програмуванні модулем вважають окрему функцію або процедуру. В об'єктно-орієнтованому програмуванні – метод. [6]

Integration testing — це фаза тестування програмного забезпечення, під час якої окремі модулі програми комбінуються та тестуються разом, у взаємодії. Інтеграційне тестування виконується після модульного тестування та перед верифікацією та валідацією ПЗ. [7]

2.2 Опис та порівняння

В сьогодення дедалі більшої популярності набирає створення тестів для застосунку. З’явилися навіть методики написання програми, які пов’язані з цим. Одна з них є Test-driven development, яку буде детальніше описано в одному з наступних розділів.

Існує велика кількість типів тестів, проте найбільш популярними і важливими є Unit і Integration тести. Потрібно брати до уваги їхній порядок написання: спочатку Unit, а потім Integration. Також важливою є відсоткова кількість покриття коду тестами: для Unit рекомендується 90%, в той час як для Integration – 80%.

Тестування можна проводити автоматично або мануально за допомогою різних фреймворків. Автоматизація тестування призводить до пришвидшення і спрощення цього процесу.

Вищезгадані методи тестування відрізняються великою кількістю аспектів. Нижче наведене порівняльну таблицю.

Unit tests	Integration tests
Має на меті відтестувати окремі частини (модулі, функції…)	Має на меті відтестувати сукупну роботу різних частин
White Box Testing (нутрощі відкриті для тестування)	Black Box Testing (нутрощі закриті для тестування)
Виконується будь-коли	Виконується зазвичай після Unit testing
Тестується лише функціонал компонента, але не взаємодію з іншими	Тестується лише взаємодія компонента з іншими
Виконуються без реалізації залежностей компонента. Часто використовуються Mock компонентів	Реалізація залежностей необхідна. Mock компонентів зазвичай не використовується
Виконується розробником	Створюється розробником або QA інженером
Легко знайти помилки	Важко знайти помилки
Легко розробляти і підтримувати	Важко розробляти і підтримувати

Рисунок 5 [8]

2.3 Важливість і причини створення

Коли програміст створює програмний продукт, він насамперед дбає про те, щоб зміни, які можуть статися з функціоналом, було легко реалізувати і швидко доставити до кінцевих користувачів. При цьому важливою є мінімізація кількості компонентів програми, які будуть змінені, оскільки більше змінюваних компонентів – більша потенційна кількість помилок, які можуть виникнути. Цей стиль написання програмного коду можна почерпнути з принципів дизайну SOLID, а саме з букви S, яка позначає принцип SRP (Single Responsibility Principle). [9] Він вказує розробнику на те, що кожен компонент програми має мати лише одну причину для змін. Також програміст має дбати про те, щоб застосунок виконував усі вимоги, які зазначені у ТЗ продукту.

Як спростити програмним інженерам підтримку існуючих програмних систем і дати їм можливість випускати оновлення, не думаючи про те, що буде пошкоджена робота інших компонентів? Часто для вирішення цієї проблеми створюють автоматизоване тестування компонентів програми. Нижче наведено, яку саму користь можна отримати від покриття коду тестами.

- Розробка стає гнучкішою, можна робити зміни в коді часто і впевнено
- Покращується якість програмного забезпечення, з'являється змога найти дефекти у системі, не витрачаючи великих ресурсів
- Тести дозволяють знайти помилки на ранніх стадіях розробки
- Створюється документація програмних модулів для сторонніх користувачів, що полегшує їм їх розуміння у використанні
- Полегшується пошук помилок, а також їхнє виправлення
- Покращується дизайн застосунку, коли архітектор системи думає про те, як потім можна буде протестувати код
- Зменшується ціна розробки системи, оскільки помилки знаходяться на ранніх стадіях розробки і зразу ж виправляються

2.4 Test-driven development

2.4.1 Визначення

Test-driven development – стиль програмування, у якому процеси написання коду, написання тестів і створення дизайну застосунку тісно взаємодіють один з одним.

2.4.2 Історія

Ідея написання тестів перед програмним кодом існує вже доволі довго. З 1976 року з'являлися різні публікації, у яких описувалися аксіоми

тестування. Одна з них, наприклад, вказує, що розробники ніколи не має тестувати свій код. Дуже важливою людиною, яка вплинула на напрямок розвитку розробки програмного забезпечення є Kent Beck. У 1994 році він написав фреймворк SUnit для автоматичного тестування на мові програмування Smalltalk, яка в ті часи була надзвичайно популярною. У 2003 Kent написав книжку “Test Driven Development: By Example”, яка, на мою думку, є основним джерелом для розробників, з якого можна зачерпнути знання про TDD. У 2006 році у TDD почали з’являтися наслідники такі як ATDD (Acceptance Test Driven Development) та BDD (Behavior Driven Development). [11]

2.4.3 Опис

Test-driven development починається зі створення дизайну програми і розробки тестів для кожної її невеликої частини. За принципами TDD розробник має писати новий код лише тоді, коли тестування не проходить. Це дозволяє уникнути дублювання коду. Нижче наведено кроки розробки застосунку за концептами TDD.

1. Додати тест
2. Запусти всі тести і перевірити, чи всі вони пройшли успішно
3. Якщо всі завершилися успішно, то перейти до кроку 1, інакше написати код для того, щоб всі тести проходили.
4. Запустити всі тести і зробити правки в коді, якщо потрібно
5. Повторити весь цикл з кроку

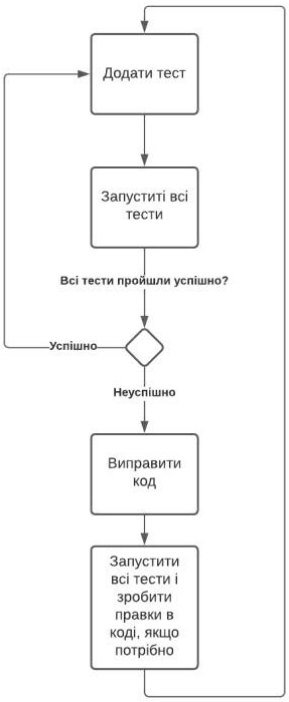

Рисунок 6

2.4.4 Переваги та недоліки

Як і у кожного стиля розробки програмного забезпечення, у TDD існують свої плюси, а також свої мінуси. Це варто розглядати при обранні процесів для створення застосунку. Нижче наведено переваги і недоліки застосування TDD. [12]

Переваги.

- Зменшення інтенсивності знаходження помилок після запуску продукту
- Зменшення кількісті роботи у фінальних стадіях розробки проекту
- Покращення дизайну застосунка, менша зв’язність і більша зв’язність компонентів

Недоліки.

- Програмісти забувають дотримуватися принципів розробки TDD, забувають запускати тести часто, пишуть забагато тестів за один раз, пишуть тривіальні тести, створюють занадто великі тести

- Через складність системи, тести можуть обраховуватися довгий проміжок часу
- Часто виникає необхідність у хорошому менеджері, який буде слідкувати за дотриманням TDD

2.5 Приклади використання у SpringBoot

Оскільки Java вже більше 10 років займає найбільшу частину на ринку інструментів для написання back-end застосунку, на ній було написано надзвичайно велику кількість фреймворків для різних цілей. SpringBoot – фреймворк, за допомогою якого можна зробити велику кількість речей. Налаштування автоматичного тестування не є винятком.

2.5.1 Unit тестування

Для того, щоб мати змогу автоматично виконувати тести, все, що потрібно зробити, це додати дві залежності у файл pom.xml.

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.4.2</version>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.4.2</version>
</dependency>
```

Наступним кроком потрібно створити клас і у ньому помісти тести, методи з анотацією @Test. Для Unit тестування програмісту часто доводиться створювати Mock класи, щоб забезпечити незалежне тестування модуля. Нижче наведено приклад тестування реалізації з використанням Mock класів і налаштуванням автоматизації виконання тестів.

У практичній частині даної роботи було створено інтерфейс MessageFactory, який вміє створювати потрібного нащадка інтерфейса Message.

```
public interface MessageFactory {

    Message getMessage(final EmailType emailType, final String fromEmail,
final String toEmail);

}
```

Було написано клас MessageFactoryImpl, який є реалізацію вищенаведеного інтерфейсу. MessageFactoryImpl залежить від ViewProvider, інтерфейса, який вміє завантажувати параметризований шаблон і повертати HTML текст.

```
public interface ViewProvider {

    String buildContent(String templatePath);

    String buildContent(String templatePath, Map<String, Object>
variablesMap);

    String buildContent(String templatePath, Map<String, Object>
variablesMap, Locale locale);

}
```

Потрібно було створити тест, у якому реалізація цього інтерфейсу не мала б значення, тому було написано клас ViewProviderMock.

```
@Setter
public class ViewProviderMock implements ViewProvider {

    private BuildContentDelegate buildContentDelegate = variablesMap -> "";

    @Override
    public String buildContent(String templatePath) {
        return buildContent(templatePath, Collections.emptyMap());
    }

    @Override
    public String buildContent(String templatePath, Map<String, Object>
variablesMap) {
        return buildContent(templatePath, variablesMap, Locale.US);
    }

    @Override
```

```
    public String buildContent(String templatePath, Map<String, Object>
variablesMap, Locale locale) {
        return buildContentDelegate.buildContent(variablesMap);
    }

    public interface BuildContentDelegate {
        String buildContent(Map<String, Object> variablesMap);
    }

}
```

Все, що залишилося зробити, це написати тест з використання MessageFactoryImpl і ViewProviderMock. Нижче наведено приклад такого тесту.

```
public class MessageFactoryTest {

    private final ViewProviderMock viewProvider = new ViewProviderMock();
    private final MessageFactory messageFactory = new
MessageFactoryImpl(viewProvider, new
LocaleConfiguration().messageSource());

    private final static String predefinedSenderEmail = "roman@gmail.com";
    private final static String predefinedReceiverEmail = "user@gmail.com";

    @Test
    public void createNewCourseWorkRequestTest() {
        viewProvider.setBuildContentDelegate(variablesMap -> {
            assert (variablesMap.containsKey("ukmaLogo"));
            assert (variablesMap.containsKey("title"));
            assert (variablesMap.containsKey("description"));
            assert (variablesMap.containsKey("language"));
            return "";
        });
        Message message =
messageFactory.getMessage(EmailType.NEW_COURSE_WORK_REQUEST,
predefinedSenderEmail, predefinedReceiverEmail);
        assert (predefinedSenderEmail.equals(message.getFromEmail()));
        assert (predefinedReceiverEmail.equals(message.getToEmail()));
    }

}
```

2.5.2 Integration тестування

Integration тестування у SpringBoot здебільшого означає те, що буде запущений тестовий сервер з певними тестовими конфігураціями. Існує багато варіантів анотацій для тестів, які відрізняються між собою різним конфігуруванням сервера. Для того, щоб їх використовувати потрібно до

залежностей, які використовувалися у Unit тестуванні, додати ще одну залежність, яка наведена нижче.

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

Хорошою практикою є створення базової конфігурації тестового середовища і подальшого використання її у всіх Integration тестах. У практичній частині було створено класи TestApplication і BaseTest, у яких виконувалася така конфігурація.

```
@Primary
@SpringBootApplication(scanBasePackages =
{"ua.edu.ukma.fin.notifications"})
public class TestApplication {
}
```

```
@ExtendWith(SpringExtension.class)
@SpringBootTest(classes = {TestApplication.class}, webEnvironment =
SpringBootTest.WebEnvironment.DEFINED_PORT)
@ContextConfiguration
public class BaseTest {
}
```

Пізніше класу, у якому безпосередньо міститься тестові методи, потрібно унаслідувати BaseTest. Нижче наведено приклад тестування створення і надсилання повідомлення на електронну пошту.

```
public class MessageFactoryTest extends BaseTest {

    @Autowired
    private EmailTransport emailTransport;

    @Autowired
    private MessageFactory messageFactory;

    private final String predefinedSenderEmail = "noreply@ukma.edu.ua";
    private final String predefinedReceiverEmail = "r.smakula@ukma.edu.ua";

    @Test
    public void sendSimpleMessageTest() throws EmailTransportException {
        Message message =
messageFactory.getMessage(EmailType.NEW_COURSE_WORK_REQUEST,
predefinedSenderEmail, predefinedReceiverEmail);
        message.sendVia(emailTransport);
```

```
    }

}
```

2.6 Результат дослідження

Після дослідження методів написання Unit та Integration тестів можна зробити висновок, що вони несуть багато користі для розробників. Було з’ясовано, що найважливіше в тестах це їхня якісь, а не кількість, адже погані тести можуть сильно нашкодити. Також було досліджено використання стилю TDD: в яких випадках його слід використовувати, його переваги і недоліки.

БРОКЕР ПОВІДОМЛЕНЬ ActiveMQ

Під час створення архітектури застосунку програмісту потрібно виділити, які компоненти будуть наявні у системі. Він може користуватися різними принципами розробки дизайну, тобто вирішувати як саме розділяти програму. Спільними між цими принципами є варіації одиниць архітектури, що саме вони представляють собою. Можна виділити три види декомпозиції, які наведено нижче.

- Розбиття на компоненти, які виконуються в одному просторі пам’яті комп’ютера (наприклад, package у структурі програми Java)
- Розбиття на модулі, які виконуються в одному просторі пам’яті комп’ютера, проте підключаються до залежної частини скомпільовані (наприклад, .jar файл у мові програмування Java)
- Розбиття на мікросервіси, які здебільшого виконуються на різних машинах і комунікація між компонентами відбувається через локальну мережу або інтернет

У практичній частині здебільшого використовувався третій вид декомпозиції. Таке рішення було зроблене на основі вимог до платформи. Під час використання мікросервісної архітектури виникає проблема передачі повідомлень між компонентами. У випадку нотифікаційного сервісу ці проблеми були наступними:

- Гарантованість доставки повідомлення від клієнта
- Витримування навантаження від користувачів
- Зрозумілий і легкий у використанні інтерфейс передачі повідомлення

Проаналізувавши вищенаведені проблеми, було зроблено рішення використати брокер повідомлень ActiveMQ.

3.1 Визначення

Apache ActiveMQ — відкрите програмне забезпечення (ліцензія Apache 2.0), Message Broker, який повністю реалізує вимоги Java Message Service 1.1 (JMS). Забезпечує особливості, важливі для програми, яка використовується на великих підприємствах («Enterprise Features»), таких як кластеризація, множинні історії повідомлень та здатність використовувати будь-яку БД (базу даних) як JMS persistence provider, крім того забезпечує відновлення у разі поновлення порушеної сесії (persistency) для віртуальної машини (VM), кешу (cache), і журналу (journal). [13]

3.2 Опис

ActiveMQ був створений його засновниками з LogicBlaze у 2004 році як брокер повідомлень з відкритим кодом, розміщеним у CodeHaus. Код та торгову марку ActiveMQ було передано Apache Software Foundation у 2007 році, де засновники продовжили розробку над кодовою базою разом із спільнотою Apache.

ActiveMQ на даний момент є одним з найпопулярніших брокерів повідомлень, які використовуються програмістами. У вільному доступі наявно багато інформації по його використанню. Даний брокер повідомлень можна використовувати у програмах Java, C, C++, Phyton, .Net та деяких інших. Існує два основних види розгортання ActiveMQ сервера: вбудоване і відокремлене. У кожного метода є свої переваги і недоліки.

ActiveMQ використовує декілька режимів для забезпечення високої доступності, які включають як і механізми блокування на рівні рядків файлової системи та бази даних, так і спільне використання сховища постійних даних через спільну файлову систему або справжню реплікацію за допомогою Apache ZooKeeper. Network of Brokers - горизонтальний механізм масштабування також підтримується. ActiveMQ дає змогу

використовувати відносно велику кількість транспортних протоколів, таких як OpenWire, STOMP, MQTT, AMQP, REST та WebSockets. [14]

3.3 Переваги і недоліки

На ринку брокерів повідомлень існує велика кількість альтернатив до ActiveMQ. Найпопулярніші серед них: RabitMQ, AmazonMQ, IBM MQ та інші. Ідеологічно вони розв’язують однакові проблеми, проте відрізняються здатностями до масштабування, підтримкою різним мов програмування, протоколами передачі даних і іншими принципами роботи.

Під час виконання практичної частини, коли потрібно було зробити вибір між ActiveMQ та іншими брокерами повідомлень, було розглянуто переваги і недоліки ActiveMQ.

Переваги.

- Легко масштабується, можна використати технологію Master/Slave для цього
- Підтримує стандарт JMS
- Підтримує протокол передачі STOMP через WebSocket, який надає можливості надсилати декілька слухачам певного topic одне повідомлення
- Забезпечує балансування навантаження на отримувачів повідомлень
- Велика кількість ресурсів наявна у відкритому доступі
- Простота використання

Недоліки.

- Не є найшвидшим брокером повідомлень, через велику кількість протоколів, які підтримуються і здатності до масштабування
- Потребує підтримки адміністратора

3.4 Оптимальна побудова архітектури сервісу

Нотифікаційний сервіс створювався таким чином, щоб його користувачі не знали про його внутрішні зміни, тобто можна легко було випускати нові версії і не впливати цим на зовнішніх клієнтів. Також бралося до уваги те, що може виникнути потреба змінити брокер повідомлень, тому користувач нічого не повинен знати про ActiveMQ. Було зроблене рішення про створення окремого модулю, який буде надавати інтерфейс нотифікаційного сервісу і його реалізацію (надсилання повідомлення у чергу).

```
@Slf4j
@RequiredArgsConstructor
public class EmailSender {

    private final String emailQueue;
    private final JmsTemplate jmsTemplate;

    public void sendEmailRequest(final EmailSendRequest request) {
        log.info("Send email request: {}", request);
        jmsTemplate.convertAndSend(emailQueue, request);
    }

}
```

Під час обміну повідомлень між двома мікросервісами потрібно встановити загальний формат повідомлень, який буде зрозумілий як і для того, хто надсилає повідомлення, так і для того, хто його приймає. Для вирішення цієї проблеми існує Message Broker. ActiveMQ реалізовує протокол розсилання повідомлень JMS (Java Message Service), який дає змогу легко зробити інтерфейс повідомлень у програмі Java. У практичній частині інтерфейси повідомлень, які надсилалися через чергу повідомлень, були винесені в окремий модуль, який використовувався як модулем, який надає інтерфейс для використання нотифікаційного сервісу, так і самим нотифікаційним сервісом. Приклад класу повідомлення наведено нижче.

```
@Value
@Builder
@RequiredArgsConstructor
public class EmailSendRequest implements Serializable {

    private static final long staticalVersionUID = -4005263471947053491L;

    private final String fromEmail;
    private final String toEmail;
    private final EmailType emailType;

}
```

Отже, у практичній частині застосунок було розбито на три модулі.

- notifications-api-client - модуль з інтерфейсом нотифікаційного сервісу
- notifications-model - модуль для визначення формату повідомлення для того, хто надсилає і отримує повідомлення
- notifications-service – модуль, в якому міститься основна логіка сервісу

3.5 Приклад використання у SpringBoot

ActiveMQ доволі легко інтегрувати із SpringBoot. На мою думку, доцільно звернути увагу на два аспекти інтеграції: використання у основній частині і написання Integration тестів.

3.5.1 Основна частина

У практичній частині було використано вбудовану ActiveMQ. Для використання її у проекті потрібно додати такі залежності.

```
<dependency>
    <groupId>javax.jms</groupId>
    <artifactId>javax.jms-api</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>activemq-broker</artifactId>
</dependency>
<dependency>
```

```
    <groupId>org.apache.activemq</groupId>
    <artifactId>activemq-client</artifactId>
</dependency>
```

Далі потрібно додати налаштування JMS, а саме налаштування MessageConverter, JmsTemplate і JmsListenerContainerFactory.

```
@Bean
MessageConverter jacksonJmsMessageConverter() {
    MappingJackson2MessageConverter converter = new
MappingJackson2MessageConverter();
    converter.setTargetType(MessageType.TEXT);
    converter.setTypeIdPropertyName("jms_type");
    converter.setObjectMapper(this.objectMapper);
    return converter;
}

@Bean
JmsTemplate jmsTemplate() {
   JmsTemplate template = new JmsTemplate();
   template.setConnectionFactory(this.connectionFactory);
   template.setMessageConverter(this.jacksonJmsMessageConverter());
   template.setDeliveryPersistent(true);
   return template;
}

@Bean
private JmsListenerContainerFactory<?> jmsListenerContainerFactory () {
    Listener listener = this.jmsProperties.getListener();
    DefaultJmsListenerContainerFactory factory = new
DefaultJmsListenerContainerFactory() {
        public DefaultMessageListenerContainer
createListenerContainer(JmsListenerEndpoint endpoint) {
            DefaultMessageListenerContainer listenerContainer =
(DefaultMessageListenerContainer)super.createListenerContainer(endpoint);
            listenerContainer.setBeanName(endpoint.getId());
            return listenerContainer;
        }
    };
    factory.setConnectionFactory(this.connectionFactory);
    factory.setConcurrency(listener.formatConcurrency());
    factory.setMessageConverter(this.jacksonJmsMessageConverter());
    return factory;
}
```

Для надсилання повідомлень у чергу потрібно використати клас EmailSender, який було описано у розділі про оптимальну архітектуру сервісу.

3.5.2 Тестування

Для написання Integration тесту з використанням ActiveMQ було написано у модулі notifications-service Mock класу EmailSender, який має надсилати повідомлення в чергу. Було застосовано клас BaseTest, про який описується у розділі про написання інтеграційних тестів на SpringBoot. Нижче наведено приклад тесту надсилання запиту на нову курсову до викладача.

```
public class QueueTest extends BaseTest {

    @Autowired
    private EmailSenderMock emailSenderMock;

    private static final String predefinedFromEmail =
"noreply@ukma.edu.ua";
    private static final String predefinedToEmail =
"smakula2012@gmail.com";

    @Test
    public void sendNewCourseWorkRequest() {
        emailSenderMock.sendEmailRequest(new
EmailSendRequest(predefinedFromEmail, predefinedToEmail,
EmailType.NEW_COURSE_WORK_REQUEST));
    }

}
```

3.6 Результати дослідження

Після даного дослідження можна зробити висновок, що ActiveMQ є чудовим інструментом для пересилання повідомлень між мікросервісами. Було досліджено переваги і недоліки цього брокера повідомлень, а також з’ясовано методи його використання у програмах написаних за допомогою фреймворку SpringBoot.

ОПИС ПРАКТИЧНОЇ ЧАСТИНИ

У практичній частині даної роботи було створено нотифікаційний сервіс для надсилання листів на електронну пошту. Було використано сучасні технології для розробки: SpringBoot – для бек-енд частини, Thymeleaf – для створення динамічних шаблонів і їх застосування, ActiveMQ – для передавання повідомлень до нотифікаційного сервісу.

4.1 Опис структури проекту

Проект складається з трьох модулів:

- notifications-api-client
- notifications-model
- notifications-service

4.1.1 Модуль notifications-api-client

Модуль notifications-api-client містить інтерфейс і реалізацію комунікації з нотифікаційним сервісом. Структура у нього доволі проста: він містить два класи, один з них EmailSender – клас для використання зовнішнім клієнтом, а інший NotificationClientConfiguration – клас застосування налаштувань до клієнта ActiveMQ. Структура класу EmailSender буде детально описана в розділі про опис API.

4.1.2 Модуль notifications-model

Модуль notifications-model має на меті надати спільний інтерфейс повідомлення для того, хто надсилає повідомлення до черги, і для того, хто його отримує. Модуль на даний момент містить лише класи EmailSendRequest і EmailType. Якщо у майбутньому виникне потреба у

створенні повідомлення з більшою кількістю параметрів, то потрібно буде створити клас-нащадок EmailSendRequest. Нижче на рисунку 7 наведено методи, які наявні у класах модуля.

Рисунок 7

4.1.3 Модуль notifications-service

Даний модуль містить бізнес логіку і конфігурацію сервісу. Він складається з класу NotificationApplication, який є вхідною точкою програми та пакетів, які перераховані нижче.

- config
- services
- settings

Пакет config містить класи, мета яких задати статичну конфігурацію SpringBoot. Ці класи мають такі назви.

- LocaleConfiguration
- MailSenderConfig
- JmsConfiguration

Пакет services містить бізнес логіку сервісу.

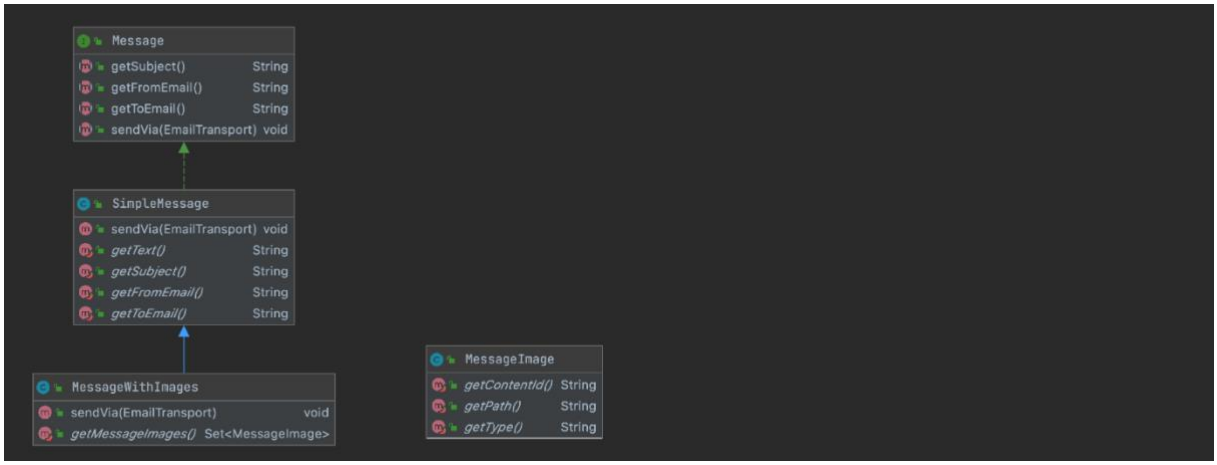

Рисунок 8

Рисунок 9

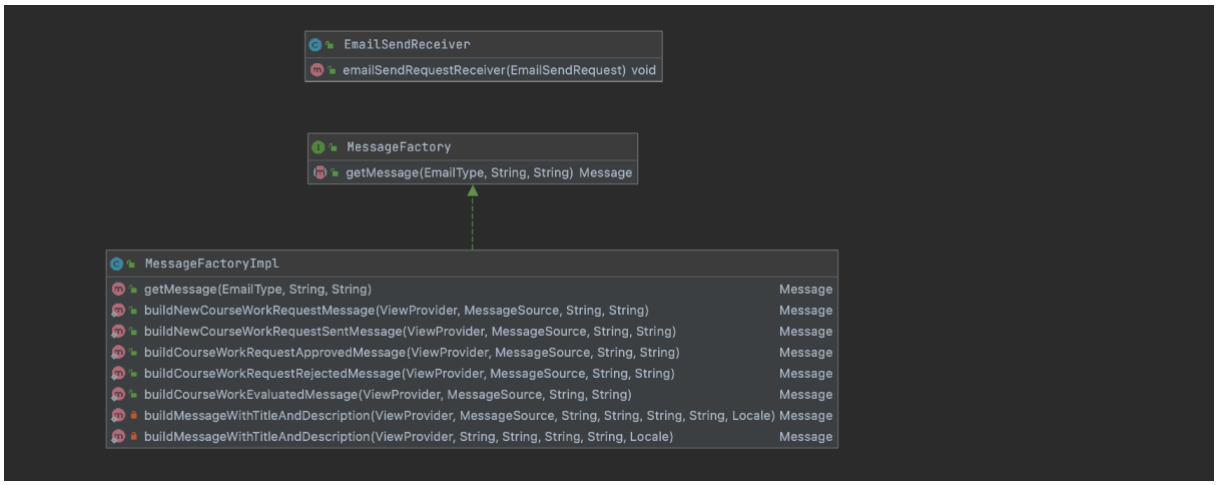

Рисунок 10

Пакет settings містить клас, який зчитує з файлу з розширенням .properties динамічні налаштування, які застосовувати до сервера.

Рисунок 11

4.2 Опис API

Під час розробки інтерфейсу нотифікаційного сервера робився акцент на його простоті і зрозумілості для користувача. Для його використання потрібно додати до залежностей проекту модуль notifications-api-client і задати значення змінній оточення з назвою email.queue, яка відповідає за назву отримувача повідомлення.

Клас EmailSender є реалізацією інтерфейса комунікації з нотифікаційним сервісом. Він містить методи, які вказані на рисунку 14.

Рисунок 12

Приклад надсилання повідомлення про запит на виконання курсової роботи наведено нижче.

```
@RequiredArgsConstructor
public class NotificationServiceUsageExample {

    private final EmailSender emailSender;

    public void sendNewCourseWorkRequestExample() {
        EmailSendRequest emailSendRequest = new
EmailSendRequest("from@gmail.com", "teacher@gmail.com",
EmailType.NEW_COURSE_WORK_REQUEST);
```

```
        emailSender.sendEmailRequest(emailSendRequest);
    }

}
```

Для перегляду інших прикладів використання інтерфейсу комунікації з нотифікаційним сервісом можна використати Integration тести, які розміщені у модулі notifications-service.

4.3 Опис фронт-енд частини

Фронт-енд частину проекту складається з шаблонів для надсилання повідомлень. Оскільки використовувався Thymeleaf, було створено лише один HTLM файл і повторно використано його для всіх типів повідомлень. Для шаблона, який міститься у файлі title_with_description_message.html можна динамічно задавати зображення логотипу університету, заголовок і опис повідомлення, а також назву університету. Для прикладу наведено два види повідомлень. На рисунку 15 зображено повідомлення про запит на виконання курсової роботи, а на рисунку 16 – повідомлення про схвалення запиту керівником.

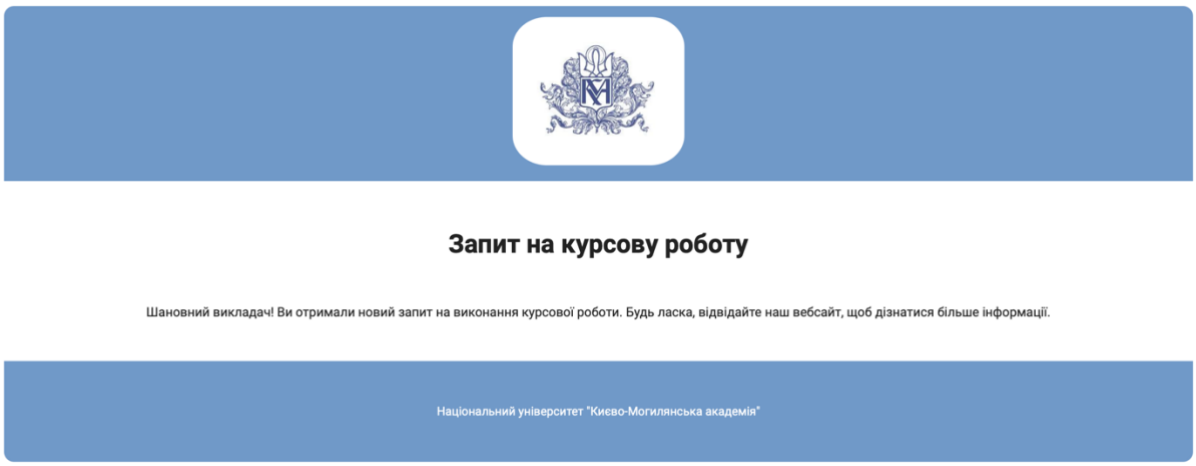

Рисунок 13


Запит на курсову роботу схвалено
Ваш запит на виконання курсової робота був схвалений викладачем.
Національний університет "Києво-Могилянська академія"

Рисунок 14

ВИСНОВОК

Після виконання даної роботи було створено інструмент для надсилання повідомлень на електронну пошту про події у хмарній системі запису на курсові роботи. Також проведено дослідження про оптимальну архітектуру такого сервісу відносно до змін, які можуть бути необхідними з часом.

Виконання цієї курсової роботи поглибило знання про методики написання тестів для застосунку. Також покращилося розуміння використання шаблонізатора у SpringBoot, стало зрозуміло чим слід керуватися під час його обрання. Було з’ясовано різні нюанси використання брокерів повідомлень, зокрема ActiveMQ.

Однією з найголовніших переваг цієї роботи є користь для університету, яка була принесена. Хочеться вірити, що схожих цифрових систем до системи запису студентів на курсові роботи у майбутньому буде з’являтися дедалі більше.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Apache FreeMarker - https://freemarker.apache.org
2. Порівняння шаблонізаторів - https://en.wikipedia.org/wiki/Comparison_of_web_template_engines
3. Шаблонізатори для SpringBoot - https://www.baeldung.com/spring-template-engines
4. Thymeleaf - https://www.thymeleaf.org
5. Thymeleaf - https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html
6. Модульне тестування - https://uk.wikipedia.org/wiki/Модульне_тестування
7. Інтеграційне тестування - https://uk.wikipedia.org/wiki/Інтеграційне_тестування
8. Unit і Integration тести - https://www.guru99.com/unit-test-vs-integration-test.html
9. Signle-responsibility principle - https://en.wikipedia.org/wiki/Single-responsibility_principle
10. Переваги Unit тестування - https://dzone.com/articles/top-8-benefits-of-unit-testing
11. TDD - https://en.wikipedia.org/wiki/Test-driven_development
12. TDD – https://www.agilealliance.org/glossary/tdd
13. ActiveMQ - https://uk.wikipedia.org/wiki/Apache_ActiveMQ
14. ActiveMQ - https://activemq.apache.org