МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

ВАПЕЛЬНИК ТАРАС СЕРГІЙОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій
канд. техн. наук, доцент
_____ О. В. Зелінська
«____» _____ 20__ р.

РОЗРОБКА ДОДАТКУ ДЛЯ МОНІТОРИНГУ РЕПОЗИТОРІЮ ПРОГРАМНИХ ПРОЕКТІВ

Спеціальність 122 «Комп’ютерні науки»

Кваліфікаційна (бакалаврська) робота

Керівник:
Антонов Ю.С., кан. фіз.-мат. наук, доцент,
доцент кафедри інформаційних
технологій

_____
(Підпис)

Оцінка _____/_____/_____
(бали/за шкалою ЄКТС/за національною шкалою)
Голова ЕК: _____
(Підпис)

Вінниця 2025

АНОТАЦІЯ

Вапельник Т.С. Розробка додатку для моніторингу репозиторію програмних проєктів. Спеціальність 122 «Комп'ютерні науки». Освітня програма «Комп'ютерні науки». Донецький національний університет імені Василя Стуса, Вінниця, 2025.

У кваліфікаційній роботі описано розробку десктопного застосунку для інтеграції з GitLab Web API з метою моніторингу стану програмних репозиторіїв. Проведено аналіз функціоналу GitLab, REST-архітектури API та особливостей автентифікації. Реалізовано інтерфейс для виводу інформації про проєкти, коміти, гілки, задачі та запити на злиття користувача. Застосунок розроблено з використанням Java, Spring, Hibernate, Gson.

Ключові слова: GitLab, Web-API, моніторинг репозиторіїв, контроль версій, Java, JavaFX, Spring Framework, Hibernate, десктопний додаток.

51 с., 4 рис., 40 джерела.

Vapelnyk T.S. Development of an Application for Monitoring Software Project Repositories. Specialty 122 "Computer Science". Educational Program "Computer Science". Vasyl Stus Donetsk National University, Vinnytsia, 2025.

The qualification paper describes the development of a desktop application integrated with GitLab Web API for monitoring the state of software repositories. The study includes analysis of GitLab functionality, REST API architecture, and authentication methods. The application retrieves information about user projects, commits, branches, issues, and merge requests. It was implemented using Java, Spring, Hibernate, and Gson.

Keywords: GitLab, Web-API, repository monitoring, version control, Java, JavaFX, Spring Framework, Hibernate, desktop application.

51 p., 4 fig., 40 sources.

ЗМІСТ

ВСТУП

У сучасному світі стрімкого розвитку інформаційних технологій зростає потреба у засобах, що забезпечують ефективне управління програмними проектами. Одним із ключових інструментів у цій сфері є системи контролю версій, такі як GitLab — багатофункціональна платформа для спільної розробки програмного забезпечення. Завдяки широкому спектру можливостей GitLab активно використовується як індивідуальними розробниками, так і великими командами.

Зі збільшенням кількості проектів, гілок, комітів та запитів на злиття стає необхідним мати централізований інструмент, який дозволяє користувачеві швидко отримувати актуальну інформацію про стан власних репозиторіїв. Особливо важливою є можливість моніторингу не лише загального списку проектів, а й перегляду детальної інформації щодо кожного з них, зокрема змін у файлах, історії комітів, відкритих запитів на злиття та задач.

У межах цієї роботи було реалізовано застосунок, що інтегрується з GitLab через Web-API та дозволяє здійснювати моніторинг стану репозиторіїв. Після введення персонального токена користувача, додаток отримує інформацію про самого користувача, перелік його проектів, а також надає змогу переглянути деталі кожного з них. Серед доступної інформації — список гілок, останні коміти, активні запити на злиття, задачі та загальний стан файлів у проекті.

Перший етап розробки включав аналіз GitLab API, вивчення подібних рішень та визначення ключових вимог до застосунку. Подальші етапи охоплювали реалізацію інтерфейсу, логіки взаємодії з API та тестування функціональності.

Таким чином, ця робота демонструє, як засоби інтеграції з Web-API можуть бути використані для створення зручних і ефективних інструментів моніторингу, які сприяють покращенню процесу розробки, підвищенню прозорості та контролю над поточним станом проектів.

РОЗДІЛ 1
АНАЛІЗ ТА КЛАСИФІКАЦІЯ ІСНУЮЧИХ СИСТЕМ

1.1 Огляд платформи GitLab та її функціональних можливостей для управління програмними проектами

Призначення та архітектура GitLab — комплексна платформа для управління повним життєвим циклом розробки програмного забезпечення (DevOps). Проєкт GitLab був започаткований 2011 року українським розробником Дмитром Запорожцем разом з Валерієм Сизовим як відкрите рішення для спільної розробки на основі Git [1]. У 2012 році співзасновник Сід Сієрдбрандз запустив сервіс GitLab.com, надавши платформу як хмарний SaaS [2]. Головна ідея GitLab — об'єднати управління кодом, CI/CD, відстеження завдань, рецензії коду та інші DevOps-інструменти в одній системі. Компанія декларує місію «Everyone can contribute» («кожен може сприяти»), що відображає прагнення залучити широке коло учасників розробки. GitLab позиціонується як «DevOps Platform» — єдина платформа для всіх етапів розробки, від написання коду до доставки в продакшн [3-4].

Ключові технології: Архітектура системи GitLab побудована за принципами багаторівневої клієнт-серверної моделі. Веб-інтерфейс реалізовано з використанням фреймворку Ruby on Rails, що забезпечує взаємодію між користувачем і сервером. На рівні бізнес-логіки працює набір сервісів, зокрема Puma як сервер застосунку та Sidekiq для обробки фонових завдань (наприклад, опрацювання подій або черг на оновлення репозиторіїв). Дані зберігаються в реляційній базі даних PostgreSQL, тоді як тимчасові та чергові дані — у Redis. Операції з Git-репозиторіями обробляються сервісом Gitaly, реалізованим мовою Go, що забезпечує ефективний доступ до вмісту репозиторіїв. Для доступу до Git через SSH використовується GitLab Shell [5]. Frontend частина платформи розроблена з використанням Vue.js, що дозволяє створювати динамічний користувацький інтерфейс [6]. Така модульна архітектура забезпечує масштабованість і спрощує підтримку окремих компонентів системи.

Моделі розгортання: GitLab доступний як SaaS-сервіс (GitLab.com) і як самостійний продукт для інсталяції на власних серверах (self-managed). Для self-hosted версії є офіційні Omnibus-пакети для Linux, Docker-образи та Helm-чарти для розгортання в Kubernetes [7]. Платформу можна запускати безпосередньо на кластері Kubernetes, що спрощує деплой контейнеризованих додатків і масштабування. Таким чином, GitLab підтримує різні сценарії — від малого on-premise рішення до масштабних хмарних інстансів.

Репозиторії (Git): У GitLab кожен проект має власний Git-репозиторій для керування версіями вихідного коду. Підтримуються гілки, теги та pull/merge-запити. При кожному push у репозиторій чи створенні нового тегу може автоматично запускатися CI/CD пайплайн. Наприклад, створення тега може слугувати тригером для запуску тестів і деплою. Через веб-інтерфейс можна переглядати історію комітів, файли, відмінності та стан кожного файлу в різних гілках. Репозиторій служить основою для аналізу активності розробки: за кількістю комітів, пул-реквестів та змін у коді можна оцінити продуктивність команди [36].

CI/CD (безперервна інтеграція та доставка): GitLab має вбудований механізм CI/CD. Пайплайни описуються у файлі .gitlab-ci.yml і розбиваються на стадії (наприклад, build, test, deploy). Кожна стадія містить завдання (jobs), які виконуються агентами GitLab Runner. При пуші коду чи створенні merge request GitLab автоматично запускає конфігуровані конвеєри. Завдання можна виконувати паралельно, кроки кешувати, а результати (артефакти) передавати між стадіями. Інтерфейс показує статус кожного конвеєра і окремого job. CI/CD автоматизує побудову, тестування й розгортання додатків, що дає змогу оперативно виявляти помилки. Моніторинг ефективності CI/CD реалізується через такі метрики, як середній час побудови, частка успішних збірок, частота деплоїв тощо. Наприклад, GitLab обчислює показник Lead Time for Changes (час від коміту до продакшн), який прямо корелює з ефективністю конвеєрів — зменшення цього часу свідчить про підвищення продуктивності [8].

Issue Tracking: Вбудований трекер завдань (Issues) дозволяє планувати й відстежувати роботу: запити на нові функції, завдання розробників, баг-репорти тощо. Кожен issue може мати виконавців, дедлайни, мітки (labels) і статус здоров'я. GitLab підтримує організацію роботи через мілістони (розробка по релізах) та борди (Kanban-like), а також дозволяє обговорювати деталі прямо у задаче (threaded discussions). Issue можна зв'язувати з комітами чи merge request – при злитті MR пов'язаний issue автоматично закривається. Така інтеграція коду і тасків допомагає відстежувати прогрес: за кількістю відкритих/закритих issue, середнім часом їх закриття чи швидкістю зростання backlog можна оцінювати продуктивність команди та якість планування [9].

Merge Requests (MR): Merge Request – це механізм код-рев'ю та контролю змін. MR створюється, коли розробник хоче об'єднати свою гілку з основною. GitLab надає єдиний інтерфейс для перегляду змін (diff), обговорень і коментарів під час рев'ю . Можна налаштувати автоматичне блокування мерджу до проходження певної кількості оглядів чи успішного проходження CI. В опис MR можна вказати, чому потрібна зміна, і пов'язати MR з issue. Після мерджу GitLab може автоматично закрити відповідний issue, що забезпечує прозорість. Merge Request допомагають гарантувати якість коду – експерти та автоматичні інструменти перевіряють правки перед злиттям . Метрики, пов'язані з MR (наприклад, середній час на рецензію або кількість ітерацій), дають уявлення про швидкість та якість процесу розробки [10].

Моніторинг ефективності та якості: Усі ці компоненти інтегровані для надання аналітики DevOps. Наприклад, GitLab надає Value Streams Dashboard – вбудовану аналітичну панель, що відображає метрики продуктивності DevOps, ризиків безпеки та оптимізації процесів [11]. Також платформа обчислює DORA-метрики (частота деплоїв, lead time, MTTR тощо), дозволяючи виявляти вузькі місця й оцінювати динаміку поліпшень. Разом, дані з репозиторіїв, статуси CI/CD, статистика issue та MR дають змогу керівникам та командам моніторити ефективність (швидкість випусків, стабільність збірок) і якість процесів (частота кодових помилок, відповідність вимогам) в режимі реального часу.

GitLab створено з урахуванням принципів DevOps: злиття розробників та операцій, автоматизація, постійний зворотний зв’язок. Як єдина платформа GitLab забезпечує спільне середовище для планування, написання коду, тестування, релізу і моніторингу, що спрощує співпрацю й зменшує розриви між командами. Інструменти автоматизації (вбудований CI/CD), системи контролю якості (Code Quality, SAST/DAST-аналізи) та можливості для спільного рецензії коду (Merge Requests) активно сприяють ітераційному вдосконаленню. Особливу увагу GitLab приділяє інтеграції з іншими сервісами:

CI/CD та інструменти DevOps: підтримуються інтеграції з Jenkins, сповіщення в Slack, нотифікації в Jira тощо.

Управління ідентифікацією: GitLab підтримує єдину аутентифікацію через LDAP і SAML , що дозволяє організаціям безпечно інтегрувати платформу в існуючу інфраструктуру.

Контейнери: GitLab може сам розгортатися у Kubernetes-кластері та керувати деплоєм додатків у контейнерах (включно з Docker Registry, Auto DevOps тощо).

1.2 GitLab Web-API: архітектура, автентифікація та основні можливості для збору даних

GitLab REST API призначено для автоматизації роботи з ресурсами GitLab та інтеграції GitLab у зовнішні інструменти. За його допомогою можна керувати проектами, проблемами, запитами на злиття та CI/CD процесами програмно, без ручного втручання. Наприклад, GitLab REST API дозволяє створювати кастомні скрипти для масового адміністрування проєктів, інтегрувати дані GitLab у зовнішні додатки і централізовано керувати правами доступу [3][12].

GitLab API побудовано за REST-архітектурою: кожен ендпоінт відображається через URL і для взаємодії з ним використовуються стандартні HTTP-методи (GET, POST, PUT/PATCH, DELETE). Наприклад, метод GET повертає ресурс у форматі JSON (HTTP 200 OK), POST створює новий ресурс (201 Created), PUT або PATCH оновлює – знову ж повертає 200 OK, а DELETE

видаляє ресурс (204 No Content). Обмін даними здійснюється у JSON: в запитах часто передають JSON-тіло (для POST/ PUT), а у відповідях повертається JSON-об’єкт або масив об’єктів. Наприклад, запит GET /api/v4/projects поверне список проектів, де в JSON-об’єктах є поля id, name, visibility тощо. HTTP статус-коди відповіді дають змогу зрозуміти результат операції (успіх або помилку) [12-13].

Для більшості приватних запитів GitLab API вимагає автентифікації. Особисті токени доступу (Personal Access Tokens, PAT) – найпоширеніший метод. PAT можна створити у профілі користувача (Налаштування → Access Tokens): потрібно вказати ім’я токена, строк дії й встановити необхідні scopes (області дії). Наприклад, scope api дає повний доступ (читання/запис) до API всіх проектів і реєстрів, а read_repository – лише дозвіл на читання репозиторію по HTTPS. Після створення токену система показує його лише один раз, тому його слід зберігати в безпечному місці. Головні рекомендації – призначати мінімально необхідні права і короткий термін дії, щоб у разі викрадення мінімізувати ризик. PAT передається в заголовку запиту PRIVATE-TOKEN: або параметром private_token [14].

Альтернативні методи автентифікації:

OAuth2 (отримання access_token через OAuth-протокол) і CI_JOB_TOKEN [15].

OAuth2-методи дозволяють стороннім додаткам отримати токен доступу через стандартний потік OAuth 2.0 – його можна передати в параметрі access_token або в заголовку Authorization: Bearer.

CI_JOB_TOKEN – це тимчасовий токен, автоматично створений під час запуску CI/CD-пайплайна. Він дійсний лише протягом виконання роботи, успадковує права користувача, що тригернув збірку, але має доступ лише до частини ресурсів (контейнерний реєстр, пакети, деякі ендпоінти). У коді конвеєра CI_JOB_TOKEN передається через змінну середовища CI_JOB_TOKEN , і ним можна автентифікуватися, наприклад, так:

curl --header "JOB-TOKEN: $CI_JOB_TOKEN" "https://gitlab.example.com/api/v4/projects/123/repository/files" [16].

GitLab API містить кілька логічних груп ендпоінтів, корисних для отримання статусу і даних репозиторію. У кожній з цих груп API можна застосувати додаткові параметри фільтрації та сортування (наприклад, пошук по назві, відбори за гілкою чи статусом), але основний формат запитів і відповідей є схожим. Нижче наведено короткий огляд основних з них з прикладами запитів і відповідей.

Проекти (Projects API) – керування проектами та їх налаштуваннями. Тут можна отримати список чи інформацію про конкретний проект, створити чи видалити проект, змінити налаштування видимості тощо. Наприклад, GET /projects/:id повертає JSON з полями проекту. Наприклад

```
curl --header "PRIVATE-TOKEN: <token>" \
     --url "https://gitlab.example.com/api/v4/projects/3"
```

має таку відповіді:

```
{
"id": 3,
"description": "Lorem ipsum dolor sit amet, consectetur adipiscing elit.",
"default_branch": "main",
"visibility": "private",
"ssh_url_to_repo":"git@example.com:diaspora/diaspora-project-site.git",
"http_url_to_repo":"http://example.com/diaspora/diaspora-project-site.git",
"web_url": "http://example.com/diaspora/diaspora-project-site",
"name": "Diaspora Project Site",
"path": "diaspora-project-site",
. . .
}
```

Тут id – ідентифікатор проекту в усьому gitlab, default_branch – гілка за замовчуванням, visibility – рівень доступності, name та path - ім'я та шлях, а також URL-адреси SSH/HTTP репозиторію [17].

Коміти (Commits API) – отримання інформації про коміти в репозиторії. Наприклад, GET / projects/:id/repository/commits поверне список комітів певної гілки. Наприклад

```
curl --header "PRIVATE-TOKEN: <token>" \
     --url "https://gitlab.example.com/api/v4/projects/5/repository/commits"
```

має таку відповіді:

```
[
{
"id": "ed899a2f4b50b4370feeea94676502b42383c746",
"short_id": "ed899a2f4b5",
"title": "Replace sanitize with escape once",
"author_name": "Example User",
"author_email": "user@example.com",
"authored_date": "2021-09-20T11:50:22.001+00:00",
. . .
},
. . .
]
```

Тут кожен об’єкт має SHA (id), повідомлення (title), ім’я та email автора, дату авторства (authored_date) тощо [18].

Гілки (Branches API) – інформація про гілки в проєкті. Команда GET /projects/:id/ repository/branches повертає перелік гілок. Наприклад

```
curl --header "PRIVATE-TOKEN: <token>" \
     --url "https://gitlab.example.com/api/v4/projects/5/repository/branches"
```

має таку відповіді:

```
[
{
"name": "main",
"merged": false,
"protected": true,
"default": true,
"can_push": true,
"web_url":"https://gitlab.example.com/my-group/my-project/-/tree/main",
"commit": {
"id": "7b5c3cc8be40ee161ae89a06bba6229da1032a0c",
```

```
"short_id": "7b5c3cc",
"created_at": "2024-06-28T03:44:20-07:00",
. . .
}
},
. . .
]
```

У цьому JSON видно ім’я гілки (name), чи змерджена вона (merged), чи є захищеною (protected , default), а також вкладений об’єкт commit з даними останнього коміту (SHA, автор, дата та ін.) [19].

Запити на злиття (Merge Requests API) – робота з merge request. Наприклад, GET / projects/:id/merge_requests повертає список MR. Наприклад

```
curl --header "PRIVATE-TOKEN: <token>" \
     --url "https://gitlab.example.com/api/v4/projects/1/merge_requests"
```

має таку відповіді:

```
[
{
"id": 1,
"iid": 1,
"project_id": 3,
"title": "test1",
"description": "fixed login page css paddings",
"state": "merged",
"created_at": "2017-04-29T08:46:00Z",
"updated_at": "2017-04-29T08:46:00Z",
"target_branch": "main",
"source_branch": "test1",
"author": {"id": 1, "name": "Administrator", . . .}
. . .
},
. . .
]
```

Тут id – глобальний ідентифікатор MR, iid – локальний номер у проекті, state – статус (merged , opened тощо), гілки-джерело та цілі (source_branch , target_branch), автор (author) та часові мітки створення/оновлення [10].

Задачі (Issues API) – робота з issue. Ендпоінт GET /projects/:id/issues повертає список задач проекту. Наприклад

```
curl --header "PRIVATE-TOKEN: <token>" \
     --url "https://gitlab.example.com/api/v4/projects/4/issues"
```

має таку відповіді:

```
[
{
"id" : 41,
"iid" : 1,
"project_id" : 4,
"title" : "Ut commodi ullam eos dolores perferendis nihil sunt.",
"description" : "Omnis vero earum sunt corporis dolor et placeat.",
"state" : "closed",
"author": {"id" : 1, "name" : "Administrator", [...]},
"labels" : ["foo", "bar"],
"upvotes": 4,
"downvotes": 0,
"created_at" : "2016-01-04T15:31:46.176Z",
"closed_at" : "2016-01-05T15:31:46.176Z",
. . .
},
. . .
]
```

Відповідь містить id , title , опис, state (відкрито/закрито), автора, мітки, кількість голосів, дати створення/закриття тощо [9].

GitLab API має певні обмеження і особливості, про які слід пам’ятати. По-перше, всі запити підлягають обмеженню швидкості (rate limiting). Якщо клієнт перевищує ліміт запитів, API повертає HTTP-статус 429 Too Many Requests.

По-друге, пагінація: більшість запитів повертають не всі записи одразу, а розбивають результат на сторінки. За замовчуванням повертається 20 записів на сторінку. У запиті можна задати параметри page (номер сторінки) і per_page (кількість елементів, макс. 100). У відповідь сервер відправляє HTTP-заголовки або Link -header із посиланнями на попередню/наступну сторінки. Наприклад,

щоб отримати другу сторінку коментарів (по 3 елементи на сторінку), можна вказати ?per_page=3&page=2 [12].

Нарешті – обробка помилок. При помилкових запитах GitLab повертає відповідний HTTP-код і JSON з полем message. Наприклад, якщо бракує обов'язкового параметра, повертається 400 Bad Request з повідомленням про відсутнє поле. Якщо токен не передано або він недійсний – 401 Unauthorized. 403 Forbidden вказує, що запит не дозволено (наприклад, недостатньо прав). Якщо ресурс не знайдено або користувач не має до нього доступу – 404 Not Found. Інші помилки: 409 Conflict (конфлікт створення), 422 Unprocessable Entity (непридатні дані), 500-503 (помилки сервера). Перевищення лімітів – 429 Too Many Requests . Тіло відповіді зазвичай містить JSON на кшталт {"message": "400 (Bad request) \"title\" not given"} для опису проблеми, або об'єкт з деталями для кожного поля [13].

1.3 Концепція моніторингу стану програмних репозиторіїв та аналіз існуючих інструментів та підходів до моніторингу GitLab репозиторіїв

Моніторинг репозиторіїв має на меті підтримку процесу розробки та доставки коду на високому рівні. Наприклад, слідкування за кількістю та темпом створення комітів і запитів на злиття допомагає оцінити продуктивність команди та наскільки швидко рухається робота над функціоналом. Моніторинг стабільності CI/CD (успіхів/помилок збірок) – гарантує, що зміни не ламають збірку, а тривалість та частота запусків пайплайнів вказують на ефективність процесу перевірки коду. Також відстежують стан задач (issues): за допомогою метрик «відкрито/закрито» можна оцінити навантаження на команду, а аналіз часу вирішення задач – швидкість реакції на запити чи помилки. В результаті головними завданнями моніторингу є контроль якості коду, оперативне виявлення проблем (наприклад, уповільнень у процесах або падінь збірок) та оптимізація процесів доставки ПЗ.

Метрики активності розробників. Наприклад, число комітів у репозиторії можна отримати через GitLab API: запитом GET /projects/:id/repository/commits

повертається список об’єктів комітів. Аналізуючи дату (committed_date) та автора кожного коміту, можна обчислити загальну кількість змін за період або інтенсивність роботи кожного розробника. Ще один показник – кількість та стан запитів на злиття (Merge Requests). Через GET /projects/:id/merge_requests отримуємо дані MR. Наприклад, фільтруючи за state=open чи state=merged , можна порахувати відкриті/закриті MR. Частота гілок оцінюється через GET / projects/:id/repository/branches, який повертає список гілок та інформацією про останній коміт у цій гілці. Таким чином, метрика «кількість активних гілок» формується за числом об’єктів у цьому списку, а аналіз даних про останній коміт (created_at, author_name тощо) дає уявлення про активність у гілках.

Метрики задач (issues). Через API задач (GET /projects/:id/issues або глобально /issues) можна отримати список issue. Наприклад, фільтруючи за state=opened чи state=closed, рахуємо кількість відкритих та закритих задач. Також, використовуючи поля created_at і closed_at, можна обчислити час вирішення задач (затримку від створення до закриття) та середній час розв’язання. Ці дані дозволяють оцінювати якість планування (чи закриваються задачі вчасно) та продуктивність реагування на проблеми.

GitLab надає широкий набір вбудованих аналітичних інструментів. Зокрема, CI/CD Analytics дає змогу переглядати статистику конвеєрів збірок: розподіл успішних/невдалих пайплайнів, медіану та їхньою тривалості тощо (ці налаштування можна фільтрувати за гілкою, періодом та іншими параметрами). У вкладці CI/CD у веб-інтерфейсі GitLab візуалізується кількість запусків, коефіцієнт успішності і «тривалісний» графік пайплайнів, що допомагає швидко відслідковувати тенденції в продуктивності [8].

Value Stream Analytics (VSA) в GitLab реалізовує аналіз повного циклу розробки – від постановки задачі до її розгортання. VSA вимірює час «ідея→продукт» шляхом відстеження етапів (стадій) у життєвому циклі задач і MR. Це дозволяє виявити затримки та «вузькі місця» в процесі розробки: наприклад, скільки днів у середньому займає код-рев’ю чи затримка між мержем MR та деплоєм. GitLab заявляє, що VSA показує “час від ідеї до релізу” та

допомагає візуалізувати DevSecOps-робочі процеси, знаходити і вирішувати неефективності. Важливо: у безкоштовній версії GitLab проєкт-level VSA обмежена – доступні лише стандартні стадії, без можливості створювати власні сценарії або додавати DORA-метрики. Тобто просунуті можливості VSA (кастомні стадії, деталізовані звіти) є лише в преміум-рівнях [20].

Окрім того, GitLab пропонує Issue/MR Analytics: графіки кількості створених та закритих issue щомісяця (Issue Analytics), статистику середнього часу до злиття MR (Merge Request Analytics), інформацію про внески розробників (Contributor/Contribution Analytics) та інші метрики продуктивності команди. Є також Repository Analytics – наприклад, діаграми розподілу мов програмування у кодовій базі та показники покриття тестами. Усі ці інструменти вбудовані в GitLab і не вимагають додаткової налаштування за межами проекту, однак їхня функціональність фрагментована по різних вкладках. Серед обмежень: переважно фіксований набір метрик і статичних дашбордів (більшість елементів, «належних GitLab», не можна редагувати, лише копіювати чи налаштовувати окремі кастомні дашборди). Таким чином, GitLab Analytics корисна для швидких оцінок, але може бути недостатньо гнучкою для глибокого налаштування метрик під специфічні потреби [11][21].

Існує низка сторонніх інструментів, які інтегруються з GitLab для моніторингу. Grafana часто використовується разом із Prometheus: GitLab має готові експортні метрики, які Prometheus збирає (через GitLab Exporter або вбудовані експортери) [22]. Далі Grafana будує з цих даних дашборди. Наприклад, Grafana Agent (інтеграція GitLab для Grafana Cloud) може збирати метрики GitLab (HTTP-запити, швидкість створення пайплайнів тощо) і виводити їх на готові дашборди. Такий підхід дає високу гнучкість візуалізації – будь-які дані Prometheus можна вивести на графіки чи гістограми. До недоліків – доволі складне налаштування (потрібно розгорнути Prometheus, конфігурувати експортери та формувати запити) та необхідність підтримувати інфраструктуру моніторингу. Перевага – відсутність прямої плати за софтвер

(Grafana/Prometheus – open-source), а також можливість вбудованого alerting через Prometheus Alertmanager [24].

Prometheus сам по собі – потужна СУБД метрік. Він збирає часо́ві ряди з GitLab (мітрики CPU, memory, job-метрики тощо) і може оповіщати про тривалі збої. GitLab «з коробки» моніториться Prometheus: у офіційній документації сказано, що GitLab постачає готовий стек Prometheus та експортери для внутрішніх сервісів. Тобто Prometheus-інтеграція найкраще підходить для DevOps-метрик продуктивності GitLab, але може бути використана і для клієнтських метрик (через GitLab Exporter, наприклад) [23].

Datadog – комерційне SaaS-рішення, яке має інтеграцію з GitLab. Воно може збирати метрики як через API GitLab (наприклад, метадані про репозиторії), так і через Prometheus-ендпоїнти GitLab. Datadog GitLab Integration «вирівнює» більшість метрик GitLab та Gitaly через Prometheus і надає готові графіки, а також модуль «CI Pipeline Visibility», що дає глибокий аналіз пайплайнів (продуктивність, історія, вузькі місця). Інтеграція Datadog досить проста – достатньо встановити Datadog Agent і налаштувати GitLab Check – проте вона потребує оплати ліцензії Datadog. Серед переваг – масштабованість, інтеграція з іншими DevOps-метриками (трейсування, логи) та вбудовані алерти. Мінус – залежність від зовнішнього сервісу і висока вартість при великому масштабі [25].

Gitential – комерційний сервіс зосереджений на аналітиці продуктивності команд розробників. Він під’єднується до GitLab (через API чи інтеграцію) і аналізує активність розробників: число комітів, пул-реквестів, рев’ю, а також оцінює «Velocity» та якість співпраці. Gitential дозволяє ідентифікувати уповільнення у процесі (наприклад, затримки на певних етапах) і відображає графіки «прочиненості» роботи команди. Плюс системи – глибока фокусування на метриках продуктивності та можливість корисного «їнжинірингу продуктивності». Недоліки – це SaaS-рішення, яке потребує оплати (передбачено як хмарний сервіс або on-premise) і концентрується тільки на даних комітів/запитів на злиття (а не дає загальних CI/CD метрик) [26].

Висновок до першого розділу

У даному розділі було проведено комплексний аналіз предметної області, пов'язаної з моніторингом стану програмних репозиторіїв на платформі GitLab. Детально розглянуто функціональні можливості GitLab як інтегрованої DevOps-платформи, зокрема її ключові компоненти: управління репозиторіями, CI/CD, систему відстеження задач та механізм Merge Requests. Особливу увагу приділено дослідженню GitLab Web-API як основного інструменту для програмного доступу до даних, включаючи його архітектуру, методи автентифікації та основні групи ендпоінтів.

Визначено та класифіковано ключові метрики, що відображають стан репозиторіїв та ефективність процесів розробки, такі як показники активності розробників, стабільність та швидкість CI/CD пайплайнів, динаміка управління задачами.

Проведений аналіз існуючих рішень для моніторингу GitLab, включаючи вбудовані аналітичні інструменти платформи та популярні сторонні системи (Grafana, Prometheus, Datadog, Gitential), виявив як їхні переваги, так і суттєві недоліки. Встановлено, що наявні інструменти часто є або надто складними в налаштуванні та підтримці, або мають обмежену гнучкість кастомізації, або є комерційними продуктами з високою вартістю. Це обґрунтовує актуальність розробки спеціалізованого, легкого у використанні веб-додатку, що надає швидкий доступ до найважливіших метрик моніторингу GitLab репозиторіїв через інтуїтивно зрозумілий інтерфейс.

РОЗДІЛ 2
ВИЗНАЧЕННЯ ВИМОГ ДО ПРОГРАМОГО ЗАБЕЗПЕЧЕННЯ ТА ПОБУДОВА МОДЕЛІ ПРОГРАМНОГО ПРОДУКТУ

2.1 Визначення ключового функціоналу додатку

Успішна розробка програмного забезпечення значною мірою залежить від чіткого визначення вимог до системи. На цьому етапі проєктування формулюються як функціональні, так і нефункціональні вимоги до застосунку для моніторингу GitLab-репозиторіїв. Вимоги базуються на результатах аналізу існуючих систем, виявлених потреб користувачів і специфіці предметної області.

Функціональні вимоги описують поведінку системи та функціональність, яку вона повинна забезпечувати. Для запропонованого додатку основні функції охоплюють автентифікацію користувача, доступ до даних репозиторіїв, їх фільтрацію та візуалізацію. Нижче наведено перелік ключових функціональних можливостей системи.

1. Автентифікація користувача за допомогою персонального токена GitLab.

Користувач, який бажає здійснювати моніторинг своїх проектів у GitLab за допомогою створеного застосунку, повинен попередньо згенерувати персональний токен доступу (*Personal Access Token*) у своєму обліковому записі на GitLab. Цей токен забезпечує аутентифікацію й дозволяє програмі взаємодіяти з GitLab API від імені користувача. Процес авторизації є першим етапом роботи з додатком і є обов’язковою умовою для подальшої взаємодії з репозиторіями.

Після запуску програми користувач бачить інтерфейс із полем для введення персонального токена. Ввівши його, користувач підтверджує намір увійти до системи. У цей момент програма виконує HTTP-запит до GitLab API для перевірки достовірності наданого токена. Якщо відповідь від API вказує на успішну автентифікацію (наприклад, повертається об’єкт користувача у форматі JSON з базовою інформацією про нього), додаток вважає токен дійсним.

У разі позитивної відповіді користувач отримує доступ до головного інтерфейсу застосунку, де вже може переглядати власні проекти. У цьому інтерфейсі передбачається завантаження даних за допомогою того самого токена без потреби повторної авторизації.

Якщо введено недійсний токен — наприклад, помилковий, з недостатніми правами доступу або той, у якого закінчився термін дії, — програма відображає повідомлення про помилку та пропонує повторити спробу. Так само, у випадках, коли немає підключення до мережі або GitLab API тимчасово недоступний, користувач буде проінформований про неможливість перевірки токена на даний момент.

Таким чином, цей сценарій забезпечує надійний і безпечний механізм ідентифікації користувача, який є основою для роботи з іншими модулями програми.

2. Завантаження та відображення списку доступних користувачеві репозиторіїв.

Після успішної авторизації та переходу до головного інтерфейсу застосунку користувач очікує побачити перелік своїх репозиторіїв, до яких він має доступ на GitLab. Цей список є основною інформаційною базою, з якою користувач буде працювати у рамках подальших дій, таких як перегляд комітів, гілок або запитів на злиття. Застосунок автоматично ініціює процес завантаження репозиторіїв, використовуючи збережений або щойно введений персональний токен для автентифікованого звернення до GitLab API.

На цьому етапі додаток надсилає GET-запит до відповідної кінцевої точки GitLab API /projects?membership=true, вказуючи параметри, що обмежують вибірку лише тими проектами, до яких має доступ авторизований користувач. Якщо таких репозиторіїв багато, реалізується обробка пагінації: додаток послідовно виконує запити для отримання наступних сторінок, доки весь список не буде завантажено.

Отримані дані, як правило, подаються у форматі JSON, де кожен проект представлено як об’єкт. Ці об’єкти перетворюються на внутрішні Java-класи,

наприклад, Project, і додаються до колекції, яка слугує джерелом даних для інтерфейсу користувача.

У графічному інтерфейсі цей список зазвичай відображається у вигляді таблиці або списку, де кожен елемент містить коротку інформацію про проект. Для зручності передбачено сортування по рівню доступу, а також вибору конкретного проекту для переходу до його детального перегляду. У разі відсутності репозиторіїв користувачеві відображається відповідне повідомлення з поясненням.

Якщо під час завантаження виникає помилка (наприклад, мережевий збій, недоступність API, або токен втратив чинність), система повідомляє про проблему і надає можливість повторити спробу. Таким чином, цей сценарій є ключовим з точки зору ініціалізації основних даних для подальшої роботи користувача в межах програми.

3. Відображення детальної інформації про проект. Користувач, який успішно увійшов у додаток і завантажив список своїх проектів із GitLab, має можливість переглянути детальну інформацію про будь-який проект зі свого списку. Інтерфейс програми відображає перелік доступних користувачу проектів із їх назвами.

При виборі конкретного проекту зі списку відкривається окреме вікно або панель, що містить детальну інформацію про обраний проект. У цьому вікні користувач може побачити основні відомості про проект, такі як опис, кількість відкритих проблем, останні коміти, гілки, дата останнього оновлення та інші важливі метадані, які отримуються безпосередньо з GitLab API.

Такий підхід дозволяє користувачу швидко отримати повний обсяг інформації про проект без необхідності завантажувати всі дані одночасно. Це підвищує зручність використання програми та оптимізує завантаження даних, зменшуючи навантаження на мережу та систему.

Процес отримання детальної інформації відбувається асинхронно: після вибору проекту програма надсилає запит до GitLab API, і поки дані завантажуються, користувачу відображається індикатор завантаження або

повідомлення про очікування. У разі виникнення помилки, наприклад, через проблеми з підключенням або авторизацією, користувачу виводиться відповідне повідомлення з поясненням.

Користувач може закрити вікно з деталями проекту і повернутися до списку, щоб переглянути інформацію про інші проекти за потреби. Такий сценарій забезпечує інтуїтивну навігацію та зручний доступ до ключових даних проектів для ефективного моніторингу.

4. Збір даних через GitLab API за обраними репозиторіями. Після вибору користувачем проекту для моніторингу, додаток автоматично ініціює процес збору актуальної інформації про стан цих проектів через GitLab API. Основна мета цього процесу — отримати найсвіжіші дані, які допоможуть користувачу відслідковувати поточний стан розробки та ключові події у вибраних репозиторіях.

Додаток послідовно надсилає запити до різних ендпоінтів GitLab API, щоб отримати комплексну інформацію про кожен проект. Зокрема, збираються дані про історію комітів: кожен коміт включає дату внесення змін, інформацію про автора, а також текстове повідомлення, яке описує суть змін. Ці дані дозволяють користувачу простежити хронологію змін у коді та оцінити активність розробників.

Крім того, програма отримує список усіх гілок проекту, із позначенням основної (default) гілки, що є важливим для розуміння структури розробки і визначення актуальної версії коду. Це дає змогу контролювати, які гілки активно підтримуються або знаходяться у процесі розробки.

Також збираються дані про запити на злиття (merge requests) — їхній статус (наприклад, відкриті, закриті, прийняті) та автори. Ця інформація є критичною для відстеження процесу рецензування коду, колаборації між розробниками та планування подальших кроків у проекті.

Окрім цього, додаток отримує структуру проекту, тобто дерево файлів і папок, що містяться у репозиторії. Це допомагає користувачу краще орієнтуватися у організації проекту, перевіряти наявність ключових файлів або

каталогів без необхідності займатися додатковим оглядом у зовнішніх інструментах.

Весь процес збору даних здійснюється асинхронно, щоб не блокувати інтерфейс користувача. Програма реалізує обробку можливих помилок, таких як тайм-аути або проблеми з авторизацією, та повідомляє користувача про статус операції. Після отримання та обробки даних інтерфейс оновлюється, демонструючи актуальний стан репозиторіїв, що дозволяє ефективно моніторити хід розробки.

5. Візуалізація зібраних даних у зручному форматі. Інформація, отримана з GitLab API, має бути представлена користувачеві у зрозумілому вигляді. Планується використовувати табличне та списокове представлення даних. Наприклад, для комітів — таблиця з колонками "Автор", "Дата", "Повідомлення"; для задач — список із позначенням статусу. За потреби можуть бути реалізовані прості графіки для візуалізації динаміки комітів або активності у задачах.

6. Додаткові можливості (за наявності):

Локалізація інтерфейсу — підтримка принаймні української та англійської мов.

Групування репозиторіїв — за групами, рівнями доступу або активністю.

Періодичне оновлення даних — автоматичне оновлення з певним інтервалом.

Сповіщення — реалізація системи оповіщення користувача про зміни (наприклад, завершення CI/CD, нові задачі).

Нефункціональні вимоги визначають якісні характеристики системи, які впливають на її надійність, продуктивність, безпеку та зручність використання. Вони є критично важливими для забезпечення стабільної роботи застосунку в реальних умовах.

1. Продуктивність.

Інтерфейс програми повинен реагувати на дії користувача без помітних затримок. Запити до GitLab API мають виконуватись асинхронно, аби не

блокувати основний потік UI. Затримка від моменту вибору репозиторію до відображення перших даних не повинна перевищувати декількох секунд (залежно від мережевих умов).

2. Надійність.

Система повинна мати вбудовані механізми обробки помилок, зокрема:

- обробку ситуацій втрати інтернет-з’єднання;
- повідомлення про помилки автентифікації (невалідний токен);
- повідомлення про внутрішні помилки GitLab API або невідповідність формату даних.

3. Безпека.

Якщо токен користувача зберігається локально, це має бути реалізовано з урахуванням захисту даних. Зокрема, рекомендується зберігати токен у зашифрованому вигляді або використовувати тимчасове збереження лише на час сесії. Усі запити до API повинні здійснюватися виключно через безпечні протоколи (HTTPS).

4. Зручність використання (Usability).

Інтерфейс має бути інтуїтивно зрозумілим навіть для користувачів, які не мають технічної освіти. Важливими критеріями є:

- послідовна структура вікон і елементів;
- зрозумілі повідомлення про помилки;
- мінімальна кількість дій для виконання основних функцій;
- можливість швидко повернутися до попередніх кроків або змінити вибір.

5. Масштабованість.

Очікується, що користувач може працювати з кількома десятками або сотнями репозиторіїв без помітної втрати продуктивності. Архітектура має забезпечувати можливість розширення функціоналу без повної перебудови логіки.

2.2 Вибір архітектури програмного продукту

При проектуванні програмного забезпечення надзвичайно важливою складовою є архітектура системи — набір структурних рішень, які визначають логіку її побудови, організацію взаємодії між компонентами, принципи обробки та передачі даних. Вибрана архітектура безпосередньо впливає на підтримуваність, гнучкість, розширюваність, продуктивність та зручність супроводу програмного продукту в майбутньому.

Для застосунку — системи моніторингу GitLab-репозиторіїв — краще реалізувати архітектуру за принципами багаторівневої (багатошарової) моделі. Такий архітектурний стиль передбачає логічний поділ системи на окремі шари, кожен з яких виконує визначений набір функцій та взаємодіє з іншими шарами через чітко окреслені інтерфейси [37]. Зокрема, застосунок буде поділено на три основні рівні: рівень представлення (інтерфейс користувача), рівень логіки (обробка даних та бізнес-правила) і рівень доступу до даних (взаємодія з API GitLab та з базою даних).

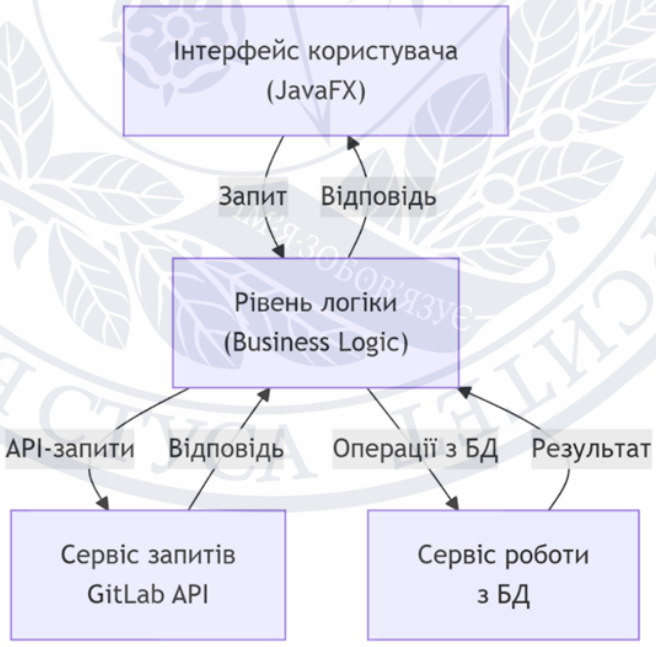

Рисунок 2.1 – Архітектура додатку для моніторингу GitLab-репозиторіїв

Рівень представлення. На цьому рівні можна реалізувати JavaFX-користувацький інтерфейс. Основним класом інтерфейсу доцільно передбачити MainWindow, який включатиме елементи графічного відображення, поділені на частини: панель навігації, таблиці для відображення репозиторіїв, гілок, користувачів та іншої інформації.

Усі взаємодії користувача з інтерфейсом (наприклад, вибір проекту, натискання кнопок, введення токена) можуть передаватися до відповідних сервісів через обробники подій. Таким чином, інтерфейс не повинен містити бізнес-логіки, а лише виконуватиме функцію візуалізації та маршрутизації взаємодій.

Рівень логіки застосунку. Цей рівень має забезпечити реалізацію бізнес-логіки: обробку даних, отриманих із зовнішніх джерел (зокрема, GitLab API), а також взаємодію з базою даних. Доцільно передбачити реалізацію окремих сервісів, таких як:

UserService, ProjectService, BranchService, CommitService, MergeRequestService, ExternalUserService, ProjectFileService, UserProjectService — кожен із яких відповідатиме за логічну обробку даних певної сутності та за взаємодію з відповідним репозиторієм і класом GitRequest. Наприклад, UserService може виконувати отримання даних користувача з API за токеном, перевірку його наявності у локальній базі та, за потреби, оновлення або створення нового запису.

Усі сервіси, ймовірно, викликатимуть методи класу GitRequest, який повертатиме оброблені об'єкти JsonObject. Отже, безпосередній парсинг відповіді API може залишатися всередині відповідного сервісу без винесення в окрему службу.

Рівень доступу до даних. Цей рівень може включати компоненти, відповідальні за збереження та отримання інформації з бази даних, а також за взаємодію з GitLab API.

GitRequest - окремий клас, який може бути призначений для виконання запитів до GitLab API. Його завдання - інкапсуляція логіки формування запитів, обробки відповідей, перевірки токенів, обробки помилок тощо.

ProjectRepository, UserRepository, CommitRepository, BranchRepository, MergeRequestRepository, ExternalUserRepository, ProjectFileRepository, UserProjectRepository - ці компоненти можна реалізувати для організації доступу до локальної бази даних відповідно до об’єктів доменної моделі.

Кожен репозиторій може взаємодіяти з відповідним сервісом та забезпечувати зберігання даних у потрібному форматі.

Модель даних. У межах проекту доцільно використати набір сутностей, які представляють об’єкти з GitLab:

- Project — інформація про репозиторій: назва, ID, група, URL;
- User — інформація про користувача програми: ім’я, ID, email;
- Branch — гілка репозиторію: назва, дата створення;
- Commit — коміт: хеш, автор, повідомлення, дата;
- MergeRequest — запит на злиття: автор, опис, статус;
- UserProject — зв’язок між користувача та його проектами, рівень доступу;
- ProjectFile — файли в проекті (за потреби);
- ExternalUser — учасники проектів, які присутні у логах.

Ці об’єкти варто описати у вигляді класів (ентіті), які застосовуватимуться у логіці та при роботі з базою даних. Для кожного класу потрібно визначити основні поля, що відповідатимуть структурі даних, які можна отримати через GitLab API.

Такий підхід є виправданим з кількох причин. По-перше, багаторівнева архітектура забезпечує чітке розділення відповідальностей між компонентами, що дозволяє зосереджуватись на конкретному аспекті системи без впливу на інші. Наприклад, змінюючи вигляд інтерфейсу, немає потреби втручатись у логіку роботи з API або у структуру моделей даних.

По-друге, це спрощує тестування та налагодження окремих частин програми, оскільки кожен рівень може бути протестований незалежно. Це особливо важливо в системах, де частина даних надходить з віддалених джерел, таких як GitLab API, і потребує перевірки обробки нештатних ситуацій (некоректні токени, помилки з'єднання тощо).

По-третє, така архітектура дозволяє легко масштабувати систему або змінювати окремі її частини. Наприклад, замість GitLab можна буде підключити підтримку інших платформ (GitHub, Bitbucket) без зміни інтерфейсу або логіки представлення даних. Аналогічно, перехід з JavaFX на інший фреймворк інтерфейсу не вимагатиме переписування бізнес-логіки.

Також обраний підхід сприяє розширюваності проекту в майбутньому. В межах поточної реалізації застосунок буде функціонувати як настільна JavaFX-програма, але його логіка може бути використана повторно в мобільному або веб-інтерфейсі. Це можливо завдяки тому, що бізнес-логіка не залежить від інтерфейсу, а вся взаємодія з GitLab реалізована через окремі сервіси.

Таким чином, використання багаторівневої архітектури є обґрунтованим з точки зору гнучкості, зручності підтримки та перспективи масштабування застосунку.

Висновок до другого розділу

Другий розділ заклав концептуальну основу для розробки системи моніторингу GitLab-репозиторіїв, зосередившись на визначенні вимог до програмного продукту та виборі оптимальної архітектури для його реалізації.

Спочатку було сформульовано ключовий функціонал та нефункціональні вимоги. До основних функціональних можливостей було віднесено автентифікацію користувача за допомогою GitLab-токена, завантаження та відображення списку доступних йому репозиторіїв. Також система повинна надавати детальну інформацію про обраний проект, включаючи історію комітів, стан гілок, запити на злиття та структуру файлів, і візуалізувати зібрані дані у зручному для користувача форматі. Було передбачено ймовірність реалізації

додаткових можливостей, як-от локалізація інтерфейсу та періодичне автоматичне оновлення даних. Окреслено важливі нефункціональні аспекти: висока продуктивність, що включає швидку реакцію інтерфейсу та асинхронне виконання запитів; надійність, що передбачає обробку можливих помилок з'єднання, автентифікації та відповідей від API; безпека, зокрема захист персонального токена користувача; зручність використання, орієнтована на інтуїтивно зрозумілий інтерфейс; та масштабованість, що враховує роботу зі значною кількістю репозиторіїв і потенціал для подальшого розширення функціоналу. Цей комплекс вимог створив чітке бачення очікуваної поведінки та якісних характеристик майбутньої системи.

Далі було здійснено вибір та обґрунтування архітектури програмного продукту. Для досягнення гнучкості, легкості підтримки та можливості подальшого розширення було обрано багаторівневу (багатошарову) модель. Така архітектура передбачає логічний поділ системи на три основні, чітко розмежовані рівні. Рівень представлення, для реалізації якого планується використати JavaFX, відповідатиме за користувацький інтерфейс та всю взаємодію з користувачем, передаючи його команди до логічного шару. Рівень логіки застосунку концентруватиме основну бізнес-логіку, реалізовану через набір спеціалізованих сервісів (наприклад, сервіси для роботи з користувачами, проектами тощо), які оброблятимуть дані, отримані з GitLab API, та керуватимуть взаємодією з рівнем доступу до даних. Рівень доступу до даних візьме на себе відповідальність за всю взаємодію з зовнішнім GitLab API (через спеціальний клас для запитів) та роботу з локальною базою даних (через відповідні класи-репозиторії). Також було визначено ключові сутності моделі даних (Проект, Користувач, Коміт та інші), що будуть представляти об'єкти предметної області. Вибір такої архітектури було обґрунтовано перевагами, які вона надає, зокрема, чітким розподілом відповідальностей між компонентами, спрощенням процесу тестування та можливістю незалежної модифікації окремих шарів системи без значного впливу на інші.

Таким чином, другий розділ сформував міцний фундамент для подальшої практичної розробки, детально описавши, яким саме має бути застосунок з точки зору його функціональності та якості, та запропонувавши структурований і обґрунтований архітектурний підхід для його побудови.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ДОДАТКУ

3.1 Вибір технологій.

Для реалізації функціоналу системи моніторингу GitLab-репозиторіїв відповідно до вимог та багаторівневої архітектури, яку визначили в 2 розділі, було обрано набір технологій, які забезпечують гнучку побудову окремих рівнів, ефективну взаємодію з API, зручну обробку даних та створення адаптивного користувацького інтерфейсу. У цьому підрозділі детально обґрунтовано вибір кожної технології з точки зору її ролі у системі, конкретних завдань та вимог до якості програмного продукту.

Мова програмування — Java

Основною мовою реалізації проекту обрано Java, оскільки вона забезпечує платформену незалежність (нефункціональна вимога: кросплатформеність), що дозволяє запускати програму на будь-якій операційній системі з встановленою JVM [27]. Java органічно поєднується з багаторівневою архітектурою, оскільки її об’єктно-орієнтована модель дає змогу чітко описати сутності предметної області (Project, User, Commit, Branch, MergeRequest тощо) у вигляді класів.

Статична типізація і підтримка винятків сприяють надійності програми (нефункціональна вимога 2), а велика кількість бібліотек (наприклад, для HTTP-запитів, логування, JSON-серіалізації) дозволяє реалізувати всі функціональні сценарії без перевантаження коду [28].

Окрім того, екосистема Java підтримує широке використання фреймворків (Spring, Hibernate), інструментів тестування (JUnit) та інтеграції з CI/CD. Це створює стабільну основу для розширення і підтримки системи в майбутньому.

Графічний інтерфейс — JavaFX

Для реалізації рівня представлення обрано JavaFX - сучасний інструмент для створення графічних інтерфейсів у Java-додатках. Його використання дозволяє задовольнити вимоги до зручності (нефункціональна вимога «Usability»): користувач отримує зрозумілий, інтуїтивний інтерфейс з

таблицями, панелями, індикаторами й динамічними елементами. Компонентна модель Scene Graph підтримує побудову складних макетів, що особливо важливо для візуалізації репозиторіїв, гілок, комітів (функціональні вимоги 2, 3, 5) [29-31].

FXML дозволяє розділити логіку інтерфейсу і структуру макету, що відповідає принципам чіткої розділеності шарів. Додатково JavaFX підтримує CSS-стилі, які забезпечують адаптивну темізацію інтерфейсу та локалізацію (функціональна вимога 6).

Обробка JSON — Gson

Для перетворення відповідей GitLab API у внутрішні об’єкти системи використовується бібліотека Gson. Її інтеграція дозволяє у кілька рядків коду перетворювати JSON-структури на Java-об'єкти, що оптимізує логіку логічного шару програми і забезпечує швидку реалізацію функцій збору даних та їх представлення.

Бібліотека має низький поріг входу, не вимагає публічних геттерів чи спеціальних конструкторів, що пришвидшує реалізацію моделей (наприклад, класів Project, Commit, User). Вона також демонструє хорошу продуктивність при роботі з помірними обсягами даних [32] - що цілком відповідає характеристикам GitLab API у контексті періодичного моніторингу.

Робота з базою даних — Spring Framework та Hibernate ORM

Рівень доступу до даних реалізується за допомогою зв’язки Spring Framework та Hibernate. Локальна база даних використовується для збереження історичних даних, кешування, інформації про зв’язки користувача з проектами, що дозволяє мінімізувати кількість зовнішніх запитів до GitLab та підвищити продуктивність застосунку (нефункціональна вимога 1).

Hibernate забезпечує автоматичне перетворення об’єктів моделі (Entity-класи) на таблиці БД і навпаки. Це дозволяє реалізувати CRUD-операції без написання SQL-коду. Також підтримується ледаче завантаження (lazy loading), що зменшує обсяг одночасно отримуваних даних [33].

Spring, у свою чергу, забезпечує інверсію керування (IoC) і зручну конфігурацію компонентів, включно з транзакційністю [38]. Зокрема, шар сервісів (UserService, ProjectService тощо) може використовувати анотації @Transactional для забезпечення узгодженості змін. Це відповідає вимогам до надійності (нефункціональна вимога 2) та масштабованості системи (нефункціональна вимога 5)[34-35].

У випадку розширення програми або підключення додаткових джерел даних, Hibernate дозволить швидко адаптувати логіку доступу до нових структур без суттєвих змін у логіці.

3.2 Реалізація графічного інтерфейсу.

Графічний інтерфейс десктопної частини застосунку реалізовано за допомогою технології JavaFX. Основним вікном додатку є об’єкт класу Stage (головне вікно програми), яке є верхньорівневим контейнером JavaFX. У методі start(Stage primaryStage) створюється первинний віконний об’єкт і встановлюється для нього сцена (Scene), що містить усі графічні елементи [31]. Для підтримки мультимовності всі підписи й написи отримуються з об’єкта ResourceBundle з урахуванням обраної локалі (української чи англійської). Головна структура інтерфейсу вибудовується за допомогою менеджерів компоновки (layout): зокрема BorderPane, VBox та HBox для відповідного розміщення компонентів. Нижче докладно описано всі основні елементи GUI, їх розташування та функціональність.

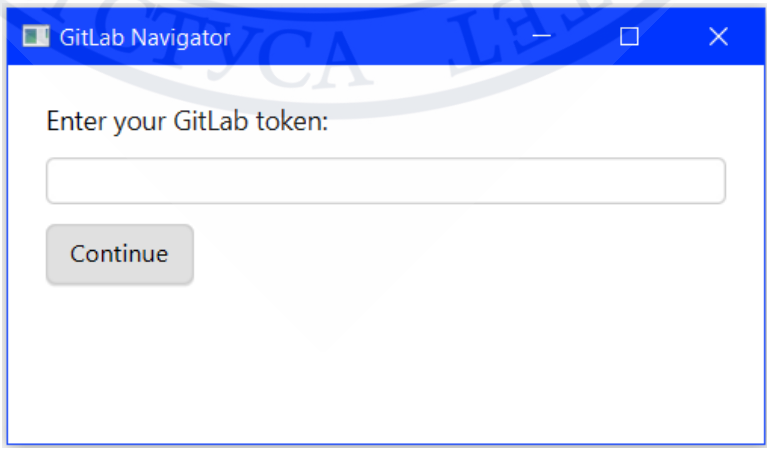

Рисунок 3.1 – Вікно введення токена

Після запуску програми користувачу спочатку відображається просте модальне вікно введення GitLab-токена. Це вікно побудоване за допомогою Stage і Scene, де головним контейнером є вертикальний VBox. У ньому послідовно розташовані компоненти: лейбл з інструкцією, поле для введення пароля (PasswordField) і кнопка підтвердження. Просторові інтервали між ними задаються через конструктор VBox(10), що визначає розмір проміжку. При натисканні на кнопку «Продовжити» спрацьовує обробник setOnAction, який перевіряє наявність введеного тексту. Якщо поле пусте, виводиться повідомлення про помилку. Інакше виконується автентифікація через виклик сервісу userService.authenticateAndSync(token). У разі успіху відкривається головне вікно застосунку; у разі помилки відображається текст повідомлення про помилку. Таким чином, це модальне вікно виконує роль початкового діалогу входу в систему. Усі візуальні компоненти тут мають базові CSS-класи для стилізації (наприклад, клас «root» для кореневого VBox), що забезпечує єдиний вигляд у темній чи світлій темі.

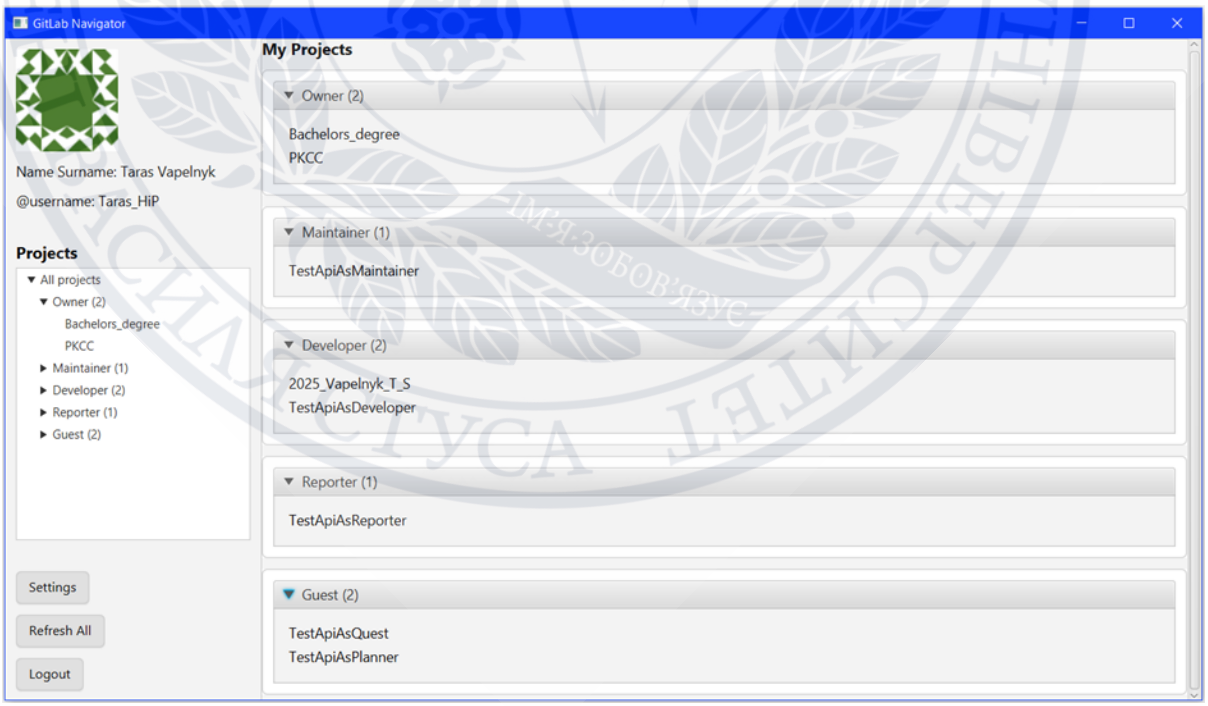

Рисунок 3.2 – Головне вікно програми

Після успішного входу формується головне вікно, спроектоване на основі компоновки BorderPane. У ліву частину розміщується вертикальна панель з фіксованою шириною близько 250 пікселів. Ця ліва панель містить декілька груп компонентів. Насамперед — блок інформації про користувача (userInfoBox), що містить віджет ImageView для аватарки і два Label з ім'ям та юзернеймом. Далі йде дерево проектів: підпис «Мої проекти» (Label зі стилем заголовка) та TreeView. Головний елемент дерева (root) названо «Всі проєкти» і за замовчуванням розгорнутий. Дочірні елементи дерева заповнюються динамічно сервісом userProjectService, що отримує проекти, згруповані за ролями доступу (наприклад, «Maintainer», «Developer» тощо). Кожна роль представлена вузлом-розділом, а під ним — вузли з назвами проектів. Подвійний клік по вузлу проекту викликає обробник, який переходить на детальну панель цього проекту (метод showProjectDetailsPanel). Інші елементи лівої панелі — це панель кнопок: кнопки «Налаштування» (gear), «Оновити» (refresh) та «Вийти» (logout). Кнопка налаштувань при натисканні відкриває спливаюче контекстне меню, яке містить два підменю: вибір теми інтерфейсу (світла/темна) і вибір мови (англійська/українська). При зміні теми викликається метод, що перезавантажує CSS-стилі сцени (підключаючи файл для темної або світлої теми), а при зміні мови встановлюється нова локаль і форма перезапускається з інтерфейсом потрібною мовою. Кнопка «Оновити» запускає перезавантаження списку проектів або відновлює головну панель, а «Вийти» повертає користувача до вікна введення токена. Завдяки такому подієвому підходу забезпечується зручна навігація: кнопки реагують на кліки через setOnAction і відповідні методи.

Центральна частина основного вікна спочатку містить ScrollPane з вертикальним вмістом — списком проектів під заголовком. Під час запуску головного вікна асинхронно (за допомогою Task) отримуються проекти і створюється набір елементів. Якщо проектів нема, відображається повідомлення про їх відсутність. Інакше для кожної групи (ролі) створюється розширювана панель TitledPane з назвою ролі і кількістю проектів, всередині якої розміщується перелік Label з назвами проектів. Цей список проектів можна

розгортати/згортати – реалізовано стандартним контролом JavaFX «загорнуті панелі». Кожен елемент списку отримує обробник кліку мишки: по натисканню на назву проекту зліва відбувається виклик showProjectDetailsPanel для відкриття вікна деталей (як і у випадку з деревом). Весь цей механізм дає змогу користувачеві переглядати свої проекти: зліва – дерево з ієрархією, по центру – розгорнутий список.

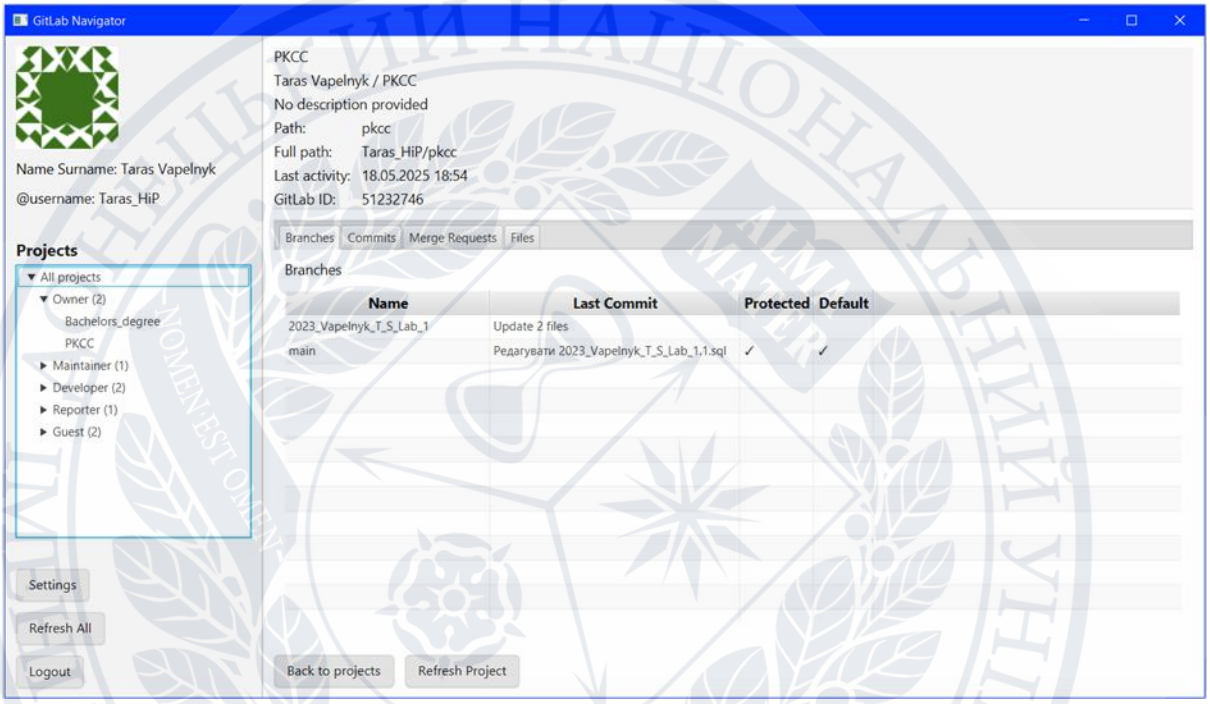

Рисунок 3.3 – Вікно деталей проекту

Після вибору конкретного проекту у головне вікно завантажується нова панель – панель деталей проекту. У цей момент центр замінюється на ScrollPane, що містить вертикальний VBox (projectDetailsPane). Інформації про проект формується у блоці projectInfoBox. Вона містить великі підписи: Label з назвою проекту, Label з namespace-повною назвою, блок з описом проекту. Далі формується мета-інформація про проект в GridPane: там розташовані пари «Мітка – Значення»: шляхи (path, path_with_namespace), дата останньої активності (форматована через DateTimeFormatter відповідно до локалі), GitLab ID проекту тощо. Використання GridPane забезпечує табличне вирівнювання метаданих у двох колонках.

Нижче розміщено TabPane з ненумерованими вкладками. У таблиці вкладок присутні розділи «Гілки», «Коміти», «Запити на злиття» та «Файли». Коли завершуються фонові задачі синхронізації, для кожної вкладки викликаються методи створення контенту (createFilesTabContent, createBranchesTabContent).

Вкладка «Гілки»: використовується TableView для відображення списку гілок репозиторію. TableView – це компонент для побудови табличних представлень даних . Він містить чотири колонки: «Ім’я гілки» (назва), «Останній коміт» (текст назви останнього коміту), «Захищена» (галочка, якщо гілка захищена) і «За замовчуванням» (галочка для основної гілки). Встановлюються відповідні CellValueFactory, які витягують дані з об’єкта Branch. Користувач може сортувати гілки по будь-якій колонці, наприклад, по імені або за останнім комітом. Таблиця займає всю доступну ширину, а дані додаються через branchesTable.getItems().addAll(branches). Якщо гілок нема, відображається відповідне повідомлення.

Вкладка «Коміти»: тут розташовується ListView зі спеціальними блоками. Кожен елемент (коміт) представлений вертикальним VBox: у верхньому рядку – назва коміту, у нижньому – блок HBox з метаданими (ім’я автора, дата коміту, хеш). Клас CommitListCell перевизначає метод updateItem, створюючи вміст для кожного коміту. Таким чином користувач бачить перелік комітів у хронологічному порядку. Для великих списків ListView забезпечує скролінг і підвантаження елементів.

Вкладка «Запити на злиття»: аналогічно комітам, для запитів на злиття (MergeRequest) використовується ListView. Кожен елемент стилізується у VBox: у заголовку відображається номер MR і його назва, нижче – стан (наприклад opened, merged), інформація про вихідну і цільову гілки, а також ім’я автора (котрий виконав merge). Це надає компактне подання для кожного merge request у списку.

Вкладка «Файли»: тут відображається ієрархічне дерево файлів проекту. Зібрані записи ProjectFile організовуються в TreeItem -ієрархію, де кореневий

вузол названий як ім’я проекту. Директорії та файли додаються у відповідні гілки дерева. Потім створюється TreeView з цим коренем. Для візуалізації статусів файлів (додано, змінено, видалено тощо) використовується власний клас FileTreeCell (розширює TreeCell). У цьому класі метод updateItem задає стиль тексту (колір, перечеркнення) залежно від статусу файлу: наприклад, зеленим кольором виділяються нові файли, помаранчевим – змінені, червоним із зачеркненням – видалені . Таким чином дерево файлів дає зручне уявлення про структуру репозиторію та колірний код змін. TreeView забезпечує можливість розгортати/згортати папки і прокручувати великий список при необхідності.

У самому низу вікна деталей проекту розташовані кнопки «Назад до проектів» та «Оновити проект». Кнопка «Назад» повертає користувача до списку проектів (центр BorderPane замінюється на початковий ScrollPane зі списком продуктів), а «Оновити проект» повторно викликає showProjectDetailsPanel для перезавантаження інформації (нової синхронізації).

3.3 Реалізація логіки застосунку

У розробленому Java-застосунку основна бізнес-логіка організована за допомогою Spring Framework, що забезпечує ін’єкцію залежностей та конфігурацію компонентів у контейнері IoC. Класи сервісів, репозиторії та утиліти відмічені анотаціями @Component, @Service, @Repository тощо, що дозволяє Spring автоматично виявляти їх під час сканування пакунків і створювати необхідні об’єкти. Залежності між компонентами визначаються через конструктори чи поля з анотацією @Autowired або за допомогою конфігураційних методів у класах, позначених @Configuration. При ін’єкції Spring керує життєвим циклом об’єктів і автоматично задає зв’язані залежності, тож самим класам не потрібно самостійно створювати чи шукати потрібні сервіси – це реалізує шаблон Inversion of Control (IoC).

Для взаємодії з GitLab API (версії 4) було створено окремий клас GitRequests, який позначений анотацією @Component у Spring, що дозволяє використовувати його як компонент фреймворку. Цей клас відповідає за

побудову HTTP-запитів до GitLab та обробку отриманих відповідей. Зокрема, метод getAllPages(String link, String token) реалізує механізм отримання пагінованих результатів: здійснюється послідовна відправка запитів з параметрами page та per_page доти, поки не буде вичерпано всі сторінки. Для встановлення HTTP-з'єднання використовується клас HttpURLConnection, до кожного запиту додається заголовок з токеном для автентифікації. У разі успішної відповіді (HTTP статус 200 OK), JSON-вміст зчитується з потоку та обробляється за допомогою бібліотеки Gson. У випадку помилки виводиться відповідне повідомлення. Таким чином, GitRequests інкапсулює низькорівневу логіку роботи з HTTP-запитами, спрощуючи взаємодію з GitLab API та автоматизуючи перетворення отриманих даних у об'єкти Java.

Додаткові методи сервісів верхнього рівня (наприклад, ProjectService , IssueService тощо) використовують GitRequests для запитів за конкретними ресурсами. Наприклад, метод syncUserProjects(User user, String token) у ProjectService формує URL типу / projects?membership=true для отримання всіх проектів, де користувач є учасником. Інші сервіси складають адреси виду /projects/{projectId}/repository/branches, /projects/ {projectId}/issues, /projects/{projectId}/merge_requests, /projects/{projectId}/ repository/files тощо, додаючи ідентифікатор проекту та необхідні параметри.

Відповіді GitLab API надходять у форматі JSON. Для роботи з цими даними застосовується бібліотека Gson – легка Java-бібліотека для серіалізації та десеріалізації JSON. Стандартний підхід: отриману JSON-стрічку через GitRequests парсити за допомогою JsonParser.parseString() чи JsonParser.parseReader(), що повертає об'єкт JsonElement (який може бути масивом чи об'єктом). Далі відповідні класи-сервіси перевіряють тип JsonElement і отримують JsonArray чи JsonObject. Наприклад ProjectService отримує JsonArray проектів, після чого для кожного JsonObject викликає внутрішній метод checkProject(JsonObject), де читаються властивості об'єкта (id, name, description та інші) і заповнюється екземпляр сутності Project. Аналогічно, CommitService перебирає масив комітів, BranchService перебирає гілки, це

відноситься до всіх сервісів. Перетворення JSON-вузлів у Java-об’єкти може відбуватися вручну (через геттери JsonObject.get("field").getAs()) або, за необхідності, використанням методу Gson.fromJson() з відповідним класом – обидва варіанти підтримуються бібліотекою . Результати десеріалізації потім використовуються для заповнення локальної бази даних і відображення у GUI. Таким чином, Gson слугує мостом між JSON-представленням даних GitLab та внутрішніми об’єктами проекту.

Для збереження даних використовується технологія об'єктно-реляційного відображення Hibernate у зв’язці з базою даних SQLite. Класам-сутностями надаються анотації JPA: кожен клас, який треба зберігати, позначений @Entity. Наприклад:

```
@Entity
@Table(name = "projects")
public class Project {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id_project")
    private int id;

    @Column(name = "gitlab_project_id", unique = true, nullable = false)
    private int gitlabProjectId;

    @Column(name = "name", nullable = false)
    private String name;

    @Column(name = "name_with_namespace", columnDefinition = "TEXT", nullable =
false)
    private String nameWithNamespace;

    @Column(name = "path", nullable = false)
    private String path;

    @Column(name = "path_with_namespace", columnDefinition = "TEXT", nullable =
false)
    private String pathWithNamespace;
```

```
    @Column(name = "gitlab_last_activity_at", nullable = false)
    private OffsetDateTime gitlabLastActivityAt;

    @Column(name = "description", columnDefinition = "TEXT")
    private String description;

    @OneToMany(mappedBy = "project")
    private List<UserProject> userProjects;

    @OneToMany(mappedBy = "project")
    private List<Commit> commits;

    @OneToMany(mappedBy = "project")
    private List<Branch> branches;

    @OneToMany(mappedBy = "project")
    private List<MergeRequest> mergeRequests;

    /*гетери та сетери*/
}
```

Клас Project анотований @Entity, отже Hibernate створить відповідну таблицю (за замовчуванням іменована як назва класу, яле в Java використовують camelCase тому я назву задав через @Table в snake_case). Поля класу, які мають зберігатися, позначаються @Id (первинний ключ) та @Column. Відношення між сутностями визначаються через анотації зв’язків: якщо Project має багато Commit, то у класі Project буде поле List commits з анотацією @OneToMany(mappedBy="project"), а в Commit = поле Project project з @ManyToOne. За такою схемою Hibernate створює у таблиці commits стовпець project_id, який є зовнішнім ключем до таблиці projects(id). Аналогічно, інші зв’язки між сутностями мають бути описані відповідними анотаціями. При цьому анотацію @OneToMany можна опустити, якщо з боку поточної сутності не передбачається необхідність отримувати доступ до пов’язаних об’єктів.

Відповідно, для кожного класу-сутності визначено Spring Data JPA репозиторій (інтерфейс, що наслідує JpaRepository). Приклад: interface ProjectRepository extends JpaRepository. Spring Data JPA на льоту генерує

реалізацію цього інтерфейсу, забезпечуючи набір стандартних CRUD-методів (save(), findById(), findAll(), delete()) без необхідності писати SQL чи власний DAO-код. Завдяки цьому, сервісні класи можуть просто викликати projectRepository.save(project) для збереження об'єкта або findById() для отримання. За потреби можна визначити додаткові методи наприклад, findByGitlabProjectId(int gitlabProjectId) і Spring автоматично згенерує потрібний запит [33].

Важливо, що доступ до БД відбувається всередині транзакції. Методи сервісів, що викликають save() чи delete(), помічені @Transactional, тому Hibernate відкриває сесію та транзакцію автоматично. Наприклад, після отримання JSON-даних з GitLab, сервіс збирає сутність Project і викликає projectRepository.save(project) – Hibernate згенерує INSERT або UPDATE у SQLite. Аналогічно, списки комітів, гілок та інших елементів записуються через відповідні репозиторії (commitRepository, branchRepository тощо).

3.4 Результати роботи застосунку

Розроблений застосунок для моніторингу GitLab-репозиторіїв є завершеним програмним продуктом, призначеним для настільного використання. Він надає користувачам інструментарій для ефективного відстеження активності у їхніх проектах на платформі GitLab. Цей підрозділ демонструє ключові аспекти функціонування системи, які стали можливими завдяки інтеграції обраних технологій та реалізованих програмних компонентів, та як вони забезпечують досягнення поставлених функціональних цілей і комфортний користувацький досвід.

Демонстрація ключового функціоналу та взаємодії з користувачем: Робота із застосунком розпочинається з екрану автентифікації, де користувачу пропонується ввести персональний GitLab-токен. Система перевіряє його валідність: у разі успіху відкривається доступ до основного інтерфейсу, а при помилці (невірний токен, відсутність мережі) користувач отримує відповідне сповіщення. Це забезпечує безпечний вхід та первинну взаємодію.

Після успішної автентифікації користувач переходить у головне вікно програми. Центральне місце тут займає список проектів, до яких користувач має доступ. Дані про проекти завантажуються асинхронно. Для зручності проекти представлені у двох форматах: ієрархічним деревом на лівій панелі та згрупованим списком у центральній частині. Ліва панель також відображає інформацію про поточного користувача (аватар, ім'я) та надає доступ до кнопок управління: «Налаштування» (для зміни теми інтерфейсу – світла/темна, та мови – українська/англійська), «Оновити дані» та «Вийти».

При виборі конкретного проекту користувач переходить до панелі деталей проекту. Ця секція програми надає вичерпну інформацію про обраний репозиторій. Відображаються загальні відомості про проект (назва, опис), його ключові метадані (шляхи, дата активності, ID). Основний контент організовано у вигляді вкладок:

«Гілки»: список гілок проекту представлений у табличному вигляді, з можливістю сортування.

«Коміти»: історія комітів відображається у вигляді списку, де кожен елемент містить інформацію про автора, дату та повідомлення коміту.

«Запити на злиття»: перелік запитів на злиття, їх статусів та авторів.

«Файли»: структура файлів та папок проекту представлена у вигляді ієрархічного дерева, де візуально (кольором) можуть позначатися статуси файлів.

Усі дані для цих вкладок завантажуються асинхронно, що забезпечує відгукливість інтерфейсу.

Інтеграція логіки, взаємодії з API та базою даних: Ефективне функціонування застосунку забезпечується злагодженою роботою його внутрішніх компонентів:

Отримання даних з GitLab API: Спеціалізований програмний модуль відповідає за формування HTTP-запитів до GitLab API, обробку пагінації відповідей, додавання токену для аутентифікації та отримання даних у форматі JSON.

Обробка та перетворення даних: Бібліотека Gson використовується сервісними класами для перетворення (десеріалізації) отриманих JSON-даних у відповідні Java-об'єкти, що представляють сутності системи (проекти, коміти, гілки тощо).

Збереження та кешування даних: Технологія Hibernate ORM у поєднанні зі Spring Data JPA використовується для взаємодії з локальною базою даних SQLite. Отримана з GitLab інформація зберігається в цій базі, що реалізує механізм кешування. Це прискорює завантаження даних при повторних запусках програми або перегляді вже синхронізованих проектів, а також зменшує навантаження на GitLab API.

Сервісний шар та управління компонентами: Spring Framework керує життєвим циклом сервісів та репозиторіїв, забезпечуючи впровадження залежностей. Сервіси реалізують основну бізнес-логіку, таку як синхронізація даних користувача та проектів, автентифікація, та координують взаємодію між інтерфейсом, модулем запитів до API та локальною базою даних. Транзакційність операцій з базою даних гарантує цілісність збережених даних.

Досягнення системних якостей: Реалізований застосунок демонструє відповідність ключовим нефункціональним вимогам:

Продуктивність: Досягається за рахунок асинхронного виконання операцій вводу-виводу (робота з API, запити до БД) у JavaFX, а також завдяки ефективному використанню локального кешу даних, що мінімізує затримки.

Надійність: Забезпечується механізмами обробки винятків на рівні сервісів та модуля взаємодії з API (наприклад, при помилках з'єднання або невалідних відповідях API). Використання транзакцій Spring для операцій з базою даних підтримує цілісність даних. Користувач отримує інформативні повідомлення у разі виникнення проблем.

Зручність використання (Usability): Інтерфейс, створений за допомогою JavaFX, є інтуїтивно зрозумілим. Підтримка локалізації та можливість зміни візуальної теми покращують користувацький досвід.

Масштабованість (в контексті обробки даних): Архітектурні рішення, такі як пагінація при отриманні великих обсягів даних з API та використання локальної бази для кешування, створюють передумови для ефективної роботи з потенційно значною кількістю проектів та пов'язаної з ними інформації.

У підсумку, розроблений програмний продукт успішно поєднує обрані технології та архітектурні підходи для створення функціонального та ефективного інструменту. Він надає користувачеві можливість зручно та надійно моніторити свої GitLab-репозиторії, реалізуючи всі заплановані можливості.

Висновок до третього розділу

У третьому розділі було детально висвітлено процес практичної розробки десктопного застосунку для моніторингу GitLab-репозиторіїв. Цей розділ охопив усі етапи створення програмного продукту, від обґрунтування вибору технологічного стеку до демонстрації функціональних можливостей готової системи.

Спочатку було представлено обґрунтування вибору ключових технологій. Java була обрана як основна мова програмування завдяки своїй платформонезалежності та багатій екосистемі. Для побудови графічного користувацького інтерфейсу було використано JavaFX, що дозволило створити сучасний та відгукливий дизайн. Обробка даних у форматі JSON, які надходять від GitLab API, здійснювалася за допомогою бібліотеки Gson. Управління компонентами, залежностями та реалізація сервісного шару бізнес-логіки покладалися на Spring Framework, тоді як взаємодія з локальною базою даних для кешування та зберігання даних була реалізована за допомогою Hibernate ORM.

Далі було детально описано реалізацію графічного інтерфейсу. Розглянуто структуру основних вікон застосунку: початкового вікна для введення GitLab-токену, головного вікна, що відображає список проектів користувача та надає інструменти для навігації й налаштувань, а також вікна з детальною інформацією про вибраний проект. Це вікно містить вкладки для перегляду гілок, історії

комітів, запитів на злиття та структури файлів репозиторію. Було приділено увагу використанню компонентів JavaFX, менеджерів компонування та механізмів обробки подій для забезпечення інтерактивності та зручності.

Наступним кроком було розкрито архітектуру та реалізацію бізнес-логіки застосунку. Описано, як Spring Framework використовується для ін'єкції залежностей та управління сервісними класами. Було представлено спеціалізований модуль для взаємодії з GitLab API, який відповідає за формування запитів, аутентифікацію та обробку відповідей. Процес перетворення JSON-даних в об'єкти Java за допомогою Gson також був розглянутий. Крім того, деталізовано роботу з локальною базою даних SQLite через JPA-сутності та репозиторії Spring Data JPA, включаючи забезпечення цілісності даних за допомогою транзакцій. Було показано, як Spring-контекст інтегрується з головним класом JavaFX-додатку для забезпечення взаємодії між шарами.

На завершення, було продемонстровано результати роботи створеного застосунку. Описано ключові сценарії взаємодії користувача з програмою, починаючи від процесу автентифікації до перегляду детальної інформації про проекти. Було підкреслено, що система успішно виконує завдання з отримання, обробки, кешування та наочного представлення даних. Застосунок відповідає заявленим функціональним та нефункціональним вимогам, зокрема щодо продуктивності, надійності та зручності для кінцевого користувача.

Таким чином, третій розділ підсумовує практичне втілення проектних рішень у повноцінний програмний продукт. Обраний набір технологій та архітектурних підходів довів свою ефективність, дозволивши створити надійний, продуктивний та зручний у використанні застосунок для моніторингу GitLab-репозиторіїв.

ВИСНОВОК

У процесі виконання роботи було розроблено десктопний додаток, призначений для моніторингу стану програмних репозиторіїв на платформі GitLab через її Web-API. Робота охоплювала етапи аналізу предметної області, проектування системи та її практичної реалізації.

У першому розділі було проведено детальний аналіз платформи GitLab як комплексного DevOps-інструменту, досліджено архітектуру та функціональні можливості її Web-API, включаючи механізми автентифікації та ключові ендпоінти для збору даних. Також було розглянуто концепцію моніторингу програмних репозиторіїв та здійснено огляд існуючих інструментів, виявлено їхні переваги, недоліки та обґрунтовано актуальність розробки власного спеціалізованого рішення.

Другий розділ був присвячений проектуванню програмного продукту. Було визначено ключові функціональні вимоги, такі як автентифікація користувача, завантаження списку репозиторіїв, відображення детальної інформації про проекти (коміти, гілки, запити на злиття, файли) та візуалізація даних. Також було сформульовано нефункціональні вимоги, що стосуються продуктивності, надійності, безпеки та зручності використання. На основі цих вимог було обрано та обґрунтовано багаторівневу архітектуру додатку, що включає рівень представлення (JavaFX), рівень бізнес-логіки (сервіси Spring) та рівень доступу до даних (взаємодія з GitLab API та локальною БД). Було розроблено модель даних системи.

Третій розділ описує практичну реалізацію додатку. Було обґрунтовано вибір технологічного стеку: мова програмування Java, JavaFX для графічного інтерфейсу, бібліотека Gson для обробки JSON-відповідей від API, Spring Framework для управління компонентами та бізнес-логікою, Hibernate ORM для взаємодії з локальною базою даних SQLite. Детально описано реалізацію графічного інтерфейсу користувача, включаючи вікна автентифікації, головне вікно зі списком проектів та панель детальної інформації з вкладками. Також

розкрито особливості реалізації бізнес-логіки, взаємодії з GitLab API та механізмів збереження і кешування даних. Продемонстровано результати роботи застосунку, які підтверджують його відповідність поставленим вимогам та ефективне виконання основних функцій.

Таким чином, виконане дослідження та розробка програмного продукту дозволили створити функціональний інструмент для моніторингу GitLab-репозиторіїв. Розроблений додаток надає користувачам зручний доступ до актуальної інформації про їхні проекти, сприяючи ефективнішому управлінню процесом розробки програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. History of GitLab. The GitLab Handbook. URL: https://handbook.gitlab.com/handbook/company/history/#2011-start-of-gitlab (дата звернення: 31.04.2025).
2. A special farewell from GitLab's Dmitriy Zaporozhets. GitLab Blog. URL: https://about.gitlab.com/blog/2021/11/10/a-special-farewell-from-gitlab-dmitriy-zaporozhets/ (дата звернення: 31.04.2025).
3. Evertse J. Mastering GitLab 12: Implement DevOps culture and repository management solutions. Birmingham: Packt Publishing, 2019. 350 p.
4. The DevOps Platform for agile business. GitLab Blog. URL: https://about.gitlab.com/blog/2021/11/03/the-devops-platform-for-agile-business/ (дата звернення: 31.04.2025).
5. GitLab architecture overview. GitLab Docs. URL: https://docs.gitlab.com/development/architecture/ (дата звернення: 31.04.2025).
6. Frontend Development Guidelines. GitLab Docs. URL: https://docs.gitlab.com/development/fe_guide/#:~:text=GitLab%20is%20built%20on%20top,page%2C%20read%20this%20explanation (дата звернення: 31.04.2025).
7. Integrate Kubernetes to your DevOps Lifecycle. GitLab. URL: https://about.gitlab.com/solutions/kubernetes/ (дата звернення: 31.04.2025).
8. CI/CD pipelines. GitLab Docs. URL: https://docs.gitlab.com/ci/pipelines/ (дата звернення: 31.04.2025).
9. Issues. GitLab Docs. URL: https://docs.gitlab.com/user/project/issues/ (дата звернення: 31.04.2025).
10. Merge requests. GitLab Docs. URL: https://docs.gitlab.com/user/project/merge_requests/ (дата звернення: 31.04.2025).
11. Analytics dashboards. GitLab Docs. URL: https://docs.gitlab.com/user/analytics/analytics_dashboards/ (дата звернення: 31.04.2025).
12. REST API. GitLab Docs. URL: https://docs.gitlab.com/api/rest/ (дата звернення: 31.04.2025).
13. REST API troubleshooting. GitLab Docs. URL: https://docs.gitlab.com/api/rest/troubleshooting/ (дата звернення: 31.04.2025).

14. Personal access tokens. GitLab Docs. URL: https://docs.gitlab.com/user/profile/personal_access_tokens/ (дата звернення: 31.04.2025).
15. REST API authentication. GitLab Docs. URL: https://docs.gitlab.com/api/rest/authentication/ (дата звернення: 31.04.2025).
16. GitLab CI/CD job token. GitLab Docs. URL: https://docs.gitlab.com/ci/jobs/ci_job_token/ (дата звернення: 31.04.2025).
17. Projects API. GitLab Docs. URL: https://docs.gitlab.com/api/projects/ (дата звернення: 31.04.2025).
18. Commits API. GitLab Docs. URL: https://docs.gitlab.com/api/commits/ (дата звернення: 31.04.2025).
19. Branches API. GitLab Docs. URL: https://docs.gitlab.com/api/branches/ (дата звернення: 31.04.2025).
20. Value stream analytics. GitLab Docs. URL: https://docs.gitlab.com/user/group/value_stream_analytics/ (дата звернення: 31.04.2025).
21. Analyze GitLab usage. GitLab Docs. URL: https://docs.gitlab.com/user/analytics/ (дата звернення: 31.04.2025).
22. Using the GitLab-Exporter chart. GitLab Docs. URL: https://docs.gitlab.com/charts/charts/gitlab/gitlab-exporter/ (дата звернення: 31.04.2025).
23. Monitoring GitLab with Prometheus. GitLab Docs. URL: https://docs.gitlab.com/administration/monitoring/prometheus/ (дата звернення: 31.04.2025).
24. GitLab integration. Grafana Cloud documentation. URL: https://grafana.com/docs/grafana-cloud/monitor-infrastructure/integrations/integration-reference/integration-gitlab/ (дата звернення: 31.04.2025).
25. GitLab integration. Datadog Docs. URL: https://docs.datadoghq.com/integrations/gitlab/?tab=host (дата звернення: 31.04.2025).
26. Gitential – Analytics for your codebase. Gitential. URL: https://gitential.com/ (дата звернення: 31.04.2025).
27. Pros and Cons of Java Development. Altexsoft Blog. URL: https://www.altexsoft.com/blog/pros-and-cons-of-java-programming/ (дата звернення: 31.04.2025).

28. Importance of Java in Modern Programming. DEV Community. URL: https://dev.to/devme/importance-of-java-in-modern-programming-ih7 (дата звернення: 31.04.2025).
29. Swing vs. JavaFX: Compare Java GUI frameworks. TheServerSide. URL: https://www.theserverside.com/tip/Swing-vs-JavaFX-Compare-Java-GUI-frameworks (дата звернення: 31.04.2025).
30. The JavaFX Advantage for Swing Developers (Release 8). Oracle Docs. URL: https://docs.oracle.com/javase/8/javafx/interoperability-tutorial/overview.htm (дата звернення: 31.04.2025).
31. Chin, S., Vos, J., Weaver, J. The Definitive Guide to Modern Java Clients with JavaFX 17: Cross-Platform Mobile and Cloud Development. – 2nd ed. – Berkeley: Apress, 2022. – 620 p.
32. The Ultimate JSON Library: JSON.simple vs GSON vs Jackson vs JSONP. Harness.io Blog. URL: https://www.harness.io/blog/ultimate-json-library-comparison (дата звернення: 31.04.2025).
33. Tudose, C. Java Persistence with Spring Data and Hibernate / Catalin Tudose. – Shelter Island: Manning Publications, 2020. – 560 p.
34. What is Spring Framework and Hibernate ORM?. GeeksforGeeks. URL: https://www.geeksforgeeks.org/what-is-spring-framework-and-hibernate-orm/ (дата звернення: 31.04.2025).
35. Introduction to ORM with Spring. Spring Framework Docs. URL: https://docs.spring.io/spring-framework/reference/data-access/orm/introduction.html (дата звернення: 31.04.2025).
36. Loeliger J., Ponuthorai P. Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development. – 3rd ed. – O'Reilly Media, 2022. – 546 p.
37. Bass L., Clements P., Kazman R. Software Architecture in Practice. – 4th ed. – Longman (Pearson Education), 2022. – 442 p.
38. Walls C. Spring in Action. – 6th ed. – Manning Publications, 2022. – 504 p.
39. Heckler M. Spring Boot: Up and Running: Building Cloud Native Java and Kotlin Applications. – O'Reilly Media, 2021. – 300 p.
40. Ullenboom C. Java: The Comprehensive Guide to Java Programming for Professionals. – Rheinwerk Computing, 2023. – 1126 p.

ДЕКЛАРАЦІЯ

про дотримання академічної доброчесності

Я, _____
_____
_____
_____
_____

Повністю вказується ПІБ та статус (освітня (освітньо-наукова) програма – для здобувачів вищої освіти, назва кваліфікаційної роботи)

що нижче підписалась/підписався, розуміючи та підтримуючи загальновизнані засади справедливості, доброчесності та законності,

ЗОБОВ'ЯЗУЮСЬ:

дотримуватися принципів та правил академічної доброчесності, що визначені законодавством України, локальними нормативними актами Донецького національного університету імені Василя Стуса, положеннями, правилами, умовами, визначеними іншими суб'єктами, та не допускати їх порушення.

ПІДТВЕРДЖУЮ:

що мені відомі положення статті 42 Закону України «Про освіту»;

що у даній роботі не представляла/представляв чиїсь роботи повністю або частково як свої власні. Там, де я скористалася/скористався працею інших, я зробила/зробив відповідні посилання на джерела інформації;

що дана робота не передавалась іншим особам і подається вперше, не порушує авторських та суміжних прав закріплених статтями 21-25 Закону України «Про авторське право та суміжні права», а дані та інформація не отримувались в недозволений спосіб.

УСВІДОМЛЮЮ:

що ця робота може бути перевірена університетом на плагіат або інші порушення академічної доброчесності, в тому числі з використанням спеціалізованих сервісів;

що у разі порушення академічної доброчесності, до мене можуть бути застосовані процедури, передбачені законодавством України та Кодексом академічної доброчесності та корпоративної етики Донецького національного університету імені Василя Стуса, іншими локальними нормативними актами університету, та я можу бути притягнута/притягнутий до академічної відповідальності.

_____

_____

(дата) (підпис)