НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
Кафедра інформаційних систем та технологій

Тема: CI server

Курсова робота
З дисципліни «Технології розроблення програмного забезпечення»

Керівник
доц. Амонс О.А.
«Допущений до захисту»
_____
(Особистий підпис керівника)
« » _____2025р.
Захищений з оцінкою
_____
(оцінка)

Виконавець
ст. Губар Б. О.
залікова книжка № ___ – _____
гр. ІА-31
_____
(особистий підпис виконавця)
« »_____2025р.

Члени комісії:

_____ _____
(особистий підпис) (розшифровка підпису)

_____ _____
(особистий підпис) (розшифровка підпису)

Київ – 2025

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
(назва навчального закладу)
Кафедра ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ
Дисципліна «Технології розроблення програмного забезпечення»
Курс 3 Група ІА-31 Семестр 5

ЗАВДАННЯ

на курсову роботу студента

Губара Богдана Олександровича
(прізвище, ім'я, по батькові)

1.Тема роботи: CI server

2. Строк здачі студентом закінченої роботи

3. Вихідні дані до роботи:

Сервер безперервної інтеграції (CI Server) повинен нагадувати функціонал таких систем, як Jenkins або GitLab CI, з можливостями автоматичного моніторингу репозиторіїв коду (наприклад, Git), запуску налаштовуваних пайплайнів (компіляція, тестування, аналіз коду), збереження артефактів збірки, ведення детальної історії та логів.

4. Зміст розрахунково – пояснювальної записки (перелік питань, що підлягають розробці)

Проектування системи (огляд існуючих рішень, опис проекту, вимоги до застосунків системи (функціональні/нефункціональні), сценарії використання системи, концептуальна модель системи, вибір бази даних, мови програмування та середовища розробки, проєктування розгортання системи), реалізація компонентів системи (структура бази даних, вибір і обґрунтування патернів реалізації, інструкція користувача)

Додатки:

Додаток А - проектування паттернів

5.Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Рис. 1.1 — Діаграма варіантів використання; Рис. 1.2–1.4 — Діаграми послідовності; Рис. 1.5 — Діаграма класів сутностей; Рис. 1.6 — Діаграма класів репозиторіїв; Рис. 1.7— Діаграма зв'язків; Рис. 1.8 — Діаграма компонентів; Рис. 1.9 — Діаграма розгортання.

6. Дата видачі завдання 17.09.2025

КАЛЕНДАРНИЙ ПЛАН

№, п/п	Назва етапів виконання курсової роботи	Строк виконання етапів роботи	Підписи або примітки
1.	Підбір та вивчення літератури	30.09.2025	
2.	Проектування та написання розділу 1	31.10.2025	
3.	Розробка та написання розділу 2	20.11.2025	
4.	Подання курсової роботи на перевірку	25.11.2025	
5.	Захист курсової роботи	08.12.2025	
6.			
7.			
8.			
9.			
10.			
11.			
12.			
13.			
14.			
15.			
16.			
17.			
18.			

Студент _____ _____Богдан ГУБАР_____
(підпис) (Ім’я ПРІЗВИЩЕ)

Керівник_____ __Олександр АМОНС__
(підпис) (Ім’я ПРІЗВИЩЕ)

«____»_____ 20___ р.

ЗМІСТ

ВСТУП

Сучасні практики розробки програмного забезпечення стали основою для прискорення інноваційних циклів та підвищення загальної якості цифрових продуктів, що стали важливою частиною бізнесу та повсякденного життя. Однією з основних складових цих практик (часто об'єднаних під терміном DevOps) є сервери безперервної інтеграції (CI), які виконують функцію повної автоматизації процесів збірки, тестування та розгортання програмного коду. Безперервна інтеграція (Continuous Integration) — це інженерна практика, яка вимагає від розробників частої інтеграції змін коду в центральний репозиторій, після чого автоматично запускаються процеси збірки та тестування. Розробка CI-сервера є важливим завданням, яке вимагає глибоких знань в області архітектури програмних систем, автоматизації та принципів управління конфігураціями. У даній курсовій роботі розглядається розробка серверного програмного забезпечення, яке здатне коректно моніторити системи контролю версій (наприклад, Git), запускати визначені користувачем пайплайни (pipelines) та забезпечувати стабільне функціонування в умовах активної паралельної розробки.

Основною метою цієї роботи є створення CI-сервера, який здатний не тільки коректно обробляти тригери (наприклад, webhooks) і формувати чергу завдань, але й виконувати скрипти збірки та тестування в ізольованому середовищі, вести детальну історію та логи всіх запусків, а також забезпечувати обробку завдань у багатопотоковому режимі або за допомогою розподілених агентів. Важливим аспектом є ефективність роботи сервера, що забезпечується використанням відповідних архітектурних рішень та алгоритмів. Одним з найбільш важливих завдань є аналіз залежностей між етапами (stages) та правильне формування відповіді про статус виконання. Для цього необхідно не тільки забезпечити коректне виконання зовнішніх команд, але й реалізувати механізми обробки помилок виконання та формування фінальних звітів і статусів (успіх/провал).

Ще однією ключовою частиною є збір логів та артефактів збірки. Збереження детальної історії є важливою складовою частиною функціонування CI-сервера, оскільки це дозволяє не лише відслідковувати результат кожної інтеграції коду, але й швидко виявляти причини збоїв у тестах або процесі збірки, аналізувати продуктивність та стабільність кодової бази. Обробка завдань у багатопотоковому режимі або через систему черг дозволяє ефективно розподіляти обчислювальні ресурси сервера і забезпечувати високий рівень продуктивності при роботі з великою кількістю розробників та проектів одночасно.

Особливу увагу слід приділити розробці системи управління виконавцями (executors), яка повинна бути гнучкою і масштабованою. Використання пулу потоків або динамічних агентів дозволяє розподіляти навантаження між кількома

процесами, що підвищує загальну пропускну здатність сервера. У той же час, асинхронна модель обробки є найбільш ефективною для управління тривалими процесами (як-от повне регресійне тестування), оскільки дозволяє серверу залишатися чутливим до нових тригерів, не блокуючи основний процес.

У результаті розробки CI-сервера в рамках цієї курсової роботи, буде реалізовано ефективне серверне програмне забезпечення, яке здатне працювати в умовах високої частоти комітів, надавати розробникам миттєвий зворотний зв'язок про якість їхніх змін і збирати статистичні дані (логи та звіти), що дозволяють моніторити ефективність усього процесу розробки та приймати відповідні рішення щодо його вдосконалення.

1 ПРОЄКТУВАННЯ СИСТЕМИ

1.1. Огляд існуючих рішень

Сучасний ринок програмного забезпечення для автоматизації процесів розробки (CI/CD) представлений великою кількістю рішень, що задовольняють різноманітні потреби команд. Серед них можна виділити як потужні, універсальні standalone-сервери, так і інтегровані в платформи рішення, що надають CI як частину загальної екосистеми. Огляд цих рішень дозволяє краще зрозуміти сучасні тенденції у сфері DevOps та визначити переваги і недоліки різних підходів.

Jenkins — це один з найбільш популярних і широко використовуваних CIсерверів у світі. Він розроблений з відкритим вихідним кодом і має надзвичайно багатий набір можливостей завдяки величезній екосистемі плагінів (тисячі розширень). Jenkins підтримує інтеграцію практично з будь-якими інструментами та технологіями, забезпечує розподілені збірки (модель master-agent) та гнучке налаштування пайплайнів як через графічний інтерфейс, так і через код (використовуючи Jenkinsfile).
Однак, незважаючи на свою універсальність, Jenkins має певні недоліки. Його налаштування та підтримка можуть бути складними та ресурсозатратними. Велика кількість плагінів може призводити до проблем із сумісністю ("plugin hell"), а його архітектура, особливо в частині основного контролера, може ставати "вузьким місцем" у високонавантажених системах порівняно з новітніми хмарними рішеннями.

GitLab CI/CD — це ще одне надзвичайно популярне рішення, яке здобуло велику популярність завдяки своїй глибокій інтеграції безпосередньо в платформу управління репозиторіями GitLab. Конфігурація пайплайнів відбувається декларативно через один файл .gitlab-ci.yml у корені проекту. GitLab CI використовує модель "runner-ів" (виконавців), що дозволяє легко масштабувати ресурси для збірки та забезпечувати високу продуктивність. Завдяки цьому він став вибором для багатьох команд, які шукають єдине "all-in-one" рішення.
Однією з переваг GitLab CI є його здатність виступати як повний DevOpsінструментарій, що включає не лише CI/CD, але й registry для контейнерів, інструменти сканування безпеки (SAST/DAST) та моніторинг. Це дозволяє ефективно побудувати весь життєвий цикл розробки в єдиному середовищі. Однак, на відміну від Jenkins, GitLab CI тісно пов'язаний з екосистемою GitLab, що може бути обмеженням, якщо компанія використовує інші системи контролю версій (наприклад, GitHub або Bitbucket) як основні.

GitHub Actions — це відносно нове, але вже дуже потужне рішення, інтегроване безпосередньо в GitHub. Воно відоме своєю гнучкістю та подієвоорієнтованою моделлю: пайплайни (workflows) можуть бути активовані не лише комітами, але й

створенням issue, коментарями, релізами та іншими подіями. GitHub Actions використовує конфігурацію через YAML-файли і має великий Marketplace готових "дій" (actions), що дозволяє швидко будувати складні процеси.
Проте GitHub Actions, подібно до GitLab, є частиною своєї платформи. Хоча він може працювати зі сторонніми інструментами, його найбільша ефективність досягається при роботі в екосистемі GitHub. Керування власними (self-hosted) runnerами може бути менш гнучким, ніж зріла master/agent архітектура Jenkins. Кожне з існуючих рішень має свої переваги та обмеження, що робить їх підходящими для різних сценаріїв використання. Для максимальної гнучкості та інтеграції з різнорідними інструментами часто обирають Jenkins. Для команд, що прагнуть отримати єдину, тісно інтегровану платформу, ідеальними є GitLab CI або GitHub Actions. Тому вибір рішення для створення CI-системи значною мірою залежить від вимог до гнучкості, наявної інфраструктури (зокрема, хостингу репозиторіїв) та специфіки проєкту.

1.2. Загальний опис проєкту

Проєкт має на меті розробку CI-сервера (Continuous Integration), який здатний автоматизувати процес збірки програмного забезпечення. Система забезпечує виконання визначених пайплайнів (pipelines) — послідовностей кроків, таких як клонування репозиторію, компіляція, тестування та архівування результатів.

Проєкт реалізовано на основі сервіс-орієнтованої архітектури (SOA), що складається з двох незалежних компонентів:

1. Web-сервер — відповідає за керування проєктами, чергою завдань, збереження історії та взаємодію з користувачем.
2. Агент збірки (Build Agent) — автономний компонент, який виконує ресурсомісткі операції на окремому вузлі та взаємодіє з сервером через REST API.

Сервер дозволяє запускати завдання за запитом користувача, керувати станами збірки (Pending, Running, Success, Failed) за допомогою патерну State, а також вести детальну історію виконань. Для забезпечення гнучкості процесу збірки використано патерн Command, що дозволяє динамічно формувати набір кроків виконання.

Важливим елементом системи є реалізація моніторингу в реальному часі. Завдяки використанню технології SignalR та патерну Observer/Mediator, користувачі

можуть спостерігати за логами виконання (console output) безпосередньо у веб-інтерфейсі без необхідності перезавантаження сторінки.

Система логування та зберігання артефактів дозволяє відслідковувати вивід консолі для кожного кроку, час виконання та фінальний статус. Збережені артефакти (наприклад, скомпільовані ZIP-архіви) доступні для завантаження через веб-інтерфейс, що реалізовано з урахуванням доступу до статичних файлів. Генерація детальних звітів про результати збірки виконується із застосуванням патерну Visitor, що дозволяє відокремити логіку візуалізації від моделей даних.

Загалом, проєкт є розподіленою системою, яка демонструє сучасні підходи до розробки ПЗ (Client-Server, SOA), використання патернів проєктування та ефективну роботу з даними, забезпечуючи надійний інструмент для автоматизації циклу розробки.

1.3. Вимоги до застосунків системи

1.3.1. Функціональні вимоги до системи

№	Назва вимоги	Короткий опис
1	Керування проєктами	• Система повинна дозволяти користувачеві створювати нові проєкти, вказуючи назву та URL Git-репозиторію. • Система повинна відображати список усіх наявних проєктів у вигляді карток. • Система повинна дозволяти видаляти проєкти разом з усією історією збірок та збереженими артефактами (каскадне видалення).
2	Виконання збірки	• Користувач повинен мати можливість запустити процес збірки (Build) вручну натисканням кнопки. • Система повинна створювати чергу завдань (Build Queue) зі статусом Pending.

		• Агент збірки (Build Agent) повинен автоматично опитувати сервер на наявність нових завдань (Polling). • Агент повинен виконувати послідовність команд (Pipeline): 1. Клонування репозиторію (git clone). 2. Компіляція коду (dotnet build). 3. Запуск тестів (dotnet test). 4. Архівування результатів (створення .zip артефакту).
3	Моніторинг та Логування	• Система повинна відображати логи виконання в реальному часі (Real-time) без перезавантаження сторінки (використовуючи WebSockets/SignalR). • Система повинна зберігати історію логів у базі даних для подальшого перегляду. • Система повинна відображати поточний статус збірки (Pending, Running, Success, Failed) за допомогою кольорової індикації.
4	Звітність та Артефакти	• Після успішного завершення збірки система повинна зберігати згенерований артефакт (ZIP-архів) на сервері. • Користувач повинен мати можливість завантажити артефакт через веб-інтерфейс. • Система повинна генерувати детальний HTML-звіт про результати збірки.

Таблиця 1.1. Функціональні вимоги

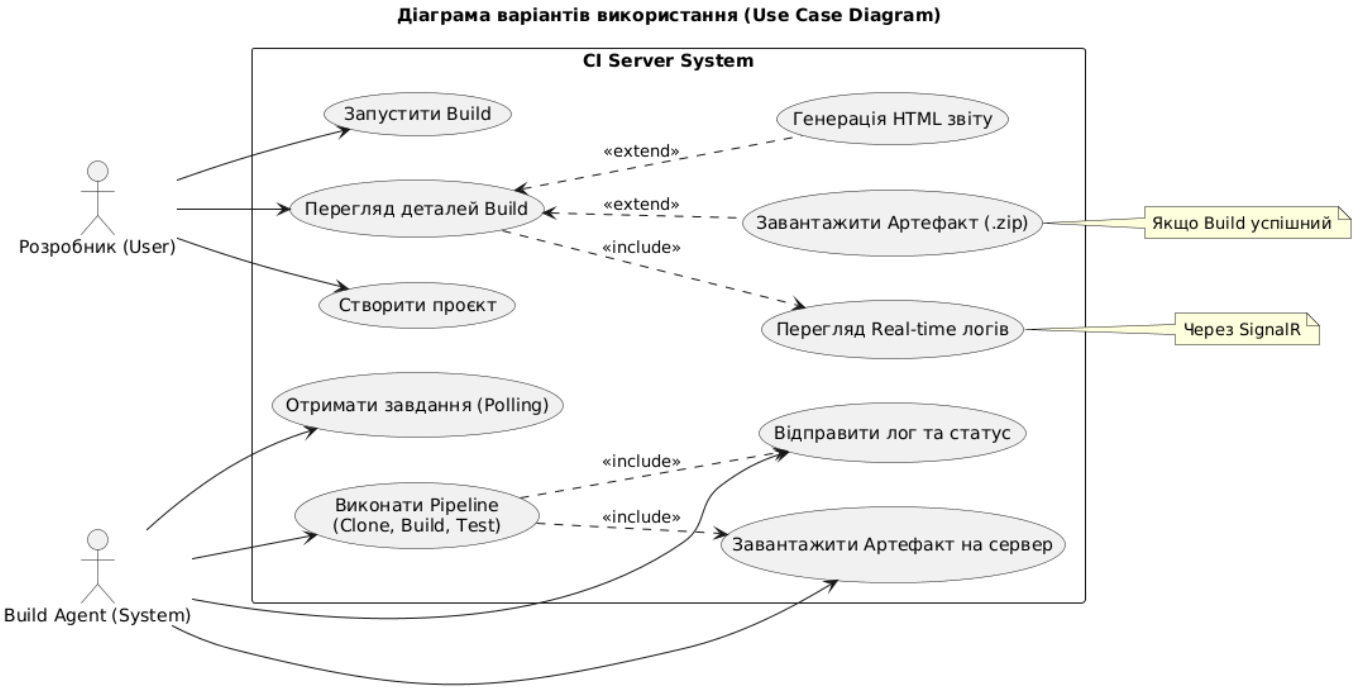

Рис. 1. 1 - Діаграма варіантів використання

1.3.2. Нефункціональні вимоги до системи

№	Назва	Опис
1	Архітектура та Масштабованість	• Система повинна бути побудована на основі сервіс-орієнтованої архітектури (SOA), розділяючи логіку веб-сервера та агента виконання. • Агент повинен бути автономним консольним додатком, здатним працювати на окремому фізичному вузлі.
2	Продуктивність	• Затримка при відображенні живих логів не повинна перевищувати 1 секунду. • Система повинна ефективно обробляти великі текстові потоки логів.
3	Надійність	• У разі помилки на одному з етапів пайплайну (наприклад, помилка компіляції), процес повинен коректно перериватися, а статус збірки змінюватися на Failed.

		• Дані про проєкти та історію збірок повинні зберігатися в реляційній базі даних (SQLite/MSSQL) із забезпеченням цілісності.
4	Зручність використання	• Веб-інтерфейс повинен бути інтуїтивно зрозумілим, мати сучасний дизайн (використання Bootstrap) та адаптуватися до розміру екрана. • Користувач повинен отримувати миттєвий візуальний зворотний зв'язок про дії системи (зміна статусів, поява логів).
5	Переносимість	• Серверна та клієнтська частини повинні бути кросплатформними (завдяки .NET Core) і працювати на Windows, Linux та macOS. • База даних повинна бути локальною та легко переносимою (SQLite).

Таблиця 1.2. Нефункціональні вимоги

1.4. Сценарії використання системи

Ручний запуск збірки	
Передумови	Агент збірки запущений та працює в режимі опитування (Polling).
Постумови	Створено запис збірки в БД зі статусом Pending. Агент отримав задачу на виконання, статус змінився на Running.
Взаємодіючі сторони	Розробник (User), CI Server (Web), Build Agent, Database.
Короткий опис	Розробник ініціює збірку через веб-інтерфейс, сервер створює задачу, яку згодом забирає вільний агент.
Основний потік подій	Розробник натискає кнопку «Запустити Build» на сторінці проєкту.
	CI Server створює новий запис Build у базі даних зі статусом Pending.
	CI Server перенаправляє користувача на сторінку деталей збірки.

	Build Agent (у фоновому режимі) надсилає запит до API сервера на отримання нових задач.
	CI Server знаходить задачу зі статусом Pending і віддає її агенту.
	CI Server оновлює статус задачі в базі даних на Running.
Винятки	Помилка бази даних (не вдалося створити запис). Сервер повертає помилку 500. Агент недоступний (задача залишається в статусі Pending, доки агент не з'явиться).

Таблиця 1.3. Сценарій використання «Ручний запуск збірки»

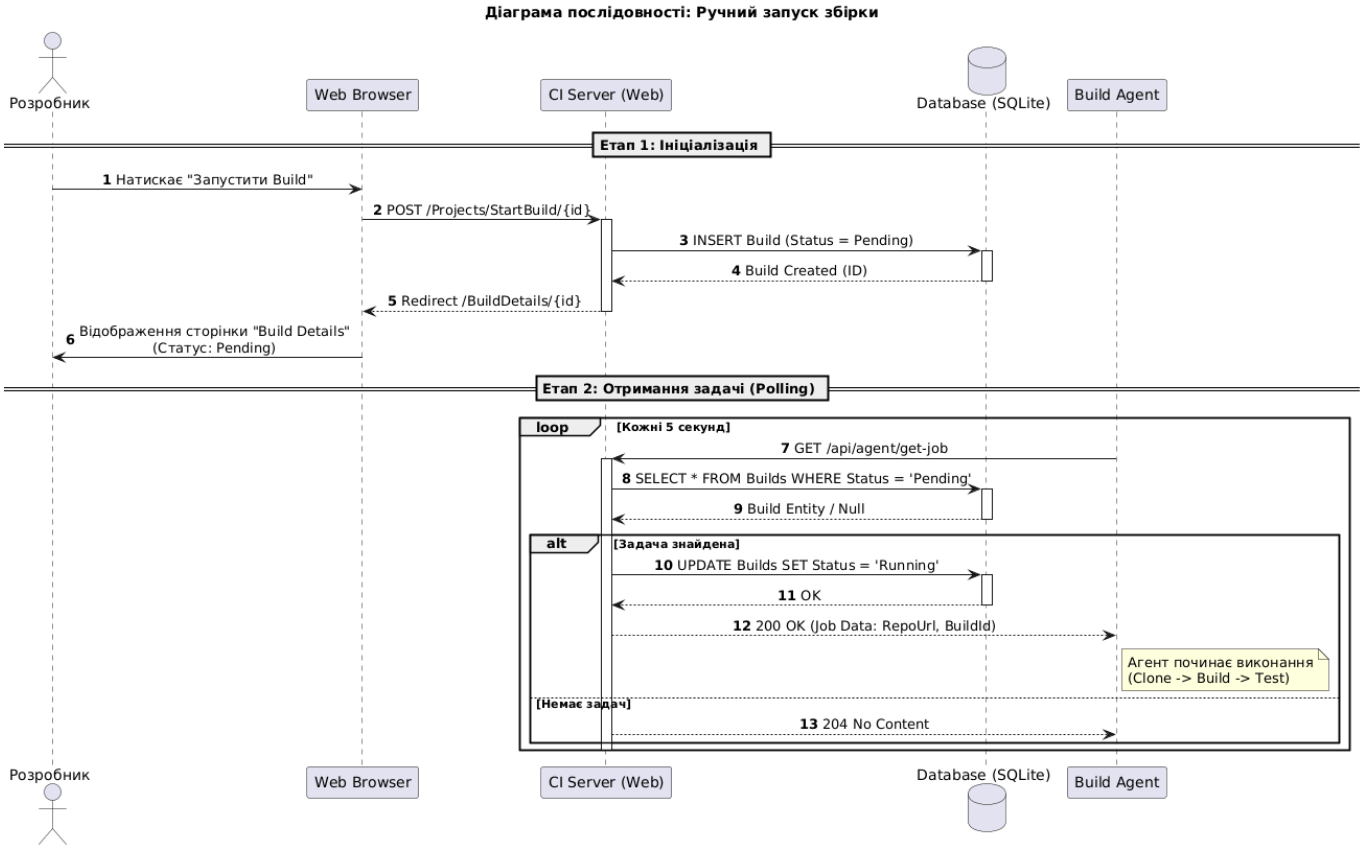

Рис. 1. 2 - Діаграма послідовності «Автоматичний запуск збірки»

Виконання пайплайну	
Передумови	У черзі є задача зі статусом Pending, є вільний Агент.
Постумови	Збірка завершена зі статусом Success або Failed, логи збережені.
Взаємодіючі сторони	CI Server, Build Agent.
Короткий опис	Агент отримує задачу та виконує послідовність команд.

Основний потік подій	1. Агент запитує задачу у Сервера.
	2. Сервер змінює статус збірки на Running.
	3. Агент клонує репозиторій.
	4. Агент виконує кроки (Build, Test).
	5. Агент відправляє логи на Сервер у реальному часі.
	6. Агент завершує роботу і оновлює фінальний статус у базі даних.
Винятки	Помилка компіляції. Агент перериває виконання, ставить статус Failed і записує це в БД.

Таблиця 1.4. Сценарій використання «Виконання пайплайну»

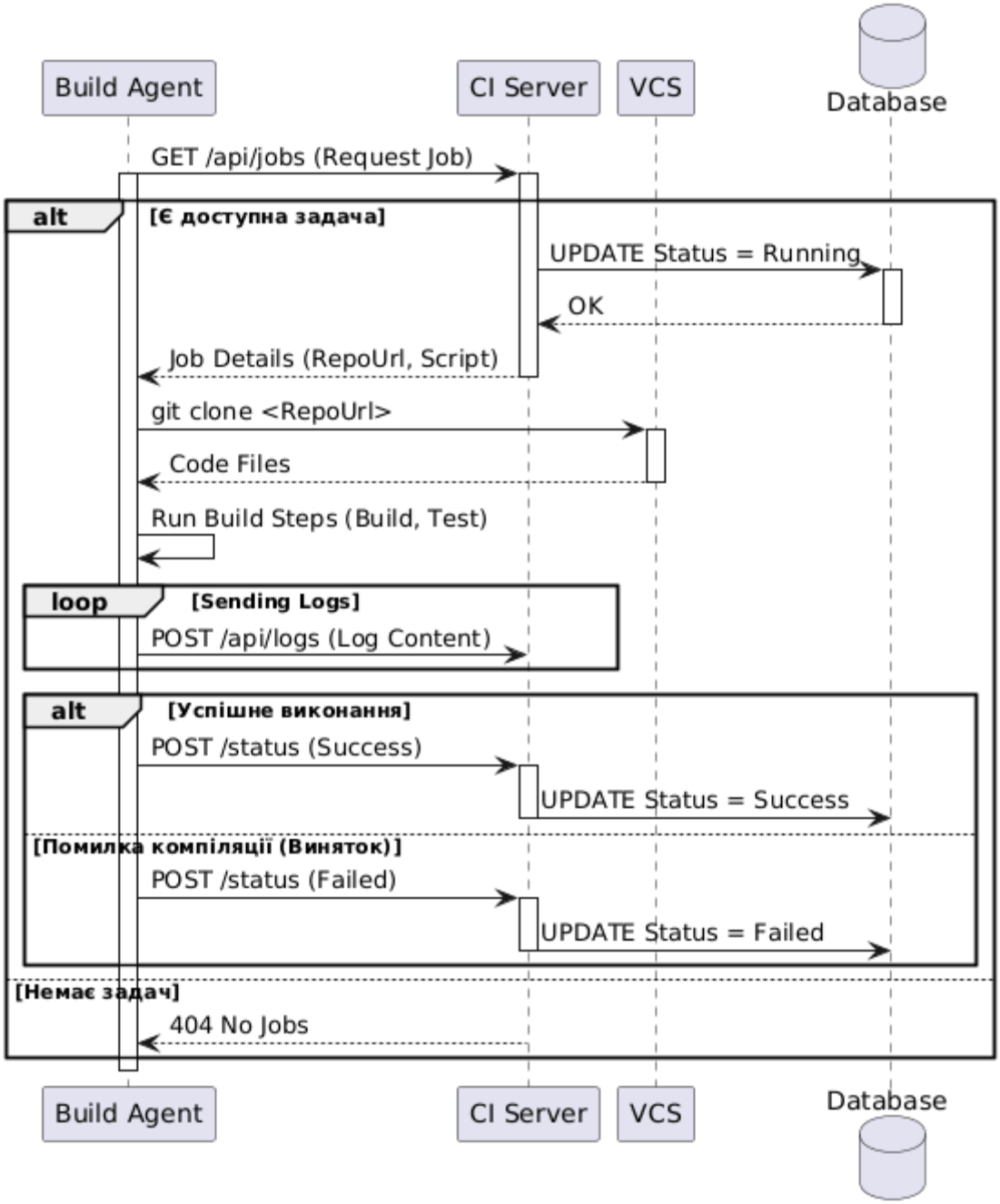

Рис. 1. 3 - Діаграма послідовності «Виконання пайплайну»

Перегляд логів	
Передумови	Збірка вже запущена або завершена, у базі є записи логів.
Постумови	Розробник бачить текстовий лог виконання.
Взаємодіючі сторони	Розробник, CI Server.
Короткий опис	Розробник відкриває сторінку збірки, щоб дізнатися причину помилки або хід виконання.

Основний потік подій	1. Розробник обирає конкретну збірку зі списку історії.
	2. Розробник натискає "View Logs".
	3. Система робить запит до БД в таблицю BuildLogs.
	4. Система відображає список рядків логу, відсортованих за часом.
Винятки	Збірка не знайдена. Система видає помилку 404.

Таблиця 1.5. Сценарій використання «Перегляд логів»

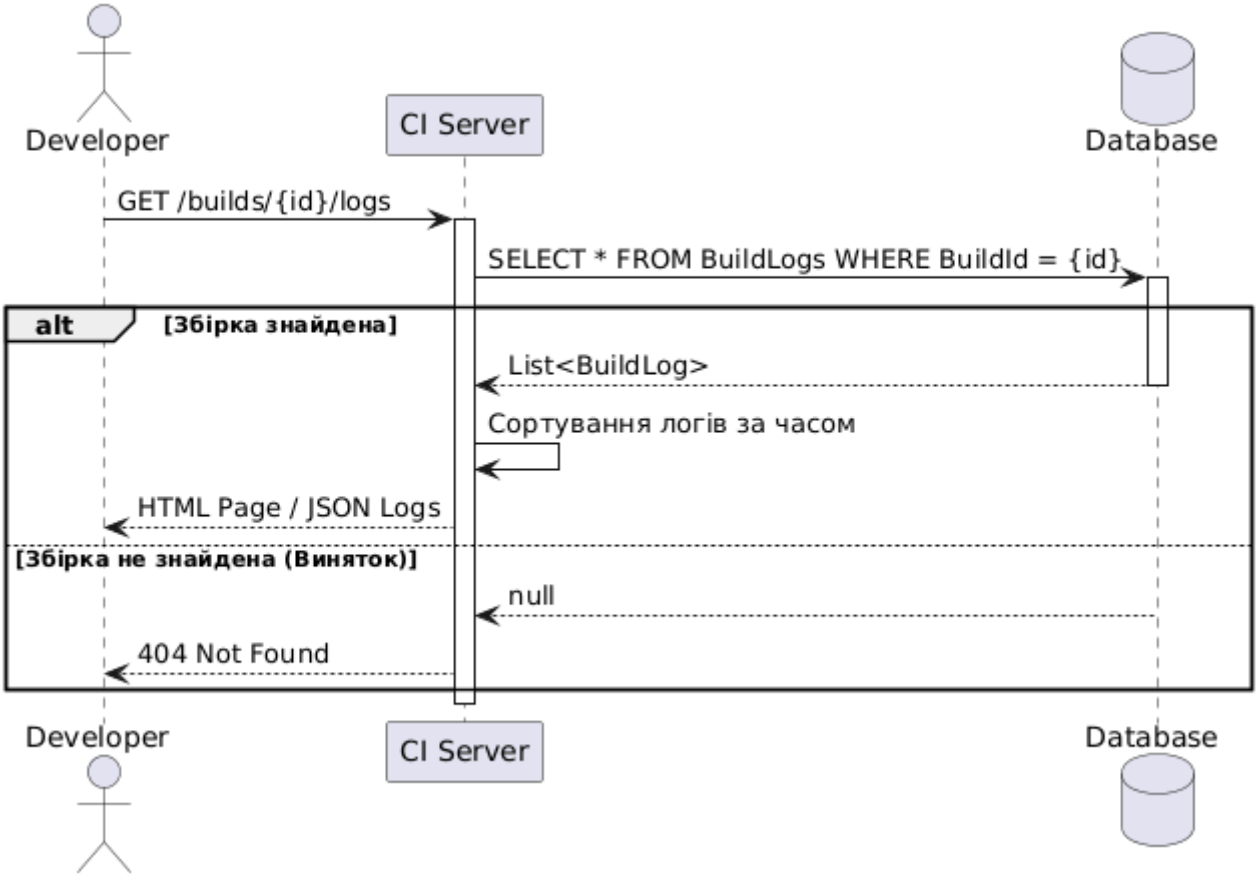

Рис. 1. 4 - Діаграма послідовності «Авторизація користувача»

1.5. Концептуальна модель системи

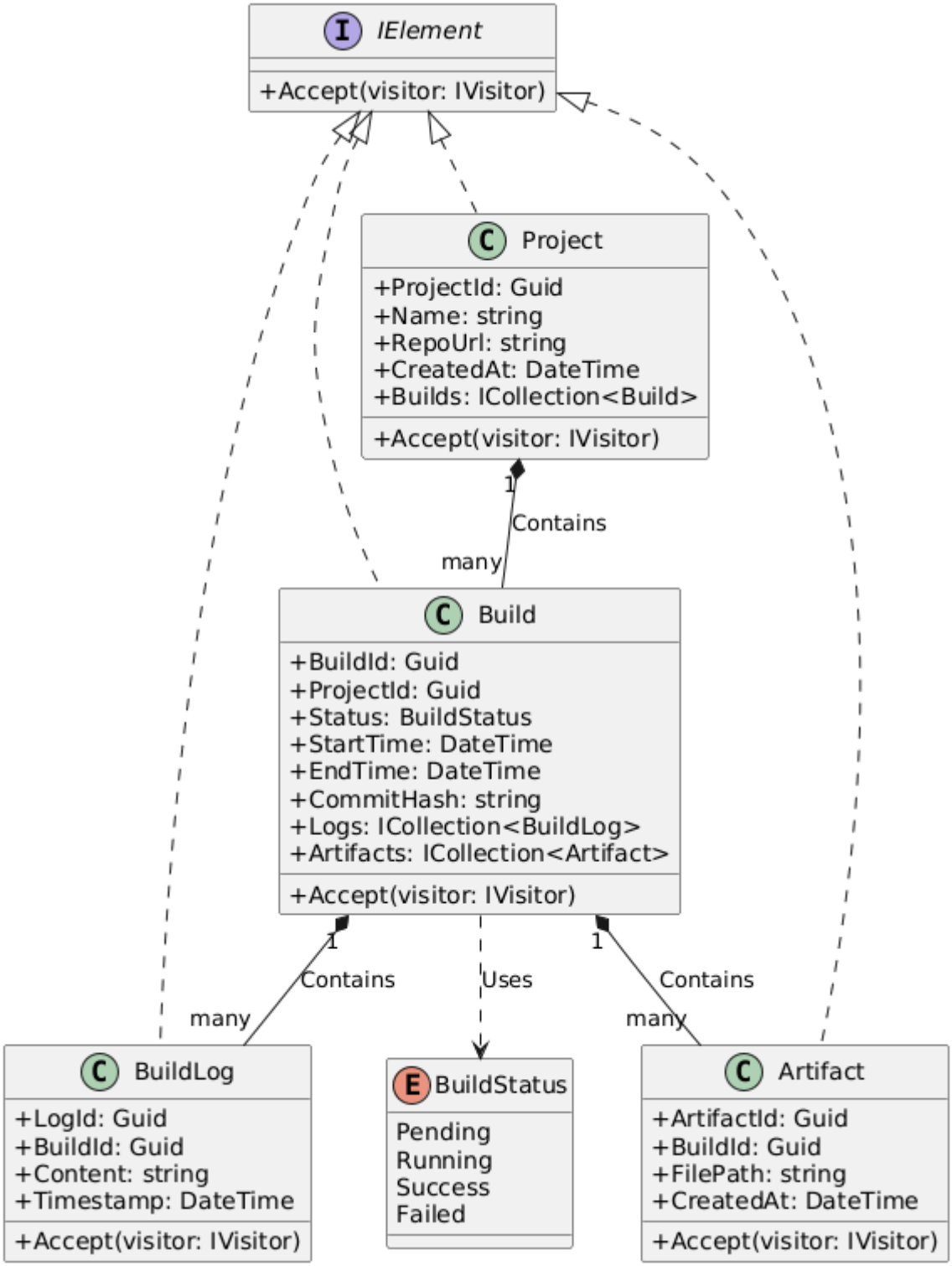

Рис. 1. 5 - Діаграма класів сутностей

Project (Проєкт):

- Є кореневою сутністю системи. Містить метадані про репозиторій програмного коду (назва, URL-адреса Git-репозиторію) та дату створення.

- Має зв'язок "один-до-багатьох" із сутністю Build, що дозволяє зберігати історію всіх запусків збірки для конкретного проєкту.

Build (Збірка):

- Представляє окрему ітерацію запуску пайплайну CI.
- Зберігає інформацію про статус виконання (BuildStatus), час початку та завершення, а також хеш коміту (CommitHash), на основі якого виконувалася збірка.
- Містить колекції залежних об'єктів: логів (Logs) та артефактів (Artifacts).

BuildLog (Лог збірки) та Artifact (Артефакт):

- BuildLog зберігає окремі записи виводу консолі з часовими мітками, що дозволяє відтворити хід виконання процесу в хронологічному порядку.
- Artifact зберігає інформацію про файли, згенеровані в процесі збірки (наприклад, скомпільовані бінарні файли або архіви), включаючи шлях до файлу на сервері.

Реалізація патерну Visitor:

- Усі сутності (Project, Build, BuildLog, Artifact) реалізують інтерфейс IElement, що містить метод Accept(IVisitor visitor).
- Це архітектурне рішення дозволяє відокремити алгоритми обробки структури об'єктів (наприклад, генерацію HTML-звіту або експорт у JSON) від самих класів даних. Логіка обходу дерева об'єктів та їх візуалізації винесена у зовнішній клас HtmlReportVisitor, що забезпечує дотримання принципу єдиної відповідальності (SRP).

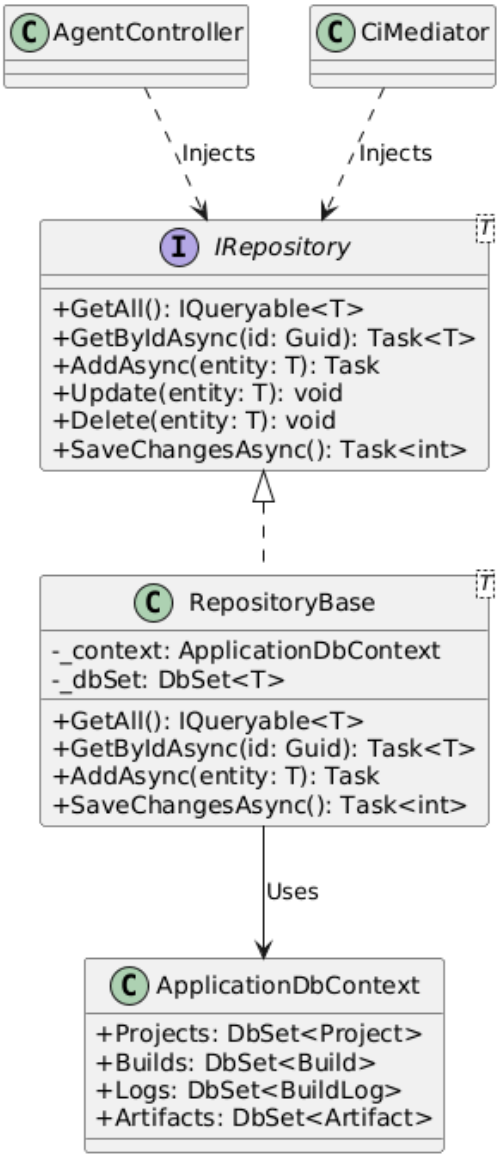

Рис. 1. 6 - Діаграма класів репозиторіїв

Діаграма ілюструє організацію рівня доступу до даних (DAL) з використанням патерну Repository (Репозиторій). Цей підхід забезпечує абстракцію над базою даних та спрощує тестування і підтримку коду.

Ключові елементи діаграми:

1. Інтерфейс IRepository<T>:
    - Визначає контракт для роботи з даними, надаючи уніфікований набір методів для виконання CRUD-операцій (Create, Read, Update, Delete).
    - Використання узагальнень (Generics) <T> дозволяє застосовувати цей інтерфейс до будь-якої сутності системи без дублювання коду.

2. Клас RepositoryBase<T>:
    - Є конкретною реалізацією інтерфейсу IRepository<T>.
    - Використовує контекст бази даних ApplicationDbContext (Entity Framework Core) для безпосередньої взаємодії з БД.
    - Інкапсулює низькорівневу логіку роботи з DbSet, звільняючи бізнес-логіку від прямої залежності від EF Core.
3. Dependency Injection (Впровадження залежностей):
    - Компоненти бізнес-логіки та API-контролери, такі як AgentController та CiMediator, не створюють екземпляри репозиторіїв напряму. Натомість вони отримують абстракцію IRepository<T> через конструктор.
    - Це забезпечує слабку зв'язність (Loose Coupling) компонентів. Наприклад, при необхідності змінити СУБД з MS SQL на SQLite або використати Mock-об'єкти для Unit-тестування, зміни в коді контролерів не будуть потрібні.

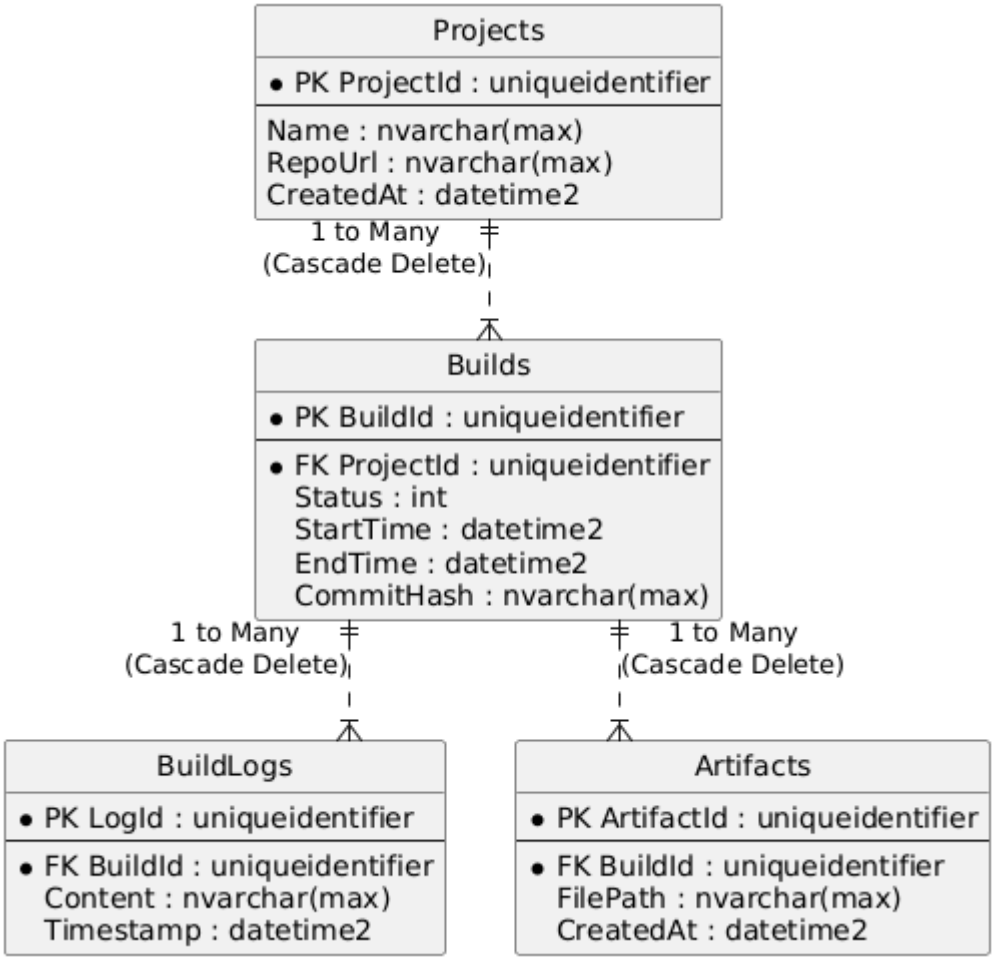

Рис. 1. 7 - Діаграма зв’язків

Ключові елементи та зв'язки:

1. Таблиці (Entities):
    - База даних складається з чотирьох основних таблиць: Projects, Builds, BuildLogs, Artifacts.
    - Кожна таблиця має первинний ключ (PK), який є унікальним ідентифікатором типу uniqueidentifier (GUID), що дозволяє уникати конфліктів ID у розподіленому середовищі.
2. Зв'язки (Relationships):
    - Project — Build: Реалізовано зв'язок "один-до-багатьох" (1:N). Один запис у таблиці Projects може посилатися на множину записів у таблиці Builds через зовнішній ключ ProjectId.

- Build — BuildLog / Artifacts: Таблиця Builds має зв'язки "один-до-багатьох" з таблицями BuildLogs та Artifacts. Це означає, що одна збірка може генерувати тисячі рядків логів та декілька файлів-артефактів.

3. Каскадне видалення (Cascade Delete):
   - Усі зв'язки налаштовані з правилом каскадного видалення. Це критично важлива вимога для цілісності системи: при видаленні Проєкту СУБД автоматично видалить усі пов'язані з ним Збірки, а при видаленні Збірки — всі її Логи та Артефакти. Це запобігає появі "сирітських" записів у базі даних.
4. Типи даних:
   - Для зберігання текстової інформації (назва проєкту, вміст логу) використовується тип nvarchar(max), що дозволяє зберігати дані змінного розміру.
   - Часові мітки зберігаються у форматі datetime2 для високої точності фіксації подій.

1.6. Вибір бази даних

Для забезпечення надійності, цілісності та збереження даних CI-сервера, таких як проєкти, історія збірок, логи та метадані артефактів, було необхідно обрати ефективну систему управління базами даних. Під час проєктування були проаналізовані ключові вимоги до системи: реляційна структура даних, портативність та сумісність з .NET.

Основною вимогою стала реляційна структура даних, оскільки сутності системи мають чіткі зв’язки «один-до-багатьох» (наприклад, Project → Builds → BuildLogs). Це вимагало підтримки зовнішніх ключів та каскадного видалення. Також важливою була портативність (Mobility) — система повинна легко розгортатися на новому середовищі без складного налаштування серверного програмного забезпечення. Крім того, необхідна була повна сумісність з платформою .NET 9 та ORM Entity Framework Core.

На основі аналізу було обрано СУБД SQLite — вбудовану реляційну базу даних, яка не потребує окремого серверного процесу, оскільки її рушій інтегрується безпосередньо в додаток. Основні переваги SQLite для цього проєкту включають:

- Файлову архітектуру: вся база даних зберігається в єдиному кросплатформному файлі (наприклад, ci_server.db), що значно спрощує перенесення проєкту, створення резервних копій та розгортання — достатньо скопіювати файл або дозволити додатку створити його автоматично.
- Zero-configuration: SQLite не вимагає адміністрування, налаштування портів, створення користувачів або прав доступу, що ідеально підходить для локальних інструментів розробки та демонстраційних систем.
- Продуктивність: для завдань CI-сервера, де кількість запитів на запис є помірною, SQLite забезпечує високу швидкість роботи завдяки відсутності накладних витрат на мережеву взаємодію, на відміну від клієнт-серверних СУБД.
- ACID-сумісність: гарантує надійність транзакцій, що критично важливо для збереження логів та статусів збірки.

Як альтернатива розглядався MS SQL Server, але його використання було визнано надлишковим через необхідність встановлення та налаштування служби SQL Server на машині користувача, вищі системні вимоги та складність перенесення бази даних між різними середовищами розробки.

Взаємодія з базою даних реалізована через Entity Framework Core (EF Core), що дозволило:

- застосувати підхід Code-First, коли структура бази даних генерується автоматично на основі класів C# (наприклад, Project, Build);
- абстрагуватися від конкретної СУБД завдяки патерну Repository, що робить бізнес-логіку незалежною від деталей реалізації бази даних;
- легко змінити провайдер даних (наприклад, мігрувати з SQL Server на SQLite) шляхом зміни лише одного рядка конфігурації;

- використовувати LINQ для побудови безпечних та типізованих запитів до даних.

Таким чином, вибір комбінації SQLite + EF Core забезпечує оптимальний баланс між продуктивністю, зручністю розробки та мобільністю кінцевого продукту.

1.7. Вибір мови програмування та середовища розробки

Мова C# була обрана з огляду на уніфіковану екосистему проєкту, що базується на архітектурі SOA і включає як вебсервер, так і консольний агент. Використання однієї мови дозволило створити спільну бібліотеку класів (CiServer.Core), яка забезпечує повторне використання коду, зокрема моделей та DTO, а також гарантує сумісність контрактів даних між клієнтом і сервером. Крім того, C# як об'єктно-орієнтована мова зі строгою типізацією ідеально підходить для реалізації складних патернів проєктування, таких як State, Command та Visitor, які становлять основу бізнес-логіки проєкту. Важливою перевагою також стала потужна підтримка асинхронного програмування (async/await), яка критично важлива для високопродуктивної обробки HTTP-запитів та виконання тривалих операцій збірки без блокування потоків.

Для реалізації серверної частини та API було обрано фреймворк ASP.NET Core, який надає низку переваг, необхідних для сучасних CI-систем. По-перше, він забезпечує високу продуктивність та кросплатформеність, дозволяючи запускати сервер і агенти збірки на різних операційних системах, таких як Windows, Linux та macOS. По-друге, архітектура MVC та підтримка Web API дозволяють чітко розділяти логіку обробки запитів і відображення даних, а також створювати зручні RESTful API, через які взаємодіють агенти системи. Вбудований механізм Dependency Injection (DI) значно спростив інтеграцію архітектурних компонентів, таких як репозиторії, сервіси та медіатор (CiMediator). Крім того, підтримка Real-time комунікації завдяки бібліотеці SignalR дозволила легко реалізувати трансляцію логів збірки в реальному часі через WebSockets.

Як основне середовище розробки було обрано Visual Studio Code (VS Code) — легкий і кросплатформений редактор коду, який завдяки розширенню C# Dev Kit надає потужні інструменти для налагодження та рефакторингу. Важливою перевагою є наявність інтегрованого терміналу, який активно використовувався для керування міграціями бази даних (EF Core CLI), запуску розподілених компонентів системи та роботи з системою контролю версій Git.

Таким чином, комбінація C#, ASP.NET Core та VS Code забезпечила оптимальні умови для розробки високопродуктивної, масштабованої та кросплатформної CI-системи.

1.8. Проєктування розгортання системи

Для повного розуміння архітектури системи необхідно розглянути її з двох точок зору: логічної та фізичної. Логічна архітектура визначає, з яких програмних частин складається додаток, тоді як фізична архітектура показує, як вони розміщуються на реальному обладнанні.

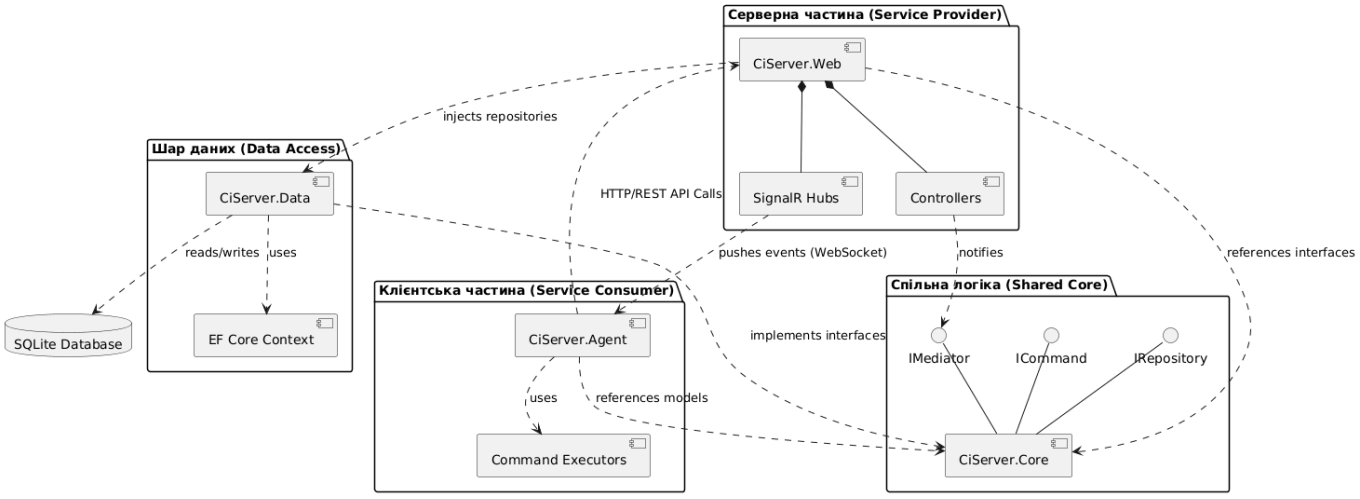

Рис. 1. 8 - Діаграма компонентів

Система складається з наступних ключових компонентів:

1. CiServer.Core (Ядро):

- Центральний компонент, що містить бізнес-логіку, сутності доменної області (Project, Build), інтерфейси патернів (ICommand, IRepository, IMediator) та DTO (Data Transfer Objects).
- Він не має залежностей від інших проєктів, що забезпечує чистоту архітектури. Усі інші компоненти залежать від Core.

2. CiServer.Web (Веб-сервер):
   - Виступає точкою входу для користувачів та агентів.
   - Містить Controllers (API для агента та MVC для користувача) та SignalR Hubs для забезпечення зв'язку в реальному часі.
   - Використовує CiServer.Core для обробки логіки та делегує доступ до даних компоненту CiServer.Data.
3. CiServer.Agent (Агент збірки):
   - Автономний клієнтський додаток, що виконує команди пайплайну (клонування, компіляція).
   - Залежить від CiServer.Core для отримання контрактів команд (ICommand).
   - Взаємодіє з сервером виключно через мережеві протоколи.
4. CiServer.Data (Доступ до даних):
   - Інкапсулює логіку роботи з базою даних. Реалізує інтерфейси репозиторіїв, визначені в Core, використовуючи ORM Entity Framework Core.

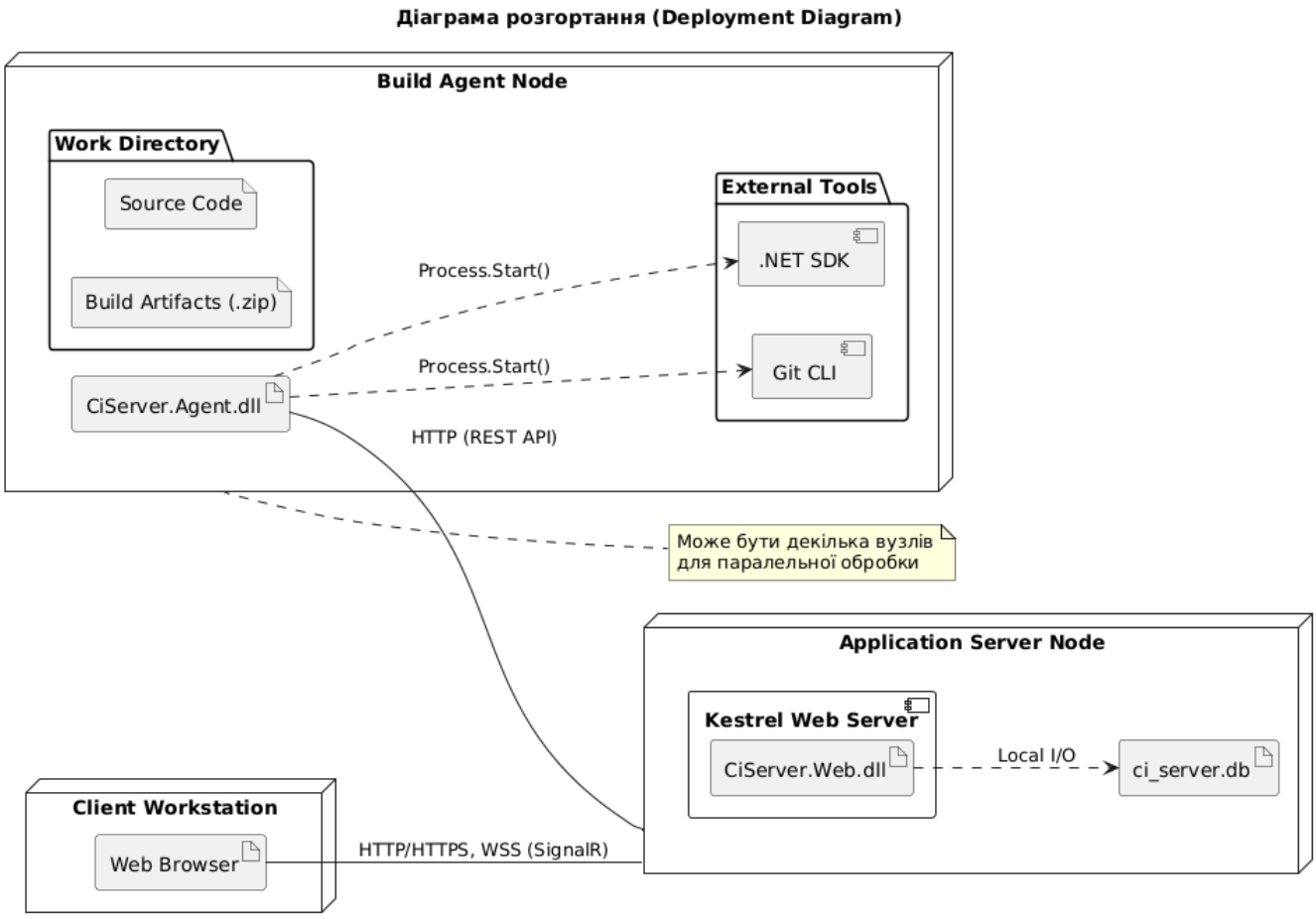

Рис. 1. 9 - Діаграма розгортання

Фізична архітектура системи є розподіленою і складається з трьох типів вузлів, кожен з яких виконує специфічні функції, забезпечуючи ефективну роботу CI-сервера.

Application Server Node (Сервер додатка) є центральним вузлом системи, на якому розгорнуто веб-застосунок CiServer.Web під управлінням вбудованого веб-сервера Kestrel (частина ASP.NET Core). На цьому вузлі також розташований файл бази даних ci_server.db (SQLite), оскільки SQLite — це вбудована СУБД, яка не вимагає окремого серверного процесу. Доступ до бази даних здійснюється безпосередньо як до локального файлу. Сервер обробляє HTTP-запити від користувачів, а також REST API запити від агентів збірки, забезпечуючи централізоване управління системою.

Build Agent Node (Вузол збірки) — це окремий фізичний або віртуальний сервер, на якому виконується консольний додаток CiServer.Agent. Для коректної роботи на цьому вузлі обов’язково мають бути встановлені необхідні інструменти розробки,

такі як Git (для клонування репозиторіїв) та .NET SDK (для компіляції та тестування проєктів). Агент створює тимчасові робочі директорії для вихідного коду, виконує процеси збірки та генерує артефакти, які потім передає на центральний сервер через мережу. Така організація дозволяє розподіляти навантаження та паралельно виконувати збірку на різних вузлах.

Client Workstation (Робоча станція клієнта) — це комп’ютер розробника або адміністратора, з якого здійснюється взаємодія з системою через веб-браузер. Зв’язок із сервером відбувається за допомогою протоколів HTTPS (для завантаження веб-сторінок) та WSS (WebSocket Secure) для отримання логів збірки в реальному часі через SignalR. Це забезпечує зручну та безпечну роботу з системою, дозволяючи користувачам оперативно відстежувати стан збірки та отримувати актуальну інформацію.

Така розподілена архітектура дозволяє легко масштабувати систему шляхом додавання нових Build Agent Node, не змінюючи конфігурацію центрального сервера. Це повністю відповідає принципам SOA (сервіс-орієнтованої архітектури), забезпечуючи гнучкість, надійність та високу продуктивність системи.

2 РЕАЛІЗАЦІЯ КОМПОНЕНТІВ СИСТЕМИ

2.1. Структура бази даних

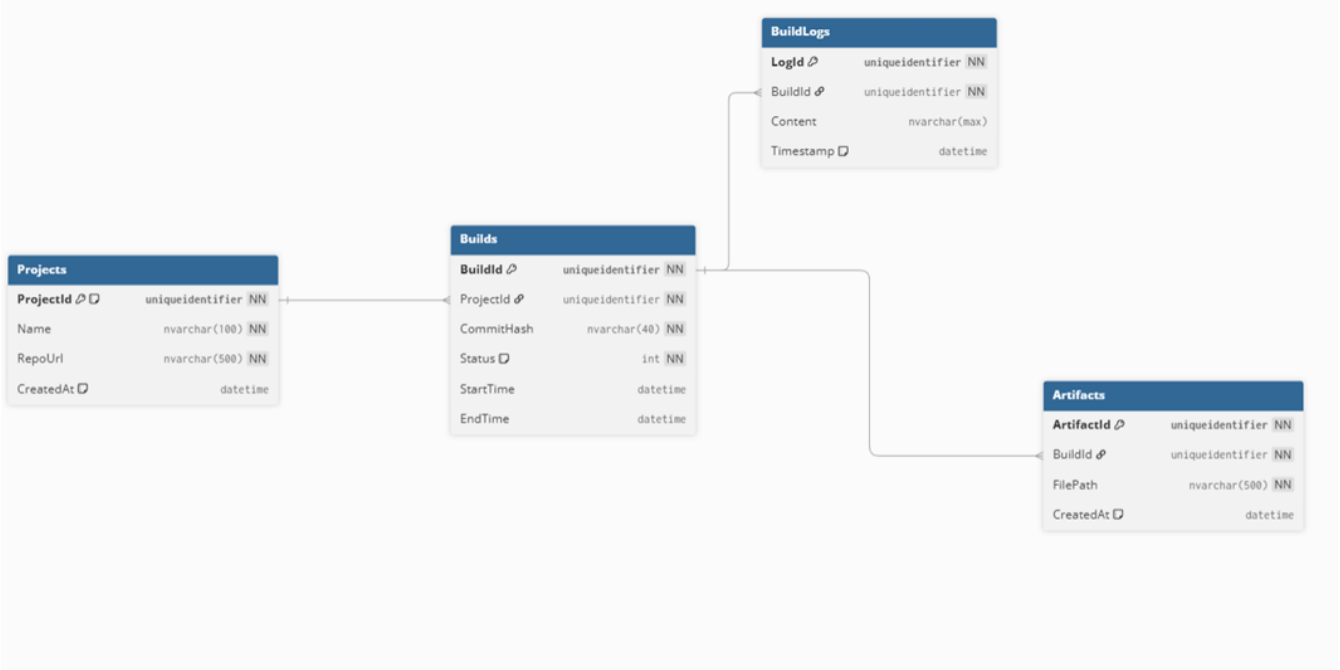

Рис. 2.1 – Проектування бази даних

- Projects (Проєкти): Головна сутність системи. Зберігає метадані та налаштування репозиторію (Назва, URL). Виступає кореневим елементом для каскадного видалення.
- Builds (Збірки): Сутність, що представляє історію запусків. Пов'язана з таблицею Projects відношенням «один-до-багатьох» (1:N). Поле Status зберігається як цілочисельний тип (int) і відображає стан процесу (Pending, Running, Success, Failed).
- BuildLogs (Логи): Зберігає детальний вивід консолі для кожного кроку виконання. Пов'язана з конкретним записом у таблиці Builds. Налаштовано каскадне видалення: при очищенні історії збірок відповідні логи видаляються автоматично.
- Artifacts (Артефакти): Зберігає інформацію про згенеровані файли (наприклад, скомпільовані архіви .zip). Містить шлях до файлу на сервері (FilePath) та прив'язку до конкретної збірки (BuildId).

2.2. Архітектура системи

2.2.1. Специфікація системи

Розроблена система «CI Server» є розподіленим програмним комплексом, побудованим на базі платформи .NET 9. Архітектура системи спроєктована за принципом сервіс-орієнтованої архітектури (SOA), що передбачає фізичне розділення логіки керування (вебсервер) та логіки виконання завдань (агент). Такий підхід забезпечує масштабованість, надійність та можливість виконання ресурсномістких операцій збірки на окремих вузлах.

Платформа	.NET 9 (Runtime & SDK)
Мова програмування	C# 12.
Веб-фреймворк	ASP.NET Core MVC (для сервера)
Доступ до даних (ORM)	Entity Framework Core 9.0
База даних	SQLite (вбудована реляційна база даних)
Комунікація Real-time	SignalR (WebSockets) для трансляції логів та оновлення статусів у реальному часі
Клієнтська частина	Razor Views (.cshtml), HTML5, CSS3 (Bootstrap 5), JavaScript.
Інструменти збірки	System.Diagnostics.Process (для виклику зовнішніх процесів: Git CLI, Dotnet CLI)
Архітектура	SOA (Service-Oriented Architecture), N-Layer (у веб-частині)

Таблиця 2.2. Технологічний стек

Структура рішення

Програмний код організовано у вигляді рішення (Solution), що складається з чотирьох логічно ізольованих проєктів, кожен з яких має свою зону відповідальності:

1. CiServer.Core (Ядро системи / Shared Kernel):

- Є центральним компонентом, від якого залежать інші проєкти.
- Містить основні сутності (Project, Build, BuildLog, Artifact).

- Визначає інтерфейси (контракти) для патернів та сервісів (IRepository, ICommand, IMediator, IRealTimeNotifier), забезпечуючи принцип інверсії залежностей (DIP).
- Містить реалізацію поведінкових патернів, що є спільними для клієнта і сервера (наприклад, логіка Mediator для обробки подій збірки).

2. CiServer.Data (Шар доступу до даних):

- Відповідає за взаємодію з базою даних SQLite.
- Містить контекст бази даних ApplicationDbContext (наслідується від DbContext).
- Реалізує інтерфейс IRepository<T> через узагальнений клас RepositoryBase<T>, використовуючи Entity Framework Core для виконання запитів.
- Забезпечує автоматичне створення схеми бази даних при старті додатка (EnsureCreated).

3. CiServer.Web (Серверна частина / Presentation Layer):

- Головний керуючий вузол системи (ASP.NET Core Web Application).
- Містить MVC Контролери (ProjectsController), які забезпечують взаємодію з користувачем через веб-інтерфейс.
- Містить API Контролери (AgentController), через які агенти отримують завдання та звітують про результати.
- Реалізує SignalR Hub (BuildHub) для передачі логів у браузер без перезавантаження сторінки.
- Відповідає за збереження та віддачу статичних файлів (артефактів збірки).

4. CiServer.Agent (Виконавець / Worker Service):

- Автономний консольний додаток, що працює на окремому процесі або сервері.
- Реалізує патерн Polling, періодично опитуючи сервер на наявність нових завдань.
- Використовує патерн Command для виконання етапів пайплайну (клонування репозиторію, компіляція, тестування).

- Відповідає за перехоплення виводу консолі (STDOUT/STDERR) та відправку його на сервер.

2.2.2. Вибір та обґрунтування патернів реалізації

1) State(стан)

Для системи реалізовано патерн State для керування життєвим циклом збірки (Build). Оскільки збірка може перебувати в різних станах (Pending, Running, Success, Failed), і поведінка системи (наприклад, можливість скасування або запуску) залежить від поточного стану, використання великих конструкцій switch-case або if-else у класі Build призвело б до заплутаного коду. Патерн State дозволяє інкапсулювати поведінку кожного статусу в окремий клас, роблячи переходи між станами (Transitions) прозорими та керованими.

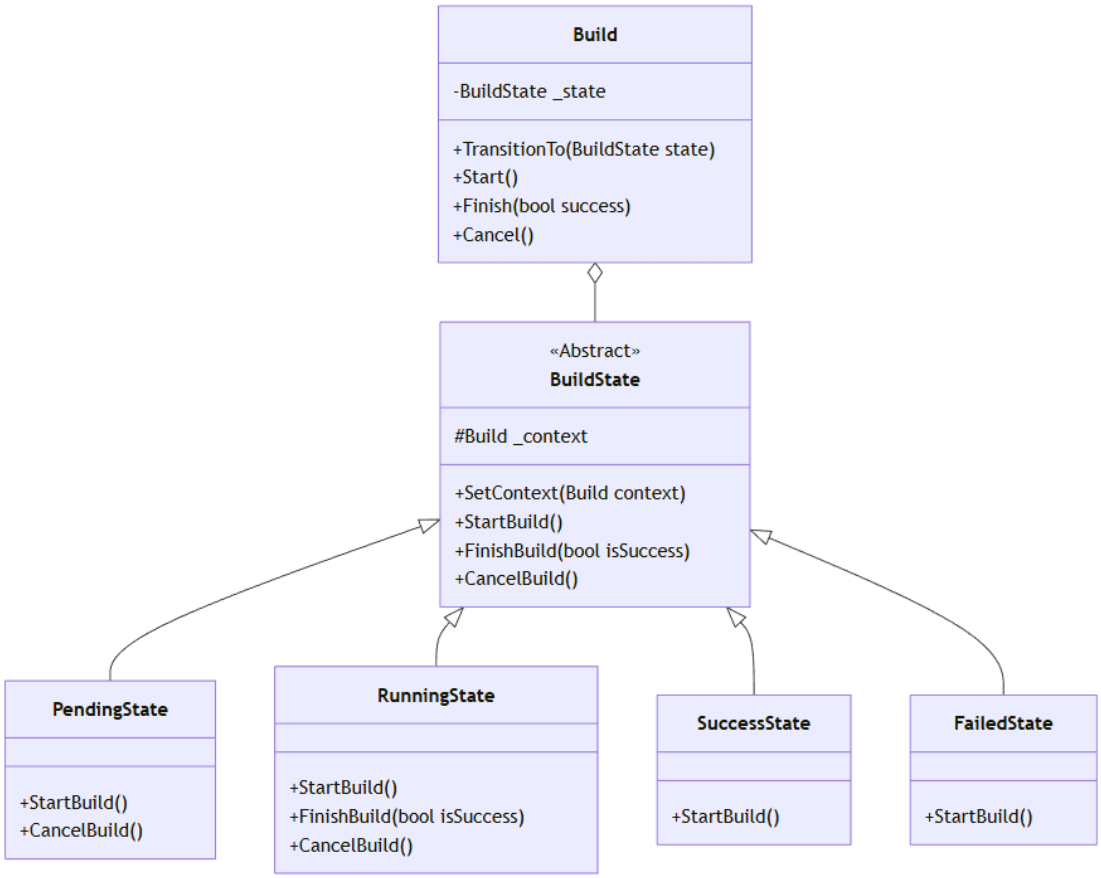

Рис. 2.2 - Структура патерну State

Build (Context) – це центральна сутність, яка зберігає посилання на поточний стан (_state) і делегує йому виконання операцій (Start, Finish, Cancel). Вона також містить метод TransitionTo, що дозволяє змінювати стан під час виконання.

BuildState (State) – це абстрактний клас, який визначає інтерфейс для всіх можливих станів збірки. Він задає методи StartBuild, FinishBuild та CancelBuild, які повинні бути реалізовані або перевизначені в конкретних станах.

Concrete States (Pending, Running, Success, Failed) – це конкретні реалізації станів.

- PendingState: Початковий стан. Дозволяє перехід у RunningState (початок збірки) або FailedState (скасування).
- RunningState: Активний стан. Дозволяє завершити збірку успішно (SuccessState) або з помилкою (FailedState). Блокує повторний запуск.
- SuccessState / FailedState: Фінальні стани. Вони забороняють будь-які подальші дії (збірка вже завершена), гарантуючи цілісність даних.

2) Command

Для системи реалізовано патерн Command для організації процесу виконання пайплайну збірки. Оскільки процес CI (Continuous Integration) складається з різних етапів (клонування репозиторію, компіляція, запуск тестів), жорстке кодування цієї послідовності в одному класі зробило б систему негнучкою і складною для розширення. Патерн Command дозволяє інкапсулювати кожну дію (операцію) в окремий об'єкт із загальним інтерфейсом. Це дає змогу будувати пайплайни динамічно (додавати або прибирати кроки), ставити команди в чергу та легко додавати нові типи завдань (наприклад, деплой або лінтер) без зміни коду виконавця.

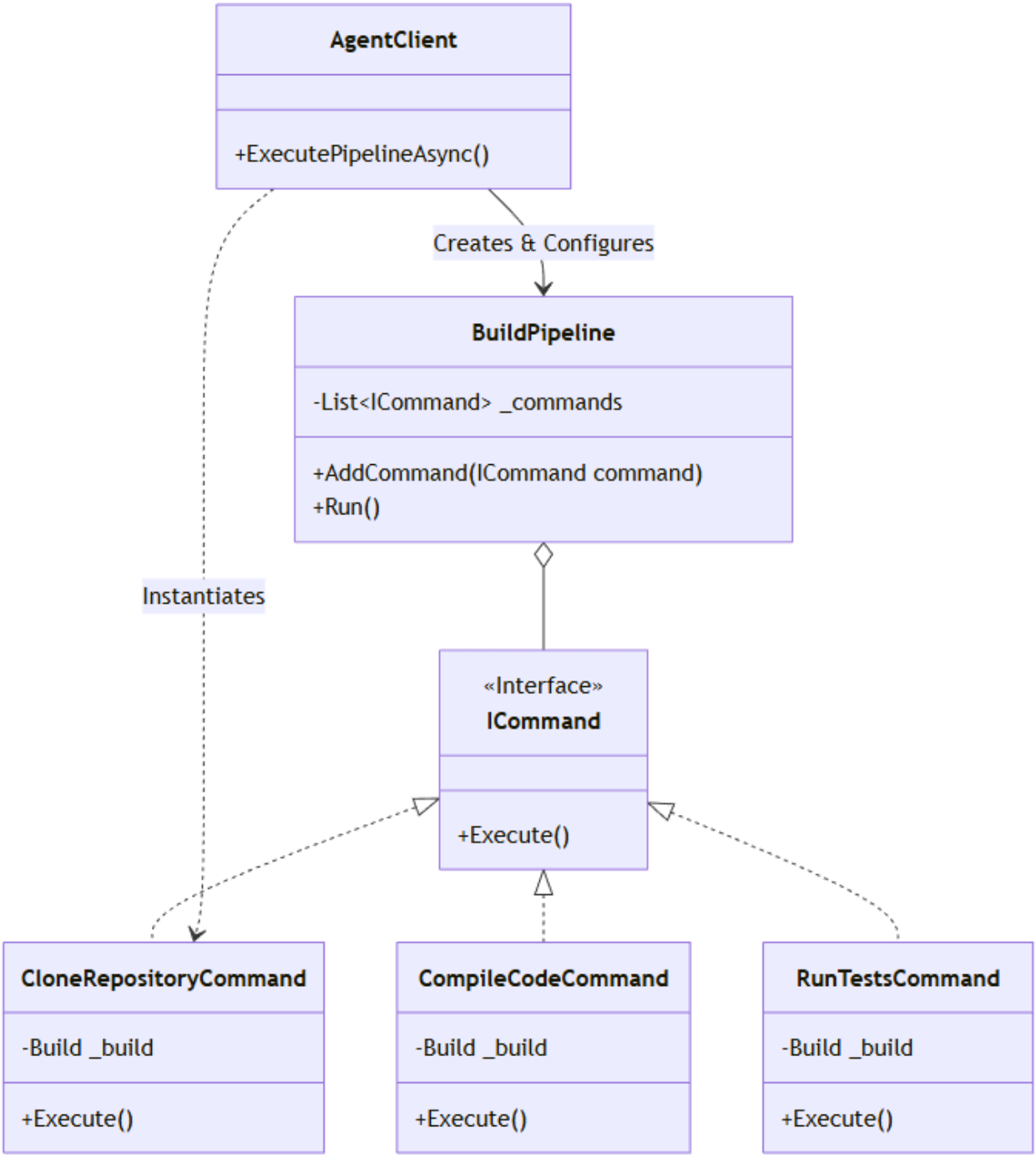

Рис. 2.3 - Структура патерну Command

ICommand (Command) – це загальний інтерфейс для всіх команд. Він оголошує єдиний метод Execute(), який запускає виконання операції.

Concrete Commands (CloneRepositoryCommand, CompileCodeCommand, RunTestsCommand) – конкретні реалізації команд. Кожен клас містить власну логіку:

- CloneRepositoryCommand відповідає за взаємодію з git.
- CompileCodeCommand відповідає за виклик dotnet build.

- Всі вони зберігають посилання на контекст (Build), необхідний для виконання (наприклад, URL репозиторію).

BuildPipeline (Invoker) – ініціатор виконання. Він зберігає список команд (_commands) і по черзі викликає у них метод Execute(). Цей клас не знає деталей реалізації команд, він лише керує порядком їх запуску.

AgentClient (Client) – клієнтський код (у файлі Program.cs агента). Він створює об'єкт пайплайну, налаштовує його (додає необхідні команди в потрібному порядку) і запускає виконання.

3) Decorator

Для системи реалізовано патерн Decorator (Декоратор) для додання функціоналу логування та звітності до команд збірки. У початковій реалізації команди (CloneRepositoryCommand, CompileCodeCommand) відповідали лише за виконання своєї безпосередньої роботи. Однак, для CI сервера критично важливо відправляти статус виконання ("Start", "Success", "Error") на сервер у реальному часі. Внесення коду HTTP-запитів безпосередньо в класи команд порушило б Single Responsibility Principle (клас відповідав би і за компіляцію, і за мережеву взаємодію) та Open/Closed Principle (довелося б змінювати робочий код). Патерн Decorator дозволив створити клас-обгортку HttpReportDecorator, який перехоплює виклик Execute(), відправляє лог на сервер, а потім делегує виконання вкладеній команді. Це дозволило додати нову поведінку динамічно, не змінюючи код самих команд.

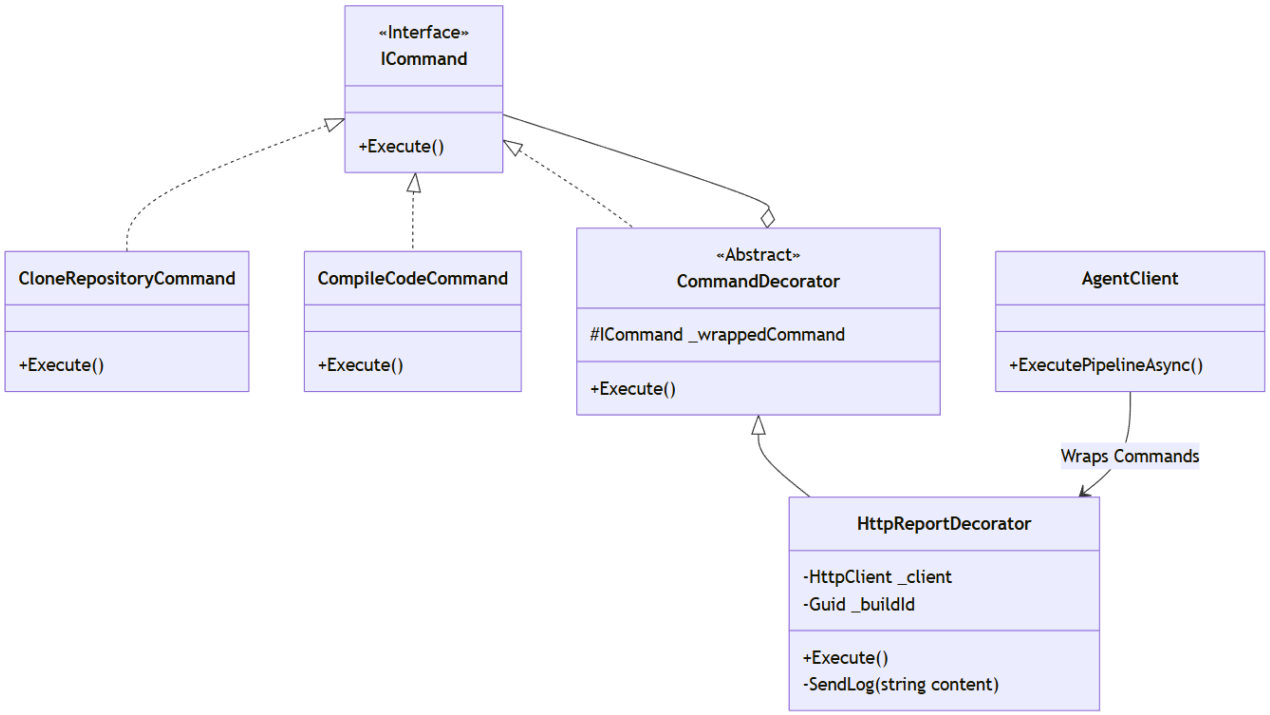

Рис. 2.4 - Структура патерну Decorator

ICommand (Component) – спільний інтерфейс для компонентів і декораторів. Гарантує, що декоратор можна використовувати там же, де і звичайну команду.

Concrete Components (CloneRepositoryCommand, CompileCodeCommand) – "чисті" класи, які виконують основну роботу (робота з Git, компіляція). Вони нічого не знають про звіти чи HTTP.

CommandDecorator (Decorator) – абстрактний клас, що зберігає посилання на об'єкт ICommand (_wrappedCommand) і делегує йому виконання методу Execute.

HttpReportDecorator (Concrete Decorator) – конкретна обгортка. У методі Execute вона:

- Відправляє лог про початок роботи.
- Викликає базовий Execute (реальну роботу).
- У разі успіху або помилки відправляє відповідний лог на сервер.

AgentClient (Client) – клієнтський код, який "одягає" команди в декоратори перед додаванням їх у пайплайн.

4) Mediator

Для системи реалізовано патерн Mediator (Посередник) для зменшення зв’язності (Coupling) між компонентами сервера. У веб-застосунках часто виникає проблема "товстих контролерів" (Fat Controllers), коли клас контролера (AgentController) не тільки приймає HTTP-запити, а й виконує бізнес-логіку: звертається до репозиторіїв, оновлює статуси в БД, розраховує час виконання тощо. Це робить контролер складним для підтримки та тестування. Використання патерну Mediator дозволило винести логіку взаємодії між компонентами в окремий клас — CiMediator. Тепер контролер лише сповіщає посередника про подію (наприклад, "JobFinished"), а посередник сам вирішує, які сервіси чи репозиторії потрібно задіяти для обробки цієї події.

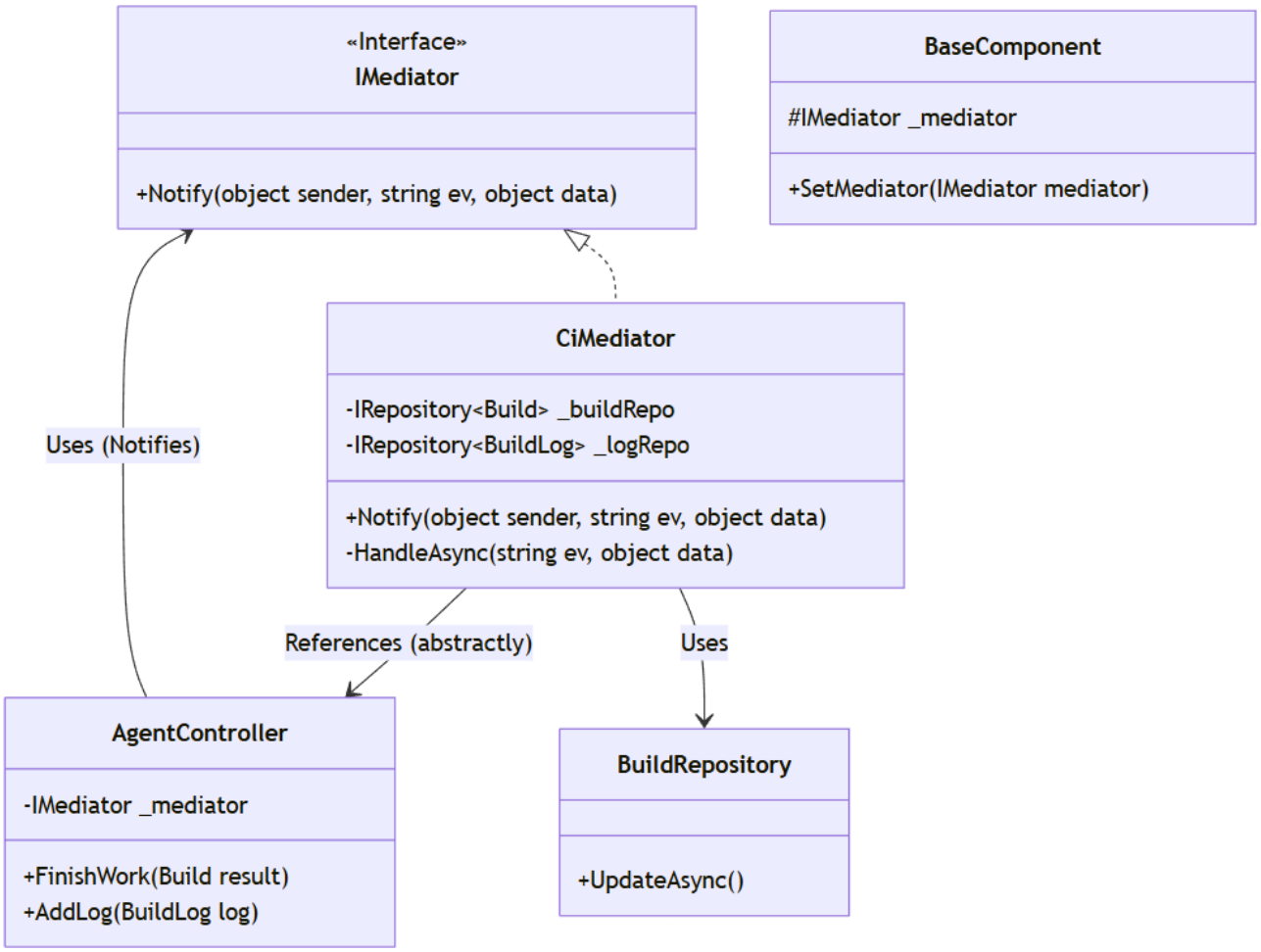

Рис. 2.5 - Структура патерну Mediator

IMediator (Mediator) – інтерфейс, що визначає метод Notify. Він дозволяє компонентам спілкуватися, не знаючи конкретних класів один одного.

CiMediator (Concrete Mediator) – реалізує логіку координації. Він містить залежності від репозиторіїв (_buildRepo, _logRepo). Коли він отримує повідомлення (наприклад, "JobFinished"), він знаходить відповідний білд у базі, оновлює його статус та час завершення.

AgentController (Component/Colleague) – компонент, який ініціює події. У методах FinishWork та AddLog він не працює з базою даних напряму. Замість цього він викликає _mediator.Notify().

Перевага: Контролер нічого не знає про те, як саме зберігається результат збірки. Якщо логіка збереження зміниться (наприклад, додасться відправка email), нам треба буде змінити лише CiMediator, не чіпаючи контролер.

5) Visitor

Для системи реалізовано патерн Visitor (Відвідувач) для генерації звітів про результати збірки. У системі існує складна ієрархічна структура даних: Project містить список Builds, які в свою чергу містять BuildLogs. Нам необхідно виконувати над цією структурою операцію експорту даних (в даному випадку — генерацію HTML-сторінки). Додавання методу ToHtml() безпосередньо в класи сутностей (Project, Build) порушило б принцип єдиної відповідальності (SRP), оскільки бізнес-сутності не повинні відповідати за візуальне представлення. Крім того, якщо в майбутньому знадобиться експорт у JSON або XML, довелося б змінювати всі ці класи. Патерн Visitor дозволив винести алгоритм обходу структури та генерації звіту в окремий клас HtmlReportVisitor. Сутності лише надають метод Accept, який дозволяє відвідувачу отримати доступ до їхнього стану.

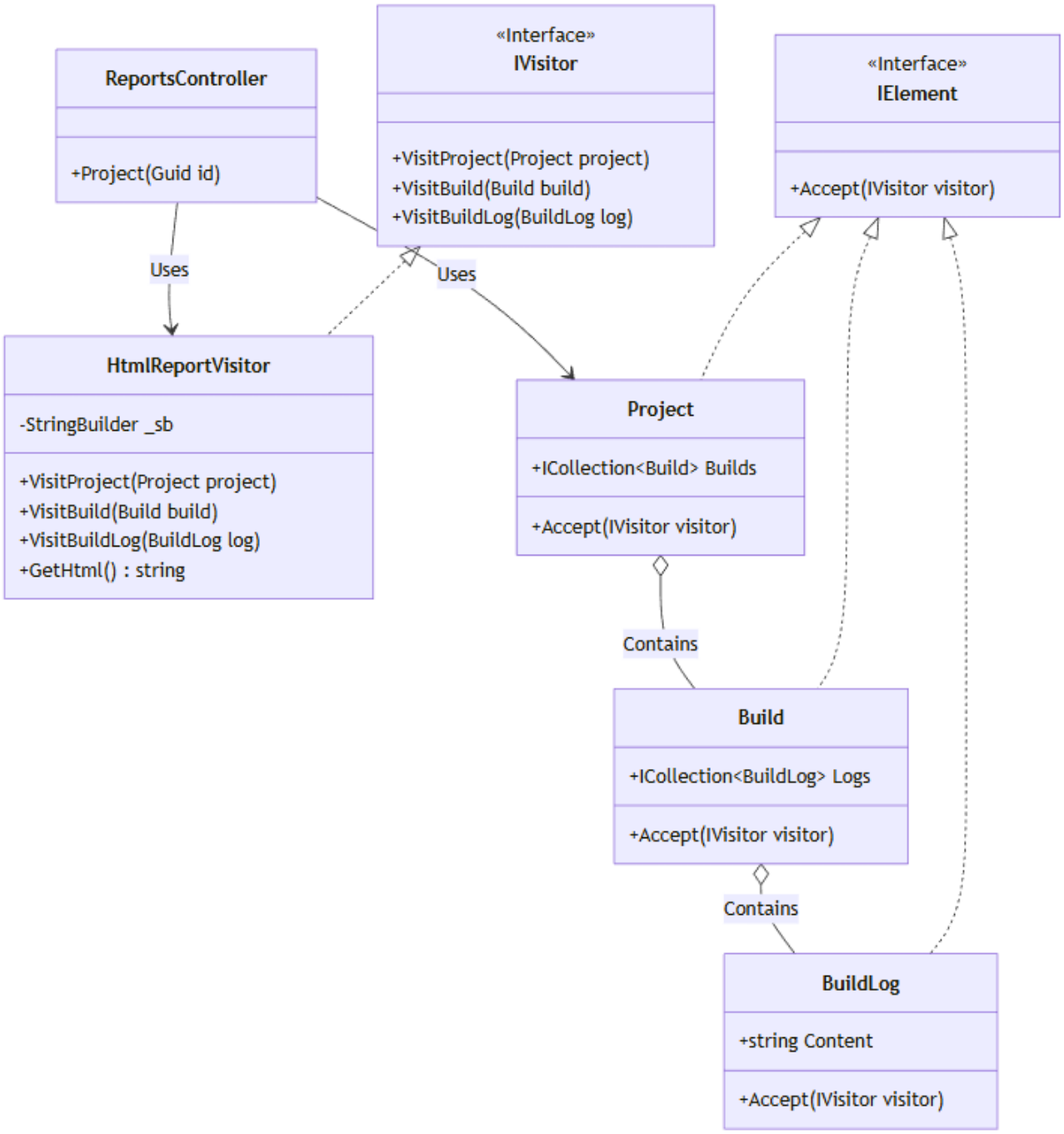

Рис. 2.6 - Структура патерну Visitor

- IVisitor (Visitor) – інтерфейс, що оголошує методи відвідування для кожного типу елемента в структурі об'єктів (VisitProject, VisitBuild, VisitBuildLog).
- HtmlReportVisitor (Concrete Visitor) – реалізує логіку генерації звіту. Він накопичує HTML-розмітку у StringBuilder. Для кожного елемента він знає, як його відобразити (наприклад, фарбує статус білда у зелений або червоний колір).

- IElement (Element) – інтерфейс, який зобов'язує класи мати метод Accept(IVisitor visitor).
- Project, Build, BuildLog (Concrete Elements) – класи даних. У методі Accept вони викликають відповідний метод відвідувача (наприклад, visitor.VisitBuild(this)), а потім передають відвідувача своїм дочірнім елементам (наприклад, Build перебирає свої Logs і викликає Accept для кожного). Це забезпечує рекурсивний обхід дерева об'єктів.
- ReportsController (Client) – клієнтський код, який створює відвідувача і запускає процес, викликаючи project.Accept(visitor).

2.3. Інструкція користувача

Експлуатація програмного комплексу «CI Server» передбачає взаємодію з двома незалежними модулями: сервером управління та агентом виконання. Для коректної роботи системи необхідно, щоб на робочій станції було встановлено середовище виконання .NET 9.0 та клієнт Git.

Розгортання та запуск системи

Робота з комплексом розпочинається з ініціалізації серверної частини. Для цього у терміналі, відкритому в директорії веб-застосунку (CiServer.Web), виконується команда запуску. Після старту сервер починає прослуховувати вхідні HTTP-запити за локальною адресою (за замовчуванням http://localhost:5086), надаючи доступ до графічного веб-інтерфейсу.

Паралельно необхідно запустити модуль виконавця. У окремому вікні терміналу, в директорії агента (CiServer.Agent), ініціюється процес, який автоматично переходить у режим опитування (Polling). Агент встановлює зв'язок із сервером через REST API та очікує на появу нових завдань у черзі.

Створення та налаштування проєктів

Взаємодія користувача з системою відбувається через веб-інтерфейс. На головній сторінці відображається панель керування проєктами. Для додавання нового репозиторію користувач переходить до форми створення, де вказує унікальну назву проєкту та HTTPS-посилання на публічний Git-репозиторій. Після підтвердження

система створює відповідний запис у базі даних, і проєкт стає доступним для виконання збірок.

Виконання збірки та моніторинг

Процес неперервної інтеграції ініціюється користувачем вручну на сторінці деталей проєкту. Після натискання кнопки запуску система створює нове завдання зі статусом Pending та автоматично перенаправляє користувача на сторінку моніторингу.

Як тільки вільний агент отримує завдання, статус збірки змінюється на Running. Ключовою особливістю системи є відображення ходу виконання в реальному часі. Завдяки технології SignalR, консольний вивід команд агента (клонування, компіляція, тестування) миттєво транслюється у вікно браузера користувача без необхідності перезавантаження сторінки. Це дозволяє оперативно виявляти помилки на ранніх етапах.

Робота з результатами

По завершенню виконання пайплайну система фіксує фінальний статус (Success або Failed). У разі успішної збірки агент автоматично архівує скомпільовані файли та завантажує їх на сервер. Користувачеві стає доступною кнопка для завантаження отриманого ZIP-архіву (артефакту). Також система надає можливість переглянути детальний HTML-звіт про збірку або видалити проєкт. При видаленні проєкту спрацьовує механізм каскадного очищення, який видаляє з бази даних усю історію запусків, а з файлової системи сервера — усі пов'язані фізичні файли артефактів.

ВИСНОВКИ

У ході виконання курсової роботи було спроєктовано та реалізовано програмний комплекс «CI Server» — систему безперервної інтеграції, призначену для автоматизації процесів збірки, тестування та управління програмним забезпеченням. В основу розробки покладено принципи сервіс-орієнтованої архітектури (SOA), що дозволило фізично розділити систему на два незалежні компоненти: вебсервер, який відповідає за керування процесами та взаємодію з користувачами, та автономний агент, призначений для виконання ресурсомістких завдань, таких як компіляція та тестування. Така архітектура забезпечує високу масштабованість системи, оскільки дозволяє легко розширювати її горизонтально шляхом додавання нових вузлів збірки без зміни конфігурації центрального сервера.

Процес реалізації системи супроводжувався вирішенням цілої низки технічних завдань. Для організації даних було обрано SQLite у поєднанні з підходом Code-First на базі Entity Framework Core, що забезпечило мобільність системи, надійне зберігання реляційних даних та підтримку каскадного видалення записів. Це дозволило уникнути складного адміністрування бази даних та спростило процеси розгортання та резервного копіювання.

Важливим аспектом стало впровадження технології SignalR на базі протоколу WebSockets, яка забезпечила миттєву трансляцію логів збірки від агента безпосередньо до браузера клієнта. Це значно підвищило зручність моніторингу процесів, дозволяючи розробникам та адміністраторам оперативно відстежувати хід виконання завдань у реальному часі.

Для забезпечення гнучкості бізнес-логіки активно застосовувалися патерни проєктування, такі як Command, State, Decorator, Mediator та Visitor. Це дозволило створити чистий, модульний код, який легко піддається розширенню, модифікації та тестуванню, що особливо важливо для систем, які постійно розвиваються.

Окрему увагу було приділено керуванню артефактами — результатами успішної збірки. Реалізовано механізм їх архівації та надання доступу через зручний

веб-інтерфейс, що дозволяє користувачам швидко знаходити та завантажувати необхідні бінарні файли.

В результаті роботи було створено повнофункціональний прототип CI-сервера, який підтримує повний цикл роботи: від створення проєкту та ручного запуску збірки до перегляду детальних звітів та завантаження готових артефактів. Система продемонструвала стабільну роботу, повністю відповідає висунутим функціональним та нефункціональним вимогам, а також має сучасний та інтуїтивно зрозумілий користувацький інтерфейс.

Набутий досвід у роботі з платформою .NET 9, асинхронним програмуванням та розподіленими системами є актуальним і цінним для вирішення сучасних завдань у сфері DevOps та розробки корпоративного програмного забезпечення. Цей проєкт став важливим кроком у вивченні та застосуванні передових технологій, які сьогодні визначають напрямки розвитку IT-індустрії.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Troelsen A., Japikse P. C# 7.0 and .NET Core 2.0 – with Visual Studio 2017. – Apress, 2017. – [Електронний ресурс]. – Режим доступу: https://dl.ebooksworld.ir/motoman/Apress.Pro.Csharp.7.With.NET.and.NET.Core.www.EBooksWorld.ir.pdf .
2. Yokogawa Electric Corporation. Collaborative Information Server (CI Server). – Yokogawa Electric Corporation, 2024. – [Електронний ресурс]. – Режим доступу: https://web-material3.yokogawa.com/BU36K01A10-01EN.pdf .
3. Entity Framework Core Documentation. Documentation for Entity Framework Core – [Електронний ресурс] // Microsoft Learn. – Режим доступу: https://learn.microsoft.com/en-us/ef/core/.
4. ASP.NET Core Documentation. Documentation for ASP.NET Core – [Електронний ресурс] // Microsoft Learn. – Режим доступу: https://learn.microsoft.com/en-us/aspnet/core/.
5. Microsoft Learn. Real-time ASP.NET with SignalR – [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/aspnet/core/signalr/introduction.
6. Visual Studio Code Documentation. Documentation for Visual Studio Code – [Електронний ресурс]. – Режим доступу: https://code.visualstudio.com/docs.
7. Refactoring.Guru. Design Patterns – [Електронний ресурс]. – Режим доступу: https://refactoring.guru/uk/design-patterns .
8. SQLite Documentation. Official Documentation for SQLite – [Електронний ресурс]. – Режим доступу: https://www.sqlite.org/docs.html.
9. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. – Addison-Wesley Professional, 1994. – [Електронний ресурс]. – Режим доступу: https://www.javier8a.com/itc/bd1/articulo.pdf .

ДОДАТКИ

Додаток А

ЛІСТИНГ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ ПАТЕРНІВ ПРОЄКТУВАННЯ

1) Реалізація патерну Command

ICommand.cs:

```
namespace CiServer.Core.Commands;

public interface ICommand
{
    void Execute();
}
```

CloneRepositoryCommand.cs:

```
using System.Diagnostics;
using CiServer.Core.Entities;

namespace CiServer.Core.Commands;

public class CloneRepositoryCommand : ICommand
{
    private readonly Build _build;
    private readonly string _workingDir;

    public CloneRepositoryCommand(Build build)
    {
        _build = build;
        _workingDir = Path.Combine(Path.GetTempPath(), "CiBuilds",
_build.BuildId.ToString());
    }

    public void Execute()
    {
        Console.WriteLine($"[GIT] Preparing workspace: {_workingDir}");

        if (Directory.Exists(_workingDir))
        {
            DeleteDirectory(_workingDir);
        }

        Directory.CreateDirectory(_workingDir);

        string repoUrl = _build.Project?.RepoUrl ?? throw new Exception("Repo URL
is null");
        Console.WriteLine($"[GIT] Cloning {repoUrl}...");
```

```
        RunProcess("git", $"clone {repoUrl} .", _workingDir);

        Console.WriteLine("[GIT] Repository cloned successfully.");
    }

    private void RunProcess(string fileName, string args, string workDir)
    {
        var startInfo = new ProcessStartInfo
        {
            FileName = fileName,
            Arguments = args,
            WorkingDirectory = workDir,
            RedirectStandardOutput = true,
            RedirectStandardError = true,
            UseShellExecute = false,
            CreateNoWindow = true
        };

        using var process = new Process { StartInfo = startInfo };

        process.OutputDataReceived += (s, e) => { if (e.Data != null)
Console.WriteLine($"   > {e.Data}"); };
        process.ErrorDataReceived += (s, e) => { if (e.Data != null)
Console.WriteLine($"   ! {e.Data}"); };

        process.Start();
        process.BeginOutputReadLine();
        process.BeginErrorReadLine();
        process.WaitForExit();

        if (process.ExitCode != 0)
        {
            throw new Exception($"{fileName} failed with exit code
{process.ExitCode}");
        }
    }

    private void DeleteDirectory(string path)
    {
        foreach (var file in Directory.GetFiles(path))
        {
            File.SetAttributes(file, FileAttributes.Normal);
            File.Delete(file);
        }

        foreach (var dir in Directory.GetDirectories(path))
        {
            DeleteDirectory(dir);
        }
```

```
        Directory.Delete(path, false);
    }
}
```

2) Реалізація патерну Mediator

CiMediator.cs:

```
using CiServer.Core.Entities;
using CiServer.Core.Interfaces;

namespace CiServer.Core.Mediator;

public class CiMediator : IMediator
{
    private readonly IRepository<Build, Guid> _buildRepo;
    private readonly IRepository<BuildLog, Guid> _logRepo;
    private readonly IRealTimeNotifier _notifier;

    public CiMediator(
        IRepository<Build, Guid> buildRepo,
        IRepository<BuildLog, Guid> logRepo,
        IRealTimeNotifier notifier)
    {
        _buildRepo = buildRepo;
        _logRepo = logRepo;
        _notifier = notifier;
    }

    public void Notify(object sender, string ev, object? data = null)
    {
        HandleAsync(ev, data).Wait();
    }

    private async Task HandleAsync(string ev, object? data)
    {
        if (ev == "JobFinished" && data is Build resultBuild)
        {
            var build = await _buildRepo.GetByIdAsync(resultBuild.BuildId);
            if (build != null)
            {
                build.RestoreState();
                bool isSuccess = resultBuild.Status == BuildStatus.Success;
                build.Finish(isSuccess);
                build.EndTime = DateTime.UtcNow;
                await _buildRepo.SaveChangesAsync();
                Console.WriteLine($"[MEDIATOR] Build {build.BuildId} finalized.
Status: {build.Status}");
            }
```

```
            await _notifier.SendStatusAsync(resultBuild.BuildId.ToString(),
resultBuild.Status.ToString());
        }
        else if (ev == "LogReceived" && data is BuildLog log)
        {
            await _logRepo.AddAsync(log);
            await _logRepo.SaveChangesAsync();

            await _notifier.SendLogAsync(
                log.BuildId.ToString(),
                log.Content,
                log.Timestamp.ToString("HH:mm:ss")
            );

            Console.WriteLine($"[MEDIATOR] Log saved & broadcasted:
{log.Content}");
        }
    }
}
```

3) Реалізація патерну Visitor

IVisitor.cs:

```
using CiServer.Core.Entities;

namespace CiServer.Core.Visitor;

public interface IVisitor
{
    void VisitProject(Project project);
    void VisitBuild(Build build);
    void VisitBuildLog(BuildLog log);
    void VisitArtifact(Artifact artifact);
}
```

HtmlReportVisitor.cs:

```
using System.Text;
using CiServer.Core.Entities;

namespace CiServer.Core.Visitor;

public class HtmlReportVisitor : IVisitor
{
    private readonly StringBuilder _sb = new StringBuilder();

    public string GetHtml()
    {
        return _sb.ToString();
    }
```

```
    public void VisitProject(Project project)
    {
        _sb.AppendLine($"<h1>Project Report: {project.Name}</h1>");
        _sb.AppendLine($"<p>Repository: <a
href='{project.RepoUrl}'>{project.RepoUrl}</a></p>");
        _sb.AppendLine("<hr/>");
        _sb.AppendLine("<h3>Build History:</h3>");
        _sb.AppendLine("<ul>");
    }

    public void VisitBuild(Build build)
    {
        string color = build.Status == BuildStatus.Success ? "green" :
                       build.Status == BuildStatus.Pending ? "gray" :
                       build.Status == BuildStatus.Running ? "blue" : "red";
        _sb.AppendLine("<li>");
        _sb.AppendLine($"<strong>Build ID:</strong> {build.BuildId} <br/>");
        _sb.AppendLine($"Status: <span style='color:{color}'>{build.Status}</span>
<br/>");
        _sb.AppendLine($"Time: {build.StartTime}");
        _sb.AppendLine("<div style='margin-left: 20px; font-family: monospace;
background: #f0f0f0; padding: 5px;'>");

        if (build.Artifacts.Any())
        {
            _sb.AppendLine("<br/><strong>Artifacts:</strong><ul>");
        }
    }

    public void VisitBuildLog(BuildLog log)
    {
        _sb.AppendLine($"<span>[{log.Timestamp:HH:mm:ss}]
{log.Content}</span><br/>");
    }

    public void VisitArtifact(Artifact artifact)
    {
        _sb.AppendLine($"<li style='list-style-type: none; margin-bottom:
5px;'>");
        _sb.AppendLine($"   <a href='{artifact.FilePath}' download style='text-
decoration: none; color: blue;'>Download Archive (.zip)</a>");
        _sb.AppendLine($"   <span style='color: grey; font-size:
small;'>({artifact.CreatedAt:HH:mm:ss})</span>");
        _sb.AppendLine("</li>");
    }
}
```

4) Реалізація патерну Repository

RepositoryBase.cs:

```
using CiServer.Core.Interfaces;
using Microsoft.EntityFrameworkCore;

namespace CiServer.Data.Repositories;

public class RepositoryBase<T, TKey> : IRepository<T, TKey> where T : class
{
    protected readonly ApplicationDbContext _context;
    protected readonly DbSet<T> _dbSet;

    public RepositoryBase(ApplicationDbContext context)
    {
        _context = context;
        _dbSet = _context.Set<T>();
    }

    public async Task<T?> GetByIdAsync(TKey id)
    {
        return await _dbSet.FindAsync(id);
    }

    public IQueryable<T> GetAll()
    {
        return _dbSet.AsQueryable();
    }

    public async Task AddAsync(T entity)
    {
        await _dbSet.AddAsync(entity);
    }

    public Task UpdateAsync(T entity)
    {
        _dbSet.Update(entity);
        return Task.CompletedTask;
    }

    public async Task DeleteAsync(TKey id)
    {
        var entity = await _dbSet.FindAsync(id);
        if (entity != null)
        {
            _dbSet.Remove(entity);
        }
    }

    public async Task SaveChangesAsync()
    {
        await _context.SaveChangesAsync();
```

```
        }
    }
```