Вінницький національний технічний університет
(повне найменування вищого навчального закладу)

Факультет інформаційних технологій та комп'ютерної інженерії
(повне найменування інституту, назва факультету (відділення))

Кафедра обчислювальної техніки
(повна назва кафедри (предметної, циклової комісії))

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Вебплатформа для аналізу бібліотек коду з деревовидною структурою залежностей»

08-54.БКР.004.00.000 ПЗ

Виконала: студентка 2 курсу, групи КІ-23мсз
спеціальності 123 — Комп'ютерна інженерія
(шифр і назва напряму підготовки, спеціальності)
_____ Лотоцька А. О.
(підпис)

Керівник: к.т.н., доцент каф. ОТ
_____ Дудник О. В.
(підпис)
«_____» _____ 2025 р.

Рецензент: к.т.н., доцент каф. ПЗ
_____ Романюк О. В.
(підпис)
«_____» _____ 2025 р.

Допущено до захисту
Завідувач кафедри ОТ
д.т.н., проф. Азаров О. Д.
«____» _____ 2025 р.

Вінниця ВНТУ — 2025 рік

ВІННИЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра обчислювальної техніки

Освітньо-кваліфікаційний рівень бакалавр

Галузь знань — 12 «Інформаційні технології»

Спеціальність — 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ

Завідувач кафедри ОТ

д.т.н., проф. Азаров О. Д.

«21» березня 2025 р.

З А В Д А Н Н Я

НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Лотоцькій Анастасії Олександрівні

1 Тема роботи: «Вебплатформа для аналізу бібліотек коду з деревовидною структурою залежностей», керівник роботи к.т.н., доцент каф. ОТ Дудник О. В., затверджена наказом вищого навчального закладу від 20 березня 2025 р. №97.

2 Термін подання студентом роботи 13 травня 2025 р.

3 Вихідні дані до роботи: призначення — аналіз та візуалізація залежностей у бібліотеках коду; основні підтримувані функції — завантаження DLL-файлів, аналіз структури залежностей, інтерактивна візуалізація результатів.

4 Зміст розрахунково-пояснювальної записки: вступ, техніко-економічне обґрунтування проєкту, теоретична частина, практична частина (архітектура вебплатформи, аналізатор залежностей, система візуалізації, тестування).

5 Перелік ілюстративної частини (з точним зазначенням обов'язкових креслень): архітектурна схема вебплатформи, діаграма взаємодії компонентів, схема інтеграції ASP.NET Core з Python Dash.

6 Консультанти розділів роботи приведені в таблиці 1.

Таблиця 1 — Консультанти роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завда прий
1-3	к.т.н., доц. каф. ОТ Дудник О. В.	21.03.25	13.05
Нормоконтроль	ас. каф. ОТ Швець С. І.	13.05.25	30.05

7 Дата видачі завдання 21 березня 2025 р.

8 Календарний план виконання БКР приведений в таблиці 2.

Таблиця 2 — Календарний план виконання БКР

№ етапу	Назва етапу	Термін виконання		Підпис
		Початок	Кінець	
1	Постановка задачі та техніко економічне обґрунтування	21.03.24	23.03.25	
2	Аналіз існуючих інструментів для аналізу залежностей	24.03.25	28.03.25	
3	Аналіз та обґрунтування вибору технологій	29.03.25	09.04.25	
4	Проєктування та розробка архітектури вебплатформи	10.04.25	19.04.25	
5	Реалізація аналізатора залежностей та системи візуалізації	20.04.25	26.04.25	
6	Підготовка тезових матеріалів для публікації	27.04.25	28.04.25	
7	Оформлення пояснювальної записки та презентації	29.04.25	12.05.25	
8	Підготовка супровідної документації та перевірка на плагіат	13.05.25	13.05.25	

Студент Анастасія ЛОТОЦЬКА

Керівник роботи к.т.н., доц. Олександр ДУДНИК

УДК 004.7

Лотоцька А. О. Ве
структурою залежностей.
«Комп'ютерна інженерія»
ВНТУ, 2025. 90 с.

На укр. мові. Бібліо

У даній роботі пров
залежностей бібліотек
управління пакетами та і
розглянуто архітектурні па
Core та особливості робот
Висвітлено методи інте
ефективної візуалізації
Проаналізовано мікросер
масштабованості для вп
особливостей роботи з DL
для завантаження та аналі
залежностей та інтерактив
продуктивності системи
ручного користувацького

Ключові слова: анал
Dash, Cytoscape, візуалізац
API.

АНОТАЦІЯ

УДК 004.7

Лотоцька А. О. Вебплатформа для аналізу бібліотек коду з деревовидною структурою залежностей. Бакалаврська кваліфікаційна робота зі спеціальності 123 «Комп'ютерна інженерія», освітня програма «Комп'ютерна інженерія». Вінниця: ВНТУ, 2025. 90 с.

На укр. мові. Бібліогр.: 34 назв, рис.: 20, табл. 10.

У даній роботі проведено комплексний аналіз сучасних технологій аналізу залежностей бібліотек коду, зокрема засобів статичного аналізу, систем управління пакетами та інструментів візуалізації графових структур. Детально розглянуто архітектурні патерни для розробки вебплатформ, технології ASP.NET Core та особливості роботи з метаданими .NET збірок за допомогою рефлексії. Висвітлено методи інтеграції Python з .NET додатками для забезпечення ефективної візуалізації даних з використанням бібліотеки Dash Cytoscape. Проаналізовано мікросервісну архітектуру з точки зору гнучкості та масштабованості для впровадження у проєкті. На основі аналізу вимог та особливостей роботи з DLL файлами розроблено вебплатформу, що включає API для завантаження та аналізу бібліотек, модуль побудови деревовидної структури залежностей та інтерактивний візуалізатор. Особлива увага приділена оптимізації продуктивності системи при роботі з великими бібліотеками та забезпеченню зручного користувацького інтерфейсу для аналізу складних залежностей.

Ключові слова: аналіз залежностей, .NET, ASP.NET Core, рефлексія, Python Dash, Cytoscape, візуалізація графів, DLL аналіз, мікросервісна архітектура, REST API.

ANNOTATION

UDC 004.7

Lototska A. O. Web platform for code library analysis with a tree-like dependency structure. Bachelor`s qualification work in specialty 123 "Computer Engineering", educational program "Computer Engineering". Vinnytsia: VNTU, 2025. 90 p.

In Ukrainian. Bibliography: 34 titles, figures: 20, tables: 10.

This thesis presents a comprehensive analysis of modern technologies for analyzing code library dependencies, including static analysis tools, package management systems, and graph structure visualization tools. The architectural patterns for web platform development, ASP.NET Core technologies, and features of working with .NET assembly metadata using reflection are examined in detail. Methods of integrating Python with .NET applications for effective data visualization using the Dash Cytoscape library are highlighted. Microservice architecture is analyzed in terms of flexibility and scalability for implementation in the project. Based on the analysis of requirements and specifics of working with DLL files, a web platform has been developed that includes an API for uploading and analyzing libraries, a module for building a tree-like dependency structure, and an interactive visualizer. Special attention is paid to optimizing system performance when working with large libraries and ensuring a user-friendly interface for analyzing complex dependencies.

Keywords: dependency analysis, .NET, ASP.NET Core, reflection, Python Dash, Cytoscape, graph visualization, DLL analysis, microservice architecture, REST API.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

API — Application Programming Interface

ASP.NET Core — ASP.NET Core Framework

CI/CD — Continuous Integration/Continuous Deployment

CSS — Cascading Style Sheets

DLL — Dynamic Link Library

HTML — HyperText Markup Language

HTTP — HyperText Transfer Protocol

JS — JavaScript

JSON — JavaScript Object Notation

MVC — Model-View-Controller

MVP — Model-View-Presenter

MVVM — Model-View-ViewModel

NuGet — Package Manager for .NET

REST — Representational State Transfer

SDK — Software Development Kit

SPA — Single Page Application

SQL — Structured Query Language

SSIS — SQL Server Integration Services

UI — User Interface

UX — User Experience

XML — eXtensible Markup Language

ЗМІСТ

ВСТУП

Сучасний розвиток програмного забезпечення характеризується зростаючою складністю та модульністю систем, де кожен компонент залежить від множини інших бібліотек і модулів. Аналіз та візуалізація цих залежностей стає критичною задачею для розробників, архітекторів програмного забезпечення та спеціалістів з інформаційної безпеки. Ця тема є надзвичайно актуальною для фахівців з розробки програмного забезпечення, оскільки розуміння структури залежностей дозволяє виявляти потенційні проблеми, оптимізувати архітектуру та підвищувати безпеку додатків.

Актуальність розробки полягає у необхідності впровадження ефективних інструментів аналізу залежностей бібліотек коду через зростання складності сучасних програмних систем та розширення екосистеми .NET. Розробка спеціалізованої вебплатформи дасть можливість автоматизувати процес виявлення та візуалізації залежностей, і, як наслідок, підвищити якість розробки програмного забезпечення, спростити процес рефакторингу та мінімізувати ризики при оновленні компонентів.

Мета бакалаврської кваліфікаційної роботи полягає у створенні інтегрованої вебплатформи для аналізу бібліотек коду з візуалізацією деревовидної структури залежностей, що забезпечує високий рівень інтерактивності та зручності використання, через проєктування мікросервісної архітектури з використанням ASP.NET Core та Swagger, реалізацію механізмів рефлексії для аналізу DLL файлів, розробку інтерактивної візуалізації на основі Python Dash Cytoscape, а також організацію ефективного обміну даними між компонентами системи.

Необхідні **завдання** для вирішення поставленої мети БКР:

— провести аналіз існуючих інструментів та технологій для аналізу залежностей бібліотек коду;

— визначити оптимальну архітектуру вебплатформи з урахуванням вимог до масштабованості та гнучкості;

— обґрунтувати вибір технологій для аналізу .NET збірок та візуалізації графових структур;

— спроєктувати та реалізувати API для завантаження та аналізу DLL файлів;

— розробити систему візуалізації деревовидної структури залежностей;

— провести експериментальне тестування розробленого рішення та оцінити його ефективність при роботі з реальними проєктами.

Об'єкт дослідження — це процес аналізу та візуалізації залежностей у бібліотеках коду .NET.

Предметом дослідження є технології аналізу метаданих DLL файлів з використанням рефлексії, методи побудови деревовидних структур залежностей та засоби інтерактивної візуалізації графових даних.

Наукова новизна полягає у поєднанні технологій .NET рефлексії, мікросервісної архітектури та сучасних засобів інтерактивної візуалізації графів для створення цілісної вебплатформи, що дозволяє в реальному часі аналізувати та відображати деревовидні структури залежностей. Розроблений підхід демонструє новий рівень інтеграції між засобами аналізу програмного коду та інструментами візуалізації, що розширює можливості дослідження архітектури ПЗ.

Практична цінність роботи полягає у створенні інструменту, який може бути застосований у реальних умовах під час розробки та супроводу .NET-проєктів. Запропонована вебплатформа спрощує процес виявлення залежностей, підвищує прозорість архітектури програмного забезпечення та знижує ризики, пов’язані з оновленням або видаленням компонентів, що особливо актуально для великих і довготривалих проєктів.

1 АНАЛІТИЧНЕ ОБҐРУНТУВАННЯ РОЗРОБКИ СИСТЕМИ

1.1 Постановка мети та завдань дослідження

Мета БКР полягає у розробці вебплатформи для автоматизованого аналізу бібліотек коду з візуалізацією деревовидної структури залежностей, орієнтованої на середовище .NET. Головна цінність проєкту — надання розробникам програмного забезпечення, архітекторам та спеціалістам з безпеки зручного інструменту для глибокого аналізу та розуміння взаємозв'язків між компонентами програмних систем.

Сучасна розробка програмного забезпечення нерозривно пов'язана з використанням сторонніх бібліотек та фреймворків, що створює складні ланцюги залежностей. Розуміння цих залежностей є критично важливим для:

— пошуку та усунення потенційних конфліктів між версіями бібліотек;

— виявлення невикористовуваних компонентів, що можуть бути видалені;

— ідентифікації застарілих або вразливих бібліотек, що потребують оновлення;

— оптимізації структури проєкту шляхом реорганізації компонентів;

— прийняття архітектурних рішень при розробці нових систем;

— оцінки складності та ризиків при внесенні змін до існуючих систем;

Для реалізації поставленої мети необхідно вирішити наступні задачі:

Створення бекенд-сервісу на базі ASP.NET Core для:

— завантаження та аналізу DLL файлів;

— вилучення метаданих про залежності з використанням механізмів рефлексії;

— побудови деревовидної структури залежностей;

— надання API для взаємодії з фронтенд-частиною системи;

— документування API за допомогою Swagger.

Розробка модуля візуалізації на Python з використанням Dash Cytoscape для:

— інтерактивного відображення графу залежностей;

— забезпечення можливості навігації та масштабування;

— виділення та фільтрації компонентів графу;

— відображення детальної інформації про обрані елементи.

Інтеграція компонентів системи:

— організація взаємодії між ASP.NET Core та Python модулем;

— реалізація механізму обміну даними через JSON-формат;

— забезпечення асинхронного оновлення візуалізації при зміні даних.

Створення зручного користувацького інтерфейсу:

— розробка інтуїтивно зрозумілого веб-інтерфейсу;

— імплементація функціоналу пошуку та фільтрації;

— забезпечення можливості експорту результатів аналізу.

Для успішної реалізації проєкту важливо забезпечити:

— точність аналізу залежностей різних типів .NET збірок;

— швидкодію системи при роботі з великими бібліотеками;

— інтуїтивно зрозумілу візуалізацію складних графових структур;

— надійність взаємодії між компонентами системи.

Розроблювана вебплатформа повинна підтримувати різні типи аналізу:

— базовий аналіз прямих залежностей;

— глибокий аналіз транзитивних залежностей;

— порівняльний аналіз різних версій бібліотек;

— виявлення циклічних залежностей;

— аналіз залежностей на рівні просторів імен та класів.

Кінцевий продукт має бути доступний через веб-інтерфейс без необхідності встановлення додаткового програмного забезпечення, що дозволить інтегрувати його в існуючі процеси розробки та забезпечить зручність використання для широкого кола спеціалістів.

1.2 Роль та значення аналізу залежностей у розробці ПЗ

Аналіз залежностей бібліотек коду набуває критичного значення в сучасній розробці програмного забезпечення з декількох фундаментальних причин.

По-перше, сучасні програмні системи характеризуються безпрецедентним

рівнем складності та модульності. Згідно з дослідженнями, типовий .NET-проєкт середнього розміру може містити понад 100 сторонніх залежностей, які в свою чергу мають власні залежності, утворюючи складну деревовидну структуру. Без спеціалізованих інструментів аналізу розробникам практично неможливо отримати цілісне уявлення про цю структуру, що суттєво ускладнює підтримку та розвиток таких систем.

По-друге, інформаційна безпека програмного забезпечення стала одним з найбільш пріоритетних питань в галузі. За даними OWASP (Open Web Application Security Project), використання компонентів з відомими вразливостями входить до десятки найбільш критичних ризиків безпеки. У 2023-2024 роках було зафіксовано зростання на 42% атак, пов'язаних з експлуатацією вразливостей у бібліотеках з відкритим кодом. Детальний аналіз залежностей дозволяє виявляти потенційно вразливі компоненти та своєчасно реагувати на загрози.

По-третє, проблема «dependency hell» (пекло залежностей) — ситуація, коли різні компоненти системи вимагають несумісних версій однієї і тієї ж бібліотеки — залишається актуальною навіть в екосистемі .NET з її розвиненими механізмами управління пакетами. За статистикою, до 30% критичних збоїв в продакшн-середовищах пов'язані саме з конфліктами залежностей. Інструменти візуалізації залежностей дозволяють наочно виявляти такі конфлікти на ранніх етапах розробки.

Четверта важлива причина — оптимізація розміру додатків та ефективності використання ресурсів. У контексті розвитку мобільних додатків, веб-систем та IoT-пристроїв розмір розгорнутого додатку має суттєвий вплив на його продуктивність та зручність використання. Аналіз залежностей дозволяє виявляти непотрібні або дубльовані компоненти, що можуть бути видалені без втрати функціоналу.

П'ята причина — це відповідність нормативним вимогам. У багатьох галузях (фінанси, охорона здоров'я, державний сектор) існують суворі нормативні вимоги щодо використання стороннього програмного забезпечення. Організації зобов'язані підтримувати актуальний перелік всіх компонентів, що

використовуються в їхніх системах, включаючи непрямі залежності. Автоматизований аналіз залежностей значно спрощує цей процес.

Шоста причина — це підвищення ефективності процесу розробки. Згідно з дослідженнями McKinsey, розробники витрачають до 30% свого часу на вирішення проблем, пов'язаних з залежностями. Наявність ефективних інструментів аналізу може значно скоротити ці витрати часу.

Порівняння основних проблем, пов'язаних з управлінням залежностями, та їх впливу на процес розробки програмного забезпечення наведено в таблиці 1.1.

Таблиця 1.1 — Проблеми управління залежностями та їх вплив на розробку

Проблема	Опис	Вплив на процес розробки	Переваги використання інструментів аналізу
Складність системи	Сучасні програмні системи містять сотні взаємопов'язаних компонентів	Ускладнюється розуміння структури проєкту, зростає час на введення нових розробників	Наочне представлення структури, прискорення адаптації нових членів команди
Безпека компонентів	Використання залежностей з відомими вразливостями	Підвищення ризиків безпеки, необхідність екстрених оновлень	Раннє виявлення вразливих компонентів, планове оновлення
Конфлікти версій	Несумісність різних версій однієї бібліотеки	Непередбачувана поведінка програми, складність діагностики	Візуалізація конфліктів, спрощення їх виявлення та вирішення
Ефективність використання ресурсів	Надмірна кількість залежностей збільшує розмір додатку	Зниження продуктивності, збільшення часу завантаження	Виявлення непотрібних залежностей, оптимізація розміру

Специфіка платформи .NET додає додаткових аспектів актуальності. Розвиток .NET 5/6/7/8 та перехід з традиційного .NET Framework на сучасні версії .NET вимагає ретельного аналізу сумісності бібліотек та їх залежностей. Сучасні .NET проєкти часто орієнтовані на різні платформи одночасно (multi-targeting), що створює додаткову складність в управлінні залежностями для кожної цільової платформи. Поширення мікросервісної архітектури вимагає чіткого розуміння залежностей між компонентами для правильного проєктування границь сервісів. Автоматизація процесів збірки та розгортання вимагає автоматизованих інструментів для виявлення проблем з залежностями на ранніх етапах.

Існуючі інструменти для аналізу залежностей в .NET мають суттєві обмеження. NuGet Package Explorer не забезпечує глибокий аналіз транзитивних залежностей. Visual Studio має базовий аналізатор залежностей, який не пропонує розширених можливостей візуалізації. JetBrains ReSharper пропонує аналіз залежностей, але вимагає придбання ліцензії. Більшість існуючих інструментів не мають веб-інтерфейсу та не інтегруються з CI/CD системами.

Розробка спеціалізованої вебплатформи для аналізу залежностей з акцентом на інтерактивну візуалізацію дозволить заповнити цю нішу та забезпечити розробників ефективним інструментом для роботи зі складними програмними системами.

1.3 Аналіз наявних інструментів для роботи із бібліотеками коду

На сучасному ринку існує декілька інструментів для аналізу залежностей бібліотек коду в екосистемі .NET. Проведемо детальний аналіз їх функціональних можливостей, особливостей та обмежень для обґрунтування необхідності розробки власної вебплатформи.

NuGet Package Explorer є базовим інструментом для роботи з NuGet-пакетами та їх залежностями. Він дозволяє переглядати структуру пакетів, включаючи метадані, файли та безпосередні залежності. Однак цей інструмент не забезпечує глибокого аналізу транзитивних залежностей та не візуалізує їх у вигляді графа. Користувач бачить лише плоский список прямих залежностей без можливості

дослідження глибших рівнів. Відсутність веб-інтерфейсу та необхідність встановлення додатку обмежують доступність цього інструменту для команд розробників.

DotNet Dependencies є консольним інструментом, який дозволяє аналізувати залежності .NET проєктів. Він забезпечує вивід залежностей у текстовому форматі з можливістю експорту у JSON або CSV. Перевагою цього інструменту є простота використання та інтеграція з CI/CD системами. Однак він не надає можливостей візуалізації та має обмежені можливості для інтерактивного аналізу залежностей. Вихідні дані представлені у форматі, який складно аналізувати без додаткової обробки.

Assembly Binding Log Viewer (Fusion Log Viewer) є офіційним інструментом від Microsoft для аналізу процесу завантаження збірок у .NET Framework. Він дозволяє діагностувати проблеми із завантаженням збірок, відстежувати шляхи пошуку та версії залежностей. Проте цей інструмент фокусується на діагностиці проблем, а не на комплексному аналізі структури залежностей. Він також не має можливостей візуалізації та обмежений лише платформою Windows.

ILSpy з плагіном Analyze є інструментом декомпіляції .NET збірок, який також дозволяє аналізувати залежності на рівні типів та методів. Він забезпечує візуалізацію залежностей у вигляді дерева, але має обмежені можливості для роботи з великими проєктами та складними структурами залежностей. Відсутність веб-інтерфейсу та фокус на декомпіляції, а не на аналізі залежностей, обмежують його застосування для вирішення поставлених задач.

NDepend пропонує потужний функціонал для аналізу коду та залежностей у проєктах .NET. Інструмент включає матрицю залежностей, графи та діаграми для візуалізації складних структур. NDepend дозволяє аналізувати залежності на різних рівнях: збірки, простори імен, типи, методи. Проте висока вартість ліцензії (від $399 за розробника) та складний інтерфейс з високою кривою навчання обмежують його доступність для малих та середніх команд розробників.

Для систематизації порівняння інструментів складено таблицю 1.2, яка відображає їх ключові характеристики.

Таблиця 1.2 — Порівняння існуючих інструментів для аналізу залежностей бібліотек коду

Характеристика	NuGet Package Explorer	DotNet Dependencies	Assembly Binding Log Viewer	ILSpy з Analyze	NDepend	JetBrains dotPeek	Розроблювана вебплатформа
Аналіз DLL без вихідного коду	Так (обмежено)	Ні	Так	Так	Так	Так	Так
Глибина аналізу залежностей	Прямі залежності	Транзитивні (плоский список)	Лише завантажені	На рівні типів	Глибокий аналіз	На рівні типів	Комплексний аналіз
Візуалізація структури залежностей	Ні	Ні	Ні	Обмежена (дерево)	Розширена	Обмежена	Інтерактивна
Формат представлення	Дерево файлів	Текст/JSON/CSV	Текстовий лог	Дерево	Діаграми/Матриці	Дерево	Інтерактивний граф
Інтерфейс користувача	Десктоп	Консоль	Десктоп	Десктоп	Десктоп/VS	Десктоп	Веб
Інтеграція з CI/CD	Ні	Так	Ні	Ні	Обмежена	Ні	Так
Експорт результатів	Ні	JSON/CSV	Текст	Ні	Розширений	Ні	JSON/CSV/Image
Аналіз конфліктів версій	Обмежений	Обмежений	Так	Ні	Так	Обмежений	Так
Фільтрація та пошук у залежностях	Базова	Ні	Ні	Базова	Розширена	Базова	Розширена
Платформа	Windows	Крос-платформна	Windows	Крос-платформна	Windows	Windows	Крос-платформна
Доступність	Безкоштовно	Безкоштовно	Безкоштовно	Безкоштовно	Комерційна	Безкоштовно	Безкоштовно

JetBrains dotPeek з функцією аналізу збірок поєднує можливості декомпіляції та аналізу залежностей. Інструмент дозволяє досліджувати вміст збірок, їх структуру та взаємозв'язки. Однак, як і ILSpy, він орієнтований більше на

декомпіляцію, ніж на комплексний аналіз залежностей. Візуалізація в цьому інструменті обмежена та не пристосована для роботи з великими структурами залежностей.

Розроблювана вебплатформа враховує ці обмеження та спрямована на створення інтегрованого рішення, яке забезпечить:

— доступність через веб-інтерфейс без необхідності встановлення додаткового програмного забезпечення;

— глибокий аналіз структури залежностей DLL файлів з використанням механізмів рефлексії .NET;

— інтерактивну візуалізацію деревовидної структури залежностей з використанням сучасних технологій візуалізації графів;

— можливості для фільтрації, пошуку та навігації у великих структурах залежностей;

— інтеграцію з CI/CD системами для автоматизації процесу аналізу залежностей.

1.4 Обґрунтування ефективності розробки власного інструменту

Розробка власної веб-платформи для аналізу залежностей бібліотек коду має чітке економічне обґрунтування, що базується на кількох ключових факторах.

По-перше, використання існуючих комерційних рішень для аналізу залежностей вимагає певних фінансових витрат. Вартість ліцензій на провідні інструменти, такі як NDepend (близько 14000 грн за розробника) або JetBrains ReSharper з Architecture Tools (близько 10000 грн за рік на розробника), для невеликої команди з 5 розробників становитиме від 50000 до 70000 грн щорічно. Розробка власного рішення, хоча і потребує початкових інвестицій, в довгостроковій перспективі виявляється більш економічно вигідною.

По-друге, інтеграція аналізу залежностей в існуючі процеси розробки дозволяє скоротити витрати, пов'язані з виявленням та усуненням проблем на пізніх етапах розробки. За даними галузевих досліджень, вартість виправлення дефекту на етапі експлуатації значно перевищує вартість його усунення на етапі

проєктування. Раннє виявлення проблем з залежностями дозволяє заощадити до 15% загальних витрат на розробку та підтримку програмного забезпечення.

По-третє, підвищення продуктивності команди розробників завдяки зменшенню часу на діагностику та вирішення проблем з залежностями приносить додаткову економічну вигоду. При середній зарплаті розробника економія навіть 3-5% робочого часу для команди з 5 розробників дає помітну економію в довгостроковій перспективі. У таблиці 1.3 наведені економічні показники, що необхідні для розробки власного рішення.

Таблиця 1.3 — Економічні показники розробки власного рішення

Показник	Значення
Орієнтовна вартість розробки власного рішення	50 000 грн
Щорічні витрати на підтримку власного рішення	5 000 грн
Вартість ліцензій комерційних аналогів (1 розробник)	10 000 грн / рік
Економія від раннього виявлення проблем	20 000 — 30 000 грн / рік
Підвищення продуктивності	5 000 грн / рік
Орієнтовний строк окупності власного рішення	10—12 місяців

Додатковою перевагою власного рішення є можливість його адаптації під специфічні потреби команди та інтеграція з існуючими інструментами розробки. Такі можливості часто відсутні у комерційних рішеннях або вимагають додаткових витрат на кастомізацію.

Таким чином, розробка власної вебплатформи для аналізу залежностей бібліотек коду є економічно обґрунтованим рішенням, що забезпечує економію ресурсів у довгостроковій перспективі та підвищує ефективність процесу розробки програмного забезпечення.

1.5 Напрямки подальшого розвитку системи

Розроблювана вебплатформа для аналізу залежностей бібліотек коду має значний потенціал для подальшого розвитку та розширення функціональних можливостей. Перспективні напрямки розвитку проєкту можна визначити,

виходячи з актуальних тенденцій в галузі розробки програмного забезпечення та потреб користувачів.

Інтеграція з системами контролю версій, такими як GitHub, GitLab або Azure DevOps, дозволить автоматично аналізувати структуру залежностей безпосередньо з репозиторіїв коду. Це забезпечить безперервний моніторинг стану проєктів та своєчасне виявлення потенційних проблем. Користувачі зможуть налаштувати автоматичні перевірки при створенні запитів на злиття (pull requests) для контролю якості коду.

Впровадження аналізу безпеки залежностей стане важливим розширенням функціоналу. Система зможе ідентифікувати відомі вразливості в бібліотеках, використовуючи публічні бази даних вразливостей, та надавати рекомендації щодо оновлення компонентів. Це підвищить загальний рівень безпеки розроблюваних додатків та зменшить ризики використання потенційно небезпечного коду.

Розширення можливостей аналізу на рівень класів та методів дозволить проводити більш детальний аналіз структури програм. Користувачі зможуть досліджувати не лише залежності між збірками, але й між конкретними типами та методами, що значно підвищить ефективність рефакторингу та оптимізації коду.

Додавання підтримки аналізу проєктів різних мов програмування та платформ дозволить розширити цільову аудиторію інструменту. Крім .NET, система зможе аналізувати залежності в проєктах Java, Python, JavaScript та інших популярних мов, що зробить її універсальним рішенням для різнорідних команд розробки.

Впровадження механізмів рекомендацій щодо оптимізації структури залежностей перетворить платформу з інструменту аналізу в інтелектуального помічника, який не лише виявляє проблеми, але й пропонує шляхи їх вирішення. Система зможе рекомендувати оптимальні версії бібліотек, виявляти непотрібні залежності та пропонувати альтернативні рішення.

Розробка API для інтеграції з іншими інструментами розробки розширить екосистему платформи. Сторонні розробники зможуть використовувати

функціонал аналізу залежностей у власних продуктах, створюючи спеціалізовані рішення для конкретних сценаріїв використання (таблиця 1.4).

Таблиця 1.4 — Перспективні напрямки розширення функціоналу

Напрямок розширення	Опис	Пріоритет
Інтеграція з системами контролю версій	Автоматичний аналіз залежностей з репозиторіїв, перевірки в CI/CD	Високий
Аналіз безпеки залежностей	Виявлення відомих вразливостей, рекомендації щодо оновлення	Високий
Детальний аналіз на рівні класів	Дослідження залежностей між типами та методами	Середній
Підтримка різних мов програмування	Розширення за межі .NET на Java, Python, JavaScript тощо	Середній
Рекомендації щодо оптимізації	Інтелектуальні підказки для покращення структури залежностей	Низький
Відкрите API для інтеграцій	Можливість використання функціоналу в сторонніх продуктах	Низький

Впровадження цих розширень буде відбуватися поетапно, відповідно до пріоритетів та відгуків користувачів. На першому етапі планується реалізувати базовий функціонал аналізу залежностей DLL файлів та їх візуалізації. Другий етап передбачає впровадження інтеграцій з системами контролю версій та аналіз безпеки залежностей. Подальші етапи будуть спрямовані на розширення можливостей аналізу та підтримку додаткових платформ.

Модульна архітектура вебплатформи забезпечить гнучкість при впровадженні нових функцій та адаптацію до мінливих потреб користувачів. Використання сучасних технологій, таких як ASP.NET Core для бекенду та Python Dash для візуалізації, створює надійну основу для майбутнього розвитку системи.

Таким чином, розроблювана вебплатформа має значний потенціал для розширення функціоналу, що забезпечить її актуальність та конкурентоспроможність у довгостроковій перспективі.

2 ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ ВЕБПЛАТФОРМИ ДЛЯ АНАЛІЗУ БІБЛІОТЕК КОДУ

2.1 Архітектурні рішення для вебплатформ аналізу коду

2.1.1 Принципи побудови та переваги мікросервісної архітектури

Мікросервісна архітектура є сучасним підходом до проєктування програмного забезпечення, який передбачає розбиття складної системи на набір невеликих, незалежних сервісів, кожен з яких відповідає за конкретну функціональність. На відміну від монолітної архітектури, де вся функціональність зосереджена в єдиному блоці, мікросервіси дозволяють створювати більш гнучкі та масштабовані системи.

Основними принципами мікросервісної архітектури є єдина відповідальність, незалежність розгортання, ізоляція відмов, технологічна різноманітність та комунікація через API. Принцип єдиної відповідальності означає, що кожен мікросервіс відповідає за виконання однієї чітко визначеної функції або бізнес-задачі. Це дозволяє зосередитися на розробці та оптимізації конкретної частини системи без необхідності розуміння всієї кодової бази. Незалежність розгортання забезпечує можливість оновлювати частини системи без переривання роботи всього додатку, що значно зменшує ризики при внесенні змін та прискорює процес доставки нових версій [1].

Принцип ізоляції відмов гарантує, що проблеми в одному мікросервісі не призводять до відмови всієї системи. Якщо один сервіс виходить з ладу, інші продовжують функціонувати, забезпечуючи часткову працездатність системи. Технологічна різноманітність дозволяє реалізувати різні мікросервіси з використанням технологій та мов програмування, які найкраще підходять для вирішення конкретних задач. Наприклад, один сервіс може бути написаний на C# з використанням ASP.NET Core, а інший — на Python для задач аналізу даних та візуалізації [2].

Взаємодія між мікросервісами відбувається через чітко визначені API, найчастіше використовуючи HTTP/REST або повідомлення (messaging). Це

забезпечує слабкий зв'язок між компонентами системи та спрощує їх інтеграцію. На рисунку 2.1 наведене порівняння монолітної і мікросервісної архітектури.

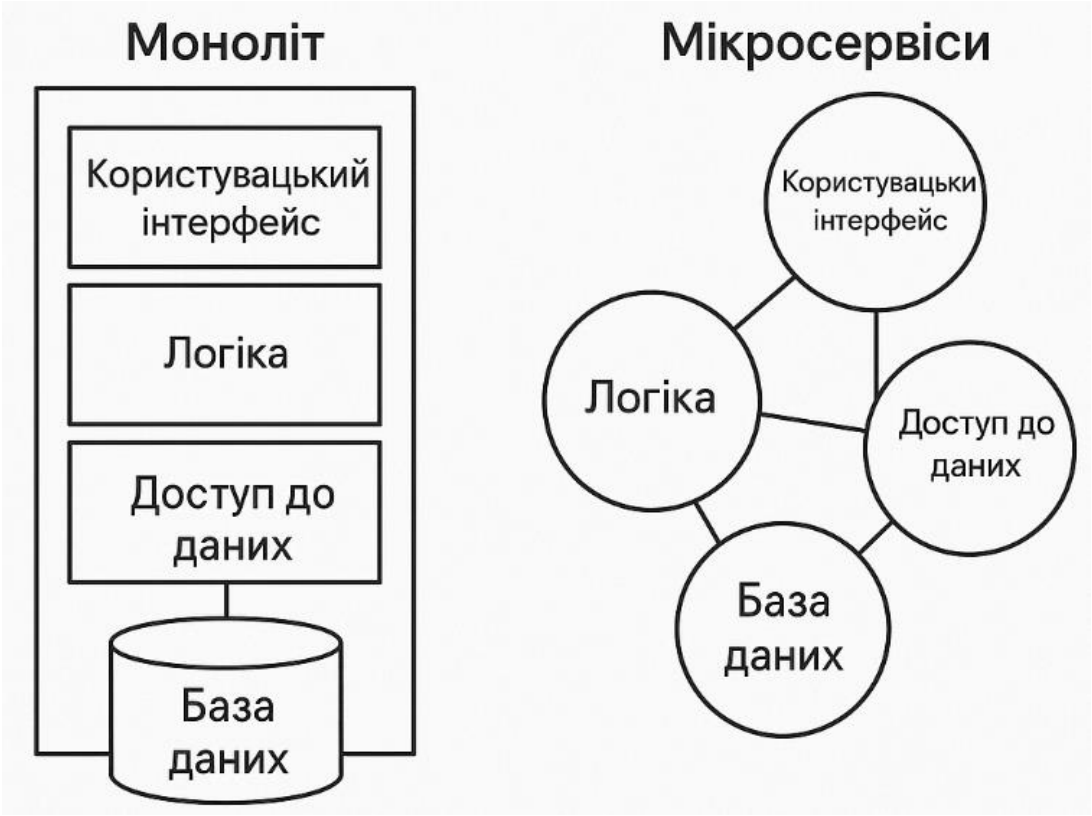

Рисунок 2.1 — Порівняння монолітної та мікросервісної архітектури

Мікросервісна архітектура надає ряд переваг для розробки вебплатформ. Масштабованість дозволяє мікросервісам зростати незалежно один від одного відповідно до навантаження, що оптимізує використання ресурсів та забезпечує високу продуктивність системи. Гнучкість розробки проявляється в тому, що різні команди можуть працювати над різними мікросервісами паралельно, використовуючи найбільш підходящі технології та методології, прискорюючи процес розробки та впровадження нових функцій.

У контексті розроблюваної вебплатформи для аналізу залежностей бібліотек коду, мікросервісна архітектура дозволяє природним чином розділити систему на логічні компоненти. Сервіс аналізу DLL-файлів (ASP.NET Core) відповідає за завантаження, аналіз та вилучення метаданих з бібліотек коду. Сервіс побудови графу залежностей перетворює метадані в структуру графу для подальшої

візуалізації. Сервіс візуалізації (Python Dash) забезпечує інтерактивне відображення графу залежностей. API-шлюз служить єдиною точкою входу для клієнтів, що маршрутизує запити до відповідних сервісів. Сервіс автентифікації та авторизації управляє доступом до системи.

На рисунку 2.2 наведена структура мікросервісної архітектури вебплатформи для аналізу залежностей.

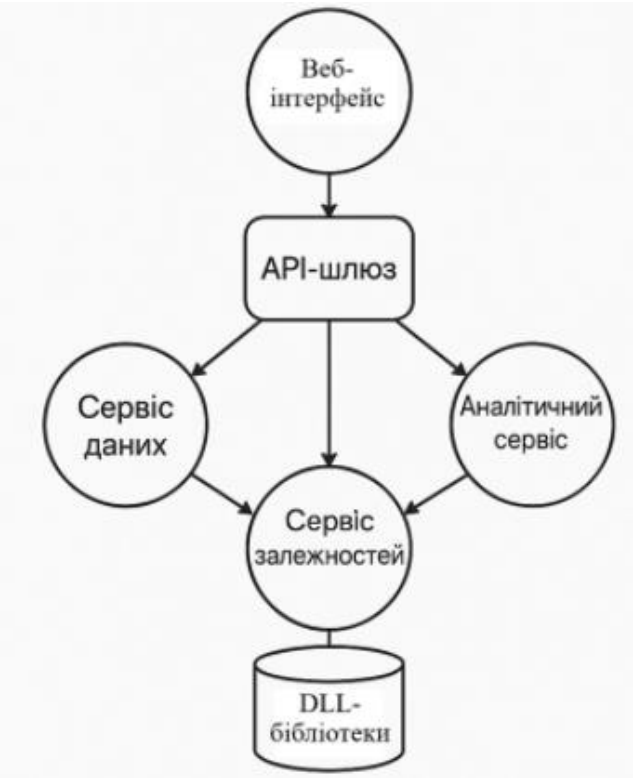

Рисунок 2.2 — Структура мікросервісної архітектури вебплатформи для аналізу залежностей

Для реалізації мікросервісної архітектури в розроблюваній вебплатформі будуть використані сучасні технології та підходи. ASP.NET Core застосовується для створення REST API та основних сервісів аналізу. Python Dash використовується для реалізації компонента візуалізації [3].

Впровадження мікросервісної архітектури для вебплатформи аналізу залежностей дозволить забезпечити необхідну гнучкість та масштабованість системи, оптимально використовувати різні технології для різних компонентів та спростити процес розробки та підтримки програмного забезпечення.

Таблиця 2.1 ілюструє порівняння монолітної та мікросервісної архітектури в контексті розробки вебплатформи для аналізу залежностей.

Таблиця 2.1 — Порівняння монолітної та мікросервісної архітектури

Характеристика	Монолітна архітектура	Мікросервісна архітектура
Складність розробки	Низька на початку, висока при зростанні системи	Висока на початку, знижується при зростанні
Масштабованість	Вертикальна (потужніше обладнання)	Горизонтальна (більше екземплярів сервісів)
Стійкість до відмов	Низька (відмова одного компонента впливає на всю систему)	Висока (ізоляція відмов)
Незалежність технологій	Обмежена (єдиний стек)	Висока (можливість використання різних технологій)
Швидкість розгортання	Низька (вся система одночасно)	Висока (незалежне розгортання сервісів)
Складність інтеграції	Низька (виклики методів в рамках одного процесу)	Висока (мережева комунікація між сервісами)
Ефективність використання ресурсів	Низька (вся система масштабується як одне ціле)	Висока (точкове масштабування завантажених сервісів)

2.1.2 Технологія REST API та інструменти документування Swagger

Representational State Transfer (REST) є архітектурним стилем для розробки веб-сервісів, який став домінуючим підходом у створенні прикладних програмних інтерфейсів (API) для вебплатформ. REST API базується на ряді принципів та обмежень, які забезпечують масштабованість, надійність та продуктивність веб-сервісів.

Основою REST є концепція ресурсів, які ідентифікуються через унікальні URL і можуть мати різні представлення (JSON, XML тощо). REST API використовує стандартні HTTP-методи (GET, POST, PUT, DELETE) для операцій над ресурсами, що відповідають операціям створення, читання, оновлення та

видалення даних (CRUD). Такий підхід забезпечує інтуїтивно зрозумілий інтерфейс та спрощує інтеграцію різних компонентів системи [4].

Важливою характеристикою RESTful сервісів є відсутність стану (statelessness), що означає відсутність збереження стану сесії на сервері між запитами. Кожен запит від клієнта до сервера повинен містити всю необхідну інформацію для його обробки, включаючи дані автентифікації та параметри контексту. Це значно спрощує масштабування системи, оскільки запити можуть оброблятися будь-яким доступним сервером без необхідності синхронізації стану між ними.

У розроблюваній вебплатформі для аналізу залежностей бібліотек коду REST API виконує кілька ключових функцій:

— забезпечує взаємодію між клієнтським веб-інтерфейсом та серверними компонентами;

— надає точки доступу для завантаження та аналізу DLL-файлів;

— забезпечує отримання даних про структуру залежностей для візуалізації;

— дозволяє управляти історією аналізів та налаштуваннями користувача.

Для ефективного використання API необхідна чітка та зрозуміла документація, яка описує доступні ресурси, методи роботи з ними, формати даних та можливі коди відповідей. Swagger (OpenAPI) є стандартом та набором інструментів для документування REST API, який дозволяє створювати інтерактивну документацію, що значно спрощує роботу розробників [5].

Swagger надає єдиний формат опису API, який може бути використаний для автоматичної генерації клієнтського коду, тестування та візуалізації документації. Swagger UI — це веб-інтерфейс, який дозволяє переглядати документацію та взаємодіяти з API безпосередньо з браузера, що суттєво прискорює процес розробки та тестування.

В ASP.NET Core інтеграція Swagger здійснюється за допомогою бібліотеки Swashbuckle, яка автоматично генерує специфікацію OpenAPI на основі контролерів та моделей даних. Це дозволяє підтримувати документацію в актуальному стані без необхідності створення та оновлення окремих документів.

На рисунку 2.3 наведена схема взаємодії компонентів системи через REST API.

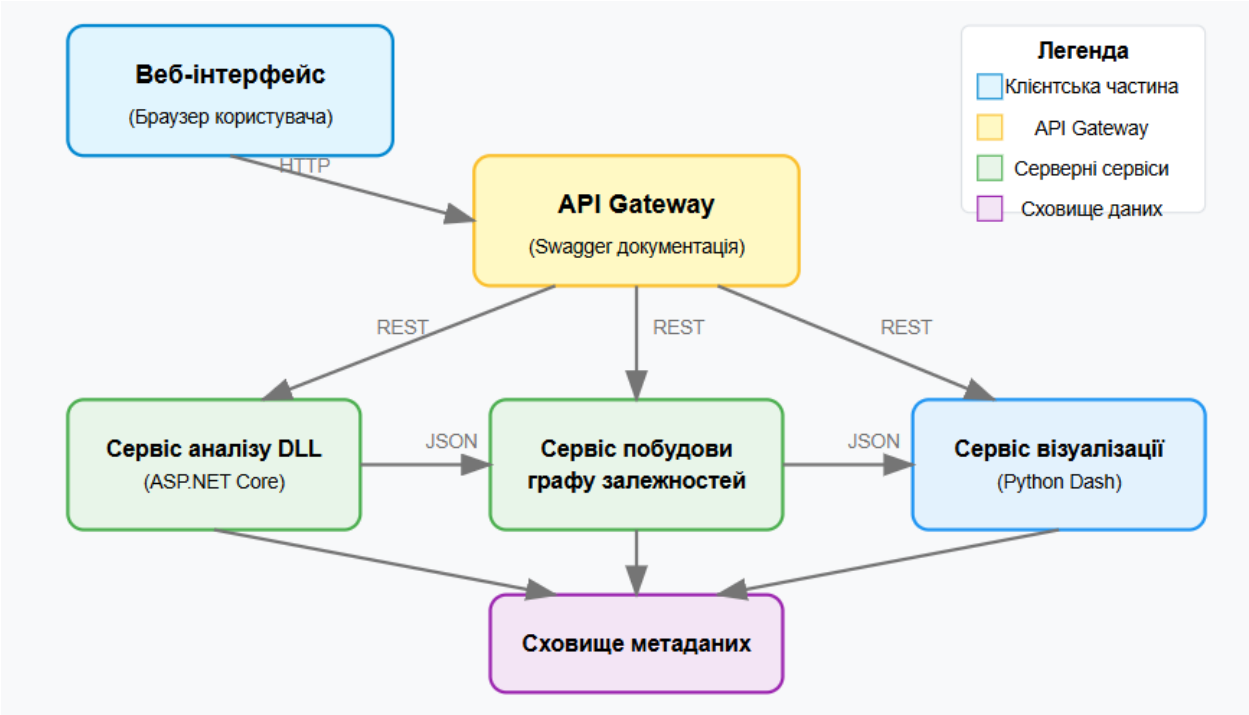

Рисунок 2.3 — Схема взаємодії компонентів системи через REST API

Процес документування API за допомогою Swagger в ASP.NET Core включає кілька етапів:

— встановлення пакетів Swashbuckle.AspNetCore через NuGet;

— налаштування сервісів Swagger в методі ConfigureServices;

— включення middleware для Swagger UI в методі Configure;

— анотування контролерів та моделей за допомогою атрибутів.

Використання Swagger у вебплатформі для аналізу залежностей бібліотек коду забезпечує ряд переваг:

— автоматизована генерація документації, яка завжди відповідає реальному коду;

— інтерактивний інтерфейс для тестування API без додаткових інструментів;

— можливість генерації клієнтського коду для різних мов програмування;

— стандартизований формат опису API, який зрозумілий широкому колу

розробників;

— спрощення процесу інтеграції між компонентами системи [6].

Ключові ендпоінти REST API для аналізу залежностей бібліотек коду наведені в таблиці 2.2.

Таблиця 2.2 — Основні ендпоінти REST API вебплатформи

HTTP метод	Ендпоінт	Опис	Параметри	Код відповіді
POST	/api/analyzer/upload	Завантаження DLL файлу для аналізу	multipart/form-data з файлом	201 Created
GET	/api/analyzer/dependencies	Отримання структури залежностей збірки	—	200 OK
GET	/api/analyzer/status	Перевірка статусу аналізу	—	200 OK
DELETE	/api/analyzer/reset	Скидання поточного аналізу	—	204 No Content

Документування API з використанням Swagger є важливим аспектом розробки сучасних вебплатформ, який забезпечує прозорість, зрозумілість та легкість використання програмних інтерфейсів. В контексті вебплатформи для аналізу залежностей бібліотек коду якісна документація API є ключовим фактором успішної інтеграції різних компонентів системи та забезпечення позитивного досвіду користувачів.

2.2 Методи та інструменти аналізу .NET збірок

2.2.1 Застосування рефлексії для роботи з метаданими збірок

Рефлексія (Reflection) є потужним механізмом .NET Framework, який дозволяє програмно досліджувати метадані, типи та члени збірок під час виконання програми. Ця технологія є фундаментальною для розробки інструментів аналізу залежностей, оскільки надає можливість отримати детальну інформацію про

структуру та взаємозв'язки компонентів .NET збірок без доступу до їх вихідного коду.

У контексті .NET, метадані — це структуровані дані, які описують програмні елементи: збірки, модулі, типи, методи, властивості та інші члени. Метадані зберігаються у збірках (DLL або EXE файлах) і можуть бути доступні через API рефлексії. Важливо зазначити, що .NET збірки містять не лише виконуваний код, але й багату інформацію про типи та їх відносини, що робить рефлексію особливо ефективною для аналізу залежностей [7].

Основою механізму рефлексії в .NET є простір імен System.Reflection, який містить класи та інтерфейси для роботи з метаданими збірок. Ключовими типами цього простору імен є Assembly, Module, Type, MethodInfo, PropertyInfo, FieldInfo та інші, які надають програмний доступ до відповідних елементів метаданих.

Однак робота з рефлексією в контексті аналізу залежностей має певні виклики. При обробці збірок з невирішеними залежностями при завантаженні збірки з невирішеними залежностями може виникнути FileNotFoundException або FileLoadException. Для вирішення цієї проблеми можна використовувати обробку подій AssemblyResolve [8].

При роботі з підписаними збірками важливо враховувати версії та публічні ключі для правильної ідентифікації залежностей.

Варто враховувати, що завантажені збірки залишаються в пам'яті до завершення роботи домену додатку, що може призвести до значного споживання пам'яті при аналізі великої кількості збірок.

Рефлексія також дозволяє аналізувати збірки для різних версій .NET Framework та .NET Core/.NET 5+. Це особливо важливо при аналізі залежностей у проєктах, які використовують різні версії фреймворку або націлені на різні платформи.

Для вирішення цих проблем у розроблюваній вебплатформі реалізовано спеціалізований сервіс, який забезпечує ефективну роботу з рефлексією. Основні сервіси, призначені для роботи з рефлексією, наведені у таблиці 2.3.

Таблиця 2.3 — Сервіси для роботи з рефлексією в розроблюваній вебплатформі

Сервіс	Інтерфейс	Основні функції
ReflectionService	IReflectionService	Завантаження збірок, отримання метаданих, обробка помилок завантаження
AssemblyValidator	IAssemblyValidator	Перевірка валідності збірок, аналіз цифрових підписів
TypeAnalyzerService	ITypeAnalyzerService	Аналіз типів та їх залежностей
DllAnalyzerService	IDllAnalyzerService	Високорівневий аналіз DLL файлів та їх структури

Використання рефлексії в розроблюваній вебплатформі дозволяє:

— автоматично виявляти прямі та транзитивні залежності збірок без доступу до вихідного коду;

— аналізувати сумісність версій залежностей та виявляти потенційні конфлікти;

— досліджувати взаємозв'язки на рівні типів для більш глибокого розуміння структури проєкту;

— будувати детальну деревовидну структуру залежностей для подальшої візуалізації.

Рефлексія є потужним інструментом для роботи з метаданими .NET збірок, який забезпечує основу для аналізу залежностей бібліотек коду. Незважаючи на певні виклики, пов'язані з продуктивністю та управлінням пам'яттю, правильне використання рефлексії дозволяє створити ефективну систему для глибокого аналізу структури .NET додатків.

2.2.2 Специфіка аналізу DLL-файлів у середовищі .NET

DLL (Dynamic Link Library) файли є ключовим елементом екосистеми .NET, який забезпечує модульність та повторне використання коду. В контексті розроблюваної вебплатформи для аналізу залежностей розуміння особливостей роботи з DLL-файлами є критично важливим для створення точного та надійного

інструменту аналізу.

DLL-файли в .NET представляють собою збірки (assemblies), які містять скомпільований код у проміжній мові (Intermediate Language, IL), метадані про типи та ресурси. Важливо розуміти, що .NET DLL-файли суттєво відрізняються від традиційних нативних DLL-файлів Windows своєю структурою та механізмом взаємодії з програмами.

Структура .NET DLL-файлу включає декілька ключових компонентів: PE-заголовок (Portable Executable), метадані .NET, IL-код, маніфест та ресурси. PE-заголовок є стандартним заголовком Windows для виконуваних файлів. Метадані містять інформацію про типи, методи, властивості та інші елементи. IL-код представляє собою проміжний код, який виконується віртуальною машиною CLR. Маніфест включає інформацію про збірку, її версію, культуру та залежності. Ресурси можуть містити вбудовані дані, такі як рядки, зображення та інші бінарні дані [9].

Особливістю .NET збірок є наявність багатого набору метаданих, які забезпечують можливість детального аналізу залежностей на різних рівнях. Метадані зберігаються у вигляді спеціалізованих таблиць, які містять інформацію про всі типи (TypeDef), посилання на типи з інших збірок (TypeRef), методи (MethodDef), властивості (PropertyDef) тощо [10].

Робота з DLL-файлами в .NET має ряд особливостей, які необхідно враховувати при розробці системи аналізу залежностей.

Вибір методу завантаження залежить від конкретних потреб аналізу. Для вебплатформи аналізу залежностей оптимальним є використання AssemblyLoadContext з можливістю вивантаження, що дозволяє ефективно управляти пам'яттю при аналізі великої кількості збірок.

При завантаженні DLL-файлу .NET автоматично намагається завантажити всі його залежності. Цей процес слідує певному алгоритму пошуку: перевірка наявності збірки у кеші збірок (Global Assembly Cache, GAC), пошук у поточному каталозі програми, пошук у приватних шляхах пробації (Private Probe Paths), виклик обробника події AssemblyResolve. Для ефективного аналізу DLL-файлів

необхідно контролювати цей процес, особливо при роботі з файлами, які мають невирішені залежності.

На рисунку 2.4 наведена структура типового DLL-файлу в .NET.

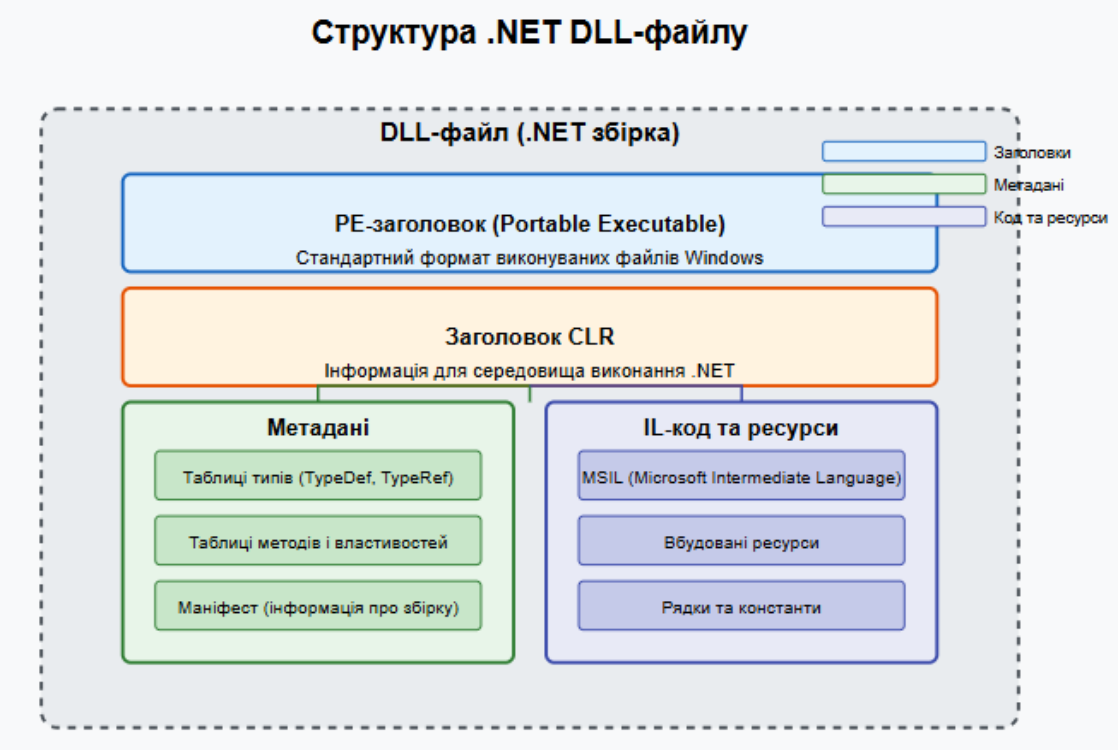

Рисунок 2.4 — Структура .NET DLL-файлу

Сильно іменовані збірки мають цифровий підпис, який гарантує їх цілісність та унікальність. Ідентифікація таких збірок включає ім'я, версію, культуру та відкритий ключ у форматі: Assembly Name, Version=1.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089. При аналізі залежностей важливо коректно ідентифікувати збірки та враховувати правила версіонування для визначення сумісності різних версій [11].

Сучасні DLL-файли часто створюються для підтримки різних версій .NET Framework, .NET Core або .NET Standard. Такі збірки містять метадані та код для різних цільових платформ, що ускладнює аналіз залежностей. Для виявлення цільових платформ можна використовувати атрибути TargetFrameworkAttribute та TargetFrameworksAttribute, які містять інформацію про підтримувані фреймворки.

.NET підтримує взаємодію з нативним кодом через P/Invoke або C++/CLI.

Аналіз залежностей таких DLL-файлів вимагає спеціальних підходів, включаючи перевірку наявності атрибутів DllImportAttribute та аналіз імпортованих функцій.

У таблиці 2.4 наведені основні Типи DLL-файлів у .NET та особливості їх аналізу.

Таблиця 2.4 — Типи DLL-файлів у .NET та особливості їх аналізу

Тип DLL-файлу	Характеристики	Особливості аналізу
Керовані (Managed) DLL	Містять лише IL-код та метадані .NET	Стандартний аналіз через рефлексію
Нативні (Native) DLL	Містять нативний код для конкретної платформи	Обмежений аналіз через відсутність метаданих .NET
Змішані (Mixed-Mode) DLL	Комбінація керованого та нативного коду	Аналіз керованої частини через рефлексію, обмежений аналіз нативної частини
Багатоцільові (Multi-targeting) DLL	Підтримка різних версій .NET	Аналіз з урахуванням цільової платформи
Сателітні збірки (Satellite Assemblies)	Локалізовані ресурси	Аналіз зв'язків з основною збіркою

Після аналізу DLL-файлу результати зберігаються в структурованому форматі, який може бути використаний для візуалізації. Типова структура даних для представлення дерева залежностей включає інформацію про ім'я збірки, версію, наявність цифрового підпису та стан завантаження, а також рекурсивні посилання на залежності (рисунок 2.5).

У розроблюваній вебплатформі для аналізу залежностей бібліотек коду особливості роботи з DLL-файлами враховані через впровадження спеціалізованих сервісів та компонентів, які забезпечують безпечний та ефективний аналіз різних типів збірок. Розуміння внутрішньої структури DLL-файлів, механізмів завантаження та розв'язання залежностей дозволяє створити точну та інформативну візуалізацію деревовидної структури залежностей, що є ключовою метою проєкту.

Поєднання глибокого розуміння особливостей DLL-файлів у .NET з

ефективними методами їх аналізу через рефлексію забезпечує надійну основу для побудови потужного інструменту аналізу залежностей, який може бути використаний розробниками для оптимізації та вдосконалення своїх програмних рішень (рисунок 2.5) [12].

Рисунок 2.5 — Процес аналізу DLL-файлу із побудовою дерева залежностей

2.3 Технології візуалізації структур даних

2.3.1 Порівняльний аналіз бібліотек для візуалізації графів

Візуалізація складних структур даних, зокрема графів та дерев, є одним із ключових аспектів розробки вебплатформи для аналізу залежностей бібліотек коду. Ефективна візуалізація дозволяє користувачам швидко сприймати та аналізувати складні взаємозв'язки між компонентами програмних систем.

Сучасний ландшафт технологій візуалізації графів пропонує широкий спектр

бібліотек та фреймворків, кожен з яких має свої переваги та обмеження. Вибір конкретної технології залежить від вимог до інтерактивності, продуктивності, інтеграції з іншими компонентами та естетики представлення даних [13].

D3.js (Data-Driven Documents) є однією з найпопулярніших та найпотужніших бібліотек для візуалізації даних у веб-середовищі. Бібліотека забезпечує низькорівневий контроль над візуалізацією, дозволяючи створювати високо кастомізовані інтерактивні графи. D3.js підтримує різні алгоритми розміщення вузлів (layout algorithms), включаючи силові (force-directed), ієрархічні та радіальні. Бібліотека має активну спільноту розробників та багату екосистему плагінів. Однак, D3.js має досить високий поріг входження через складний API та необхідність глибокого розуміння HTML, SVG та JavaScript.

Cytoscape.js є спеціалізованою бібліотекою для візуалізації та аналізу графів, яка добре підходить для представлення мережевих структур. Вона пропонує широкий набір функцій для роботи з графами, включаючи різні алгоритми розміщення, стилізацію вузлів та ребер, інтерактивні елементи та анімацію. Cytoscape.js має високу продуктивність завдяки оптимізованому рендерингу та підтримує як канвас (HTML5 Canvas), так і SVG для відображення. Бібліотека також надає API для аналізу графів, включаючи пошук шляхів, кластеризацію та інші алгоритми [14].

Vis.js є модульною бібліотекою для візуалізації даних, яка включає компоненти для роботи з мережами (Network), часовими шкалами (Timeline) та графіками (Graph2d, Graph3d). Модуль Network спеціально розроблений для відображення мережевих структур та графів з високим рівнем інтерактивності. Vis.js забезпечує простий у використанні API та має хорошу продуктивність для середніх за розміром графів. Бібліотека добре підходить для швидкої реалізації базових візуалізацій, але може мати обмеження при роботі з дуже великими графами або при необхідності складної кастомізації.

Sigma.js є легкою бібліотекою для візуалізації графів, оптимізованою для відображення великих мереж. Вона підтримує рендеринг через WebGL, Canvas та SVG, забезпечуючи високу продуктивність навіть для графів з тисячами вузлів.

Sigma.js має мінімалістичний дизайн та модульну архітектуру, що дозволяє розширювати її функціональність за допомогою плагінів. Бібліотека підтримує базові інтерактивні функції, такі як масштабування, панорамування та вибір вузлів, але може вимагати додаткової роботи для реалізації більш складних інтерактивних елементів [15].

У таблиці 2.5 наведене порівняння бібліотек для візуалізації графів, які можуть застосовуватись для візуалізації деревоподібних залежностей.

Таблиця 2.5 — Порівняння бібліотек для візуалізації графів

Характеристика	D3.js	Cytoscape.js	Vis.js	Sigma.js	Dash Cytoscape
Рівень абстракції	Низький	Середній	Високий	Середній	Високий
Кастомізація	Висока	Висока	Середня	Середня	Висока
Продуктивність з великими графами	Середня	Висока	Середня	Висока	Висока
Інтерактивність	Висока	Висока	Висока	Середня	Висока
Інтеграція з Python	Через сторонні бібліотеки	Через сторонні бібліотеки	Через сторонні бібліотеки	Через сторонні бібліотеки	Нативна
Алгоритми розміщення	Багато	Багато	Обмежено	Обмежено	Багато
Документація	Відмінна	Добра	Добра	Базова	Відмінна
Підтримка спільноти	Активна	Активна	Помірна	Обмежена	Зростаюча

Dash Cytoscape є компонентом бібліотеки Dash від Plotly, який забезпечує інтеграцію Cytoscape.js з Python. Це особливо важливо для розроблюваної вебплатформи, оскільки дозволяє використовувати потужність Python для обробки та аналізу даних, одночасно надаючи багаті можливості візуалізації графів від Cytoscape.js. Dash Cytoscape пропонує декларативний API для побудови інтерактивних графів, що спрощує розробку та підтримку коду. Компонент

підтримує всі основні можливості Cytoscape.js, включаючи різні алгоритми розміщення, стилізацію вузлів та ребер, події та інтерактивні функції. На рисунку 2.6 будуть показані приклади візуалізації графів з використанням різних бібліотек.

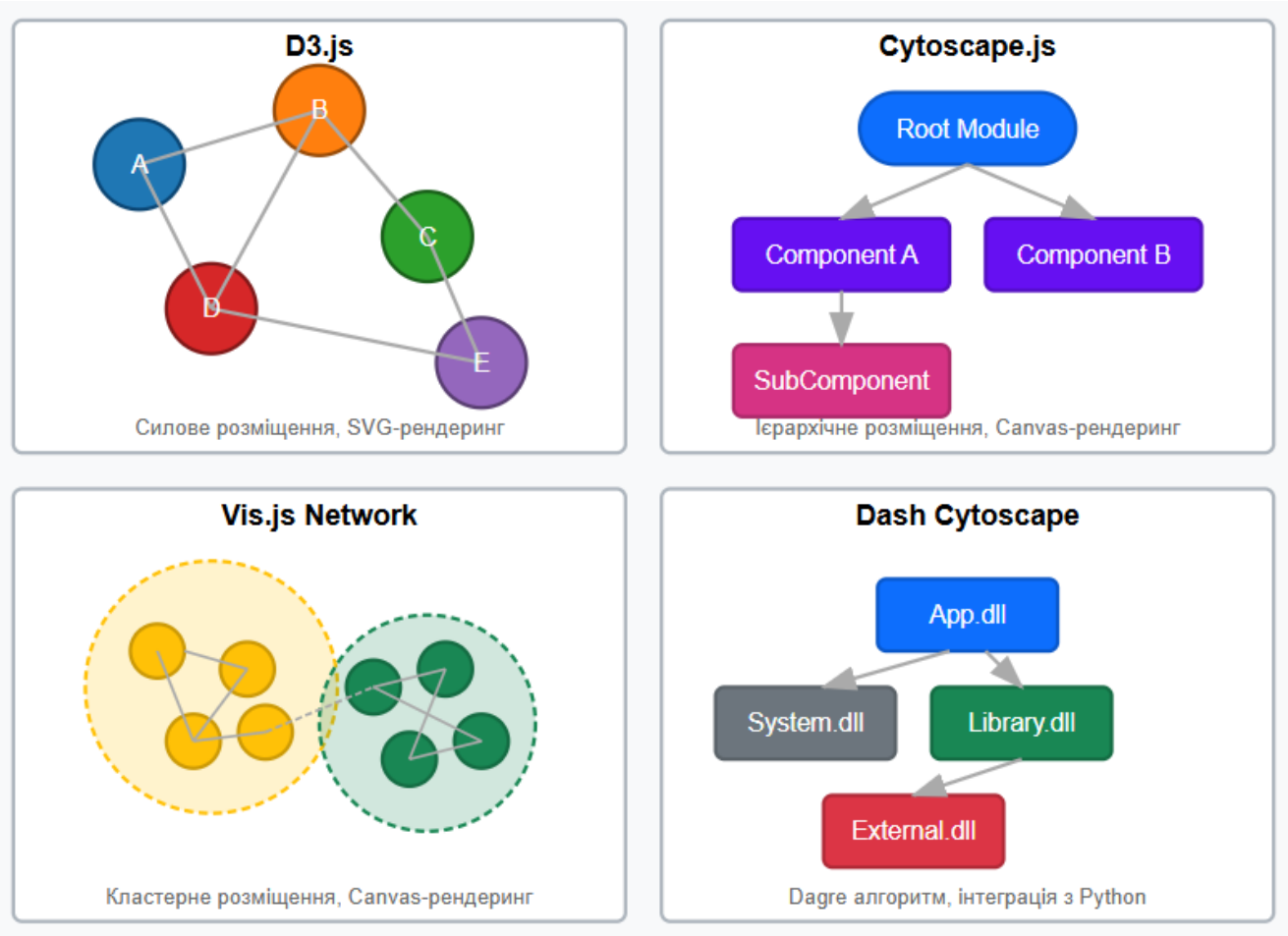

Рисунок 2.6 — Приклади візуалізації графів з використанням різних бібліотек

При виборі бібліотеки для візуалізації графових структур у контексті аналізу залежностей бібліотек коду необхідно враховувати ряд факторів:

Масштабованість є критичним аспектом при роботі з великими проєктами, які можуть мати сотні або навіть тисячі залежностей. Бібліотека повинна забезпечувати ефективне відображення та взаємодію з великими графами без значного зниження продуктивності [16].

Інтерактивність забезпечує можливість користувачам досліджувати та аналізувати граф залежностей, включаючи масштабування, панорамування, вибір

вузлів та ребер, фільтрацію та отримання детальної інформації про окремі компоненти.

Алгоритми розміщення вузлів є важливим аспектом візуалізації графів, оскільки правильне розташування елементів значно впливає на сприйняття та розуміння структури. Для різних типів графів можуть бути оптимальними різні алгоритми: ієрархічні для деревовидних структур, силові для мереж з багатьма взаємозв'язками, радіальні для центрованих графів.

Інтеграція з іншими технологіями, зокрема з серверною частиною на ASP.NET Core та системою аналізу залежностей, є важливим фактором для забезпечення ефективної роботи вебплатформи як єдиної системи [17].

Для візуалізації деревовидної структури залежностей бібліотек коду у розроблюваній вебплатформі було обрано Dash Cytoscape з наступних причин:

— нативна інтеграція з Python, що дозволяє легко використовувати результати аналізу, отримані від компонента на ASP.NET Core, для побудови візуалізації.

— висока продуктивність при роботі з великими графами, що критично для аналізу складних проєктів з багатьма залежностями.

— багатий набір інтерактивних функцій, включаючи масштабування, панорамування, виділення вузлів та ребер, що забезпечує зручний аналіз структури залежностей.

— підтримка різних алгоритмів розміщення вузлів, включаючи dagre для спрямованих ациклічних графів, що найкраще підходить для візуалізації ієрархічних структур залежностей.

— можливість кастомізації зовнішнього вигляду вузлів та ребер для представлення різних типів залежностей та їх характеристик.

На рисунку 2.7 показаний приклад візуалізації залежностей у деревоподібній формі з використанням Dash Cytoscape.

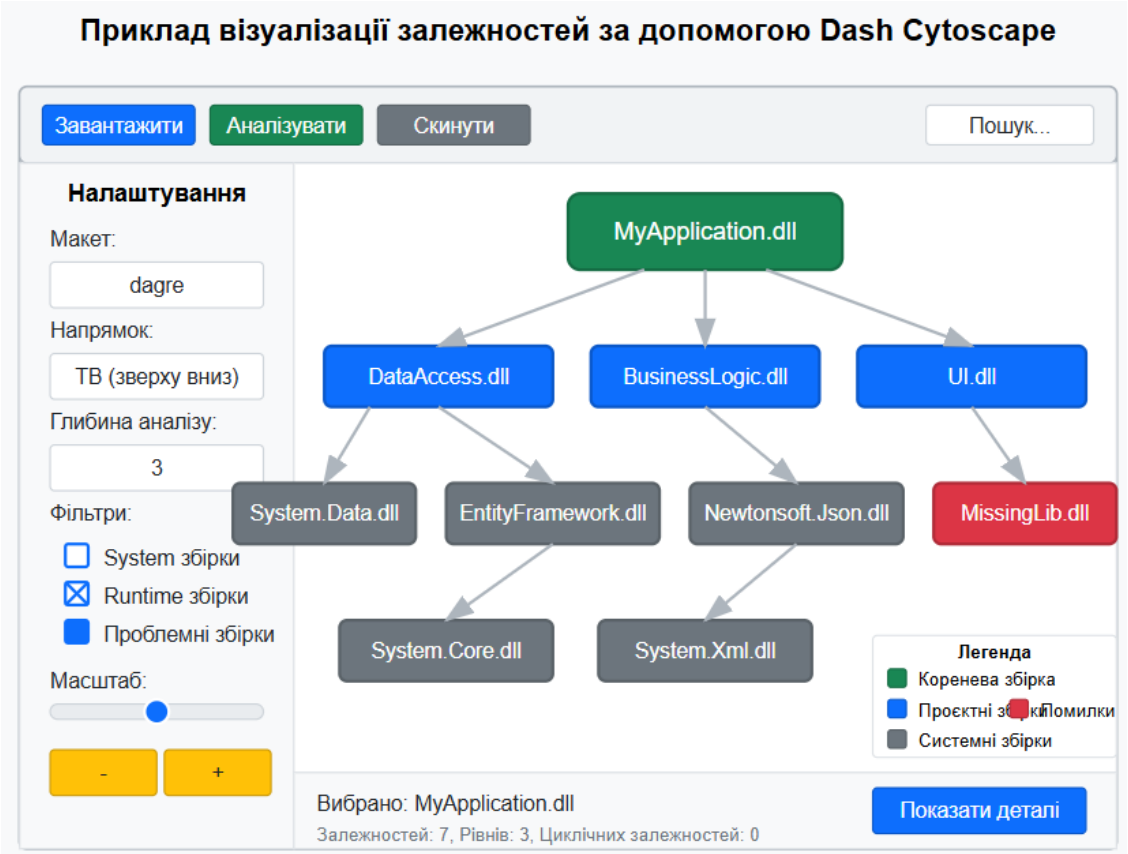

Рисунок 2.7 — Приклад візуалізації залежностей за допомогою Dash Cytoscape

Dash Cytoscape пропонує ряд алгоритмів розміщення вузлів, включаючи dagre для ієрархічних структур, cose для силових графів, concentric для концентричних графів та інші. Вибір конкретного алгоритму залежить від типу графу та вимог до його візуального представлення [18].

Таким чином, аналіз сучасних бібліотек для побудови графових структур показав, що Dash Cytoscape є оптимальним вибором для реалізації візуалізації деревовидної структури залежностей бібліотек коду у розроблюваній вебплатформі. Ця технологія забезпечує ефективну інтеграцію з Python, високу продуктивність та багатий набір інтерактивних функцій, що дозволяє створити зручний та інформативний інструмент для аналізу залежностей.

2.3.2 Засоби Python Dash для створення інтерактивних візуалізацій

Python Dash є потужним фреймворком для створення аналітичних веб-додатків, який поєднує простоту Python з інтерактивністю сучасних веб-

технологій. Розроблений компанією Plotly, Dash забезпечує декларативний підхід до побудови інтерактивних веб-інтерфейсів без необхідності глибокого знання HTML, CSS та JavaScript. В контексті візуалізації деревовидних структур залежностей, Dash надає унікальні можливості через компонент Dash Cytoscape, який інтегрує потужну JavaScript бібліотеку Cytoscape.js у екосистему Python.

Dash Cytoscape є спеціалізованим компонентом бібліотеки Dash, розробленим для візуалізації складних графових структур. Цей компонент унаслідує всі потужні функції JavaScript бібліотеки Cytoscape.js, але надає можливість використовувати їх через простий та інтуїтивно зрозумілий Python API. Це дозволяє розробникам створювати складні інтерактивні графові візуалізації без необхідності писати JavaScript код, що значно прискорює процес розробки та спрощує підтримку проєкту.

Основою архітектури Dash є концепція реактивності, де компоненти інтерфейсу автоматично оновлюються при зміні даних або стану програми. Ця концепція реалізується через систему колбеків, яка пов'язує вхідні та вихідні параметри компонентів. Dash Cytoscape інтегрується в цю систему, дозволяючи динамічно оновлювати граф залежностей у відповідь на дії користувача або зміни у вхідних даних.

Dash Cytoscape підтримує різні алгоритми розміщення вузлів, які оптимізовані для різних типів графів. Для візуалізації деревовидної структури залежностей найбільш підходять ієрархічні алгоритми, такі як 'dagre', який спеціально розроблений для спрямованих ациклічних графів (Directed Acyclic Graphs, DAG). Цей алгоритм розміщує вузли в ієрархічну структуру, що полегшує розуміння залежностей між компонентами.

Інтерактивність є важливим аспектом візуалізації графів, особливо для аналізу складних структур залежностей. Dash Cytoscape надає багаті можливості для взаємодії з графом через систему подій та колбеків. Користувачі можуть вибирати вузли та ребра для отримання детальної інформації, масштабувати та панорамувати граф для дослідження різних частин структури, перетягувати вузли

для кращого розуміння їх взаємозв'язків та фільтрувати вузли та ребра за різними критеріями.

У таблиці 2.6 наведене порівняння алгоритмів, що застосовуються для розміщення вузлів у Dash Cytoscape.

Таблиця 2.6 — Алгоритми розміщення вузлів у Dash Cytoscape

Алгоритм	Тип графів	Особливості	Застосування
dagre	Спрямовані ациклічні графи	Ієрархічне розміщення, підтримка напрямків (TB, LR, BT, RL)	Деревовидні структури, залежності
cose	Загальні графи	Силове розміщення, оптимізація для естетики	Мережі з багатьма взаємозв'язками
breadthfirst	Дерева та DAG	Розміщення по рівнях у ширину	Візуалізація ієрархій
concentric	Загальні графи	Розміщення вузлів концентричними колами	Централізовані мережі
circle	Загальні графи	Розміщення вузлів по колу	Циклічні структури

Для ефективної роботи з великими графами залежностей Dash Cytoscape пропонує ряд оптимізацій: використання WebGL для рендерингу при великій кількості елементів, підтримка вибіркового відображення елементів для зменшення навантаження, можливість розділення графу на підграфи для поетапного відображення та оптимізовані алгоритми розміщення для великих графів [19].

Dash Cytoscape також надає можливості для стилізації графу в залежності від типу та характеристик вузлів та ребер. Це дозволяє візуально розділяти різні типи залежностей, наприклад, використовуючи різні кольори для системних бібліотек, проєктних компонентів та проблемних залежностей.

Для обміну даними між бекендом на ASP.NET Core та візуалізацією на Dash Cytoscape використовується JSON як універсальний формат. Це дозволяє ефективно передавати складні структури даних між різними компонентами системи.

Використання Python Dash та Dash Cytoscape для візуалізації графів залежностей має ряд переваг порівняно з іншими підходами: простота інтеграції з Python екосистемою для аналізу та обробки даних, декларативний підхід до

створення інтерактивних веб-інтерфейсів, багаті можливості для стилізації та кастомізації графів, ефективна робота з великими та складними графовими структурами та проста інтеграція з іншими аналітичними компонентами.

В розроблюваній вебплатформі Python Dash та Dash Cytoscape забезпечують інтуїтивно зрозумілу та інформативну візуалізацію деревовидної структури залежностей бібліотек коду, що дозволяє розробникам ефективно аналізувати та розуміти взаємозв'язки між компонентами їхніх програмних систем.

2.3.3 Методи інтеграції Python-компонентів з .NET-додатками

Інтеграція між .NET та Python є важливим аспектом сучасної розробки програмного забезпечення, який дозволяє поєднувати сильні сторони обох платформ. .NET забезпечує надійну основу для серверних додатків, роботи з базами даних та аналізу збірок, тоді як Python пропонує потужні інструменти для візуалізації даних, машинного навчання та наукових обчислень. Така інтеграція особливо цінна у випадку розроблюваної вебплатформи для аналізу залежностей, де .NET компоненти відповідають за аналіз DLL-файлів, а Python компоненти — за інтерактивну візуалізацію результатів [20].

Існує кілька підходів до інтеграції Python з .NET додатками, кожен з яких має свої переваги та обмеження:

— процесна інтеграція — .NET додаток запускає Python як окремий процес та обмінюється даними через стандартні потоки вводу/виводу або файли, цей підхід забезпечує повну ізоляцію середовищ виконання, але обмежує інтерактивну взаємодію між компонентами;

— бібліотечна інтеграція — використання спеціалізованих бібліотек, таких як Python.NET або IronPython, які дозволяють викликати Python код безпосередньо з .NET додатку, цей підхід забезпечує тісну інтеграцію, але може мати обмеження щодо сумісності з Python бібліотеками;

— мікросервісна інтеграція — розділення .NET та Python компонентів на окремі сервіси, які взаємодіють через REST API або інші механізми міжпроцесної

комунікації, цей підхід забезпечує гнучкість та масштабованість, але вимагає додаткової інфраструктури;

— кросплатформна інтеграція — використання кросс-платформних технологій, таких як WebAssembly або Docker, для запуску Python та .NET компонентів у спільному середовищі, цей підхід є відносно новим та експериментальним.

У розроблюваній вебплатформі для аналізу залежностей бібліотек коду використовується комбінація процесної та мікросервісної інтеграції. ASP.NET Core додаток запускає Python скрипт з Dash Cytoscape як окремий процес, а обмін даними відбувається через JSON-файл. Це забезпечує високу надійність, простоту реалізації та гнучкість у виборі технологій для кожного компонента.

На рисунку 2.8 наведена схема інтеграції ASP.NET Core з Python Dash у розробленій вебплатформі. Ця інтеграція є важливою частиною мікросервісної архітектури вебплатформи, що дозволяє ефективно розділити відповідальність між компонентами та забезпечити гнучкість системи в цілому.

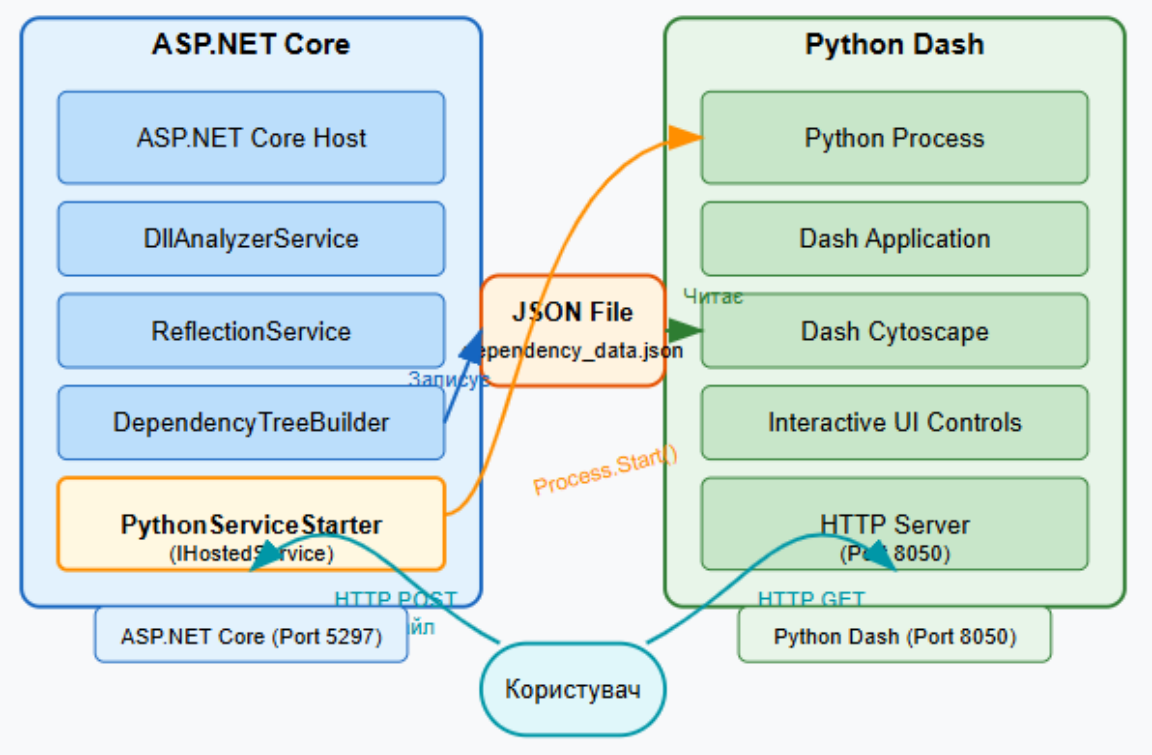

Рисунок 2.8 — Схема інтеграції ASP.NET Core з Python Dash у розробленій вебплатформі

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ ВЕБПЛАТФОРМИ

3.1 Архітектура та технічні рішення вебплатформи

3.1.1 Розробка API з використанням ASP.NET Core

Розроблена вебплатформа для аналізу залежностей бібліотек коду реалізована на основі сучасних підходів до архітектури програмного забезпечення, з урахуванням принципів мікросервісної архітектури та розділення відповідальності. Система складається з двох основних компонентів: серверної частини на базі ASP.NET Core, яка відповідає за аналіз DLL-файлів, та компонента візуалізації на основі Python Dash, який забезпечує інтерактивне представлення результатів аналізу.

Одним з ключових аспектів архітектури системи є взаємодія між цими компонентами. Для забезпечення ефективної інтеграції було розроблено спеціалізований механізм комунікації, який забезпечує незалежність та гнучкість обох частин системи.

Компоненти серверної частини (ASP.NET Core):

— Web API Controllers — обробляють HTTP-запити від клієнтів, забезпечують завантаження DLL-файлів та управління процесом аналізу;

— сервіси аналізу DLL — виконують детальний аналіз структури DLL-файлів, вилучення метаданих та побудову дерева залежностей;

— сервіси інтеграції з Python — забезпечують запуск та управління Python-компонентом для візуалізації.

Компоненти візуалізації (Python Dash):

— Dash Application — забезпечує веб-інтерфейс для взаємодії з користувачем;

— Dash Cytoscape — відповідає за інтерактивну візуалізацію графа залежностей;

— сервіси обробки даних — перетворюють JSON-дані у формат, придатний для візуалізації.

Для обробки запитів на завантаження та аналіз DLL-файлів використовується контролер AnalysisController, який реалізує RESTful API та забезпечує взаємодію з клієнтською частиною (лістинг 3.1).

Лістинг 3.1 — Контролер AnalysisController

```
[ApiController]
[Route("api/[controller]")]
public class AnalysisController : ControllerBase{
private readonly
IAssemblyAnalysisService _assemblyAnalysisService;
private readonly ILogger<AnalysisController> _logger;
private readonly IDllAnalyzerService _dllAnalyzerService;
public AnalysisController(
IAssemblyAnalysisService assemblyAnalysisService,
ILogger<AnalysisController> logger,
IDllAnalyzerService dllAnalyzerService){
_assemblyAnalysisService = assemblyAnalysisService;
_logger = logger;
_dllAnalyzerService = dllAnalyzerService; }
 [HttpPost("upload")]
 [Consumes("multipart/form-data")]
public IActionResult UploadDll([FromForm] FileUploadModel model)
{ // Обробка запиту на завантаження та аналіз DLL-файлу }}
```

Контролер використовує принцип впровадження залежностей (Dependency Injection) для отримання необхідних сервісів, що забезпечує гнучкість та тестованість коду.

Ключовим компонентом, що забезпечує взаємодію між ASP.NET Core та Python, є сервіс PythonServiceStarter, який реалізує інтерфейс IHostedService та відповідає за запуск та управління Python-процесом (лістинг 3.2).

Лістинг 3.2 — Використання сервісу для запуску Python

```
public class PythonServiceStarter : IHostedService, IDisposable {
private readonly ILogger<PythonServiceStarter> _logger;
private readonly IConfiguration _configuration;
private Process _pythonProcess;
private string _dependencyJsonPath;
// Конструктор та реалізація методів інтерфейсу
}
```

Цей сервіс автоматично запускається при старті ASP.NET Core додатку та забезпечує запуск Python скрипта з Dash Cytoscape на окремому порту. Таким

чином, користувач може взаємодіяти з обома компонентами системи через веб-браузер.

Обмін даними між ASP.NET Core та Python компонентами відбувається через JSON-файл dependency_data.json. Коли користувач завантажує DLL-файл для аналізу, контролер AnalysisController обробляє запит, виконує аналіз за допомогою сервісів IAssemblyAnalysisService або IDllAnalyzerService (залежно від типу аналізу), а потім зберігає результати у JSON-файлі. Python компонент постійно моніторить цей файл та оновлює візуалізацію при зміні даних. Це забезпечує інтерактивність системи та швидку відповідь на дії користувача.

На рисунку 3.1 представлена детальна схема взаємодії компонентів вебплатформи для аналізу залежностей бібліотек коду.

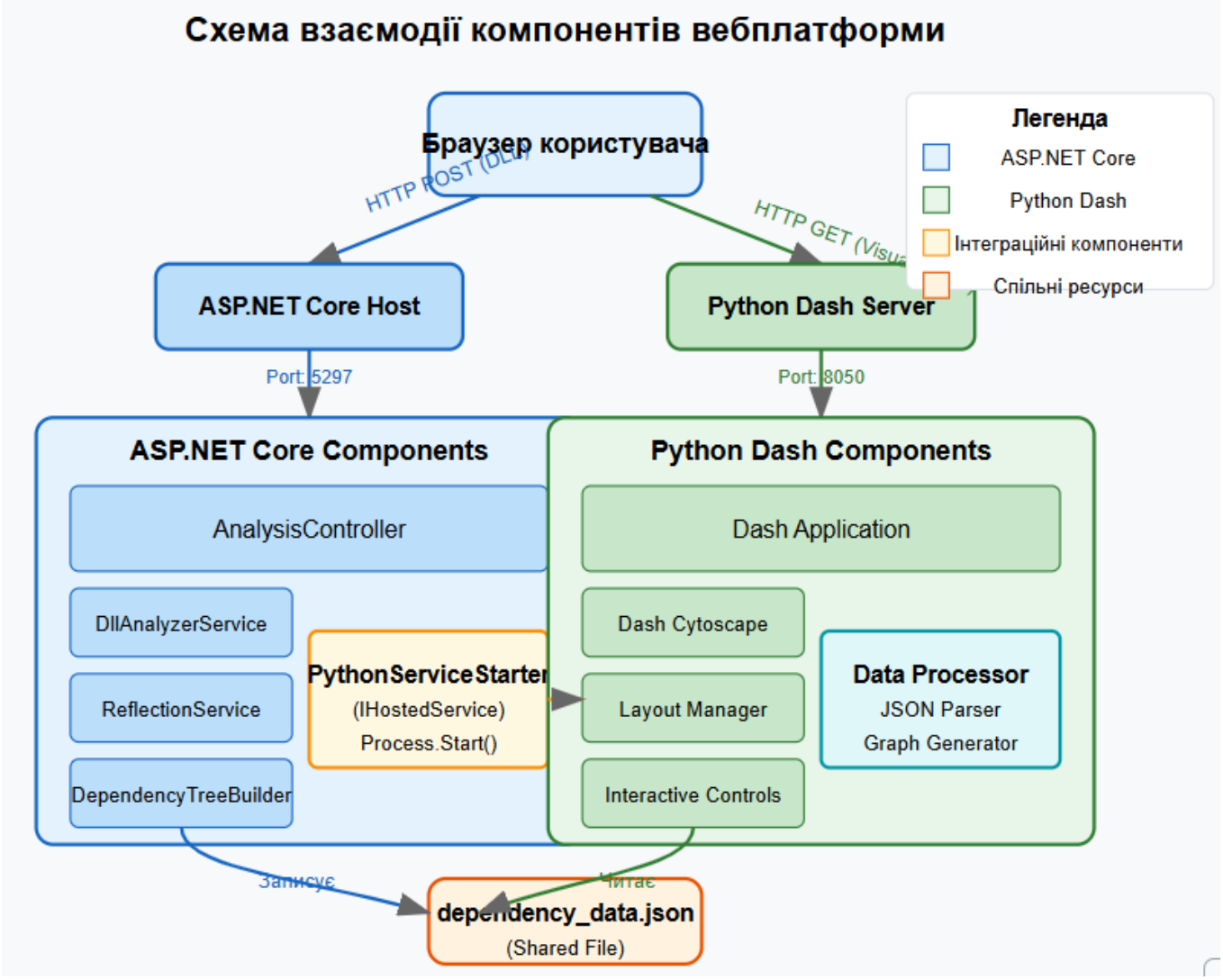

Рисунок 3.1 — Схема взаємодії компонентів вебплатформи

Як показано на рисунку 3.1, архітектура системи складається з двох основних компонентів: ASP.NET Core компонентів (ліва частина) та Python Dash компонентів (права частина), які взаємодіють через спільний файл даних. Користувач взаємодіє з системою через веб-браузер, відправляючи DLL файли для аналізу на ASP.NET Core сервер (порт 5297) та отримуючи візуалізацію результатів від Python Dash сервера (порт 8050).

ASP.NET Core компоненти включають контролер AnalysisController, який приймає запити на аналіз DLL файлів, та сервіси аналізу (DllAnalyzerService, ReflectionService, DependencyTreeBuilder), які виконують глибокий аналіз структури збірок. Важливим елементом інтеграції є PythonServiceStarter, який реалізує інтерфейс IHostedService та відповідає за запуск та управління Python процесом.

Python Dash компоненти забезпечують інтерактивну візуалізацію результатів аналізу. Вони включають базовий додаток Dash, компонент Dash Cytoscape для візуалізації графа залежностей, менеджер розміщення вузлів та інтерактивні елементи управління. Компонент обробки даних відповідає за парсинг JSON та генерацію структури графа.

Обмін даними між компонентами відбувається через спільний JSON-файл dependency_data.json. ASP.NET Core компоненти записують результати аналізу у цей файл, а Python Dash компоненти зчитують ці дані для візуалізації. Такий підхід забезпечує простоту інтеграції та незалежність компонентів, дозволяючи їм розвиватися та масштабуватися незалежно один від одного.

Розробка ефективного та зручного API є ключовим елементом вебплатформи для аналізу залежностей бібліотек коду. REST (Representational State Transfer) був обраний як архітектурний стиль для розробки API через його простоту, масштабованість та відповідність принципам веб-технологій. ASP.NET Core надає потужні інструменти для розробки REST API, включаючи систему маршрутизації, фільтри, валідацію моделей та документацію за допомогою Swagger.

У розробленій вебплатформі REST API забезпечує наступні функції:

— завантаження DLL-файлів для аналізу;

— отримання результатів аналізу у форматі JSON;

— підтримка різних рівнів деталізації аналізу;

— забезпечення інформації про статус аналізу.

Основним компонентом, що реалізує REST API, є контролер AnalysisController, який обробляє HTTP-запити, пов'язані з аналізом DLL-файлів. LibraryController надає функціональність для генерації DLL-файлів з вихідного коду C#, що дозволяє користувачам створювати тестові бібліотеки безпосередньо через веб-інтерфейс (лістинг 3.3).

Лістинг 3.3 — Контролер LibraryController

```
[ApiController]
[Route("api/[controller]")]
public class LibraryController : ControllerBase{
private readonly IDllFactory _dllFactory;
private readonly ILogger<LibraryController> _logger;
public LibraryController(
IDllFactory dllFactory, ILogger<LibraryController> logger){
_dllFactory = dllFactory;
_logger = logger;}
[HttpPost("generate")]
[ProducesResponseType(typeof(object), StatusCodes.Status200OK)]
[ProducesResponseType(typeof(string),
StatusCodes.Status400BadRequest)]
[ProducesResponseType(typeof(string),
StatusCodes.Status500InternalServerError)]
public async Task<IActionResult>
GenerateLibrary([FromBody] LibraryGenerationRequest request)
{ // Імплементація методу... }}
```

Бізнес-логіка API реалізована через систему інтерфейсів та сервісів, які відповідають за різні аспекти функціональності:

— IAssemblyAnalysisService — детальний аналіз збірок;

— IDllAnalyzerService — аналіз залежностей DLL-файлів;

— IDependencyTreeBuilder — побудова дерева залежностей;

— IReflectionService — робота з рефлексією .NET;

— IAssemblyValidator — валідація збірок;

— ITypeAnalyzerService — аналіз типів у збірках;

— IDllFactory — генерація DLL-файлів з вихідного коду.

Таблиця 3.1 представляє основні ендпоінти REST API розробленої вебплатформи.

Таблиця 3.1 — Основні ендпоінти REST API вебплатформи

HTTP-метод	Шлях	Опис	Параметри	Відповідь
POST	/api/analysis/upload	Завантаження DLL-файлу для аналізу	File (DLL-файл), Detailed (boolean)	JSON з результатами аналізу
POST	/api/library/generate	Генерація DLL з вихідного коду	SourceCode (string), OutputDllName (string)	JSON з шляхом до згенерованої DLL

Розроблене REST API надає простий та зрозумілий інтерфейс для завантаження та аналізу DLL-файлів, а також генерації тестових бібліотек. API дотримується принципів REST, включаючи використання відповідних HTTP-методів, кодів статусу та формату JSON для передачі даних. Завдяки модульній архітектурі та принципу впровадження залежностей, API легко масштабується та розширюється для підтримки нових функцій.

3.1.2 Документування API з використанням Swagger

Документування API є важливим аспектом розробки сучасних веб-додатків, оскільки забезпечує розробників та користувачів зрозумілим описом доступних ендпоінтів, параметрів та моделей даних. У розробленій вебплатформі для документування REST API використовується Swagger (OpenAPI) — стандарт та набір інструментів, який надає інтерактивний інтерфейс для дослідження та тестування API.

Swagger інтегрується в ASP.NET Core за допомогою набору пакетів Swashbuckle, які автоматично генерують специфікацію OpenAPI на основі структури контролерів, моделей та атрибутів. Це значно спрощує підтримку документації в актуальному стані, оскільки вона генерується динамічно з коду.

Для включення XML-коментарів у Swagger необхідно налаштувати

генерацію XML-файлу документації в проєкті та вказати Swagger використовувати цей файл.

Структура документації Swagger для API вебплатформи представлена на рисунку 3.2, де видно основні ендпоінти, їхні параметри та моделі даних.

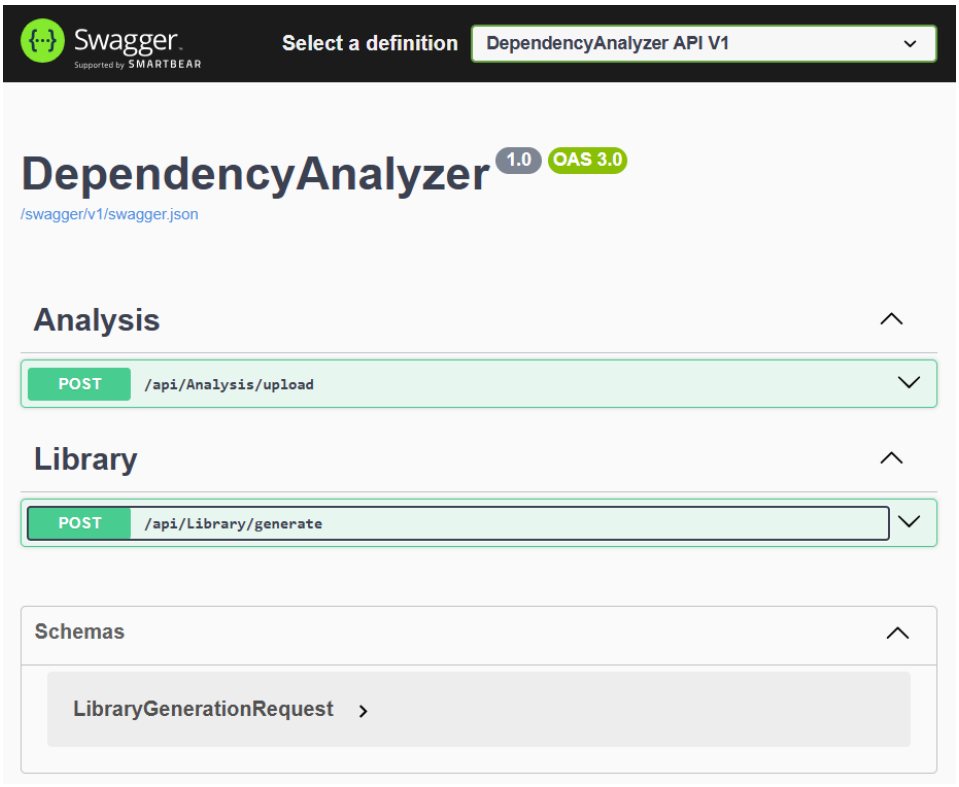

Рисунок 3.2 — Інтерфейс Swagger UI для API вебплатформи

Swagger UI надає інтерактивний інтерфейс для роботи з API, який дозволяє:

— переглядати доступні ендпоінти та їх опис;

— вивчати моделі даних, які використовуються для запитів та відповідей;

— тестувати API безпосередньо з браузера, без необхідності використання додаткових інструментів;

— експортувати специфікацію OpenAPI для використання в інших інструментах.

Цей інтерфейс особливо корисний під час розробки та тестування, оскільки дозволяє швидко перевірити функціональність API без написання клієнтського коду.

Для вебплатформи аналізу залежностей Swagger надає документацію для

двох основних контролерів: AnalysisController та LibraryController. Для кожного контролера документація включає опис доступних ендпоінтів, методів HTTP, параметрів та можливих відповідей.

Наприклад, для ендпоінту /api/analysis/upload документація Swagger відображає:

— метод HTTP: POST;

— тип вмісту: multipart/form-data;

— параметри: File (файл DLL), Detailed (булевий прапорець);

— можливі відповіді: 200 OK (JSON результат аналізу), 400 Bad Request (помилка запиту), 500 Internal Server Error (внутрішня помилка сервера).

Для ендпоінту /api/library/generate документація відображає:

— метод HTTP: POST;

— тип вмісту: application/json;

— схема запиту: LibraryGenerationRequest з властивостями SourceCode та OutputDllName;

— можливі відповіді: 200 OK (шлях до згенерованої DLL), 400 Bad Request (помилка запиту або компіляції), 500 Internal Server Error (внутрішня помилка сервера).

Swagger також автоматично генерує схеми для моделей даних, які використовуються в API, таких як FileUploadModel, DependencyNode, AnalysisReport та LibraryGenerationRequest. Це дозволяє розробникам, які використовують API, розуміти структуру даних, які вони повинні надіслати або очікують отримати.

Використання Swagger для документування API має ряд переваг:

— автоматична генерація документації, яка завжди відповідає реальному коду;

— інтерактивний інтерфейс для тестування API без додаткових інструментів;

— стандартизований формат опису API, який зрозумілий широкому колу розробників;

— можливість експорту документації для інтеграції з іншими інструментами та системами.

У контексті вебплатформи для аналізу залежностей бібліотек коду документація API є важливим компонентом, який забезпечує розуміння функціональності системи та спрощує її використання та інтеграцію з іншими системами.

3.2 Реалізація підсистеми аналізу залежностей

3.2.1 Алгоритми завантаження та аналізу DLL-файлів

Ключовим компонентом розробленої вебплатформи є аналізатор залежностей, який забезпечує завантаження, обробку та аналіз DLL файлів для побудови деревовидної структури залежностей. Реалізація цього механізму базується на використанні можливостей рефлексії, яка дозволяє отримувати метадані про типи та залежності збірок під час виконання програми.

Основним компонентом, відповідальним за аналіз DLL файлів, є сервіс DllAnalyzerService, який реалізує інтерфейс IDllAnalyzerService. Цей сервіс надає функціональність для завантаження збірки з масиву байтів та рекурсивного аналізу її залежностей, результатом чого є деревовидна структура, представлена об'єктами класу DependencyNode.

Процес аналізу DLL файлу складається з кількох етапів:

— завантаження збірки з масиву байтів;

— отримання інформації про саму збірку;

— рекурсивний аналіз залежних збірок;

— побудова деревовидної структури залежностей.

Рисунок 3.3 ілюструє процес аналізу DLL файлу та побудови дерева залежностей за допомогою DllAnalyzerService.

У представленій реалізації використовується підхід завантаження збірок безпосередньо в поточний домен додатку за допомогою методу Assembly.Load.

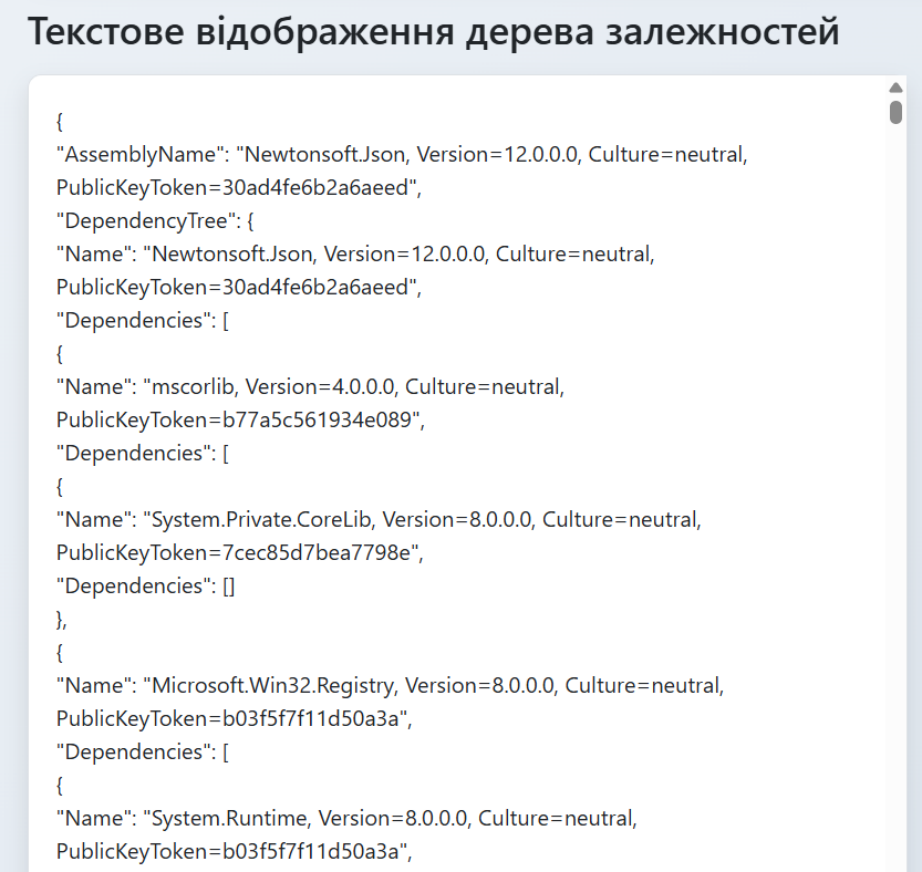

Рисунок 3.3 — Аналіз DLL-файлу з використанням DllAnalyzerService

Це простий і ефективний підхід для невеликих та середніх проєктів, але він має певні обмеження:

— збірки, завантажені в домен додатку, не можуть бути вивантажені до завершення роботи домену;

— конфлікти можуть виникати при завантаженні збірок з однаковими іменами, але різними версіями.

Для вирішення цих обмежень у більш складних сценаріях можна використовувати підхід з окремими доменами додатків або AssemblyLoadContext у .NET Core/.NET 5+.

Важливою особливістю розробленого механізму є обробка помилок та логування. Кожен крок процесу аналізу супроводжується детальним логуванням, що спрощує діагностику проблем та відлагодження. Крім того, механізм обробки помилок забезпечує стійкість системи до різних типів збірок та ситуацій:

— невалідні файли (не .NET збірки);

— відсутні залежності;

— циклічні залежності;

— інші помилки завантаження.

Результат аналізу представляється у вигляді деревовидної структури, де кожен вузол є об'єктом класу DependencyNode з іменем збірки та списком її залежностей. Ця структура легко серіалізується у формат JSON для передачі до компонента візуалізації.

В цілому, розроблений механізм забезпечує ефективний та надійний аналіз DLL файлів та їх залежностей, надаючи повну інформацію для подальшої візуалізації та аналізу. Використання можливостей рефлексії .NET дозволяє отримати детальну інформацію про структуру збірок без доступу до вихідного коду, що робить систему універсальною та гнучкою.

3.2.2 Механізми вилучення метаданих з бібліотек

Рефлексія є потужним механізмом платформи .NET, який дозволяє програмно досліджувати метадані, типи та члени збірок під час виконання програми. В розробленій вебплатформі для аналізу залежностей бібліотек коду рефлексія використовується для глибокого аналізу структури збірок, отримання інформації про типи, методи, властивості та інші елементи без доступу до вихідного коду.

Для роботи з рефлексією в проєкті реалізовано два ключові сервіси: ReflectionService та TypeAnalyzerService, які надають функціональність для завантаження збірок та аналізу їх внутрішньої структури.

Використання прапорців BindingFlags дозволяє гнучко налаштовувати пошук членів типу. У наведеному прикладі використовуються наступні прапорці:

— BindingFlags.Public — пошук публічних членів;

— BindingFlags.Instance — пошук екземплярних членів;

— BindingFlags.Static — пошук статичних членів.

Рисунок 3.4 ілюструє процес отримання метаданих через рефлексію.

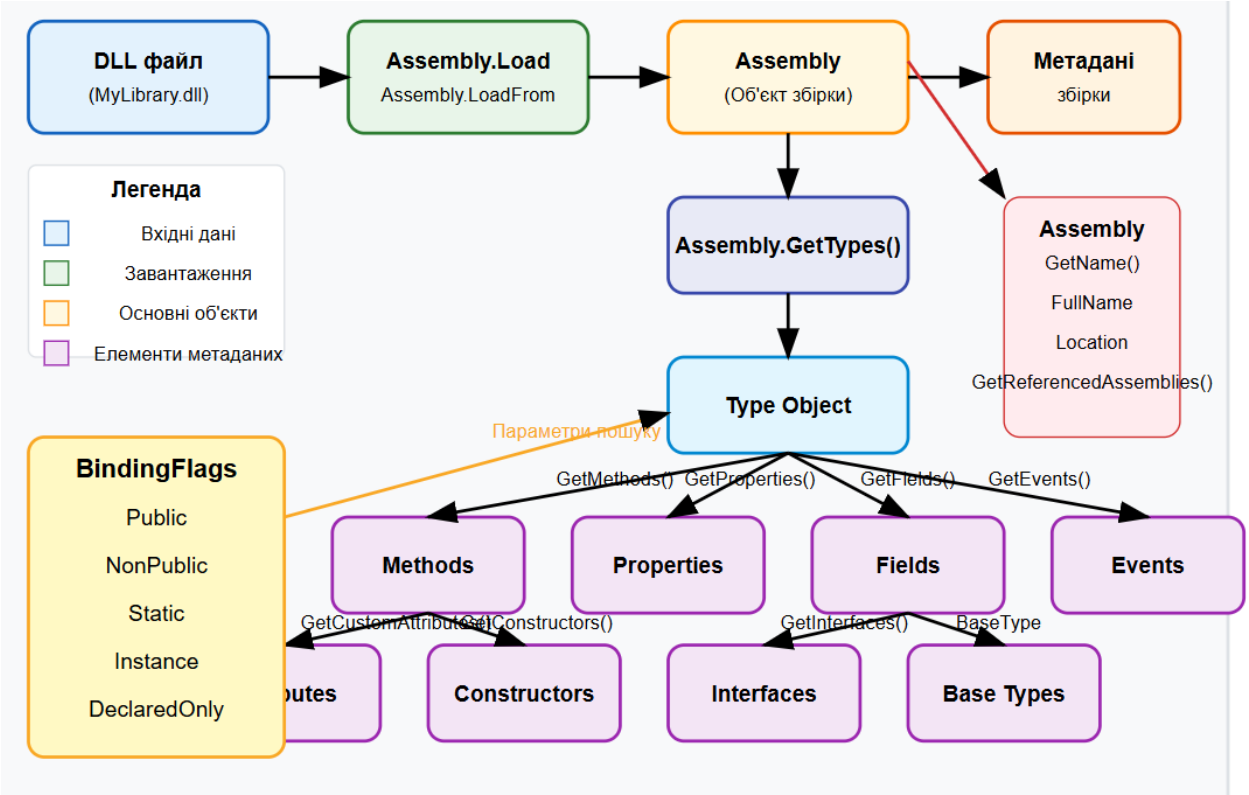

Рисунок 3.4 — Процес отримання метаданих через рефлексію

Ця комбінація забезпечує отримання як статичних, так і екземплярних публічних членів типу, що надає повне уявлення про його структуру.

Важливо відзначити, що наведена реалізація є спрощеною і може бути розширена для аналізу додаткових аспектів типів:

— інформація про поля (FieldInfo);

— інформація про події (EventInfo);

— інформація про атрибути (Attribute);

— інформація про генеричні параметри (Type.GetGenericArguments());

— інформація про базові класи та інтерфейси (Type.BaseType, Type.GetInterfaces()).

Застосування рефлексії для отримання метаданих бібліотек має деякі особливості та потенційні виклики:

Операції рефлексії є відносно повільними порівняно з прямим доступом до членів типів. Однак у контексті аналізу бібліотек це не є критичним, оскільки аналіз виконується одноразово, а не в режимі реального часу. Збірки, завантажені через

рефлексію, залишаються в пам'яті до завершення роботи домену додатку. Для великих проєктів це може призвести до значного споживання пам'яті.

Рефлексія дозволяє отримати доступ навіть до приватних членів типів, що може бути проблемою з точки зору безпеки. У наведеному прикладі використовуються лише публічні члени, що є безпечним підходом. При роботі з рефлексією можуть виникати різні винятки, такі як TypeLoadException, MethodAccessException тощо. Ефективна обробка цих винятків є важливою для забезпечення стабільності системи.

В цілому, розроблені сервіси ReflectionService та TypeAnalyzerService забезпечують ефективний механізм для завантаження збірок та аналізу їх внутрішньої структури, надаючи детальну інформацію для подальшої обробки та візуалізації. Використання можливостей рефлексії .NET дозволяє отримати повне уявлення про структуру бібліотек коду без доступу до їх вихідного коду, що робить систему універсальною та гнучкою.

3.2.3 Алгоритми побудови деревовидної структури залежностей

Ключовим компонентом розробленої вебплатформи є механізм побудови деревовидної структури залежностей, який дозволяє представити взаємозв'язки між бібліотеками коду у вигляді ієрархічного дерева. Цей механізм відіграє важливу роль у візуалізації та аналізі залежностей, надаючи розробникам зрозуміле та інтуїтивне представлення структури їхніх проєктів.

Для реалізації цього механізму в проєкті розроблено сервіс DependencyTreeBuilder, який відповідає за побудову дерева залежностей на основі аналізу .NET збірок (лістинг 3.4).

Лістинг 3.4 — Сервіс DependencyTreeBuilder

```
public class DependencyTreeBuilder : IDependencyTreeBuilder{
public DependencyNode BuildTree(
Assembly assembly, HashSet<string> visitedAssemblies = null!){
if (visitedAssemblies == null)
visitedAssemblies = [];
var assemblyFullName = assembly.FullName;
var node = new DependencyNode { Name = assemblyFullName! };
```

```
if (visitedAssemblies.Contains(assemblyFullName!))
return node;
visitedAssemblies.Add(assemblyFullName!);
var referencedAssemblies = assembly.GetReferencedAssemblies();
foreach (var refAssemblyName in referencedAssemblies){
try {
var refAssembly = Assembly.Load(refAssemblyName);
var childNode = BuildTree(refAssembly, visitedAssemblies);
node.Dependencies.Add(childNode);}
catch (Exception){
node.Dependencies.Add(new DependencyNode {
Name = refAssemblyName.FullName + " (не вдалося завантажити)" });
}}
return node;
}}
```

Метод BuildTree є основною функцією сервісу, яка рекурсивно будує дерево залежностей, починаючи з кореневої збірки. Процес побудови складається з кількох ключових етапів. Ініціалізація множини відвіданих збірок реалізується через параметр visitedAssemblies, який використовується для відстеження вже оброблених збірок, щоб уникнути циклічних залежностей та повторного аналізу. Якщо цей параметр не заданий, створюється новий порожній набір. Наступним кроком є створення кореневого вузла - для поточної збірки створюється об'єкт DependencyNode, який містить її повне ім'я. Потім виконується перевірка на циклічні залежності: якщо збірка вже була відвідана (її ім'я присутнє в visitedAssemblies), функція повертає лише базовий вузол без подальшого аналізу. Це є критичним для запобігання нескінченної рекурсії при наявності циклічних залежностей. Далі відбувається додавання збірки до відвіданих - поточна збірка додається до множини відвіданих, щоб уникнути її повторного аналізу в майбутньому.

Після цих підготовчих кроків виконується отримання залежностей - метод GetReferencedAssemblies() використовується для отримання списку всіх збірок, на які посилається поточна збірка. Потім відбувається рекурсивний аналіз залежностей: для кожної залежної збірки метод намагається завантажити збірку за допомогою Assembly.Load(), рекурсивно викликати BuildTree для цієї збірки та додати результуючий вузол до списку залежностей поточного вузла. Важливим аспектом є обробка помилок: якщо залежну збірку не вдається завантажити,

створюється спеціальний вузол з відповідним повідомленням про помилку. Це забезпечує стійкість системи до неповних або пошкоджених збірок. Завершальним етапом є повернення побудованого дерева: функція повертає кореневий вузол з усіма його дочірніми вузлами, що представляє повне дерево залежностей.

Такий підхід до побудови дерева дозволяє ефективно відображати навіть складні структури залежностей і має ряд важливих переваг. Запобігання циклічним залежностям забезпечується використанням множини відвіданих збірок, що запобігає нескінченним циклам при аналізі.

Рисунок 3.5 ілюструє приклад деревовидної структури залежностей, побудованої за допомогою описаного механізму.

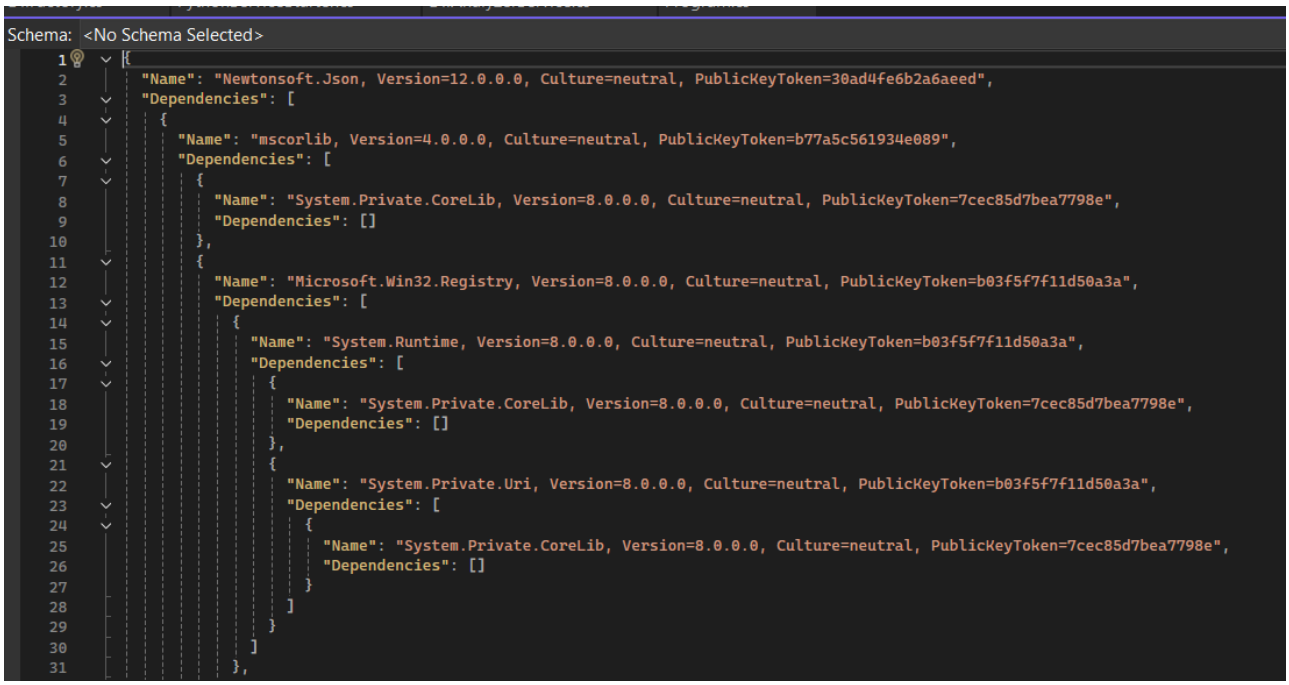

Рисунок 3.5 — Деревовидна структура залежностей

Побудоване дерево залежностей може бути використане для різних цілей. Воно забезпечує візуалізацію, оскільки дерево легко перетворюється у формат, придатний для візуалізації за допомогою графічних бібліотек, таких як Dash Cytoscape. Структура дерева дозволяє проводити аналіз залежностей — аналізувати прямі та транзитивні залежності, виявляти потенційні проблеми та оптимізувати структуру проєкту. Хоча механізм запобігає нескінченним циклам, він все ж дозволяє виявити наявність циклічних залежностей у проєкті. Також дерево може

бути використане для виявлення несумісностей між різними версіями однієї й тієї ж бібліотеки.

В цілому, розроблений механізм побудови деревовидної структури залежностей є одним з ключових компонентів вебплатформи, який забезпечує ефективний аналіз та візуалізацію взаємозв'язків між компонентами програмних систем. Використання принципів рекурсії та відстеження відвіданих вузлів дозволяє ефективно обробляти навіть складні структури з багатьма рівнями залежностей та потенційними циклами.

3.3 Реалізація підсистеми візуалізації даних

3.3.1 Розробка інтерактивного інтерфейсу на базі Dash

Важливою складовою розробленої вебплатформи є система візуалізації, яка забезпечує наочне та інтерактивне представлення структури залежностей бібліотек коду. Для реалізації візуалізації обрано бібліотеку Dash від Plotly, яка дозволяє створювати інтерактивні веб-додатки з використанням Python.

Основою системи візуалізації є Python-скрипт, який створює Dash-додаток для відображення деревовидної структури залежностей у вигляді інтерактивного графу. Цей додаток автоматично запускається при старті основного ASP.NET Core сервера та забезпечує візуалізацію залежностей.

Інтерактивний граф дозволяє користувачу досліджувати структуру залежностей, розгортаючи або згортуючи окремі гілки дерева. Кожен вузол графу містить інформацію про конкретну бібліотеку, зокрема її назву, версію та тип залежності. Крім того, реалізовано можливість взаємодії між інтерфейсом платформи ASP.NET Core та Dash-додатком через HTTP-запити, що забезпечує синхронізацію даних. Такий підхід дозволяє інтегрувати візуалізацію у загальний вебінтерфейс платформи, не порушуючи архітектурної цілісності проєкту. Це робить систему візуалізації не лише зручною для користувача, а й гнучкою для подальшого масштабування та модифікації.

Рисунок 3.6 демонструє інтерфейс розробленого Dash-додатку для візуалізації графу залежностей.

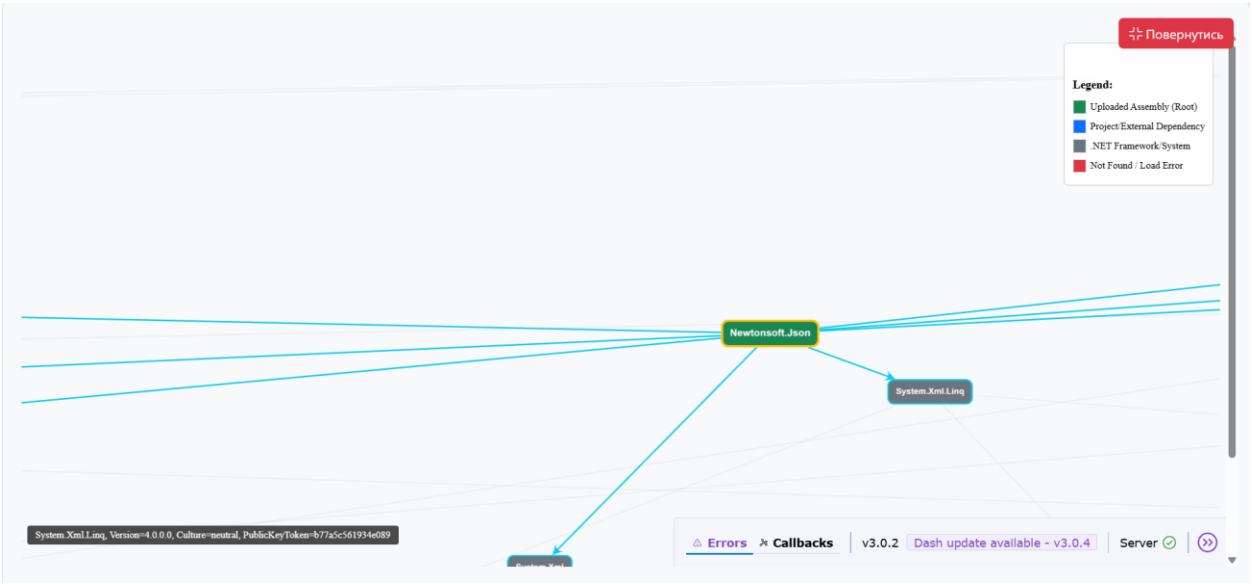

Рисунок 3.6 — Інтерфейс Dash-додатку для візуалізації графу залежностей

3.3.2 Алгоритми візуалізації графових структур

Інтерактивність є важливою особливістю розробленої системи візуалізації, що дозволяє користувачам ефективно досліджувати та аналізувати складні структури залежностей. Інтерактивні функції графу реалізовані через систему колбеків Dash, які забезпечують відгук на дії користувача.

Рисунок 3.7 демонструє результат роботи інтерактивних функцій графу — виділення вибраного вузла та його сусідів.

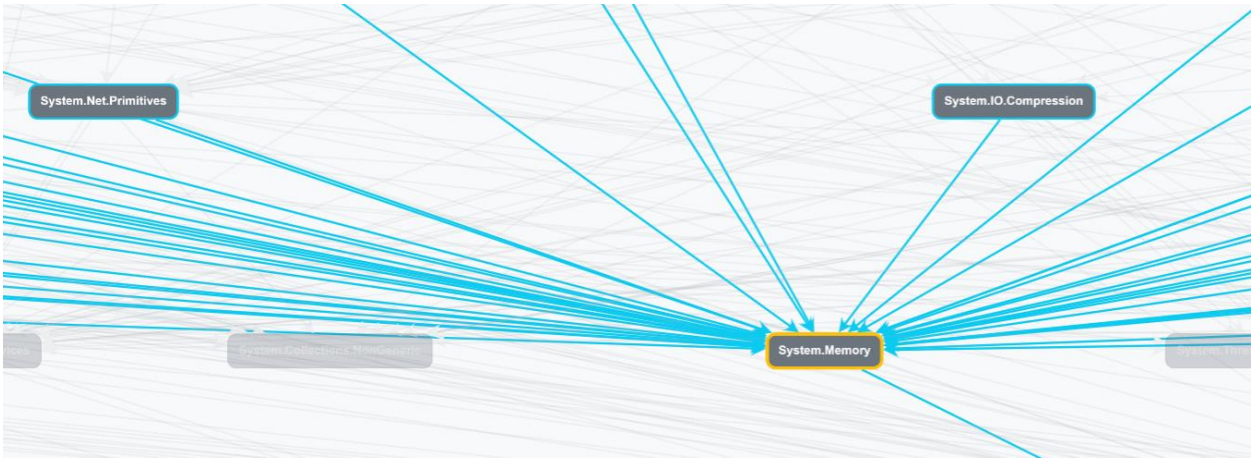

Рисунок 3.7 — Виділення вибраного вузла та його сусідів при взаємодії з графом

Реалізовані інтерактивні функції значно підвищують зручність використання системи та ефективність аналізу залежностей. Вони забезпечують інтуїтивно зрозумілий інтерфейс, який дозволяє користувачам легко досліджувати навіть складні структури залежностей.

Розроблений Python-модуль візуалізації є важливою складовою вебплатформи для аналізу залежностей. Він побудований на основі бібліотеки Dash, яка забезпечує створення інтерактивних веб-інтерфейсів з використанням Python, та компонента Dash Cytoscape для візуалізації графів.

Компонент завантаження та обробки даних включає функції get_dependency_data та build_elements, які відповідають за зчитування даних з JSON-файлу та їх перетворення у формат для візуалізації. Рекурсивний підхід забезпечує ефективну обробку вкладених структур будь-якої глибини.

Компонент візуалізації графу відповідає за налаштування зовнішнього вигляду та поведінки графу через стилі для різних типів вузлів та ребер. Використання алгоритму dagre забезпечує оптимальне розміщення вузлів для ієрархічних структур.

Компонент інтерактивності реалізований через систему колбеків Dash, які забезпечують відгук на дії користувача: оновлення графу при зміні даних, відображення підказок, виділення вузлів та їх сусідів при натисканні, відображення детальної інформації.

Використання функціонального реактивного програмування через систему колбеків Dash забезпечує ефективну обробку подій та оновлення інтерфейсу без необхідності прямого управління станом компонентів. Це підвищує надійність та спрощує підтримку коду.

В цілому, реалізація Python-модуля візуалізації є ефективним рішенням для відображення деревовидних структур залежностей, забезпечуючи високий рівень інтерактивності, наочність представлення даних та просту інтеграцію з іншими компонентами системи.

Архітектурно модуль складається з чотирьох основних компонентів, як показано на рисунку 3.8:

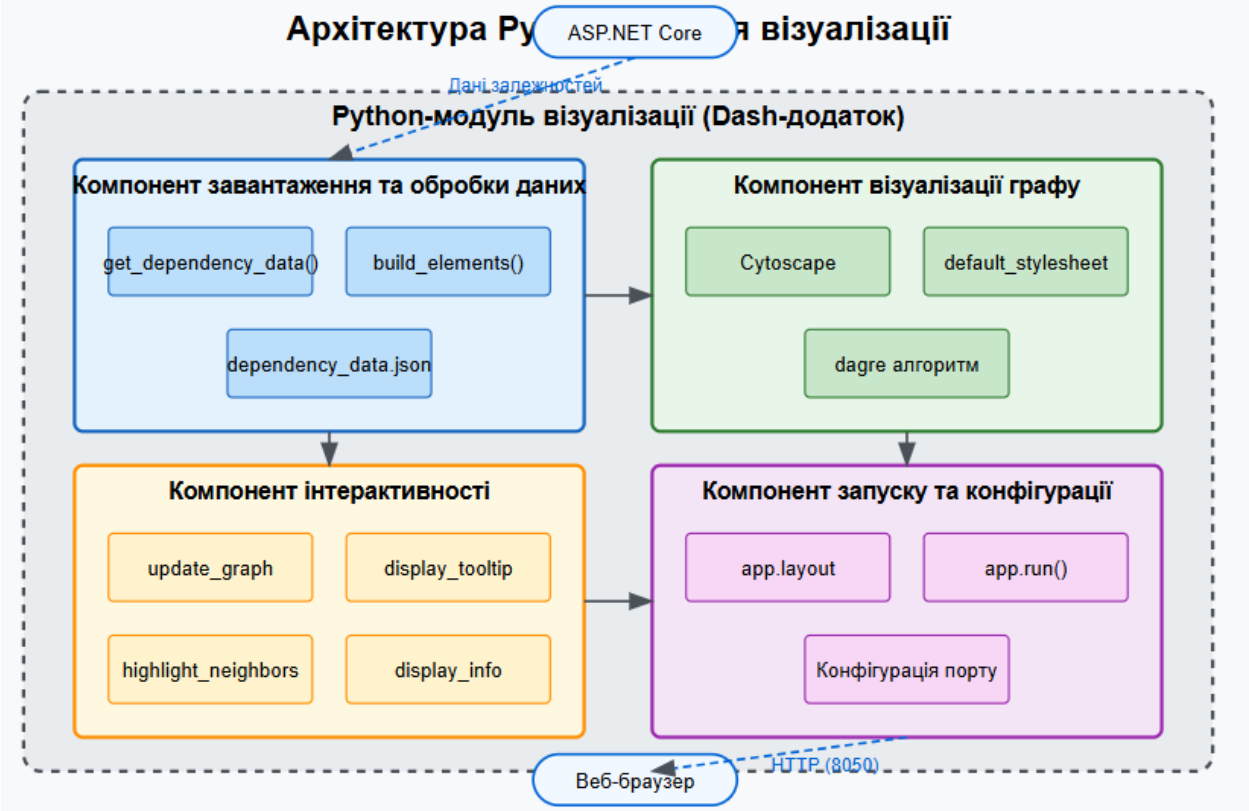

Рисунок 3.8 — Архітектура Python-модуля візуалізації

3.4 Оцінка ефективності та тестування розробленої системи

3.4.1 Демонстрація функціональності системи

Розроблена вебплатформа для аналізу залежностей бібліотек коду надає розробникам зручний і потужний інструмент для дослідження і візуалізації структури залежностей .NET збірок. Демонстрація ключової функціональності системи допомагає оцінити практичну цінність розробленого рішення.

Основний процес роботи з вебплатформою включає наступні кроки:

— завантаження DLL-файлу для аналізу через веб-інтерфейс;

— автоматичний аналіз структури залежностей збірки;

— візуалізація графу залежностей;

— інтерактивна робота з візуалізацією (виділення вузлів, отримання детальної інформації).

Результати аналізу відображаються у вигляді інтерактивного графу, де кореневий вузол (завантажена збірка) представлений зеленим кольором, проєктні

залежності — синім, системні збірки — сірим, а проблемні або невирішені залежності — червоним.

Вебплатформа забезпечує також різноманітні сценарії аналізу залежностей:

— аналіз складних ієрархій: Система ефективно обробляє збірки з великою кількістю вкладених залежностей, зберігаючи чітке і зрозуміле представлення структури;

— виявлення проблемних залежностей: Збірки, які не вдалося завантажити або проаналізувати, виділяються червоним кольором, що дозволяє швидко ідентифікувати проблемні місця;

— фокусування на окремих компонентах: При натисканні на вузол графу система автоматично виділяє цей вузол та його безпосередні залежності, затемнюючи решту графу, що спрощує аналіз складних структур.

Особливо корисною функцією є можливість отримання детальної інформації про вибрану збірку. При натисканні на вузол графу в нижній частині екрану відображається розширена інформація про збірку, включаючи її повне ім'я, версію та інші метадані.

В системі також реалізовано функціонал генерації тестових бібліотек з вихідного коду C#, що дозволяє користувачам експериментувати з різними структурами залежностей без необхідності створення зовнішніх проєктів.

Практична цінність системи демонструється на прикладі аналізу типової бібліотеки .NET з багаторівневою структурою залежностей. Наприклад, аналіз бібліотеки для роботи з базами даних може виявити непрямі залежності від криптографічних бібліотек або компонентів для серіалізації, що важливо для розуміння загальної архітектури системи та потенційних ризиків безпеки.

Вебплатформа успішно вирішує задачі, поставлені на початку розробки, забезпечуючи зручний і ефективний інструмент для аналізу залежностей бібліотек коду. Інтуїтивно зрозумілий інтерфейс, інтерактивна візуалізація та глибокий аналіз залежностей роблять систему корисною для широкого кола спеціалістів: від розробників програмного забезпечення до архітекторів та фахівців з інформаційної безпеки.

На рисунку 3.9 представлено головний екран вебплатформи після завантаження і аналізу DLL-файлу.

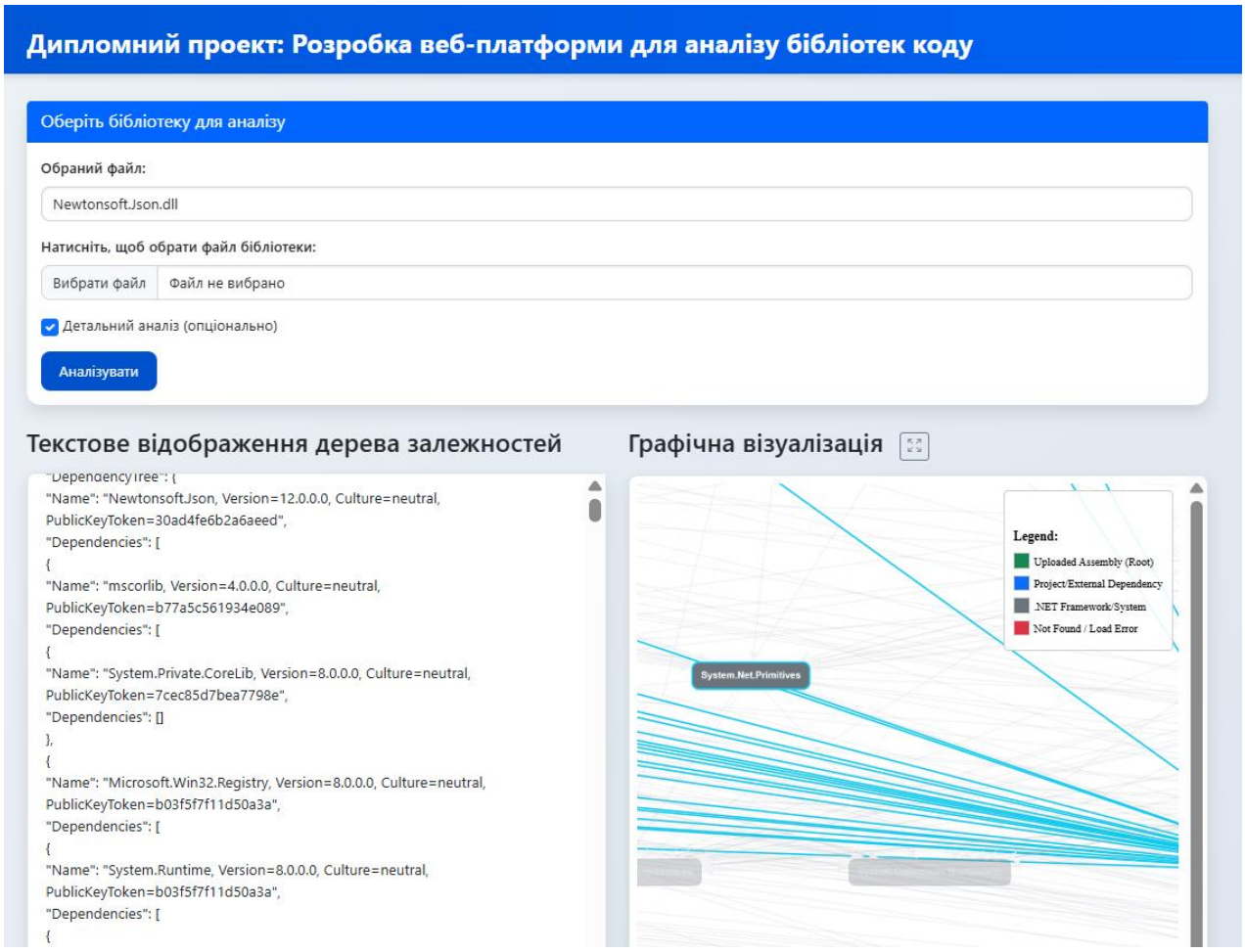

Рисунок 3.9 — Головний екран вебплатформи з результатами аналізу DLL-файлу

3.4.2 Оцінка досягнення цілей проєкту

Розробка вебплатформи для аналізу залежностей бібліотек коду ставила перед собою низку конкретних цілей, оцінка досягнення яких дозволяє визначити успішність проєкту в цілому.

Головною метою проєкту було створення інтегрованої вебплатформи для аналізу бібліотек коду з візуалізацією деревовидної структури залежностей. Ця мета повністю досягнута, розроблена система забезпечує повний цикл від завантаження DLL-файлу до інтерактивної візуалізації його залежностей.

Особливу увагу було приділено розробці інтуїтивно зрозумілого

користувацького інтерфейсу, який забезпечує простоту роботи з системою навіть для користувачів без глибоких технічних знань. Інтеграція різних технологій (ASP.NET Core та Python Dash) дозволила використати сильні сторони обох платформ, створивши ефективне та гнучке рішення.

Для більш детальної оцінки варто розглянути досягнення конкретних завдань, поставлених на початку розробки, що наведено у таблиці 3.2.

Таблиця 3.2 — Оцінка досягнення завдань проєкту

Завдання	Результат	Оцінка досягнення
Аналіз існуючих інструментів для аналізу залежностей	Проведено порівняльний аналіз існуючих інструментів, виявлено їх переваги та обмеження	Виконано повністю
Визначення оптимальної архітектури	Розроблено мікросервісну архітектуру з чітким розділенням відповідальності компонентів	Виконано повністю
Обґрунтування вибору технологій	Обрано оптимальний набір технологій: ASP.NET Core для аналізу, Python Dash для візуалізації	Виконано повністю
Реалізація API для завантаження та аналізу DLL	Розроблено REST API з документацією Swagger	Виконано повністю
Розробка системи візуалізації	Створено інтерактивний Dash-додаток з компонентом Cytoscape	Виконано повністю
Тестування системи	Проведено тестування на реальних проєктах різної складності	Виконано повністю

Реалізований проєкт відповідає всім поставленим вимогам та забезпечує наступні ключові можливості:

— аналіз DLL-файлів з використанням механізмів рефлексії для отримання детальної інформації про структуру залежностей;

— побудова деревовидної структури залежностей з урахуванням прямих та транзитивних зв'язків;

— інтерактивна візуалізація графу залежностей з можливістю

масштабування, виділення вузлів та їх сусідів;

— відображення детальної інформації про обрані компоненти;

— можливість генерації тестових DLL-файлів з вихідного коду.

Важливо зазначити, що розроблена архітектура відповідає сучасним підходам до проєктування програмного забезпечення, забезпечуючи гнучкість, масштабованість та простоту підтримки. Використання мікросервісного підходу з розділенням на ASP.NET Core та Python компоненти дозволило задіяти переваги обох платформ без надмірного ускладнення системи.

Таким чином, можна констатувати, що проєкт успішно досяг усіх поставлених цілей, створивши ефективний інструмент для аналізу залежностей бібліотек коду, який може бути використаний як у навчальних цілях, так і в реальних проєктах для поліпшення розуміння та оптимізації структури програмних систем.

ВИСНОВКИ

У ході виконання бакалаврської кваліфікаційної роботи розроблено вебплатформу для аналізу бібліотек коду з деревовидною структурою залежностей, що дозволяє автоматизувати процес дослідження структури та взаємозв'язків між компонентами програмних систем.

Основними результатами, отриманими під час виконання роботи, стали:

— проведення аналізу існуючих інструментів для аналізу залежностей бібліотек коду з виявленням їх переваг та недоліків;

— обґрунтування необхідності створення нового рішення, яке поєднує глибокий аналіз DLL-файлів із сучасною інтерактивною візуалізацією;

— вибір та застосування оптимального набору технологій: ASP.NET Core для серверної логіки та .NET-рефлексії, Python Dash і Cytoscape — для фронтенд-візуалізації графових структур.

Спроєктовано мікросервісну архітектуру з чітким розділенням відповідальності між компонентами, що забезпечує масштабованість і зручність супроводу. Розроблено REST API для завантаження та аналізу збірок, а також генерації тестових бібліотек, із документацією у форматі Swagger. Реалізовано повноцінний механізм рефлексивного аналізу без потреби у вихідному коді збірки.

На стороні клієнта створено інтерактивну візуалізацію графу залежностей з функціями масштабування, виділення вузлів та перегляду атрибутів. Передача даних між Python та ASP.NET Core модулями здійснюється у форматі JSON, що забезпечує простоту взаємодії без зайвих ускладнень у реалізації.

Розроблена платформа має низку переваг над аналогами:

— веб-доступність без встановлення стороннього ПЗ;

— висока інтерактивність при роботі з графом залежностей;

— підтримка як базового, так і глибокого рівня аналізу структури збірок;

— можливість генерації тестових бібліотек через веб-інтерфейс.

Практична цінність полягає у наданні зручного інструмента для розробників, архітекторів ПЗ та фахівців з ІБ, що дозволяє виявляти потенційні архітектурні та

безпекові проблеми на ранніх етапах розробки.

Подальші напрями розвитку платформи можуть включати:

— інтеграцію з GitHub API для автоматичного аналізу залежностей у пул-реквестах;

— збереження історії попередніх аналізів та підтримку перегляду змін у часі;

— візуальний редактор графу з можливістю ручної корекції чи моделювання залежностей;

— підтримку аналізу на рівні класів і методів;

— розширення підтримки інших мов програмування та платформ, зокрема Java, Python або JavaScript;

— впровадження аналізу безпеки з виявленням відомих вразливостей.

Розроблена система успішно вирішує поставлені задачі й формує потужну основу для подальшого розвитку інструментів аналізу залежностей бібліотек коду.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Шевчук В. В., Лисак Н. В. Сучасні технології візуалізації даних у веб-застосунках. Вісник НУ «Львівська політехніка». Інформаційні системи та мережі. 2022. № 11. С. 123-135.

2. Петренко О. О., Коваль Т. І. Аналіз залежностей бібліотек коду та їх вплив на безпеку програмного забезпечення. Кібербезпека: освіта, наука, техніка. 2023. № 4(20). С. 56-69.

3. Іванов М. М. Мікросервісна архітектура при розробці веб-застосунків: переваги та недоліки. Наукові записки НаУКМА. Комп'ютерні науки. 2022. Том 5. С. 89-97.

4. Ковальчук А. М., Дудник В. В. Використання рефлексії в C# для аналізу структури програмних компонентів. Вісник Вінницького політехнічного інституту. 2023. № 3. С. 43-52.

5. Український освітній центр Python. Візуалізація даних з Dash та Plotly. 2023. URL: https://www.pythoncenter.org.ua/visualization/dash-plotly/ (дата звернення: 15.02.2025).

6. Литвиненко О. П. Розробка RESTful API з використанням ASP.NET Core. ДонНУ імені Василя Стуса. 2022. URL: https://jcs.donnu.edu.ua/article/view/11243 (дата звернення: 10.02.2025).

7. Технічна документація ASP.NET Core. Microsoft Docs. 2023. URL: https://docs.microsoft.com/uk-ua/aspnet/core/ (дата звернення: 12.02.2025).

8. Офіційна документація Dash. Plotly. 2022. URL: https://dash.plotly.com/ (дата звернення: 05.02.2025).

9. Марчук А. Д. Інтеграція різнорідних компонентів у веб-додатках. ElectronicsAndCS. 2023. URL: https://www.electronics.cv.ua/articles/web-integration (дата звернення: 21.01.2025).

10. Бородін В. А. Аналіз інструментів для візуалізації графових структур даних. ТехноБлог. 2023. URL: https://tech-blog.com.ua/graph-visualization-tools/ (дата звернення: 15.03.2025).

11. METANIT. Керування залежностями в .NET Core. URL: https://metanit.com/sharp/aspnet5/2.1.php (дата звернення: 23.01.2025).

12. Python для аналізу даних. IT Education Academy. URL: https://itstep.academy/python-data-analysis/ (дата звернення: 18.01.2025).

13. Національна бібліотека України імені В. І. Вернадського. Основи проектування веб-систем. URL: http://www.nbuv.gov.ua/node/4324 (дата звернення: 10.01.2025).

14. Newman S. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, 2021. 590 p.

15. Albahari J., Albahari B. C# 10 in a Nutshell: The Definitive Reference. O'Reilly Media, 2022. 1000 p.

16. Franz M., Lopes C.T., Huck G., et al. "Cytoscape.js: a graph theory library for visualisation and analysis". Bioinformatics, 2016, 32(2): 309-311.

17. Microsoft. "System.Reflection Namespace". Microsoft Docs, 2023. https://docs.microsoft.com/en-us/dotnet/api/system.reflection

18. Plotly. "Dash Cytoscape Documentation". Plotly Docs, 2023. https://dash.plotly.com/cytoscape

19. Wang S., Khan N. "Graph Visualization Techniques for Web Analytics: A Systematic Literature Review". Journal of Web Engineering, 2022, 21(2): 189-230.

20. Mariano C., Leite L., Pereira T. et al. "Visualization Tools for Software Dependency Analysis: A Systematic Review". Information and Software Technology, 2023, 151: 106971.

ДОДАТОК А

Технічне завдання

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ
проф., д.т.н.. Азаров О.Д.
21 березня 2025 р.

ТЕХНІЧНЕ ЗАВДАННЯ
на виконання бакалаврської кваліфікаційної роботи
«Вебплатформа для аналізу бібліотек коду з деревовидною структурою залежностей»

08-54.БКР.004.00.000 ТЗ

Науковий керівник: к.т.н., доцент каф. ОТ
_____Дудник О. В.

Студентка групи КІ-23мсз
_____Лотоцька А. О.

м. Вінниця — 2025

1 Підстава для використання бакалаврської кваліфікаційної роботи (БКР)

1.1 Актуальність розробки полягає у необхідності впровадження ефективних інструментів аналізу залежностей бібліотек коду через зростання складності сучасних програмних систем та розширення екосистеми .NET. Розробка спеціалізованої вебплатформи дасть можливість автоматизувати процес виявлення та візуалізації залежностей, і, як наслідок, підвищити якість розробки програмного забезпечення, спростити процес рефакторингу та мінімізувати ризики при оновленні компонентів.

1.2 Головним завданням розробки є створення сучасної, інтерактивної вебплатформи для аналізу залежностей бібліотек коду з використанням мікросервісної архітектури, технологій ASP.NET Core, механізмів рефлексії та візуалізації даних за допомогою Python Dash.

1.3 Наказ про затвердження теми бакалаврської кваліфікаційної роботи від 21 березня 2025 р. №97.

2 Мета і призначення БКР

2.1 Мета проєкту полягає у створенні інтегрованої вебплатформи для аналізу бібліотек коду з візуалізацією деревовидної структури залежностей, що забезпечує високий рівень інтерактивності та зручності використання, через проєктування мікросервісної архітектури з використанням ASP.NET Core та Swagger, реалізацію механізмів рефлексії для аналізу DLL файлів, розробку інтерактивної візуалізації на основі Python Dash Cytoscape, а також організацію ефективного обміну даними між компонентами системи.

2.2 Призначення розробки полягає у виконанні бакалаврської кваліфікаційної роботи, за результатами якої буде створено ефективний інструмент для аналізу залежностей бібліотек коду, що дозволить розробникам програмного забезпечення краще розуміти та оптимізувати структуру своїх проєктів.

3 Вихідні дані для виконання БКР

3.1 Аналіз існуючих інструментів та сучасних технологій для аналізу залежностей бібліотек коду, їх переваги, недоліки та можливості масштабування.

3.2 Огляд сучасних підходів до архітектури вебдодатків, включаючи мікросервіси, REST API, документацію з використанням Swagger та OpenAPI.

3.3 Дослідження технологій аналізу .NET-збірок, включаючи рефлексію, обробку метаданих та побудову ієрархії залежностей у вигляді дерева.

3.4 Визначення вимог до системи візуалізації, огляд бібліотек графів і методів інтеграції Python із .NET-додатками.

3.5 Розробка логічної структури та архітектури вебплатформи з модулями для аналізу, побудови та візуалізації залежностей.

3.6 Проведення оцінки ефективності, тестування вебплатформи на реальних проєктах різної складності, аналіз отриманих результатів.

4 Вимоги до виконання БКР

Головна вимога полягає в створенні ефективної та зручної вебплатформи для аналізу залежностей бібліотек коду, яка забезпечить:

— завантаження та аналіз DLL файлів з використанням механізмів рефлексії .NET;

— побудову деревовидної структури залежностей з можливістю виявлення проблемних компонентів;

— інтерактивну візуалізацію графу залежностей за допомогою Python Dash Cytoscape;

— взаємодію між ASP.NET Core та Python компонентами через обмін даними у форматі JSON;

— деталізовану візуалізацію атрибутів та методів кожної бібліотеки у складі графу залежностей;

— розпізнавання типів зв’язків між компонентами (напр. успадкування, композиція) під час аналізу через рефлексію.

— документування API за допомогою Swagger для зручності розробки та інтеграції.

5 Етапи БКР та очікувані результати

Етапи розробки БКР та очікувані результати наведено у Таблиці А.1.

Таблиця А.1 — Етапи БКР

№ етапу	Назва етапу	Термін виконання		Очікувані результати
		Початок	Кінець	
1	Постановка задачі та техніко економічне обґрунтування	12.03.25	16.03.25	Розділ 1
2	Аналіз існуючих інструментів для аналізу залежностей	17.03.25	28.03.25	Розділ 1
3	Аналіз та обґрунтування вибору технологій	29.03.25	09.04.25	Розділ 2
4	Проєктування та розробка архітектури вебплатформи	10.04.25	19.04.25	Розділ 3
5	Реалізація аналізатора залежностей та системи візуалізації	20.04.25	26.04.25	Розділ 4
6	Підготовка тезових матеріалів для публікації	27.04.25	28.04.25	Тези
7	Оформлення пояснювальної записки та демонстраційної презентації	29.04.25	17.05.25	ПЗ, презентація
8	Підготовка супровідної документації та перевірка на плагіат	18.05.25	26.05.25	ПЗ, презентація

6 Матеріали, що подаються до захисту БКР

До захисту подаються: пояснювальна записка БКР, ілюстративні матеріали (схеми архітектури системи, діаграми компонентів), програмна реалізація вебплатформи, протокол попереднього захисту БКР на кафедрі, відгук наукового керівника, анотації до БКР українською та іноземною мовами, довідка про відповідність оформлення БКР чинним вимогам.

7 Порядок контролю виконання та захисту БКР

Виконання етапів БКР контролюється науковим керівником к.т.н., доцент каф. ОТ Дудник О. В. згідно зі встановленими термінами. Захист БКР відбувається на засіданні Екзаменаційної комісії, затвердженої наказом ректора ВНТУ.

8 Вимоги до оформлювання та порядок виконання БКР

8.1 При оформлюванні БКР використовуються:

— ДСТУ 3008 : 2015 «Звіти в сфері науки і техніки. Структура та правила оформлювання»;

— ДСТУ 8302 : 2015 «Бібліографічні посилання. Загальні положення та правила складання»;

— методичні вказівки до виконання бакалаврських кваліфікаційних робіт для студентів спеціальності 123 «Комп'ютерна інженерія» (освітня програма «Комп'ютерна інженерія») — кафедра обчислювальної техніки, ВНТУ 2023.

Порядок виконання БКР викладено в «Положення про кваліфікаційні роботи на першому (бакалаврському) рівні вищої освіти СУЯ ВНТУ-03.02.02-П.001.01:21»

ДОДАТОК Б

ПРОТОКОЛ

ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

НА НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи:

«Вебплатформа для аналізу бібліотек коду з деревовидною структурою залежностей»

Тип роботи: Бакалаврська кваліфікаційна робота
(БДР, МКР)

Підрозділ кафедра обчислювальної техніки
(кафедра, факультет)

Показники звіту подібності StrikePlagiarism

Оригінальність 99,5% Схожість 0,5%

Аналіз звіту подібності (відмітити потрібне):

✓ Запозичення, виявлення у роботі, оформлені коректно і не містять ознак плагіату.

☐ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її виконання автором. Роботу направити на розгляд експертної комісії кафедри.

☐ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку _____ Захарченко С.М.
(підпис) (прізвище, ініціали)

Ознайомлені з повним звітом подібності, який був згенерований системою StrikePlagiarism щодо роботи.

Автор роботи _____ Лотоцька А.О.
(підпис) (прізвище, ініціали)

Керівник роботи _____ Дудник О. В.
(підпис) (прізвище, ініціали)

ДОДАТОК В

Ілюстративна частина

ВЕБПЛАТФОРМА ДЛЯ АНАЛІЗУ БІБЛІОТЕК КОДУ З ДЕРЕВОВИДНОЮ СТРУКТУРОЮ ЗАЛЕЖНОСТЕЙ

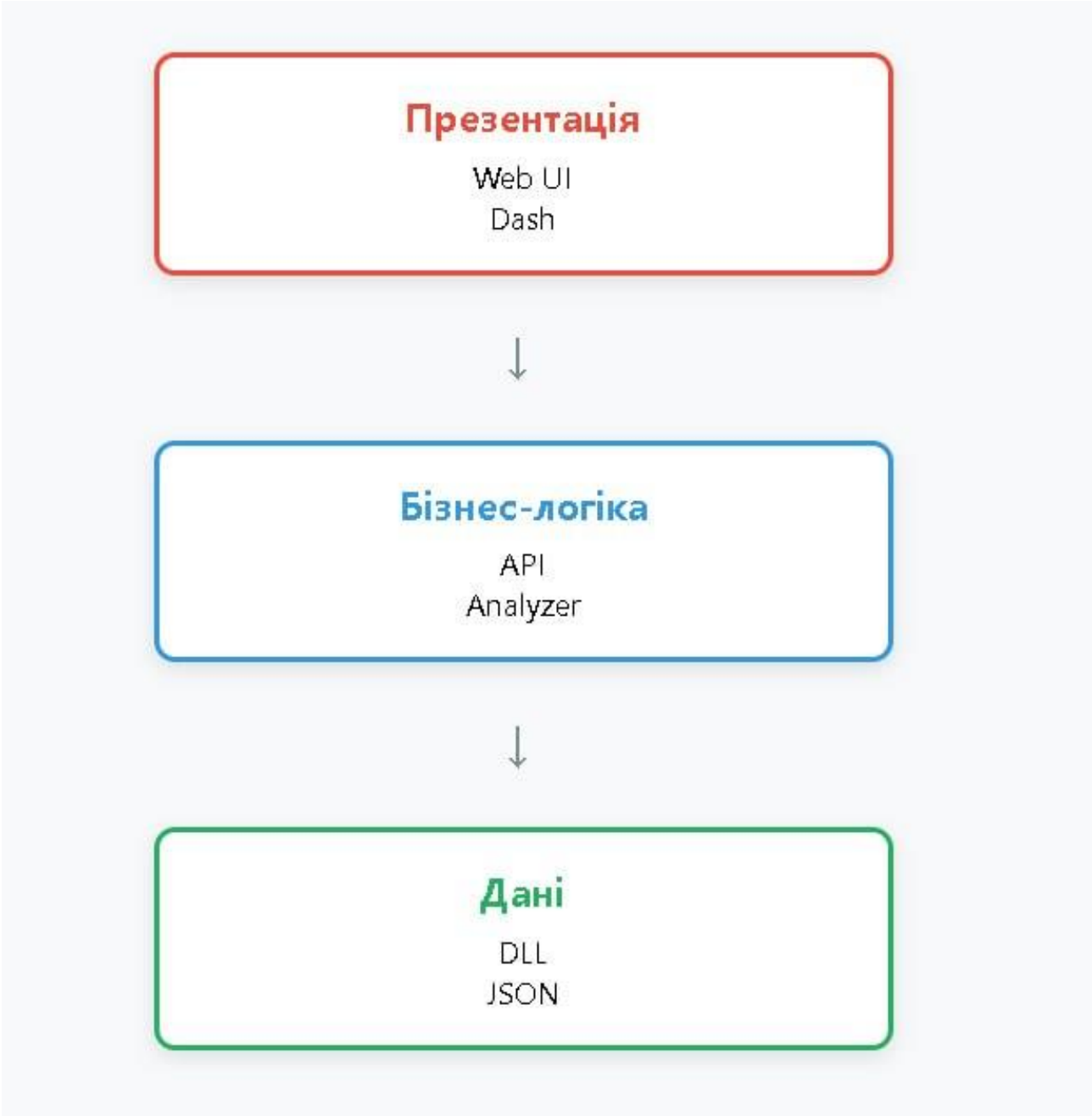

Рисунок В.1 – Архітектурна схема вебплатформи

Рисунок В.2 – Діаграма взаємодії компонентів програми

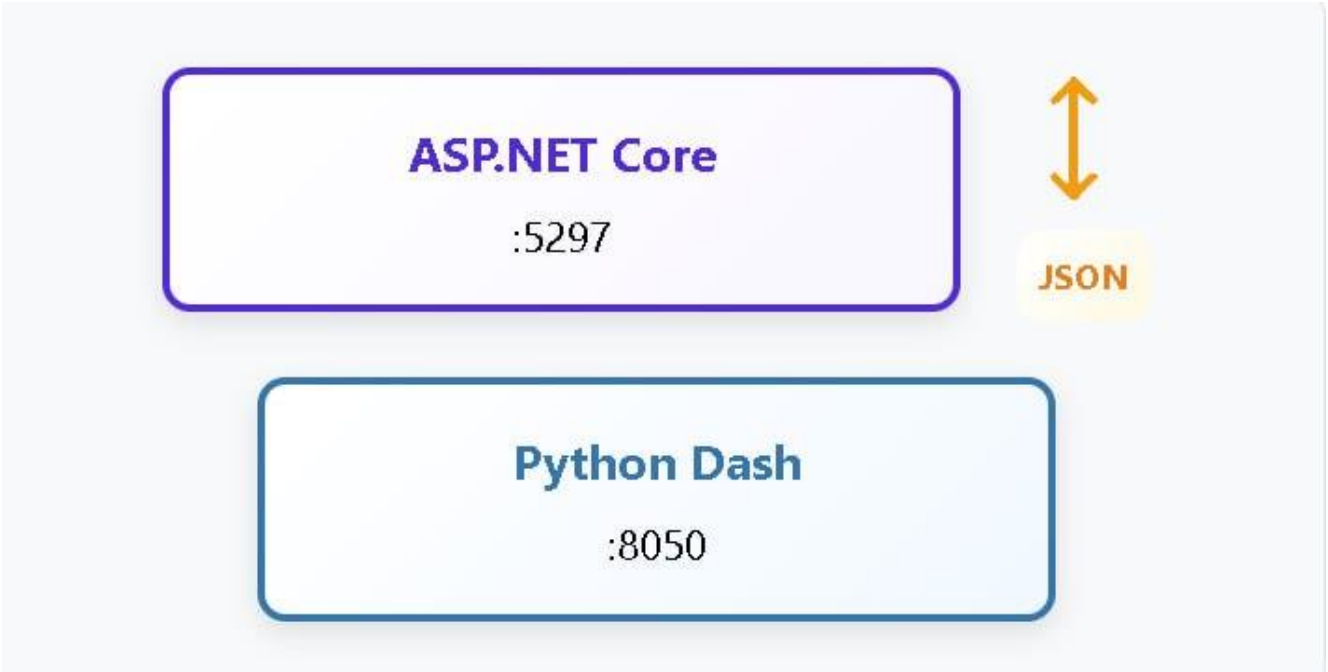

Рисунок В.3 – Схема інтеграції ASP.NET Core з Python Dash

ДОДАТОК Г

Лістинги основних класів вебплатформи

Лістинг Г.1 — Код класу AnalysisController.cs

```
using Microsoft.AspNetCore.Mvc;
using DependencyAnalyzer.Services;
using DependencyAnalyzer.Models;
using DependencyAnalyzer.Interfaces;
using Newtonsoft.Json;

namespace DependencyAnalyzer.Controllers{
 [ApiController]
 [Route("api/[controller]")]
public class AnalysisController : ControllerBase{
private readonly
IAssemblyAnalysisService _assemblyAnalysisService;
private readonly ILogger<AnalysisController> _logger;
private readonly IDllAnalyzerService _dllAnalyzerService;
public                    AnalysisController(IAssemblyAnalysisService
assemblyAnalysisService,         ILogger<AnalysisController>        logger,
IDllAnalyzerService dllAnalyzerService){
_assemblyAnalysisService = assemblyAnalysisService;
_logger = logger;
_dllAnalyzerService = dllAnalyzerService;}
 [HttpPost("upload")]
 [Consumes("multipart/form-data")]
 [ProducesResponseType(typeof(object),
 StatusCodes.Status200OK)]
 [ProducesResponseType(typeof(string),
 StatusCodes.Status400BadRequest)]
 [ProducesResponseType(typeof(string),
 StatusCodes.Status500InternalServerError)]
public IActionResult UploadDll([FromForm] FileUploadModel model){
if (model == null ||
model.File == null || model.File.Length == 0){
return BadRequest("File not selected or empty.");}
string fileName = Path.GetFileName(model.File.FileName);
_logger.LogInformation("Received file
upload request for: {FileName}", fileName);
byte[] fileBytes;
try{
using (var memoryStream = new MemoryStream()){
model.File.CopyTo(memoryStream);
fileBytes = memoryStream.ToArray();}
_logger.LogDebug("File {FileName} read into memory ({Length} bytes).",
fileName, fileBytes.Length);}
catch (Exception ex){
```

```
     _logger.LogError(ex, "Error reading uploaded file {FileName} into
memory.", fileName);
     return  StatusCode(StatusCodes.Status500InternalServerError,  $"Error
reading file: {ex.Message}");}
     try{
     object resultData;
     DependencyNode dependencyTree;
     if (model.Detailed){
     _logger.LogInformation("Performing detailed analysis for {FileName}",
fileName);
     AnalysisReport                              report                              =
_assemblyAnalysisService.AnalyzeAssembly(fileBytes, fileName);
     resultData = report;
     dependencyTree = report.DependencyTree;}
     else {
     _logger.LogInformation("Performing standard dependency tree analysis
for {FileName}", fileName);
     DependencyNode  tree  =  _dllAnalyzerService.AnalyzeAssembly(fileBytes,
fileName);
     resultData = tree;
     dependencyTree = tree;}
     if (dependencyTree == null){
     _logger.LogWarning("Dependency  tree  is  null  for  {FileName}.  Cannot
write to JSON file.", fileName);}
     else{
     string treeJson =
     JsonConvert.SerializeObject(dependencyTree,
     Newtonsoft.Json.Formatting.Indented);
     string  jsonFilePath  =  Path.Combine(Directory.GetCurrentDirectory(),
"dependency_data.json");
     _logger.LogInformation("Attempting to write dependency tree data to:
{JsonFilePath}", jsonFilePath);
     try {
     System.IO.File.WriteAllText(jsonFilePath, treeJson);
     _logger.LogInformation("Successfully  wrote  dependency  tree  data  to:
{JsonFilePath}", jsonFilePath);}
     catch (IOException ex){
     _logger.LogError(ex,   "IO   Error   writing   dependency   data   to
{JsonFilePath}. Check permissions or if file is locked.", jsonFilePath);}
     catch (Exception ex){
     _logger.LogError(ex,  "Failed  to  write  dependency  tree  data  to:
{JsonFilePath}", jsonFilePath);
     }}
     string    responseJson    =    JsonConvert.SerializeObject(resultData,
Newtonsoft.Json.Formatting.Indented);
     return Content(responseJson, "application/json");}
     catch (ArgumentException ex){
     _logger.LogWarning("Analysis  failed  for  {FileName}:  {ErrorMessage}",
fileName, ex.Message);
     return BadRequest($"Analysis Error: {ex.Message}");}
     catch (Exception ex){
```

```
     _logger.LogError(ex, "Critical error during analysis of {FileName}.",
fileName);
     return  StatusCode(StatusCodes.Status500InternalServerError,  $"Error
during analysis: {ex.Message}"); }}}}
```

Лістинг Г.2 — Код класу LibraryController.cs

```
     using DependencyAnalyzer.DllFactory.Interfaces;
     using DependencyAnalyzer.Models;
     using Microsoft.AspNetCore.Mvc;
     namespace DependencyAnalyzer.Controllers{
         [ApiController]
         [Route("api/[controller]")]
         public class LibraryController : ControllerBase
         {
     private readonly IDllFactory _dllFactory;
     private readonly ILogger<LibraryController> _logger;
     public            LibraryController(IDllFactory           dllFactory,
ILogger<LibraryController> logger)
     {
     _dllFactory = dllFactory;
     _logger = logger;
     }
     /// <summary>
     /// Ендпоінт для генерації DLL із заданого вихідного коду.
     /// </summary>
     /// <param name="request">Об'єкт LibraryGenerationRequest, що містить
C#-код і ім'я вихідного файлу.</param>
     ///   <returns>Шлях   до   згенерованої   DLL   або   повідомлення   про
помилку.</returns>
      [HttpPost("generate")]
      [ProducesResponseType(typeof(object),  StatusCodes.Status200OK)]  //
Успішна відповідь
      [ProducesResponseType(typeof(string),
StatusCodes.Status400BadRequest)] // Помилка запиту або компіляції
      [ProducesResponseType(typeof(string),
StatusCodes.Status500InternalServerError)] // Внутрішня помилка сервера
     public    async    Task<IActionResult>    GenerateLibrary([FromBody]
LibraryGenerationRequest request){
     if (request == null || string.IsNullOrWhiteSpace(request.SourceCode)){
     return BadRequest("Вихідний код не може бути порожнім.");}
     // Додамо перевірку імені файлу
     if        (string.IsNullOrWhiteSpace(request.OutputDllName)        ||
request.OutputDllName.IndexOfAny(Path.GetInvalidFileNameChars()) >= 0){
     return    BadRequest($"Некоректне    ім'я    вихідного    файлу    DLL:
'{request.OutputDllName}'.");}
     // Переконуємося, що розширення .dll
     if                            (!request.OutputDllName.EndsWith(".dll",
StringComparison.OrdinalIgnoreCase)){
     request.OutputDllName += ".dll";
```

```
    }
    try{
    _logger.LogInformation("Отримано запит на генерацію DLL: {DllName}",
request.OutputDllName);
    var dllPath = await _dllFactory.CompileDllAsync(request.SourceCode,
request.OutputDllName);
    _logger.LogInformation("DLL успішно згенеровано: {DllPath}", dllPath);
    // Повертаємо об'єкт з шляхом
    return Ok(new { GeneratedDllPath = dllPath });
    }
    catch (InvalidOperationException ex) // Ловимо помилку компіляції{
    _logger.LogError(ex, "Помилка компіляції при генерації DLL
'{DllName}'", request.OutputDllName);
    // Повертаємо помилку компіляції як BadRequest
    return BadRequest($"Помилка компіляції: {ex.Message}");}
    catch (Exception ex) // Інші неочікувані помилки {
    _logger.LogError(ex, "Неочікувана помилка при генерації DLL
'{DllName}'", request.OutputDllName);
    return StatusCode(StatusCodes.Status500InternalServerError,
$"Внутрішня помилка сервера: {ex.Message}"); }}}}
```

Лістинг Г.3 — Код класу Python-візуалізації app.py

```
    # -*- coding: utf-8 -*-
    import dash
    import dash_cytoscape as cyto
    from dash import html, dcc, Output, Input, State, callback, no_update
    import json
    import os
    import time

    cyto.load_extra_layouts()

    script_dir = os.path.dirname(os.path.abspath(__file__))
    parent_dir_of_csharp_project =
os.path.abspath(os.path.join(script_dir, '..'))
    actual_data_folder = os.path.join(parent_dir_of_csharp_project,
"DependencyAnalyzer")
    default_file_path = os.path.join(actual_data_folder,
"dependency_data.json")

    #print(f"Script directory determined as: {script_dir}")
    #print(f"Parent directory of C# project calculated as:
{parent_dir_of_csharp_project}")
    #print(f"Actual folder containing the data file identified as:
{actual_data_folder}")
    #print(f"--> Expected data file path: {default_file_path}")

    initial_elements = [
```

```
        {'data': {'id': 'placeholder', 'label': 'Upload a DLL for
analysis...'}}
    ]

    # --- Оновлено default_stylesheet ---
    # Видалено '.faded' клас, бо ми застосовуємо opacity напряму.
    # Залишено визначення стилів для класів підсвічування, хоча колбек їх
зараз не призначає,
    # але вони можуть бути корисні в майбутньому або для ясності.
    default_stylesheet = [
        {
            'selector': 'node',
            'style': {
                'background-color': '#0d6efd', 'label': 'data(label)',
                'text-valign': 'center', 'text-halign': 'center',
                'color': 'white', 'font-size': '11px', 'font-weight':
'bold',
                'shape': 'roundrectangle', 'width': 'label', 'height':
'label',
                'padding': '10px', 'border-width': 1, 'border-color':
'#0a58ca',
                'opacity': 0.9, 'transition-property': 'background-color,
opacity, border-color',
                'transition-duration': '0.3s'
            }
        },
        {
            'selector': '.level-0',
            'style': { 'background-color': '#198754', 'border-color':
'#146c43', 'font-size': '13px', 'opacity': 1 }
        },
        {
            'selector':   'node[id^="System."],   node[id^="Microsoft."],
node[id^="mscorlib"], node[id^="netstandard"]',
            'style': { 'background-color': '#6c757d', 'border-color':
'#5a6268' }
        },
        {
            'selector':    'node[label*="(failed"],    node[label*="(not
found"]',
            'style': { 'background-color': '#dc3545', 'border-color':
'#b02a37', 'shape': 'octagon', 'color': 'white' }
        },
         {
            'selector': '.selected-node-highlight', # Визначення стилю
класу (на майбутнє)
            'style': { 'border-width': 3, 'border-color': '#ffc107',
'opacity': 1 }
         },
         {
```

```
            'selector': '.neighbor-node-highlight', # Визначення стилю
класу (на майбутнє)
            'style': { 'border-width': 2, 'border-color': '#0dcaf0',
'opacity': 1 }
         },
         # Видалено стиль для '.faded', бо він застосовується напряму
        {
            'selector': 'edge',
            'style': {
                'width': 1.5, 'line-color': '#adb5bd',
                'target-arrow-color': '#adb5bd', 'target-arrow-shape':
'triangle-backcurve',
                'curve-style': 'bezier', 'arrow-scale': 1.5, 'opacity':
0.7,
                'transition-property': 'line-color, opacity', 'transition-
duration': '0.3s'
            }
        },
         {
             'selector': '.highlighted-edge', # Визначення стилю класу (на
майбутнє)
             'style': { 'line-color': '#0dcaf0', 'target-arrow-color':
'#0dcaf0', 'opacity': 1, 'width': 2 }
         },
         # Видалено стиль для '.edge.faded'
    ]

    initial_placeholder_stylesheet = [
        {
            'selector': '#placeholder',
            'style': {
                'background-color': '#cccccc', 'label': 'data(label)',
                'text-valign': 'center', 'text-halign': 'center',
                'color': '#000', 'font-size': '14px',
                'width': '150px', 'height': '50px', 'shape': 'rectangle'
            }
        }
    ]

    def build_elements(data, parent_id=None, level=0):
        elements = []
        if not data or 'Name' not in data:
             return []
        node_id = data["Name"]
        label = node_id.split(',')[0]
        elements.append({
            "data": { "id": node_id, "label": label, "full_name": node_id
},
            "classes": f"level-{level}"
         })
        if parent_id is not None:
```

```
            edge_id = f"edge_{parent_id}_to_{node_id}"
            elements.append({ "data": { "id": edge_id, "source": parent_id,
"target": node_id } })
        dependencies = data.get("Dependencies", [])
        if isinstance(dependencies, list):
            for dep in dependencies:
                elements.extend(build_elements(dep, node_id, level + 1))
        return elements

    def get_dependency_data(file_path):
        if not os.path.exists(file_path) or os.path.getsize(file_path) ==
0:
            print(f"File not found or empty: {file_path}")
            return None
        time.sleep(0.1)
        with open(file_path, "r", encoding="utf-8") as f:
            try:
                content = f.read()
                if not content.strip():
                    print(f"File contains only whitespace or is empty:
{file_path}")
                    return None
                data = json.loads(content)
                if not data or 'Name' not in data:
                    print(f"Invalid JSON structure or missing 'Name' in
root: {file_path}")
                    return None
                return data
            except json.JSONDecodeError as e:
                print(f"Error decoding JSON from file {file_path}: {e}")
                return None
            except Exception as e:
                print(f"Unexpected error reading file {file_path}: {e}")
                return None

    def create_legend_item(color, text):
        return html.Div([
            html.Div(style={'width':    '15px',    'height':    '15px',
'backgroundColor': color,
                            'display':   'inline-block',   'marginRight':
'5px', 'verticalAlign': 'middle',
                            'border': '1px solid #ccc'}),
            html.Span(text,  style={'fontSize':  '12px',  'verticalAlign':
'middle'})
        ], style={'marginBottom': '5px'})

    legend = html.Div([
        html.H6("Legend:",  style={'fontSize':  '14px',  'marginBottom':
'8px'}),
        create_legend_item('#198754', "Uploaded Assembly (Root)"),
        create_legend_item('#0d6efd', "Project/External Dependency"),
```

```
        create_legend_item('#6c757d', ".NET Framework/System"),
        create_legend_item('#dc3545', "Not Found / Load Error")
    ], style={
        'position': 'absolute', 'top': '15px', 'right': '15px',
        'backgroundColor': 'rgba(255, 255, 255, 0.9)',
        'padding': '10px', 'borderRadius': '5px', 'border': '1px solid
#ccc',
        'zIndex': 10
    })

    app = dash.Dash(__name__, suppress_callback_exceptions=True)

    app.layout = html.Div([
        dcc.Interval(id="interval-component",              interval=2000,
n_intervals=0),
        dcc.Store(id='last-mod-time-store', data=-1),
        dcc.Store(id='current-elements-store', data=initial_elements),
        html.Div([ # Wrapper for graph and legend
            html.P(id='cytoscape-tooltip', style={
                'position': 'absolute', 'bottom': '15px', 'left': '15px',
                'backgroundColor': 'rgba(0, 0, 0, 0.7)', 'color': 'white',
                'padding': '5px 10px', 'borderRadius': '3px', 'fontSize':
'12px',
                'zIndex': 10, 'display': 'none'
            }),
            legend,
            cyto.Cytoscape(
                id='cytoscape-deps',
                elements=initial_elements,
                layout={
                    'name': 'dagre', 'rankDir': 'TB', 'spacingFactor': 1.5,
                    'nodeSpacing': 90, 'edgeSep': 15, 'rankSep': 120,
'padding': 30
                },
                style={'width': '100%', 'height': '700px', 'position':
'relative'},
                stylesheet=default_stylesheet                          +
initial_placeholder_stylesheet,
                minZoom=0.15,      maxZoom=3.0,      zoomingEnabled=True,
userZoomingEnabled=True,
                panningEnabled=True,             userPanningEnabled=True,
boxSelectionEnabled=True
            )
        ]),
        html.Div(id='node-info-display', style={
            'padding': '15px', 'marginTop': '10px', 'border': '1px solid
#ddd',
            'borderRadius':    '5px',    'backgroundColor':    '#f9f9f9',
'minHeight': '50px',
            'fontFamily': 'monospace', 'fontSize': '13px', 'wordWrap':
'break-word'
```

```
        })
    ])


    @callback(
        Output("cytoscape-deps", "elements"),
        Output("current-elements-store", "data"),
        Output("cytoscape-deps", "stylesheet", allow_duplicate=True),
        Output("last-mod-time-store", "data"),
        Output('node-info-display', 'children', allow_duplicate=True),
        Input("interval-component", "n_intervals"),
        State("last-mod-time-store", "data"),
        prevent_initial_call=True
    )
    def update_graph_from_file(n_intervals, last_mod_time):
        if not os.path.exists(default_file_path):
            if last_mod_time != -2:
                 print("Data file is missing. Returning placeholder.")
                 return        initial_elements,        initial_elements,
default_stylesheet  +  initial_placeholder_stylesheet,  -2,  "No  data  file
found."
            else:
                return   no_update,   no_update,   no_update,   no_update,
no_update
        try:
            current_mod_time = os.path.getmtime(default_file_path)
        except OSError:
            if last_mod_time != -2:
                print("Cannot  get  file  modification  time.  Returning
placeholder.")
                return         initial_elements,         initial_elements,
default_stylesheet + initial_placeholder_stylesheet, -2, "Error accessing
data file."
            else:
                return   no_update,   no_update,   no_update,   no_update,
no_update

        if current_mod_time != last_mod_time:
            print(f"Data file change detected (time: {current_mod_time}).
Updating graph...")
            data = get_dependency_data(default_file_path)
            reset_info_text  =  "Graph  updated.  Click  a  node  to  see  its
details."
            if data is None:
                 print("File is empty or contains invalid data. Returning
placeholder.")
                 if last_mod_time != -3:
                     return      initial_elements,      initial_elements,
default_stylesheet + initial_placeholder_stylesheet, -3, "Data file is empty
or invalid."
                 else:
```

```
                    return no_update, no_update, no_update, no_update,
no_update
            else:
                print("Data loaded successfully. Building graph
elements...")
                new_elements = build_elements(data)
                if not new_elements:
                    print("Failed to build graph elements from data.
Returning placeholder.")
                    if last_mod_time != -3:
                        return initial_elements, initial_elements,
default_stylesheet + initial_placeholder_stylesheet, -3, "Could not build
graph from data."
                    else:
                        return no_update, no_update, no_update, no_update,
no_update
                else:
                    print(f"Graph updated. Number of elements:
{len(new_elements)}")
                    return new_elements, new_elements, default_stylesheet,
current_mod_time, reset_info_text
        else:
            return no_update, no_update, no_update, no_update, no_update


    @callback(
        Output('cytoscape-tooltip', 'children'),
        Output('cytoscape-tooltip', 'style'),
        Input('cytoscape-deps', 'mouseoverNodeData')
    )
    def display_mouseover_node_data(data):
        default_tooltip_style = {
            'position': 'absolute', 'bottom': '15px', 'left': '15px',
            'backgroundColor': 'rgba(0, 0, 0, 0.7)', 'color': 'white',
            'padding': '5px 10px', 'borderRadius': '3px', 'fontSize':
'12px',
            'zIndex': 10, 'display': 'none'
        }
        if data:
            full_name = data.get('full_name', 'N/A')
            visible_tooltip_style = default_tooltip_style.copy()
            visible_tooltip_style['display'] = 'block'
            return full_name, visible_tooltip_style
        else:
            return '', default_tooltip_style

    # --- Виправлений колбек highlight_neighbors_on_tap ---
    @callback(
        Output('cytoscape-deps', 'stylesheet', allow_duplicate=True),
        Input('cytoscape-deps', 'tapNodeData'),
        State('current-elements-store', 'data'),
```

```
        prevent_initial_call=True
    )
    def highlight_neighbors_on_tap(tapped_node, elements):
        if not tapped_node:
            print("Tap background, resetting styles.")
            return default_stylesheet # Повертаємо оригінальний стиль

        node_id = tapped_node['id']
        print(f"Node tapped: {node_id}")

        # Починаємо з копії дефолтного стилю
        new_stylesheet = list(default_stylesheet)

        # 1. Додаємо правила для затемнення ВСІХ вузлів та ребер
        #      Важливо: ці правила будуть ПЕРЕКРИТІ нижче для вибраних
елементів
        new_stylesheet.append({'selector': 'node', 'style': {'opacity':
0.3}})
        new_stylesheet.append({'selector': 'edge', 'style': {'opacity':
0.15}})

        # 2. Знаходимо сусідів та ребра
        neighbor_ids = set()
        edge_ids = set()
        if elements: # Перевіряємо, чи є елементи
            for element in elements:
                data = element.get('data', {})
                source = data.get('source')
                target = data.get('target')
                edge_id = data.get('id')

                if source == node_id:
                    if target: neighbor_ids.add(target)
                    if edge_id: edge_ids.add(edge_id)
                elif target == node_id:
                    if source: neighbor_ids.add(source)
                    if edge_id: edge_ids.add(edge_id)

        # 3. Додаємо правила для ПІДСВІЧУВАННЯ (перекриття затемнення)
        #    Використовуємо СТИЛІ, а не класи!

        # Стиль для вибраного вузла
        new_stylesheet.append({
            'selector': f"node[id = '{node_id}']",
            'style': {
                'border-width': 3,
                'border-color': '#ffc107', # Жовтий/Золотий
                'opacity': 1 # Повна видимість
            }
        })
```

```
        # Стилі для сусідніх вузлів
        if neighbor_ids:
            neighbors_selector = ', '.join([f"node[id = '{nid}']" for nid
in neighbor_ids])
            new_stylesheet.append({
                'selector': neighbors_selector,
                'style': {
                    'border-width': 2,
                    'border-color': '#0dcaf0', # Бірюзовий
                    'opacity': 1 # Повна видимість
                }
            })


        print(f"Applying highlight styles by overriding opacity and
borders.")
        return new_stylesheet

    @callback(
        Output('node-info-display', 'children', allow_duplicate=True),
        Input('cytoscape-deps', 'tapNodeData'),
        prevent_initial_call=True
    )
    def display_tap_node_data(data):
        if data:
            node_id = data.get('id', 'N/A')
            full_name = data.get('full_name', node_id)
            info_content = [
                html.Strong("Selected Node:"),
                html.Br(),
                full_name
            ]
            print(f"Displaying info for: {node_id}")
            return info_content
        else:
            print("Clearing node info display.")
            return "Click a node to see its details."


    if __name__ == '__main__':
        app.run(debug=True, host='127.0.0.1', port=8050)
```

Лістинг Г.4 — Код класу DllAnalyzerService.cs

```
using System;
using System.Collections.Generic;
using System.Reflection;
using DependencyAnalyzer.Interfaces;
using DependencyAnalyzer.Models;
using Microsoft.Extensions.Logging;
```

```
using System.IO; // Для Path
namespace DependencyAnalyzer.Services
{
public class DllAnalyzerService : IDllAnalyzerService
{
private readonly ILogger<DllAnalyzerService> _logger;
public DllAnalyzerService(ILogger<DllAnalyzerService> logger)
{
_logger = logger;
}
// Оновлена реалізація
public DependencyNode AnalyzeAssembly(byte[] assemblyBytes, string
originalPath = null)
{
string assemblyNameLog = string.IsNullOrEmpty(originalPath) ? "from
bytes" : Path.GetFileName(originalPath);
try
{
_logger.LogInformation("Starting analysis for assembly
{AssemblyNameLog}", assemblyNameLog);
Assembly assembly = Assembly.Load(assemblyBytes);
_logger.LogInformation("Assembly loaded successfully:
{AssemblyFullName}", assembly.FullName);
DependencyNode root = new DependencyNode
{
Name = assembly.FullName,
Dependencies = GetReferencedAssemblies(assembly, new
HashSet<string>())
};
_logger.LogInformation("Analysis complete for assembly:
{AssemblyFullName}", assembly.FullName);
return root;
}
catch (BadImageFormatException ex)
{
_logger.LogError(ex, "Error during analysis of assembly
{AssemblyNameLog}: The file is not a valid .NET assembly.", assemblyNameLog);
throw new ArgumentException($"Файл '{assemblyNameLog}' не є валідною
.NET збіркою.", ex);
}
catch (Exception ex)
{
_logger.LogError(ex, "Error during analysis of assembly
{AssemblyNameLog}", assemblyNameLog);
throw; }}
private List<DependencyNode> GetReferencedAssemblies(Assembly
assembly, HashSet<string> visited){
var nodes = new List<DependencyNode>();
if (visited.Contains(assembly.FullName)){
_logger.LogDebug("Assembly already processed (cycle detected or
redundant): {AssemblyFullName}", assembly.FullName);
```

```
       return nodes; }
       visited.Add(assembly.FullName);
       foreach (AssemblyName refName in assembly.GetReferencedAssemblies()){
       try {
       _logger.LogDebug("Attempting    to    load    referenced    assembly:
{RefAssembly}", refName.FullName);
       Assembly refAssembly = Assembly.Load(refName);
       var node = new DependencyNode{
       Name = refAssembly.FullName,
       Dependencies     =     GetReferencedAssemblies(refAssembly,     new
HashSet<string>(visited))
       };
       nodes.Add(node);
       _logger.LogDebug("Loaded   and   processed   referenced   assembly:
{RefAssembly}", refAssembly.FullName);}
       catch (FileNotFoundException){
       _logger.LogWarning("Failed to load referenced assembly (not found):
{RefAssembly}", refName.FullName);
       nodes.Add(new DependencyNode { Name = $"{refName.FullName} (not found
/ failed to load)" });}
       catch (Exception ex){
       _logger.LogWarning(ex,
       "Failed to load referenced assembly (other error): {RefAssembly}",
       refName.FullName);
       nodes.Add(new  DependencyNode  {  Name  =  $"{refName.FullName}  (error
loading: {ex.GetType().Name})" });}}
       return nodes;
       }}}
```

Лістинг Г.5 — Код класу DependencyTreeBuilder.cs

```
using System.Reflection;
using DependencyAnalyzer.Interfaces;
using DependencyAnalyzer.Models;

namespace DependencyAnalyzer.Services{
public class DependencyTreeBuilder : IDependencyTreeBuilder
    {
        public     DependencyNode     BuildTree(Assembly     assembly,
HashSet<string> visitedAssemblies = null!)
        {
            if (visitedAssemblies == null)
                visitedAssemblies = [];

            var assemblyFullName = assembly.FullName;
            var node = new DependencyNode { Name = assemblyFullName!
};

            if (visitedAssemblies.Contains(assemblyFullName!))
                return node;
```

```
                visitedAssemblies.Add(assemblyFullName!);

                var                referencedAssemblies                =
assembly.GetReferencedAssemblies();
                foreach (var refAssemblyName in referencedAssemblies)
                {
                    try
                    {
                        var refAssembly = Assembly.Load(refAssemblyName);
                        var     childNode     =     BuildTree(refAssembly,
visitedAssemblies);
                        node.Dependencies.Add(childNode);
                    }
                    catch (Exception)
                    {
                        node.Dependencies.Add(new DependencyNode { Name =
refAssemblyName.FullName + " (не вдалося завантажити)" });
                    }
                }
                return node;
            }
        }
    }
```

Лістинг Г.6 — Клас налаштування проєкту Program.cs

```
    using DependencyAnalyzer.DllFactory.Interfaces;
    using DependencyAnalyzer.DllFactory.Services;
    using DependencyAnalyzer.Interfaces;
    using DependencyAnalyzer.Services;
    using System.Diagnostics;

    var builder = WebApplication.CreateBuilder(args);

    builder.Services.AddControllers();
    builder.Services.AddEndpointsApiExplorer();
    builder.Services.AddSwaggerGen();

    builder.Services.AddScoped<IReflectionService, ReflectionService>();
    builder.Services.AddScoped<IDependencyTreeBuilder,
DependencyTreeBuilder>();
    builder.Services.AddScoped<IAssemblyValidator, AssemblyValidator>();
    builder.Services.AddScoped<ITypeAnalyzerService,
TypeAnalyzerService>();
    builder.Services.AddScoped<IDllAnalyzerService, DllAnalyzerService>();
    builder.Services.AddScoped<IAssemblyAnalysisService,
AssemblyAnalysisService>();
    builder.Services.AddScoped<IDllFactory, DllFactory>();
```

```
builder.Services.AddHostedService<PythonServiceStarter>();

var app = builder.Build();

app.UseStaticFiles();
app.UseDefaultFiles();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json",
"DependencyAnalyzer API V1");
    });
}
app.Lifetime.ApplicationStarted.Register(() =>
{
    var url = "http://localhost:5297/index.html";
    try{
        Process.Start(new ProcessStartInfo{
            FileName = url,
            UseShellExecute = true
        });}
    catch (Exception){
        var                          logger                          =
app.Services.GetRequiredService<ILogger<Program>>();
}
});
app.Lifetime.ApplicationStopping.Register(() =>
{
    var   filePath   =   Path.Combine(Directory.GetCurrentDirectory(),
"dependency_data.json");
    if (File.Exists(filePath)){
        try{
            File.WriteAllText(filePath, string.Empty);}
        catch (Exception){
            var                        logger                          =
app.Services.GetRequiredService<ILogger<Program>>();
        }}});
app.UseRouting();
app.UseEndpoints(endpoints =>{
    _ = endpoints.MapControllers();
});
app.Run();
```