МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет кораблебудування імені адмірала Макарова

Навчально-науковий інститут комп'ютерних наук та управління проєктами

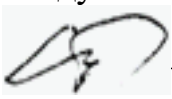
Кафедра програмного забезпечення автоматизованих систем

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма «Інженерія програмного забезпечення»

«Допущений до захисту»

Завідувач кафедри

 _ проф. Приходько С.Б.

«__» __ 2026 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти «бакалавр»

на тему: Розробка вебзастосунку для моніторингу та управління версійністю розгорнутих застосунків та їх розгортання за допомогою GitLab API та Gitlab CI

Виконала: студентка групи 4151з

Сук Є. А.

(підпис, ПІБ)

Керівник роботи:

канд. техн. наук, доцент

Макарова Л. М.

(підпис, ПІБ)

Миколаїв – 2026 р.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет кораблебудування імені адмірала Макарова
Навчально-науковий інститут комп'ютерних наук та управління проєктами
Кафедра програмного забезпечення автоматизованих систем
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма «Інженерія програмного забезпечення»

«ЗАТВЕРДЖУЮ»
Гарант освітньої програми
_____ доц. Макарова Л.М.
(підпис)
«_07_» ___04___ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ
на здобуття ступеня вищої освіти «бакалавр»

Студентці Сук Євгенії Андріївні._____
(Прізвище, ім'я, по батькові)

1. Тема роботи: Розробка вебзастосунку для моніторингу версійності розгорнутих застосунків та їх розгортання за допомогою GitLab API та Gitlab CI____
Керівник роботи: доцент кафедри ПЗАС, к.т.н., доцент Макарова Л. М. _____
Затверджені наказом ректора №275-уч від 07.04.2026 р.
2. Термін подання роботи: 01.06.2026 р._____
3. Вихідні дані по роботі: _____
_____
4. Перелік питань, що належать до розробки (найменування розділів) _____
Титульний аркуш; Завдання на кваліфікаційну роботу; Анотація (українською та англійською мовами); Зміст; Перелік умовних позначень, символів, одиниць та термінів (при необхідності); Вступ; Аналіз практичної задачі інженерії програмного забезпечення; Проєкт програмного забезпечення; Результати розробки програмного забезпечення; Розділ з охорони праці; Висновки; Список використаних джерел; Додатки (технічне завдання, текст програми, опис програми, інструкція користувача, програма та методика випробувань програмного забезпечення).
5. Перелік презентаційних матеріалів_____
1 Титульний слайд, 2 Мета і завдання кваліфікаційної роботи, 3 Аналоги ПЗ, 4 Аналіз вимог (діаграма варіантів використання), 5 Архітектура ПЗ (діаграма пакетів), 6 Ескізний проєкт (діаграма діяльності «Процес моніторингу застосунку»), 7 Ескізний проєкт (діаграма діяльності «Запуск розгортання через GitLab API»), 8 Ескізний проєкт (діаграма станів та переходів), 9 Ескізний проєкт (структура бази даних MongoDB), 10 Ескізний проєкт (прототипи інформаційної панелі та модальних вікон), 11 Технічний проєкт (діаграма класів), 12 Технічний проєкт (діаграма послідовності «Отримання даних про розгорнутий проєкт»), 13 Технічний проєкт (діаграма послідовності «Додавання нового проєкту»), 14 Робочий проєкт (вибір засобів розробки), 15 Результати розробки (головна сторінка застосунку), 16 Результати розробки (робоча область обраної групи), 17 Результати розробки (модальне вікно проєкту), 18 Висновки, 19 Заключний слайд.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
4	Гурець Н.В., ст. викладач		

7. Дата видачі завдання 07.04.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

Номер	Назва етапів роботи	Терміни виконання	Примітка
1.	Підготовка розділу ВСТУП	30.03.2026	ПП*
2.	Аналіз практичної задачі інженерії ПЗ	06.04.2026	ПП*
3.	Аналіз вимог до ПЗ	13.04.2026	ПП*
4.	Розробка архітектури ПЗ	20.04.2026	
5.	Розробка проекту ПЗ	04.05.2026	
6.	Реалізація та тестування ПЗ	11.05.2026	
7.	Підготовка розділу Результати розробки ПЗ	15.05.2026	
8.	Підготовка розділу з охорони праці	18.05.2026	ПП*
9.	Підготовка розділу ВИСНОВКИ	22.05.2026	
10.	Оформлення списку використаних джерел та додатків	25.05.2026	
11.	Подання на кафедру ПЗАС тексту остаточного варіанту роботи, підписаного її керівником, у роздрукованому та електронному форматі разом із заявами щодо самостійності виконання роботи та ідентичності друкованої та електронної версій роботи (Додатки 1 і 2 «Порядку здійснення заходів з перевірки робіт на наявність текстових збігів/ідентичності/схожості із використанням програмно-технічних засобів», який введений в дію наказом ректора НУК за №20 від 20.01.2020 р.)	01.06.2026	не пізніше, ніж за 14 діб до захисту (згідно п.4.1 зазначеного Порядку
12.	Підготовка презентації та доповіді	05.06.2026	
13.	Попередній захист роботи на засіданні кафедри ПЗАС	08.06.2026	
14.	Подання на кафедру ПЗАС електронних версій наступних документів у форматі pdf: кваліфікаційної роботи; файлу-опису кваліфікаційної роботи (згідно Додатку до наказу ректора НУК за №287-уч від 19.05.2020 р.); презентації доповіді, а також Авторського договору з додатком	15.06.2026	

* - за результатами переддипломної практики (ПП), яка була з 02.02.2026 до 22.03.2026 р.

Студентка _____ Сук Є.А.
(підпис) (ПІБ)

Керівник роботи _____ Макарова Л.М.
(підпис) (ПІБ)

АНОТАЦІЯ

Кваліфікаційна робота присвячена розв’язанню практичної задачі інженерії програмного забезпечення, що характеризується комплексністю та невизначеністю умов - розробці вебзастосунку для моніторингу версійності розгорнутих застосунків та їх розгортання за допомогою GitLab API та GitLab CI. У роботі проаналізовано предметну область, розглянуто існуючі програмні продукти та виконано постановку задачі на створення інтеграційної платформи. У першому розділі проведено аналіз проблеми фрагментації даних про стан розгортання, розглянуто існуючі рішення та обґрунтовано необхідність створення універсальної системи для подолання їхніх обмежень.

У другому розділі описано процес проектування та розробки вебзастосунку: проаналізовано вимоги до реалізації автоматичного збору даних через HTTP-запити, побудовано модель системи управління конфігураціями для цільових версій застосунків та механізм взаємодії з GitLab API для дистанційного запуску процесів оновлення.

У третьому розділі наведено результати реалізації вебзастосунку, який забезпечує єдину точку доступу до інформації про CI/CD процеси та фактичні параметри розгорнутих сервісів. Розробка дозволяє не лише виявляти невідповідність версій, а й оперативно ініціювати розгортання нових версій безпосередньо з панелі моніторингу.

Кваліфікаційна робота викладена на 108 сторінках друкованого тексту та містить 18 рисунків, 3 таблиці, список використаних джерел з 26 найменувань та 5 додатків.

ABSTRACT

The qualification paper is dedicated to solving a practical software engineering problem characterized by complexity and uncertainty of conditions: the development of a web application for monitoring the versioning of deployed applications and their deployment using the GitLab API and GitLab CI. The paper analyzes the subject area, reviews existing software products, and outlines the task of creating an integration platform.

The first chapter analyzes the problem of deployment status data fragmentation, reviews existing solutions, and justifies the need to create a universal solution to overcome their limitations.

The second chapter describes the process of designing and developing the web application: it analyzes the requirements for implementing automatic data collection via HTTP requests, builds a configuration management model for target application versions, and outlines the interaction mechanism with the GitLab API for the remote initiation of update processes.

The third chapter presents the results of implementing the web application, which provides a single point of access to information about CI/CD processes and the actual parameters of deployed services. The developed software makes it possible not only to detect version mismatches but also to promptly initiate the deployment of new versions directly from the monitoring dashboard.

The qualification paper consists of 108 pages of printed text and contains 18 figures, 3 tables, a list of 26 references, and 5 appendices.

ЗМІСТ

ВСТУП

У сучасному цифровому світі, де швидкість розгортання та безперервність роботи програмного забезпечення знаходяться в центрі уваги, ефективне управління життєвим циклом застосунків стає ключовим аспектом продуктивності та надійності IT-інфраструктури. Розвиток технологій хмарних обчислень, мікросервісної архітектури та зростання потреб користувачів у стабільних програмних продуктах створює підґрунтя для розробки спеціалізованих вебзастосунків, спрямованих на покращення моніторингу та управління розгортанням [1].

Актуальність теми. В умовах підтримки життєвого циклу множини взаємопов'язаних програмних проєктів виникає проблема фрагментації даних про стан розгортання. Інформація про версію коду у репозиторії та версію фактично працюючого сервісу у середовищі виконання знаходиться у різних системах, що значно ускладнює контроль за синхронним оновленням компонентів. Наявні аналоги не вирішують проблему повністю: спеціалізовані інструменти залежать від конкретних технологій , а популярні платформи спостереження працюють переважно в режимі «читання» і не мають функцій для дистанційного запуску оновлень через GitLab API. Це зумовлює необхідність розробки інтеграційної платформи, яка б поєднала моніторинг і управління в одному вебзастосунку.

Метою кваліфікаційної роботи є проектування та реалізація вебзастосунку – інтеграційної платформи, яка забезпечує єдину точку доступу до інформації про стан CI/CD процесів та фактичні параметри розгорнутих застосунків. Результатом розробки стане інструмент для швидкого виявлення застарілих сервісів та їх оперативного оновлення з єдиного інтерфейсу.

Для досягнення поставленої мети передбачено виконання таких завдань:

1. Аналіз предметної області та практичної проблематики управління розгортанням застосунків.

2. Дослідження існуючих програмних рішень, оцінка їхніх можливостей і обмежень.

3. Формулювання функціональних та нефункціональних вимог до інтеграційної платформи.

4. Проектування архітектури вебзастосунку для збору даних через HTTP-запити та взаємодії з GitLab API.

5. Реалізація механізму автоматичного збору даних про реальні версії працюючих сервісів у реальному часі.

6. Розробка програмної системи управління конфігураціями для контролю цільових версій проектів.

7. Створення функціоналу для дистанційного запуску CI/CD пайплайнів через GitLab API.

8. Опис принципів роботи розробленого програмного продукту та особливостей його використання.

Предметом дослідження є методи та інструменти моніторингу стану та управління розгортанням програмних засобів у розподілених середовищах із використанням GitLab API та Gitlab CI.

З огляду на вищезазначене, розробка спеціалізованого вебзастосунку для моніторингу версійності та управління розгортанням є надзвичайно актуальною. Запропоноване рішення дозволить не лише бачити застарілі версії, а й оперативно їх оновлювати, що забезпечує значну перевагу над існуючими готовими рішеннями. Дана кваліфікаційна робота спрямована на вирішення зазначених проблем шляхом розробки програмного продукту, який відповідає сучасним стандартам автоматизації.

1. АНАЛІЗ ПРАКТИЧНОЇ ЗАДАЧІ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ПОСТАНОВКА ЗАДАЧІ НА РОЗРОБКУ ВЕБЗАСТОСУНКУ ДЛЯ МОНІТОРИНГУ ВЕРСІЙНОСТІ РОЗГОРНУТИХ ЗАСТОСУНКІВ ТА ЇХ РОЗГОРТАННЯ ЗА ДОПОМОГОЮ GITLAB API ТА GITLAB CI

1.1 Аналіз практичної задачі інженерії програмного забезпечення для автоматизації контролю розгортання

Сучасна інженерія програмного забезпечення характеризується стрімким переходом від монолітних архітектур до розподілених систем. Такий підхід дозволяє розробляти, тестувати та розгортати окремі програми незалежно одна від одної, що суттєво підвищує гнучкість команд розробки. Відповідно, стандартом індустрії стали практики безперервної інтеграції та безперервного розгортання, які автоматизують процеси збірки та доставки коду [2]. Однак, разом із перевагами, такий підхід приносить нові виклики в управлінні інфраструктурою.

Дана робота присвячена розробці вебзастосунку для моніторингу версійності та управління розгортанням. В умовах підтримки життєвого циклу множини різноманітних програм виникає критична проблема фрагментації даних про стан розгортання.

Проблематика розробки даного програмного комплексу пов’язана з тим, що інформація про версію коду в репозиторії та версію фактично працюючої програми знаходиться у різних системах. На практиці це означає, що розробники бачать успішне виконання CI/CD пайплайну у системі контролю версій, але не завжди можуть бути впевнені, що саме ця версія наразі успішно функціонує у цільовому середовищі виконання [3]. Це ускладнює контроль за оновленнями та збільшує час на пошук і усунення несправностей.

Аналізуючи існуюче програмне забезпечення, можна відзначити, що наявні спеціалізовані інструменти є технологічно залежними. Хоча вони ефективні у своєму середовищі, вони не здатні універсально збирати дані з програм,

розроблених на інших мовах чи фреймворках. З іншого боку, популярні платформи моніторингу, такі як Grafana та Prometheus, функціонують переважно в режимі «читання». Вони відображають поточний стан метрик, але не мають функцій для збереження цільових версій і не дозволяють дистанційно запускати процеси оновлення через GitLab API.

Це зумовлює необхідність створення інтеграційної платформи, яка б поєднувала моніторинг і активне управління в одному застосунку та забезпечувала єдину точку доступу до інформації про стан CI/CD процесів. Програмний комплекс повинен містити власну підсистему управління конфігураціями, яка дозволить зберігати та редагувати інформацію про цільові версії. Важливою вимогою до програмної системи є інформаційна та програмна сумісність: вебзастосунок має підтримувати збір даних з різнорідних програм незалежно від їхнього технологічного стеку.

Складність цієї задачі обумовлюється необхідністю інтеграції та узгодженої роботи в межах одного вебзастосунку трьох різних механізмів:

1. Реалізація автоматичного збору телеметричних даних про реальні версії працюючих програм шляхом налаштування прямих HTTP-запитів.
2. Забезпечення збереження еталонних версій для автоматизованого порівняння та перевірки на невідповідність.
3. Створення механізму для дистанційного запуску процесів оновлення через взаємодію з GitLab API, що дозволяє закрити цикл від виявлення проблеми до її усунення.

Таким чином, розробка даного вебзастосунку надасть інструменти для швидкого виявлення застарілих програм та їх оперативного оновлення безпосередньо з єдиної панелі моніторингу, мінімізуючи ручну роботу інженерів.

1.2 Аналіз програм-аналогів

На сучасному ринку існує низка інструментів для управління інфраструктурою, проте вони часто орієнтовані на загальний ринок і не

враховують специфічні потреби конкретних команд розробки. Нижче наведено аналіз основних аналогів.

1.2.1 GitLab CI/CD Dashboard

Є одним із найбільш розповсюджених та базових інструментів для автоматизації процесів розробки, який постачається «з коробки» у складі платформи GitLab [4]. Він відіграє ключову роль в управлінні життєвим циклом коду, надаючи інженерам єдиний центр керування конвеєрами постачання.

Основні можливості:

- Детальна візуалізація стану пайплайнів: Система забезпечує графічне відображення кожного етапу конвеєра безперервної інтеграції та розгортання (наприклад, збірка, тестування, реліз). Це дозволяє командам розробників у режимі реального часу відстежувати прогрес виконання завдань та швидко ідентифікувати етапи, на яких виникають збої, помилки або затримки;
- Збереження вичерпної історії збірок: Платформа автоматично архівує результати всіх попередніх виконань пайплайнів, включаючи детальні консольні логи. Це створює надійну базу для аудиту, ретроспективного аналізу помилок та відстеження стабільності коду на різних етапах його життєвого циклу.

Переваги:

- Глибока інтеграція з репозиторієм: Інструмент вбудований безпосередньо в GitLab, тому його не потрібно довго та складно налаштовувати. Розробники бачать результати перевірки коду в тій самій системі, де він зберігається та проходить через CI/CD процеси.

Недоліки:

- Відсутність інструментів для прямого моніторингу версій працюючих сервісів через HTTP: Платформа фіксує факт успішного завершення процесу розгортання у своєму середовищі, проте вона не здійснює активного опитування кінцевих цільових серверів. Іншими словами, GitLab CI/CD показує ту версію, яка була відправлена на сервер, але не може гарантувати, що саме ця

версія зараз фактично працює та коректно відповідає на мережеві запити. Це створює «сліпу зону» в інфраструктурі, яку неможливо закрити без додаткових інструментів;

- Неможливість гнучкого групування проєктів поза межами GitLab-груп: Архітектура платформи має жорстку ієрархічну структуру. Якщо складний програмний продукт складається з десятків мікросервісів, які з організаційних причин рознесені по різних просторах імен або групах у GitLab, адміністратору немає можливості об’єднати їх на одній інформаційній панелі для візуального моніторингу їхнього загального стану.

1.2.2 Spring Boot Admin

Є вузькоспеціалізованим рішенням, яке набуло значної популярності серед команд розробки, що використовують мову програмування Java [5]. Фактично, він працює як зручна графічна панель, що збирає технічні дані з програм та виводить їх у зручному для читання форматі.

Основні можливості:

- Глибокий моніторинг стану застосунків: Система збирає та наочно відображає ключові технічні показники програми в режимі реального часу. Адміністратор може бачити статус «здоров’я» мікросервісу, використання оперативної пам’яті, навантаження на процесор та стан підключення до бази даних без необхідності використовувати консольні команди;
- Управління логами та конфігураціями: Інструмент дозволяє зручно переглядати системні повідомлення прямо у вікні браузера. Більше того, він дає змогу «на льоту» змінювати рівень деталізації цих логів, не перезапускаючи при цьому саму програму.

Переваги:

- Швидка інтеграція та зручність для Java-команд: Оскільки інструмент створений спеціально для екосистеми Spring, його підключення до існуючого Java-проєкту займає мінімум часу. Він надає розробникам інтуїтивно

зрозумілий інтерфейс, який працює практично «з коробки» і не вимагає написання додаткового коду для налаштування базового моніторингу.

Недоліки:

– Жорстка залежність від конкретного фреймворку: Інструмент розроблений виключно для програм, побудованих на базі Spring Boot, і не здатний стати універсальною точкою контролю для мікросервісів, написаних на інших мовах. Крім того, він функціонує виключно як інструмент спостереження і не надає можливостей для активного управління процесами розгортання.

1.2.3 Grafana

Є одним із найпопулярніших інструментів для побудови інформаційних панелей та візуалізації телеметрії у світі відкритого вихідного коду [6]. Ця платформа дозволяє об’єднувати дані з різних джерел в єдиному місці, створюючи наочну картину стану всієї системи.

Основні можливості:

– Візуалізація метрик та моніторинг: Основною функцією системи є побудова графіків і панелей на основі даних, зібраних із систем спостереження.

Переваги:

– Потужна графічна аналітика: Інструмент пропонує надзвичайно гнучкі можливості для налаштування зовнішнього вигляду панелей. Користувачі можуть створювати унікальні дашборди, які ідеально відповідають специфічним потребам їхнього моніторингу, відстежуючи найрізноманітніші показники.

Недоліки:

– Надмірна складність конфігурації: Використання цієї платформи для вирішення такого тривіального завдання, як просте порівняння номерів версій розгорнутих застосунків, є неефективним. Її налаштування вимагає значного часу та зусиль, що робить її занадто громіздкою для простих потреб;

– Відсутність функціоналу для управління розгортанням: Програма функціонує виключно в режимі спостереження. Вона не має вбудованих механізмів для активного дистанційного запуску процесів розгортання чи

оновлення сервісів через взаємодію з GitLab API, що не дозволяє замкнути цикл від виявлення проблеми до її усунення.

Наведений ґрунтовний аналіз існуючих на ринку продуктів яскраво підкреслює необхідність розробки індивідуального, спеціалізованого рішення. Жоден із розглянутих інструментів не здатен самостійно закрити весь спектр потреб в контексті поставленої задачі. Саме тому створення нової платформи, яка дозволить гармонійно поєднати моніторинг стану сервісів та управління їх розгортанням у зручному та єдиному веб-інтерфейсі, є технічно обґрунтованим та актуальним кроком.

1.3 Постановка задачі на розробку вебзастосунку для моніторингу версійності розгорнутих застосунків та їх розгортання за допомогою GitLab API та Gitlab CI

На основі проведеного аналізу предметної області та виявлених обмежень існуючих програмних рішень, виникає необхідність створення інтеграційної платформи, яка б поєднувала моніторинг і активне управління в одному застосунку. Головною задачею є розробка спеціалізованого вебзастосунку, що забезпечить єдину точку доступу до інформації про стан CI/CD процесів та фактичні параметри розгорнутих застосунків. Такий інструмент дозволить швидко виявляти застарілі сервіси та оперативно ініціювати їх оновлення безпосередньо з інформаційної панелі, мінімізуючи ручну роботу інженерів.

У рамках виконання роботи необхідно спроєктувати та реалізувати три основні компоненти вебзастосунку. До них належать Backend-сервіси для обробки бізнес-логіки управління проєктами, конфігураціями та логічними групами, модуль HTTP-клієнта для забезпечення безперервного збору телеметричної інформації про версії розгорнутих застосунків у реальному часі, а також клієнтська частина у вигляді веб-інтерфейсу для візуалізації стану сервісів та зручної інтерактивної взаємодії користувача з GitLab API.

Базовий набір функцій розроблюваного вебзастосунку повинен охоплювати повний цикл управління інфраструктурою. Зокрема, вебзастосунок

має забезпечувати управління логічними середовищами і картками проєктів із реєстрацією їхніх цільових веб-адрес. Окрім цього, передбачається автоматичний моніторинг технічних параметрів шляхом регулярного опитування сервісів для отримання таких метаданих, як версія Java, версія Spring Boot та поточна версія збірки самого застосунку. Для ефективного управління розгортанням необхідна тісна інтеграція з GitLab, що дозволить відображати доступні гілки репозиторію та дистанційно запускати CI/CD пайплайни безпосередньо з інтерфейсу картки проєкту.

Щодо специфікації виконуваних функцій, застосунок повинен вміти отримувати детальну інформацію про стан проєкту за його ідентифікатором шляхом виконання HTTP GET запиту за актуальною адресою розгорнутого сервісу, формуючи на виході об’єкт із даними про середовище. Управління ієрархією та структурою груп вимагає суворої валідації вхідних параметрів та їх збереження у нереляційній базі даних за допомогою відповідного рівня доступу. Також архітектура має забезпечувати безшовне оновлення конфігураційних параметрів: здійснювати пошук записів у базі, оновлювати поля та миттєво відображати ці зміни на стороні клієнта.

Для забезпечення стабільної роботи вебзастосунку необхідно врахувати мінімальні вимоги до апаратного та програмного забезпечення. Серверна частина розробляється мовою Java з використанням фреймворку Spring Boot та бази даних MongoDB, що вимагає для свого розгортання сервера з щонайменше двоядерним процесором, 4 ГБ оперативної пам'яті, від 20 ГБ вільного дискового простору та встановленим середовищем виконання Java. Клієнтська частина є односторінковим застосунком, побудованим на базі React, TypeScript та Redux Toolkit. Вона є кросплатформною і потребує на боці кінцевого користувача лише наявності сучасного веб-браузера.

2. ПРОЄКТУВАННЯ ВЕБЗАСТОСУНКУ ДЛЯ МОНІТОРИНГУ ВЕРСІЙНОСТІ РОЗГОРНУТИХ ЗАСТОСУНКІВ ТА ЇХ РОЗГОРТАННЯ ЗА ДОПОМОГОЮ GITLAB API ТА GITLAB CI

2.1 Аналіз вимог до вебзастосунку для моніторингу версійності розгорнутих застосунків та їх розгортання за допомогою GitLab API та Gitlab CI

2.1.1. Побудова моделі варіантів використання вебзастосунку

Діаграма варіантів використання – це ілюстрація функціональних можливостей програмного продукту з погляду її користувачів, де демонструється, які дії можуть бути виконані користувачами та вебзастосунком в реакції на конкретні події або запити [7].

Діаграма варіантів використання вебзастосунку, що проєктується, представлена на рис. 2.1.

Основні варіанти використання, що зображені на діаграмі для вебзастосунку моніторингу версійності та управління розгортанням, включають управління логічними групами середовищ та проєктами в них.

Користувачі можуть переглядати загальний список груп, а також створювати нові групи для логічного об’єднання мікросервісів, редагувати їх назви або видаляти існуючі.

Після вибору певної групи користувач може додавати до неї нові проєкти, вказуючи їхні актуальні веб-адреси, редагувати параметри цих проєктів або видаляти їх зі списку.

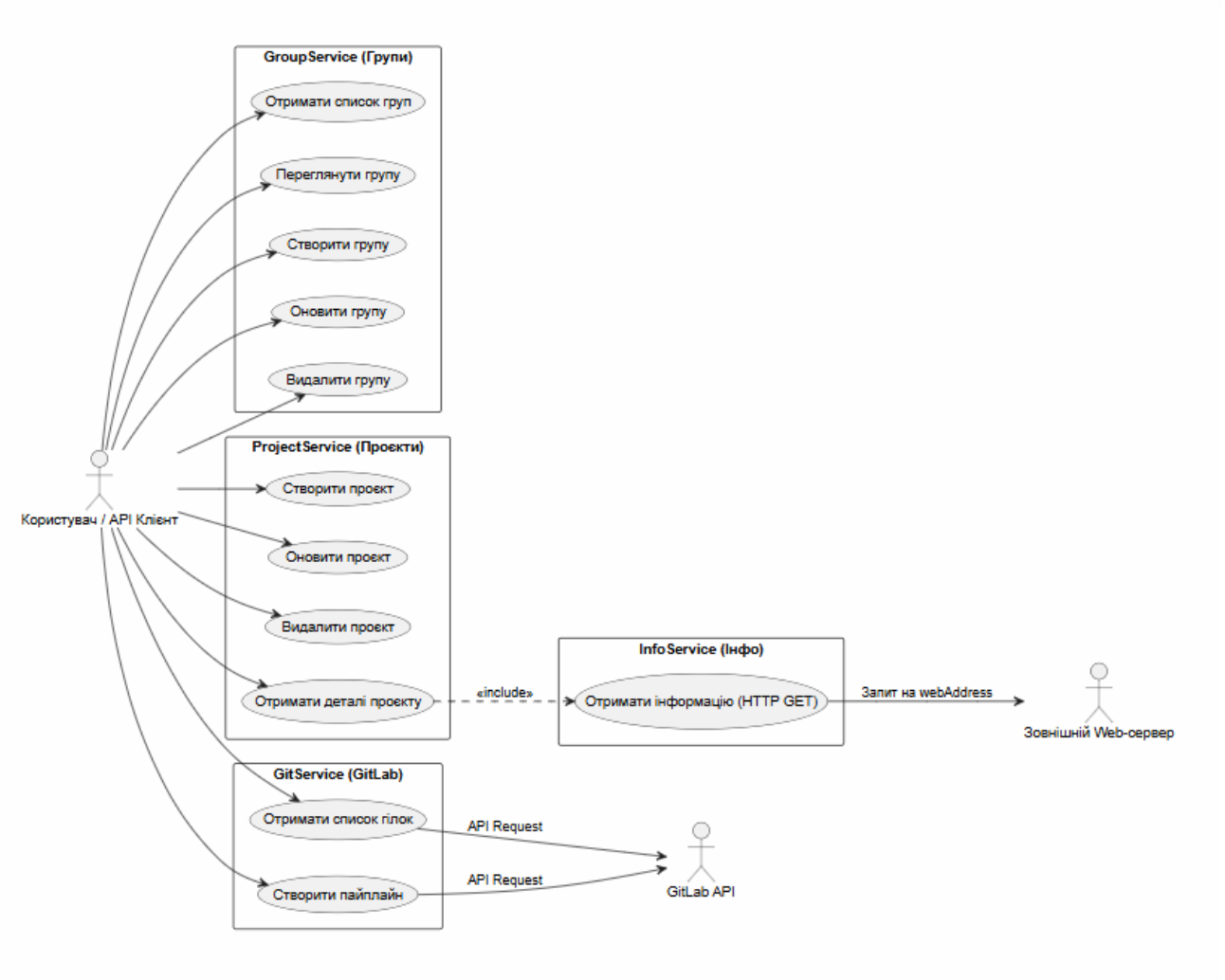


Рисунок 2.1 – Діаграма варіантів використання вебзастосунку для моніторингу версійності розгорнутих застосунків та їх розгортання

Після вибору конкретного проєкту користувачі можуть здійснювати моніторинг його стану, а також безпосередньо ініціювати процес розгортання нової версії за допомогою інтеграції з GitLab API.

2.1.2. Специфікація варіантів використання вебзастосунку

Для детального опису функціональних можливостей системи, представлених на діаграмі варіантів використання, було розроблено специфікації для найбільш критичних прецедентів. Специфікації формалізують взаємодію користувача з програмним продуктом та визначають очікувану поведінку вебзастосунку.

Нижче наведено специфікації для основних функціональних модулів: додавання проєкту, моніторингу телеметрії та ініціалізації розгортання.

Таблиця 2.1 – Специфікація прецеденту «Додавання нового проєкту»

Поле	Опис
Назва	Додавання нового проєкту до групи середовищ
Актор	Користувач (адміністратор)
Передумови	Користувач авторизований; вибрана логічна група для розміщення проєкту
Основний сценарій	1. Користувач натискає кнопку створення проєкту. 2. Вебзастосунок відкриває модальне вікно для введення даних. 3. Користувач вводить назву, URL, Gitlab ID та посилання на репозиторій. 4. Програма валідує вхідні дані. 5. Вебзастосунок зберігає проєкт у базі даних MongoDB та оновлює інтерфейс.
Альтернативний сценарій	Якщо валідація не пройдена, програма підсвічує помилкові поля та просить виправити дані.
Постумови	Проєкт відображається у робочій області групи.

Таблиця 2.2 – Специфікація прецеденту «Моніторинг стану проєкту»

Поле	Опис
Назва	Отримання інформації про стан розгорнутого застосунку
Актор	Користувач
Передумови	Проєкт зареєстрований у вебзастосунку, налаштована коректна веб-адреса для опитування
Основний сценарій	1. Користувач відкриває робочу область групи. 2. Вебзастосунок ініціює асинхронний HTTP GET-запит до віддаленого мікросервісу. 3. Віддалений сервіс повертає телеметричну інформацію у форматі JSON. 4. Застосунок виконує парсинг даних та витягує актуальні версії технологій (Java, Spring Boot). 5. Дані відображаються на відповідній картці проєкту.
Альтернативний сценарій	Якщо віддалений мікросервіс недоступний, програма перехоплює помилку мережі та відображає на картці статус збою з відповідним повідомленням.
Постумови	Актуальні технічні параметри сервісу відображені в інтерфейсі для аналізу.

Таблиця 2.3 – Специфікація прецеденту «Запуск розгортання (CI/CD)»

Поле	Опис
Назва	Дистанційний запуск CI/CD пайплайну
Актор	Користувач
Передумови	У налаштуваннях профілю встановлено дійсний Personal Access Token для доступу до GitLab
Основний сценарій	1. Користувач натискає кнопку ініціалізації розгортання на картці проєкту. 2. Вебзастосунок звертається до GitLab API для отримання переліку доступних гілок репозиторію. 3. Користувач вибирає гілку та ціль (Goal) у модальному вікні. 4. Користувач натискає «ЗАПУСТИТИ». 5. Вебзастосунок відправляє POST-запит із заголовком авторизації до GitLab API. 6. Програма інформує користувача про успішний старт процесу.
Альтернативний сценарій	Якщо термін дії токена закінчився або він недійсний, застосунок виводить сповіщення про помилку автентифікації доступу до GitLab.
Постумови	GitLab ініціює виконання конвеєра розгортання, статус якого можна відстежити у GitLab CI.

Ці специфікації демонструють чіткий алгоритм роботи ключових функцій програмного продукту.

2.2 Архітектура вебзастосунку для моніторингу версійності розгорнутих застосунків та їх розгортання за допомогою GitLab API та Gitlab CI

Розроблений вебзастосунок базується на клієнт-серверній архітектурі, що забезпечує чітке розділення відповідальності між інтерфейсом користувача та серверною частиною.

Серверна частина вебзастосунку реалізує багатошарову (layered) архітектуру, що є стандартом для побудови надійних застосунків [8]. Вона складається з наступних логічних рівнів:

- Рівень маршрутизації (Controller Layer): приймає HTTP-запити від клієнта та повертає відповіді.

– Сервісний рівень (Service Layer): містить бізнес-логіку застосунку, керує моніторингом та інтеграцією з GitLab API.

– Рівень доступу до даних (Repository Layer): відповідає за взаємодію з базою даних (MongoDB) та виконання CRUD-операцій.

Для візуалізації логічної архітектури розробленого вебзастосунку та взаємозв'язків між його основними компонентами побудовано діаграму пакетів (Рисунок 2.2).

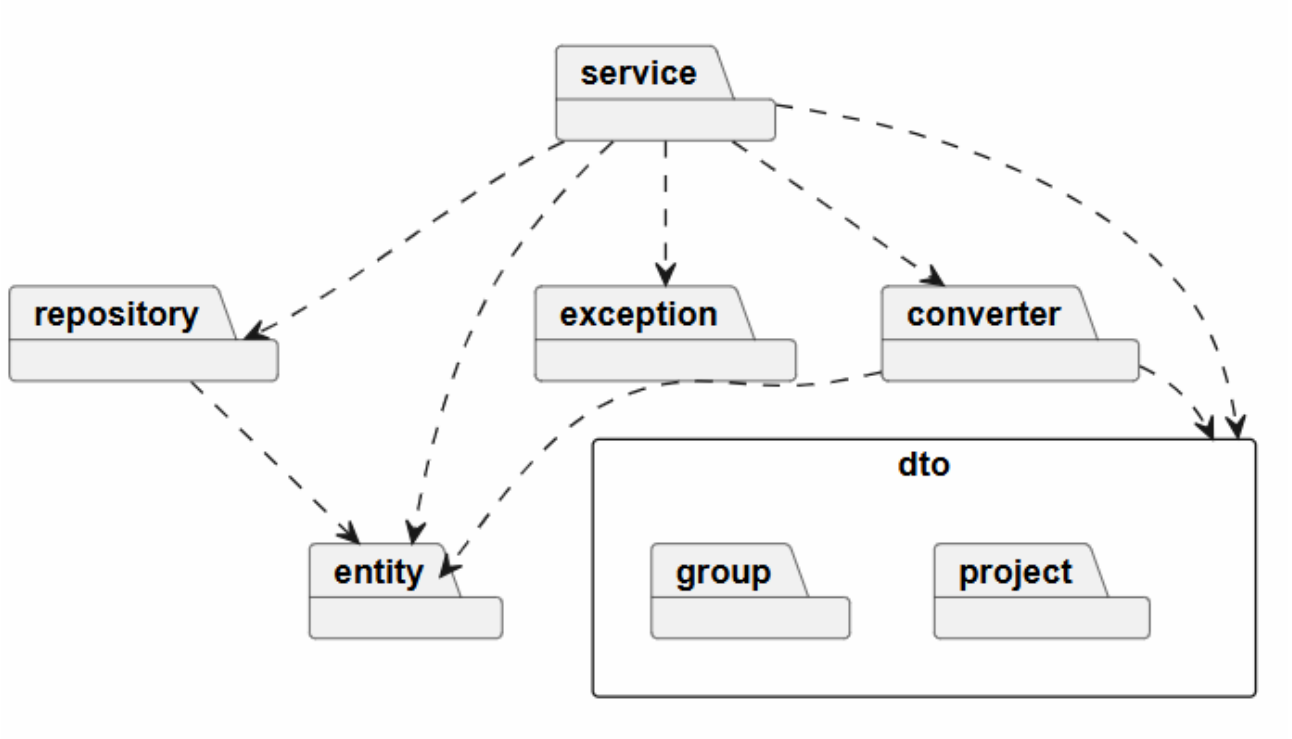


Рисунок 2.2 – Архітектура вебзастосунку для моніторингу версійності розгорнутих застосунків та їх розгортання

Діаграма зображує структуру вебзастосунку, який логічно розділений на чотири основні взаємопов'язані пакети:

– Пакет управління конфігураціями: базовий модуль, який відповідає за збереження стану вебзастосунку. Він працює з базою даних, зберігає інформацію про логічні групи та налаштовані проєкти.

– Пакет моніторингу: модуль, що відповідає безпосередньо за збір телеметрії. Він автономно здійснює HTTP-запити до розгорнутих мікросервісів та отримує від них відповіді.

- Пакет інтеграції з GitLab: модуль, що забезпечує взаємодію нашого застосунку зі стороннім сервісом GitLab (запуск процесів збірки CI/CD).
- Пакет відображення панелі керування: модуль клієнтської частини (фронтенд), з яким взаємодіє користувач.

Такий архітектурний підхід робить вебзастосунок слабкозв'язаною та зручною для масштабування.

2.3 Проєкт вебзастосунку для моніторингу версійності розгорнутих застосунків та їх розгортання за допомогою GitLab API та Gitlab CI

Процес проєктування програмного забезпечення розділено на три ключові етапи: ескізне, технічне та робоче проєктування. Це дозволяє послідовно деталізувати логіку роботи, внутрішню структуру та технологічний стек програмного продукту.

2.3.1. Ескізний проєкт вебзастосунку

Діаграми діяльності

Діаграма діяльності – це вид поведінкової діаграми в моделюванні програмного забезпечення, який показує послідовність виконання дій та переходів між ними. Вона використовується для візуалізації бізнес-процесів та алгоритмів програми, включаючи основні кроки, розгалуження, перевірки умов та паралельні потоки виконання задач. У розробленому вебзастосунку такі діаграми створено для трьох найголовніших функцій.

Перший важливий процес – це збір даних про те, яка саме версія застосунку зараз працює на сервері. На рис. 2.3 показано, як програмна система опитує розгорнуті мікросервіси.

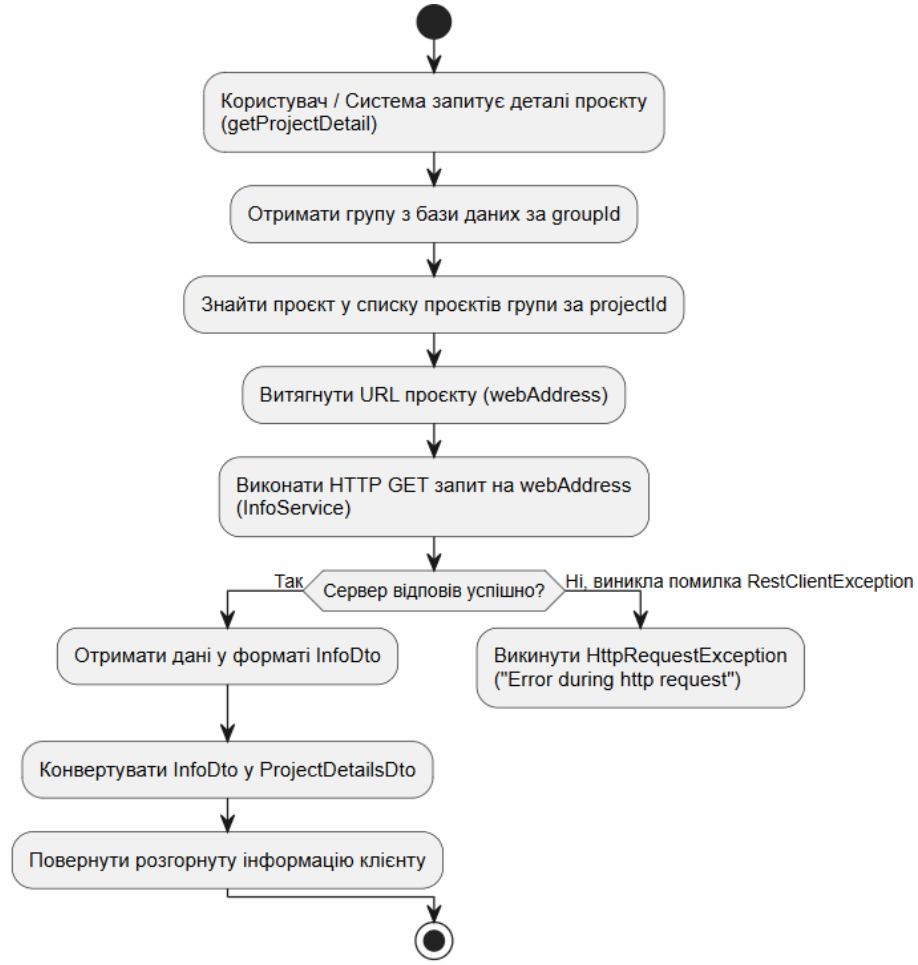


Рисунок 2.3 – Діаграма діяльності для процесу моніторингу застосунку

Процес ініціюється шляхом відкриття адміністратором загального списку проєктів і натискає на конкретний застосунок, щоб перевірити його стан. Програма бере збережену веб-адресу цього проєкту і відправляє туди спеціальний HTTP-запит. Далі йде розгалуження: якщо віддалений сервер відповідає нормально, застосунок читає отриманий JSON-файл, знаходить там номери версій і виводить їх на екран користувачу. Якщо ж сервер не відповідає або сталася помилка мережі, програма припиняє спроби, показує повідомлення про збій, і алгоритм завершується.

Другий ключовий сценарій – це дистанційний запуск оновлення застосунку за допомогою інтеграції з GitLab API. Цей процес зображено на рис. 2.4.

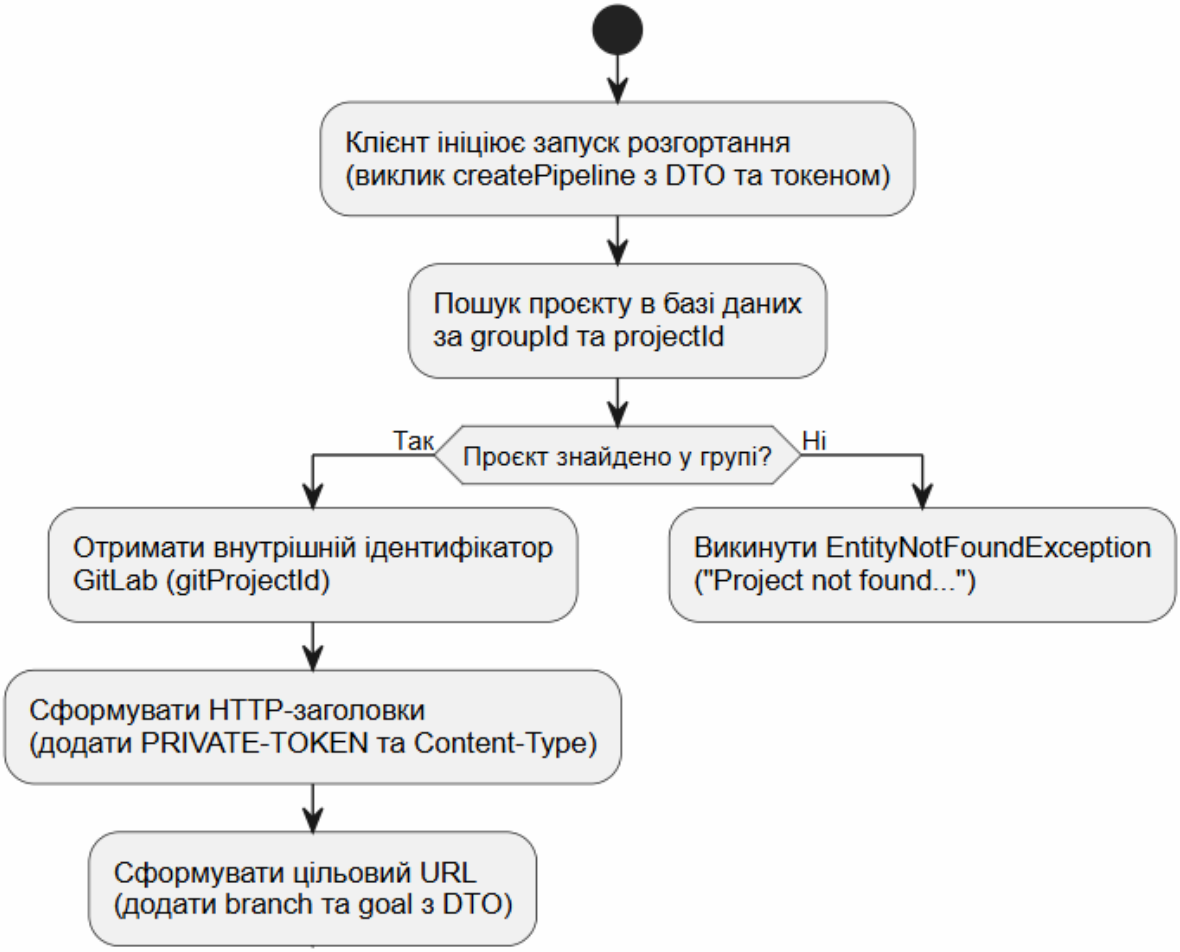


Рисунок 2.4 – Діаграма діяльності для процесу запуску розгортання через GitLab API

У разі виявлення на панелі моніторингу застарілої версії проєкту передбачено можливість безпосередньої ініціалізації його оновлення. Для запобігання випадковому запуску програмний продукт попередньо відображає вікно підтвердження дії. Після підтвердження серверна частина розробленого вебзастосунку формує відповідний запит та відправляє його до платформи GitLab. Далі програма очікує на обробку запиту: у випадку успішного прийняття команди платформою розпочинається автоматичний процес розгортання, а в інтерфейсі користувача відображається сповіщення про успішний старт. У разі виникнення збою генерується відповідне повідомлення про помилку.

Третій процес описує базову роботу з даними: як саме користувач додає нові проєкти до вебзастосунку або змінює їхні налаштування. Відповідну діаграму наведено на рис. 2.5.

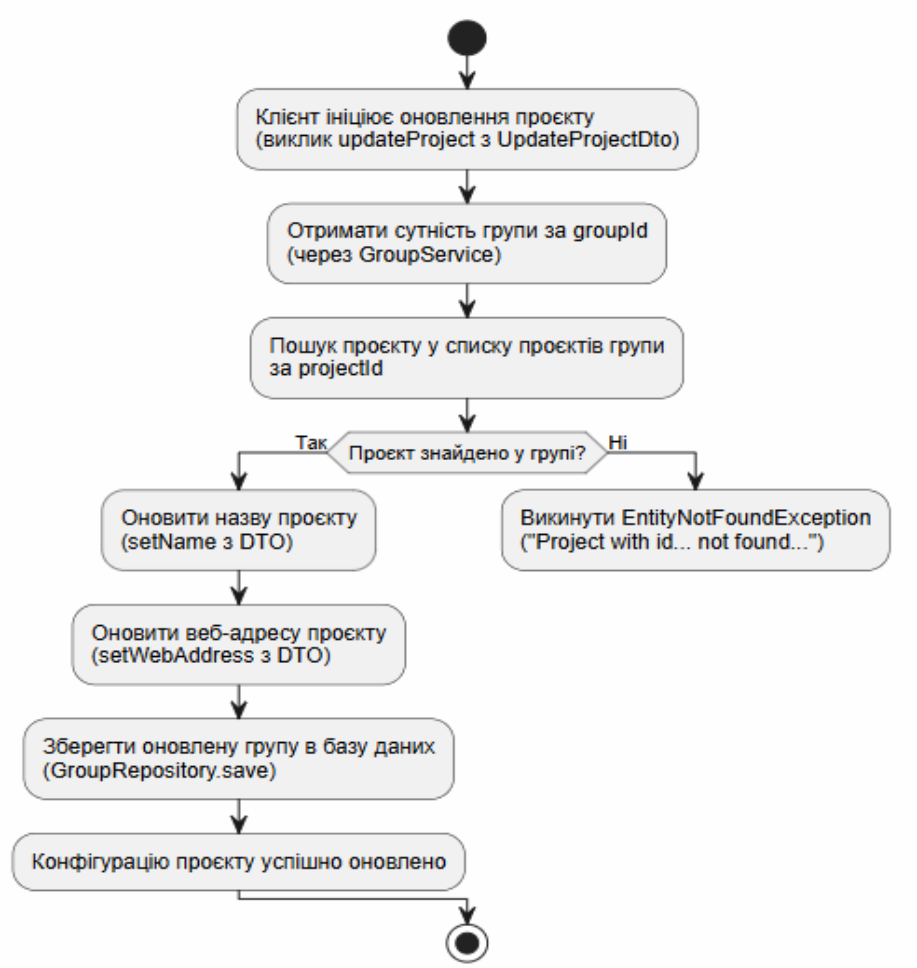


Рисунок 2.5 – Діаграма діяльності для процесу управління конфігурацією проєкту

Щоб програмна система могла моніторити застосунок, його спочатку потрібно зареєструвати. Користувач вибирає потрібну логічну групу і відкриває форму додавання проєкту. Там він вписує назву мікросервісу та його цільову веб-адресу. Після натискання кнопки збереження програма перевіряє, чи правильно введені дані. Якщо є помилки, форма підсвічує їх і просить виправити. Якщо всі дані введені коректно, новий проєкт зберігається в базі даних MongoDB. Після цього список проєктів на екрані оновлюється, і користувач бачить щойно створений запис.

Мокапи інтерфейсу

Мокапи – це візуальні макети майбутнього інтерфейсу програми. Вони показують, як виглядатиме вебзастосунок ще до того, як програмісти почнуть

писати код. У кваліфікаційній роботі створення мокапів допомагає заздалегідь продумати зручність застосунку, розташування кнопок та логіку роботи, щоб користувачам було комфортно ним користуватися.

Оскільки програмне забезпечення створене для моніторингу великої кількості мікросервісів, головною метою було зробити інтерфейс простим і не перевантаженим зайвою інформацією.

Головна сторінка вебзастосунку – це інформаційна панель, де зібрані всі дані про проєкти. На рисунку 2.6 представлено її макет.

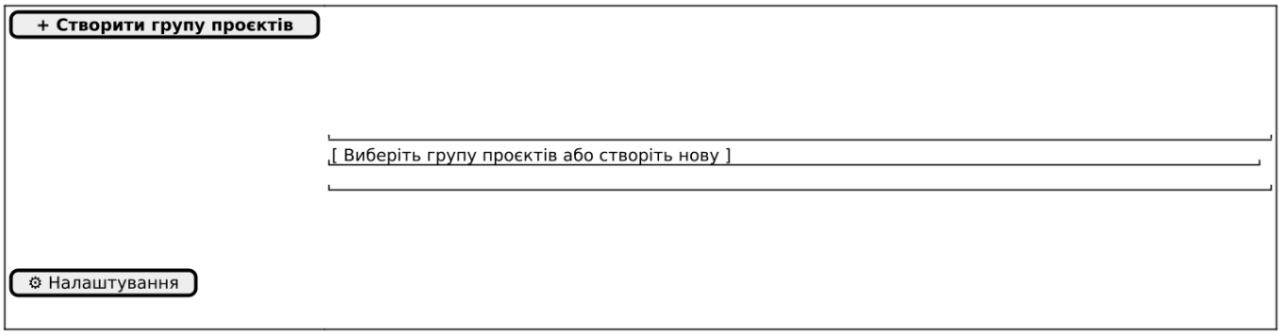


Рисунок 2.6 – Мокап головної інформаційної панелі застосунку

Інтерфейс головної сторінки поділено на кілька зрозумілих частин:

- Верхня панель: тут знаходиться назва програмного продукту та головне меню.
- Робоча область: тут відображаються створені групи середовищ. Всередині кожної групи показано список її проєктів.
- Інформація про проєкти: біля кожного мікросервісу відображається його назва, посилання та головні дані моніторингу.
- Кнопки керування: біля кожної групи чи проєкту є кнопки, щоб швидко їх відредагувати, видалити або додати нові.

Щоб не переходити на інші сторінки для створення або редагування груп і проєктів, було використано модальні вікна. На рисунку 2.7 показано макет вікна для роботи з групою.

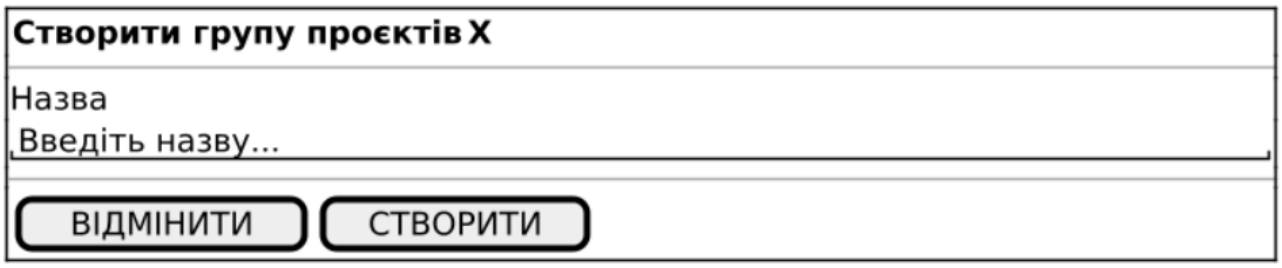


Рисунок 2.7 – Мокап форми додавання групи

Це вікно максимально просте. Користувачу потрібно лише ввести назву групи в єдине поле і натиснути кнопку збереження або скасування.

Для налаштування окремого проєкту використовується інше вікно, макет якого наведено на рисунку 2.8.

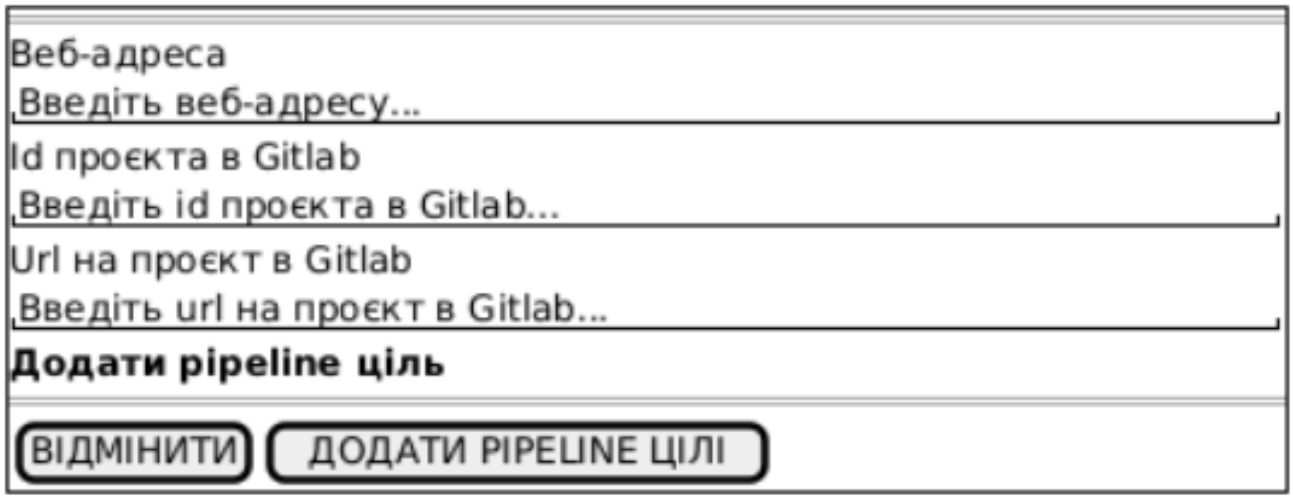


Рисунок 2.8 – Мокап форми додавання проєкту

У цьому вікні передбачено більше налаштувань. Крім назви, обов’язковим полем є веб-адреса проєкту. Саме за цим URL **застосунок** автоматично опитуватиме сервіс, щоб збирати актуальні дані про версії його технологій.

Для забезпечення можливості оперативного розгортання застосунків безпосередньо з робочої області проєкту було спроєктовано підсистему запуску CI/CD конвеєрів. Макет модального вікна, яке відповідає за цю функцію, наведено на рисунку 2.9.

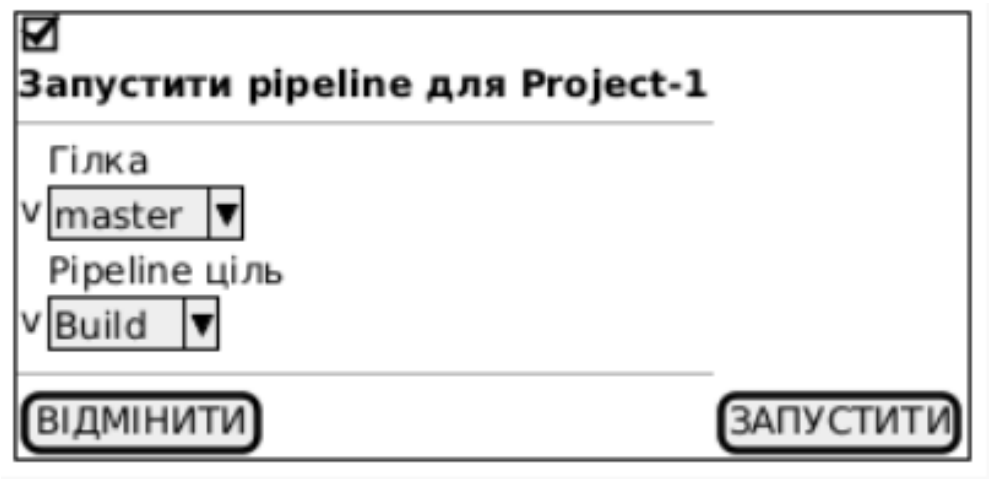


Рисунок 2.9 – Мокап форми запуску pipeline

Інтерфейс вікна запуску пайплайну передбачає використання випадаючих списків для вибору цільових параметрів розгортання:

- Вибір гілки (Branch): елемент інтерфейсу, що дозволяє адміністратору обрати саме ту версію коду з репозиторію, яка підлягає розгортанню.
- Вибір Pipeline цілі (Goal): елемент, який дозволяє визначити конкретний етап конвеєра, який необхідно ініціювати. Дані для цього вибору формуються на основі збереженої в базі даних конфігурації проєкту.

Використання такого спливаючого вікна для запуску оновлень робить інтерфейс інтуїтивно зрозумілим. Завдяки цьому підходу користувач може дистанційно відправляти команди до GitLab API та комфортно стежити за станом застосунків, не відволікаючись на переходи між зайвими сторінками.

Загалом, використання карток для відображення даних та модальних вікон для редагування робить інтерфейс інтуїтивно зрозумілим. Користувач може комфортно стежити за станом застосунків, не відволікаючись на переходи між зайвими сторінками.

2.3.2 Технічний проєкт вебзастосунку

Технічний проєкт описує внутрішню структуру, взаємодію компонентів, схему збереження даних та етапи програмної реалізації вебзастосунку.

Діаграма класів

Діаграма класів — це тип статичної структурної діаграми в UML, який описує структуру програмної системи, показуючи її класи, їхні атрибути, операції (методи) та зв'язки між об'єктами. Діаграму класів розробленого вебзастосунку наведено на рис. 2.10.

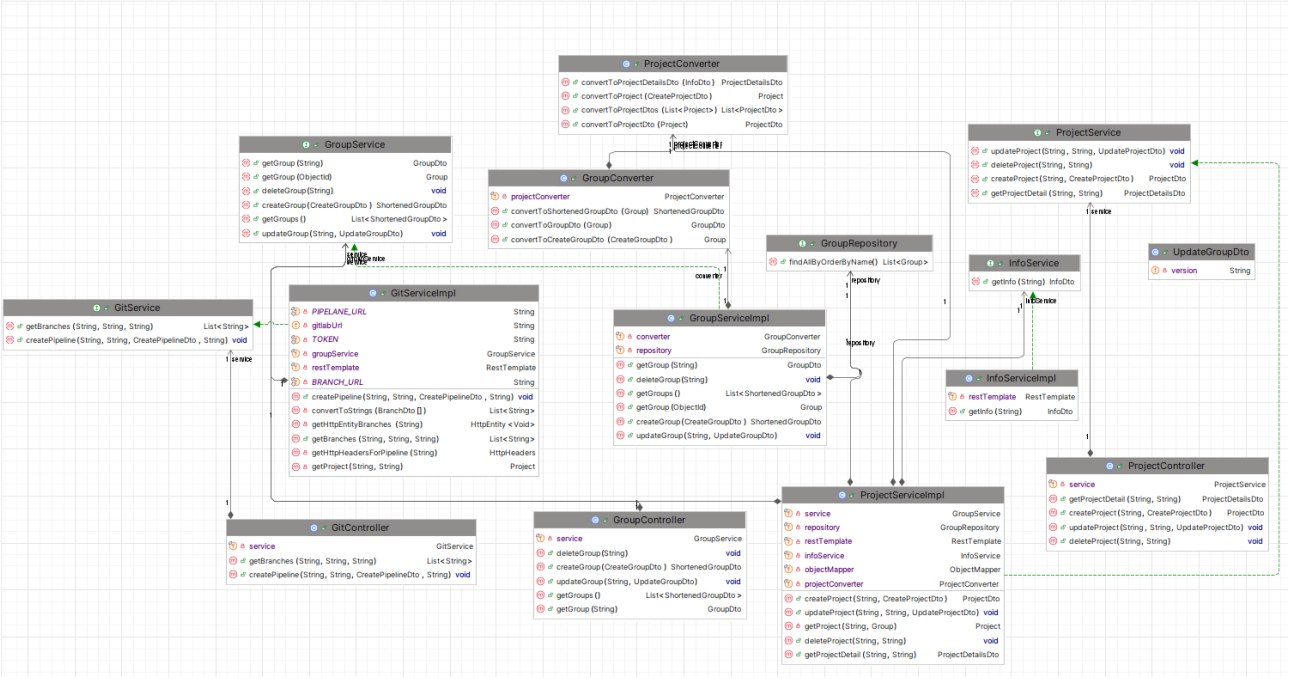


Рисунок 2.10 – Діаграма класів вебзастосунку для моніторингу версійності розгорнутих застосунків та їх розгортання

Для деталізації структури системи наведемо коротку специфікацію основних класів, зображених на діаграмі:

- ProjectController / GroupController: Класи API, що приймають запити від клієнта, валідують вхідні дані та передають їх до сервісного шару.
- ProjectService / GroupService: Класи бізнес-логіки. Відповідають за обробку даних, збереження в БД та виклик інших сервісів.
- GitLabIntegrationService: Клас, що інкапсулює логіку взаємодії з GitLab API (отримання списку гілок та запуску пайплайнів).
- ProjectRepository / GroupRepository: Інтерфейси доступу до даних, що успадковують методи MongoRepository для виконання CRUD-операцій.
- ProjectDTO / GroupDTO: Класи передачі даних, які використовуються для обміну інформацією між клієнтом та сервером.

Діаграма послідовності

Діаграма послідовності – це вид діаграми взаємодії в моделюванні програмного забезпечення, який показує обмін повідомленнями між об’єктами або компонентами програмної системи у час. Вона використовується для візуалізації динаміки роботи програми, включаючи хронологічний порядок викликів, передачу даних та взаємодію елементів під час виконання конкретного сценарію використання.

На рис. 2.11 представлена діаграма послідовності, яка детально ілюструє процес збору телеметричних даних та перевірки актуальних версій технологій розгорнутого проєкту у вебзастосунку. Цей процес є центральним для вирішення проблеми моніторингу фрагментованої інфраструктури.

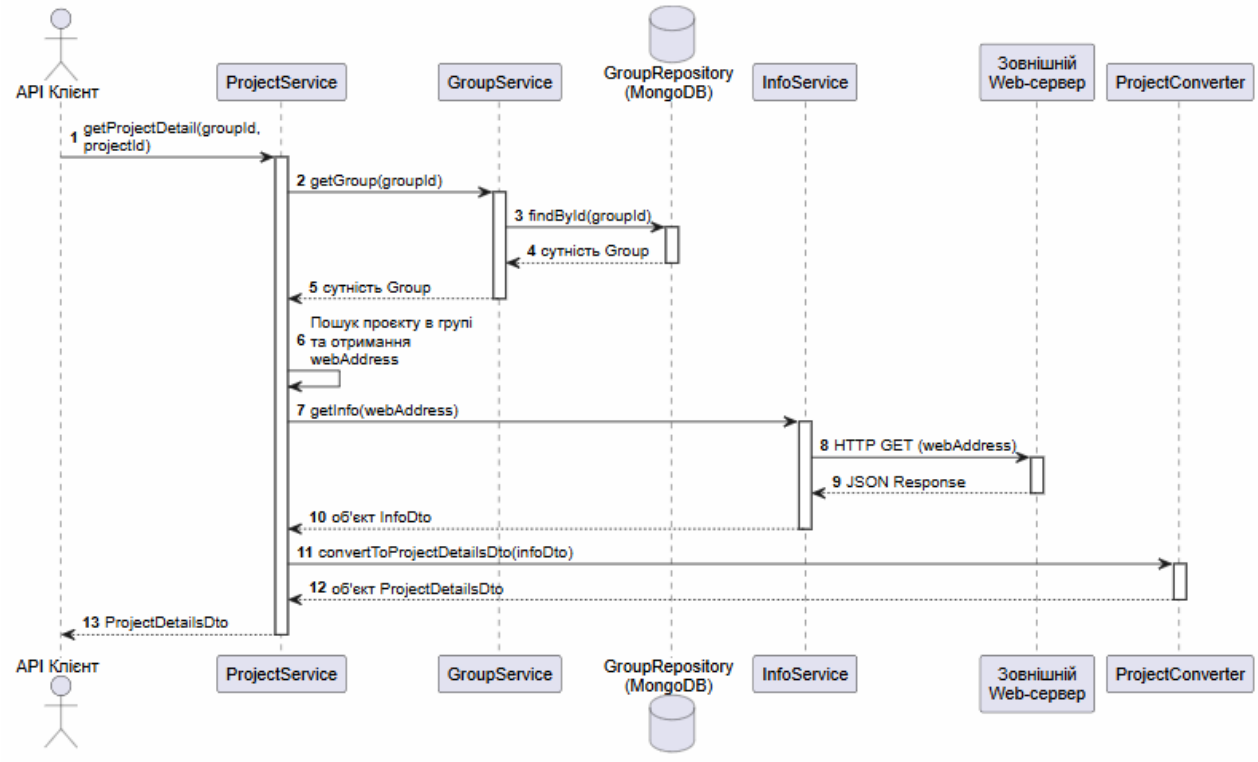


Рисунок 2.11 – Діаграма послідовності отримання даних про розгорнутий проєкт

Для початку ініціалізації цього процесу користувач, взаємодіючи з графічним інтерфейсом вебзастосунку, обирає конкретний проєкт, стан якого він бажає перевірити. Після цього користувач надсилає відповідний запит до модуля

представлення для отримання детальної аналітичної інформації. Модуль представлення, прийнявши ініціативу, передає запит контролеру. Своєю чергою, контролер звертається до бази даних для запиту конфігураційної інформації, яка необхідна для встановлення з’єднання, зокрема, цільової веб-адреси обраного проєкту.

База даних успішно знаходить потрібний запис та надсилає відповідні дані контролеру. Використовуючи отриману адресу, контролер ініціює прямий мережевий запит безпосередньо до віддаленого працюючого застосунку для збору поточних метрик та перевірки версій використовуваних технологій. Після успішного отримання та обробки відповіді від віддаленого застосунку, контролер формує кінцевий набір даних і повідомляє модуль представлення про успішне зчитування інформації. На завершальному етапі цього сценарію модуль представлення оновлює інтерфейс користувача та відображає деталі проєкту, надаючи адміністратору актуальну картину стану сервісу.

Другим критично важливим сценарієм використання вебзастосунку є процес управління реєстром сервісів, зокрема, додавання інформації про нові застосунки до загальної бази. Діаграма послідовності для процесу реєстрації нового проєкту в логічній групі середовищ представлена на рис. 2.12.

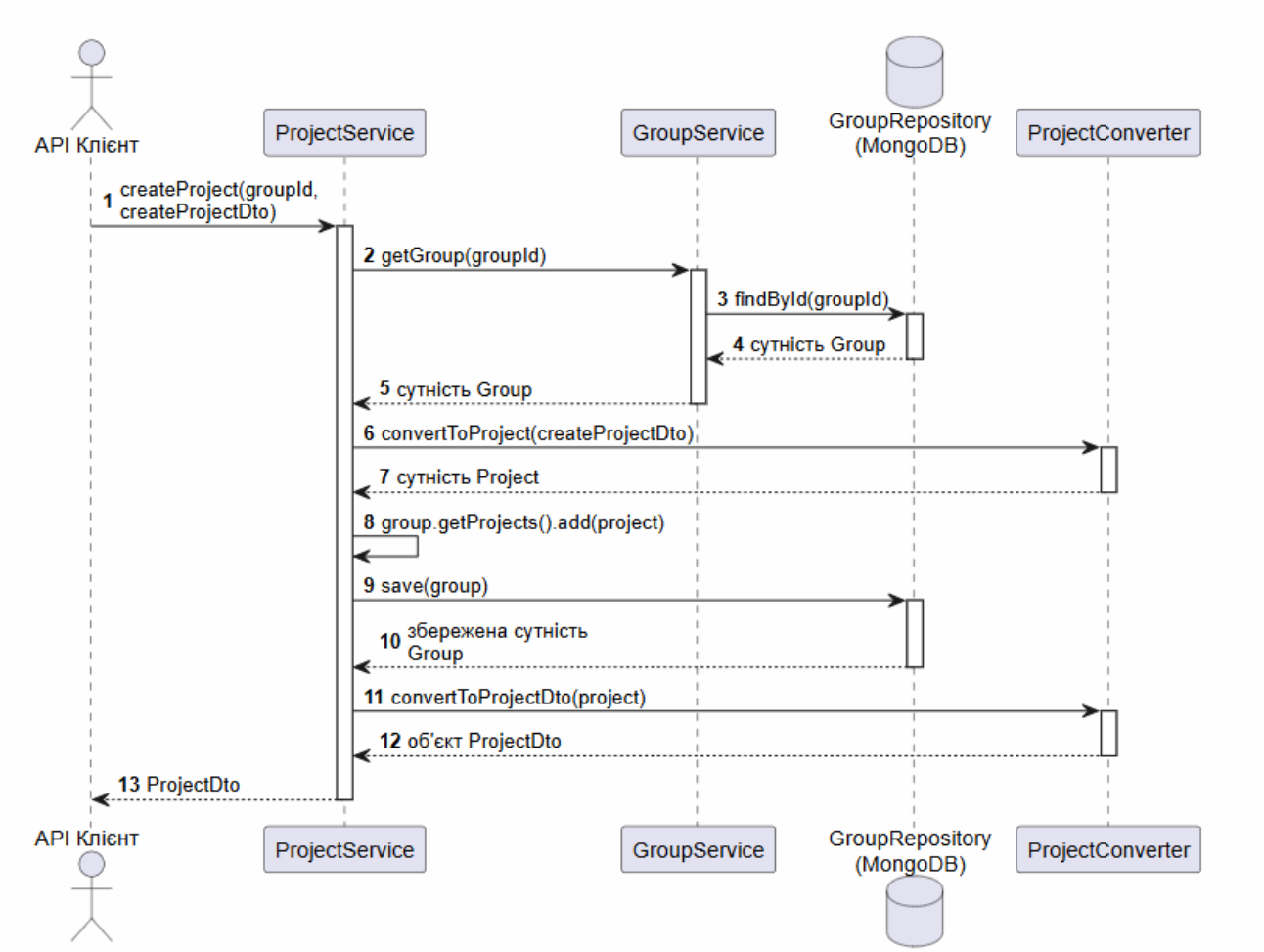


Рисунок 2.12 – Діаграма послідовності додавання нового проєкту

Дана діаграма ілюструє процес розширення вебзастосунку новими компонентами. Коли у користувача виникає необхідність додати новий проєкт для подальшого моніторингу, модуль представлення відображає йому відповідну форму для введення базових конфігураційних даних. Після того як користувач заповнює форму та ініціює збереження, ці дані надсилаються до модуля представлення, який структурує їх та передає контролеру для проведення процедури валідації.

Контролер здійснює перевірку отриманих даних на коректність та повноту. У разі успішного проходження валідації, контролер передає сформований об’єкт до бази даних для безпосереднього збереження в базі даних. База даних виконує операцію запису та підтверджує успішне збереження даних, про що інформує контролер. Отримавши підтвердження про успішне завершення транзакції, контролер передає відповідний сигнал модулю представлення. Нарешті, модуль

представлення оновлює внутрішній список проєктів та відображає оновлений перелік на екрані, інформуючи користувача про успішне завершення операції. Завдяки такій послідовності забезпечується надійність збереження даних та узгодженість відображуваної інформації.

Діаграма станів та переходів

Діаграма станів та переходів – це вид поведінкової діаграми в моделюванні програмного забезпечення, який показує різні стани, в яких може перебувати об’єкт протягом свого життєвого циклу, а також події, що викликають переходи між цими станами. Вона використовується для візуалізації динаміки зміни стану конкретного компонента вебзастосунку, включаючи його початковий стан, проміжні етапи роботи, умови зміни стану та можливі кінцеві результати. Якщо попередня діаграма діяльності показувала загальний алгоритм роботи, то діаграма станів фокусується на «житті» одного конкретного елемента. У нашому застосунку таким ключовим елементом є картка розгорнутого проєкту.

Діаграму станів та переходів для об’єкта проєкту в процесі моніторингу та розгортання представлено на рис. 2.13.

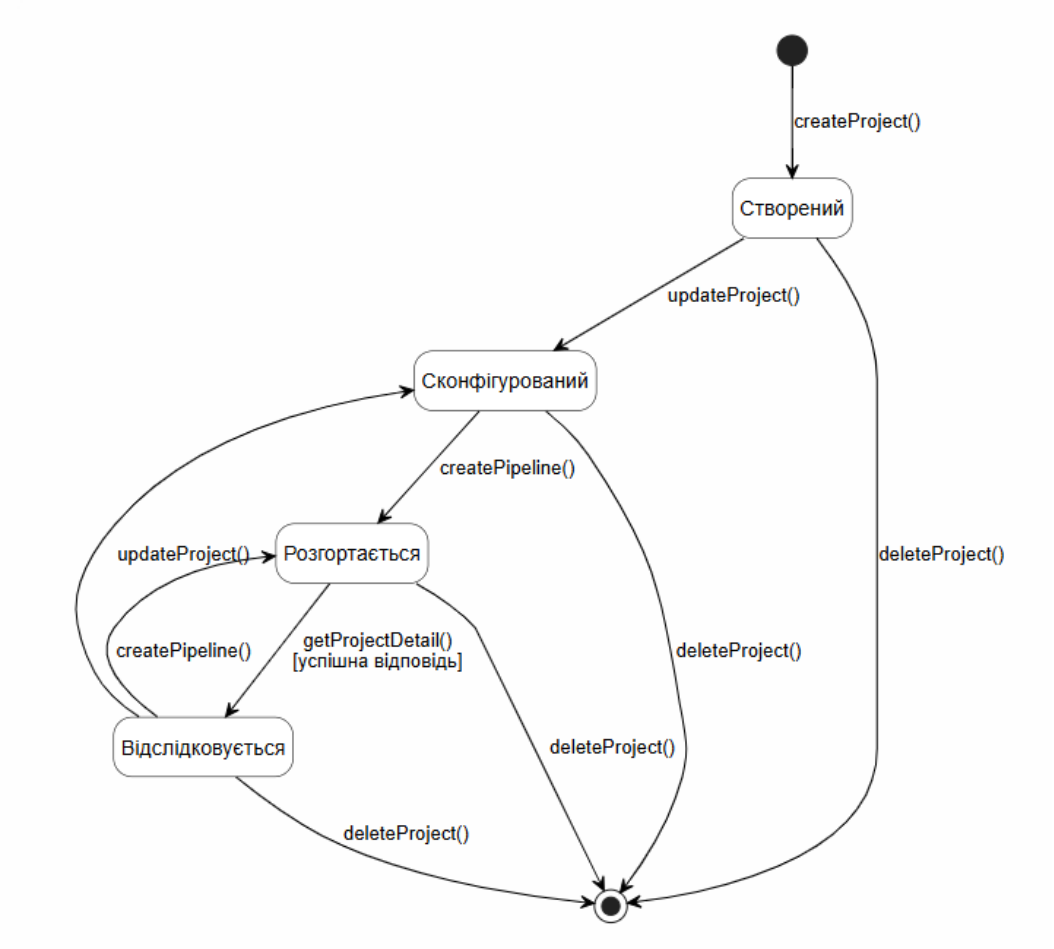


Рисунок 2.13 – Діаграма станів та переходів для об’єкта проєкту

Життєвий цикл проєкту в нашому вебзастосунку починається зі стану «Створений». У цей стан проєкт потрапляє одразу після того, як адміністратор додає його до певної логічної групи та вказує його базові налаштування. На цьому етапі вебзастосунок ще не має жодних телеметричних даних про реальний стан застосунку.

Коли програма або сам користувач ініціює перевірку метрик, проєкт переходить у стан «Опитування». У цей момент бекенд виконує HTTP-запит до віддаленого сервера. Залежно від того, яку відповідь отримає програма, відбувається перехід в один із трьох можливих станів:

– Якщо віддалений сервер не відповідає або виникає помилка мережі, проєкт переходить у стан «Помилка з’єднання». Користувач бачить відповідне попередження на панелі моніторингу.

– Якщо запит пройшов успішно, застосунок перевіряє отримані версії технологій. Якщо версія відповідає очікуваній, проєкт переходить у стан «Актуальний».

– Якщо ж під час перевірки виявляється, що на сервері працює стара версія застосунку, він переходить у стан «Застарілий».

У разі необхідності оновлення застосунку ініціюється процес розгортання через програмну взаємодію з GitLab API. Для цього бекенд системи формує цільовий HTTP-запит методом POST, передаючи необхідні конфігураційні параметри, зокрема цільову гілку репозиторію, та токен авторизації. Це дозволяє віддалено запустити виконання CI/CD пайплайну безпосередньо в інфраструктурі GitLab. Оскільки саме розгортання відбувається на сторонній платформі, для перевірки його результатів програма використовує механізм повторного опитування. Здійснення нового запиту за веб-адресою проєкту дозволяє отримати актуальні телеметричні дані та остаточно переконатися, що оновлення пройшло успішно і сервіс функціонує коректно.

Структура бази даних

Організація збереження даних у розробленому вебзастосунку базується на використанні нереляційної документо-орієнтованої СУБД MongoDB [9]. Вибір саме цього рішення зумовлений архітектурними потребами вебзастосунку, зокрема необхідністю ефективної обробки конфігураційних даних у вигляді ієрархічних структур. Головною одиницею збереження інформації є колекція Group, що акумулює в собі відомості про логічні об’єднання середовищ та підпорядковані їм мікросервіси.

Документна модель колекції Group візуалізована нижче.

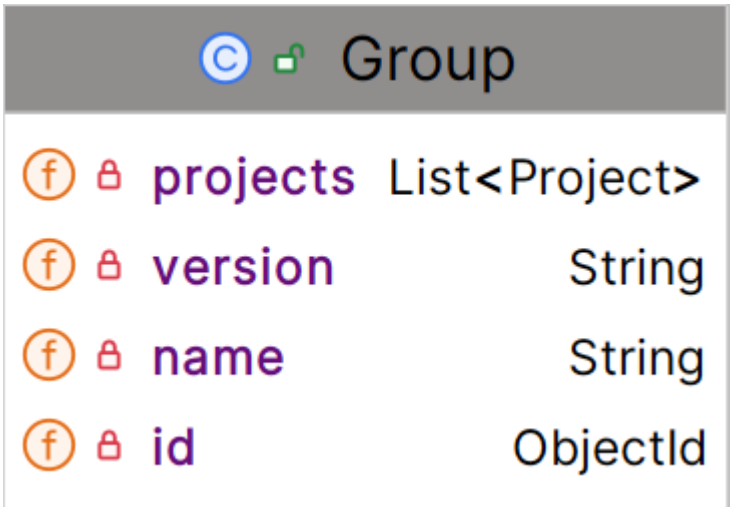


Рисунок 2.14 – Структура документу колекції Group

Вона включає наступний набір атрибутів:

1) _id (тип objectId, обов’язкове поле) – системний ідентифікатор, що генерується механізмами MongoDB для забезпечення унікальності кожного запису;

2) name (тип string, обов'язкове поле) – текстова назва сформованої групи середовищ;

3) _class (тип string, обов'язкове поле) – службові метадані, які фреймворк Spring використовує для коректного відображення (мапінгу) BSON-документа на відповідний Java-клас під час зчитування;

4) projects (тип array of objects, обов'язкове поле) – масив вбудованих (embedded) документів, кожен з яких описує окремий проєкт (мікросервіс) у межах поточної групи.

У свою чергу, кожен елемент масиву projects характеризується такою структурою:

1) _id (objectId) – унікальний ключ конкретного мікросервісу;

2) name (string) – найменування розгорнутого проєкту;

3) webAddress (string) – URL-адреса хоста розгорнутого сервісу, до якої вебзастосунок звертається за допомогою інструменту RestTemplate для прямого збору телеметрії;

4) gitProjectId (string) – внутрішній ідентифікатор проєкту на платформі GitLab, який використовується для маршрутизації API-запитів;

5) gitProjectUrl (string) – посилання на репозиторій проєкту в системі контролю версій;

6) goals (array of strings) – перелік доступних цілей (пайплайн-конфігурацій) для розгортання даного застосунку.

Інтеграційна взаємодія серверної частини застосунку з базою даних реалізована за допомогою інструментарію Spring Data MongoDB [10]. На рівні програмного коду доступ до даних інкапсульовано в інтерфейсі GroupRepository, який є спадкоємцем базового класу MongoRepository<Group, ObjectId>. Такий підхід позбавляє від необхідності ручного написання низькорівневих запитів, надаючи розробнику високорівневу абстракцію з готовими CRUD-методами.

Важливим архітектурним рішенням стала відмова від нормалізації даних у класичному реляційному розумінні. Замість виділення проєктів у відокремлену колекцію та їхнього зв’язування за зовнішніми ключами, було застосовано патерн вкладених документів. Це забезпечило вебзастосунку низку вагомих переваг:

1. Гарантія транзакційної цілісності без застосування складних блокувань. Оскільки MongoDB підтримує атомарність операцій на рівні окремого документа, будь-які мутації відбуваються шляхом оновлення цілісного об'єкта Group. Це унеможливлює виникнення аномалій даних.

2. Оптимізація швидкості зчитування. Формування ієрархічного дерева на інформаційній панелі клієнтського інтерфейсу вимагає виконання лише одного запиту до СУБД. Відсутність необхідності в операціях злиття таблиць (аналогів SQL-команди JOIN) суттєво зменшує системну затримку під час обробки моніторингових метрик.

3. Архітектурна однорідність. Інформація від сторонніх платформ, таких як GitLab API та віддалені сервіси, надходить у вебзастосунок у форматі JSON. Компонент ObjectMapper трансформує ці відповіді у Java-об’єкти, після чого СУБД зберігає їх у бінарному форматі BSON. Таким чином досягається

безшовний і природний потік інформації на всіх етапах її життєвого циклу – від отримання з мережі до збереження на жорсткому диску.

2.3.3 Робочий проєкт вебзастосунку

Вибір правильного стеку технологій є одним із найважливіших етапів створення програмного забезпечення, оскільки від цього залежить швидкість розробки, надійність програмного продукту, легкість його підтримки та можливості для подальшого масштабування. Зважаючи на специфіку розроблюваного вебзастосунку, було обрано сучасний та перевірений набір інструментів.

Основною мовою програмування для розробки серверної частини було обрано Java. Це потужна об’єктно-орієнтована мова програмування, яка славиться своєю надійністю, високою продуктивністю та розвиненою екосистемою. Використання Java дозволило реалізувати чітку та строгу архітектуру системи, ефективно управляти пам’яттю та забезпечити стабільну роботу програми під час паралельного опитування розгорнутих застосунків. Головним каркасом для побудови серверної логіки став фреймворк Spring Boot [11]. Цей інструмент є стандартом у сучасній розробці корпоративних додатків на мові Java. Його використання дозволило значно скоротити час на базове налаштування проєкту завдяки принципу «конвенції замість конфігурації». За допомогою Spring Boot було легко реалізовано:

- Архітектуру REST API (контролери) для обміну даними з клієнтською частиною [12].
- Впровадження залежностей (Dependency Injection) між сервісами, репозиторіями та маперами.
- Використання компонента RestTemplate для виконання прямих HTTP-запитів до розгорнутих проєктів та до платформи GitLab.

Оскільки система активно працює з мережевими запитами, важливу роль відіграє бібліотека Jackson. Віддалені мікросервіси та GitLab API повертають

дані у вигляді «сирого» тексту у форматі JSON. ObjectMapper дозволяє застосунку швидко та ефективно читати цей текст знаходити у ньому потрібні метрики та перетворювати їх на зручні Java-об’єкти для подальшого відображення користувачу.

У якості системи управління базами даних було обрано MongoDB. Це сучасна документо-орієнтована NoSQL база даних. Її головна перевага для даного проєкту полягає у тому, що вона зберігає дані у форматі, дуже схожому на JSON. Оскільки вебзастосунок постійно отримує JSON-дані від зовнішніх сервісів і передає їх на фронтенд, використання MongoDB дозволяє уникнути складних перетворень, характерних для традиційних реляційних баз даних. Крім того, MongoDB легко розширюється, що дозволить у майбутньому додавати нові поля до конфігурацій проєктів без зміни структури бази.

Взаємодія між клієнтською частиною застосунку та описаним вище сервером відбувається за протоколом HTTP з використанням архітектурного стилю REST. Це означає, що бекенд та фронтенд працюють повністю незалежно один від одного і спілкуються виключно шляхом обміну стандартизованими JSON-пакетами.

Для розробки клієнтської частини вебзастосунку було обрано бібліотеку React [13]. Її компонентна архітектура дозволила розбити складний інтерфейс панелі моніторингу на незалежні, перевикористовувані блоки, що суттєво спростило розробку та подальшу підтримку коду. Основною мовою програмування фронтенду виступає TypeScript – надбудова над JavaScript, яка додає строгу статичну типізацію [14]. Це значно підвищує надійність застосунку: завдяки інтерфейсам типів більшість помилок при обробці JSON-даних від бекенду виявляються ще на етапі написання коду. Збирання проєкту та оптимізація ресурсів для робочого середовища здійснюється за допомогою сучасного інструменту Vite, який забезпечує надзвичайно високу швидкість збірки.

Оскільки вебзастосунок має складну логіку та працює з великим масивом даних у реальному часі, для управління глобальним станом використано Redux

Toolkit [15]. Це офіційний та найбільш ефективний інструментарій для екосистеми Redux, який дозволяє оптимізувати роботу з асинхронними HTTP-запитами до Spring Boot сервера та уникнути зайвих перемалювань інтерфейсу. Для забезпечення навігації між різними розділами вебзастосунку без перезавантаження сторінки інтегровано бібліотеку React Router [16].

Особливу увагу під час розробки фронтенду було приділено зручності взаємодії користувача з вебзастосунком та надійності введених даних. Візуальне оформлення побудовано на базі адаптивного фреймворку React Bootstrap з використанням стилістичних тем Bootswatch та іконок React Bootstrap Icon [17]. Процес створення та редагування конфігурацій реалізовано за допомогою React Hook Form, що забезпечує високу продуктивність форм [18]. Для строгої перевірки коректності даних на стороні клієнта інтегровано бібліотеку Yup. Для інформування адміністратора про результати операцій, такі як успішний запуск розгортання через GitLab API або помилки мережевого з'єднання, використовується система спливаючих повідомлень React Toastify.

Програмна реалізація вебзастосунку

Кодування сервісного шару управління групами проєктів

У процесі розробки серверної частини вебзастосунку одним із першочергових завдань стала реалізація механізму управління логічними групами середовищ. Оскільки розроблена програмна платформа призначена для моніторингу великої кількості розгорнутих мікросервісів, ці мікросервіси необхідно було логічно структурувати та об'єднувати. Для вирішення цього завдання було створено спеціальний сервісний клас GroupServiceImpl.

У межах обраної багаторівневої архітектури застосунку цей клас реалізує сервісний шар. Він виступає проміжним компонентом між REST-контролером, який обробляє вхідні HTTP-запити від клієнтського застосунку, та шаром доступу до даних (репозиторієм). Основне призначення даного модуля полягає в інкапсуляції бізнес-логіки вебзастосунку та виконанні базових CRUD-операцій (створення, читання, оновлення, видалення) над сутностями груп середовищ.

З метою оптимізації мережевого трафіку та приховування внутрішньої структури бази даних, у модулі активно застосовується патерн DTO (Data Transfer Object) за допомогою спеціалізованого компонента GroupConverter. Це дозволяє передавати на сторону клієнта лише ті дані, які необхідні для коректної роботи користувацького інтерфейсу.

Логіка роботи сервісу базується на таких основних методах:

- Отримання списку груп (getGroups). Цей метод використовується для формування базового представлення даних на головній сторінці застосунку. Він ініціює запит до бази даних для отримання всіх існуючих записів про групи. Далі, за допомогою конвертера, кожна сутність перетворюється у скорочений об’єкт ShortenedGroupDto, який містить лише базову інформацію (ідентифікатор та назву), необхідну для побудови списку в інтерфейсі.
- Перегляд деталей групи (getGroup). Для отримання розширеної інформації про конкретну групу (зокрема, переліку пов’язаних з нею проєктів) реалізовано даний метод. Він приймає рядковий ідентифікатор, здійснює пошук відповідної сутності та повертає повний набір даних у форматі GroupDto.
- Обробка виняткових ситуацій при пошуку. Важливим елементом надійності застосунку є внутрішній перевантажений метод getGroup, який приймає параметр типу ObjectId. Його завдання полягає у безпечному вилученні сутності з бази даних. У випадку відсутності запису за вказаним ідентифікатором, метод не допускає аварійного завершення програми, а натомість генерує виняток EntityNotFoundException. Це дозволяє коректно обробити помилку на рівні контролера та повернути користувачу відповідний статус-код.
- Створення нової групи (createGroup). Метод обробляє вхідні дані від користувача, інкапсульовані в об’єкті CrudGroupDto. Конвертер перетворює ці дані у повноцінну сутність Group, яка потім зберігається в базі даних. Для забезпечення синхронізації стану інтерфейсу без додаткових запитів, метод одразу повертає створений об’єкт у форматі ShortenedGroupDto.

- Оновлення існуючої групи (updateGroup). Цей метод забезпечує можливість редагування атрибутів групи. Він виконує пошук існуючої сутності за ідентифікатором, оновлює необхідні поля (наприклад, назву групи) даними з CrudGroupDto та виконує перезапис об’єкта в базі даних, при цьому зберігаючи цілісність усіх пов’язаних даних.
- Видалення групи (deleteGroup). Для підтримки актуальності даних у базі даних та управління життєвим циклом сутностей реалізовано метод видалення. Він виконує вилучення запису про відповідну групу з бази даних за її унікальним ідентифікатором.

Таким чином, розроблений клас забезпечує повний цикл управління групами середовищ, інтегруючи бізнес-логіку з механізмами доступу до даних та обробкою виключень, що робить серверну частину застосунку стабільною та стійкою до некоректних запитів.

Кодування сервісного шару управління проєктами та моніторингу їх стану

Наступним логічним етапом розробки серверної частини вебзастосунку стала програмна реалізація підсистеми управління безпосередньо самими проєктами, які входять до складу раніше створених груп. Для виконання цього завдання, а також для забезпечення ключового функціоналу системи – моніторингу версійності технологій, було розроблено сервісний клас ProjectServiceImpl.

Даний модуль інтегрований у загальну архітектуру як компонент бізнес-логіки. На відміну від сервісу груп, ProjectServiceImpl вирішує складніші завдання: окрім стандартного управління даними, він відповідає за мережеву взаємодію з віддаленими мікросервісами. Для цього в класі задіяні такі інструменти, як RestTemplate для виконання синхронних HTTP-запитів та ObjectMapper для синтаксичного аналізу отриманих JSON-відповідей.

Внутрішня логіка управління даними побудована таким чином, що сутності проєктів зберігаються всередині сутностей груп. Тому всі операції зміни

проєктів передбачають оновлення стану відповідної батьківської групи. Робота модуля забезпечується наступними основними методами:

- Створення нового проєкту (createProject). Метод приймає ідентифікатор цільової групи та вхідні дані проєкту. За допомогою сервісу груп здійснюється пошук батьківської сутності, після чого конвертер перетворює DTO у новий об’єкт проєкту. Створений проєкт додається до внутрішнього списку групи, і оновлений стан групи фіксується в базі даних.
- Оновлення конфігурації проєкту (updateProject). Цей функціонал є критично важливим, оскільки саме тут задається та редагується веб-адреса (webAddress) розгорнутого мікросервісу. Метод виконує пошук конкретного проєкту всередині групи за його ідентифікатором, оновлює необхідні базові параметри (назву, веб-адресу) та конфігураційні дані для інтеграції з GitLab (ідентифікатор репозиторію, посилання та цілі розгортання), після чого перезаписує збережену групу.
- Видалення проєкту (deleteProject). Метод забезпечує вилучення даних про мікросервіс, моніторинг якого більше не потрібен. Відбувається фільтрація масиву проєктів обраної групи, під час якої видаляється запис із відповідним ідентифікатором, після чого оновлена колекція зберігається.
- Отримання телеметрії та версійності (getProjectDetail). Це ключовий метод, який реалізує головну мету вебзастосунку – моніторинг розгорнутих застосунків. Метод отримує актуальну веб-адресу проєкту і за допомогою RestTemplate ініціює прямий HTTP GET-запит до цього мікросервісу. Отримавши відповідь у форматі JSON, програма за допомогою ObjectMapper виконує обхід дерева структури відповіді та витягує значення актуальних версій технологій. Зібрані дані упаковуються в ProjectDetailsDto і передаються на клієнтську частину для відображення.
- Обробка мережевих винятків та помилок пошуку. У процесі мережевої взаємодії з віддаленими сервісами існує висока ймовірність затримок або збоїв. Тому метод getProjectDetail містить блок обробки виключень, який перехоплює помилки HTTP-запитів та збої обробки JSON, перетворюючи їх у

кастомний виняток HttpRequestException. Крім того, для уникнення помилок при пошуку проєктів реалізовано приватний допоміжний метод getProject, який у разі відсутності цільового проєкту генерує виняток EntityNotFoundException.

Таким чином, розроблений клас ProjectServiceImpl не лише забезпечує структуроване зберігання конфігурацій мікросервісів, але й успішно реалізує механізм активного опитування віддалених застосунків, що є основою підсистеми моніторингу та управління версійністю.

Кодування сервісного шару інтеграції з GitLab API

Одним із ключових завдань вебзастосунку є можливість не лише спостерігати за станом мікросервісів, а й дистанційно управляти їх розгортанням. Для реалізації цього функціоналу було розроблено спеціалізований сервісний клас GitServiceImpl.

У межах архітектури застосунку цей модуль виступає клієнтом для програмної взаємодії із зовнішньою платформою через її відкритий інтерфейс [19]. Як і в підсистемі моніторингу, основним інструментом для виконання мережевих викликів тут слугує компонент RestTemplate. Для гнучкості налаштувань базова адреса платформи не "зашивається" в код жорстко, а автоматично завантажується з конфігураційного файлу застосунку за допомогою анотації @Value.

Програмна логіка модуля забезпечує виконання наступних основних операцій:

– Отримання списку доступних гілок репозиторію (getBranches). Цей метод необхідний для того, щоб надати користувачеві вибір конкретної версії коду перед запуском оновлення. Сервіс формує HTTP GET-запит до GitLab, обов’язково додаючи до заголовків запиту спеціальний ключ PRIVATE-TOKEN із токеном авторизації. Після успішного отримання відповіді у форматі JSON, програма за допомогою допоміжних методів витягує з неї назви гілок і повертає їх на клієнтську частину у вигляді зручного списку рядків.

– Ініціалізація конвеєра розгортання (createPipeline). Метод відповідає за дистанційний запуск CI/CD пайплайну. Він приймає об’єкт CreatePipelineDto,

який містить обрану користувачем гілку та мету розгортання. На основі цих даних формується цільовий URL, після чого сервіс відправляє HTTP POST-запит. Важливою деталлю реалізації є налаштування заголовка запиту у формат application/x-www-form-urlencoded, що є технічною вимогою GitLab для коректної передачі конфігураційних змінних.

– Пошук та валідація проєкту (getProject). Для забезпечення надійності роботи перед відправкою будь-якого запиту до зовнішнього API вебзастосунок має переконатися в наявності необхідних даних. Цей приватний метод звертається до сервісу груп (GroupService), знаходить цільовий проєкт у локальній базі даних та вилучає його ідентифікатор для GitLab (gitProjectId). Якщо запитуваний проєкт з якихось причин відсутній у базі, метод безпечно перериває процес і генерує виняток EntityNotFoundException.

Таким чином, розроблений клас надійно ізолює логіку мережевої взаємодії з GitLab API від інших компонентів застосунку. Він забезпечує безпечну передачу авторизаційних даних та дозволяє адміністратору зручно запускати процеси оновлення мікросервісів безпосередньо з інтерфейсу моніторингу.

Кодування сервісного шару отримання інформації про стан сервісів

Одним із важливих завдань вебзастосунку є можливість безперервного збору актуальних даних про стан віддалених мікросервісів або веб-вузлів. Для реалізації цього функціоналу було розроблено спеціалізований сервісний клас InfoServiceImpl.

У межах архітектури застосунку цей модуль виступає клієнтом для виконання базових запитів до зовнішніх ресурсів. Основною складовою для виконання мережевих викликів тут слугує компонент RestTemplate. Для забезпечення чистоти коду та автоматичної ін’єкції залежностей через конструктор використовується анотація @RequiredArgsConstructor бібліотеки Lombok, а сам клас зареєстровано в контексті Spring як компонент бізнес-логіки за допомогою анотації @Service.

Програмна логіка модуля забезпечує виконання наступної ключової операції:

– Отримання інформаційних даних за веб-адресою. Цей метод необхідний для здійснення запиту до конкретного вузла та зчитування його поточних показників. Він приймає цільову адресу у вигляді текстового рядка. Сервіс формує стандартний HTTP GET-запит за допомогою методу getForObject компонента RestTemplate та автоматично десеріалізує отриману відповідь в об’єкт передачі даних InfoDto. Важливою деталлю реалізації є механізм обробки позаштатних ситуацій: у разі виникнення проблем із мережевою взаємодією або недоступності кінцевої точки, метод перехоплює цю помилку та генерує власний виняток HttpRequestException із відповідним повідомленням, що дозволяє уникнути падіння застосунку.

Таким чином, розроблений клас надійно інкапсулює логіку виконання інформаційних запитів до зовнішніх систем. Він забезпечує уніфікований підхід до обробки мережевих помилок та дозволяє іншим компонентам застосунку зручно й безпечно отримувати структуровані дані про віддалені сервіси.

Кодування модуля маршрутизації та API управління групами

Важливою складовою архітектури сучасного вебзастосунку є забезпечення зручного програмного інтерфейсу для взаємодії клієнтської частини з серверною. Для реалізації функціоналу управління проєктами в межах створених груп було розроблено спеціалізований контролер ProjectController.

У вебзастосунку цей модуль виконує роль вхідної точки для прийому та первинної обробки HTTP-запитів, пов’язаних із життєвим циклом проєктів. Клас позначено анотацією @RestController, що вказує на його приналежність до веб-шару та забезпечує автоматичну серіалізацію результатів роботи методів у формат JSON. Завдяки анотації @RequiredArgsConstructor (бібліотека Lombok) реалізовано автоматичну ін’єкцію залежності бізнес-логіки – компонента ProjectService. Крім того, на рівні всього класу застосовано анотацію @Validated, яка активує механізм перевірки вхідних параметрів ще до моменту передачі управління внутрішнім сервісам.

Програмна логіка контролера підтримує виконання базових операцій (CRUD) та забезпечує обробку наступних запитів:

– Отримання детальної інформації про проєкт (getProjectDetail). Метод обробляє вхідні HTTP GET-запити. Він зчитує ідентифікатори групи та проєкту безпосередньо з URL-адреси як параметри шляху. Обов’язковість та непорожність цих параметрів суворо контролюється анотацією @NotBlank. Звернувшись до сервісного шару, метод повертає деталізовані дані у вигляді DTO-об’єкта ProjectDetailsDto.

– Створення нового проєкту (createProject). Для ініціалізації нового проєкту в базі даних метод приймає HTTP POST-запит. Окрім ідентифікатора цільової групи, метод очікує тіло запиту, яке автоматично перетворюється в об’єкт CreateProjectDto. Усі передані користувачем дані проходять перевірку завдяки анотації @Valid. Після успішного збереження контролер повертає клієнту базову інформацію про новостворений об’єкт.

– Оновлення даних проєкту (updateProject). Метод відповідає за модифікацію існуючого запису шляхом обробки HTTP PUT-запитів. Він ідентифікує цільовий проєкт за вказаними шляховими змінними та приймає оновлену конфігурацію у вигляді валідованого об’єкта UpdateProjectDto. Операція делегується сервісу і не передбачає повернення додаткових даних, сигналізуючи про успішність лише відповідним HTTP-статусом.

– Видалення проєкту (deleteProject). Для безпечного вилучення проєкту з бази даних використовується стандартний HTTP DELETE-запит. Метод приймає необхідні ідентифікатори, валідує їх на рівні контролера та передає відповідну команду до сервісного шару для остаточного видалення сутності.

Таким чином, розроблений клас надійно відокремлює логіку маршрутизації та валідації вхідних даних від безпосередньої бізнес-логіки застосунку. Він забезпечує стандартизований та безпечний REST-інтерфейс для управління проєктами, гарантуючи цілісність інформації ще на етапі її надходження до вебзастосунку.

Кодування модуля маршрутизації та API управління проєктами

Одним із ключових аспектів функціонування застосунку є можливість логічного об’єднання мікросервісів, для чого було реалізовано механізм управління групами. Для забезпечення доступу до цього функціоналу через програмний інтерфейс (API) розроблено спеціалізований контролер GroupController.

Як і у випадку з управлінням проєктами, цей модуль виконує роль вхідної точки для прийому та первинної обробки HTTP-запитів від клієнтського застосунку. Клас позначено анотацією @RestController, що вказує на його приналежність до REST-архітектури та забезпечує автоматичну серіалізацію об’єктів відповіді у формат JSON. Залежність від компонента бізнес-логіки ін’єктується автоматично завдяки анотації @RequiredArgsConstructor бібліотеки Lombok. Для забезпечення цілісності даних на рівні всього класу застосовано анотацію @Validated, яка активує механізм перевірки вхідних параметрів до моменту виконання безпосередньої логіки методів.

Програмна логіка контролера підтримує виконання базових операцій (CRUD) та забезпечує обробку наступних запитів:

- Отримання списку всіх груп (getGroups). Метод обробляє HTTP GET-запити за базовою адресою /group. Він звертається до сервісного шару та повертає клієнту колекцію об’єктів ShortenedGroupDto, що містять базову інформацію про всі наявні в базі даних групи. Використання спрощеного DTO дозволяє оптимізувати мережевий трафік під час виведення загального списку на головній панелі.
- Отримання детальної інформації про групу (getGroup). Для перегляду розширених даних про конкретну групу метод приймає HTTP GET-запит із вказаним ідентифікатором у шляху URL. Цей параметр проходить сувору валідацію на непорожність за допомогою анотації @NotBlank. У результаті успішного виконання клієнт отримує об’єкт GroupDto, що містить як параметри самої групи, так і пов’язані з нею проєкти.
- Створення нової групи (createGroup). Метод відповідає за реєстрацію нової групи у вебзастосунку шляхом обробки HTTP POST-запитів. Він приймає

вхідні дані з тіла запиту, які автоматично перетворюються на об’єкт CreateGroupDto. Завдяки анотації @Valid дані проходять перевірку на відповідність встановленим обмеженням. У відповідь метод повертає базову інформацію про щойно створену сутність.

– Оновлення даних групи (updateGroup). Операція модифікації існуючого запису реалізована через обробку HTTP PUT-запитів. Метод ідентифікує цільову групу за допомогою шляхової змінної та приймає оновлену конфігурацію з тіла запиту у вигляді об’єкта UpdateGroupDto, який також підлягає обов’язковій валідації. Метод делегує виконання операції сервісу і не повертає даних клієнту, сигналізуючи про успішне оновлення лише відповідним статусом HTTP-відповіді.

– Видалення групи (deleteGroup). Для безпечного вилучення логічної групи з бази даних метод обробляє стандартний HTTP DELETE-запит. Ідентифікатор цільової групи зчитується з URL-адреси, валідується на рівні контролера, після чого передається до сервісного шару для остаточного видалення сутності з бази даних.

Таким чином, розроблений клас надійно відокремлює логіку маршрутизації REST-запитів та валідації вхідної інформації від внутрішньої бізнес-логіки застосунку. Він надає клієнтській частині застосунку зрозумілий, безпечний та оптимізований інтерфейс для управління об’єктами груп.

Кодування модуля маршрутизації та API інтеграції з GitLab

Для забезпечення інтеграційної взаємодії клієнтської частини вебзастосунку із зовнішньою платформою управління вихідним кодом GitLab було розроблено спеціалізований контролер GitController. У межах загальної архітектури вебзастосунку даний модуль виконує функцію комунікаційного шлюзу. Його основним завданням є прийом керівних команд від користувацького інтерфейсу, безпечне зчитування авторизаційних даних та їх подальша маршрутизація до відповідного сервісного шару.

На рівні програмного коду клас позначено стандартною анотацією @RestController, що визначає його приналежність до веб-шару та забезпечує

автоматичну серіалізацію і десеріалізацію об’єктів у форматі JSON. Впровадження залежності компонента бізнес-логіки реалізовано через конструктор за допомогою анотації @RequiredArgsConstructor, тоді як валідація вхідних запитів на рівні класу гарантується використанням анотації @Validated.

Програмна логіка контролера інкапсулює обробку двох ключових HTTP-запитів, які є необхідними для дистанційного управління процесами розгортання:

– Отримання списку гілок репозиторію (getBranches). Метод призначений для обробки вхідних HTTP GET-запитів, що надходять від інформаційної панелі моніторингу. Ідентифікатори цільової групи та проєкту зчитуються безпосередньо з URL-адреси за допомогою анотації @PathVariable. Архітектурно важливою особливістю реалізації є механізм роботи з токеном безпеки: з метою запобігання компрометації конфіденційних даних через тіло чи параметри запиту, авторизаційний токен вилучається виключно з HTTP-заголовків за допомогою анотації @RequestHeader. Після отримання необхідних параметрів метод делегує виконання запиту сервісному шару та повертає клієнту масив доступних гілок у вигляді списку текстових рядків.

– Ініціалізація конвеєра розгортання (createPipeline). Для дистанційного запуску процесу CI/CD метод обробляє HTTP POST-запити. Окрім ідентифікаторів маршруту та токена авторизації із заголовків, метод приймає тіло запиту, яке автоматично десеріалізується в об’єкт передачі даних CreatePipelineDto. Безпосередня операція формування кінцевого запиту до віддаленої платформи делегується модулю GitService. При цьому метод не передбачає повернення додаткового набору даних клієнту, сигналізуючи про успішне прийняття команди на розгортання відповідним статус-кодом HTTP-відповіді.

Отже, розроблений програмний компонент надійно відокремлює логіку маршрутизації API-запитів від складної мережевої взаємодії з платформою GitLab. Застосований підхід гарантує високий рівень безпеки під час передачі токенів доступу та забезпечує фронтенд-застосунок зручним і стандартизованим

інтерфейсом для оперативного дистанційного запуску процесів розгортання мікросервісів.

Програмна реалізація шару об’єктів передачі даних

Для забезпечення безпечної та оптимізованої взаємодії між клієнтською частиною вебзастосунку та сервером було реалізовано шар об’єктів передачі даних (Data Transfer Object, DTO). Використання патерну DTO дозволяє відокремити внутрішні сутності бази даних від зовнішнього представлення, приховати деталі реалізації, а також мінімізувати обсяг даних, що передаються через мережу.

Для зменшення кількості шаблонного коду у всіх DTO-класах активно використовуються анотації бібліотеки Lombok. Для коректної серіалізації та десеріалізації об’єктів у формат JSON застосовуються анотації бібліотеки Jackson, зокрема @JsonProperty, яка чітко визначає назви полів у JSON-структурі.

Архітектурно об’єкти передачі даних у розробленому застосунку можна поділити на три основні категорії: для роботи з групами середовищ, для роботи з проєктами та допоміжні класи.

1. Класи для роботи з логічними групами середовищ:

- CreateGroupDto – об’єкт, що використовується для прийому даних від користувача під час створення нової логічної групи. Він містить поле name. Для забезпечення цілісності вхідних даних застосовується анотація @NotBlank, яка гарантує, що назва групи не може бути порожньою.
- UpdateGroupDto – об’єкт, призначений для оновлення параметрів існуючої групи.
- ShortenedGroupDto – базовий об’єкт для відправки даних клієнту. Містить лише ідентифікатор та назву. Використовується для побудови загальних списків в інтерфейсі, де детальна інформація не потрібна.
- GroupDto – розширений об’єкт, який наслідує властивості ShortenedGroupDto і додатково містить масив проєктів. Для коректної роботи патерну «Будівельник» при наслідуванні класів використано спеціальну

анотацію Lombok @SuperBuilder. Цей об’єкт повертається, коли користувач запитує повну інформацію про конкретну групу.

1. Класи для роботи з проєктами:

- CreateProjectDto – об’єкт для прийому даних від користувача під час реєстрації нового проєкту. Містить поля для початкової конфігурації: назву, цільову веб-адресу, а також параметри інтеграції. Поля назви та веб-адреси є обов’язковими, що контролюється анотацією @NotBlank.
- UpdateProjectDto – об’єкт, що використовується для операцій редагування існуючої конфігурації проєкту, дозволяючи безпечно оновити як базові адреси моніторингу, так і прив’язку до репозиторію GitLab.
- ProjectDto – об’єкт, що описує базовий стан проєкту всередині групи і відправляється на клієнтську частину (містить базові поля id, name, webAddress, а також необхідні для інтерфейсу параметри GitLab: gitProjectId, gitProjectUrl та goals).
- ProjectDetailsDto – спеціалізований об’єкт, розроблений виключно для підсистеми моніторингу телеметрії. Він формується сервісним шаром після успішного HTTP-опитування мікросервісу і містить фактичні версії технологій: javaVersion та springBootVersion.

2. Класи для інтеграції з GitLab API:

- BranchDto – об’єкт, що використовується для десеріалізації відповіді від GitLab API під час запиту списку доступних гілок репозиторію. Він містить поле name, яке в подальшому вилучається сервісом та передається на клієнтську частину у вигляді масиву рядків для вибору цільової гілки.
- CreatePipelineDto – об’єкт для ініціалізації конвеєра безперервного розгортання (CI/CD пайплайну). Він приймає від користувацького інтерфейсу назву обраної гілки та спеціальний параметр мети розгортання, які динамічно підставляються в параметри HTTP-запиту до платформи GitLab.

3. Допоміжні класи обробки помилок:

Для стандартизації відповідей сервера у випадку виникнення виняткових ситуацій реалізовано клас ErrorDto. Він містить єдине текстове поле message, яке

передає на сторону користувацького інтерфейсу зрозумілий опис причини помилки.

Таким чином, розроблена структура об’єктів передачі даних забезпечує гнучку маршрутизацію інформації, валідацію вхідних запитів та чітке розмежування між внутрішньою бізнес-логікою та зовнішніми інтерфейсами взаємодії.

Створення компонентів для роботи з проєктами

Програмний модуль Projects виступає центральним елементом клієнтського інтерфейсу для адміністрування розгорнутих мікросервісів у межах конкретної групи. Він надає адміністратору інструментарій для візуального контролю, конфігурування, безпечного вилучення та ініціалізації CI/CD конвеєрів через GitLab API. Вхідними даними для цього модуля є groupId, projects та showLoader. Ці властивості слугують для ідентифікації поточного середовища, передачі масиву цільових застосунків та контролю фази завантаження (що визначає, чи рендерити реальні показники, чи відображати тимчасові графічні заглушки під час обміну даними із сервером).

Для організації інтерактивної взаємодії застосовується кастомний хук useAppModal, який відповідає за виклик та керування станом діалогових вікон. Через нього ініціалізуються незалежні об’єкти, що забезпечують ізольований контекст для кожної операції – від зміни базових налаштувань URL до відправки команд на розгортання в систему контролю версій.

Архітектура модуля спирається на чітке розмежування логіки обробки дій та їхнього візуального представлення. Базовий компонент транслює дані до дочірнього ProjectsView, який генерує адаптивну сітку карток. Формування верхньої панелі кожної картки делеговано компоненту ProjectHeader. Саме в ньому інкапсульовано обробники кліків по елементах керування, які при активації викликають метод openModal відповідної структури, безпечно передаючи контекст безпосередньо до необхідного модального вікна.

З метою оптимізації сприйняття швидкодії вебзастосунку інтегровано інструментарій бібліотеки react-loading-skeleton. Аналізуючи булеве значення

showLoader, застосунок динамічно підміняє реальні дані на анімовані блоки-скелетони у заголовках та інформаційних полях. Після успішного вирішення мережевих запитів ці тимчасові компоненти безшовно замінюються на фактичний масив проєктів, усуваючи проблему раптового зміщення макета сторінки під час рендерингу.

Відповідні діалогові вікна активуються виключно за запитом користувача, дозволяючи виконувати критичні операції з мікросервісами, не перериваючи загальний процес моніторингу на головній панелі застосунку.

Своєю чергою, React-компонент ProjectDetails, наведений у додатку К, є ядром підсистеми автоматизованого моніторингу. Він генерує інтерфейс для відображення зібраної телеметрії віддаленого застосунку та приймає конфігураційні параметри groupId, projectId і маркер стану showLoader. Ці ідентифікатори необхідні для точного спрямування запитів до бекенду, тоді як маркер завантаження блокує передчасне виконання скриптів.

Внутрішній стан та синхронізація із сервером забезпечуються потужностями Redux Toolkit Query. Хук useGetGroupQuery вилучає еталонну версію системи, що закріплена за групою, тоді як useGetProjectDetailsQuery безпосередньо опитує віддалений мікросервіс для збору фактичних метрик. Використання конфігураційного параметра skip дозволяє призупинити ці виклики до повного завантаження батьківських компонентів, що раціонально розподіляє навантаження на мережу.

Візуальна частина компонента будується шляхом динамічного перебору властивостей об’єкта телеметрії за допомогою Object.entries. Зчитані пари «ключ-значення» маршрутизуються до допоміжного мікрокомпонента KeyValue. Для імплементації розумного підсвічування задіяна бібліотека classnames. Програмна логіка автоматично зіставляє еталонну версію застосунку із фактичною. У разі виявлення розбіжності версій, програма миттєво присвоює елементу CSS-клас text-warning, акцентуючи увагу адміністратора на застарілому вузлі яскравим сигнальним кольором.

Елементи ініціалізації дій на картках проєктів розроблені з урахуванням захисного програмування. Зокрема, кнопка старту пайплайну з’являється у DOM-дереві лише тоді, коли для проєкту задано дійсний gitProjectId, що запобігає спробам розгортання неповністю сконфігурованих сервісів. Візуальне оформлення кнопок базується на стилях Bootstrap та доповнюється семантичними SVG-іконками, які інтуїтивно зрозуміло відображають суть кожної функції.

Створення компонентів для роботи з групами

Програмні модулі Groups та Group виступають базовими елементами навігації та адміністрування логічних середовищ у клієнтському застосунку. Вони забезпечують `інструментарій для формування глобального бічного меню, перегляду загального списку груп, а також детального налаштування обраного середовища. Архітектура компонентів підтримує відображення вкладених маршрутів та керування станом завантаження, динамічно перемикаючи інтерфейс між візуальними заглушками та реальними конфігураційними даними залежно від фази обміну даними із сервером.

Для організації інтерактивної взаємодії та маршрутизації застосовується розгалужена система кастомних та вбудованих хуків. Бібліотека react-router-dom дозволяє зчитувати ідентифікатор активної групи безпосередньо з URL-адреси. Управління діалоговими вікнами делеговано хуку useAppModal, який ізольовано ініціалізує структури для виклику вікон створення нової групи, додавання мікросервісу, а також видалення поточного середовища чи конфігурації профілю користувача.

Взаємодія з серверною частиною та синхронізація станів забезпечуються потужностями Redux Toolkit Query. Компонент глобальної навігації Groups використовує хук useGetGroupsQuery для завантаження загального переліку середовищ. Своєю чергою, деталізоване подання ініціює запит useGetGroupQuery для зчитування розширеної інформації про конкретний об’єкт. Для миттєвого збереження змінених параметрів застосовується хук мутації

useUpdateGroupMutation, що дозволяє відправляти оновлені дані на бекенд асинхронно, без перезавантаження вебсторінки.

Візуальна архітектура застосунку розділена на глобальну навігаційну панель, побудовану на базі бібліотеки react-pro-sidebar, та центральну робочу область, куди через компонент Outlet динамічно ін’єктується вміст обраної групи. З метою оптимізації сприйняття швидкодії вебзастосунку інтегровано інструментарій react-loading-skeleton. Під час виконання мережевих запитів програма використовує масиви фіктивних об’єктів, рендерячи анімовані скелетони на місці навігаційних посилань та заголовків. Це ефективно усуває ефект раптового зміщення макета та забезпечує плавний візуальний перехід.

Окрема увага при розробці була приділена компоненту GroupView, який інкапсулює логіку редагування атрибутів групи без необхідності відкриття додаткових форм. Він використовує обгортку FormProvider з екосистеми react-hook-form. Налаштування полів відбувається через кастомний хук useAppForm, який автоматично синхронізується з отриманими даними через useEffect. Це рішення дозволяє адміністратору змінювати назву та цільову версію групи безпосередньо в тексті заголовка за допомогою полів EditableFormField. Збереження змін до бази даних ініціюється автоматично при втраті фокусу полем введення, що максимально спрощує процес адміністрування.

Елементи контекстного управління реалізовані через інтеграцію бібліотеки @szhsin/react-menu. Компонент GroupHeader містить компактне випадаюче меню, яке дозволяє примусово оновити метрики телеметрії, викликати модальне вікно створення проєкту або ініціювати видалення поточної групи. Інтерфейс меню та кнопок стилізовано за допомогою CSS-фреймворку Bootstrap, а використання семантичних SVG-іконок покращує візуальну ієрархію. Крім того, застосовано патерн захисного рендерингу: у разі відсутності зареєстрованих мікросервісів у групі, застосунок виводить графічну підказку BackgroundHint замість порожньої сітки, інтуїтивно спрямовуючи користувача до наступної логічної дії.

Тестування функціоналу

Для перевірки працездатності розробленого вебзастосунку було проведено функціональне тестування ключових програмних модулів. Основну увагу приділено перевірці інтеграції з GitLab API та обробці виняткових ситуацій.

Тест 1. Перевірка інтеграції з GitLab API (отримання гілок)

- Дія: Натиснути кнопку ініціалізації розгортання (іконка ракети) на картці створеного проєкту.
- Очікуваний результат: Відкривається модальне вікно «Запустити pipeline». Застосунок успішно звертається до GitLab API, використовуючи збережений токен, та завантажує випадаючий список актуальних гілок репозиторію.

Тест 2. Перевірка ініціації CI/CD пайплайну

- Дія: У модальному вікні запуску пайплайну обрати потрібну гілку коду, вказати ціль розгортання та натиснути кнопку «ЗАПУСТИТИ».
- Очікуваний результат: Серверна частина успішно формує та відправляє POST-запит до GitLab. Користувач отримує Toast-сповіщення про успішний старт. В інфраструктурі GitLab ініціюється відповідний процес автоматичної збірки.

Тест 3. Обробка аварійних ситуацій (негативний тест)

- Дія: Вказати для проєкту недійсну веб-адресу (Url) та ініціювати збір телеметрії.
- Очікуваний результат: Застосунок коректно перехоплює помилку з'єднання, не перериваючи роботу застосунку загалом, і виводить на інтерфейс користувача повідомлення про неможливість отримати дані для конкретного сервісу.

3. РЕЗУЛЬТАТИ РОЗРОБКИ ВЕБЗАСТОСУНКУ ДЛЯ МОНІТОРИНГУ ВЕРСІЙНОСТІ РОЗГОРНУТИХ ЗАСТОСУНКІВ ТА ЇХ РОЗГОРТАННЯ ЗА ДОПОМОГОЮ GITLAB API ТА GITLAB CI

Розроблений вебзастосунок для моніторингу версійності розгорнутих застосунків та їх розгортання за допомогою GitLab API та Gitlab CI є односторінковим, що забезпечує швидку та плавну взаємодію з користувачем без перезавантаження сторінок у браузері. Клієнтська частина реалізована з використанням бібліотеки React та мови TypeScript. Для забезпечення навігації між представленнями застосовується бібліотека React Router.

Візуальне оформлення інтерфейсу користувача побудовано за принципом інформаційної панелі з використанням CSS-фреймворку Bootstrap, що гарантує адаптивність та коректне відображення елементів керування.

Головне вікно застосунку логічно розділене на дві основні зони: статичну бічну панель навігації та динамічну центральну робочу область. Зовнішній вигляд головної сторінки застосунку в початковому стані наведено на рисунку 3.1. У центральній частині екрана при цьому виводиться фонове інформаційне повідомлення «Виберіть групу проєктів...».

Ліва бічна панель слугує основним елементом навігації. Вона має темне контрастне оформлення та містить кнопку «Створити групу проєктів», під якою формується перелік усіх доступних логічних груп середовищ, завантажених із сервера.

При натисканні на назву конкретної групи в бічному меню відбувається зміна вмісту центральної робочої області, де відображається детальна інформація про обране середовище та пов’язані з ним розгорнуті застосунки (рис. 3.2).

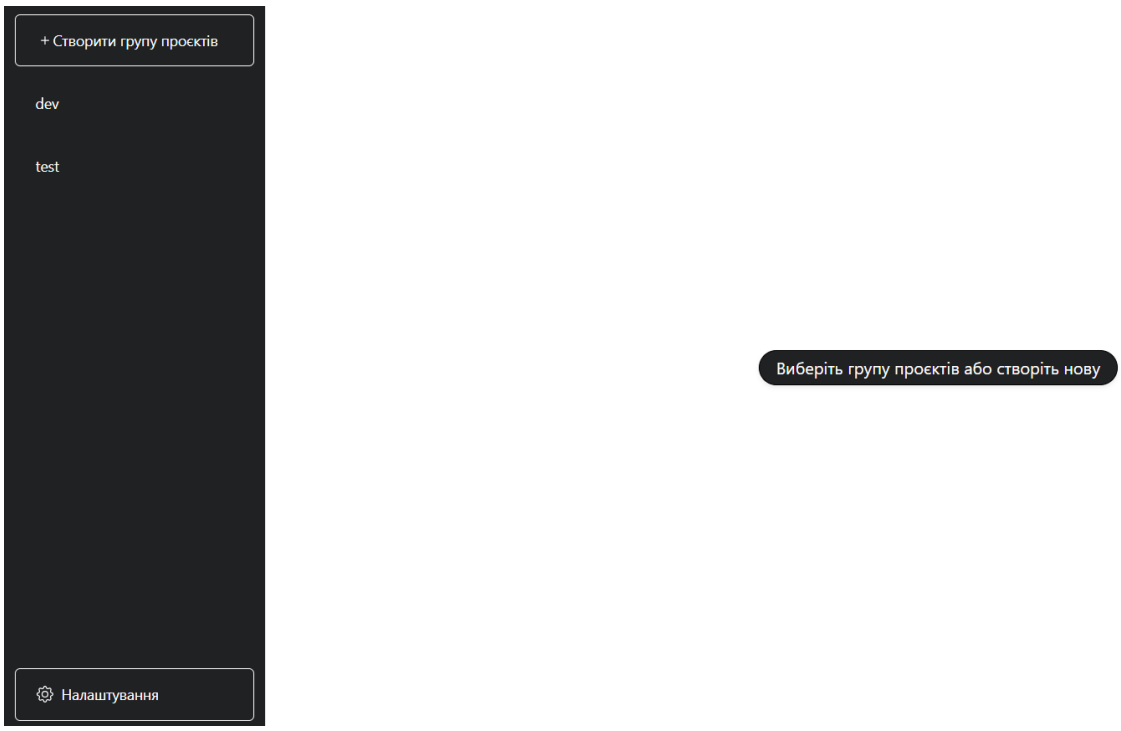


Рисунок 3.1 – Головна сторінка вебзастосунку для моніторингу версійності розгорнутих застосунків та їх розгортання за допомогою GitLab API та Gitlab CI

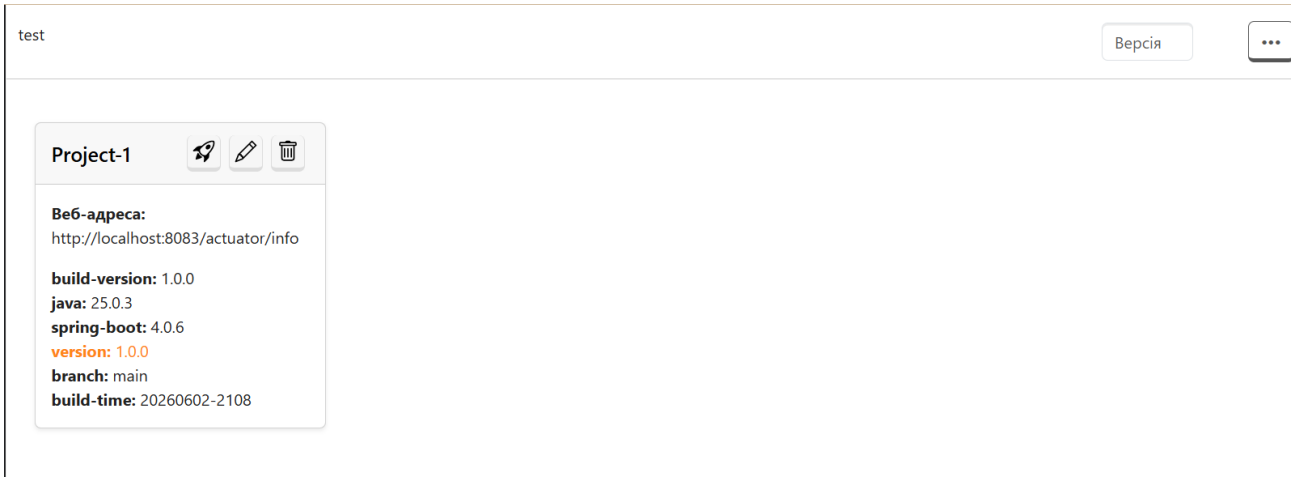


Рисунок 3.2 – Робоча область обраної групи з переліком проєктів

У верхній частині робочої області відображається назва обраної групи. Завдяки компоненту EditableFormField реалізовано можливість її швидкого редагування безпосередньо в тексті. Поруч розміщено кнопку виклику контекстного меню дій, яке дозволяє створити новий проєкт у цій групі або повністю видалити її.

Основну частину екрана займає сітка інформаційних карток розгорнутих проєктів (мікросервісів). Кожна картка відображає:

- назву проєкту з кнопками управління (редагування та видалення);
- цільову веб-адресу мікросервісу;
- параметри телеметрії.

Для збору телеметрії клієнтський застосунок виконує асинхронні запити до сервера. Під час очікування відповіді в картках проєктів відображаються анімовані елементи завантаження (патерн Skeleton), що покращує користувацький досвід та запобігає візуальному блокуванню інтерфейсу. Після успішного отримання даних на екрані виводяться актуальні версії технологій (Java та Spring Boot) для кожного мікросервісу.

Для виконання операцій додавання, редагування та видалення сутностей (груп і проєктів) використовуються модальні вікна. Такий підхід дозволяє концентрувати увагу користувача на поточній задачі. Макет вікна конфігурації проєкту наведено на рисунку 3.3.

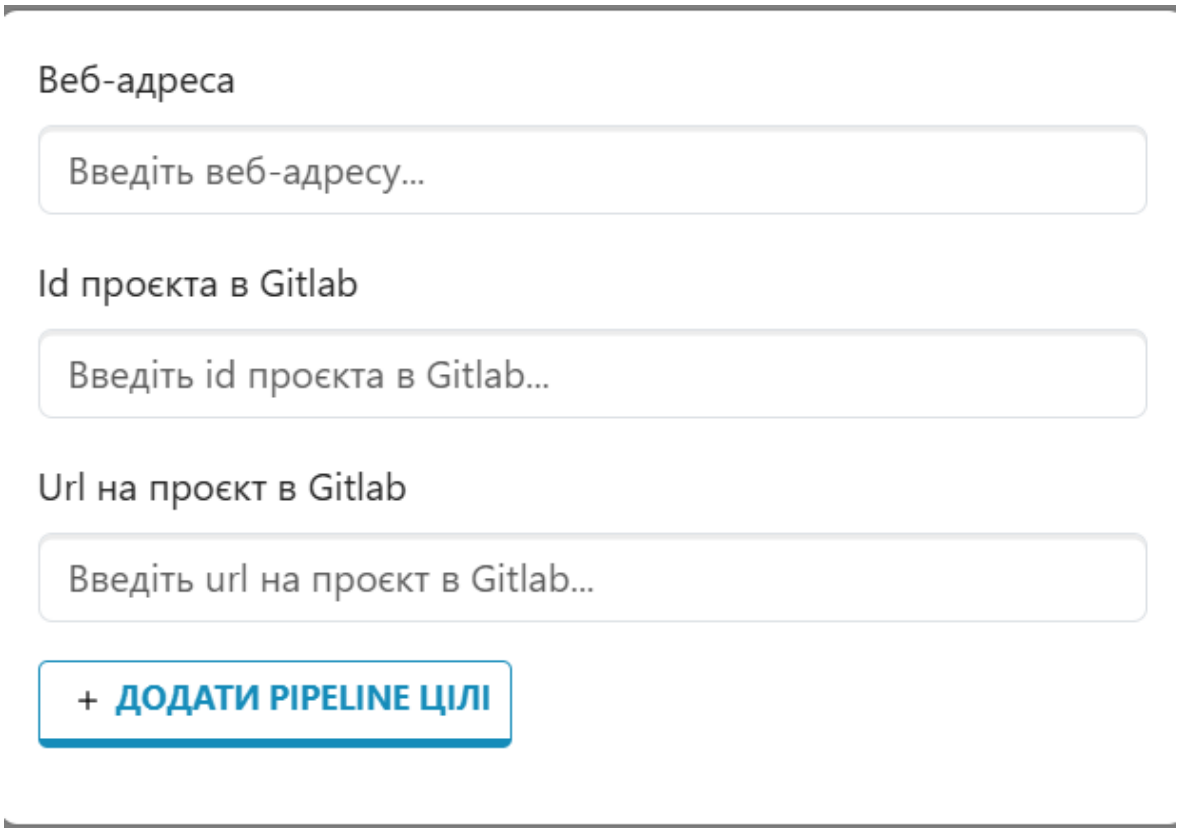


Рисунок 3.3 – Модальне вікно додавання проєкту

Для забезпечення можливості оперативного розгортання застосунків безпосередньо з робочої області проєкту було реалізовано підсистему запуску CI/CD конвеєрів. Функціонал ініціалізується натисканням на відповідну іконку ракети на картці проєкту, що відкриває модальне вікно запуску пайплайну, макет якого наведено на рисунку 3.4.

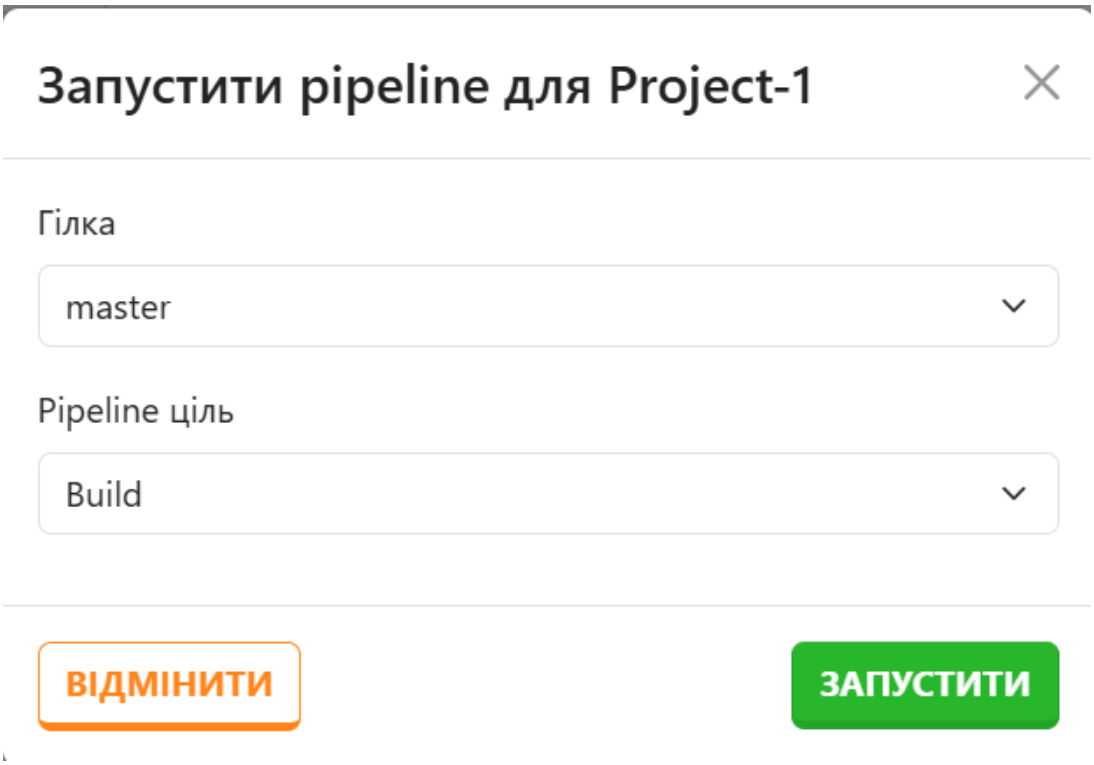


Рисунок 3.4 – Інтерфейс вікна запуску CI/CD конвеєра

Інтерфейс вікна запуску пайплайну передбачає використання випадаючих списків для вибору цільових параметрів розгортання:

– Вибір гілки (Branch): Вебзастосунок автоматично звертається до GitLab API для отримання актуального переліку гілок репозиторію обраного проєкту. Це дозволяє адміністратору обрати саме ту версію коду, яка підлягає розгортанню.

– Вибір Pipeline цілі (Goal): Цей елемент інтерфейсу дозволяє визначити етап конвеєра, який необхідно ініціювати. Дані для вибору формуються на основі конфігурації проєкту, збереженої в базі даних.

Взаємодія з GitLab відбувається в асинхронному режимі. Після того, як користувач обирає параметри та натискає кнопку «Запустити», клієнтська частина відправляє POST-запит до серверної частини системи. Бекенд, використовуючи автентифікаційний токен, формує відповідний запит до GitLab API з передачею необхідних змінних середовища.

У разі успішного прийняття запиту платформою GitLab, програма виводить сповіщення про успішний старт розгортання. Для відстеження результатів виконання конвеєра, користувач може перейти до відповідного розділу Pipelines у веб-інтерфейсі GitLab за посиланням, що доступне в деталях проєкту, де відображається статус виконання та логи кожного етапу збірки. Такий підхід забезпечує повний цикл автоматизованого управління версіями: від виявлення застарілих сервісів у вебзастосунку до їх оперативного оновлення через CI/CD конвеєри.

Форми введення даних побудовані з використанням бібліотеки react-hook-form, що забезпечує миттєву валідацію обов’язкових полів на стороні клієнта. Усі результати дій користувача, включаючи повідомлення про успішне збереження даних або виникнення помилок на сервері, супроводжуються спливаючими сповіщеннями (Toast-повідомленнями) у куті екрана.

4. ОХОРОНА ПРАЦІ

Процес розробки вебзастосунку для моніторингу версійності розгорнутих застосунків та їх розгортання за допомогою GitLab API та Gitlab CI відбувається у стандартних умовах офісного приміщення з використанням засобів обчислювальної техніки. Оскільки основна діяльність інженера-програміста передбачає тривалу та інтенсивну роботу за екраном монітора, виникає об'єктивна необхідність у ретельній організації безпечного та комфортного робочого середовища. Головна мета цього розділу полягає в аналізі умов праці на робочому місці, виявленні потенційних небезпек та шкідливих виробничих факторів, а також у формуванні комплексу профілактичних заходів для їх усунення згідно з вимогами Закону України «Про охорону праці» [20].

4.1 Характеристика робочого місця та аналіз шкідливих виробничих факторів при роботі з комп'ютерною технікою

Процес розробки вебзастосунку для моніторингу версійності розгорнутих застосунків та їх розгортання за допомогою GitLab API та Gitlab CI відбувається у стандартних умовах офісного приміщення з використанням засобів обчислювальної техніки. Оскільки основна діяльність інженера-програміста передбачає тривалу та інтенсивну роботу за екраном монітора (до 8 годин на добу), виникає об'єктивна необхідність у ретельній організації безпечного та комфортного робочого середовища.

Робоче місце програміста облаштоване в приміщенні з площею близько 20 м² та об'ємом 60 м³, що цілком задовольняє діючі санітарні норми для роботи з персональними комп'ютерами. Робоча зона укомплектована письмовим столом, ергономічним кріслом із підтримкою поперекового відділу хребта, а також сучасним системним блоком та рідкокристалічним дисплеєм. Додатково використовуються периферійні пристрої введення: клавіатура та оптична миша.

Робота програміста належить до категорії робіт, що характеризуються високим рівнем нервово-емоційного напруження, значним навантаженням на зоровий аналізатор та тривалою гіподинамією (нестачею рухової активності). Відповідно до положень НПАОП 0.00-7.15-18 «Вимоги щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями», під час проєктування архітектури системи, написання та зневадження коду на працівника може впливати низка небезпечних та шкідливих виробничих факторів [21]. Згідно із загальноприйнятою класифікацією, ці фактори поділяються на фізичні та психофізіологічні. Фізичні небезпечні та шкідливі фактори:

- Невідповідність параметрів мікроклімату: тривале перебування у закритому приміщенні з комп'ютерною технікою може призводити до підвищення температури повітря, зниження вологості та накопичення вуглекислого газу. Це спричиняє швидку втомлюваність, зниження концентрації уваги та головний біль.
- Недостатня або нерівномірна освітленість робочої зони: відсутність належного природного або штучного освітлення, а також поява відблисків від вікон на екрані монітора змушують працівника напружувати зір.
- Електромагнітне та електростатичне випромінювання: працююча комп'ютерна техніка є джерелом електромагнітних полів широкого спектра частот. Тривалий вплив цих полів без належного захисту може негативно позначатися на імунній та нервовій системах.
- Підвищений рівень фонового шуму: постійний шум від систем охолодження системних блоків, серверного обладнання та кондиціонерів створює монотонне акустичне навантаження, яке з часом знижує працездатність.
- Небезпека ураження електричним струмом: оскільки вся обчислювальна техніка працює від електромережі з напругою 220 В, існує ризик ураження струмом у разі випадкового пошкодження ізоляції кабелів живлення або несправності блоків живлення.

Психофізіологічні шкідливі фактори:

– Перенапруження зорового аналізатора: специфіка розробки (читання дрібного програмного коду, аналіз логів сервісів, пошук помилок) вимагає постійної фіксації погляду на екрані, що призводить до синдрому «сухого ока» та спазму акомодації.

– Нервово-емоційне напруження: обробка великих обсягів складної інформації, необхідність утримання в пам'яті архітектурних зв'язків розроблюваного застосунку та жорсткі терміни виконання завдань (дедлайни) викликають високий рівень стресу.

– Статичне навантаження на опорно-руховий апарат: тривала робота в сидячій позі призводить до застою крові в органах малого таза, перенапруження м'язів шийного та поперекового відділів хребта. Це є основною причиною розвитку остеохондрозу та радикуліту у фахівців ІТ-сфери.

4.2 Заходи щодо нормалізації умов праці при роботі з комп'ютерною технікою

З метою нейтралізації виявлених шкідливих виробничих факторів та збереження здоров'я і високої працездатності інженера-програміста, на робочому місці впроваджується комплекс організаційних, технічних та санітарно-гігієнічних заходів. Нормалізація параметрів мікроклімату.

Робота за комп'ютером відноситься до категорії легкої фізичної праці згідно з Державними санітарними нормами ДСН 3.3.6.042-99 [22]. Для забезпечення комфортного теплового стану працівника приміщення обладнане припливно-витяжною вентиляцією та системою кондиціонування повітря. Встановлено такі оптимальні параметри мікроклімату:

– температура повітря: +22…24°С у холодний період року та +23…25°С у теплий;

– відносна вологість повітря: на рівні 40–60%;

– швидкість руху повітря: не перевищує 0,1 м/с. Регулярне наскрізне провітрювання приміщення дозволяє уникнути накопичення вуглекислого газу

та підтримувати необхідний рівень кисню в повітрі, що запобігає швидкій втомлюваності.

Забезпечення раціонального освітлення.

Правильне освітлення є критично важливим для профілактики зорової втоми. У приміщенні застосовується система комбінованого освітлення: природне світло доповнюється загальним рівномірним штучним освітленням. Згідно з ДБН В.2.5-28:2018, мінімально допустима освітленість на рівні робочого столу становить 400 лк [23]. Для штучного освітлення використовуються сучасні LED-панелі зі світловою температурою 4000К (нейтральне біле світло), які не створюють ефекту мерехтіння. Робочий стіл розміщено перпендикулярно до вікон, щоб природне світло падало зліва або спереду. На вікнах встановлені вертикальні жалюзі для захисту від прямих сонячних променів та усунення відблисків на екрані монітора.

Захист від шуму та електромагнітного випромінювання. Рівень звукового тиску на робочому місці жорстко контролюється і, відповідно до вимог ДСН 3.3.6.037-99, не перевищує допустимої норми у 50 дБА [24]. Зниження шуму досягається за рахунок використання сучасних безшумних систем охолодження системного блока, ізоляції серверного обладнання в окремому приміщенні та застосування звукопоглинальних матеріалів у оздобленні стін офісу. Сучасні рідкокристалічні (LCD/IPS) монітори практично не випромінюють небезпечних електромагнітних хвиль. Однак для мінімізації будь-якого впливу системний блок розташований на максимальному віддаленні від працівника, а кабелі живлення акуратно зібрані в органайзери.

Ергономіка робочого місця та режим праці.

Для запобігання захворюванням опорно-рухового апарату робоче місце спроєктовано за принципами ергономіки:

– висота робочої поверхні столу становить 720 мм, що дозволяє зручно розмістити руки на клавіатурі;

- робоче крісло має функцію регулювання висоти сидіння, кута нахилу спинки та оснащене підлокітниками, що суттєво знімає статичну напругу з м'язів плечового пояса та хребта;
- екран монітора розташований на відстані 60–70 см від очей користувача, а його верхній край знаходиться на рівні очей або трохи нижче. Для зняття нервово-емоційного та зорового напруження впроваджено раціональний режим праці та відпочинку. Згідно з санітарними правилами, після кожних 2 годин безперервного написання коду передбачена регламентована перерва тривалістю 15 хвилин. Під час перерви програмісту рекомендується виконувати комплекс вправ для очей та коротку гімнастику для розслаблення м'язів шиї і спини.

4.3 Електробезпека при роботі з комп'ютерною технікою

Відповідно до вимог Правил улаштування електроустановок, робоче місце інженера-програміста обладнується трипровідною системою електроживлення із захисним заземленням [25]. Усі системні блоки, монітори та периферійні пристрої підключаються до мережі через сертифіковані мережеві фільтри із захистом від перенапруги. Для забезпечення безпеки експлуатації техніка регулярно перевіряється на цілісність ізоляції кабелів. З метою запобігання травматизму забороняється використання пошкоджених розеткових блоків та саморобних подовжувачів.

Для забезпечення таких умов виконаємо розрахунок загального рівномірного штучного освітлення приміщення методом коефіцієнта використання світлового потоку. Цей метод застосовується для розрахунку загального рівномірного освітлення горизонтальних поверхонь при відсутності великих затінюючих предметів.

Вихідні дані для розрахунку:

- Габарити приміщення: довжина A = 5 м; ширина B = 4 м; висота H = 3 м.
- Площа приміщення: $S = A * B = 5 * 4 = 20$ м².

- Характеристики поверхонь (світле офісне приміщення): коефіцієнт відбиття стелі ρст = 70%; коефіцієнт відбиття стін ρс = 50%; коефіцієнт відбиття підлоги ρп = 10%.
- Висота робочої поверхні столу від підлоги hp = 0,8 м.
- Висота звису світильника від стелі hc = 0 м (світильники вмонтовані безпосередньо у підвісну стелю типу «Армстронг»).

Визначимо розрахункову висоту підвісу світильника над робочою поверхнею (h): h = H - hp - hc = 3,0 - 0,8 - 0 = 2,2 м.

Обчислимо індекс приміщення (i), який враховує геометрію кімнати: i = (A × B) / (h × (A + B)) = (5 × 4) / (2,2 × 9) = 20 / 19,8 ≈ 1,01.

Використовуючи довідкові таблиці для світильників із рівномірним розсіюванням світла (враховуючи індекс приміщення i ≈ 1,0 та задані коефіцієнти відбиття), знаходимо коефіцієнт використання світлового потоку: η = 46% (або 0,46).

Для розрахунку обираємо сучасні енергоефективні LED-панелі (світлодіодні світильники розміром 600х600 мм), які не створюють шкідливого для очей мерехтіння. Світловий потік одного такого світильника становить Fсв = 3200 лм.

Додаткові розрахункові коефіцієнти:

- kз = 1,5 – коефіцієнт запасу;
- Z = 1,1 – коефіцієнт нерівномірності освітлення.

Визначимо необхідну кількість світильників (N): N = (Ен × S × kз × Z) / (Fсв × η) = (400 × 20 × 1,5 × 1,1) / (3200 × 0,46) = 13200 / 1472 ≈ 8,96 шт.

З міркувань симетрії приймаємо до встановлення 9 світильників (розміщення у вигляді матриці 3х3).

Перевіримо фактичну освітленість (Еф), яку забезпечать 9 обраних світильників: Еф = (N × Fсв × η) / (S × kз × Z) = (9 × 3200 × 0,46) / (20 × 1,5 × 1,1) = 13248 / 33 ≈ 401,45 лк.

Визначимо похибку розрахунку: Δ = ((Еф - Ен) / Ен) × 100% = ((401,45 - 400) / 400) × 100% = +0,36%.

Висновок до розрахунку: Отримана похибка (+0,36%) знаходиться в межах допустимих норм (від -10% до +20%). Встановлення 9 світлодіодних LED-панелей повністю забезпечить нормативну освітленість понад 400 лк, що гарантує комфортні умови праці.

4.4 Пожежна безпека при роботі з комп'ютерною технікою

Через велику кількість пластикових компонентів, ізоляційних матеріалів та паперових носіїв приміщення розробників відноситься до категорії пожежної небезпеки «В». З метою запобігання загорянням реалізуються вимоги Правил пожежної безпеки в Україні [26].

Організаційний напрямок включає обов'язкове проведення протипожежних інструктажів, затвердження планів евакуації та призначення осіб, відповідальних за пожежну безпеку. Технічний напрямок передбачає обладнання приміщення системами автоматичної пожежної сигналізації з тепловими та димовими сповіщувачами. Зважаючи на те, що в кімнаті знаходиться електронно-обчислювальна техніка під напругою, використання пінних чи водяних засобів пожежогасіння категорично заборонено. Для ліквідації можливих осередків займання робочі місця укомплектовано переносними вуглекислотними вогнегасниками, які є діелектриками та не пошкоджують мікросхеми серверів і комп'ютерів.

ВИСНОВКИ

У кваліфікаційній роботі вирішена актуальна практична задача інженерії програмного забезпечення, що характеризується комплексністю та невизначеністю умов, яка полягає у створенні спеціалізованого вебзастосунку для централізованого моніторингу, управління версійністю розгорнутих застосунків (мікросервісів) та автоматизації процесів їх розгортання. Розроблений програмний продукт дозволяє ефективно контролювати стан програмної інфраструктури, оптимізувати роботу DevOps-інженерів та забезпечити прозорість життєвого циклу програмного забезпечення.

На початкових етапах виконання роботи було проведено детальний аналіз предметної області та існуючих підходів до моніторингу мікросервісних архітектур. На основі зібраних даних було спроєктовано багаторівневу архітектуру програми та розроблено ергономічну візуальну модель користувацького інтерфейсу. Продумана система навігації, використання панельного макета (Dashboard) та інтеграція системи модальних вікон дозволили створити інтуїтивно зрозуміле середовище для роботи з великими масивами конфігураційних даних без перевантаження робочого простору користувача.

Програмну реалізацію серверної частини (бекенду) виконано із застосуванням сучасних інструментів та технологій екосистеми Java. Розроблено надійний шар бізнес-логіки, який забезпечує управління логічними групами середовищ та конфігураціями проєктів. Успішно впроваджено механізм активного мережевого опитування віддалених мікросервісів для збору телеметрії, що дозволяє вебзастосунку в режимі реального часу отримувати та аналізувати дані про актуальні версії технологій (зокрема, Java та Spring Boot). Використання патернів об’єктів передачі даних (DTO) гарантувало безпеку та оптимізацію мережевого обміну інформацією.

Клієнтську частину вебзастосунку (фронтенд) реалізовано у вигляді сучасного односторінкового застосунку (SPA) з використанням бібліотеки React та суворо типізованої мови TypeScript. Впровадження передових бібліотек для управління станом (Redux Toolkit) та кешування запитів дозволило забезпечити

високу швидкість відгуку інтерфейсу, асинхронне оновлення даних моніторингу без перезавантаження сторінок та коректну обробку мережевих затримок за допомогою візуальних патернів завантаження.

Значною перевагою роботи стала інтеграція розробленої програми з платформою GitLab. Використання можливостей GitLab API дозволило налагодити автоматизовану взаємодію з репозиторіями вихідного коду, а налаштування конвеєрів безперервної інтеграції та розгортання (GitLab CI) забезпечило автоматизацію процесів доставки нових версій застосунків у цільові середовища. Це об’єднало етапи розробки, розгортання та моніторингу в єдиний безперервний цикл.

У результаті виконання кваліфікаційної роботи отримано повністю функціональний, стабільний та масштабований програмний продукт. Усі поставлені на початку роботи завдання виконано в повному обсязі. Розроблений програмний продукт відповідає сучасним вимогам до інструментів моніторингу та повністю готова до впровадження у процеси розробки і супроводу програмного забезпечення.

Під час виконання роботи було розглянуто питання охорони праці та безпеки в надзвичайних ситуаціях. Проведено аналіз умов на робочому місці інженера-програміста та виявлено потенційні небезпечні й шкідливі виробничі фактори. Розроблено комплекс профілактичних заходів для їх мінімізації, зокрема виконано розрахунок системи штучного освітлення, який підтвердив її відповідність чинним державним санітарним нормам. Також визначено базові вимоги щодо забезпечення електро- та пожежної безпеки в приміщенні розробки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. DevOps Monitoring: Best Practices and Tools. Datadog. URL: https://www.datadoghq.com/devops-monitoring/ (дата звернення: 13.04.2026).

2. Continuous integration vs. delivery vs. deployment. Atlassian. URL: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment (дата звернення: 12.04.2026).

3. What is CI/CD? GitLab. URL: https://about.gitlab.com/topics/ci-cd/ (дата звернення: 12.04.2026).

4. GitLab CI/CD: Continuous Integration & Delivery. GitLab Docs. URL: https://docs.gitlab.com/ee/ci/ (дата звернення: 14.04.2026).

5. Spring Boot Admin Reference Guide. Codecentric. URL: https://codecentric.github.io/spring-boot-admin/current/ (дата звернення: 15.04.2026).

6. Grafana Documentation: Dashboards and Visualizations. Grafana Labs. URL: https://grafana.com/docs/ (дата звернення: 15.04.2026).

7. UML Specification. Object Management Group (OMG). URL: https://www.omg.org/spec/UML/ (дата звернення: 20.04.2026).

8. Мельник А. О., Коваленко О. С. Архітектура сучасних вебзастосунків : підручник. Київ : КПІ, 2025. 240 с.

9. The MongoDB Manual. MongoDB Documentation. URL: https://www.mongodb.com/docs/manual/ (дата звернення: 23.04.2026).

10. Spring Data MongoDB Reference Guide. Spring. URL: https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/ (дата звернення: 29.04.2026).

11. Spring Boot Reference Documentation. Spring. URL: https://docs.spring.io/spring-boot/docs/current/reference/html/ (дата звернення: 22.04.2026).

12. REST API Tutorial. REST API. URL: https://restfulapi.net/ (дата звернення: 25.04.2026).

13. React – The library for web and native user interfaces. React Docs. URL: https://react.dev/ (дата звернення: 26.04.2026).

14. TypeScript: Typed JavaScript at Any Scale. TypeScript Documentation. URL: https://www.typescriptlang.org/docs/ (дата звернення: 02.05.2026).

15. Redux Toolkit: The official, opinionated, batteries-included toolset for efficient Redux development. Redux. URL: https://redux-toolkit.js.org/ (дата звернення: 05.05.2026).

16. React Router: Declarative routing for React. React Router Docs. URL: https://reactrouter.com/ (дата звернення: 06.05.2026).

17. React Bootstrap: The most popular front-end framework, rebuilt for React. React Bootstrap. URL: https://react-bootstrap.netlify.app/ (дата звернення: 28.04.2026).

18. Get Started | React Hook Form. React Hook Form. URL: https://react-hook-form.com/get-started (дата звернення: 04.05.2026).

19. GitLab API Documentation: REST API. GitLab Docs. URL: https://docs.gitlab.com/ee/api/ (дата звернення: 11.04.2026).

20. Про охорону праці: Закон України від 14.10.1992 № 2694-XII. Редакція від 12.09.2025. URL: https://zakon.rada.gov.ua/laws/show/2694-12 (дата звернення: 18.04.2026).

21. Про затвердження Вимог щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями: Наказ Міністерства соціальної політики України від 14.02.2018 № 207. URL: https://zakon.rada.gov.ua/laws/show/z0508-18 (дата звернення: 18.04.2026).

22. ДСН 3.3.6.042-99. Санітарні норми мікроклімату виробничих приміщень : Затв. постановою Головного державного санітарного лікаря України від 01.12.1999 № 42. URL: https://zakon.rada.gov.ua/rada/show/va042282-99 (дата звернення: 19.04.2026).

23. ДБН В.2.5-28:2018. Природне і штучне освітлення : Наказ Міністерства регіонального розвитку, будівництва та житлово-комунального господарства України від 03.10.2018 № 264. URL: https://e-

construction.gov.ua/laws_detail/3074958732556240833 (дата звернення: 19.04.2026).

24. ДСН 3.3.6.037-99. Санітарні норми виробничого шуму, ультразвуку та інфразвуку : Затв. постановою Головного державного санітарного лікаря України від 01.12.1999 № 37. URL: https://zakon.rada.gov.ua/rada/show/va037282-99 (дата звернення: 20.04.2026).

25. Про затвердження Правил улаштування електроустановок : Затв. наказом Міненерговугілля України від 21.07.2017 № 476. URL: https://zakon.rada.gov.ua/rada/show/v0476732-17 (дата звернення: 21.04.2026).

26. Про затвердження Правил пожежної безпеки в Україні : Затв. наказом МВС України від 30.12.2014 № 1417. Редакція від 27.02.2026. URL: https://zakon.rada.gov.ua/laws/show/z0252-15 (дата звернення: 22.04.2026).

ДОДАТОК А – ТЕХНІЧНЕ ЗАВДАННЯ НА РОЗРОБКУ ВЕБЗАСТОСУНКУ

Вступ

Найменування програмного забезпечення: Вебзастосунок для моніторингу та управління версійністю розгорнутих застосунків та їх розгортання за допомогою GitLab API та Gitlab CI.

Коротка характеристика області застосування: Інформаційні технології, автоматизація процесів розробки (DevOps), безперервна інтеграція та розгортання (CI/CD), моніторинг стану серверної інфраструктури та управління мікросервісними архітектурами.

1. Підстави для розробки

Розробка виконується на основі завдання на кваліфікаційну роботу на здобуття ступеня вищої освіти «бакалавр» (спеціальність 121 «Інженерія програмного забезпечення»). Підставою для розробки є необхідність вирішення проблеми фрагментації даних про стан розгортання та відсутності готових інструментів, що поєднують безперервний моніторинг реальних версій застосунків із можливістю їх оперативного дистанційного оновлення.

2. Призначення програмного забезпечення

Вебзастосунок призначений для створення інтеграційної платформи, яка забезпечує єдину точку доступу до інформації про стан CI/CD процесів та фактичні технічні параметри розгорнутих мікросервісів.

Цілі створення вебзастосунку:

- Автоматизований збір телеметрії про версії технологій розгорнутих застосунків у режимі реального часу.
- Швидке виявлення застарілих сервісів.
- Оперативне ініціювання оновлень безпосередньо з інформаційної панелі моніторингу з використанням інтеграції з GitLab API.

3. Вимоги до програмного забезпечення

3.1. Вимоги до функціональних характеристик:

Застосунок повинен забезпечувати виконання наступних функцій:

- Управління логічними середовищами (групами): створення, перегляд, редагування та видалення логічних груп проєктів.
- Управління проєктами (мікросервісами) всередині груп: реєстрація нових застосунків із зазначенням їхньої цільової веб-адреси та конфігураційних реквізитів у GitLab.
- Автоматичний моніторинг шляхом регулярного виконання HTTP-запитів до розгорнутих сервісів для збору метаданих (версія застосунку, версія Java, версія Spring Boot).
- Автоматичне порівняння цільової (еталонної) версії проєкту з фактичною версією розгорнутого мікросервісу з візуальною індикацією розбіжностей.
- Взаємодія з GitLab API: отримання актуального списку гілок репозиторію.
- Дистанційний запуск CI/CD пайплайнів для обраного проєкту безпосередньо з інтерфейсу вебзастосунку.

3.2. Вимоги до надійності та безпеки:

- Вебзастосунок повинен коректно обробляти помилки мережевої взаємодії (наприклад, тимчасова недоступність віддаленого мікросервісу) без аварійного завершення роботи застосунку, виводячи відповідні повідомлення про помилки на клієнтській панелі.
- Авторизація запитів до платформи GitLab повинна здійснюватися виключно через безпечну передачу приватних токенів у HTTP-заголовках запитів, уникаючи їх передачі у відкритому вигляді або через URL-параметри.
- Усі вхідні дані від користувача (назви груп, URL-адреси, ідентифікатори) повинні проходити сувору валідацію на рівні клієнта та сервера перед збереженням до бази даних.

4. Вимоги до апаратного та програмного забезпечення

4.1. Вимоги до серверної частини (Backend):

– Мова програмування: Java.

– Фреймворк: Spring Boot.

– Система управління базами даних: MongoDB (документо-орієнтована NoSQL).

– Мінімальні апаратні вимоги до сервера: двоядерний процесор, від 4 ГБ оперативної пам'яті, від 20 ГБ вільного дискового простору.

4.2. Вимоги до клієнтської частини (Frontend):

– Технологічний стек: React, TypeScript, Redux Toolkit, Vite, React Router, React Bootstrap.

– Клієнтська частина повинна працювати як односторінковий застосунок (SPA) і бути кросплатформною.

– Вимоги до пристрою користувача: наявність сучасного веббраузера.

5 Вимоги до програмної документації

Склад програмної документації повинен включати в себе:

- технічне завдання;
- програму та методику випробувань;
- керівництво користувача;
- опис програми;
- код програми.

6 Етапи роботи

Етапи та терміни виконання робіт представлені у таблиці А.1.

Таблиця А.1 – Етапи та терміни виконання робіт

Номер	Назва етапів роботи	Термін виконання
1	Аналіз практичної задачі інженерії програмного забезпечення	12.02.2026
2	Аналіз вимог до програмного забезпечення	09.03.2026
3	Розробка архітектури програмного забезпечення	06.04.2026
4	Розробка проєкту програмного забезпечення	14.04.2026
5	Реалізація та тестування програмного забезпечення	30.04.2026

7 Порядок контролю та приймання

Контроль за аналізом та проєктуванням кожної окремої частини програмного забезпечення на кожному етапі з урахуванням вимог, визначених у технічному завданні. Кожна стадія розробки повинна бути представлена в зазначені строки та узгоджена із замовником.

Прийом проводиться відповідно до програми і методики випробувань і документується за допомогою протоколу проведення випробувань. У разі знаходження помилок під час прийому програмного виробу складається акт про знайдені помилки, який підписуються представниками замовника і розробника і затверджуються керівником організації – замовника та організації – розробника. Розробник повинен протягом не більше ніж двох тижнів виправити зазначені помилки і оповістити замовника про повторне проведення перевірки.

ДОДАТОК Б – КОД ВЕБЗАСТОСУНКУ

Лістинг 1 – GroupServiceImpl

```
@Service
@RequiredArgsConstructor
public class GroupServiceImpl implements GroupService {

    private final GroupRepository repository;
    private final GroupConverter converter;

  @Override
    public List<ShortenedGroupDto> getGroups() {
        return repository.findAllByOrderByName().stream()
                .map(converter::convertToShortenedGroupDto)
                .toList();
    }

    @Override
    public GroupDto getGroup(String id) {
        return             converter.convertToGroupDto(getGroup(new
ObjectId(id)));
    }

    @Override
    public Group getGroup(ObjectId id) {
        return repository.findById(id)
                .orElseThrow(()                ->                new
EntityNotFoundException(String.format("Group  with  id  =  [%s]  not
found", id)));
    }

    @Override
    public ShortenedGroupDto createGroup(CreateGroupDto dto) {
        return
converter.convertToShortenedGroupDto(repository.save(converter.con
vertToCreateGroupDto(dto)));
    }

    @Override
    public void updateGroup(String id, UpdateGroupDto dto) {
        Group group = getGroup(new ObjectId(id));
        group.setName(dto.getName());
        repository.save(group);
    }

    @Override
    public void deleteGroup(String id) {
        repository.deleteById(new ObjectId(id));
    }
}
```

Лістинг 2 – ProjectServiceImpl

```
@Service
@RequiredArgsConstructor
public class ProjectServiceImpl implements ProjectService {

    private final GroupRepository repository;
    private final ProjectConverter projectConverter;
    private final GroupService service;
    private final InfoService infoService;

    @Override
    public ProjectDto createProject(String groupId,
CreateProjectDto dto) {
        Group group = service.getGroup(new ObjectId(groupId));
        Project project = projectConverter.convertToProject(dto);
        group.getProjects().add(project);
        repository.save(group);
        return projectConverter.convertToProjectDto(project);
    }

    @Override
    public void updateProject(String groupId, String projectId,
UpdateProjectDto dto) {
        Group group = service.getGroup(new ObjectId(groupId));
        Project project = getProject(projectId, group);
        project.setName(dto.getName());
        project.setGitProjectId(dto.getGitProjectId());
        project.setGitProjectUrl(dto.getGitProjectUrl());
        project.setGoals(dto.getGoals());
        project.setWebAddress(dto.getWebAddress());
        repository.save(group);
    }

    @Override
    public void deleteProject(String groupId, String projectId) {
        Group group = service.getGroup(new ObjectId(groupId));
        group.getProjects().removeIf(project ->
project.getId().toString().equals(projectId));
        repository.save(group);
    }

    @Override
    public ProjectDetailsDto getProjectDetail(String groupId,
String projectId) {
        String webAddress = getProject(projectId,
service.getGroup(new ObjectId(groupId))).getWebAddress();
        InfoDto infoDto = infoService.getInfo(webAddress);
        return
projectConverter.convertToProjectDetailsDto(infoDto);
    }

    private Project getProject(String projectId, Group group) {
        return group.getProjects().stream()
```

```
                .filter(project                                    ->
project.getId().toString().equals(projectId))
                .findFirst()
                .orElseThrow(()                ->                new
EntityNotFoundException(String.format("Project with id = [%s] not
found in group with id = [%s]", projectId, group.getId())));
    }
}
```

Лістинг 3 – GitServiceImpl

```
@Service
@RequiredArgsConstructor
public class GitServiceImpl implements GitService {

    private static final String TOKEN = "PRIVATE-TOKEN";
    private     static     final     String     BRANCH_URL     =
"%s/api/v4/projects/%s/repository/branches";
    private     static     final     String     PIPELANE_URL     =
"%s/api/v4/projects/%s/pipeline?ref=%s&variables[][key]=%s&variabl
es[][value]=true";

    private final GroupService groupService;
    private final RestTemplate restTemplate;

    @Value("${gitlab.url}")
    private String gitlabUrl;

    @Override
    public   List<String>   getBranches(String   groupId,   String
projectId, String token) {
        BranchDto[] branches = restTemplate.exchange(
                        BRANCH_URL.formatted(gitlabUrl,
getProject(groupId, projectId).getGitProjectId()),
                        HttpMethod.GET,
getHttpEntityBranches(token), BranchDto[].class)
                .getBody();
        return convertToStrings(branches);
    }

    @Override
    public void createPipeline(String groupId, String projectId,
CreatePipelineDto dto, String token) {
        restTemplate.exchange(
                PIPELANE_URL.formatted(gitlabUrl,
getProject(groupId, projectId).getGitProjectId(), dto.getBranch(),
dto.getGoal()),
                HttpMethod.POST,                                   new
HttpEntity<Void>(getHttpHeadersForPipeline(token)), String.class);
    }

    private Project getProject(String groupId, String projectId) {
        return                              groupService.getGroup(new
ObjectId(groupId)).getProjects()
```

```
                .stream()
                .filter(p          ->          p.getId().equals(new
ObjectId(projectId)))
                .findFirst()
                .orElseThrow(()                ->                new
EntityNotFoundException(String.format("Project with id = [%s] not
found in group with id = [%s]", projectId, groupId)));
    }

    private HttpEntity<Void> getHttpEntityBranches(String token) {
        HttpHeaders httpHeaders = new HttpHeaders();
        httpHeaders.set(TOKEN, token);
        httpHeaders.set(HttpHeaders.ACCEPT_ENCODING, "identity");

httpHeaders.setAccept(List.of(MediaType.APPLICATION_JSON));
        return new HttpEntity<>(httpHeaders);
    }

    private List<String> convertToStrings(BranchDto[] branches) {
        return Optional.ofNullable(branches)
                .stream()
                .flatMap(Arrays::stream)
                .map(BranchDto::getName)
                .toList();
    }

    private HttpHeaders getHttpHeadersForPipeline(String token) {
        HttpHeaders httpHeaders = new HttpHeaders();
        httpHeaders.set(TOKEN, token);

httpHeaders.setContentType(MediaType.APPLICATION_FORM_URLENCODED);
        return httpHeaders;
    }
}
```

Лістинг 4 – InfoServiceImpl

```
@Service
@RequiredArgsConstructor
public class InfoServiceImpl implements InfoService {

    private final RestTemplate restTemplate;

    @Override
    public InfoDto getInfo(String webAddress) {
        try {
            return            restTemplate.getForObject(webAddress,
InfoDto.class);
        } catch (RestClientException ex) {
            throw   new   HttpRequestException("Error   during   http
request", ex);
        }
    }
}
```

Лістинг 5 – ProjectController

```
@Validated
@RestController
@RequiredArgsConstructor
public class ProjectController {

    private final ProjectService service;

    @GetMapping("/group/{groupId}/project/{projectId}/details")
    public ProjectDetailsDto
getProjectDetail(@PathVariable("groupId") @NotBlank String groupId,
@PathVariable("projectId") @NotBlank String projectId) {
        return service.getProjectDetail(groupId, projectId);
    }

    @PostMapping("/group/{groupId}/project")
    public ProjectDto createProject(@PathVariable("groupId")
@NotBlank String groupId, @RequestBody @Valid CreateProjectDto dto)
{
        return service.createProject(groupId, dto);
    }

    @PutMapping("/group/{groupId}/project/{projectId}")
    public void updateProject(@PathVariable("groupId") @NotBlank
String groupId, @PathVariable("projectId") @NotBlank String
projectId,
                              @RequestBody @Valid UpdateProjectDto
dto) {
        service.updateProject(groupId, projectId, dto);
    }

    @DeleteMapping("/group/{groupId}/project/{projectId}")
    public void deleteProject(@PathVariable("groupId") @NotBlank
String groupId, @PathVariable("projectId") @NotBlank String
projectId) {
        service.deleteProject(groupId, projectId);
    }
}
```

Лістинг 6 – GroupController

```
@Validated
@RestController
@RequiredArgsConstructor
public class GroupController {

    private final GroupService service;

    @GetMapping("/group")
    public List<ShortenedGroupDto> getGroups() {
        return service.getGroups();
    }
```

```
    @GetMapping("/group/{id}")
    public GroupDto getGroup(@PathVariable("id") @NotBlank String
id) {
        return service.getGroup(id);
    }

    @PostMapping("/group")
    public ShortenedGroupDto createGroup(@RequestBody @Valid
CreateGroupDto dto) {
        return service.createGroup(dto);
    }

    @PutMapping("/group/{id}")
    public void updateGroup(@PathVariable("id") @NotBlank String id,
@RequestBody @Valid UpdateGroupDto dto) {
        service.updateGroup(id, dto);
    }

    @DeleteMapping("/group/{id}")
    public void deleteGroup(@PathVariable("id") @NotBlank String id)
{
        service.deleteGroup(id);
    }
}
```

Лістинг 7 – GitController

```
@Validated
@RestController
@RequiredArgsConstructor
public class GitController {

    private final GitService service;

    @GetMapping("/group/{groupId}/project/{projectId}/git/branch")
    public List<String> getBranches(@PathVariable String groupId,
@PathVariable String projectId,
                                    @RequestHeader("gitToken")
String token) {
        return service.getBranches(groupId, projectId, token);
    }


@PostMapping("/group/{groupId}/project/{projectId}/git/pipeline")
    public void createPipeline(@PathVariable String groupId,
@PathVariable String projectId,
                               @RequestBody CreatePipelineDto dto,
@RequestHeader("gitToken") String token) {
        service.createPipeline(groupId, projectId, dto, token);
    }
}
```

Лістинг 8 – Projects

```
const projectsMock = [{ id: "-1" }, { id: "-2" }, { id: "-3" }, { id: "-4" }] as ProjectDto[];

export const Projects = ({
  groupId,
  projects,
  showLoader,
}: {
  groupId?: string;
  projects?: ProjectDto[];
} & LoaderStruct) => {
  return (
    <ProjectsView
      groupId={groupId}
      showLoader={showLoader}
      projects={showLoader ? projectsMock : projects}
      headerComponent={showLoader ? <Skeleton height="25px" containerClassName="w-100" /> : <ProjectHeader />}
      staticFieldsComponent={showLoader ? <Skeleton /> : <ProjectStaticFields />}
    />
  );
};

interface ProjectsViewProps {
  groupId?: string;
  showLoader: boolean;
  projects?: ProjectDto[];
  headerComponent: ReactElement;
  staticFieldsComponent: ReactElement;
}

const ProjectsView = ({ groupId, showLoader, projects, headerComponent, staticFieldsComponent }: ProjectsViewProps) => {
  return (
    <div className="w-100 py-3" style={{ minHeight: "100vh" }}>
      <div className="container">
        <div className="row row-cols-2 row-cols-xl-3 row-cols-xxl-4 g-4">
          {projects?.map((project) => (
            <div className="col" key={project.id}>
              <div  className="card  h-100  shadow-sm  position-relative">
                <div className="card-header d-flex justify-content-between align-items-center">
                  {Children.map(headerComponent, (child) => {
                    return cloneElement(child, { data: { project, groupId } });
                  })}
                </div>

                <div  className="card-body  d-flex  flex-column justify-content-between">
```

```
                  <div>
                    <div className="card-text mb-3">
                      {Children.map(staticFieldsComponent,  (child)
=> {
                        return cloneElement(child, { data: { project
} });
                      })}
                    </div>
                    <ProjectDetails               groupId={groupId}
projectId={project.id} showLoader={showLoader} />
                  </div>
                </div>
              </div>
            ))}
          </div>
        </div>
      </div>
    );
};

const ProjectHeader = ({ data }: { data?: { project: ProjectDto;
groupId: string } }) => {
  const updateProjectModalStruct = useAppModal("updateProject");
  const deleteProjectModalStruct = useAppModal("deleteProject");
  const runPipelineModalStruct = useAppModal("runPipeline");

  return (
    data && (
      <>
        <a
          className="text-decoration-none text-black"
          target="_blank"
          href={data.project.gitProjectUrl}
          rel="noreferrer"
        >
          <h5 className="mb-0 text-break">{data.project.name}</h5>
        </a>
        <div className="d-flex align-items-center">
          {data.project.gitProjectId && (
            <Button
              onClick={() =>
                runPipelineModalStruct.openModal({
                  groupId: data.groupId,
                  ...data.project,
                })
              }
              variant="light"
              className="w-auto p-0 m-1"
            >
              <RocketTakeoff size="32px" viewBox="-4 -4 24 24" />
            </Button>
          )}
```

```
            <Button
              onClick={() =>
                updateProjectModalStruct.openModal({
                  groupId: data.groupId,
                  ...data.project,
                })
              }
              variant="light"
              className="w-auto p-0 m-1"
            >
              <Pencil size="32px" viewBox="-4 -4 24 24" />
            </Button>
            <Button
              onClick={() =>
                deleteProjectModalStruct.openModal({
                  groupId: data.groupId,
                  ...data.project,
                })
              }
              variant="light"
              className="w-auto p-0 m-1"
            >
              <Trash size="32px" viewBox="-4 -4 24 24" />
            </Button>
          </div>
        </>
      )
    );
};

const ProjectStaticFields = ({ data }: { data?: { project: ProjectDto
} }) => {
  return (
    data && (
      <>
        <strong>Веб-адреса:</strong> {data.project.webAddress}
      </>
    )
  );
}; const projectsMock = [{ id: "-1" }, { id: "-2" }, { id: "-3" },
{ id: "-4" }] as ProjectDto[];

export const Projects = ({
  groupId,
  projects,
  showLoader,
}: {
  groupId?: string;
  projects?: ProjectDto[];
} & LoaderStruct) => {
  return (
    <ProjectsView
      groupId={groupId}
```

```
      showLoader={showLoader}
      projects={showLoader ? projectsMock : projects}
      headerComponent={showLoader   ?   <Skeleton   height="25px"
containerClassName="w-100" /> : <ProjectHeader />}
      staticFieldsComponent={showLoader   ?   <Skeleton   />   :
<ProjectStaticFields />}
    />
  );
};

interface ProjectsViewProps {
  groupId?: string;
  showLoader: boolean;
  projects?: ProjectDto[];
  headerComponent: ReactElement;
  staticFieldsComponent: ReactElement;
}

const   ProjectsView   =   ({   groupId,   showLoader,   projects,
headerComponent, staticFieldsComponent }: ProjectsViewProps) => {
  return (
    <div className="w-100 py-3" style={{ minHeight: "100vh" }}>
      <div className="container">
        <div className="row row-cols-2 row-cols-xl-3 row-cols-xxl-4
g-4">
          {projects?.map((project) => (
            <div className="col" key={project.id}>
              <div   className="card   h-100   shadow-sm   position-
relative">
                <div className="card-header d-flex justify-content-
between align-items-center">
                  {Children.map(headerComponent, (child) => {
                    return cloneElement(child, { data: { project,
groupId } });
                  })}
                </div>

                <div   className="card-body   d-flex   flex-column
justify-content-between">
                  <div>
                    <div className="card-text mb-3">
                      {Children.map(staticFieldsComponent,  (child)
=> {
                        return cloneElement(child, { data: { project
} });
                      })}
                    </div>
                    <ProjectDetails               groupId={groupId}
projectId={project.id} showLoader={showLoader} />
                  </div>
                </div>
              </div>
            </div>
```

```
          ))}
        </div>
      </div>
    </div>
  );
};

const ProjectHeader = ({ data }: { data?: { project: ProjectDto;
groupId: string } }) => {
  const updateProjectModalStruct = useAppModal("updateProject");
  const deleteProjectModalStruct = useAppModal("deleteProject");
  const runPipelineModalStruct = useAppModal("runPipeline");

  return (
    data && (
      <>
        <a
          className="text-decoration-none text-black"
          target="_blank"
          href={data.project.gitProjectUrl}
          rel="noreferrer"
        >
          <h5 className="mb-0 text-break">{data.project.name}</h5>
        </a>
        <div className="d-flex align-items-center">
          {data.project.gitProjectId && (
            <Button
              onClick={() =>
                runPipelineModalStruct.openModal({
                  groupId: data.groupId,
                  ...data.project,
                })
              }
              variant="light"
              className="w-auto p-0 m-1"
            >
              <RocketTakeoff size="32px" viewBox="-4 -4 24 24" />
            </Button>
          )}
          <Button
            onClick={() =>
              updateProjectModalStruct.openModal({
                groupId: data.groupId,
                ...data.project,
              })
            }
            variant="light"
            className="w-auto p-0 m-1"
          >
            <Pencil size="32px" viewBox="-4 -4 24 24" />
          </Button>
          <Button
            onClick={() =>
```

```
              deleteProjectModalStruct.openModal({
                groupId: data.groupId,
                ...data.project,
              })
            }
            variant="light"
            className="w-auto p-0 m-1"
          >
            <Trash size="32px" viewBox="-4 -4 24 24" />
          </Button>
        </div>
      </>
    )
  );
};

const ProjectStaticFields = ({ data }: { data?: { project: ProjectDto
} }) => {
  return (
    data && (
      <>
        <strong>Веб-адреса:</strong> {data.project.webAddress}
      </>
    )
  );
};
```

Лістинг 8 – ProjectDetails

```
const projectDetailsMock = {"-1": "-1", "-2": "-2"};

export const ProjectDetails = ({
                                    groupId,
                                    projectId,
                                    showLoader,
                                }: {
    groupId?: string;
    projectId: string;
} & LoaderStruct) => {
    const {data: group} = groupApi.useGetGroupQuery(groupId, {skip:
showLoader});

    const              {data,              isFetching}              =
projectApi.useGetProjectDetailsQuery(
        {groupId, projectId},
        {skip: showLoader}
    );

    const showSkeleton = showLoader || isFetching;

    return (
        <ProjectDetailsView
            projectDetails={showSkeleton  ?  projectDetailsMock  :
data}
```

```
            version={group?.version}
        >
            {showSkeleton ? <Skeleton/> : <KeyValue/>}
        </ProjectDetailsView>
    );
};

const ProjectDetailsView = ({
                                projectDetails,
                                version,
                                children,
                            }: {
    projectDetails?: ProjectDetailsDto | Record<string, string>;
    version?: string;
    children: ReactElement;
}) => {
    return (
        <>
            {projectDetails &&
                Object.entries(projectDetails).map(([key,   value])
=> (
                    <div
                        key={key}
                        className={classNames("card-text", {
                            "text-warning":  key  ===  "version"  &&
version !== value
                        })}
                    >
                        {Children.map(children, (child) => {
                            return cloneElement(child, {data: {key,
value}});
                        })}
                    </div>
                ))}
        </>
    );
};

const KeyValue = ({data}: { data?: { key: any; value: any } }) => {
    return (
        <>
            {data && (
                <>
                    <strong>{data.key}:</strong> {data.value}
                </>
            )}
        </>
    );
};
```

Лістинг 9 – Groups

```
import {Menu, MenuItem, MenuItemStyles, Sidebar} from "react-pro-
sidebar";
```

```
import {NavLink, Outlet} from "react-router-dom";
import {useAppModal} from "../../hook/useAppModal.ts";
import {ShortenedGroupDto} from "../../dto/ShortenedGroupDto.ts";
import {groupApi} from "../../api/groupApi.ts";
import                    {MainBackground}                    from
"../../component/background/MainBackground.tsx";
import Skeleton from "react-loading-skeleton";
import {Children, cloneElement, ReactElement} from "react";
import {Gear, Plus} from "react-bootstrap-icons";

const menuItemStyles: MenuItemStyles = {
    root: {
        fontSize: "14px",
        fontWeight: 400,
    },
    button: {
        margin: "10px",
        borderRadius: "5px",
        "&:hover": {
            backgroundColor: "#2d2e39",
        },
        "&.active": {
            backgroundColor: "#444555",
        },
    },
};

const groupsMock = [{id: "-1"}, {id: "-2"}, {id: "-3"}, {id: "-4"}]
as ShortenedGroupDto[];

export const Groups = () => {
    const {data, isFetching} = groupApi.useGetGroupsQuery();

    return (
        <GroupsView
            groups={isFetching ? groupsMock : data}
            disabledLinks={isFetching}
            groupItemComponent={isFetching       ?       <Skeleton
height="20px"/> : <GroupItem/>}
        />
    );
};

const GroupsView = ({
  groups,
  disabledLinks,
  groupItemComponent,
}: {
  groups?: ShortenedGroupDto[];
  disabledLinks: boolean;
  groupItemComponent: ReactElement;
}) => {
  return (
```

```
    <div className="d-flex h-100 overflow-hidden">
      <div className="d-flex">
        <Sidebar
          backgroundColor="#202123"
          rootStyles={{
            color: "white",
          }}
          className="fixed-bottom"
          style={{ height: "100vh" }}
        >
          <div
            style={{
              display: "flex",
              flexDirection: "column",
              height: "100%",
            }}
          >
            <Menu menuItemStyles={menuItemStyles}>
              <CreateItem/>
              {groups?.map((group) => (
                <MenuItem  key={group.id}  disabled={disabledLinks}
component={<NavLink to={`/group/${group.id}`} />}>
                  {Children.map(groupItemComponent, (child) => {
                    return cloneElement(child, { data: { group } });
                  })}
                </MenuItem>
              ))}
            </Menu>

            <Menu     menuItemStyles={menuItemStyles}     style={{
marginTop: "auto" }}>
              <SettingsItem />
            </Menu>
          </div>
        </Sidebar>
      </div>
      <MainBackground>
        <Outlet />
      </MainBackground>
    </div>
  );
};

const SettingsItem = () => {
    const modalStruct = useAppModal("userSettings");
    return (
        <MenuItem
            onClick={() => modalStruct.openModal()}
            style={{borderRadius: "5px", border: "1px solid white"}}
        >
            <div className="d-flex flex-row align-items-center">
                <Gear size="16px" className="me-2"/>
                Налаштування
```

```
            </div>
        </MenuItem>
    );
};

const CreateItem = () => {
    const modalStruct = useAppModal("createGroup");

    return (
        <MenuItem onClick={() => modalStruct.openModal()} style={{
borderRadius: "5px", border: "1px solid white" }}>
            <div className="d-flex flex-row align-items-center">
                <Plus color="white" size="16" />
                Створити групу проєктів
            </div>
        </MenuItem>
    );
};

const GroupItem = ({data}: { data?: { group: ShortenedGroupDto } })
=> {
    return data && <span>{data.group.name}</span>;
};
```

Лістинг 10 – Group

```
import { Form } from "react-bootstrap";
import { FormProvider } from "react-hook-form"; // Додано імпорт
import { useAppForm } from "../../hook/useAppForm";
import groupSchema from "./validation";
import { groupApi } from "../../api/groupApi.ts";
import { MenuButton, MenuItem, Menu } from "@szhsin/react-menu";
import { useAppModal } from "../../hook/useAppModal.ts";
import          {          EditableFormField          }          from
"../../component/form/EditableFormField.tsx";
import          {          BackgroundHint          }          from
"../../component/background/BackgroundHint.tsx";
import  {  Projects  }  from  "../projects/Projects.tsx";  //  Додано
імпорт
import { useParams } from "react-router-dom";
import Skeleton from "react-loading-skeleton";
import { Children, cloneElement, useEffect } from "react";
import  {  GroupDto  }  from  "../../dto/group/GroupDto.ts";  //
Виправлено шлях
import { FormField } from "../../component/form/FormField.tsx";

export const Group = () => {
  const { id } = useParams();
  const      {      data,      isFetching,      refetch      }      =
groupApi.useGetGroupQuery(id);

  return (
    <GroupView
      group={data}
```

```
      showLoader={isFetching}
      refetch={refetch}
      headerComponent={isFetching   ?   <Skeleton   height="30px"
className="my-2" /> : <GroupHeader />}
    />
  );
};

interface GroupViewProps {
  group?: GroupDto;
  showLoader: boolean;
  headerComponent: any;
  refetch: () => void;
}

export const GroupView = ({ group, showLoader, headerComponent,
refetch }: GroupViewProps) => {
  const [updateGroup] = groupApi.useUpdateGroupMutation();
  const { formInstance, submitForm } = useAppForm({
    defaultValues: { id: group?.id, version: group?.version, name:
group?.name },
    validationSchema: groupSchema,
    onFormSubmit: updateGroup,
  });

  useEffect(() => {
    formInstance.reset({ id: group?.id, name: group?.name, version:
group?.version });
  }, [group, formInstance.reset]);

  return (
    <FormProvider {...formInstance}>
      <Form className="w-100">
        <div style={{ height: "100px" }}>
          <div className="bg-white pb-3">
            <div className="container-fluid">
              <div className="row pt-3 align-items-center position-
sticky top-0 z-3">
                {Children.map(headerComponent, (child) => {
                  return cloneElement(child, { data: { submitForm,
group, refetch } });
                })}
              </div>
            </div>
          </div>
          <hr className="mt-0" />
        </div>

        <div  className="overflow-y-auto  px-3"  style={{  height:
`${window.innerHeight - 100}px` }}>
          {!showLoader && group?.projects?.length === 0 ? (
            <BackgroundHint>Створіть новий проєкт</BackgroundHint>
          ) : (
```

```
            <Projects                                groupId={group?.id}
projects={group?.projects} showLoader={showLoader} />
          )}
        </div>
      </Form>
    </FormProvider>
  );
};

const GroupHeader = ({ data }: { data?: { group: GroupDto;
submitForm: () => void; refetch: () => void } }) => {
  const createProjectModalStruct = useAppModal("createProject");
  const deleteGroupModalStruct = useAppModal("deleteGroup");

  return (
    data && (
      <>
        {/* Виправлено сітку col-10 + col-2 = 12 */}
        <div className="col-10">
          <EditableFormField id="name" maxLength={50} />
        </div>
        <div  className="col-2  d-flex  justify-content-end  align-
items-center">
          <FormField id="version" offsetFree className="d-flex mb-0
me-3" onBlur={data.submitForm}>
            <Form.Control type="text" placeholder="Версія" style={{
width: "90px" }} />
          </FormField>
          <Menu menuButton={<MenuButton className="btn btn-outline-
dark">•••</MenuButton>}>
            <MenuItem                onClick={data.refetch}>Оновити
дані</MenuItem>
            <MenuItem                 onClick={()                 =>
createProjectModalStruct.openModal(data.group)}>Створити
проєкт</MenuItem>
            <MenuItem                 onClick={()                 =>
deleteGroupModalStruct.openModal(data.group)}>Видалити
групу</MenuItem>
          </Menu>
        </div>
      </>
    )
  );
};

export const SelectGroupHint = () => {
  return  <BackgroundHint>Виберіть  групу  проєктів  або  створіть
нову</BackgroundHint>;
};
```

ДОДАТОК В – ОПИС ВЕБЗАСТОСУНКУ

1. Загальні відомості

Програмне забезпечення є повностековим (full-stack) вебзастосунком, розробленим для автоматизації моніторингу, управління версіями та розгортання програмних продуктів.

- Серверна частина (Backend): розроблена на мові програмування Java.
- Клієнтська частина (Frontend): розроблена на базі бібліотеки React з використанням мови TypeScript.
- Інтеграція: взаємодія з інфраструктурою GitLab здійснюється через GitLab REST API.

2. Функціональне призначення

Програмне забезпечення призначене для DevOps-інженерів та розробників і вирішує наступні завдання:

- Отримання та відображення ієрархічної структури груп та проєктів з GitLab.
- Моніторинг поточних версій розгорнутих мікросервісів та застосунків.
- Автоматичне порівняння цільової (еталонної) версії проєкту з фактичною версією розгорнутого застосунку з візуальною індикацією розбіжностей.
- Створення нових проєктів у межах існуючих груп безпосередньо з вебзастосунку.
- Видалення існуючих груп та проєктів.
- Ініціація процесів розгортання (CI/CD pipelines) за допомогою GitLab CI.

3. Опис логічної структури

Програмне забезпечення побудовано за класичною клієнт-серверною архітектурою.

Серверна частина (Java) відповідає за бізнес-логіку та безпечну взаємодію із зовнішніми API. До її логічної структури входять:

- Рівень контролерів (Controllers): приймає HTTP-запити від клієнтської частини та маршрутизує їх.
- Рівень сервісів (Services): реалізує основну бізнес-логіку (наприклад, GroupServiceImpl), обробляє дані та формує запити до GitLab API.
- Рівень передачі даних (DTO): об'єкти (наприклад, GroupDto) для безпечної та структурованої передачі інформації між шарами вебзастосунку.

Клієнтська частина (React) побудована на компонентному підході та містить:

- Компоненти відображення (GroupHeader, Projects): відповідають за рендеринг списків груп, проєктів та елементів управління (кнопки, форми).
- Модуль управління станом та хуки (useAppModal): кастомні рішення для контролю поведінки модальних вікон (створення/видалення проєктів) та життєвого циклу сторінки.
- Модуль автентифікації: забезпечує зберігання та передачу токенів доступу (Personal Access Token) для безпечних запитів на бекенд.

4. Використовувані технічні засоби

Для функціонування серверної та клієнтської частин необхідні наступні технічні засоби:

- Для сервера: встановлене середовище виконання Java (Java Runtime Environment / JDK), оперативна пам'ять від 4 ГБ.
- Для клієнта: сучасний веббраузер з підтримкою JavaScript/ES6 (Google Chrome, Firefox, Safari) та інтернет-з'єднання.
- Інфраструктура: наявність облікового запису GitLab з відповідними правами доступу та згенерованим токеном.

5. Виклик та завантаження

Для запуску вебзастосунку необхідно ініціалізувати обидві його частини:

1) Запуск серверної частини: виконується збірка Java-проєкту (наприклад, за допомогою Maven або Gradle) та запуск скомпільованого .jar файлу. Сервер за замовчуванням запускається на локальному порту (наприклад, 8080).

2) Запуск клієнтської частини: у директорії фронтенду виконується команда встановлення залежностей npm install, після чого запускається локальний сервер розробки командою npm start (порт 3000).

3) Після успішного запуску користувач відкриває веббраузер за адресою клієнтського застосунку.

6. Вхідні та вихідні дані

- Вхідні дані: ідентифікатори користувача (GitLab Token), текстові дані з форм створення груп або проєктів, номери версій застосунків.
- Вихідні дані: структуровані JSON-відповіді від серверної частини, які динамічно перетворюються клієнтом у візуальний інтерфейс користувача (списки, таблиці, модальні вікна з результатами виконання операцій).

ДОДАТОК Г – ІНСТРУКЦІЯ КОРИСТУВАЧА ВЕБЗАСТОСУНКУ

1. Призначення програми

Програмне забезпечення призначене для автоматизації роботи DevOps-інженерів та розробників. Воно надає єдину інформаційну панель для:

- автоматичного збору телеметрії та перевірки фактичних версій розгорнутих мікросервісів;
- візуального виявлення застарілих застосунків шляхом порівняння з еталонною версією;
- дистанційної ініціації процесів розгортання (CI/CD) через взаємодію з GitLab API;
- управління логічними середовищами (групами) та картками проєктів.

2. Умови виконання програми

Для успішної роботи з вебзастосунком необхідні наступні умови:

- Апаратні вимоги: персональний комп'ютер із процесором від 1 ГГц та оперативною пам'яттю від 4 ГБ.
- Програмні вимоги: сучасний веббраузер (Google Chrome, Mozilla Firefox, Safari) з підтримкою JavaScript.
- Мережа: підключення до локальної мережі або Інтернету для доступу до бекенд-сервера, віддалених мікросервісів та платформи GitLab.
- Доступ: наявність згенерованого Personal Access Token від GitLab-акаунта для запуску пайплайнів.

3. Підготовка до роботи (Запуск)

Для початку роботи необхідно запустити серверну та клієнтську частини системи:

1) Запуск сервера (Backend): відкрийте термінал у папці з Java-проєктом та виконайте запуск зібраного .jar файлу (або запустіть через середовище

розробки IntelliJ IDEA). Переконайтеся, що база даних MongoDB працює, а сервер успішно стартував.

2) Запуск клієнта (Frontend): відкрийте термінал у папці клієнтської частини (React) та введіть команду npm start (або npm run dev). Застосунок автоматично відкриється у браузері за адресою http://localhost:3000 (або іншим портом, вказаним у конфігурації).

4. Робота з програмою

4.1. Головний екран та навігація

При відкритті застосунку користувач потрапляє на головний екран, зовнішній вигляд якого наведено на рисунку Г.1. Ліворуч розташована бічна панель навігації зі списком усіх створених логічних груп (середовищ). У нижній частині бічної панелі знаходиться кнопка «Налаштування» (іконка шестірні), яка дозволяє відкрити вікно конфігурації профілю для безпечного введення токена доступу GitLab. Центральна робоча область відображає деталі обраної групи та містить інструменти для моніторингу.

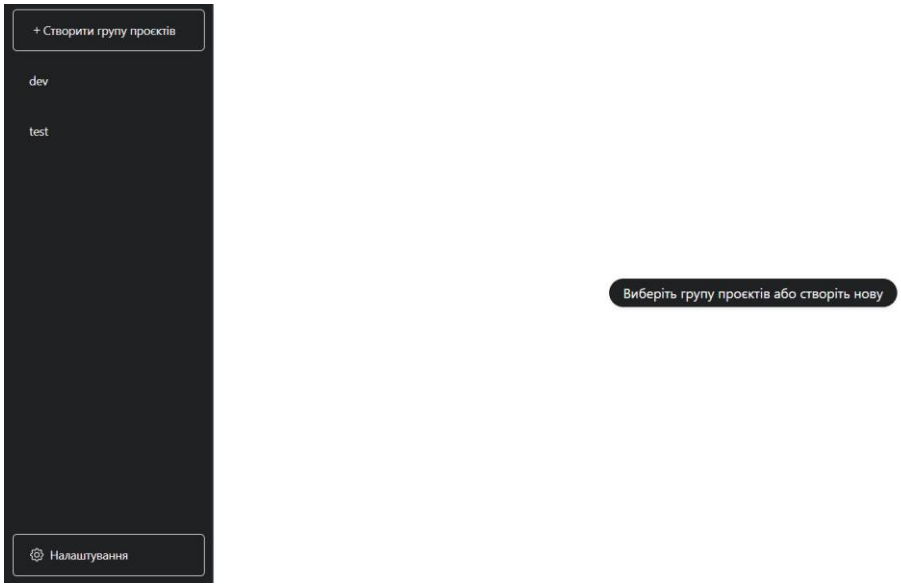


Рисунок Г.1 – Головний екран застосунку

4.2. Створення та управління групами

Для об'єднання мікросервісів в одне логічне середовище користувач спочатку має створити нову групу. Для цього необхідно виконати такі кроки:

1) Натисніть кнопку «Створити групу проєктів» у лівому бічному меню.

2) На екрані з'явиться модальне вікно (рисунок Г.2). У поле «Назва» введіть бажане ім'я для нового середовища.

3) Натисніть кнопку «СТВОРИТИ» для підтвердження дії (або «ВІДМІНИТИ» для закриття вікна без збереження).

Після успішного створення група з'явиться у бічному списку навігації. Якщо обрати її, у верхній частині центральної робочої області з'явиться контекстне меню управління (кнопка у вигляді трьох крапок «•••», як показано на рисунку Г.1). Відкривши це меню, користувач може:

- Оновити дані: примусово оновити телеметрію для всіх проєктів поточної групи.
- Створити проєкт: викликати вікно для додавання нового мікросервісу.
- Видалити групу: ініціювати повне видалення обраного середовища з бази даних.
-

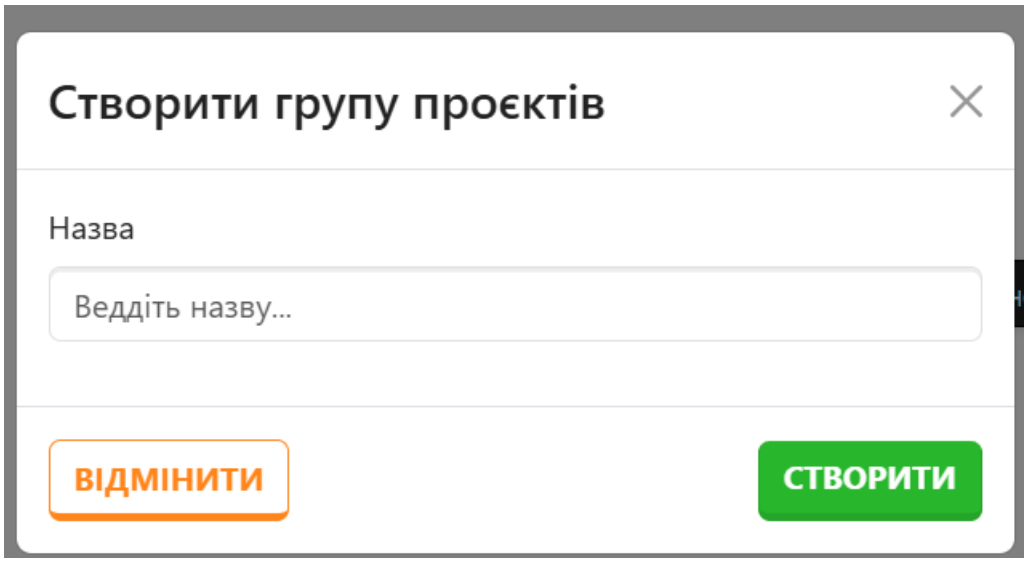


Рисунок Г.2 – Модальне вікно створення нової групи

4.3. Моніторинг версій та виявлення розбіжностей

Основний функціонал системи полягає у виявленні застарілих сервісів. У полі «Версія» в шапці групи користувач вводить цільову (еталонну) версію для всього середовища. Застосунок автоматично виконує HTTP-запити до всіх проєктів у цій групі та отримує їх фактичні телеметричні дані.

Як видно на рисунку Г.3, якщо фактична версія розгорнутого мікросервісу не збігається з цільовою, вебзастосунок автоматично підсвітить її попереджувальним помаранчевим кольором на картці проєкту. При наведенні курсора на підсвічену версію з'являється інформаційна підказка з деталями розбіжності (очікувана та фактична версії). Це чітко сигналізує адміністратору про необхідність оновлення.

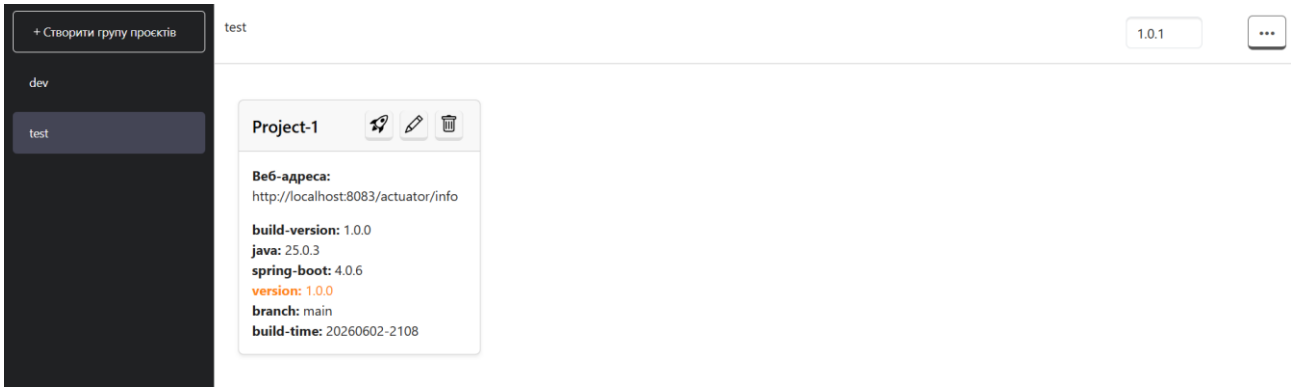


Рисунок Г.3 – Візуальна індикація розбіжності фактичної та еталонної версій

4.4. Додавання та управління карткою проєкту

Щоб підключити новий мікросервіс до моніторингу, необхідно обрати пункт «Створити проєкт» у контекстному меню обраної групи. На екрані з'явиться модальне вікно (рисунок Г.4), у якому користувачу необхідно заповнити такі реєстраційні дані:

- Назва: ім'я нового мікросервісу для відображення у вебзастосунку;
- Url: цільова веб-адреса розгорнутого застосунку, за якою програмна система буде автоматично збирати телеметрію;
- Id проекта в GitLab: унікальний числовий ідентифікатор репозиторію в інфраструктурі GitLab;

– Url на GitLab: пряме веб-посилання на репозиторій проєкту.

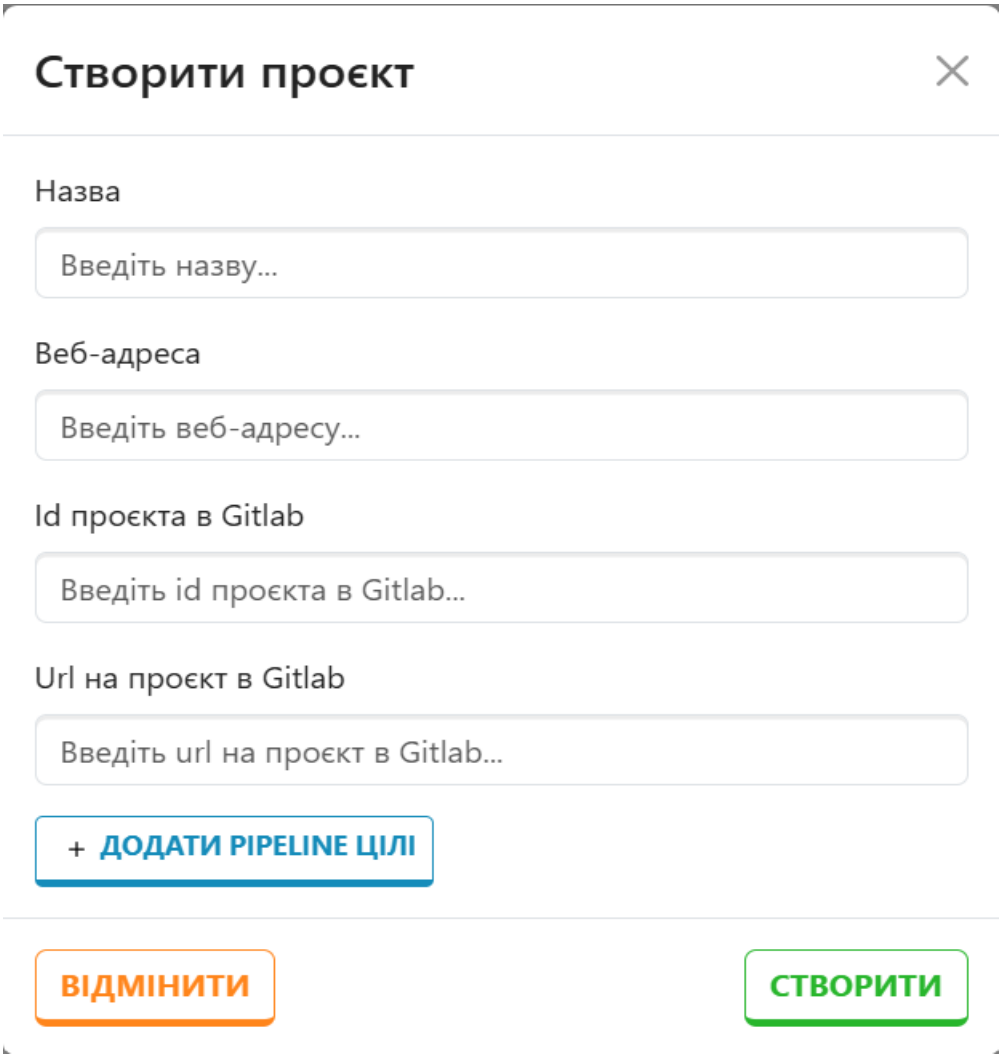


Рисунок Г.4 – Модальне вікно додавання проєкту

Після створення проєкт з'являється у робочій області групи. На кожній картці розгорнутого проєкту у верхньому правому куті розташовано три кнопки управління (рисунок Г.3):

– Іконка ракети: дистанційна ініціалізація розгортання (CI/CD) через GitLab.

– Іконка олівця: відкриває модальне вікно для редагування налаштувань проєкту (наприклад, зміни веб-адреси).

– Іконка кошика: ініціює видалення проєкту з поточної групи (потребує підтвердження).

4.5. Ініціалізація розгортання (CI/CD)

Якщо на картці проєкту виявлено застарілу версію, користувач може оновити її дистанційно, натиснувши кнопку запуску пайплайну (іконку ракети). Після цього відкривається модальне вікно «Запустити pipeline», зовнішній вигляд якого наведено на рисунку Г.5.

У цьому вікні застосунок автоматично підвантажує доступні гілки з віддаленого репозиторію GitLab. Користувачу необхідно обрати параметри для запуску конвеєра:

– Гілка: цільова гілка коду для збірки (наприклад, monitoring, master або main);

– Pipeline ціль: необхідний етап або конфігурація розгортання (наприклад, Build).

Після вибору параметрів необхідно натиснути кнопку «ЗАПУСТИТИ» для старту (або «ВІДМІНИТИ» для закриття вікна). Застосунок відправить відповідну команду в GitLab API для початку процесу автоматичної збірки та оновлення мікросервісу.

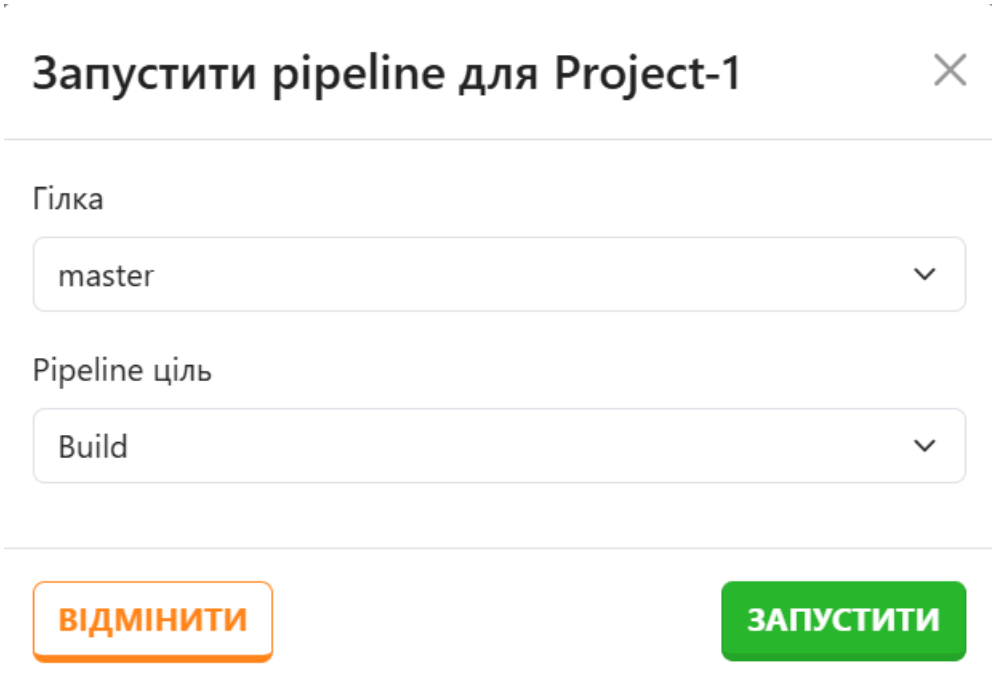


Рисунок Г.5 – Вікно ініціалізації процесу розгортання

ДОДАТОК Д ПРОГРАМА ТА МЕТОДИКА ВИПРОБОВУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1. Об'єкт випробувань

Об'єктом випробувань є розроблений програмний продукт — вебзастосунок для моніторингу версійності розгорнутих застосунків та їх розгортання за допомогою GitLab API та Gitlab CI. Програмне забезпечення призначене для контролю стану програмної інфраструктури, оптимізації роботи фахівців та забезпечення прозорості життєвого циклу застосунків.

2. Мета випробувань

Метою випробувань є перевірка працездатності розробленого вебзастосунку, перевірка алгоритмів автоматичного збору даних про фактичні версії розгорнутих мікросервісів, тестування механізмів візуальної індикації розбіжностей версій, а також підтвердження коректності інтеграції з GitLab API для дистанційної ініціації процесів розгортання (CI/CD).

3. Умови та порядок випробувань

Випробування проводяться на робочій станції розробника за таких умов:

- Серверна частина запущена локально (порт 8080) та має стабільне підключення до бази даних MongoDB.
- Клієнтська частина запущена у сучасному веббраузері (Google Chrome).
- Для тестування підготовлено тестовий мікросервіс із відомою актуальною версією та тестовий репозиторій у GitLab.
- У налаштуваннях профілю вебзастосунку введено валідний токен доступу (Personal Access Token) із правами взаємодії з інфраструктурою GitLab.

4. Методи випробувань (Тестові сценарії)

Тест 1. Перевірка управління логічними групами

- Дія: У лівому бічному меню натиснути «Створити групу проєктів», ввести назву нового середовища (наприклад, dev) та зберегти. Встановити цільову (еталонну) версію для створеної групи.
- Очікуваний результат: Група успішно зберігається в базі даних, відображається у бічному навігаційному меню та в центральній робочій області. Системні помилки відсутні.

Тест 2. Перевірка додавання проєкту та збору телеметрії

- Дія: Через контекстне меню створеної групи викликати модальне вікно «Створити проєкт». Заповнити поля (Назва, Url, Id проекта в GitLab, Url на GitLab) дійсними даними тестового мікросервісу та підтвердити створення.
- Очікуваний результат: Картка проєкту з'являється у робочій області групи. Застосунок успішно виконує HTTP-запит за вказаним Url, отримує телеметрію та коректно відображає на картці фактичну версію збірки.

Тест 3. Перевірка візуальної індикації розбіжності версій

- Дія: В інтерфейсі шапки групи змінити цільову (еталонну) версію на таку, що свідомо відрізняється від фактичної версії розгорнутого мікросервісу (наприклад, очікувана 1.0.1, а фактична 1.0.0).
- Очікуваний результат: Програма виявляє розбіжність і автоматично підсвічує фактичну версію на картці проєкту попереджувальним помаранчевим кольором. При наведенні курсора на підсвічену версію коректно відображається чорна спливаюча підказка з деталями (очікувана та фактична версії).

Тест 4. Перевірка інтеграції з GitLab API (отримання гілок)

- Дія: Натиснути кнопку ініціалізації розгортання (іконка ракети) на картці створеного проєкту.
- Очікуваний результат: Відкривається модальне вікно «Запустити pipeline». Застосунок успішно звертається до GitLab API, використовуючи

збережений токен, та завантажує випадаючий список актуальних гілок репозиторію (наприклад, monitoring, master).

Тест 5. Перевірка ініціації CI/CD пайплайну

- Дія: У модальному вікні запуску пайплайну обрати потрібну гілку коду, вказати ціль розгортання (Pipeline ціль, наприклад, Build) і натиснути кнопку «ЗАПУСТИТИ».
- Очікуваний результат: Серверна частина успішно формує та відправляє POST-запит до GitLab. Користувач отримує повідомлення (Toast-сповіщення) про успішний старт. В інфраструктурі GitLab ініціюється відповідний процес автоматичної збірки.

Тест 6. Обробка аварійних ситуацій (негативний тест)

- Дія: Вказати для проєкту недійсну веб-адресу (Url) та ввести недійсний GitLab Token у налаштуваннях профілю.
- Очікуваний результат: Застосунок продовжує стабільну роботу (не завершується аварійно). На картці проєкту відображається повідомлення про помилку підключення замість телеметрії, а при спробі завантажити гілки у модальному вікні виводиться повідомлення про помилку авторизації доступу до GitLab.