Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів створення та розгортання багатомодульних Java застосунків
(тема)

Виконав:
студент (ка) 2 курсу, групи ІПЗм-22-1

Верес М.Д.
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)

Тип програми освітньо-наукова

Керівник доц. Голян Н.В.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри _____ З.В.Дудар
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Вересу Максиму Дмитровичу
(прізвище, ім'я, по батькові)

1. Тема роботи «Дослідження методів створення та розгортання багатомодульних Java застосунків»

Затверджена наказом по університету від 29.03.2024 №250Ст

2. Термін подання студентом роботи до екзаменаційної комісії 13.06.2024

3. Вихідні дані до роботи опис багатомодульної архітектури, вимоги та практики до розробки архітектури застосунків на Java, мова програмування Java, технології Java 9, Ant, Maven, Gradle, середовище розробки IntelliJ IDEA 2023.3

4. Перелік питань, що потрібно опрацювати в роботі

аналіз багатомодульної архітектури, аналіз етапів створення та розгортання багатомодульних застосунків, аналіз методів підвищення ефективності етапів створення та розгортання, аналіз та порівняння існуючих системи для дослідження, застосування рішень та аналіз отриманих результатів

КАЛЕНДАРНИЙ ПЛАН

Номер	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз сучасного стану питання та обґрунтування теми	16.01.2024 – 01.02.2024	*виконано*
2	Аналіз предметної галузі	02.02.2024 – 08.02.2024	*виконано*
3	Аналіз існуючих методів та алгоритмів	09.02.2024 – 01.03.2024	*виконано*
4	Розбір існуючих технологій	02.03.2024 – 15.03.2024	*виконано*
5	Вибір оптимальних рішень	15.03.2024 – 29.03.2024	*виконано*
6	Аналіз результатів досліджень та розробка системи рекомендацій	30.03.2024 – 30.04.2024	*виконано*
7	Написання та оформлення статті	01.05.2024 – 15.04.2024	*виконано*
8	Підготовка пояснювальної записки.	15.04.2024 – 27.05.2024	*виконано*
9	Підготовка презентації та доповіді	28.05.2024 – 03.06.2024	*виконано*
10	Нормоконтроль	04.06.2024 – 08.06.2024	*виконано*
11	Рецензування	09.06.2024 – 15.06.2024	*виконано*
12	Занесення диплома в електронний архів	15.06.2024	*виконано*
13	Попередній захист	15.06.2024	*виконано*
14	Допуск до захисту у зав. кафедри	18.06.2024	*виконано*

Дата видачі завдання 22 січня 2024 р.

Студент _____ Верес М.Д.
(підпис)

Керівник роботи _____ доц. Голян Н.В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка містить:138 с., 47 рис., 11 табл., 44 джерел.

БАГАТОМОДУЛЬНА АРХІТЕКТУРА, ПРОГРАМНИЙ МОДУЛЬ, РОЗРОБКА, КОМПІЛЯЦІЯ, ЗБІРКА ПРОЕКТУ, РОЗГОРТАННЯ, ЛІНІЙНА ЗГОРТКА, КОНТЕЙНЕРІЗАЦІЯ, ХМАРНА ПЛАТФОРМА, JAVA.

Об’єкт дослідження – процес створення та розгортання багатомодульних проектів та застосунків, що використовують Java Virtual Machine.

Мета роботи – дослідження існуючих проблем під час створення та розгортання багатомодульних проектів та застосунків Java та пошук засобів їх усунення.

Метод рішення – аналіз життєвого циклу багатомодульних проектів Java, покроковий аналіз кроків життєвого циклу, порівняння та розбір рішень, розробка системи рекомендацій для роботи із багатомодульними проектами Java.

Результат роботи – завершений аналіз життєвого циклу багатомодульних проектів Java, наведення рекомендацій щодо оптимізації процесів створення та розгортання багатомодульних проектів Java.

MULTIMODULE ARCHITECTURE, PROGRAM MODULE, DEVELOPMENT, COMPILATION, PROJECT BUILD, DEPLOYMENT, LINEAR CONVOLUTION, CONTAINERIZATION, CLOUD PLATFORM? JAVA.

The object of research is the process of creating and deploying multi–modular projects and applications that use Java Virtual Machine.

The purpose of the work is to investigate existing problems when creating and deploying multi-modular Java projects and finding the means of eliminating them.

The method of solution – analysis of the life cycle of multi-module Java projects step by step analysis of life cycle steps, guidance of methods to improve their effectiveness.

The result of the work is a completed analysis of the life cycle of Java multi-module projects, recommendations plan for improving the efficiency of the methods of creating and deploying Java multi-modular projects.

Заява щодо самостійного виконання кваліфікаційної роботи та можливості її публікації в електронному архіві відкритого доступу ElArKhNURE.

Я, Верес Максим Дмитрович, студент(ка) гр. ІПЗм-22-1, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження методів створення та розгортання багатомодульних Java застосунків», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений(на) з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ВСТУП

Наразі, Java займає місце однієї з найпопулярніших мов програмування в сучасному світі IT. Завдяки пов’язаним технологіям, в корпоративному світі IT відбувається розробка дуже різноманітних видів програмного забезпечення, починаючи від веб-застосунків до мобільних застосунків, від корпоративних систем до вбудованих пристроїв.

Популярність Java пояснюється рядом переваг та особливостей, серед яких є кросплатформенність, парадигма об’єктно-орієнтованого програмування, безпека та надійність, підтримка сторонніх бібліотек та її загальна поширеність із великою кількістю фреймворків, документації та постійно зростаюча спільнота розробників.

Дуже часто, проекти, що використовують Java мають такі архітектурні особливості, що містять в собі більше ніж один програмний модуль. Як абстрактний приклад можна привести схему, коли бекенд-застосунок на Java має модуль, що відповідає за комунікацію із середовищами зберігання (наприклад, база даних), модуль, що відповідає за бізнес-логіку застосунку та модуль, що відповідає за API.

Отже, й, чим складніше архітектура проекту, тим більше ризиків із зниження його ефективності під час всього життєвого циклу існує. Для Java дослідження шляхів підвищення ефективності також є доволі важливою темою. Використання більш оптимізованих підходів буде спрощувати підтримку та розвиток програмного забезпечення, покращить швидкість розгортання та поставки й, в цілому, сприяє впровадженню кращої масштабованості та зменшенню витрат бізнесу на підтримку програмного проекту, створюючи більш стабільне та надійне технічне рішення. Таким чином, кінцеві витрати бізнесу на підтримку проектів Java можуть бути меншими, в порівнянні із довільним неоптимізованим сценарієм розробки та розгортання Java-застосунків.

Проведений аналіз та система рекомендацій в подальшому можуть бути використані для покращення саме процесів розробки та створення ефективних

інструментів для заощадження витрат та ресурсів на підтримку розробки та розгортання багатомодульних Java-застосунків.

Також система рекомендацій може бути використана при створені нових багатомодульних застосунків на Java або сумісних із JVM мовах, так і при оптимізації вже існуючих подібних програмних проектів. Більшість з розглянутих раніше покращень можуть бути застосовані не тільки для багатомодульних застосунків, а й для тих, що мають відмінну архітектуру або ж особливості, проте мають однакові передумови до аналізу забезпечення шляхів оптимізації.

Варто зазначити, що рекомендації є не суворими, та дуже залежать від контексту задач та середовища, в якому йде розробка програмного продукту. Кожна з рекомендацій може бути використана як в групі відповідно до плану, або ж розглянута окремо, що робить ці рекомендації доволі гнучкими не зважаючи на структурований план щодо їх впровадження до програмного проекту.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз предметної галузі

Розробка багатомодульних застосунків на Java передбачає створення програмного забезпечення, яке складається з кількох взаємодіючих між собою модулів або компонентів. Розглянемо загальний опис процесу розробки багатомодульних застосунків із використанням Java.

Аналіз вимог – починаючи з аналізу бізнес-вимог, розробники спілкуються з клієнтом або зацікавленими сторонами, щоб зрозуміти, які функції та властивості повинен мати застосунок. На цьому етапі також визначається, які модулі будуть потрібні та як вони будуть взаємодіяти між собою.

Проектування архітектури – розробники розробляють архітектурний план застосунку, визначаючи структуру модулів, їх функції та взаємозв'язки. Часто використовуються патерни проектування для підтримки високої рівної абстракції та забезпечення модульності.

Розробка модулів – кожен модуль розробляється окремо, використовуючи підходи ООП. Класи та інтерфейси реалізують функціональність, а взаємодія між модулями забезпечується інтерфейсами та API.

Тестування – після розробки модулів вони піддаються тестуванню. Виконуються різні типи тестів, включаючи одиничні, інтеграційні та системні тести, щоб переконатися у їх коректності та взаємодії.

Інтеграція модулів – після успішного завершення тестування модулі інтегруються разом в єдиний застосунок. Це може включати інтеграційне тестування, щоб переконатися, що модулі правильно співпрацюють між собою.

Розгортання та супровід – застосунок готовий для розгортання на виробничому середовищі. Після випуску виробництво розробники можуть продовжувати підтримку та вдосконалення застосунку, виправляючи помилки та вдосконалюючи функціональність.

Цей процес може бути деталізований та адаптований залежно від конкретних потреб та особливостей проекту. Використання ефективних інструментів для

керування версіями, автоматизації тестування та розгортання може значно полегшити та прискорити розробку багатомодульних застосунків на Java.

1.2 Загальний опис багатомодульної архітектури

Відповідно до опису кращих практик багатомодульної архітектури від Google [1], багатомодульна архітектура (див. рис. 1) має місце використання в ситуаціях, коли на кодова база проекту активно масштабується, а легкість сприйняття та загальна якість коду с часом знижуються. Також, в такому випадку, не завжди наявний достатній рівень підтримки структури коду, для спрощення її подальшої підтримки. Щоб вирішити дану проблему, можна застосувати метод модулярізації проекту.

Модулярізація – засіб структурування кодової бази проекту із метою підвищення зручності підтримки та запобігання подальших проблем із кодовою базою під час її масштабування.

Суть модулярізації полягає в організації кодової бази проекту на слабопов'язані та автономні частини, кожна з яких являє собою окремий **модуль**. Кожен з таких модулів є незалежним та створений для чіткої цілі.

Таким чином, розділення проблеми на дрібніші та простіші для рішення підзадачі, ми отримуємо знижену складність проектування та обслуговування для великої системи (див. рис. 1.1).

На рисунку позначено декілька шарів роботи програмного проекту. Перший являє собою шар представлення даних що складається із двох модулів, наступним йде шар надання бізнес-функціоналу або ж бізнес-логіки застосунку що представляє собою один єдиний модуль. Далі розглядається шар роботи із даними проекту який має два модулі та останнім в проекті є кінцевий модуль комунікації із мережею.

Отже, можна побачити, що кожен зображений модуль є окремою автономною одиницею, що виконує окрему власну функцію та має залежності на інші наведені модулі. Завдяки цьому кожна одиниця може розглядатися окремо в контексті питання про масштабованість проекту.

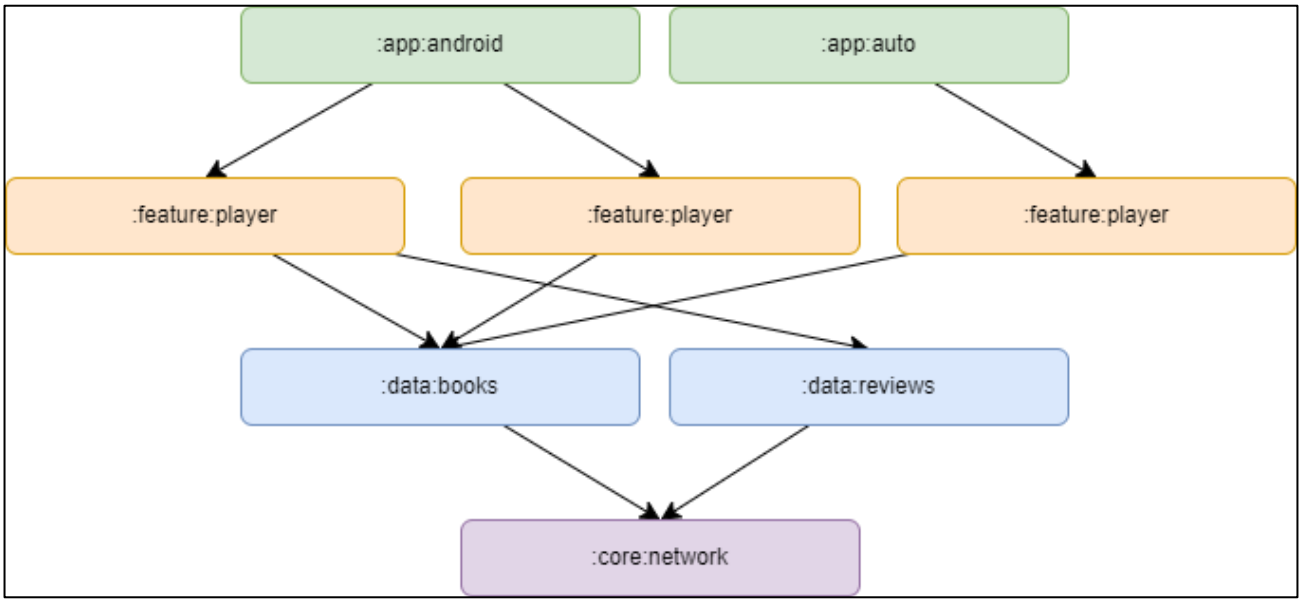

Рисунок 1.1 – Приклад графу залежностей багатомодульної кодової бази (за даними [1])

Підсумуємо наведені дані щодо багатомодульної архітектури, та наведемо список її переваг та недоліків використовуючи аналіз від Genesis [2].

Переваги модульної архітектури:

– краща організація коду. Це дозволяє команді масштабуватися без зайвих проблем;
– легке управління та підтримка. Невеликі функціональні команди зможуть розробляти певні модулі та відповідати за певну область застосунку. Це допоможе пришвидшити розробку;
– зручність тестування. Кожний модуль можна тестувати незалежно, що полегшує виявлення та виправлення проблем. Це призводить до кращого тестового покриття та надійності коду;
– скорочення часу збірки. З правильним підходом до модульної архітектури можна оптимізувати час холодної та гарячої збірки.

Недоліки модульної архітектури:

– ретельне планування та координація. Варто завчасно продумати та перепроєктувати систему, щоби мати прогнозований результат її використання і не стикатися з великою кількістю проблем;

– ризики конфліктів та несумісності в управлінні залежностями.

1.3 Загальний опис мови програмування Java та її технологій

Для більш детального занурення у контекст дослідження, наведемо загальний опис мови програмування Java, її особливості, місце застосування, та технології з якими дуже часто поєднується процес розробки із використанням даної мови.

Java є високорівневою, об’єктно-орієнтованою мовою програмування, розробкою якої займалась компанія Sun Microsystems (що надалі була придбана корпорацією Oracle). Перший публічний реліз мови з’явився в 1995-му році. Розробкою керували технічні спеціалісти Джеймс Гослінг, Майкл Шерідан та Патрік Нафталін. Спочатку, Java створювалась як частина проекту Green Project, який був спрямований на створення мови програмування, що могла бути застосована в розробці програмного забезпечення для побутової електроніки. Саме тому під час створення мови, було заложено принцип кросплатформенності. Однак, наявність подібної крос-платформенної природи та інших переваг дозволило вийти Java за межи використання лише в сфері програмування побутової електроніки. Можна сказати навіть більше, наразі Java має велике застосування в корпоративній розробці, майже повністю залишаючи сферу програмування побутової електроніки для інших мов програмування, таких як C++. Отже, можна виділити наступні ключові особливості мови Java.

1.3.1 Об’єктно-орієнтована мова програмування

Java створена, використовуючи концепцію об’єктно-орієнтованого програмування (ООП), таким чином наявна можливість створення програм із використанням взаємодіючих між собою об’єктів.

1.3.2 Кросплатформенність або платформонезалежність

Розглянутий вище один з головних принципів Java, що робить можливим виконувати програми на будь-якому пристрої, на якому встановлено відповідну віртуальну машину Java (JVM). Реалізований даний підхід завдяки тому, що Java-

програми компілюються в байткод, який потім буде виконаний на різних платформах JVM.

1.3.3 Безпека

Java реалізована із різними вбудованими механізмами безпеки, такими як система контролю доступу до об’єктів, механізми перевірки типів, що дозволяють уникнути різноманітні типи програмних помилок, наприклад, таких як переповнення буфера або різні проблеми роботи з пам’яттю.

1.3.4 Автоматичне управління пам’яттю

Java має автоматичну збірку сміття (так званий процес garbage collection), що звільняє розробника від необхідності в ручному режимі керувати пам’яттю програми, зменшуючи ризик витоку пам’яті та інших проблем, значно облегшуючи процес розробки.

1.3.5 Багатопоточність

Java підтримує створення та управління відразу багатьма потоками використовуючи один процес. Що дозволяє використовувати підхід багатопоточного програмування.

1.3.6 Розширені стандартні бібліотеки та підтримка сторонніх бібліотек

Java постачається із широким спектром стандартних бібліотек, які мають різноманітні інструменти для роботи з мережами, базами даних, графікою та веб-розробки.

Структура Java-програми включає в себе класи та інтерфейси. Класи використовуються для визначення об’єктів, їх властивостей та методів. Інтерфейси визначають функціональність, яка може бути реалізована класом. Кожна програма Java складається принаймні з одного публічного класу, який містить метод, що є вхідною точкою програми.

Загальна архітектура Java-програм включає в себе код, який компілюється у байткод Java, а потім виконується на віртуальній машині Java (JVM). Вона також включає в себе стандартні бібліотеки Java (Java Standard Edition), що містять класи та інтерфейси для різноманітних завдань програмування.

1.4 Загальний опис забезпечення багатомодульної архітектури в Java

Розглянемо загально основні пункти Java Platform Module System (JPMS), використовуючи дані з ресурсу Baeldung[3].

Починаючи з Java 9, було введено новий рівень абстракції над пакетами, відомий як Система модулів платформи Java (Java Platform Module System).

Модуль в JPMS – це група тісно пов’язаних пакетів та ресурсів разом з новим файлом дескриптора модуля.

Пакети всередині модуля є ідентичними до вже існуючих пакетів в Java.

Кожен модуль відповідає за свої ресурси, наприклад такі як медіа або файли конфігурації.

На даний момент існує чотири типи модулів:

– системні модулі – це модулі, перелічені під час запуску команди list-modules. Вони включають модулі Java SE та JDK;
– модулі застосунку – це модулі, що зазвичай створюються, коли є необхідність використання модульності. Вони названі та визначені у скомпільованому файлі Module-info.class, включеному до зібраного JAR-файлу;
– автоматичні модулі – неофіційні модулі, що можуть бути включені до проекту, додаючи існуючі JAR-файли в шлях до модуля. Ім'я модуля буде отримано від імені JAR. Автоматичні модулі будуть мати повний доступ на читання до всіх інших модулів, завантажених по шляху;
– безіменний модуль — коли клас або JAR завантажується в шлях до класів, але не в шлях до модуля, він автоматично додається до безіменного модуля. Це універсальний модуль для забезпечення зворотної сумісності із раніше написаним кодом Java.

Модулі можна розповсюджувати одним із двох способів: у вигляді JAR-файлу або у вигляді «розібраного» скомпільованого проекту.

Також є можливість створення багатомодульні проекти, що складаються з «основної програми» та кількох бібліотечних модулів. Однак, під час створення модулів необхідно уважно слідкувати, щоб в кожному JAR-файлі був тільки один модуль. Коли йде налаштування файлу збірки, необхідно обов'язково об'єднати кожен модуль в проекті в окремий jar-файл.

Щоб налаштувати модуль, необхідно помістити в корінь пакетів спеціальний файл на ім'я Module-info.java.

Цей файл відомий як дескриптор модуля та містить усі дані, необхідні для створення та використання нового модуля.

Приклад дескриптору модуля з статті Baeldung зображений на рисунку 1.2.

```
module hello.modules {
    exports com.baeldung.modules.hello;
}
```

Рисунок 1.2 – Приклад дескриптора модуля JPMS (за даними [3])

Існують різні директиви, та приклади їх використання, що може містити модуль. Основні з них, що використовуються в JPMS:

– «requires» – ця директива модуля дозволяє оголошувати залежності модуля;
– «requires static» – використовуючи статичну директиву require, створюється залежність, доступна лише під час компіляції;
– «requires transitive» – використовуються, щоб змусити всіх наступних споживачів також прочитати необхідні залежності;
– «exports» – директива експорту використовується, щоб надати доступ всім загальнодоступним членам іменованого пакета;
– «export … to» – використовується для того, щоб відкрити публічні класи або обмежити, які модулі мають доступ до API;

- «uses» – використовується для позначення сервісів, які використовує модуль. Ім'я класу при використанні директиви є або інтерфейсом, або абстрактним класом сервісу, а не класом реалізації;
- «provides … with» – директива використовується, коли модуль також може бути постачальником послуг, що можуть бути спожитими іншими модулями;
- «open» – оскільки Java 9 забезпечує строгу інкапсуляцію, тепер є потрібним явно надавати дозвіл іншим модулям на відображення класів. Якщо необхідно й надалі забезпечувати повне відображення, як це було в старих версіях Java, є можливість просто «відкрити» весь модуль;
- «opens» – використовується якщо є потрібність дозволити відображення приватних типів, але не потрібно, щоб весь код був розкритий, таким чином надаючи доступ до певних пакетів;
- «opens…to» – директиву можна використовувати, щоб вибірково відкривати пакети для попередньо затвердженого списку модулів.

Отже, було розглянуто Java Platform Module System. Наразі рівень популярності використання даної технології на низькому рівні через відсутність достатньої підтримки сторонніми інструментами та фреймворками.

Одним із найбільш використаних рішень щодо інтеграції та роботи з залежностями модулів є використання систем збірки проекту.

1.5 Аналіз літературних джерел

Повертаючись до наведених в попередніх пунктах особливостях Java, можна відмітити її поширеність, в тому числі й досить велику спільноту розробників. Також, повертаючись до основних етапів створення та розгортання багатомодульних застосунків на Java, першим є етап написання та компіляції коду. Кращі практики, які можуть бути використані на даному етапі описані в дуже великій кількості літературних джерел. Одним з найбільш відомих та перевірених часом є «Effective Java» Джошуа Блоха [4]. Основні поради та кращі практики, що можна віднести для даного дослідження, що були наведені в джерелі:

- замість публічних конструкторів рекомендується використовувати статичні фабричні методи, що можуть мати іменовані параметри та повертати підтипи класу;
- рекомендується використовувати в першу чергу інтерфейси, а не реалізації для забезпечення гнучкості та змінності коду. Тобто уникати витоку деталей реалізації;
- правильне перевизначення методів «equals()», «hashCode()» та «toString()» дозволяє більш правильно працювати з колекціями, перевіряти об’єкти на рівність та надавати коректні рядкові представлення об’єктів;
- використання перечислень (enum) замість константних статичних об’єктів дозволяє спростити читабельність та підтримку коду;
- не варто використовувати ключове слово «synchronized» для всього методу в випадках, коли в цьому немає необхідності. Замість цього варто використовувати синхронізовані блоки для зменшення накладних витрат;
- використовувайте Java generics для створення більш безпечного та читабельного коду, який не потребує явного приведення одного типу до іншого.

Іншим корисним, але вже не дуже новим джерелом, що охоплює методи та практики з автоматизації процесу розгортання програмного забезпечення «Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation» авторів Девіда Фарлі та Джез Хамбл [5]. Основні кращі практики та поради, що описані авторами:

- зосередження на автоматизації всього процесу від збирання програмного коду до тестування та розгортання допомагає знизити час доставки та покращити якість програмного забезпечення.
- Continuous Integration (CI) та Continuous Deployment (CD), так само, як й було розглянуто вище, дозволяють розробникам автоматизувати процеси збірки, тестування та розгортання, що дозволяє пришвидшити та зробити безпечнішим впровадження змін в продакшн;

- використання інфраструктури як коду (Infrastructure as Code, IaC), визначення інфраструктури (наприклад, серверів, баз даних тощо) як коду дозволяє автоматизувати процес розгортання та забезпечує цілісність середовищ розгортання;
- розділення застосунку на невеликі, незалежні сервіси сприяє пришвидшенню розгортання та вдосконалює частини застосунку, роблячи більш легким процес масштабування;
- зосередження на побудові широкого спектру автоматизованих тестів сприяє більш швидкому виявленню помилок та знижує ризик пов'язані з якістю програмного забезпечення перед розгортанням;
- підтримка надійності та швидкості розгортання програмного забезпечення досягається завдяки активному моніторингу та збору метрик у реальному часі.

Третім й останнім найбільш приближеним до теми дослідження літературним джерелом є книга «Building Microservices» Сема Ньюмена [6]. Книга є цінним ресурсом для розробників, які цікавляться створенням рішень із використанням мікросервісних архітектур. Вона надає докладний огляд різноманітних методів, кращих практик та підходів, для побудови мікросервісів, включаючи використання Java. Основними порадами та практиками є:

- розгляд базових концепцій та принципів мікросервісної архітектури, визначає переваги та недоліки порівняно з традиційними монолітними архітектурами;
- автором розглянуто питання, що пов'язані з проектуванням мікросервісів, наприклад, таких як визначення меж сервісів, принципи організації комунікації між ними, та методи розробки API;
- в книзі наведено приклади реалізації мікросервісів за допомогою різноманітних технологій та фреймворків Java, наприклад, таких як Spring Boot, Dropwizard та інших;

– автором описано стратегії моніторингу та логування мікросервісів, включаючи використання інструментів та підходів щодо забезпечення надійності та відновлення сервісів;
– також книгою висвітлено питання, що пов’язані з тестуванням мікросервісів, включаючи автоматизовані тести, тестування взаємодії між сервісами та розгляд стратегій розгортання.

Отже, було проаналізовано додаткові джерела, що містять інформацію, яка може бути використана для подальшого дослідження методів, практик та інструментів для підвищення ефективності створення та розгортання багатомодульних застосунків на Java. Виділено найбільш важливі тези, інформація з яких буде використана у подальшому дослідженні.

1.6 Постановка задачі

Результатом аналізу предметної області став відомий основний напрям дослідження та його задача. Вона полягає в розгляданні кожного з етапів створення та розгортання багатомодульних Java-застосунків, дослідженні особливостей цих етапів, аналізу їх потенційних слабких місць та наведенні кращих практик, що можуть бути використані для створення узагальненої методології для підвищення ефективності розглянутих процесів.

Випливаючи з попередніх відомостей, можемо сформувати наступну схему дослідження:

– загальний огляд процесу розробки та побудови збірок багатомодульних проектів на Java використовуючи системи збірки;
– виділення етапів створення та розгортання багатомодульних застосунків на Java;
– аналіз кожного з наведених етапів, пошук його слабких місць та дослідження методів підвищення ефективності, існуючих інструментів тощо;
– аналіз характеристик та вибір оптимальних методів та інструментів;

– розробка кінцевої системи рекомендацій для підвищення ефективності створення та розгортання багатомодульних застосунків на Java;
– наведення висновків щодо роботи.

Як одним з головних методів пошуку оптимальних рішень буде використано лінійну адитивну згортку, для аналізу ефективності характеристик розглянутих інструментів та рішень та пошуку кращого з них відповідно до розглянутих вимог.

Дані для аналізу характеристик буде використано відкриті джерела (в випадку з розглянутими інструментами – сайти з офіційною документацією актуальною відповідно до дати звернення), а також дані, що можуть бути отримані під час практичного дослідження.

2 АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ І АЛГОРИТМІВ

2.1 Короткий опис найбільш популярних систем збірки проекту для Java-проектів.

Використовуючи матеріал Baeldung[7] із порівняння найбільш відомих систем збірки для використання в Java проектах, наведемо короткий опис кожної із них.

2.1.1 Apache Ant

Apache Ant («Another Neat Tool») — це бібліотека Java, яка використовується для автоматизації процесів створення програм Java. Крім того, Ant може бути використаний для написання деяких програм, що використовують технології відмінні від Java.

У багатьох аспектах Ant дуже схожий на Make[8], сама по собі бібліотека є досить простою, тому її використання може бути без будь-яких особливих умов.

Файли збірки Ant написані у форматі XML, і за домовленістю вони називаються build.xml.

Різні фази процесу створення називаються «цілями» («goals»).

2.1.2 Apache Maven

Apache Maven — це інструмент керування залежностями та автоматизації збірки, який переважно використовується для додатків Java. Maven продовжує використовувати XML-файли так само, як Ant, але набагато більш керованим способом, використовуючи принцип конвенція над конфігурацією (convention over configuration) [9].

У той час як Ant забезпечує гнучкість і вимагає, щоб усе було написано з нуля, Maven покладається на конвенції та надає попередньо визначені команди (цілі).

Конфігураційний файл Maven, що містить інструкції зі створення та керування залежностями, за домовленістю називається pom.xml. Крім того, Maven також передбачає сувору структуру проекту, тоді як Ant також забезпечує гнучкість.

2.1.3 Gradle

Gradle — це інструмент для керування залежностями та автоматизації збірки, створений на основі концепцій Ant і Maven.

Одне з перших речей, які ми можемо відзначити про Gradle, це те, що він не використовує файли XML, на відміну від Ant або Maven.

Gradle використовує DSL на основі Groovy або Kotlin. Конфігураційний файл Gradle за домовленістю називається build.gradle у Groovy або build.gradle.kts у Kotlin.

2.2 Розгортання Java-застосунків

Завдяки підтримки принципу кросплатформенності, Java та інші застосунки, що використовують JVM мають настільки широкі можливості щодо розгортання, що їх просто неможливо перечислити на даний момент.

Узагальнено схему розгортання (див. рис. 3), можна відобразити наступним чином.

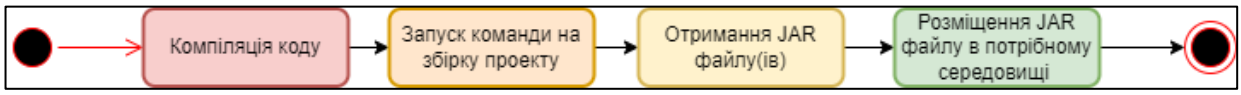

Рисунок 2.1 – Узагальнена схема із розгортання Java-застосунку (створено самостійно)

Є зрозумілим, що дана схема є доволі узагальненою, проте в певній мірі відокремлює окремі кроки із переведення Java-коду в повноцінно робочу програмну систему, що розміщена на необхідному середовищі.

Таким чином виділимо наступні кроки даного процесу:

– компіляція коду – процес перетворення класів написаних на JVM-сумісній мові у Java bytecode[10];
– запуск команди на збірку проекту – процес використання певної системи для збірки у певному середовищі. Наприклад, збірка проекту на локальному комп’ютері використовуючи Maven;

– отримання JAR-файлу(ів) – отримання результату збірки (виконавчі файли для модуля/модулів) та даних, щодо збірки проекту із можливим подальшим завантаженням JAR-файлу у певний репозиторій версій програмного продукту;
– розміщення JAR-файлу в потрібному середовищі – процес розгортання, можливого налаштування та запуску застосунку у певному кінцевому середовищі. Наприклад, розміщення JAR-файлу, його файлів конфігурації та його запуск на сервері у хмарі.

Ці кроки дещо перекликаються із процесом CI/CD[11], проте є більш узагальненими для розглянутої задачі.

Проблема роботи із багатомодульними застосунками полягає в тому, що на кожному із цих кроків дуже часто присутні можливі втрати ефективності (часові, зайве використання пам’яті та ін.). Отже, відповідно до цілей роботи, під час дослідження буде розглянуто можливі шляхи оптимізації кожного із наведених кроків для багатомодульних застосунків на Java, що може призвести до підвищення ефективності процесу створення та розгортання, аналіз наведених методів та розрахунки щодо мір підвищення ефективності.

2.3 Аналіз методів підвищення ефективності компіляції коду багатомодульного проекту

Процес компіляції коду в Java – це процес перетворення вихідного коду, написаного на мові програмування, сумісній із віртуальною машиною Java (наприклад, сама мова Java, мова Groovy, Kotlin та ін.) в байт-код, що може бути виконаний на цій віртуальній машині.

Виконання даного процесу забезпечується за допомогою інструменту, що називається компілятор Java.

Основні кроки компіляції включають в себе наступне:

– написання вихідного коду – створення вихідного коду на сумісній із JVM мові, наприклад Java. В такому випадку це може бути файл з розширенням .java;

– компіляція – безпосередньо процес перетворення компілятором вихідного коду, що має вигляд файлу із розширенням .class, що містить байт-код. Байт-код є проміжним кодом, що не є носієм для конкретної архітектури, замість цього використаний віртуальною машиною Java для виконання програми;
– виконання на віртуальній машині Java (JVM) – процес запуску згенерованого байт-коду на будь-якій віртуальній машині Java (JVM), однією із задач якої є його інтерпретація або компіляція в машинний код для конкретної платформи в режимі реального часу.

Основна перевага використання підходу із байт-кодом полягає в тому, що програми можуть бути виконані на будь-якому пристрої, що має відповідну реалізацію віртуальної машини Java. Таким чином забезпечується один із головних принципів Java – кросплатформенність, яка дозволяє переносити код між різними платформами без перекомпіляції вихідного коду.

Для підвищення ефективності компіляції коду в Java в першу чергу необхідно виділити основні характеристики процесу, що можуть бути розглянуті в розрізі даного питання.

Таблиця 2.1 – Характеристики для підвищення ефективності кроку компіляції коду (виконана самостійно)

Назва	Міра	Опис
Час компіляції	Секунда	Час, що витрачається на процес компіляції
Використання пам'яті	Байт	Обсяг використаної пам’яті під час компіляції
Якість Java-коду	Якісна характеристика; Залежить від системи аналізу	Кількість попереджень та потенційних помилок в коді Java

Розглянемо можливі методи підвищення ефективності наведених характеристик.

2.3.1 Час компіляції та використання пам’яті

Прискорення часу компіляції великої міри залежить від конкретності проекту та використовуваних інструментів.

Основним із методів прискорення компіляції є інкрементна компіляція коду. Є відомим, що не всі компілятори підтримують дану функціональність.

Відповідно до статі Deepanshu Rustagi на сайті GeeksForGeeks[12], можемо навести наступні особливості та переваги з недоліками.

Інкрементальний компілятор — це тип компілятора, який повторно компілює лише ті частини програми, які змінилися після останньої компіляції. Такий підхід може значно заощадити час і ресурси, особливо для великих програм, які потребують тривалого часу компіляції.

Особливості інкрементного компілятора:

- під час процесу розробки програми модифікації вихідної програми можуть призвести до перекомпіляції всього вихідного тексту. Ці накладні витрати зменшуються в інкрементальному компіляторі;
- помилки під час виконання можна виправити так само, як і помилки під час компіляції, зберігаючи модифікацію програми як є;
- процес компіляції відбувається швидше;
- робота з пакетними програмами стає дуже гнучкою за допомогою інкрементного компілятора;
- у інкрементному компіляторі таблиця структури програми підтримується для розподілу пам’яті цільового коду. Коли нові зміни компілюються, відповідно до них створюється новий запис в таблиці структури програми. Таким чином, виділення пам'яті, необхідне для інкрементного компілятора, не повинно бути безперервним;
- допомагає відстежувати залежності від вихідної програми.

Маємо наступні переваги використання інкрементного компілятора:

– швидший час компіляції: інкрементні компілятори можуть значно заощадити час, перекомпілюючи лише ті частини програми, які змінилися, а не всю програму;
– покращена швидкість розробки: інкрементні компілятори можуть пришвидшити процес розробки, дозволяючи розробникам швидше побачити наслідки внесених змін;
– більш ефективне використання ресурсів: перекомпілюючи лише змінені частини програми, інкрементальні компілятори можуть використовувати менше ресурсів, таких як цикли процесора та пам’ять;
– краще звітування про помилки: інкрементні компілятори можуть надавати більш детальні звіти про помилки та інформацію про налагодження, ніж традиційні компілятори, оскільки вони можуть більш ретельно відстежувати зміни програми.

А також маємо наступні недоліки використання інкрементного компілятора:

– підвищена складність: інкрементні компілятори, як правило, складніші для реалізації, ніж традиційні компілятори, оскільки вони вимагають більш складного відстеження та керування змінами програми;
– збільшене використання пам’яті: залежно від реалізації інкременним компіляторам може знадобитися більше пам’яті для зберігання інформації про зміни та залежності програми;
– ризик помилок: інкрементні компілятори можуть створити ризик помилок, якщо зміни в програмі не відстежуються належним чином або залежності не вирішуються належним чином;
– обмеження певної платформи: деякі мови програмування або платформи можуть не підтримувати інкрементну компіляцію або можуть мати обмеження, які роблять її менш ефективною чи ефективною.

Загалом інкрементні компілятори можуть бути корисним інструментом для підвищення ефективності та швидкості процесу компіляції, але вони вимагають ретельного впровадження та керування.

Даний підхід може бути корисним при відсутності використання систем збірки проекту, однак більшість сучасних проектів використовує ту чи іншу систему збірки. Також далеко не кожен компілятор Java підтримує інкрементність, одним із тих, що надають такий функціонал є компілятор Eclipse Compiler for Java (ECJ)[13].

Аналіз та приклади використання інкрементної компіляції буде наведено надалі в розділі аналізу методів підвищення ефективності збірки.

Наступним методом, що може прискорити час компіляції є оптимізація параметрів компілятора.

Для підвищення ефективності швидкості компіляції можуть бути налаштовані параметри компілятора для оптимізації процесу. Наприклад, використання параметрів, які розпаралелюють процес компіляції або збільшують обсяг доступної пам'яті, може покращити продуктивність.

```
javac -J-Xmx4g -J-Xms1g -proc:only MyClass.java
```

Рисунок 2.2 – Приклад використання параметрів компілятора (створено самостійно)

В цілому, процес прискорення та зменшення споживання пам’яті компілятором є дуже специфічним до кожної його імплементації, версії та платформи на якій він використовується. Результати можуть бути різними також відповідно до апаратних можливостей середовища, де виконується компіляція коду. Наприклад, апаратні можливості можуть не підтримувати багатопоточність та мати дуже малий обсяг оперативної пам’яті, що буде перекреслювати навіть використання інкрементого компілятора із розпаралеленням компіляції. Також багато із наведеної вище роботи з оптимізації на даний час роблять системи збірки проекту, які будуть проаналізовані надалі.

Таким чином, можна сказати, що оптимізація даних характеристик може бути виконана під час використання та налаштування відповідної системи збірки в проекті, тобто для реальних проектів в більшості випадків немає сенсу робити оптимізації на такому доволі низькому рівні.

2.3.2 Якість Java-коду

Якість Java-коду, може бути характеризована по-різному. В даному розділі, щодо підвищення ефективності компіляції, зазначимо, її як складену характеристику, що поєднує в собі модульність, міра вразливості коду, дотримання Java Code Conventions[14] та забезпечення кращих практик до архітектури програмного забезпечення.

Можна виділити наступні аспекти якості коду, що можуть впливати на ефективність компіляції.

2.3.2.1 Розмір файлів та проекту

Великі файли та проекти можуть призвести до збільшення часу компіляції, оскільки компілятор повинен обробляти більше даних. Модульна структура та поділ коду на невеликі файли можуть полегшити процес компіляції.

Проведемо експеримент щодо часу компіляції коду. Для експерименту використано середовище із 32 Гб оперативної пам’яті та процесором Intel Core I9-9900K із частотою 3.0 Ггц та 8 фізичними ядрами (16 логічними ядрами), Windows 10. Візьмемо простішу програму, що виведе «Hello world!» у консоль.

Код програми:

```
public class Main {

    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Отримано час збірки (включаючи час компіляції) внутрішньою системою збірки IDE IntelliJ IDEA – 913 мс.

Наступним кроком візьмемо трохи складніший код, що буде містити в собі програму, що в циклі записує в файл числа із позначенням чи число парне або непарне із повторенням до десяти тисяч.

Код програми:

```
import java.io.FileWriter;
import java.io.IOException;

public class Main {

    private static final int LIMIT = 10000;
    private static final String FILE_NAME = "numbers.txt";

    public static void main(String[] args) {
        try (final var fileWriter = new FileWriter(FILE_NAME)) {
            for (var i = 0; i < LIMIT; ++i) {
                if (i % 2 == 0) {
                    fileWriter.write("Even number: " + i);
                } else {
                    fileWriter.write("Odd number: " + i);
                }
                fileWriter.write("\n");
            }
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Під час збірки тією ж самою системою отримуємо час – 1 секунда 27 мілісекунд.

Порівнюючи результати, можемо побачити різницю у сповільненні часу збірки на 12,5% після збільшення вихідного коду для компіляції.

Отже, чим менше вихідний код, тим швидший час на його компіляцію та збірку. Варто відмітити про те, що відповідно до імплементації компілятора (наприклад, JIT[15]), можуть бути автоматично задіяні певні оптимізації коду під час компіляції, що може дещо покращити її часові характеристики та в цілому ефективність байт-коду.

2.3.2.2 Структура коду

Чітка та логічна структура коду полегшує розуміння коду як для розробників, так і компілятора. Оптимізований та ефективний код може зменшити час компіляції, оскільки менше обчислень та операцій повинні бути виконані під час компіляції.

Проте можна згадати про попередній пункт. Як правило компілятори Java автоматично проводять оптимізації, якщо це можливо, що може дещо покращити

якість байт-коду, навіть, якщо вихідний код Java мав певного роду проблеми із структурою.

2.3.2.3 Використання анотацій та мета-програмування

Використання аннотацій та мета-програмування може призвести до генерації додаткового коду. Це може впливати на час компіляції. Використання аннотацій для компілятора (наприклад, @SuppressWarnings[16], @Generated[17], тощо) може допомогти зменшити кількість помилок та попереджень під час компіляції.

2.3.3 Інструменти підтримки якості Java-коду

Розглянемо, інструменти, які можуть бути використані для підтримки якості вихідного Java-коду. Отже, як було проаналізовано, оптимізація вихідного Java-коду підвищує ефективність процесу компіляції.

2.3.3.1 Checkstyle

Checkstyle [18] – це інструмент для перевірки коду на відповідність стандартам оформлення (code style). Дозволяє налаштовувати правила форматування коду. Існує у вигляді плагіну для IDE (IntelliJ IDEA, Eclipse та ін.).

2.3.3.2 FindBugs / SpotBugs

FindBugs / SportBugs [19] – це аналізатор для виявлення потенційних помилок та проблем в процесі виконання коду. Існує у вигляді плагіну для IDE (IntelliJ IDEA, Eclipse та ін.).

2.3.3.3 PMD

PMD [20] – це аналізатор коду, що використовує статичний аналіз для виявлення потенційних помилок, дублювання коду та інших антипатернів. Існує у вигляді плагіну для IDE (IntelliJ IDEA, Eclipse та ін.).

2.3.3.4 SonarQube

SonarQube [21] – це платформа для управління якістю коду, яка об'єднує різні аналізатори коду і надає докладні звіти про якість коду. Може бути використаний як плагін для IDE (SonarLint [22] – офіційний плагін від розробників SonarQube).

2.3.3.5. JUnit / TestNG

JUnit [23] / TestNG [24] – фреймворки для написання та виконання модульних тестів. Вони допомагають забезпечити якість коду через автоматизоване тестування.

2.3.3.6. JaCoCo

JaCoCo [25] – це інструмент для вимірювання покриття коду тестами. Вказує, які саме частини коду були викликані під час виконання тестів та процент їх покриття.

2.3.3.7. Error Prone

Error Prone [26] – це плагін компілятора для виявлення потенційних помилок в коді на етапі компіляції.

В цілому, наведені інструменти можуть бути інтегровані та включені до процесу збірки проекту або використовувати їх як частину інфраструктури розробки. Їх суть полягає в забезпеченні підвищення якості коду, як складеної наведеної вище характеристики, полегшення виявлення та виправлення помилок, а також сприяють дотриманню стандартів програмування, що в свою чергу підвищує ефективність компіляції та збірки проекту.

Результатом аналізу методів підвищення ефективності компіляції коду багатомодульного проекту в пункті 2.1 стало отримання методів, що дозволюють прискорити процес компіляції проекту та оптимізувати споживання пам'яті.

Найпростішою характеристикою для проведення оптимізацій процесу компіляції на високому рівні є підвищення якості коду, до якої було наведено засоби її оцінки та підтримки.

Методи для оптимізації на більш низкому рівні більш обмежені та є специфічними відповідно до середовища компіляції вихідного коду та апаратних можливостей платформи. Однак, також було наведено загальні рекомендації щодо використання особих типів компілятора (інкременті компілятори) та оптимізації його параметрів під час компіляції.

Наведені вище методи підвищення ефективності компіляції також використовуються в подальших пунктах, де буде наведено аналіз та розрахунки для методів підвищення ефективності збірки багатомодульного проекту.

2.4 Аналіз процесу CI/CD в Java

Для початку, наведемо основні поняття, пов’язані із процесом неперервної інтеграції та неперервної доставки.

Відповідно до статі Cloudfresh «10 причин, чому CI/CD важливі для DevOps» [27], CI/CD, що означає неперервна інтеграція та неперервна доставка (або неперервне розгортання), – це практика розробки програмного забезпечення, яка включає такі ключові елементи:

- неперервна інтеграція: часте об’єднання змін коду у спільний репозиторій. Після кожної інтеграції автоматично запускаються тести для забезпечення якості коду та виявлення можливих проблем;
- неперервна доставка: на основі неперервної інтеграції автоматизує процес підготовки коду до розгортання, щоб код завжди був готовий до випуску;
- неперервне розгортання: робить ще один крок, автоматично розгортаючи зміни коду у продакшн після успішного проходження тестів та перевірок якості.

Ці практики дозволяють командам розробників швидше, надійніше та з меншою кількістю помилок доставляти програмне забезпечення. Вони сприяють

оптимізованому робочому процесу, що покращує якість коду та прискорює випуск оновлень.

Різниця між CI та CD полягає в тому, що в процесі неперервної інтеграції (CI), розробники часто об’єднують зміни коду в спільний репозиторій протягом певного проміжку робочого часу (наприклад, робочого дня). Кожна така зміна викликає процес автоматичної збірки та тестування із метою швидкого виявлення проблем.

Під час процесу неперервної доставки (CD) зосередження йде на тому, щоб програмне забезпечення було готове до релізу, мінімізуючи ручну роботу завдяки застосування та випуску збірок проекту, що пройшли тестування.

Неперервне розгортання йде далі, автоматично випускаючи зміни у продакшн після успішного проходження тестів. Це дозволяє надзвичайно швидко й часто випускати оновлення та робити виправлення. Отже, CI часто інтегрує та тестує зміни для швидкого зворотного зв’язку. CD використовує автоматизацію, щоб зв’язати CI та продакшн. Безперервне розгортання автоматично випускає перевірені зміни в продакшн для прискорення доставки.

Розглянемо ключові переваги та роль інструментів CI/CD та чому це є доволі важливою частиною кожного проекту.

В першу чергу, завдяки інструментам CI/CD йде автоматизація та зв’язка всього процесу доставки програмного забезпечення (а отже, й процесу збірки та розгортання). Ефективно налаштовані інструменти збирають код, автоматично запускають тести та розгортають програмне забезпечення у середовища, використовуючи уніфікований підхід. Завдяки чому забезпечується швидкий зворотний зв’язок, зменшується ризик створення помилок та кінцева протестована версія програмного забезпечення швидше надходить до кінцевого споживача.

Таким чином ключовими перевагами неперервної інтеграції та розробки є:

– автоматизація збірки – усунення ручної роботи та пришвидшення циклів випуску програмного забезпечення;
– автоматизація тестування – виявлення дефектів розроблюваного програмного забезпечення ще на дорелізному етапі;

– автоматизація розгортання – процес розгортання програмного застосунку в певному середовищі відбувається майже за натисканням однієї кнопки;
– наскрізні конвеєри – зв’язка етапів збірки, тестування та розгортання.
– підвищення ефективності аналітики та звітів – автоматизація надання інформації щодо процесу доставки.

Java, як і будь який інший стек технологій має власні особливості впровадження інструментів та практик CI/CD.

Використання систем контролю версій (наразі найбільш популярною є реалізації репозиторіїв системи Git, наприклад, GitHub, вітчизняний GitLab тощо) дозволяє розробникам працювати над кодом у спільному середовищі, зберігати його історію та впроваджувати зміни у репозиторії у більш прозорій формі.

Для автоматизації процесів збірки, автоматизації тестування та розгортання під час розробки Java-проектів використовуються такі інструменти як Jenkins, Travis CI, CircleCI або GitLab CI. Завдяки даним інструментам можна виконувати автоматичну збірку проектів та запуск тестів кожного разу (або використовуючи заданий проміжок часу), коли в репозиторії відбуваються зміни.

Для керування збіркою та залежностями в Java використовуються згадані вище системи Apache Ant, Apache Maven, Gradle та ін. Завдяки цьому, можливо описувати конфігурацію кожного модуля проекту та його залежності, автоматизувати процес збірки та керувати розробкою.

Забезпечення якості коду та вчасного виявлення помилок забезпечується різним типом тестів (юніт-тестів, інтеграційних та ін.). Найбільш часто використовуються також згадані вище фреймворки JUnit, TestNG, також Selenium, які можуть бути поєднанні із використанням BDD тестів для якіснішого тестового покриття функціоналу застосунку [28].

Автоматизація розгортання Java застосунків на тестових, проміжних та продакшн середовищах може бути реалізована завдяки таким інструментам як Docker, Kubernetes, або Ansible. Також сценарії розгортання можуть бути створені та налаштовані власноруч.

2.5 Аналіз методів підвищення ефективності збірки багатомодульного проекту

Спочатку наведемо основні поняття, що відносяться до процесу збірки проекту.

Відповідно до статті про Continuous Integration Мартіна Фаулера [29], збірка проекту — це процес, що охоплює всі кроки, необхідні для створення артефакту програмного забезпечення. У світі Java це зазвичай включає:

- генерація вихідних джерел (іноді);
- компіляція вихідних джерел;
- компіляція тестових джерел;
- виконання тестів (модульні тести, інтеграційні тести тощо);
- упаковка (у формати jar, war, ejb-jar, ear);
- виконання перевірок працездатності (статичні аналізатори, такі як Checkstyle, Findbugs, PMD, тестове покриття тощо);
- формування звітів.

Деякі частини цього процесу (компіляція вихідних джерел та компіляція тестових джерел, або ж компіляція коду в цілому) було розглянуто в попередньому пункті.

Система збірки в Java – це інструмент, який використовується для автоматизації наведених вище процесів компіляції, тестування, пакування та інших операцій, необхідних для створення виконуваного або бібліотечного файлу з вихідним кодом Java. Система збірки дозволяє легко управляти залежностями, автоматизує виконання завдань і впорядковує розробку програмного забезпечення.

Повертаючись до пункту 1.3, найбільш популярними системами збірки в Java наразі є:

- Apache Ant;
- Apache Maven;
- Gradle.

2.5.1 Аналіз використання Ant

Використовуючи наведену в пункті 1.3.1 інформацію, наведемо, що файли збірки Ant написані у форматі XML, і за домовленістю вони називаються build.xml.

Наведемо, приклад файлу збірки build.xml для простого застосунку HelloWorld. Для цього використаємо матеріал з тієї ж самої статті Baeldung, інформація з якої використовувалась у пункті 1.4:

```
<project>
    <target name="clean">
        <delete dir="classes" />
    </target>

    <target name="compile" depends="clean">
        <mkdir dir="classes" />
        <javac srcdir="src" destdir="classes" />
    </target>

    <target name="jar" depends="compile">
        <mkdir dir="jar" />
        <jar destfile="jar/HelloWorld.jar" basedir="classes">
            <manifest>
                <attribute name="Main-Class"
                  value="antExample.HelloWorld" />
            </manifest>
        </jar>
    </target>

    <target name="run" depends="jar">
        <java jar="jar/HelloWorld.jar" fork="true" />
```

```
    </target>
</project>
```

В даному файлі наведено чотири “цілі”: clean , compile , jar і run. Наприклад, ми можемо скомпілювати код, виконавши команду “ant compile”. Після запуску команди, буде запущена ціль clean, яка видалить каталог “classes”. Після цього ціль compile повторно створить каталог і скомпілює в нього папку src.

Головна перевага Ant – гнучкість. Дана система збірки не зобов’язує використати жодні умови кодування чи структуру проекту. Отже, це означає, що Ant вимагає від розробників писати всі команди самостійно, що може призвести до величезних файлів збірки XML, які важко підтримувати.

Оскільки немає ніяких угод, знання Ant не гарантує, що будь-який специфічний для певного проекту файл збірки Ant буде можливо швидко зрозуміти. Необхідність часу на зрозуміння структури незнаймого файлу збірки Ant є недоліком порівняно з іншими, новішими інструментами.

Спочатку Ant не мав вбудованої підтримки керування залежностями. Однак, оскільки керування залежностями стало майже обов’язковою властивістю систем збірки у подальшому, було розроблено підпроект Apache Ant під назвою Apache Ivy[30]. Проект інтегрований з Apache Ant і дотримується тих самих принципів дизайну.

Однак початкові обмеження Ant через відсутність вбудованої підтримки керування залежностями та розчарування під час роботи з некерованими файлами збірки XML призвели до створення Maven.

2.5.2 Аналіз використання Apache Maven

Так само, використавши матеріал пункту 2.1 та 2.1.2 тощо, наведемо, що Maven продовжує використовувати XML-файли так само, як Ant, але набагато більш керованим способом, використовуючи принцип конвенція над конфігурацією (convention over configuration).

Також цілі в Maven є попередньо визначеними, забезпечуючи в цілому чітко визначену структуру проекту.

Таким чином, Maven дозволяє розробнику зосередитися на тому, що має робити збірка, і зазначає структуру для цього. Іншим позитивним аспектом Maven є те, що він надає вбудовану підтримку для керування залежностями, що є дуже важливою характеристикою для майже усіх сучасних Java-проектів, особливо багатомодульних.

Ось приклад файлу pom.xml для того ж Java-проекту з головним класом HelloWorld:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>baeldung</groupId>
    <artifactId>mavenExample</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <description>Maven example</description>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

Однак, відмінно від Ant, структура проекту також стандартизована та відповідає умовам Maven (див. рис. 5).

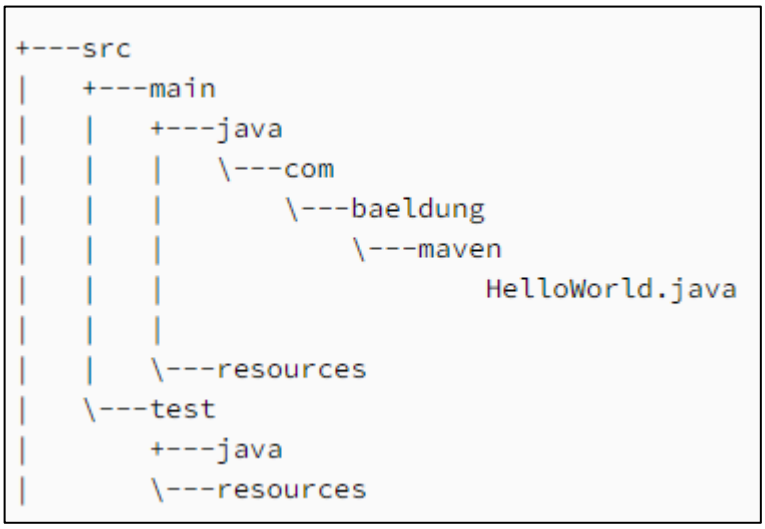

Рисунок 2.3 – Приклад структури Maven-проекту зі статті Baeldung для HelloWorld застосунку (за даними [7])

На відміну від Ant, в Maven немає потреби визначати кожну фазу процесу збирання вручну. Замість цього є можливість просто викликати вбудовані команди Maven.

Наприклад, є можливість скомпілювати код, виконавши «mvn compile».

Як зазначено на офіційних сторінках, Maven можна вважати фреймворком для виконання плагінів, оскільки вся робота виконується плагінами [31]. Maven підтримує широкий спектр доступних плагінів, і кожен з них можна додатково налаштувати.

Наведемо приклад використання одного із доступних плагінів – Apache Maven Dependency Plugin, який має ціль копіювання залежностей, що виконує копіювання залежностей до вказаного каталогу.

Щоб почати використання плагіну, включимо цей плагін у файл pom.xml і налаштуємо вихідний каталог для залежностей:

```
<build>
    <plugins>
```

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-dependency-plugin</artifactId>
                <executions>
                    <execution>
                        <id>copy-dependencies</id>
                        <phase>package</phase>
                        <goals>
                            <goal>copy-dependencies</goal>
                        </goals>
                        <configuration>
                            <outputDirectory>target/dependencies
                              </outputDirectory>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
```

Цей плагін буде виконано на етапі пакета , тому при запуску команди «mvn package», даний плагін почне свою роботу і скопіює залежності в папку «target/dependencies».

Maven дуже популярний, оскільки файли збірки стандартизовані, і для підтримки файлів збірки потрібно значно менше часу, порівняно з Ant. Однак, незважаючи на те, що конфігураційні файли Maven більш стандартизовані, ніж файли Ant, конфігураційні файли Maven все ще є великими та громіздкими.

Суворі конвенції Maven є недоліком, тому що вони набагато менш гнучкі, ніж Ant. Також варто відмітити дуже складне налаштування цілі, завдяки якому писати власні сценарії збірки набагато важче порівняно з Ant.

Незважаючи на те, що Maven намагається робити деякі серйозні вдосконалення щодо полегшення та стандартизації процесів побудови додатків, він все ще містить основний недолік того, що він набагато менш гнучкий, ніж Ant. Це призвело до створення Gradle, який поєднує в собі найкраще з обох світів – гнучкості Ant і функцій Maven.

2.5.3 Аналіз використання Gradle

Одну з перших речей, яку можна відзначити про Gradle, це те, що він не використовує файли XML, на відміну від Ant або Maven.

Згодом розробники все більше й більше цікавилися наявністю та роботою з предметно-спеціальною мовою, яка, простіше кажучи, дозволяла б їм вирішувати проблеми в певному домені за допомогою мови, адаптованої для цього конкретного домену. Gradle зміг імплементувати даний підхід, який використовує DSL на основі мов Groovy або Kotlin. Це призвело до менших файлів конфігурації з меншим безладом, оскільки мова була спеціально розроблена для вирішення конкретних проблем домену.

Ось приклад файлу build.gradle для використаного раніше простого проекту Java з попереднім основним класом HelloWorld:

```
apply plugin: 'java'


repositories {
    mavenCentral()
}


jar {
    baseName = 'gradleExample'
    version = '0.0.1-SNAPSHOT'
}
```

```
dependencies {
    testImplementation 'junit:junit:4.12'
}
```

Ми можемо скомпілювати код, виконавши команду «gradle classes».

За своєю суттю Gradle навмисно надає дуже мало функціональних можливостей. Плагіни додають усі корисні функції. У наведеному вище прикладі було використано плагін «Java», який дозволяє компілювати код Java та надає інші функції.

Gradle назвав кроки збірки «завданнями», на відміну від «цілей» Ant або «фаз» Maven. В прикладі з Maven було використано плагін залежностей Apache Maven, і це конкретна мета — скопіювати залежності у вказаний каталог. З Gradle є можливість зробити те саме, використовуючи завдання:

```
task copyDependencies(type: Copy) {
   from configurations.compile
   into 'dependencies'
}
```

Ми можемо запустити це завдання, виконавши «gradle copyDependencies».

Наведемо короткий підсумок, щодо аналізу використання розглянутих систем збірки. Було розглянуто Ant, Maven і Gradle – три інструменти автоматизації збірки Java. На даний момент, Maven займає більшість ринку інструментів для збирання та є найбільш поширеним. Однак Gradle отримав хороше застосування в більш складних кодових базах з наступних причин:

– наразі, Gradle використовується багатьох складних проектах з відкритим кодом, наприклад фреймворк Spring;
– Gradle швидший за Maven для більшості сценаріїв завдяки поступовим збіркам;
– Gradle пропонує розширені послуги аналізу та налагодження.

Однак Gradle має більш високий поріг входження, особливо для розробників не знайомих з Groovy або Kotlin.

2.5.4 Вибір оптимальної системи збірки для багатомодульного проекту в Java

Використовуючи наведену в попередніх пунктах інформацію, проведемо порівняння характеристик та спробуємо виділити найбільш оптимальну систему збірки для використання в багатомодульних проектах.

Відповідно до вимог задачі щодо дослідження ефективності процесу збірки, оберемо характеристики для порівняння систем збірки.

2.5.4.1 Керування залежностями

Характеристика є якісною, відноситься до ступеню здібності розглянутої системи збірки керувати можливостями.

Для наведеної характеристики, наведемо 3 можливі значення:

– відсутнє – позначає повну відсутність вбудованного функціоналу щодо керування залежностями;
– наявне – позначає те, що система збірки має базовий вбудований функціонал щодо керування залежностями;
– розширене – позначає наявність базового вбудованого функціоналу щодо керування залежностями, а також розширені можливості для налаштування.

2.5.4.2 Легкість сприйняття структури та конфігурації

Якісна характеристика, що описує властивість системи збірки, щодо доступності використання її функціоналу та легкості сприйняття її структури.

Значеннями для даної характеристики будуть:

– складно – система збірки потребує певних зусиль при її використанні (наприклад, власноруч створювати цілі, потребує часу на розуміння її файлу конфігурації в незнайомому проекті та ін.);

– доступно – система збірки має спрощену для використання структуру або її функціонал.

2.5.4.3 Підтримка паралельної збірки

Якісна характеристика, що описує наявність можливості збірки модулів паралельно, що може значно прискорити час її виконання.

Значення характеристики:

– відсутня – система збірки не підтримує паралельну збірку за замовчуванням;
– наявна – система збірки підтримує паралельну збірку за замовчуванням.

2.5.4.4 Підтримка інкрементної збірки

Повертаючись до пункту 2.1.1, інкрементна виконує повторну збірку лише тих фрагментів, що мали якість зміни порівняно з останньою збіркою, що значно підвищує швидкість збірки в цілому.

Якісна характеристика описує якість систем збірки, щодо наявності підтримки даного функціоналу.

Можливі значення:

– відсутня – система збірки не підтримує інкрементну збірку за замовчуванням;
– наявна – система збірки підтримує інкрементну збірку за замовчуванням.

2.5.4.5 Ступінь гнучкості та можливості розширення

Якісна характеристика, яка описує ступінь можливостей створення нового або модифікування існуючого функціоналу системи збірки тим чи іншим шляхом. Значення характеристики:

– обмежений – система збірки має обмежені можливості щодо розширення функціоналу через особливості її властивостей;

– високий – система збірки має значні можливості для розширення її функціоналу.

2.5.4.6 Підтримка створення плагінів

Якісна характеристика, що описує ступінь підтримки системою збірки можливості із створення і використання плагінів для проекту.

Значення:

– відсутня – система збірки не підтримує функціонал;
– наявна – система збірки підтримує функціонал.

2.5.4.7 Підтримка керування властивостями та змінними

Якісна характеристика, що описує ступінь підтримки системою збірки можливості із створення змінних, їх подальшого використання, а також використання властивостей.

Значення:

– відсутня – система збірки не підтримує функціонал;
– наявна – система збірки підтримує функціонал.

2.5.4.8 Якість обробки помилок

Якісна характеристика для опису ступеню якості обробки помилок, що можуть з’явитись під час процесу збірки.

Значення:

– відсутня – система збірки не підтримує даний функціонал;
– базова – система збірки надає базову інформацію щодо помилок;
– розширена – система збірки надає докладну інформацію щодо помилок.

2.5.4.9 Підтримка кешування залежностей

Якісна характеристика, що описує наявність в системі збірки підтримки кешування залежностей.

Значення:

– відсутня – система збірки не підтримує функціонал;
– наявна – система збірки підтримує функціонал.

2.5.4.10 Зведення характеристик систем збірки в одну таблицю

Для подальшого використання наведених основних характеристик систем збірки, зведемо всі дев’ять в одну таблицю.

Щоб спростити сприйняття наведеної інформації в таблиці, що буде створена, використаємо скорочення для наведених характеристик:

– керування залежностями – DM (dependency management);
– легкість сприйняття структури та конфігурації – CS (structure clarity);
– підтримка паралельної збірки – PS (parallelism support);
– підтримка інкрементної збірки – IS (incremental support);
– ступінь гнучкості та можливості розширення – FLX (flexibility);
– підтримка створення плагінів – PES (plugins ecosystem support);
– підтримка керування властивостями та змінними – VS (variables support);
– якість обробки помилок – EP (errors processing);
– підтримка кешування залежностей – DCS (dependency caching support).

Таблиця 2.2 – Характеристики систем збірки (виконана самостійно)

	Система збірки		
Характеристика	Apache Ant	Apache Maven	Gradle
DM	відсутнє	наявне	розширене
CS	складно	доступно	доступно
PS	відсутня	наявна	наявна
IS	відсутня	наявна	наявна
FLX	високий	обмежений	високий
PES	відсутня	наявна	наявна
VS	наявна	наявна	наявна
EP	базова	розширена	розширена

Кінець таблиці 2.2

DCS	відсутня	наявна	наявна

Завдяки попереднім крокам можна побачити розглянуті характеристики систем збірки в табличному наданні.

2.5.4.11 Перетворення якісних значень характеристик у кількісні

Відповідно до наведених характеристик, зробимо їх перетворення у кількісні. Значення якісних характеристик, що наведені у таблиці будуть перетворені у числа, що розташовані у порядку зростання від нуля та позначують ступінь «корисності» тієї чи іншої характеристики. Таким чином, ми отримаємо наступні можливі значення.

DM:

– відсутнє – 0;
– наявне – 1;
– розширене – 2.

CS:

– складно – 0;
– доступно – 1.

PS:

– відсутня – 0;
– наявна – 1.

IS:

– відсутня – 0;
– наявна – 1.

FLX:

– обмежений – 0;
– високий – 1.

PES:

– відсутня – 0;

– наявна – 1.

VS:

– відсутня – 0;

– наявна – 1.

EP:

– відсутня – 0;

– базова – 1;

– розширена – 2.

DCS:

– відсутня – 0;

– наявна – 1.

З наведеного вище, отримаємо показники коефіцієнтів, замість відповідних якісних значень. Завдяки цьому, інформація може бути розглянута з математично-аналітичної точки зору в подальшому та можуть бути застосовані методи вибору оптимального рішення відповідно до поставленої задачі обрання оптимальної системи збірки для багатомодульних застосунків Java. Наприклад, може бути використано функцію згортку.

Спочатку, наведемо нову таблицю із перетвореними якісними характеристиками у відповідні числові характеристики, в даному випадку всі з яких виглядають як цілі значення в інтервалах від нуля до одного, або від нуля до двох (див. табл. 2.3).

Таблиця 2.3 – Оновлені характеристики систем збірки після їх перетворення у числові (виконана самостійно)

	Система збірки		
Характеристика	Apache Ant	Apache Maven	Gradle
DM	0	1	2
CS	0	1	1
PS	0	1	1

Кінець таблиці 2.3

IS	0	1	1
FLX	1	0	1
PES	0	1	1
VS	1	1	1
EP	1	2	2
DCS	0	1	1

2.5.4.12 Аналіз наведених характеристик щодо відповідності до принципу «за максимумом»

Проведемо аналіз наведених характеристик, та розглянемо їх з точки зору відповідності до принципу «за максимумом».

Характеристики DM, EP – описують певну «перевагу» від наявності та якості функціоналу. Отже характеристики є нормованими.

Характеристики PS, IS, PES, VS, DCS – також описують перевагу від наявного функціоналу, таким чином дані характеристики є нормованими.

Характеристики CS, FLX – визначають «виграш» у якості функціоналу. Характеристики є нормованими.

Випливаючи з результатів аналізу, можна сказати, що наведені вище характеристики дотримуються принципу «за максимумом» та не потребують нормування у даному випадку.

2.5.4.13 Аналіз множин за принципом Парето

Проведемо аналіз наведених множин за принципом Парето [32], що зможе допомогти викреслити варіанти, що відразу є гіршими за всіма показниками. Проаналізуємо найгірші значення кожної із характеристик (див. табл. 2.4).

Таблиця 2.4 – Нижчі показники характеристик (виконана самостійно)

Характеристика	Система з найгіршим показником	Показник
DM	Apache Ant	0
CS	Apache Ant	0
PS	Apache Ant	0
IS	Apache Ant	0
FLX	Apache Maven	0
PES	Apache Ant	0
VS	-	-
EP	Apache Ant	1
DCS	Apache Ant	0

З аналізу можна побачити, що Apache Ant програє майже у всіх категоріях, однак він не може бути виключений з таблиці за принципом Парето через те, що є характеристики, в яких Apache Ant не є найгіршим. Судячи з даних, можна передбачити, що в подальшому аналізі Apache Ant все одно не буде оптимальним рішенням, однак є необхідним розглянути наступні кроки вибору оптимальної системи збірки для багатомодульних застосунків Java.

Проте, можна побачити, що показник VS у всіх розглянутих системах збірки є однаковим, та в даному випадку може бути виключений з аналізу.

2.5.4.14 Нормування оцінок за шкалами

Для подальшого аналізу даних, виконаємо нормування оцінок. Для нормування оцінок використаємо принцип мінімакс нормування [33].

Мінімакс нормування – метод нормування, в якому максимальне та мінімальне значення в масиві даних використовуються для нормування всіх інших значень до діапазону між 0 і 1. Формула мінімакс нормування

$$X' = \frac{X - min}{max - min} \quad (2.1)$$

де X – значення, що необхідно нормувати,

min – мінімальне значення за шкалою,

max – максимальне значення за шкалою.

Для більш зручного відображення даних мінімальне та максимальне значення кожної розглянутої характеристики за шкалою буде наведено у таблиці 2.5.

Таблиця 2.5 – Мінімальні та максимальні значення характеристик (виконана самостійно)

Характеристика	Мінімальне значення	Максимальне значення
DM	0	2
CS	0	1
PS	0	1
IS	0	1
FLX	0	1
PES	0	1
EP	1	2
DCS	0	1

Використовуючи розглянуті дані, можемо розрахувати нормовані значення за наведеною вище формулою за кожною з характеристик.

Для кращого розуміння процесу нормалізації даних, розглянемо нормалізацію однієї з характеристик – DM. Характеристика має мінімальне значення за шкалою – 0 та максимальне значення – 2. Використаємо дані з таблиці 3. Для Apache Ant X = 0. В наведеній вище формулі, верхня частина буде дорівнювати нулю, отже нормоване значення буде також дорівнювати 0. Тепер розглянемо Apache Maven, де X = 1. Підставимо значення у формулу:

$$X'(Apache\ Maven) = \frac{1-0}{2-0} = \frac{1}{2} \quad (2.2)$$

Отже, маємо нормоване значення, що дорівнює 0,5. Для Gradle значення буде дорівнювати одиниці, так як це є максимальним значенням.

Виконуючи подібні перетворення для всіх характеристик, отримаємо таблицю із нормованими значеннями (див. табл. 2.6).

Таблиця 2.6 – Таблиця із нормованими значеннями за мінімакс нормуванням (виконана самостійно)

	Система збірки		
Характеристика	Apache Ant	Apache Maven	Gradle
DM	0	0,5	1
CS	0	1	1
PS	0	1	1
IS	0	1	1
FLX	1	0	1
PES	0	1	1
EP	0	1	1
DCS	0	1	1

2.5.4.15 Надання вагових коефіцієнтів характеристикам

Наддамо кожній з характеристик вагові коефіцієнти, що відображають їх важливість при використанні для багатомодульних Java проектів.

Вагові коефіцієнти будуть поділятися наступним чином. Низька важливість матиме коефіцієнт – 0,25, середня важливість – 0,5, максимальна важливість – 1. Отже коефіцієнти мають значення від 0,25 до 1.

Характеристика DM є дуже важливою. Підтримка керування залежностями є однією з надзвичайно важливих функцій для більшості сучасних проектів. Отже вона матиме максимальну важливість.

Характеристика CS – є важливою. Зручна структура використання системи збірки впливає на якість розробки та її ефективність тощо. Важливість – середня.

Характеристики PS, IS та DCS – в контексті задачі роботи є дуже важливою, отже надає можливості значно прискорити швидкість збірки проекту, таким чином підвищивши рівень ефективності процесу збірки багатомодульного проекту. Важливість – максимальна.

Характеристика FLX – має середню важливість через контекст задачі роботи. Гнучкі можливості розширення системи збірки надають перевагу в ефективності процесу збірки, завдяки чому можна оптимізувати певні процеси системи збірки.

Характеристика PES – має максимальну важливість. Створювання плагінів дозволяє використовувати більш ефективний з точки зору споживання пам’яті та часу блоки коду, що можуть значно підвищити ефективність процесу збірки.

Характеристика EP – в контексті задачі має низьку важливість. Відстеження помилок є важливим в цілому процесом під час розробки будь-якого функціоналу або ПЗ, однак завдяки більш важливим характеристикам, таким як FLX та PES цей процес може бути більш оптимізованим для системи збірки, що має низькі показники EP.

Використовуючи аналіз важливості характеристик, можемо навести таблицю із коефіцієнтами важливості кожної з них поділену на загальне значення суми коефіцієнтів 6,25 (див. табл. 2.7).

Завдяки наведеним ваговим коефіцієнтам можна побачити яка з характеристик є найбільш важливою відповідно до контексту дослідження процесу розгортання багатомодульних застосунків, що використовують Java. Дані коефіцієнти надалі будуть використані під час використання засобів згортки для дослідження найбільш оптимальної системи збірки для багатомодульних застосунків, що використовують не тільки Java, але й інші сумісні із JVM мови програмування.

Таблиця 2.7 – Нормовані характеристики з ваговими коефіцієнтами характеристик (виконана самостійно)

Характеристика	Система збірки			Ваговий коефіцієнт
	Apache Ant	Apache Maven	Gradle	
DM	0	0,5	1	0,16
CS	0	1	1	0,08
PS	0	1	1	0,16
IS	0	1	1	0,16
FLX	1	0	1	0,08
PES	0	1	1	0,16
EP	0	1	1	0,04
DCS	0	1	1	0,16

2.5.4.16. Використано лінійну адитивну згортку

Для рішення задачі вибору оптимальної системи збірки багатомодульних проектів при врахуванні багатьох критеріїв, оптимальним буде використання лінійної адитивної згортки з ваговими коефіцієнтами, отже вона дозволяє обирати які критерії є більш важливими в контексті дослідження. Формула лінійної адитивної згортки:

$$Z' = \max_{i=1,m} \sum_{j=1}^{n} \alpha_j \beta_j x_{ij} \quad (2.3)$$

де α_j – нормуючі множники,

β_j – вагові коефіцієнти, що відображають відносний внесок окремих критеріїв до загального критерію.

Для прикладу, розрахуємо значення згортки для Apache Maven.

$$Z'(Apache\ Maven) = 0{,}5 * 0{,}16 + 0{,}08 + 0{,}16 + 0{,}16 + 0{,}16 + 0{,}04 + 0{,}16 = 0{,}84 \quad (2.4)$$

Отже, значення оптимальності для Maven в даному випадку дорівнює 0,84. Значення оптимальності для інших систем збірки наведені в таблиці 2.8.

Таблиця 2.8 – Оптимальність систем збірки (виконана самостійно)

Система Збірки	Оптимальність
Apache Ant	0,08
Apache Maven	0,84
Gradle	1

Максимальне значення лінійної адитивної згортки, що дорівнює 1 має Gradle, це можна було спрогнозувати ще на етапі, коли була побудована таблиця 2.7, однак значення лінійної адитивної згортки тільки підтвердило думку. Найнижчий результат має Apache Ant із значенням 0,08. Maven є близьким до Gradle за корисністю, проте, якщо аналізувати коефіцієнти корисності кожної з систем збірки, Gradle є ефективнішим на 19% за Maven та на 92% ефективнішим за Apache Ant. Отже, випливаючи з результатів аналізу оптимальності систем збірок для багатомодульних проектів на Java, можна спрогнозувати, що використання Gradle більш ефективним для процесу збірки багатомодульного Java проекту.

Також, необхідно врахувати рекомендації щодо підвищення ефективності процесу компіляції коду та пункту 2.1.

2.5.5 Налаштування паралельної збірки в Gradle

Паралельна збірка в gradle може бути запущена як командою із флагом «--parallel», наприклад «gradle build –parallel», так і налаштована в файлі build.gradle:

```
// Увімкнути паралельну збірку
```

```
gradle.startParameter.parallel = true


// Налаштувати кількість потоків під час паралельної збірки (необов'язково)
gradle.startParameter.maxWorker = 4
```

В наведеному вище прикладі, паралельна збірка вмикається завданням значення «true» властивості «gradle.startParameter.parallel». Для коригування кількості доступних потоків для системи збірки, можна використовувати властивість «gradle.startParameter.maxWorker», значенням якої є власне кількість доступних потоків.

2.5.6 Налаштування інкрементної збірки в Gradle

Для налаштування інкрементної збірки в Gradle, ми можемо виконати наступні модифікації в файлі build.gradle:

```
// Увімкнути інкрементну збірку
tasks.withType(JavaCompile) {
    options.incremental = true
}
```

Для того, щоб увімкнути інкрементну збірку, ми зазначаємо, що для задачі компіляції коду Java («JavaCompile»), властивість «options.incremental» буде мати значення «true».

2.5.7 Поєднання паралельної та інкрементної збірки в Gradle

Використовуючи інформацію з попередніх пунктів 2.2.5 та 2.2.6, можна увімкнути паралельну та інкрементну збірку в Gradle одночасно, завдяки чому значно підвищиться ефективність процесу компіляції коду та збірки проекту. Після відповідних модифікацій файлу build.gradle нові рядки матимуть наступний вигляд:

```
// Увімкнути паралельну збірку
gradle.startParameter.parallel = true
// Налаштувати кількість потоків під час паралельної збірки (необов'язково)
gradle.startParameter.maxWorker = 4
// Увімкнути інкрементну збірку
tasks.withType(JavaCompile) {
    options.incremental = true
}
```

2.5.8 Запуск тестів в паралельному режимі в Gradle

Ще одним з методів підвищення швидкості збірки (а отже й ефективності процесу збірки) в Gradle є можливість виконувати тести паралельно.

Для цього також можна зробити наступні модифікації в build.gradle:

```
test {
    // Увімкнути паралельне виконання для юніт-тестів
    maxParallelForks = 4
}
integrationTest {
    // Увімкнути паралельне виконання для інтеграційних тестів
    maxParallelForks = 2
}
```

Властивості «maxParallelForks» строго кажучи, задають кількість доступних потоків виконання для тестів.

Слід пам'ятати що не всі тести можуть бути виконаними паралельно. Певні з них можуть мати загальний доступ до контексту, що призведе проблеми із синхронізацією даних, або звертатися до зовнішніх ресурсів (часто таке

відбувається в інтеграційному тестуванні). Таким чином необхідно чітко розуміти, якщо тести можуть бути виконані паралельно.

Отже, під час пункту 2.2 було розглянуто засоби підвищення ефективності процесу збірки багатомодульних Java-проектів. Використано математичне прогнозування та пошук оптимальної системи збірки проектів, якою виявилася Gradle. Також розглянуто рекомендації із пункту 2.1 із використанням даної системи збірки, завдяки яким було наведено засоби включення інкрементної збірки, паралельної збірки а також паралельного виконання тестів в Gradle.

2.6 Аналіз методів підвищення ефективності процесів після збірки багатомодульного проекту

Проаналізуємо які процеси можуть бути оптимізовані та які методи можна застосувати на етапі, коли компіляція та збірка проекту є завершеною.

2.6.1 Мінімізація та оптимізація розміру JAR-файлу

Після того, як було отримано вихідний JAR-файл, можна використати інструменти для мінімізації та обфускації [34] коду. Одним з таких інструментів може бути ProGuard [35].

ProGuard – це інструмент для мінімізації, обфускації та оптимізації Java-байткоду. Він призначений для зменшення розміру та підвищення безпеки Java-застосунків. Принцип роботи ProGuard включає наступні етапи:

- мінімізація (Minification) – ProGuard здійснює мінімізацію шляхів, імен змінних та методів, а також покращення імен класів, які не є частиною API. Це допомагає скоротити розмір коду та зробити його менш зрозумілим для аналізу;
- обфускація (Obfuscation) – обфускація полягає в перейменуванні класів, методів та змінних в незрозумілі та короткі імена. Наприклад, заміна «*longVariable*» на «*a*» або «*myMethod*» на «*a()*». Це робить код важчим для розуміння при зверненні до нього сторонніми особами;

– вилучення коду (Code Shrinking) – ProGuard може вилучити код, який не використовується у програмі, таким чином допомагаючи зменшити розмір виконуваного файлу;
– оптимізація (Code Optimization) – під час оптимізації ProGuard може виконати ряд оптимізацій, таких як згортання констант, вилучення недосяжного коду, спрощення виразів та інші оптимізації, які покращують швидкість та ефективність коду;
– заборона відладки та вилучення інформації про джерело (Debug Information Removal) – ProGuard може прибрати інформацію для відладки, таку як рядки відлагодження та назви файлів з вихідного коду, щоб ускладнити аналіз та злам коду;
– обробка виключень (Exception Handling) – процес обфускації може бути важливим у випадках використання виключень у коді, оскільки він може впливати на стек виклику.

Важливо враховувати, що в деяких випадках обфускація може призвести до проблем у випадках, коли ім'я класу, методу чи змінної використовується зовнішнім кодом або конфігурацією. Тому перед використанням ProGuard важливо вивчати документацію та налаштовувати його правильно для конкретного проекту.

2.6.2 Кешування результатів

Для збереження проміжних артефактів та результатів збірки можна використовувати системи кешування результатів. Це може допомогти уникнути повторного виконання одних і тих самих операцій для модулів, які не змінилися.

В Gradle за замовчуванням кеш збірки вимкнуто [36]. Увімкнути кешування результатів збірки можна двома шляхами:

– при збірці використовувати прапорець «*--build-cache*». Таким чином кешування результатів збірки буде увімкнуто для збірки де було застосовано наведений вище прапорець;
– встановити значення властивості «*org.gradle.caching*» в «*true*» («*org.gradle.caching=true*») в файлі gradle.properties. Таким чином Gradle

буде намагатися знову використати результати збірки з попередніх запусків для всіх збірок, поки не буде явно наведено прапорець «*--no-build-cache*», який вимикає даний функціонал для збірки де він був застосований.

Коли кеш збірки ввімкнено, він зберігатиме вихідні дані збірки в Gradle User Home.

2.6.3 Стратегії завантаження залежностей

Для збереження та подальшого використання артефактів, можна використовувати стратегії завантаження залежностей, такі як завантаження артефактів з локальних репозиторіїв або використання проксі-серверів для збереження артефактів.

Наприклад, в Gradle можна налаштувати джерела для завантаження артефактів. Приклад build.gradle із використанням різних джерел для завантаження артефактів:

```
repositories {
    mavenCentral()
    maven {
        url "https://repo.spring.io/release"
    }
    maven {
        url "https://repository.jboss.org/maven2"
    }
}
```

В даному прикладі ми маємо стандартне джерело «*mavenCentral*» та два власноруч налаштованих, що використовують URL-посилання на репозиторії.

Також можуть бути використані власні плагіни для завантаження отриманих артефактів у певний репозиторій (наприклад, налаштований JFrog) для їх подальшого використання.

2.6.4 Оптимізація аналізу коду

Для оптимізації якості коду після збірки, можна використовувати інструменти для його аналізу (наприклад, SonarQube, що був розглянутий в пункті 2.3.3.4). В даному випадку, можна налаштувати задачу в Gradle, що буде автоматично запускати аналіз коду після завершення його компіляції.

Для цього, створимо нову задачу в Gradle, що буде виконуватись після задач компіляції коду та компіляції тестів в проекті та використаємо плагін SonarQube.

Приклад налаштування нової задачі:

```
plugins {
    id 'org.sonarqube' version '3.3'
}

// Налаштування властивостей SonarQube
sonarqube {
    properties {
        property 'sonar.host.url', 'http://your-sonarqube-server'
        property 'sonar.projectKey', 'your-project-key'
        property 'sonar.projectName', 'Your Project Name'
        // Можливо додати інші властивості SonarQube, якщо це необхідно
    }
}

// Створення задачі для запуску аналізу SonarQube після компіляції
task sonarAnalysis(dependsOn: ['compileJava', 'compileTestJava']) {
    doLast {
```

```
            // Запустити аналіз SonarQube використовуючи його налаштовані властивості
            sonarqube.run()
        }
    }

    // Очікуємо, що задача sonarqube завершиться після запуску sonarAnalysis
    sonarAnalysis.finalizedBy sonarqube
```

Як можна побачити з прикладу, спочатку необхідно додати плагін SonarQube актуальної версії (на даний момент це є 3.3).

Наступний крок – створення конфігурації для зв'язку із платформою SonarQube: посилання на платформу, секретний ключ доступу до проекту та його назва. Важливо зазначити, що значення властивості «*sonar.projectKey*» повинно бути приховане та не використовуватись у публічному просторі.

Далі створюємо задачу для запуску аналізу після завершення компіляції основного коду та коду тестів.

Як результат, маємо автоматичний запуск перевірок якості коду на етапі після закінчення збірки, що дозволить розробнику вчасно запобігти розгортання проблемних модулів у публічне середовище.

2.6.5 Автоматизація тестування

Зазвичай, всі тести в Gradle виконуються автоматично під час запуску процесу збірки. Проте, якщо існують окремо налаштовані команди для іншого роду тестів (наприклад, інтеграційних), то додання умови їх автоматичного виконання також дозволить запобігти помилок у зібраних модулях. Особливо це є важливим для багатомодульних застосунків з складною код-базою та великою кількістю залежностей. Однак не завжди всі інтеграційні тести можуть бути виконані без підготовки певних умов (як приклад – коли проект має інтеграцію зі іншими

проектами та не може бути з’єднаний з іншим проектом без налаштувань властивостей доступу та відсутності його тестового «замінника»).

2.6.6 Моніторинг та оптимізація продуктивності

Для моніторингу продуктивності в процесі виконання збірки та після нього можна використовувати різні інструменти для моніторингу продуктивності. Навіть базовий моніторинг дозволить відстежити проблемні місця в модулях та оптимізувати процес збірки. Особливо це є важливим коли збірка виконується в автоматизованій системі (наприклад, Jenkins) та має обмежений ресурс пам’яті та процесорної потужності.

Після закінчення збірки локально інструменти для моніторингу зможуть надати можливість спостерігати час та споживання пам’яті для кожного з модулів, бачити більш детальні логи щодо збірки. Для автоматизованих систем гарною практикою буде нотифікація команди розробки щодо результатів збірки (наприклад через месенджери такі як Slack, або електронною поштою). Звіт повинен містити як мінімум статус збірки, її лог або посилання на нього та час збірки.

Також для автоматизованих систем, можна налаштувати окремий процес, що кожен проміжок часу (як приклад – раз в день, кожен ранок або ніч перед початком робочого дня) буде дивитись на поточну гілку з кодом проекту та автоматично намагатись зібрати проект. Це корисно як для малих, так і великих проектів, та й не тільки тих, що використовують Java.

Завдяки цьому можна буде своєчасно оновлювати дані метрик якості коду в SonarQube або інших подібних інструментах чи системах, бачити які залежності не перейшли перевірку на вразливості CVE [37] та інші активності, пов’язані із контролем якості коду. Це дозволить відразу дізнатись про стан та наявні проблеми із артефактами багатомодульного проекту та зменшити зусилля команди щодо роботи над підтримкою процесу збірки багатомодульних проектів, що навіть при використанні наведених в попередніх пунктах рекомендаціях можуть займати певний час, особливо в випадках, коли трапляються специфічні помилки або неточності.

2.6.7 Інтеграція з іншими інструментами

Для забезпечення ізольованого середовища виконання та відокремлення залежностей є гарною практикою використання інших інструментів, таких як системи контейнеризації (наприклад, Docker [38]).

Docker – це платформа для розробки, розгортання та виконання додатків у контейнерах. Контейнери в Docker упаковують застосунок та його залежності разом в єдиний контейнер, що може бути запущений на будь-якому обладнанні, яке підтримує Docker. Іншими словами Docker дозволяє вирішувати проблеми із залежностями середовища, розгортати застосунки швидко та ефективно, а також спрощує масштабування та управління застосунками в різних середовищах.

Docker є потужним інструментом для контейнеризації застосунків, і може допомогти значно полегшити роботу з багатомодульними застосунками Java наступними перевагами:

– ізоляція модулів – кожен модуль може бути упакований в окремий Docker-контейнер. Це забезпечить ізоляцію між модулями та уніфікацію залежностей;
– забезпечення переносимості – Docker дозволяє створювати контейнери, які містять всі залежності та середовище виконання. Це робить додаток переносимим між різними середовищами, незалежно від конфігурацій ОС та встановлених бібліотек;
– спрощення залежностей – Docker може також бути використаний для визначення і управління залежностями, зокрема базами даних чи службами. Це допомагає спростити налаштування середовища розробки та тестування;
– швидке розгортання – контейнери можна легко та швидко розгортати, що особливо важливо для тестування та постійної інтеграції (CI/CD). За допомогою Docker Compose можна також легко налаштовувати та запускати багатомодульні додатки;

– визначення середовища роботи – використання Dockerfile дозволяє визначити середовище роботи для кожного модуля, включаючи конфігурації, залежності та параметри запуску;
– масштабування – Docker допомогає управляти та масштабувати окремі сервіси незалежно один від одного;
– інтеграція з іншими інструментами – Docker може легко інтегруватися з іншими інструментами для автоматизації розгортання, моніторингу та оркестрації контейнерів (наприклад, Kubernetes [39]);
– збільшення ізоляції та безпеки – контейнери Docker забезпечують ізоляцію ресурсів та середовища, що підвищує безпеку та допомагає уникнути конфліктів між модулями.

Загалом, Docker сприяє полегшенню розробки, тестування та розгортання багатомодульних Java-застосунків, роблячи їх більш гнучкими, переносимими та ефективними.

Отже, в пункті 2.6 було розглянуто існуючі рекомендації щодо поліпшення ефективності процесів після збірки багатомодульного проекту. Було наведено декілька налаштувань, що можуть бути виконані для розглянутої раніше системи збірки Gradle. Також наведено рекомендації із процесу автоматизованої збірки проекту та використання систем контейнеризації, що надає перевагу в керуванні залежностями модулів, їх ізоляцією та саме основне – розгортанням їх у різних середовищах.

Кожна із рекомендацій може надавати різні переваги та мати різні недоліки для різних проектів. Отже наведені вище рекомендації в більшості є специфічними відповідно до структури багатомодульних проектів на Java та виступають у ролі кращих інженерних практик для використання під час циклу розробки та використання багатомодульних проектів.

Команда, що відповідає за проект (розробники, тестувальники, DevOps чи будь-які інші сторони) повинна аналізувати чи є пріоритетним впровадження певних рекомендацій, або ж чи взагалі принесе користь відповідно до вимог та налаштувань програмного проекту та середовища розробки.

2.7 Аналіз методів підвищення ефективності розгортання JAR-файлів багатомодульного проекту

Відповідно до наведеного раніше опису процесу розгортання проекту, можна виділити два ключові питання, що можуть бути проаналізовані із метою підвищення ефективності: що саме розгортати та де розгортати.

2.7.1 Використання інструментів контейнеризації

Щодо першого, процес створення самої структури розгортання може бути спрощено при використані систем контейнеризації, наприклад, Docker, який був розглянутий в пункті 2.6.7.

Наведемо приклад використання Docker для розгортання. Система є дипломною роботою 2022 року на тему бекенд-частина платформи для опитувань серед студентів ХНУРЕ. Структура проекту являє собою Java застосунок, що використовує систему збірки Maven (див. рис. 2.4).

Проект NURE Survey містить класичну монолітну одномодульну структуру Java проекту із використанням системи збірки проектів Apache Maven. Пакети Java-коду програми містяться в директорії «src» та його піддиректорії «main». Також в проекті є директорія, що містить необхідні для системи збірки Apache Maven файли, а саме «.mvn». В директорії проекту можна побачити файл «db_init.sql», що необхідний розробникам під час конфігурації локального середовища, а саме створення бази даних. Файли «docker-compose.yml», «Dockerfile_Core», «Dockerfile_Db» відповідають за розгортання середовища у контейнеризованому вигляді, використовуючи інструмент контейнеризації Docker. Приклад наповнення даних файлів буде розглянуто далі. Файл «docker_env.env» відповідає за конфігурацію змінних середовища для інструменту контейнеризації Docker. Іншими словами, даний файл відповідає за налаштування властивостей специфічних до кожного середовища (або секретних даних). Інші файли відносяться до системи збірки Apache Maven, або UI-частини проекту та наразі не будуть розглядатись в даному прикладі.

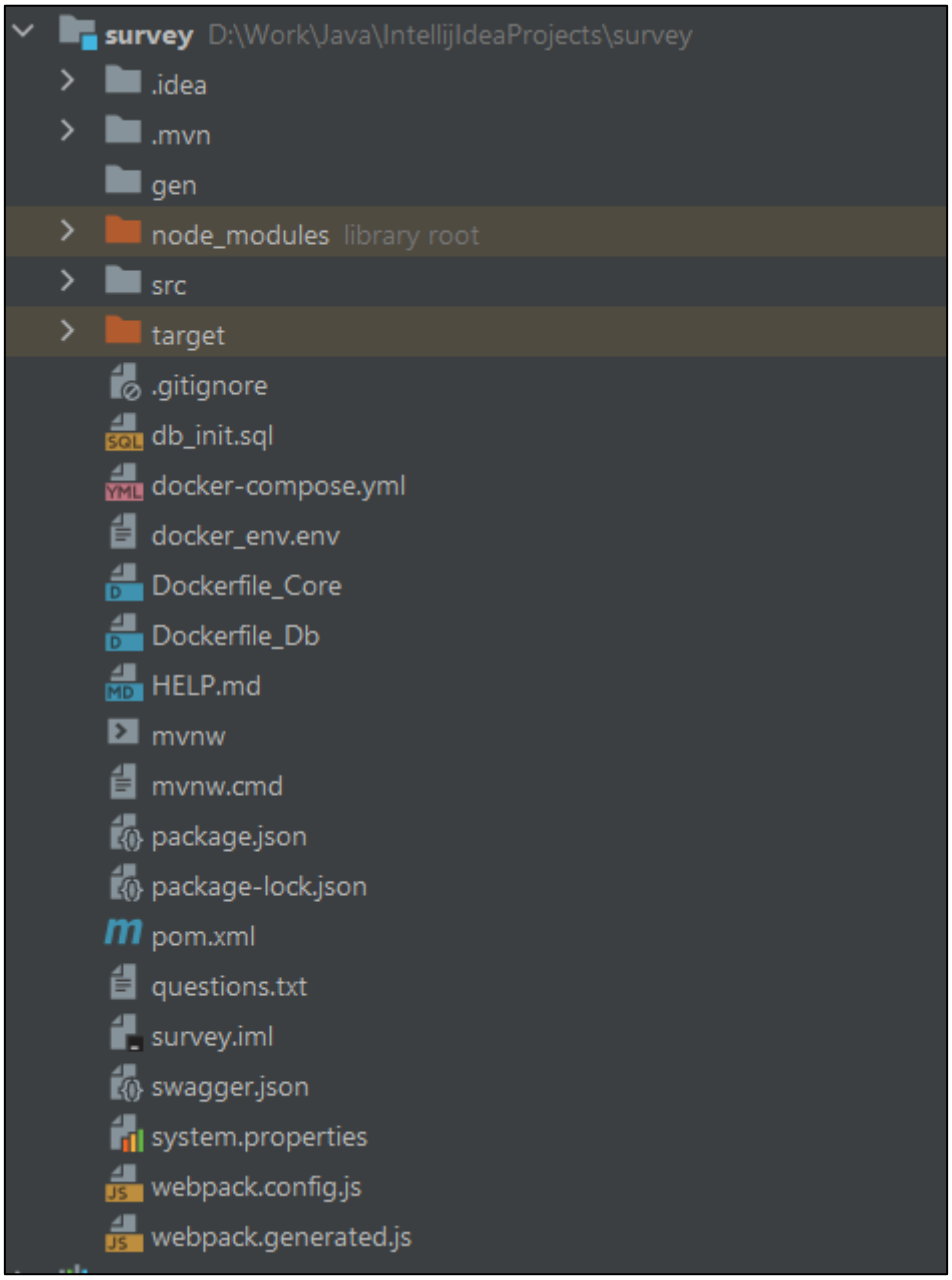

Рисунок 2.4 – Структура проекту NURE_SURVEY (створено самостійно)

На даному прикладі, наведемо спосіб використання Docker та Docker-Compose. Для цього в першу чергу розглянемо файл, із «інструкцією» для інструменту контейнеризації Docker із розгортання у контейнері кінцевого артефакту (jar-файлу) використовуючи OpenJDK версії 11. Цей файл має назву «Dockerfile_Core» (див. рис. 2.5). Також для розгортання використовуються властивості середовища із згаданого вище файлу «docker_env.env» в якому наводяться значення змінних. Більш детально використання змінних середовища, буде розглянуто далі.

Dockerfile_Core

Safe mode, limited functionality. Trust the project to access full IDE functionality.

```
1  FROM openjdk:11
2  ARG JAR_FILE=target/*.jar
3  COPY ${JAR_FILE} app.jar
4  ENTRYPOINT ["java","-jar","/app.jar"]
5
6  ARG datasource_url
7  ARG datasource_user_name
8  ARG datasource_user_password
9  ARG google_oauth2_client_id
10 ARG google_oauth2_client_secret
11 ARG url_origin
12 ARG cloudinary.cloud_name
13 ARG cloudinary.api_key
14 ARG cloudinary.api_secret
15
16 ENV DATASOURCE_URL /$datasource_url
17 ENV DATASOURCE_USER_NAME /$datasource_user_name
18 ENV DATASOURCE_USER_PASSWORD /$datasource_user_password
19 ENV GOOGLE_OAUTH2_CLIENT_ID /$google_oauth2_client_id
20 ENV GOOGLE_OAUTH2_CLIENT_SECRET /$google_oauth2_client_secret
21 ENV URL_ORIGIN /$url_origin
22 ENV CLOUDINARY_CLOUD_NAME /$cloudinary.cloud_name
23 ENV CLOUDINARY_API_KEY /$cloudinary.api_key
24 ENV CLOUDINARY_API_SECRET /$cloudinary.api_secret
```

Рисунок 2.5 – Dockerfile для основного проекту «*Dockerfile_Core*» (створено самостійно)

Файл «*Dockerfile_Core*» містить дані для розгортання саме модулю проекту. Строчки 1-4 зазначають який JDK повинен бути використаний (в даному випадку OpenJDK для одинадцятої версії Java), та який JAR-файл повинен бути виконаний.

Строчки 6-14 зазначають аргументи, які використовуються модулем проекту. Це є дані для підключення до бази даних, систем авторизації і так далі.

Строчки 16-24 встановлюють значення аргументів рівними до значень змінних середовища.

```
FROM postgres:13.3
COPY db_init.sql /docker-entrypoint-initdb.d/
```

Рисунок 2.6 – Dockerfile для бази даних «Dockerfile_Db» (створено самостійно)

В даному файлі зазначається версія бази даних (PostgreSQL 13.3) та у другій строчці копіювання SQL-скрипту для її ініціалізації.

docker_env.env

```
POSTGRES_DB=nure_survey
POSTGRES_USER=
POSTGRES_PASSWORD=
PGADMIN_DEFAULT_EMAIL=
PGADMIN_DEFAULT_PASSWORD=
DATASOURCE_URL=jdbc:postgresql://postgres_container:5432/nure_survey
DATASOURCE_USER_NAME=
DATASOURCE_USER_PASSWORD=
GOOGLE_OAUTH2_CLIENT_ID=
GOOGLE_OAUTH2_CLIENT_SECRET=
MAIL_SMTP_USER_NAME=
MAIL_SMTP_USER_PASSWORD=
MAIL_SURVEY_FEEDBACK_RECIPIENTS=
URL_ORIGIN=
CLOUDINARY_CLOUD_NAME=
CLOUDINARY_API_KEY=
CLOUDINARY_API_SECRET=
```

Рисунок 2.7 – Змінні середовища Docker в файлі «docker_env.env» (створено самостійно)

Файл «docker_env.env» містить змінні, які будуть використані Docker для їх встановлення у програмному модулі та для підключення до бази даних. В даному випадку це є локальним наданням файлу, розробник сам їх заповнює для локальних потреб розробки. При розгортанні, дані властивості автоматично встановлюються іншими інструментами.

Для об’єднання розгорнутих контейнерів та їх більш детального керування використано Docker-Compose. Приклад файлу «*docker-compose.yml*» для розглянутого проекту:

```
version: "3.7"
services:
  postgres:
    container_name: postgres_container
    image: survey-core-db:latest
    build:
      context: .
      dockerfile: Dockerfile_Db
    command:
      - "postgres"
      - "-c"
      - "max_connections=50"
      - "-c"
      - "shared_buffers=1GB"
      - "-c"
      - "effective_cache_size=4GB"
      - "-c"
      - "work_mem=16MB"
      - "-c"
      - "maintenance_work_mem=512MB"
      - "-c"
      - "random_page_cost=1.1"
      - "-c"
      - "temp_file_limit=10GB"
      - "-c"
      - "log_min_duration_statement=200ms"
      - "-c"
      - "idle_in_transaction_session_timeout=10s"
      - "-c"
      - "lock_timeout=1s"
      - "-c"
      - "statement_timeout=60s"
      - "-c"
      - "shared_preload_libraries=pg_stat_statements"
      - "-c"
      - "pg_stat_statements.max=10000"
      - "-c"
      - "pg_stat_statements.track=all"
    env_file:
      - docker_env.env
    ports:
      - "5433:5432"
    restart: unless-stopped
    deploy:
      resources:
        limits:
          cpus: '1'
          memory: 4G
    networks:
      - postgres
```

```
  pgadmin:
    container_name: pgadmin_container
    image: dpage/pgadmin4:5.7
    env_file:
      - docker_env.env
    environment:
      PGADMIN_CONFIG_SERVER_MODE: "False"
    volumes:
      - ./pgadmin:/var/lib/pgadmin
    ports:
      - "5050:80"
    restart: unless-stopped
    deploy:
      resources:
        limits:
          cpus: '0.5'
          memory: 1G
    networks:
      - postgres

  core:
    container_name: survey_core_container
    build:
      context: .
      dockerfile: Dockerfile_Core
    image: survey-core:latest
    env_file:
      - docker_env.env
    ports:
      - "8081:8081"
    networks:
      - postgres

networks:
  postgres:
    driver: bridge
```

Для кожного проекту конфігурація буде відрізнятись, проте для розглянутого випадку, ми маємо керування розгортанням контейнеру бази даних, контейнера інструменту PGAdmin та контейнеру із модулем проекту.

Також налаштовано одну группу з'єднання для того, щоб програмний модуль мав можливість мати з'єднання із контейнером PostgreSQL. Таким чином дана програмна система може бути розгорнута автоматично без необхідності керувати залежностями між модулями та підготовлювати середовище окрім встановлення Docker.

Отже, в даному випадку був розглянутий метод підвищення ефективності процесу розгортання JAR-файлів багатомодульних проектів завдяки інструментам

контейнеризації. Замість Docker може бути використаний будь-який інший інструмент в залежності від потреб проекту або інших пунктів (наприклад, питання щодо ліцензії використання).

2.7.2 Використання інструментів автоматизації для процесу CI/CD в Java

Розглянемо доволі потужний стек технологій, що значно прискорюють розгортання застосунків. Впровадження інструментів автоматизації процесу постійної інтеграції та постійної доставки (CI/CD) для проектів як на Java, так і для монолітних багатомодульних структур, має значний ряд переваг, частину з яких було наведено в аналізі процесу CI/CD.

Існує багато різноманітних інструментів, які можуть виконувати функції з автоматизації CI/CD процесів. Далі буде проаналізовано автоматизація на прикладі одного з найбільш популярних засобів, а саме – Jenkins.

Jenkins є потужним та дуже корисним відкритим для використання інструментом в розробці програмного забезпечення. Для проектів, що використовують JVM-сумісні мови, даний інструмент може бути особливо корисним, отже він має високий ступінь інтегрованості з іншими інструментами для розробки на Java, наприклад, раніше розглянутими системи збірки Maven та Gradle та робить більш контрольованим процеси розробки.

Основними функціями Jenkins є автоматизація процесів розробки завдяки автоматизуванню будь-якого із етапів, включаючи збірку, тестування, аналіз якості коду, розгортання та ін. Зміни, після збірки можуть бути інтегровані автоматично у центральний репозиторій та автоматично запускати процеси будування та тестування, для більш швидкого виявлення помилок. Збірки, що пройшли всі перевірки можуть бути автоматично перетворені в нову версію продукту та підготовлені до розгортання у відповідному середовищі.

Розглянемо приклад використання Jenkins використовуючи простий Java-застосунок, що виведе стрічку «Hello World» у консоль.

```
public class Main {
```

```
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

В Jenkins створимо нову задачу із назвою «build-snapshot», що буде відповідати за збірку проекту використовуючи систему збірки Gradle (див. рис. 2.8).

Рисунок 2.8 – Налаштування взаємодії з GitHub у Jenkins (створено самостійно)

Як джерело даних будемо використовувати систему контроля версій Git та репозиторій GitHub.

Налаштуємо автоматичну очистку робочого простору задачі після старту процесу збірки. Це дозволить уникнути ситуацій із надмірним використанням постійної пам’яті на сервері, де використовується інструмент Jenkins. Таким чином, при одночасному виконанні декількох задач, що мають високий коефіцієнт використання постійної пам’яті ми будемо мати менший ризик виникнення проблем її переповнення. Це особливо корисно для проектів, що містять в собі ряд більш дрібних проектів, збірки яких можуть відбуватись одночасно на одному сервері інструмента Jenkins.

Рисунок 2.9 – Налаштування середовища збірки в Jenkins (створено самостійно)

Далі налаштовано властивості середовища збірки. Як можна побачити на рисунку, спочатку робочі дані будуть стерті, проект кожного разу буде перезавантажено з репозиторію, а у разі помилок та занадто довгого виконання збірка буде автоматично припинена через 3 хвилини.

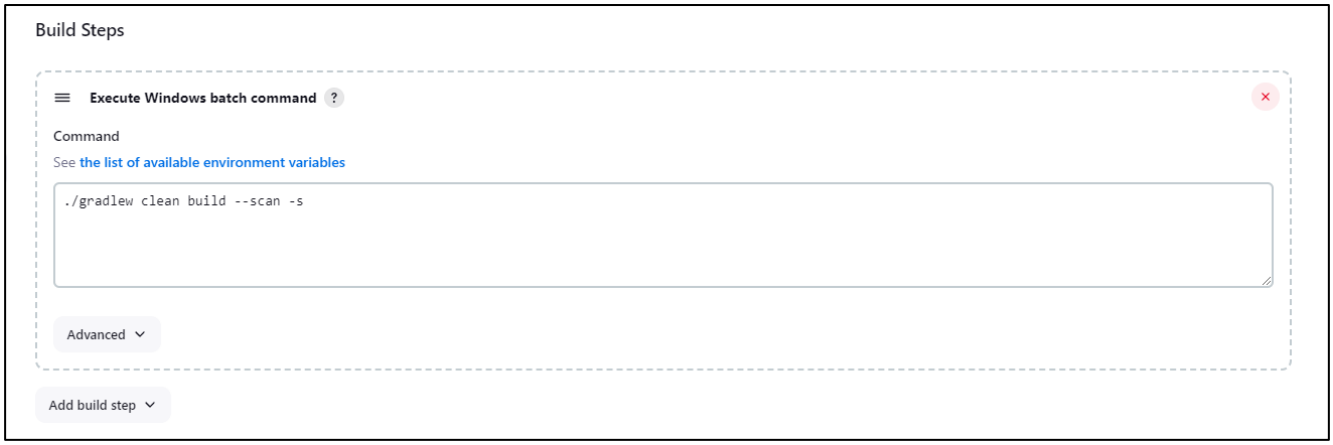

Рисунок 2.10 – Крок збірки в Jenkins (створено самостійно)

Далі налаштуємо саме крок збірки, який буде здійснено системою Gradle. Для цього зазначимо, щоби в робочому середовищі було відкрито командну строку

Windows batch (для Jenkins, що встановлені на іншому середовищі, необхідно використовувати Shell) та виконано команду «./gradlew clean build –scan -s» яка, в свою чергу дасть команду самому Gradle очистити та перезібрати проект. Також додано параметри «--scan» та «-s» для більш детального аналізу логів у разі помилок.

Після записку збірки ми отримаємо наступний результат.

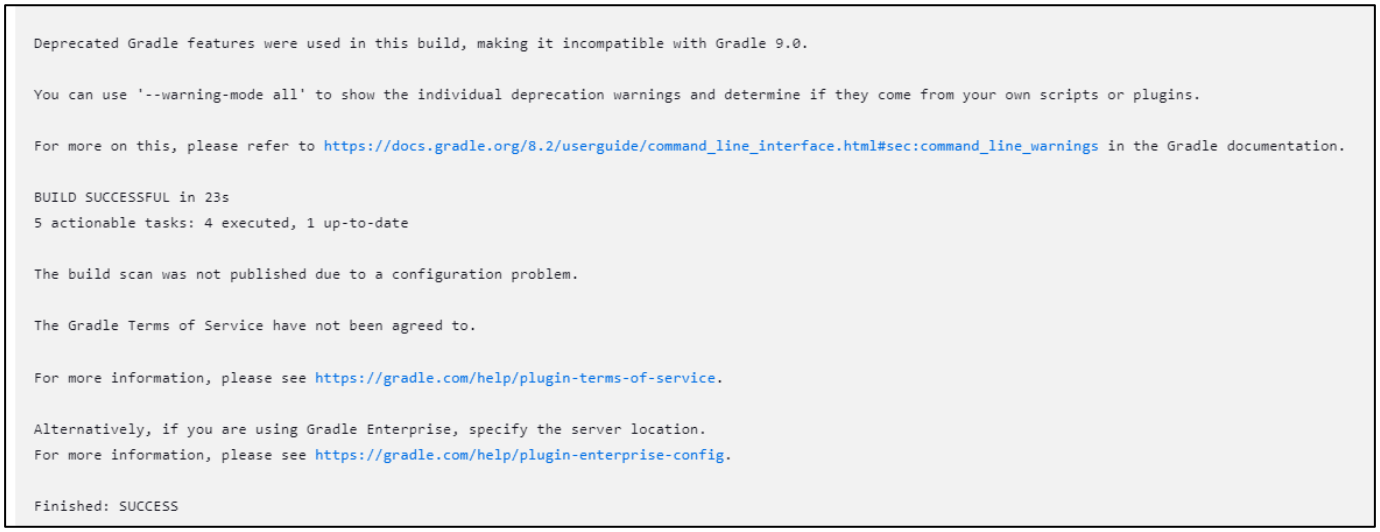

Рисунок 2.11 – Фрагмент консолі виводу збірки в Jenkins (створено самостійно)

Проект успішно зібрано, а також у віконці із збірками, ми побачимо зелений статус та час останньої збірки.

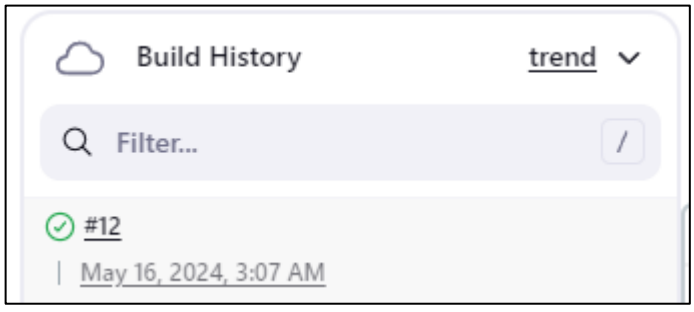

Рисунок 2.12 – Вікно історії збірок в Jenkins (створено самостійно)

В подальшому, можуть бути створені окремі задачі, котрі будуть залежати від результату збірки проекту. Наприклад, задача для випуску нової версії застосунку, що буде використовувати результат останньої збірки задачі «build-snapshot».

2.7.3 Вибір оптимального хмарного середовища для розгортання

В сучасному світі корпоративної розробки, дуже велика кількість проектів обирають одне із найбільш популярних хмарних середовищ для розгортання програмних модулів замість використання власних потужностей. Розглянемо основні причини, чому хмарні середовища наразі мають більшу перевагу.

Масштабованість – хмарні платформи надають можливість масштабувати ресурси за потребою. Можливо легко збільшити або зменшити обсяг обчислювальних та мережевих ресурсів в залежності від обсягу роботи або трафіку. Це особливо важливо для проектів, які можуть зазнавати змін в завантаженні з часом.

Доступність та надійність – хмарні платформи забезпечують високий рівень доступності, оскільки вони використовують розподілені системи та резервне зберігання даних. Проект може бути розгорнутий на різних регіонах, забезпечуючи високу доступність для користувачів з усього світу.

Ефективне використання ресурсів – хмарні платформи дозволяють ефективно використовувати ресурси, оскільки вони можуть автоматично налаштовувати кількість інстансів в залежності від навантаження. Оплата йде лише за фактично використані ресурси, що робить ефективними витрати на інфраструктуру.

Швидкість розгортання та масштабування – хмарні платформи надають можливість швидко розгортати та масштабувати середовище для проекту. Немає необхідності чекати на придбання нового обладнання чи його налаштування. Завдяки широкому спектру готових служб, є можливість легко використовувати бази даних, кешування, системи моніторингу та інші сервіси, що полегшує роботу.

Автоматизація та управління – хмарні платформи забезпечують ряд інструментів для автоматизації рутинних завдань, таких як розгортання, масштабування та моніторинг. Інтерфейси управління та API дозволяють легко контролювати і взаємодіяти з середовищем.

Безпека – хмарні платформи надають продвинені засоби безпеки, такі як шифрування даних, ідентифікація та авторизація на рівні користувача, системи контролю доступу тощо. Зазвичай, хмарні платформи забезпечують високий стандарт захисту даних.

Отже, спробуємо зробити оптимальний вибір хмарної платформи на прикладі сервісу для опитувань, що був розглянутий раніше. Для цього виділимо найбільш популярні хмарні платформи, а саме:

– Amazon Web Services (AWS) [40];
– Microsoft Azure [41];
– IBM Cloud [42];
– Google Cloud (GCloud) [43];
– Oracle Cloud [44].

Не зупиняючись на описі кожної з платформ (що є окремим дослідженням), оберемо основні критерії, що впливають на оптимальність вибору платформи для розглянутої програмної системи для опитувань.

Таким чином ми маємо бекенд-частину із UI для внутрішнього використання (конструктор опитувань та вікно адміністрування) та базу даних. Використовуючи дані із сценаріїв реального використання даної системи, ми знаємо, що нам необхідно хоча б віртуальних ядра та 2 гігабайти оперативної пам’яті. Для порівняння хмарних платформ будемо використовувати один і той самий регіон – східна частина США. Дана система мала функціонал із збереження звітів щодо опитувань та аналізу результатів, необхідний оптимальний простір для зберігання звітів на місяць – хоча б 5 гігабайт.

Це означає, що характеристиками, що матимемо наступні характеристики хмарних платформ для розглядання:

а) вартість використання серверу (дол. США) (не включаючи вартість використання сторонніх сервісів, таких як сховища даних, системи безпеки та ін.) в місяць (VM$);

1) AWS – AWS має сервіс EC2 для оренди інстансів. Кожен інстанс може мати свій план використання. Для потреб даного застосунку є план «t4g.small» із вартістю використання 6,13$ в місяць;
2) Microsoft Azure – Найбільш дешевшою віртуальною машиною в Microsoft Azure, що задовольняє потреби порівнянного застосунку є машини класу «A2», що мають характеристики – 2 ядра, 3,5 гігабайта оперативної пам’яті та тимчасове сховище – 60Гб. Вартість використання подібної віртуальної машини в місяць – 57,67$;
3) IBM Cloud – віртуальна машина в IBM Cloud, що є дешевшою та містить характеристики найближчі до необхідних є «cx2-2x4» із 2 ядрами та 4 гігабайтами пам’яті. Вартість використання в місяць – 73,36$;
4) Google Cloud – віртуальна машина в Google Compute Engine – «e2-small» має характеристики в 2 гігабайта оперативної пам’яті та 2 віртуальних процесора. Вартість використання в місяць – 12,23$;
5) Oracle Cloud – Oracle Cloud має сервіс віртуальної машини із процесором Ampere, конфігурацією «VM.Standard.A1.Flex», 4 віртуальними ядрами та 24 гігабайтами оперативної пам’яті. Вартість використання на місяць – 4,25$.

б) доступна оперативна пам’ять відповідно до моделі віртуальної машини (RAM);
   1) AWS – машини типу «t4g.small» мають 2 гігабайти оперативної пам’яті;
   2) Microsoft Azure – віртуальна машина «A2» в Azure має 3,5 гігабайти оперативної пам’яті;
   3) IBM Cloud – віртуальна машина «cx2-2x4» має 4 гігабайти оперативної пам’яті;
   4) Google Cloud – віртуальна машина «e2-small» має 2 гігабайти оперативної пам’яті;

5) Oracle Cloud – віртуальна машина конфігурації «VM.Standard.A1.Flex» із процесором типу Ampere має доступні 24 гігабайти оперативної пам’яті.

в) доступна кількість віртуальних ядер відповідно до моделі віртуальної машини (VCPU);

1) AWS – машини типу «t4g.small» мають 2 віртуальних ядра;
2) Azure – машини класу «A2» мають 2 віртуальних ядра;
3) IBM Cloud – машини «cx2-2x4» мають 2 віртуальних ядра;
4) Cloud – машини класу «e2-small» мають 2 віртуальних ядра;
5) Cloud – віртуальна машина конфігурації «VM.Standard.A1.Flex» із процесором типу Ampere має доступні 4 віртуальних ядра.

г) кількість безкоштовних гігабайт для зберігання логів при моніторингу метрик застосунку (LOGS);

1) AWS – AWS CloudWatch має безкоштовний рівень використання, що надає 5 Гб для логів;
2) Microsoft Azure – Azure Monitor не має безкоштовного місця для логів. Вартість використання становить 0,50$ за один гігабайт логів в базовому рівні використання;
3) IBM Cloud – сервіс IBM Cloud Monitoring with Sysdig також не має безкоштовної моделі використання, в тому числі місця для логів;
4) Google Cloud – сервіс Google Logging storage надає безкоштовно 50 гігабайт логів;
5) Cloud – Oracle Cloud має 10 безкоштовних гігабайт логів.

д) вартість використання сервісу зберігання файлів (не включаючи тарифікацію за зберігання або зчитування даних) за 5 гігабайт зберігання на рівні Cold storage на місяць (ST$).

1) AWS – AWS S3 має ціну використання 0,12$ за 5 гігабайт сховища на місяць;
2) Microsoft Azure – Azure Blob-storage рівня зберігання Cold має вартість використання 0,018$ за 5 гігабайт сховища на місяць;

3) IBM Cloud – IBM Cloud Storage для Cold зберігання 5 гігабайт файлів на місяць із використанням тарифного плану Smart коштує 0,039$;
4) Google Cloud – Google Cloud Storage для Cold зберігання 5 гігабайт файлів на місяць коштуватиме 0,1$;
5) Oracle Cloud – Oracle Cloud Object Storage – Infrequent Access має вартість використання 5 гігабайт на місяць, що дорівнює 0,05$.

Наведемо опис та аналіз шкал за кожним з обраних критеріїв з зазначенням типу шкали.

Таблиця 2.9 – Аналіз шкал характеристик платформ (виконана самостійно)

Характеристика	Обмеження	Тип шкали	Значення критерію
VM$	$X \geq 0$	числова, абсолютна	6,13 57,67 73,36 12,23 4,25
RAM	$X > 0$	числова, абсолютна	2 3.5 4 24
VCPU	$X > 0, X \in N$	числова, абсолютна	2 4
LOGS	$X \geq 0$	числова, абсолютна	0 5 10 50
ST$	$X \geq 0$	числова, абсолютна	0,12 0,018 0,039 0,1 0,05

Наведемо векторний опис альтернатив використовуючи табличне надання (див. табл. 2.10).

Таблиця 2.10 – Векторний опис альтернатив за обраними критеріями (виконана самостійно)

Назва сервісу	VM$	RAM	VCPU	LOGS	ST$
AWS	6,13	2	2	5	0,12
Microsoft Azure	57,67	3,5	2	0	0,018
IBM Cloud	73,36	4	2	0	0,039
Google Cloud	12,23	2	2	50	0,1
Oracle Cloud	4,25	24	4	10	0,05

Далі перетворимо векторний опис з метою приведення всіх шкал до принципу оптимальності «за максимумом».

Перетворення VM$ – максимальне значення – 73,36. Нова характеристика – VMS (Virtual Machine Savings). Відображає економію при оренді віртуальної машини відповідно до вимог.

Перетворення RAM – мінімальне значення – 2. Нова характеристика – ARAM (Additional RAM). Відображає додаткову кількість оперативної пам’яті після 2-х необхідних гігабайт.

Перетворення VCPU – мінімальне значення – 2. Нова характеристика – AVCPU (Additional Virtual CPU). Відображає додаткову кількість віртуальних ядер процесора окрім 2-х необхідних.

Перетворення LOGS – Перетворення LOGS не є необхідним в даному контексті, так як дана характеристика вже оптимізована за принципом оптимальності «за максимумом».

Перетворення ST$ – максимальне значення – 0,12. Нова характеристика – STS (Storage Savings). Відображає економію при оренді 5 гігабайт сховища рівня Cold Storage.

Таблиця 2.11 – Таблиця із новими характеристиками (виконана самостійно)

Назва сервісу	VMS	ARAM	AVCPU	LOGS	STS
AWS	67,23	0	0	5	0
Microsoft Azure	15,69	1,5	0	0	0,102
IBM Cloud	0	2	0	0	0,081
Google Cloud	61,13	0	0	50	0,02
Oracle Cloud	69,11	22	2	10	0,07

Використаємо метод аналізу характеристик за принципом Парето.

Найгірша VMS – IBM Cloud. найгірші ARAM – AWS, Google Cloud, найгірші AVCPU – всі, крім Oracle Cloud, найгірші LOGS – Microsoft Azure, IBM Cloud, найгірший STS – AWS.

Отже, за принципом Парето не можливо виявити ні однієї хмарної платформи, що могла бути одразу викреслена із порівняння.

Наступним етапом є нормування оцінок за шкалами, використовуючи принцип мінімаксу, так само, як це використовувалось в пункті 2.5.4.14.

Таблиця 2.12 – Дані після нормування (виконана самостійно)

Назва сервісу	VMS	ARAM	AVCPU	LOGS	STS
AWS	0,9728	0	0	0,1	0

Кінець таблиці 2.12

Microsoft Azure	0,2270	0,0682	0	0	1
IBM Cloud	0	0,0909	0	0	0,7941
Google Cloud	0,8845	0	0	1	0,1961
Oracle Cloud	1	1	1	0,2	0,6863

Наведемо вагових коефіцієнтів до кожної з характеристик.

Вагові коефіцієнти, також як в пункті 2.5.4.15 будуть поділятися наступним чином. Низька важливість матиме коефіцієнт – 0,25, середня важливість – 0,5, максимальна важливість – 1. Отже коефіцієнти мають значення від 0,25 до 1.

Характеристиками з найбільшою важливістю в даному випадку є VMS, ARAM, AVCPU.

Характеристика з середньою важливістю – STS.

Характеристика з низькою важливістю – LOGS.

Наведемо нові значення вагових коефіцієнтів до кожної з характеристик у табличному вигляді (див. табл. 2.13).

Таблиця 2.13 – Таблиця із доданими нормованими коефіцієнтами ваги (виконана самостійно)

Назва сервісу	VMS	ARAM	AVCPU	LOGS	STS
AWS	0,9728	0	0	0,1	0
Microsoft Azure	0,2270	0,0682	0	0	1
IBM Cloud	0	0,0909	0	0	0,7941
Google Cloud	0,8845	0	0	1	0,1961

Кінець таблиці 2.13

Oracle Cloud	1	1	1	0,2	0,6863
Коефіцієнт ваги	3/12 (0,25)	3/12 (0,25)	3/12 (0,25)	1/12 (0,0833)	2/12 (0,1667)

Використаємо лінійну адитивну згортку для вибору оптимального рішення.

$$Z'(AWS) = 0{,}9728 * 0{,}25 + 0{,}1 * 0{,}0833 \cong 0{,}2432 + 0{,}0083 = 0{,}2515$$

$$Z'(Microsoft\ Azure) = 0{,}2270 * 0{,}25 + 0{,}0682 * 0{,}25 + 0{,}1667 \cong 0{,}05675 + 0{,}01705 + 0{,}1667 = 0{,}2405$$

$$Z'(IBM\ Cloud) = 0{,}0909 * 0{,}25 + 0{,}7941 * 0{,}1667 \cong 0{,}022725 + 0{,}13237647 \cong 0{,}1551$$

$$Z'(Google\ Cloud) = 0{,}8845 * 0{,}25 + 0{,}0833 + 0{,}1961 * 0{,}1667 \cong 0{,}221125 + 0{,}0833 + 0{,}03268987 \cong 0{,}3371$$

$$Z'(Oracle\ Cloud) = 0{,}25 + 0{,}25 + 0{,}25 + 0{,}2 * 0{,}0833 + 0{,}6863 * 0{,}1667 \cong 0{,}25 + 0{,}25 + 0{,}25 + 0{,}01666 + 0{,}11440621 \cong 0{,}8811$$

Таким чином, отримуємо наступну таблицю із коефіцієнтами корисності кожної з хмарних платформ.

Таблиця 2.14 – Таблиця із коефіцієнтами корисності (виконана самостійно)

Назва сервісу	VMS	ARAM	AVCPU	LOGS	STS	Z
AWS	0,9728	0	0	0,1	0	0,2515
Microsoft Azure	0,2270	0,0682	0	0	1	0,2405

Кінець таблиці 2.14

IBM Cloud	0	0,0909	0	0	0,7941	0,1551
Google Cloud	0,8845	0	0	1	0,1961	0,3371
Oracle Cloud	1	1	1	0,2	0,6863	0,8811
Коефіцієнт ваги	3/12 (0,25)	3/12 (0,25)	3/12 (0,25)	1/12 (0,0833)	2/12 (0,1667)	

Отже, було розраховано коефіцієнти корисності для кожного із хмарних сервісів. Випливаючи із результатів можна спрогнозувати, що найбільш оптимальним є вибір Oracle Cloud, коефіцієнт корисності якого дорівнює 0,8811. Найгіршим варіантом є IBM Cloud із коефіцієнтом 0,1551.

Oracle Cloud, як можна побачити в таблиці, є кращим у трьох одних з найбільш важливих критеріях – VMS, ARAM, AVCPU, що робить цю платформу більш вигідною та ефективною для розгортання застосунку.

Підводячи підсумок щодо пункту 2.4, можна зазначити, що як і в попередніх пунктах рішення для підвищення ефективності можуть бути специфічними відповідно до вимог та характеристик проекту. Було розглянуто практичне використання інструментів контейнеризації (Docker), та наведено аналіз вибору оптимального хмарного середовища для Java-застосунку.

2.8 Створення системи рекомендацій

2.8.1 Виведення плану системи рекомендацій

Використовуючи наведений раніше матеріал, спробуємо систематизувати рекомендації використання яких спростить процес створення багатомодульних проектів на Java перетворюючи їх на інструкцію-шаблон.

Дамо назву першому кроку інструкції «Архітектурний аналіз». На даному етапі потрібно приділити увагу планованому функціоналу застосунка. Якщо бізнес-логіка складається з однієї простої функції, слід використовувати мікросервісну

архітектуру, замість монолітної. Також необхідно проаналізувати з командою та стейкхолдерами потенційне навантаження на програмний продукт для розуміння потенціалу масштабування.

Наступний крок матиме назву «Технічний аналіз та планування». Обираються оптимальні технології, фреймворки та інструменти для вирішення технічних задач (використовуючи наведений раніше матеріал). Обов'язкове використання системи контролю версій, бажано Git. Обираються засоби CI/CD та середовища розгортання застосунку. Створюються технічні середовища для внутрішнього застосування в компанії. Такі як задачі автоматизації, налаштування безпеки, стандарти та ін.

Більш практичний крок «Оптимізована імплементація» відповідає безпосередньо за створення кодової бази програмного проекту. На даному етапі застосовуються всі рекомендації, що були наведені раніше для етапів компіляції коду та збірки проекту, такі як оптимізація коду, інструментів збірки проекту, локальної перевірки коду та тестування та ін. Застосуються ручні перевірки якості коду через код ревью при об'єднанні гілок у репозиторії проекту до головної.

Наступний крок «Оптимізована доставка та інтеграція». Кодова база використовується в засобах автоматизації CI/CD (наприклад, Jenkins). Код автоматизовано збирається в артефакт (JAR або WAR-файл), проходить перевірки, та автоматизовано розгортається у необхідному середовищі. Також при цьому необхідно виконати різні задачі етапу після збірки проекту відповідні до вимог та контексту багатомодульного проекту на Java. Наприклад, автоматично генерується звіт із покриття коду тестами для кожного модуля, завдяки інструментам автоматичного звітування. Використовуються різні системи із моніторингу якості коду, такі як оновлення даних щодо якості коду, наявності вразилвостей та проблем із безпекою проекту в автоматизованій системі SonarQube або її аналогах, якщо подібні системи налаштовані в корпоративному середовищі проекту. Після успішних та неуспішних збірок, йде автоматична нотифікація відповідальних осіб зі сторони команди проекту та можливих команд, що відповідають за технічну частину корпоративного середовища.

Розглянемо план зображений графічно (див. рис. 2.13).

Рисунок 2.13 – План систематизованих рекомендацій (створено самостійно)

На рисунку зображено поступові етапи плану систематизованих рекомендацій.

2.8.2 Використання системи рекомендацій на реальному прикладі

Як приклад, розглянемо використання хоча б частини систематизованих рекомендацій на реальному проекті. Для цього буде використаний згаданий раніше проект с системи опитувань для ХНУРЕ (якщо буде більш точним, його бекенд-частина разом з UI-частиною конструктора опитувань).

Зазначимо основні відомості про технічні аспекти проекту. Проект має монолітну архітектуру, яка має лише один великий модуль, містить шар представлення даних (UI та REST API), пакети для бізнес-логіки, роботи із середовищами зберігання даних (реляційною базою даних PostgreSQL та хмарним середовищем зберігання зображень Cloudinary) та ін. Рівень Java – 11, система збірки проекту – Maven, фреймворк – Spring Boot. База даних та програмний модуль контейнеризовано завдяки Docker.

Застосуємо пункт «Архітектурний аналіз». З архітектурного застосування можемо побачити, що проект не може бути перетворений на мікросервісную

архітектуру через те, що має спільну кодову базу в конструкторі опитувань, так і в REST API. REST API, в свою чергу використовується для окремого застосунка, що є клієнтською частиною системи. Також, помітно наступну проблему – масштабованість. Система розраховувалась на можливість опитування одночасно до 100 студентів, однак, використання єдиного модуля, що відповідає як за REST API, так і за UI частину потенційно може блокувати роботу одне одного у разі високих навантажень, та робить неможливим розподілення ресурсів окремо на API та UI, іншими словами, даний функціонал не може бути розгорнутий окремо один від одного. Отже, перша зміна, для покращення ефективності – перехід до багатомодульної архітектури.

Використаємо пункт «Технічний аналіз та планування». Перехід від Maven до Gradle в даному випадку буде припущено, отже це не є необхідним в контексті даного прикладу, та призведе до значних затрат ресурсів із мінімальним покращенням ефективності. Подібні ситуації можливі і в реальній розробці, коли проект необов'язково відразу переводити на іншу технологію якщо даний перехід коштуватиме занадто більше ефекту від переходу. Отже, наступним кроком буде створення відокремлення від репозиторію проекту (fork), та початок впровадження змін до проекту.

Для втілення змін, розіб'ємо кодову базу проекту (див. рис. 2.14) (в більшості пов'язані із кожним логічним шаром пакети програмного проекту) на окремі модулі. Такі пакети як «rest» та «ui» буде використано як основу для двох нових модулів представлення данних, які відповідно використовуються як UI-частина із конструктором опитувань імплементована на Vaadin. Друга частина буде використовуватись як REST API, що відповідає за комунікацію із основною клієнтською частиною проекту NURE Survey, а саме «Nure Survey UI», яка є окремим підпроектом та не розглядається в даній задачі. Інші пакети будуть перетворені на такі модулі як «service», що відповідає за бізнес-логіку застосунку, «domain», що зберігає доменні сутності проекту, «persistence», що відповідає за комунікацію із середовищами зберігання даних.

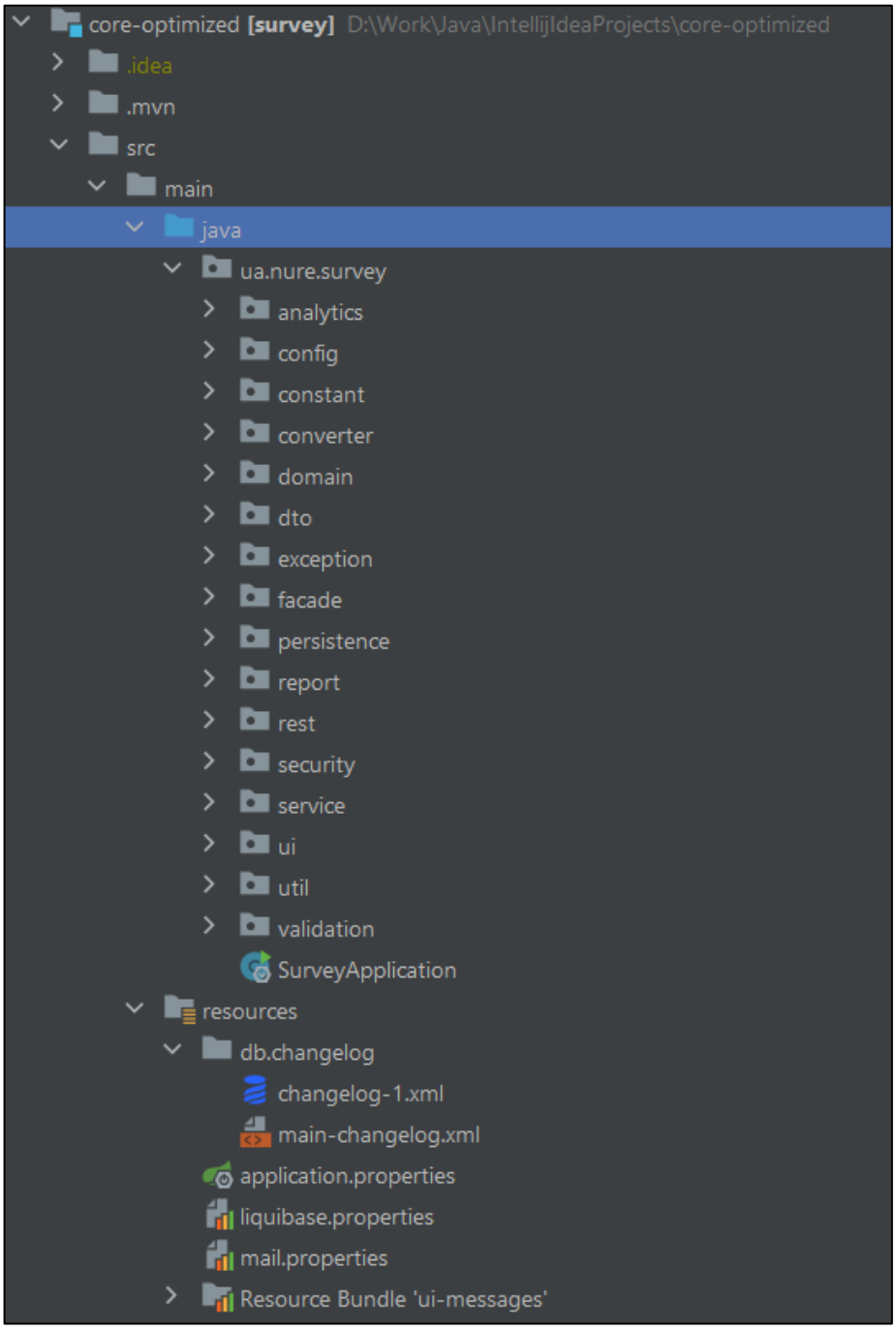

Рисунок 2.14 – Структура проекту до початку змін (створено самостійно)

Після переходу на багатомодульну архітектуру, використовуючи Maven, ми маємо наступну структуру проекту (див. рис. 2.15).

Як можна побачити, ми отримали відокремлені автономні модулі, два з яких можуть бути використані як незалежні артефакти («core-rest» та «core-ui»). Таким чином вирішено питання щодо розгортання окремої частини функціоналу на окремих серверах та масштабованості. Знижено рівень пов’язаності коду проекту.

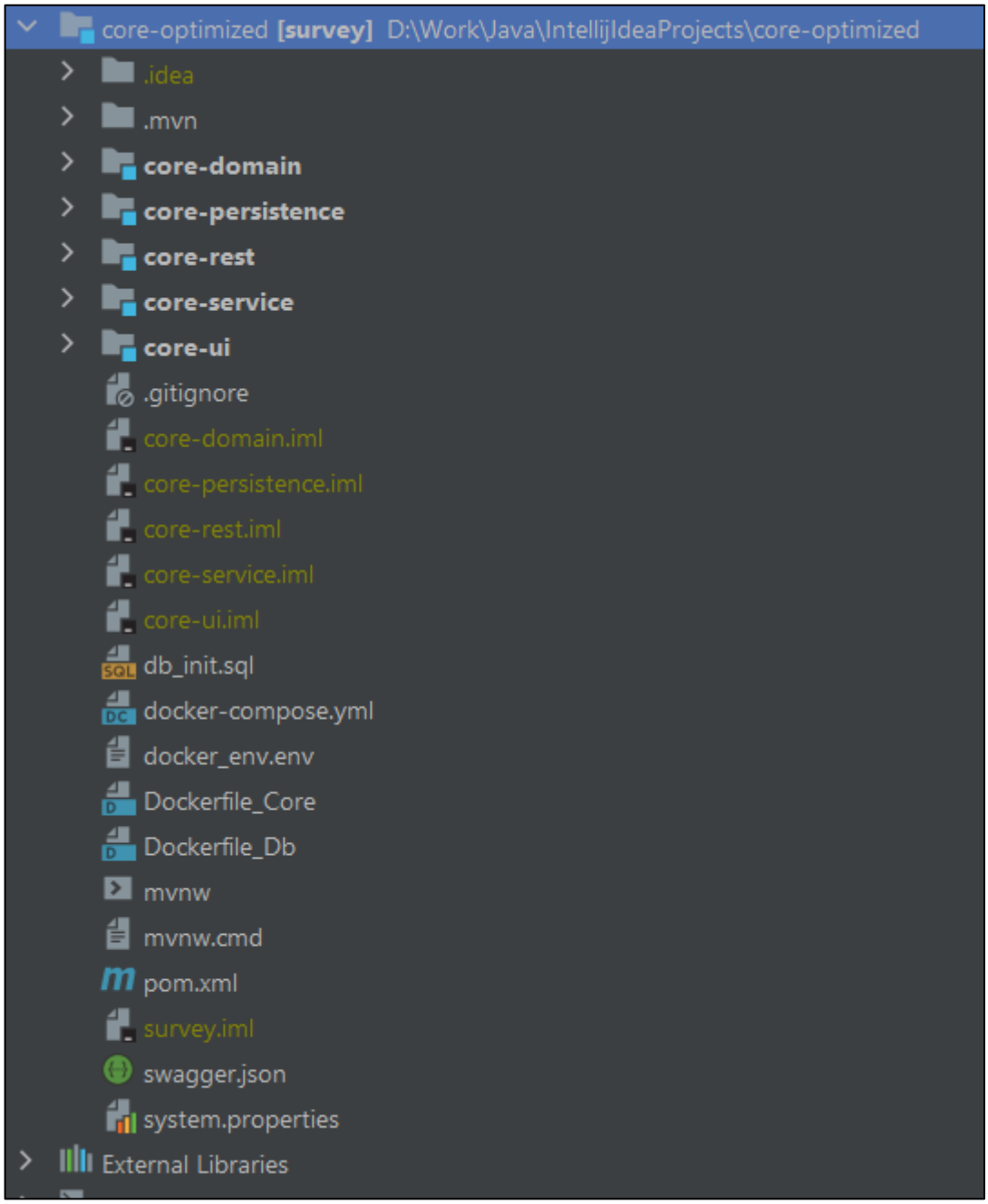

Рисунок 2.15 – Змінена багатомодульна структура (створено самостійно)

Виконаємо збірку проекту.

Зазначимо, що час збірки наразі буде дещо більшим, ніж раніше, оскільки багатомодульна архітектура вимагає більшої кількості ресурсів на процес збірки, що в рази більш цінне, ніж час збірки, однак, наразі розглянутий проект NURE Survey не буде мати таких проблем із масштабованістю, отже ми розглядаємо його в контексті прикладу.

Подивимось на логи збірки проекту (див. рис. 2.16).

```
[INFO] Reactor Summary for survey 0.3.7:
[INFO]
[INFO] survey ..................................... SUCCESS [  1.704 s]
[INFO] core-domain ................................ SUCCESS [  3.418 s]
[INFO] core-persistence ........................... SUCCESS [  1.515 s]
[INFO] core-service ............................... SUCCESS [  9.113 s]
[INFO] core-rest .................................. SUCCESS [ 36.991 s]
[INFO] core-ui .................................... SUCCESS [ 22.542 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  01:15 min
[INFO] Finished at: 2024-05-17T17:56:50+03:00
[INFO] ------------------------------------------------------------------------

Process finished with exit code 0
```

Рисунок 2.16 – Частина логів збірки оновленого проекту (створено самостійно)

Загальний час збірки становить 1 хвилина та 15 секунд. Збірка виконувалась на комп’ютері із жорстким диском (не SSD), процесором Intel Core I9 9900K, та оперативною пам’яттю DDR4 обсягом 32 гігабайти із частотою 3200 мегагерц, операційною системою Windows 10, становить 1 хвилина 15 секунд, включаючи запуск тестів.

Спробуємо оптимізувати витрачений час, використовуючи паралельну збірку та подивимось на результат.

Для цього поки що використаємо 4 віртуальних процесори (тобто ступінь паралелізму буде дорівнювати 4). Наступним кроком виконаємо команду системи збірки Apache Maven «mvn -T 4 clean install», що підчистить директорію збірки артефактів, після цього виконає їх збірку із використанням чотирьох програмних потоків.

Отже, майже кожен програмний модуль зможе збиратися паралельно, якщо модулі від яких він є залежний вже були зібрані раніше. Це повинне додати певного рівня прискорення для локальної збірки проекту. Це було би особливо відчутно у разі налаштування подібної збірки в крупних проектах.

```
[INFO] Reactor Summary for survey 0.3.7:
[INFO]
[INFO] survey ..................................... SUCCESS [  1.698 s]
[INFO] core-domain ................................ SUCCESS [  3.322 s]
[INFO] core-persistence ........................... SUCCESS [  1.525 s]
[INFO] core-service ............................... SUCCESS [  8.473 s]
[INFO] core-rest .................................. SUCCESS [ 41.150 s]
[INFO] core-ui .................................... SUCCESS [ 25.569 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  56.469 s (Wall Clock)
[INFO] Finished at: 2024-05-17T18:02:14+03:00
[INFO] ------------------------------------------------------------------------

Process finished with exit code 0
```

Рисунок 2.17 – Результат паралельної збірки із чотирма ядрами (створено самостійно)

Як можна побачити з рисунку, швидкість дещо вище, ми отримали результат у приблизно 56.5 секунд, що швидше за попередній на приблизно 25%.

Спробуємо зробити те ж саме, але при цьому буде більша доступна кількість віртуальних процесорів в два рази, тобто в даному випадку рівень паралелізму буде дорівнювати 8.

Теоретично це повинне надати прискорення, однак вже не стільки відчутне. Відповідно до закону Амдала, ступінь прискорення знижається експоненціально із підвищенням ступеню паралелізму.

Виправимо та зробимо зміни до команди системи збірки Apache Maven, змінивши опцію для рівня паралелізму на 8. Команда буде виглядати наступним образом «mvn -T 8 clean install».

Рівень прискорення процесу збірки системою Apache Maven від використання рівня паралелізму 8 замість рівня паралелізму 4 можна побачити на рисунку 2.18.

```
[INFO] Reactor Summary for survey 0.3.7:
[INFO]
[INFO] survey ..................................... SUCCESS [  1.810 s]
[INFO] core-domain ................................ SUCCESS [  3.358 s]
[INFO] core-persistence ........................... SUCCESS [  1.525 s]
[INFO] core-service ............................... SUCCESS [  8.352 s]
[INFO] core-rest .................................. SUCCESS [ 39.803 s]
[INFO] core-ui .................................... SUCCESS [ 25.904 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  55.145 s (Wall Clock)
[INFO] Finished at: 2024-05-17T18:06:25+03:00
[INFO] ------------------------------------------------------------------------

Process finished with exit code 0
```

Рисунок 2.18 – Результат паралельної збірки із рівнем паралелізму 8 (створено самостійно)

Як можна побачити, результат майже такий самий, що і був, тобто більшість максимально можливого прискорення від паралелізму вже досягнуто. Різниця становить лише 1.3 секунди, що є дуже незначним в даному випадку у порівнянні із використаними ресурсами локального середовища, такими як навантаження на центральний процесор комп'ютера, резервування оперативної пам'яті та навантаження на жорсткий диск (HDD) комп'ютера.

Можна зробити висновок, що використання рівня паралелізму 4 системою збірки Apache Maven в даному випадку є достатнім для оптимізації вже багатомодульної бекенд-частини проекту NURE Survey. Також можна допустити, що чим більше модулів в проекті, тим більше вплив паралелізму буде відчутний для часу збірки проекту. Знов повертаючись до закону Амдала, в подібному випадку ми будемо мати більше «порцій роботи».

Іншим засобом прискорення швидкості збірки є використання інкрементної збірки. В Apache Maven інкрементна збірка використовується автоматично, якщо не використовувати команду «clean». Спробуємо використати інкрементну збірку.

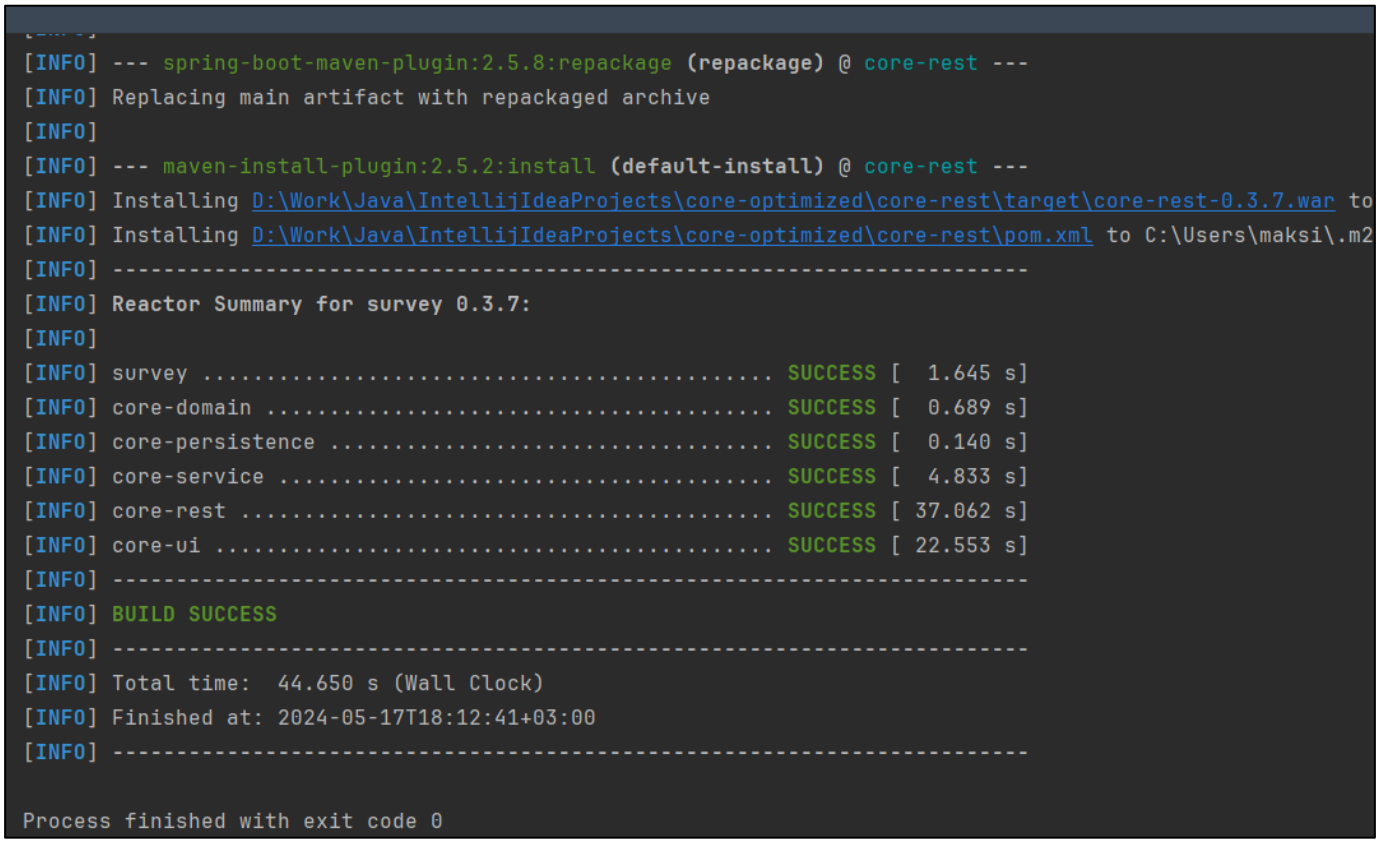

Рисунок 2.19 – Результат інкрементної збірки (створено самостійно)

З рисунку можемо побачити як інкрементна збірка разом з паралелізмом рівня 4 надали значне прискорення у приблизно 40%. Наразі збірка займає 44.65 с у порівнянні із попередньою, зображеною на рисунку 2.16, що мала результат у одну хвилину та 15 секунд.

Наступним кроком оновимо версії всіх залежностей на найбільш актуальні, для усунення можливих вразливостей проекту та можливої оптимізації їх роботи яку розробники залежностей могли надати із новими версіями.

Код проекту від початку роботи вже мав тести(модульні та інтеграційні), що були створені за часи роботи безпосередньо над NURE Survey

Виконання тестів відбувається автоматично під час виконання команди «mvn clean install». Завдяки цьому, розробник відразу може побачити, в якому функціоналі присутня помилка.

Для більш детального прикладу, запустимо тести окремо за допомоги команди «mvn test». (див. рис. 2.20).

```
[INFO] Results:
[INFO]
[INFO] Tests run: 37, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  36.469 s
[INFO] Finished at: 2024-05-20T15:23:13+03:00
[INFO] ------------------------------------------------------------------------
```

Рисунок 2.20 – Результат виконання тестів (створено самостійно)

В продакшн проектах кількість тестів може буде значно більшою, а необхідне покриття коду сягати як 75% так і в окремих випадках 100%.

Щодо якості коду, в даному випадку використаємо плагін-аналізатор для середовища розробки IntelliJ IDEA – SonarLint.

Як було розглянуто раніше, SonarLint це інструмент для статичного аналізу коду, що допомагає розробникам виявляти та виправляти помилки, баги, вразливості та проблеми з якістю коду на ранніх етапах розробки. Він інтегрується з IDE (інтегрованими середовищами розробки) та забезпечує миттєвий зворотний зв'язок про якість коду прямо під час написання.

Завдяки аналізу коду в реальному часі розробник має можливість побачити потенційні проблеми відразу прямо в редакторі. В даному випадку, для проекту бекенд-частини NURE Survey використовувалось середовище розробки IntelliJ IDEA, яка є наразі одним з найбільш популярних середовищ розробки у світі Java. Середовище IntelliJ IDEA доволі зручно дозволяє встановлювати різноманітні плагіни, серед яких можна знайти й розглянутий раніше SonarLint. Для цього достатньо перейти у налаштування, обрати вкладу «плагіни» та знайти необхідний плагін. В даному контексті, використання SonarLint є безкоштовним для розробників.

Встановимо SonarLint для проекту (див. рис. 2.21).

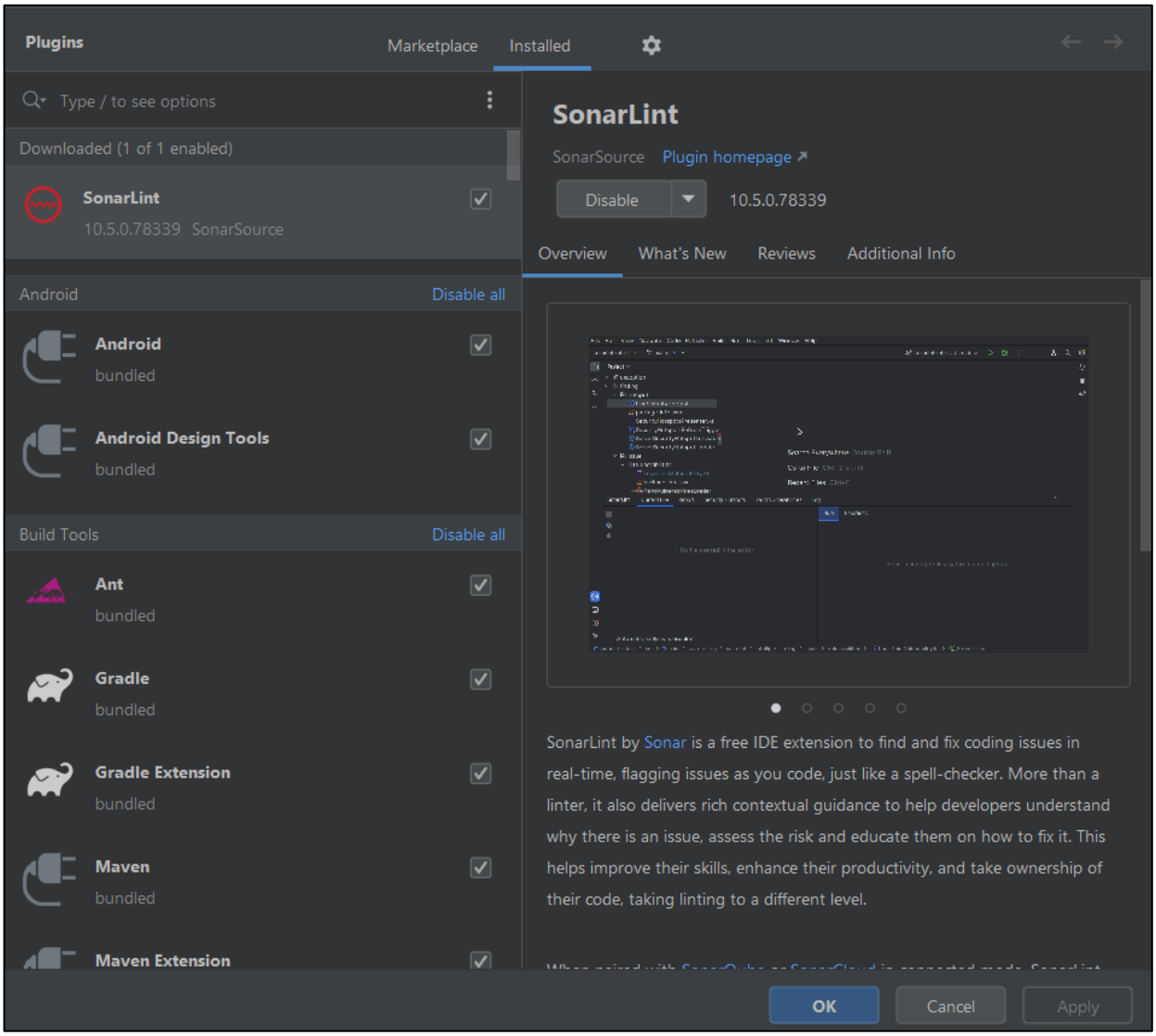

Рисунок 2.21 – Вікно встановленого плагіну SonarLint (створено самостійно)

Плагін має вікно з виявленими потенційними проблемами відкритого файлу. Також вікно містить підказки до можливих рішень й інші налаштування.

Для бекенд частини проекту NURE Survey використаємо налаштування плагіну SonarLint за замовчуванням. SonarLint використовує набір правил для Java, які охоплюють різні аспекти кодування, такі як стиль, можливі помилки, проблеми з продуктивністю, безпекою та підтримуваністю коду. Бібліотека стандартних правил є доволі великою та забезпечує покриття потреб розглянутого проекту, отже все залишаємо як є.

В вікні роботи із SonarLint розробник може власноруч запустити сканування якості коду програми, однак воно автоматизоване та відбувається після змін.

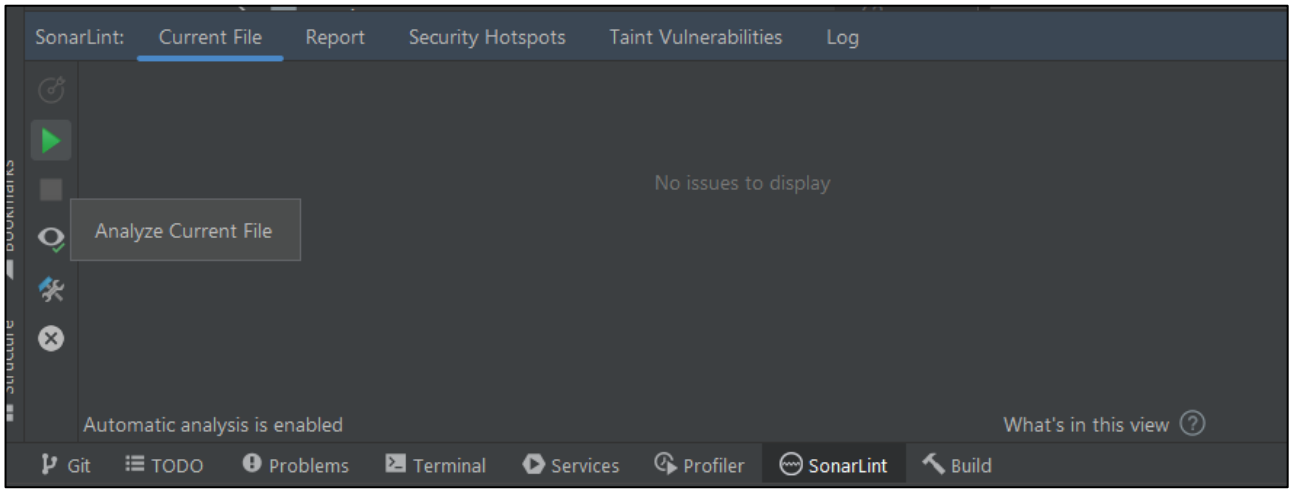

Рисунок 2.22 – Вікно роботи із плагіном SonarLint (створено самостійно)

Як було згадано в пункті про Docker, даний проект вже має файли для контейнеризації, проте після переходу на багатомодульну архітектуру, необхідно перенести відповідний Docker файл до кожного з модулів.

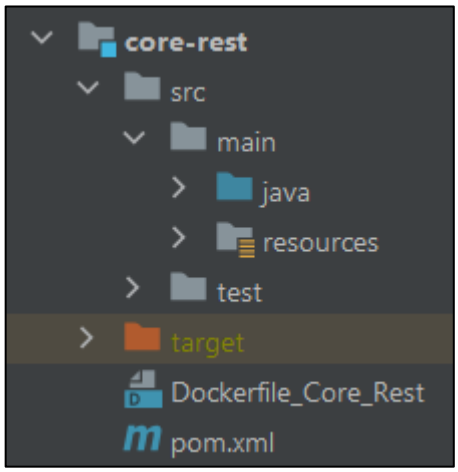

Рисунок 2.23 – Файл докер для REST модуля (створено самостійно)

Помістимо файл «Dockerfile_Core_Rest» який буде відповідати за конфігурацію власне артефакту створеного програмного модуля «core-rest». Це дозволить робити налаштування для контейнеризації окремо від налаштувань іншого артефакту «core-ui». Завдяки цьому також значно зменшується ризик неможливості впровадження змін в масштабованість програмних застосунків. Також можливе налаштування специфічних правил розгортання артефактів, наприклад розділення мережі під час комунікації із базою даних та ін.

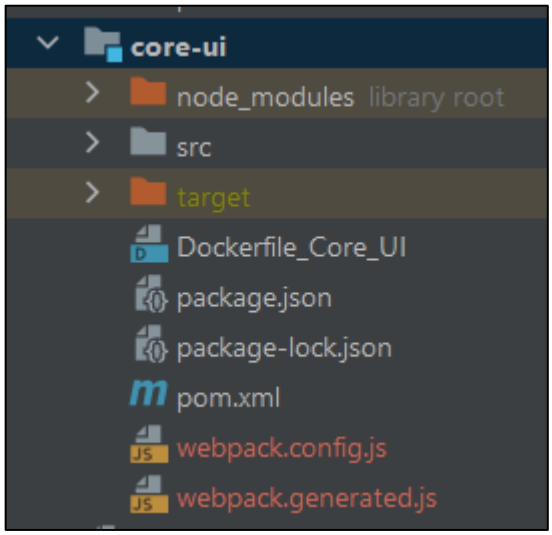

Рисунок 2.24 – Файл докер в UI модулі (створено самостійно)

```
core-ui:
  container_name: survey_core_ui_container
  build:
    context: .
    dockerfile: core-ui/Dockerfile_Core_UI
  image: survey-core-ui:latest
  env_file:
    - docker_env.env
  ports:
    - "8081:8081"
  networks:
    - postgres
```

Рисунок 2.25 – Змінена конфігурація для core-ui в docker-compose файлі (створено самостійно)

В зміненій конфігурації ми відокремлюємо ім’я для кожного з артефактів створених програмних модулів. Для артефакту UI контейнер буде мати ім’я «survey_core_ui_container», окремий контекст збірки, шлях до файлу Docker буде з доданим префіксом шляху «core-ui/» та образ контейнеру також матиме нове ім’я пов’язане з UI-частиною.

```yaml
core-rest:
  container_name: survey_core_rest_container
  build:
    context: .
    dockerfile: core-rest/Dockerfile_Core_Rest
  image: survey-core-rest:latest
  env_file:
    - docker_env.env
  ports:
    - "8081:8081"
  networks:
    - postgres
```

Рисунок 2.26 – Змінена конфігурація для core-rest в docker-compose файлі (створено самостійно)

Тепер, коли більшість оптимізації кроку оптимізованої імплементації виконана, ми можемо переходити до наступного кроку з оптимізованої доставки та інтеграції.

Для початку проведемо аналіз, які покращення можна було б зробити для проекту в рамках прикладу. Головним пунктом є впровадження відсутніх на момент оптимізації проекту інструментів для процесів постійної інтеграції та доставки та автоматизація процесів збірки та розгортання розглянутого багатомодульного проекту.

В цьому прикладі, для впровадження процесів постійної інтеграції та постійної доставки використаємо вже розглянутий раніше в попередніх пунктах інструмент Jenkins.

Створимо задачу для автоматизованої збірки проекту, що буде раз в певний час перевіряти репозиторій проекту, а саме гілку «master» на наявність змін. В випадку їх присутності, буде створено автоматичний запуск процесу збірки.

Дамо назву задачі «nure_survey.snapshot».

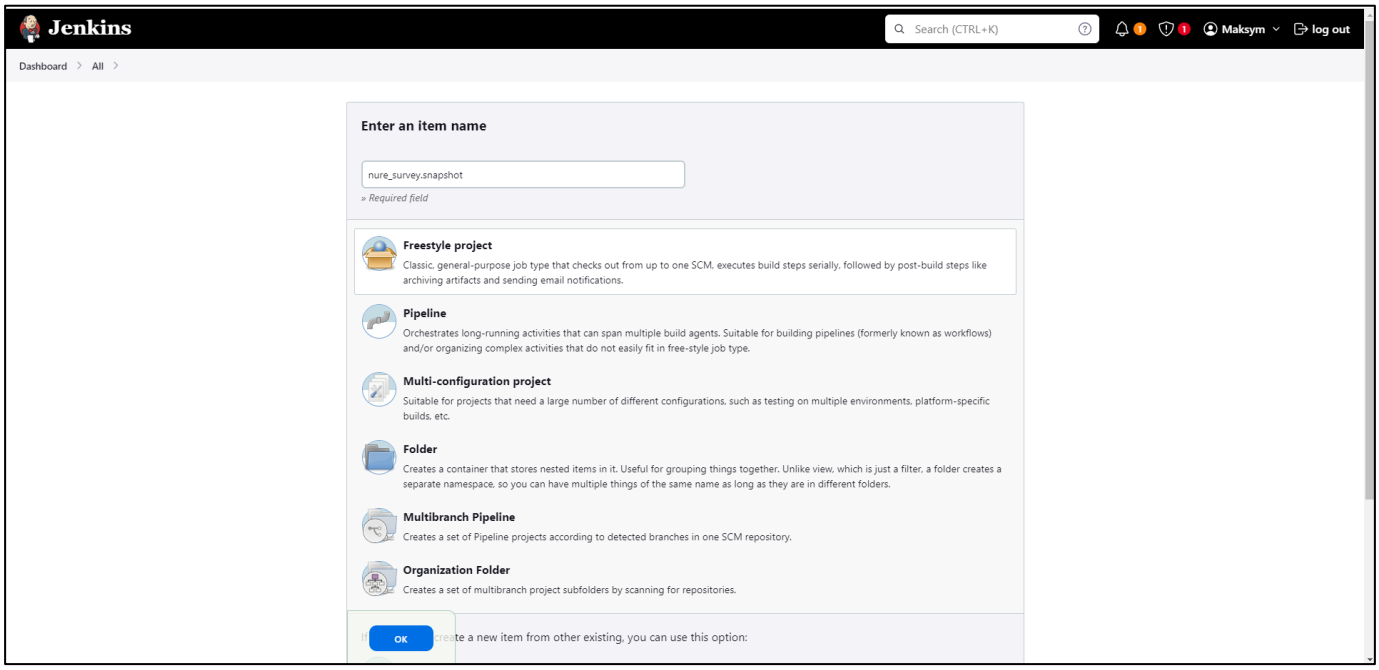

Рисунок 2.27 – Вибір типу задачі для «nure_survey.snapshot» (створено самостійно)

Далі додаємо дані репозиторію проекту в GitLab.

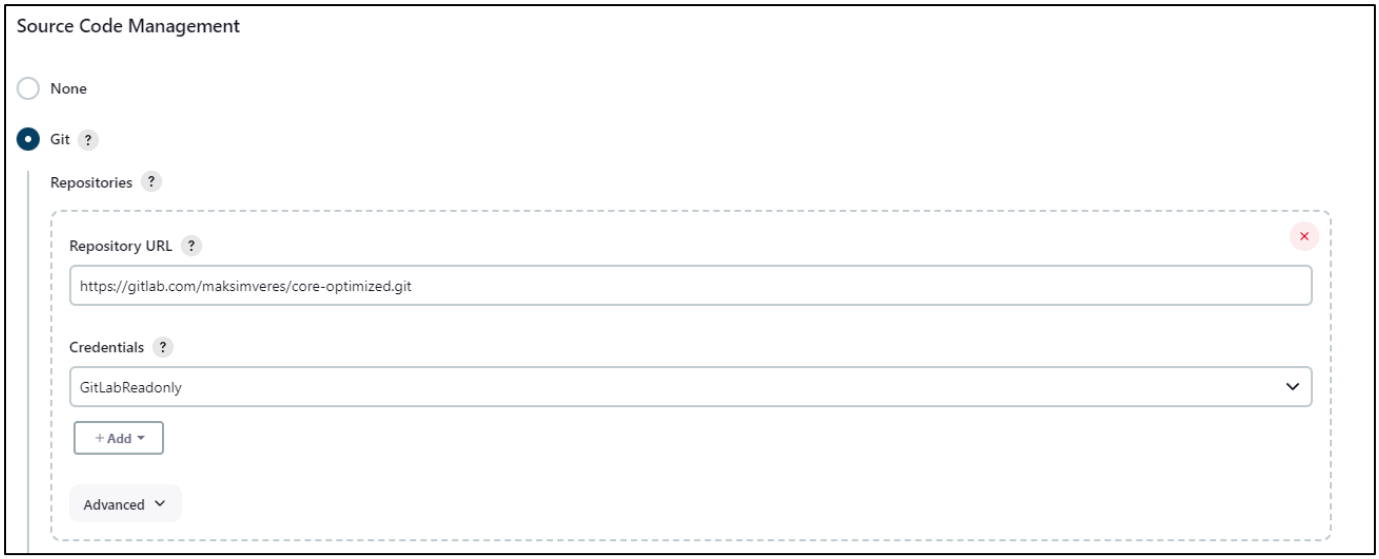

Рисунок 2.28 – Налаштування репозиторію (створено самостійно)

Наступним кроком налаштуємо дані для автоматизованого запуску задачі, коли в репозиторії відбуваються зміни. Сканування змін відбуватиметься кожну годину.

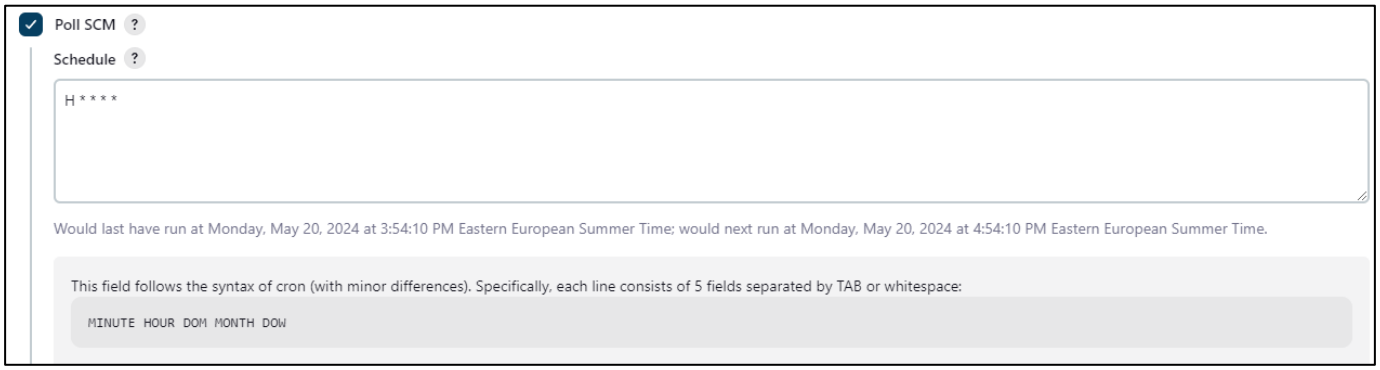

Рисунок 2.29 – Налаштування розкладу сканування змін в репозиторії (створено самостійно)

І останній крок – налаштування збірки.

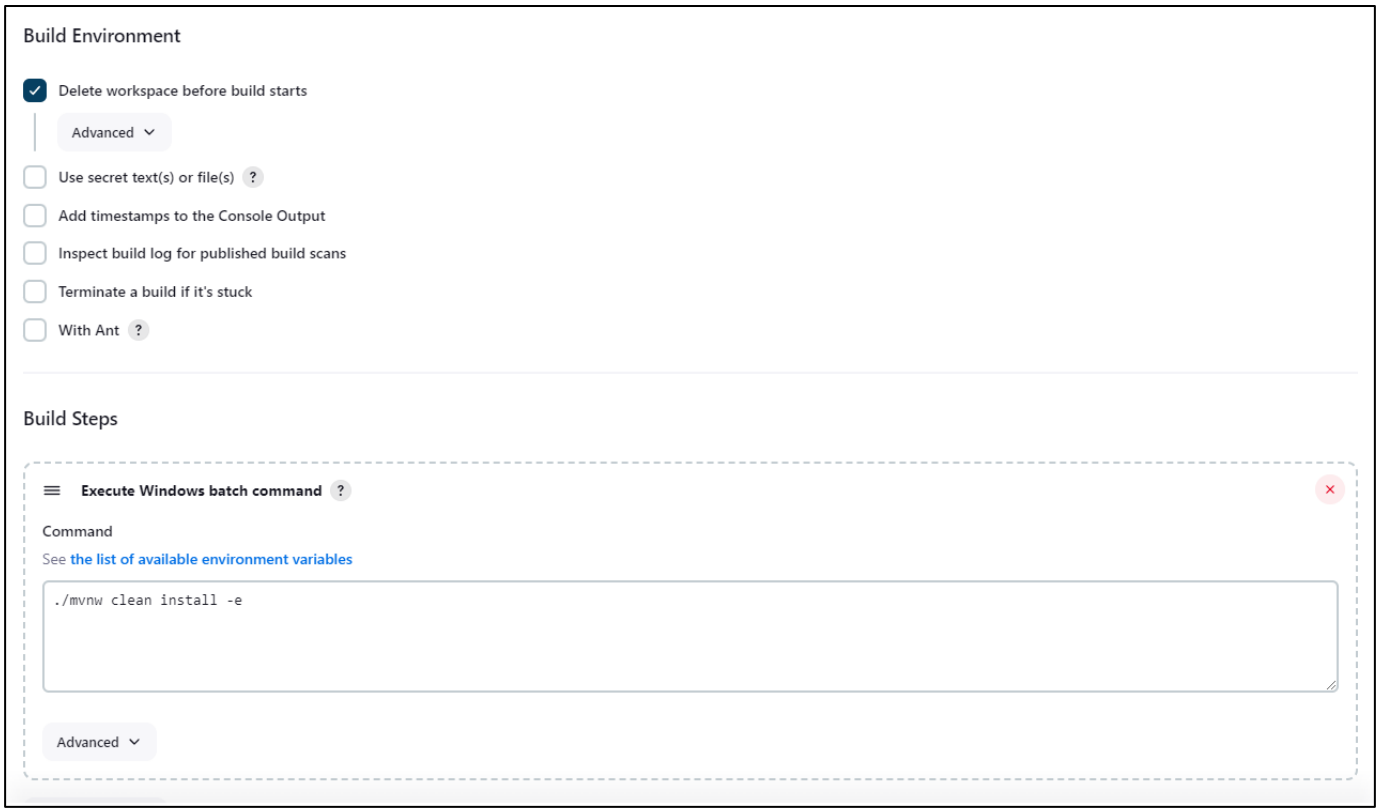

Рисунок 2.30 – Налаштування збірки (створено самостійно)

Ми маємо налаштування сканування репозиторію за правилом «H * * * *». Це означає, що сканування репозиторію на зміни буде відбуватись щогодинно. Також маємо налаштування видалення файлів із робочого простору до старту збірки і саме команду, яка використовує систему збірки Apache Maven «./mvnw clean install -e».

Виконаємо задачу та отримаємо результат (див .рис. 2.31).

```
[INFO] Reactor Summary for survey 0.3.8:
[INFO]
[INFO] survey ..................................... SUCCESS [  6.490 s]
[INFO] core-domain ................................ SUCCESS [  3.968 s]
[INFO] core-persistence ........................... SUCCESS [  1.645 s]
[INFO] core-service ............................... SUCCESS [  8.127 s]
[INFO] core-rest .................................. SUCCESS [ 47.744 s]
[INFO] core-ui .................................... SUCCESS [02:02 min]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  03:11 min
[INFO] Finished at: 2024-05-20T16:39:22+03:00
[INFO] ------------------------------------------------------------------------
Finished: SUCCESS
```

Рисунок 2.31 – Результат збірки в консолі Jenkins (створено самостійно)

Успішну збірку можна побачити в історії збірок даної задачі.

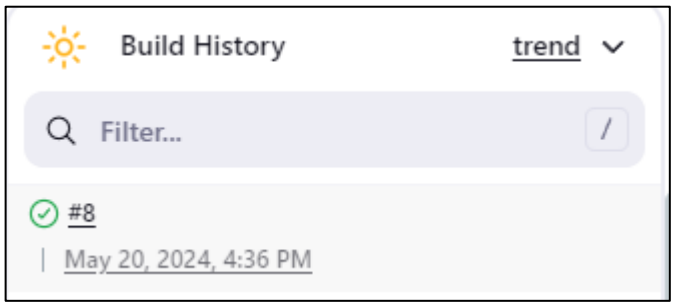

Рисунок 2.32 – Історія збірок (створено самостійно)

Наразі ми маємо створену задачу для створення зліпку збірки проекту, що дозволить перевіряти як успішно проект буде збиратися у чистому середовищі та автоматизовано показувати поточний статус коду проекту.

Рисунок 2.33 – Створена задача «nure_survey.snapshot» (створено самостійно)

Тепер можна створити задачу для створення артефакту проекту певної версії. Назвемо її «nure_survey.release». В результаті виконання цієї задачі, ми повинні отримати готові артефакти для розгортання у середовищі.

При створенні, репозиторій вказуємо такий самий, як і раніше. Також, відмінність від попередньої задачі у відсутності сканування змін в репозиторії та доданому архівуванні артефактів.

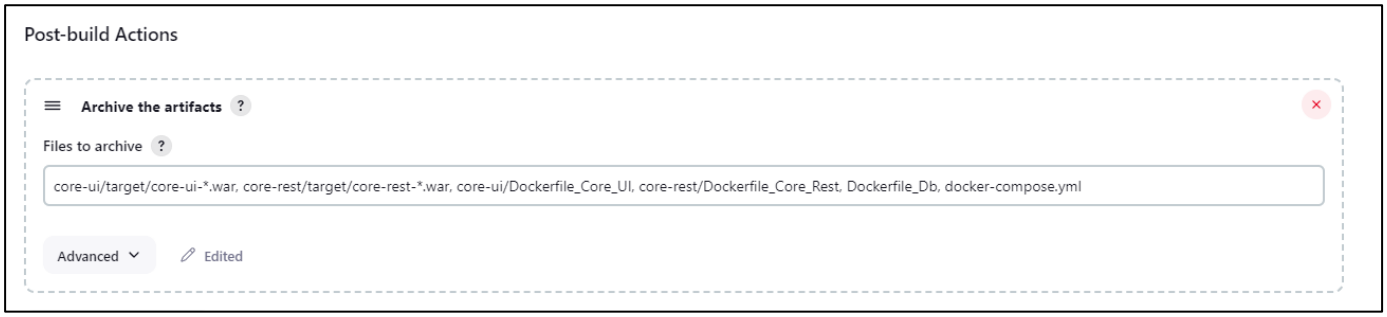

Рисунок 2.34 – Налаштування архівування артефактів (створено самостійно)

Запустимо нову задачу та подивимось, що архівування артефактів дійсно відбувається.

```
[INFO] Reactor Summary for survey 0.3.8:
[INFO]
[INFO] survey ..................................... SUCCESS [  1.456 s]
[INFO] core-domain ................................ SUCCESS [  3.362 s]
[INFO] core-persistence ........................... SUCCESS [  1.677 s]
[INFO] core-service ............................... SUCCESS [  8.191 s]
[INFO] core-rest .................................. SUCCESS [ 42.549 s]
[INFO] core-ui .................................... SUCCESS [01:10 min]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  02:08 min
[INFO] Finished at: 2024-05-20T16:59:50+03:00
[INFO] ------------------------------------------------------------------------
Archiving artifacts
Finished: SUCCESS
```

Рисунок 2.35 – Лог задачі «nure_survey.release» із архівуванням артефактів (створено самостійно)

Також можемо побачити, що артефакти відображаються в відповідному віконці.

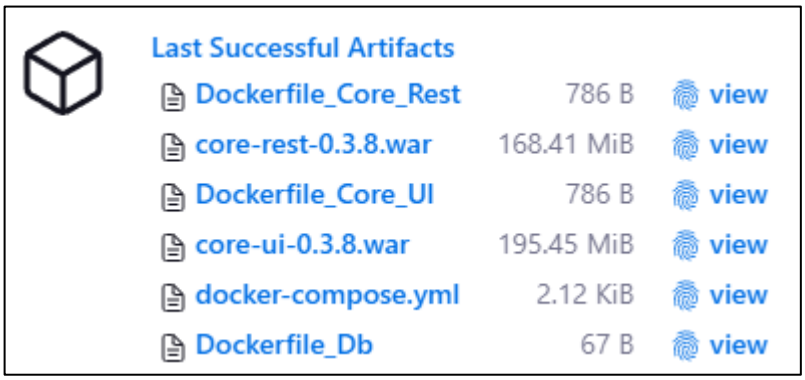

Рисунок 2.36 – Архівовані артефакти (створено самостійно)

Для імітації розгортання в певному середовищі для прикладу створимо задачу «nure_survey.deploy», яка візьме артефакт із задачі «nure_survey.release» та скопіює його у необхідну папку для подальшого запуску.

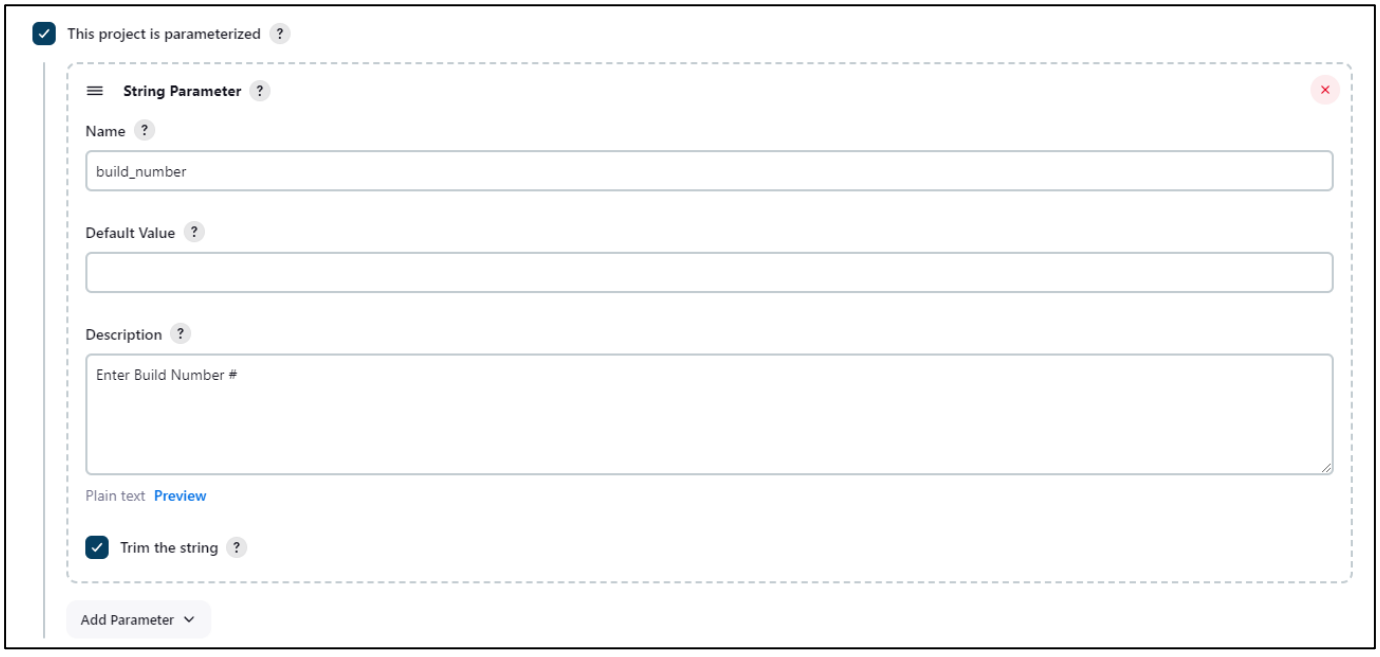

Рисунок 2.37 – Параметр номеру збірки (створено самостійно)

Для цього в задачу додамо параметр, який буде вказувати яку саме версію необхідно розгорнути. Цей параметр буде дорівнювати номеру збірки проекту із розглянутої раніше задачі «nure_survey.release».

Параметр буде використано далі у налаштуваннях процесу копіювання.

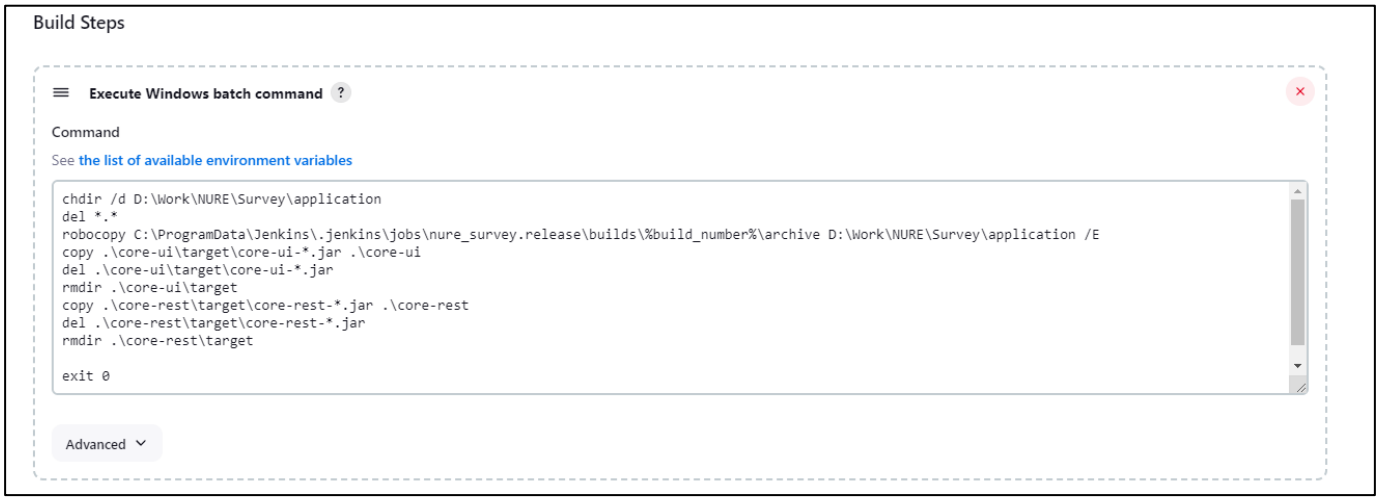

Рисунок 2.38 – Скрипт копіювання артефакту (створено самостійно)

І наступний крок – додавання скрипту на копіювання даних. Спочатку він очищує необхідну директорію, після копіює артефакти вказаної збірки версії.

Після розгортання отримаємо наступну структуру проекту в директорії розгортання (див. рис. 2.39).

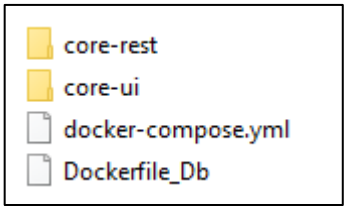

Рисунок 2.39 – Структура розгорнутого застосунку (створено самостійно)

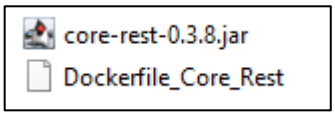

Рисунок 2.40 – Файли в core-rest (створено самостійно)

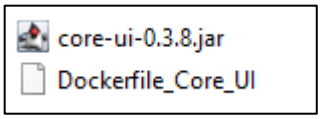

Рисунок 2.41 – Файли в core-ui (створено самостійно)

Можемо побачити, що скрипт копіювання працює як очікується та копіює артефакти до необхідних середовищ.

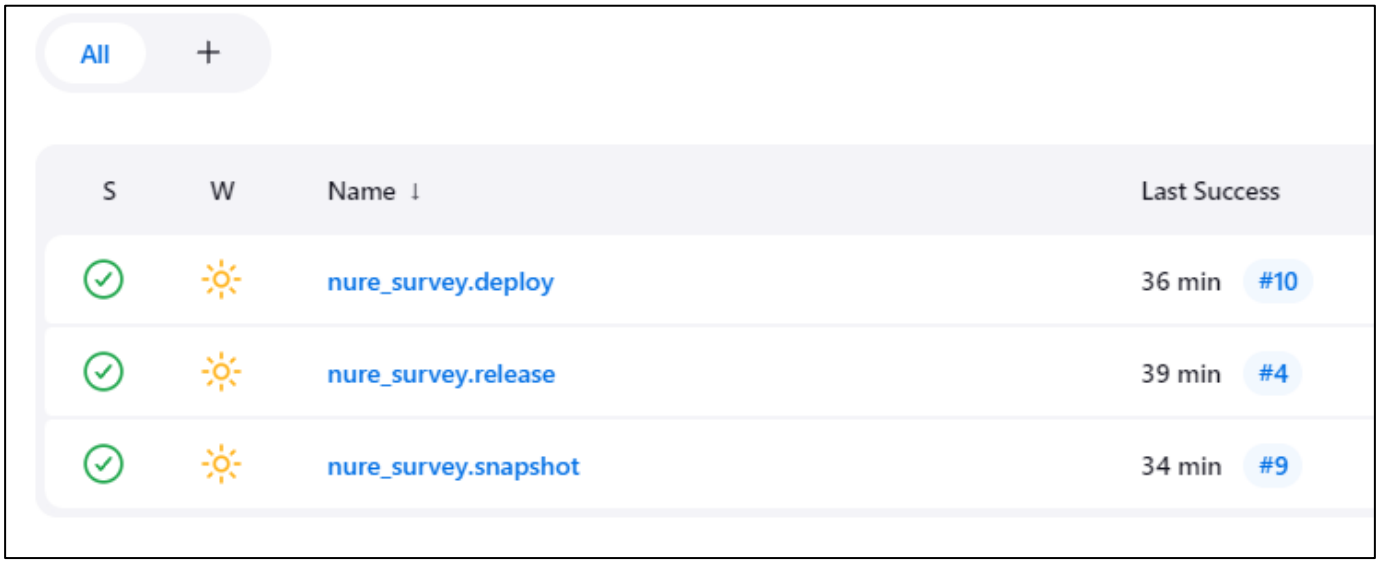

Рисунок 2.42 – Створені задачі в Jenkins (створено самостійно)

Далі в реальних кейсах, створюється процес безпечного розгорнення властивостей, включаючи секретну інформацію (наприклад, паролі, токени та ін.), необхідних для запуску застосунка. Та автоматизується із використанням скриптів запуск.

Створимо простий bat файл, в якому в явному вигляді наведемо властивості.

```
set DATASOURCE_URL=jdbc:postgresql://localhost:5432/nure_survey
set DATASOURCE_USER_NAME=[redacted]
set DATASOURCE_USER_PASSWORD=[redacted]
set GOOGLE_OAUTH2_CLIENT_ID=[redacted]
set GOOGLE_OAUTH2_CLIENT_SECRET=[redacted]
set MAIL_SMTP_USER_NAME=nuresurvey@gmail.com
set MAIL_SMTP_USER_PASSWORD=[redacted]
set MAIL_SURVEY_FEEDBACK_RECIPIENTS=maksimveres@gmail.com
set CLOUDINARY_CLOUD_NAME=[redacted]
set CLOUDINARY_API_KEY=[redacted]
set CLOUDINARY_API_SECRET=[redacted]

java -jar core-ui-0.3.8.jar
```

Рисунок 2.43 – Bat-файл для запуску застосунку в середовищі (створено самостійно)

Запустимо додаток та переконаємося, що все працює після розгортання.

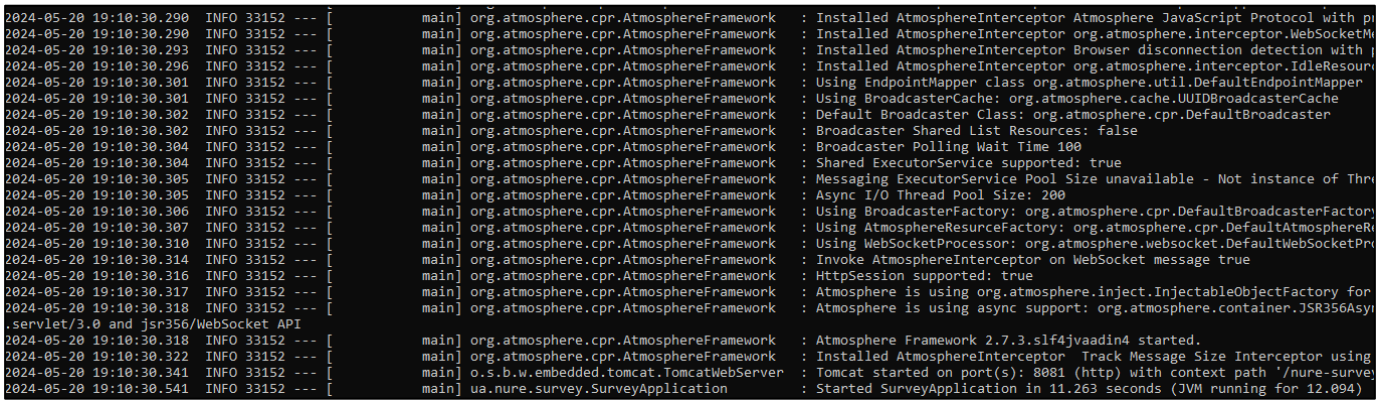

Рисунок 2.44 – Запущений UI через bat-файлі (створено самостійно)

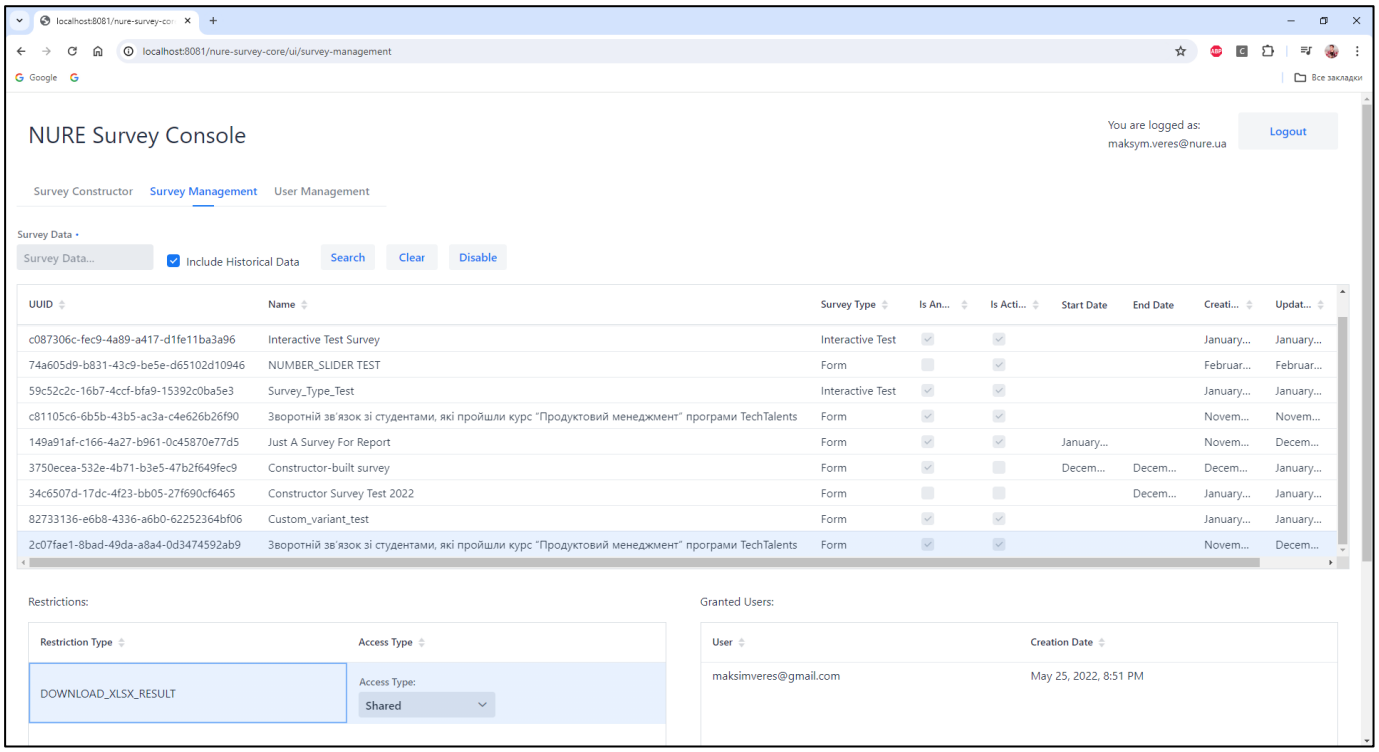

Рисунок 2.45 – Запущений застосунок (створено самостійно)

Як можна побачити з рисунку, застосунок працює на локальній адресі «localhost:8081/nure-survey-core».

Отже було проведено оптимізацію бекенд-частини проекту «NURE Survey» використовуючи наведені рекомендації. Вирішено архітектурні проблеми, проблеми із масштабованістю та створено приклад CI/CD процесу, прискорено процес збірки та підвищено надійність проекту завдяки автоматизованим задачам збірки.

ВИСНОВКИ

В ході роботи було визначено проблему щодо кожного з етапів створення та розгортання багатомодульних застосунків Java.

Було наведено основні теоретичні відомості щодо багатомодульної архітектури, принципів використання багатомодульності в Java. Визначено основні етапи життєвого циклу багатомодульного застосунку Java, на кожному з яких було визначено потенційні проблеми та наведено певні варіанти їх рішення або зменшення їх впливу.

Наведено засоби підвищення ефективності під час компіляції коду багатомодульного застосунку Java та щодо використання компіляторів, які підтримують інкрементну компіляцію. Надано рекомендації щодо інтеграції з інструментами перевірки якості коду, що робить процес створення багатомодульного застосунку більш ефективним ще на етапі розробки.

Для етапу збірки проекту, розглянуто основні системи збірки проектів, проаналізовано їх характеристики, та використовуючи лінійну адитивну згортку спрогнозовано найбільш ефективну з точки зору використання для багатомодульних застосунків – система збірки Gradle. Наведено приклади її ефективного використання та налаштування.

Для етапу після збірки проекту, розглянуто засоби моніторингу метрик збірки, засобів кешування артефактів та використання інструментів з контейнеризації.

Останнім етапом, для підвищення ефективності, що був проаналізований є етап розгортання багатомодульного застосунку. Під час аналізу були наведені практичні приклади із використання інструментів контейнеризації (Docker). Також проведено аналіз поточних хмарних платформ щодо ефективності їх використання для багатомодульних застосунків Java. За допомогою порівняння характеристик та лінійної адитивної згортки, було обрано найбільш ефективне – Oracle Cloud.

В якості кінцевого результату роботи, було розроблено систему загальних рекомендацій для ефективної розробки багатомодульних застосунків Java. Кожна

рекомендація не є обов’язковою до виконання та залежить від місця застосування. Однак загальне їх дотримування, може значно підвищити ефективність процесів як для числових характеристик (час збірки, споживання ресурсів та ін.), так і для якісних характеристик (якість структури проекту, якість та безпека коду та ін.). Частина наведеної системи рекомендацій була використана на практичному прикладі у вигляді удосконалення проекту NURE Survey та вирішення його основних проблем із архітектурою та процесом CI/CD.

Для більш детального аналізу, дана тема потребує подальшої роботи із створенням більшої кількості практичних прикладів. Також, розглядаючи наведені вище рекомендації, дана тема підводить до необхідності аналізу можливостей створення нового більш універсального та ефективного інструменту для CI/CD процесів із Java-застосунками або значного покращення існуючих рішень.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Guide to Android app modularization | Android Developers. Android Developers. URL: https://developer.android.com/topic/modularization (дата звернення: 15.05.2024).

2. Козир В. Побудова модульної архітектури проєкту на Android. Досвід Headway. Genesis. URL: https://www.gen.tech/post/modulna-arhitektura-android-dosvid-headway (дата звернення: 15.05.2024).

3. A Guide to Java 9 Modularity | Baeldung. Baeldung. URL: https://www.baeldung.com/java-9-modularity (дата звернення: 15.05.2024).

4. Joshua Block, Effective Java. 3rd Edition, Addison-Wesley Professional, 2018, ISBN: 978-0134685991.

5. David Farley, Jez Humble, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley Professional, 2010, ISBN-13: 978-0321601919.

6. Sam Newman, Building Microservices, O'Reilly, 2021, ISBN: 978-1492034025

7. Ant vs Maven vs Gradle | Baeldung. Baeldung. URL: https://www.baeldung.com/ant-maven-gradle (дата звернення: 15.05.2024).

8. Make- GNU Project - Free Software Foundation. The GNU Operating System and the Free Software Movement. URL: https://www.gnu.org/software/make (дата звернення: 15.05.2024).

9. Мартін Р. Чиста архітектура. – Київ: Фабула, 2019. – 368 с.

10. Chapter 6. The Java Virtual Machine Instruction Set. Oracle. URL: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html (дата звернення: 15.05.2024).

11. What is CI/CD? Red Hat. URL: https://www.redhat.com/en/topics/devops/what-is-ci-cd (дата звернення: 15.05.2024).

12. Incremental Compiler in Compiler Design. GeeksforGeeks. URL: https://www.geeksforgeeks.org/incremental-compiler-in-compiler-design (дата звернення: 15.05.2024).

13. ECJ - Eclipse Compiler for Java. The Coder Lounge. URL: https://blog.deepakazad.com/2010/05/ecj-eclipse-java-compiler.html (дата звернення: 15.05.2024).

14. Java Code Conventions. Oracle | Cloud Applications and Cloud Platform. URL: https://www.oracle.com/technetwork/java/codeconventions-150003.pdf (дата звернення: 15.05.2024).

15. IBM SDK, Java Technology Edition 8. IBM - United States. URL: https://www.ibm.com/docs/en/sdk-java-technology/8?topic=compiler-how-jit-optimizes-code (дата звернення: 15.05.2024).

16. Excluding warnings using @SuppressWarnings. IBM - United States. URL: https://www.ibm.com/docs/en/radfws/9.6.1?topic=code-excluding-warnings (дата звернення: 15.05.2024).

17. Generated (Java Platform SE 8). Oracle. URL: https://docs.oracle.com/javase/8/docs/api/javax/annotation/Generated.html (дата звернення: 15.05.2024).

18. Checkstyle 10.17.0. Checkstyle. URL: https://checkstyle.sourceforge.io/ (дата звернення: 15.05.2024).

19. SpotBugs. SpotBugs GitHub. URL: https://spotbugs.github.io (дата звернення: 15.05.2024).

20. PMD. PMD GitHub. URL: https://pmd.github.io (дата звернення: 15.05.2024).

21. Code Quality Tool & Secure Analysis with SonarQube. SonarQube. URL: https://www.sonarqube.org/ (дата звернення: 15.05.2024).

22. Catching Issues in the IDE with SonarLint. SonarCloud. URL: https://docs.sonarsource.com/sonarcloud/improving/sonarlint/ (дата звернення: 15.05.2024).

23. JUnit 5. JUnit. URL: https://junit.org/ (дата звернення: 15.05.2024).

24. TestNG Documentation. TestNG. URL: https://testng.org/ (дата звернення: 15.05.2024).

25. JaCoCo Java Code Coverage Library. JaCoCo. URL: https://www.jacoco.org/jacoco/ (дата звернення: 15.05.2024).

26. Error Prone. Error Prone. URL: https://errorprone.info/ – (дата звернення: 15.05.2024).

27. 10 причин, чому CI/CD важливі для DevOps. CloudFresh. URL: https://cloudfresh.com/ua/cloud-blog/10-prichin-chomu-ci-cd-vazhlivi-dlya-devops/ (дата звернення: 15.05.2024).

28. Behavior Driven Development Approach in the Modern Quality Control Process / O. Bezsmertnyi, Golian N. та ін. 2020 IEEE International Conference on Problems of Infocommunications. Science and Technology (PIC S&T), м. Kharkiv, Ukraine, 6–9 жовт. 2020 р. 2020. URL: https://doi.org/10.1109/picst51311.2020.9467891 (дата звернення: 15.05.2024).

29. Continuous Integration. Martin Fowler. URL: https://martinfowler.com/articles/continuousIntegration.html (Дата звернення 15.05.2024).

30. Apache Ivy. Apache Ant. URL: https://ant.apache.org/ivy/ (Дата звернення 15.05.2024).

31. Introduction to Maven Plugin Development. Apache Maven. URL: https://maven.apache.org/guides/introduction/introduction-to-plugins.html (Дата звернення 15.05.2024).

32. Leiba Y., Shirokopetleva M., Gruzdo I. RESEARCH ON METHODS OF DETERMINING CUSTOMER LOYALTY AND ASSESSING THEIR LEVEL OF SATISFACTION. Innovative Technologies and Scientific Solutions for Industries. 2023. № 2 (24). С. 104–117. URL: https://doi.org/10.30837/itssi.2023.24.104 (дата звернення: 07.06.2024).

33. Data Normalization in Data Mining. GeeksForGeeks. URL: https://www.geeksforgeeks.org/data-normalizationin-data-mining/ (Дата звернення 15.05.2024).

34. Shrink, obfuscate, and optimize your app. Developers Android. URL: https://developer.android.com/build/shrink-code (Дата звернення 15.05.2024).

35. Java Obfuscator and Android App Optimizer | ProGuard. Guardsquare, URL: https://www.guardsquare.com/proguard (Дата звернення 15.05.2024).

36. Build Cache. Gradle. URL: https://docs.gradle.org/current/userguide/build_cache.html (Дата звернення 15.05.2024).

37. CVE. CVE MITRE. URL: https://cve.mitre.org/ (Дата звернення 15.05.2024).

38. Docker. Docker. URL: https://www.docker.com/ (Дата звернення 15.05.2024).

39. Kubernetes. Kubernetes. URL: https://kubernetes.io/ (Дата звернення 15.05.2024).

40. What is AWS. Amazon Web Services. URL: https://aws.amazon.com/what-is-aws/?nc1=h_ls (Дата звернення 15.05.2024).

41. Microsoft Azure. Microsoft. URL: https://azure.microsoft.com/ (Дата звернення 15.05.2024).

42. IBM Cloud. IBM. URL: https://www.ibm.com/cloud (Дата звернення 15.05.2024).

43. Google Cloud. Google. URL: https://cloud.google.com/ (Дата звернення 15.05.2024).

44. Oracle Cloud. Oracle. URL: https://www.oracle.com/ua/ (Дата звернення 15.05.2024).