Міністерство освіти і науки України

Український державний університет науки і технологій

Факультет «Комп'ютерні технології і системи»

Кафедра «Комп'ютерні інформаційні технології»

Пояснювальна записка

до кваліфікаційної роботи бакалавра

на тему: «Проектування та розгортання маштабованоі системи для збору й аналізу телеметрії в Kubernetes за допомогою Elasticsearch і Grafana»

за освітньою програмою: «12 Інженерія програмного забезпечення»

зі спеціальності: «121 Інженерія програмного забезпечення»

Виконав: студент групи «ПЗ2016»

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

(підпис студента) (Ім’я ПРІЗВИЩЕ)

|  |  |  |  |
| --- | --- | --- | --- |
| Керівник: |  |  |  |

(підпис) (посада, Ім’я ПРІЗВИЩЕ)

|  |  |
| --- | --- |
| Засвідчую, що у цій роботі немає запозичень з праць інших авторів без відповідних посилань. | |
| Студент |  |

(підпис)

Дніпро – 2025 рік

Ministry of Education and Science of Ukraine

Ukrainian State University of Science and Technologies

Faculty «Computer technologies and systems»

Department «Computer information technology»

Explanatory Note

to Bachelor’s Thesis

on the topic: «Design and deployment of a scalable system for collecting and analyzing telemetry in Kubernetes using Elasticsearch and Grafana»

according to educational curriculum «12 Software engineering»

in the Speciality: «121 Software engineering»

|  |  |  |
| --- | --- | --- |
| Done by the student of the group PZ2016: |  |  |
| Scientific Supervisor: |  |  |
| Normative controller: |  |  |

Dnipro – 2025

Міністерство освіти і науки України

Український державний університет науки і технологій

Факультет: «Комп'ютерні технології і системи»

Кафедра: «Комп'ютерні інформаційні технології»

Рівень вищої освіти: бакалавр

Освітня програма: «12 Інженерія програмного забезпечення»

Спеціальність: «121 Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри КІТ

/Вадим ГОРЯЧКІН/

(підпис)

Дата

**ЗАВДАННЯ**

На кваліфікаційну роботу

бакалавра студенту

1. Тема роботи: «Проектування та розгортання маштабованоі системи для збору й аналізу телеметрії в Kubernetes за допомогою Elasticsearch і Grafana»

Керівник роботи: Шаповал Ірина Вікторівна, старший викладач

затверджені наказом № 328ст від 03.03.2025.

|  |  |  |
| --- | --- | --- |
| 2. Строк подання студентом роботи: | | 19.06.2025 |
| 3. Вихідні дані до роботи: |  |  |
| 4. Зміст пояснювальної записки (перелік питань, які потрібно опрацювати): | | |
| вступ, збір вимог до моніторингу, зовнішнє і внутрішнє | | |
| проектування, тестування та налагодження, висновки, література. | | |
| 5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень): презентація, відео-демонстрація роботи програми. | | |

**КАЛЕНДАРНИЙ ПЛАН**

|  |  |  |  |
| --- | --- | --- | --- |
| № з/п | Назва етапів кваліфікаційної роботи | Строк виконання етапів роботи | Примітка |
| 1 | Постановка задачі | 03.04.2025 – 05.04.2025 |  |
| 2 | Огляд літератури та аналіз аналогів | 06.04.2025 – 09.04.2025 |  |
| 3 | Розробка структур вхідних і вихідних даних | 10.04.2025 – 24.04.2025 |  |
| 4 | Визначення вимог до програми. Вибір та обґрунтування мови програмування | 25.04.2025 – 02.05.2025 |  |
| 5 | Узгодження та затвердження ТЗ | 03.05.2025 – 10.05.2025 | 30% |
| 6 | Розробка та програмування логіки програми | 11.05.2023 – 18.05.2023 |  |
| 7 | Розробка і реалізація графічного відоюраження метрик (Kibana, Grafana) | 19.05.2025 – 26.05.2025 | 60% |
| 8 | Відлагодження програми | 27.05.2025 – 02.06.2025 |  |
| 9 | Розробка, узгодження та затвердження програмної документації | 03.06.2025 – 18.06.2025 | 100% |
| 10 | Подання кваліфікаційної роботи до кафедри | 19.06.2025 – 26.06.2025 |  |
| 11 | Захист кваліфікаційної роботи на засіданні Екзаменаційної комісії | 27.06.2025 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Студент |  |  |  |

(підпис (Ім’я ПРІЗВИЩЕ)

|  |  |  |  |
| --- | --- | --- | --- |
| Керівник роботи |  |  |  |

**(підпис (Ім’я ПРІЗВИЩЕ)**

## РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи бакалавра 82 с., 10 рис., 9 табл., 12 джерел.

**Метою роботи** було проектування та розгортання масштабованої системи для збору й аналізу телеметрії в Kubernetes з використанням Elasticsearch і Grafana. Основними завданнями було створення надійної та гнучкої архітектури, що дозволяє ефективно збирати, зберігати та візуалізувати дані про роботу контейнеризованих додатків та інфраструктури, а також налаштування алертингу для проактивного виявлення проблем. За результатами роботи було створено інтегроване рішення, що відповідає всім вимогам сучасних підходів до моніторингу розподілених систем.

Робота складається з 7 розділів:

* вступ — у цьому розділі описується актуальність проблеми моніторингу контейнеризованих додатків, мета та завдання дослідження. Складається з 3 сторінок;
* збір вимог до програмного забезпечення — у цьому розділі аналізуються існуючі системи моніторингу, розглядаються теоретичні основи контейнеризації та оркестрації, принципи роботи Kubernetes, системи збору та аналізу логів і метрик. Складається з 20 сторінок;
* зовнішнє і внутрішнє проектування — у цьому розділі описується функціональне призначення системи, визначаються вхідні і вихідні дані, проводиться проектування архітектури системи моніторингу, тестового додатку, системи алертів та дашбордів для візуалізації даних. Складається з 23 сторінок;
* реалізація та тестування — цей розділ містить інформацію про розгортання Kubernetes кластера, налаштування компонентів системи моніторингу (ELK Stack, Prometheus, Grafana), розгортання тестового додатку та результати тестування системи. Складається з 18 сторінок;
* висновки — підсумовуються результати роботи та визначаються перспективи подальших досліджень. Складається з 2 сторінок;
* список літератури — включає бібліографічний список використаної літератури. Складається з 2 сторінок;
* додатки — містять конфігураційні файли та код, що використовувався для розгортання та налаштування системи, а також приклади дашбордів. Складається з 14 сторінок.

Кількість таблиць: 9 штук. Кількість рисунків: 10 штук.

Ключові слова: KUBERNETES, МОНІТОРИНГ, ELASTICSEARCH, GRAFANA, PROMETHEUS, КОНТЕЙНЕРИЗАЦІЯ, ОРКЕСТРАЦІЯ, ЛОГИ, МЕТРИКИ, ТЕЛЕМЕТРІЯ, МАСШТАБОВАНІСТЬ, АНАЛІЗ ДАНИХ, ВІЗУАЛІЗАЦІЯ, АЛЕРТИНГ.

ЗМІСТ

[ВСТУП 9](#_Toc138003767)

[1 ЗБІР ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 11](#_Toc138003768)

[1.1 Аналіз існуючих рішень для моніторингу](#_Toc138003769)

[1.1.1 Огляд платформи «DataDog»     12](#_Toc138003770)

[1.1.2 Огляд платформи «Dynatrace»     13](#_Toc138003771)

[1.1.3 Огляд стеку ELK](#_Toc138003772).

[1.1.4 Огляд Prometheus Stack     16](#_Toc138003773)

[1.2.1 Принципи роботи Kubernetes. 17](#_Toc138003774)

[Висновки до пункту 1 Error: Reference source not found](#_Toc138003775)

[2 ЗОВНІШНЄ І ВНУТРІШНЄ ПРОЕКТУВАННЯ 21](#_Toc138003776)

[2.1 Зовнішнє проектування 21](#_Toc138003777)

[2.1.1 Функціональне призначення 21](#_Toc138003778)

[2.1.2 Експлуатаційне призначення 21](#_Toc138003779)

[2.1.3 Функціональні вимоги 21](#_Toc138003780)

[2.1.4 Вхідні та вихідні дані 21](#_Toc138003781)

[2.1.4.1 Вхідні дані 21](#_Toc138003782)

[2.1.4.2 Вихідні дані 23](#_Toc138003783)

[2.1.5 Опис зовнішнього інформаційного середовища 24](#_Toc138003784)

[2.2 Внутрішнє проектування 26](#_Toc138003785)

[2.2.1 Аналіз зовнішніх специфікацій систем 26](#_Toc138003786)

[2.2.1.1 Моделювання словника системи 26](#_Toc138003787)

[3 ТЕСТУВАННЯ ТА НАЛАГОДЖЕННЯ 48](#_Toc138003788)

[Висновки до пункту 3 51](#_Toc138003789)

[ВИСНОВОК 52](#_Toc138003790)

[ЛІТЕРАТУРА 53](#_Toc138003791)

[ДОДАТКИ 54](#_Toc138003792)

[ДОДАТОК А – ТЕКСТ ПРОГРАМИ 56](#_Toc138003793)

ПЕРЕЛІК УМОВНИХ ПОЗНАК, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ (програмне забезпечення) – програмний продукт, який застосовується на комп’ютерних системах таких як персональні комп’ютери, ноутбуки, мобільні пристрої, планшети, телефони а також деколи і на різній побутовій техніці на кшталт пральних машин, мікрохвильовок, холодильників та інше.

Кастомізація – підлаштовування, налаштування чогось під себе. Те ж саме що й персоналізація.

Нота – ступінь або звук, що має характеристики: октава, знак альтерації, протяжність та інші.

Тональність – це висотне розташування ладу. Усього існує 30 тональностей.

Лад – це сукупність звуків, які на основі спорідненості між ними об’єднані в систему, що має тоніку.

Тоніка – базова нота тональності або ладу, тобто та нота, від якої йде побудова тональності чи ладу.

Півтон – одиниця виміру музичних інтервалів, є найменшою відстанню між звуками за висотою. Зазвичай, візуально на фортепіано, у ролі півтону виступає одна клавіша.

Тон – одиниця виміру музичних інтервалів, містить в собі два півтони. Зазвичай, візуально на фортепіано, у ролі тону виступає дві клавіші підряд.

Ступінь – це звук музичної системи (звукоряду). Усього основних ступенів сім – «до», «ре», «мі», «фа», «соль», «ля», «сі».

Знак альтерації – знак, що відображає підвищення або зниження ступеню на півтона або тон. Таких знаків декілька: дієз (♯), дубль-дієз, бемоль (♭), дубль-бемоль, а також бекар (♮) – знак відміни дій інших знаків.

Нотний стан – це система з п’яти паралельних горизонтальних ліній, на якій розміщують ноти.

Октава – інтервал від «до» до «сі» включаючи обидві ноти, відображає звукоряд певної висоти. Таких октав може бути на різних інструментах різну кількість, і усі октави мають власні назви та власну висоту звучання.

Мажор – музичний лад, аккорди якого будуються на великій терції. Зазвичай цей лад має позитивне, радісне звучання.

Мінор – музичний лад, аккорди якого будуються на малій терції. Зазвичай цей лад має похмуре, сумне звучання.

# ВСТУП

Музика, як і у минулому, сьогодні, так і надалі залишатиметься невід’ємною частиною нашого життя. Технології просунулася вперед і тим самим змінилася сама музика – з’явились безліч нових інструментів та різноманітних жанрів, покращилась якість звучання, спростився процес створення нових композицій, пісень, аранжувань. Але будь-який інструмент у руках людини без гарної практики та базових знань ­– просто не розкритий інструмент, на якому неможливо відтворити щось дійсно приголомшливе. У всіх сферах є умовна "база знань", де потрібно досягти максимального рівня навичок, щоб отримати фундамент, завдяки якому людина стане майстром справи. У музиці важлива кожна нота, зіграна на інструменті, адже кожна нота задає тон та настрій композиції. Але все ж таки, чому музика настільки важлива для нашого світу? А важлива вона тим, що завдяки їй часто люди піднімають собі настрій, задають атмосферу у своєму повсякденному житті, мають різні спогади, пов’язані з окремими піснями. В тяжкі для людини часи, музика надає натхнення рухатись вперед та направляє на різноманітні вчинки, надає мотивацію, підтримує дух людини аби не дати їй впасти духом та поринути у депресію. Словом, «Музика дозволяє рухатись вперед і бути закоханим у неї». А ще є такі люди, як меломани, що просто не уявляють своє життя без музики, адже меломани обожнюють абсолютно різні жанри.

Завдяки технологіям нашого часу можна знайти різноманітні програми, що допомагають музикантам у вивченні та виявленні особистих проблем у музиці. Такі програми допомагають музикантові у його становленні, як висококваліфікованого фахівця. Тому, аби допомогти усім музикантам, а саме: як початківцям, так і більш досвідченим, – було вирішено створити ПЗ «Elegia», яка сприятиме у відточенні музичних навичок та знань музичної теорії, методом використовування запропонованих інструментів ПЗ для досягнення почуття, що передає та сама фраза: «Музика допомагає рухатися вперед…».

Завданням цієї дипломної роботи є вдосконалення навичок знань музиканта. Робота передбачає за собою написану музичну програму, що має тренажерні тести та інструментарій для музикантів із простим інтерфейсом для легкого використання. Для користувача ПЗ, здебільшого, є два режими у кожному розділі – режим тренування та режим інструменту, завдяки яким він зможе відточити свої знання в музичній теорії на практиці. Показувати результат тестування користувачеві – важлива складова навчання. Окрім зміцнення знань, «Elegia» матиме ряд інструментів із роботою для піаніно. Користувачем може бути будь-яка зацікавлена у музиці ​​людина – як музикант-початківець, так і професійний музикант зі стажем.

# 1 ЗБІР ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

## 1.1 Перегляд аналогових конкурентів

Серед сучасних рішень для моніторингу контейнеризованих середовищ та інфраструктури, окрім Prometheus, варто виділити дві потужні комерційні платформи — DataDog та Dynatrace. DataDog представляє собою комплексну SaaS-платформу для моніторингу інфраструктури та додатків, що відрізняється швидким розгортанням завдяки готовим інтеграціям з понад 450 технологіями. Платформа використовує агентну модель збору даних, де легкий агент встановлюється на кожний сервер або контейнер і збирає метрики, логи та трейси, забезпечуючи візуальну кореляцію між ними. Особливу увагу варто звернути на розширені можливості DataDog для моніторингу Kubernetes, включаючи автоматичне виявлення подів, сервісів, відстеження життєвого циклу контейнерів та візуалізацію взаємозалежностей між компонентами системи. В свою чергу, Dynatrace пропонує унікальний підхід "один агент для всього" через технологію OneAgent, яка автоматично виявляє всі компоненти середовища та збирає необхідну телеметрію без ручного налаштування. Ключовою перевагою Dynatrace є запатентована система штучного інтелекту Davis, що аналізує мільярди залежностей для виявлення причинно-наслідкових зв'язків між подіями та автоматичного визначення кореневих причин проблем. Обидві платформи пропонують першокласну інтеграцію з Kubernetes, включаючи спеціалізовані оператори для розгортання агентів, глибокий моніторинг кластера та компонентів, а також потужні аналітичні інструменти. Проте, на відміну від відкритого Prometheus, ці рішення мають значні недоліки — високу вартість при масштабуванні, зберігання даних у хмарній інфраструктурі стороннього провайдера та закриту екосистему з обмеженими можливостями кастомізації.

## Огляд платформи DataDog

DataDog — це комплексна SaaS-платформа для моніторингу інфраструктури, додатків та цифрових користувацьких взаємодій. Заснована у 2010 році, сьогодні вона є одним із провідних рішень для спостереження (observability) та моніторингу в корпоративному сегменті.

### Архітектура та принципи роботи

DataDog використовує агентну модель збору даних. Легкий агент встановлюється на кожний сервер, контейнер або віртуальну машину. Він збирає метрики, логи та трейси, відправляючи їх у централізоване сховище DataDog для подальшої обробки та аналізу.

Основні компоненти архітектури DataDog:

1. **DataDog Agent** — легкий, відкритий програмний агент, написаний на Go, який збирає метрики, логи та трейси з хостів і сервісів.
2. **Інтеграції** — готові модулі для збору даних від популярних технологій та сервісів.
3. **Central DataDog Platform** — хмарна платформа для зберігання, обробки та візуалізації даних.
4. **API і SDK** — інструменти для інтеграції та кастомізації.

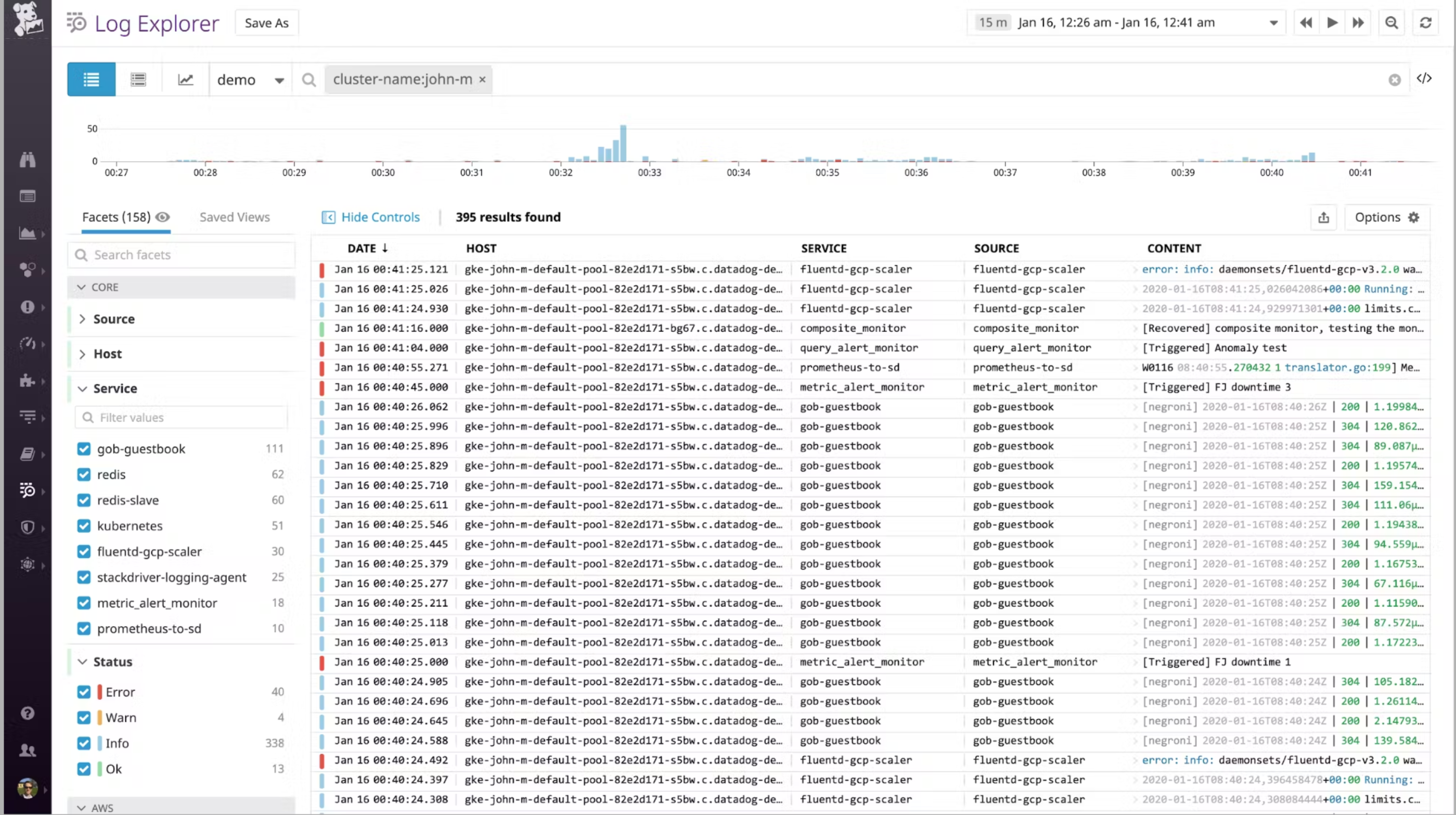


Рисунок 1.1 – Приклад дашборда DataDog

Переваги:

* **Швидкість впровадження**: готові інтеграції з понад 450 технологіями дозволяють швидко почати збір даних;
* **Уніфікований моніторинг**: єдина платформа для метрик, логів та трейсів;
* **Масштабованість**: здатність обробляти дані від десятків тисяч серверів;

Недоліки:

* **Висока вартість при масштабуванні**: ціна залежить від кількості хостів та функцій, що використовуються.
* **Зберігання даних у хмарі**: чутливі дані зберігаються на серверах DataDog, що може не відповідати вимогам безпеки деяких організацій.
* **Складність налаштування просунутих функцій**: деякі можливості вимагають глибокого розуміння платформи.

## 1.1.2 Огляд платформи Dynatrace

Dynatrace — це інтелектуальна платформа для моніторингу та обсерваційності, що використовує штучний інтелект для автоматичного виявлення та діагностики проблем у складних ІТ-середовищах. Заснована у 2005 році в Австрії, компанія Dynatrace зробила значний перехід від традиційного APM (Application Performance Management) до повноцінної платформи обсерваційності з власною AI-системою Davis. (сам приклад дашборду наведено на рис. 1.2).

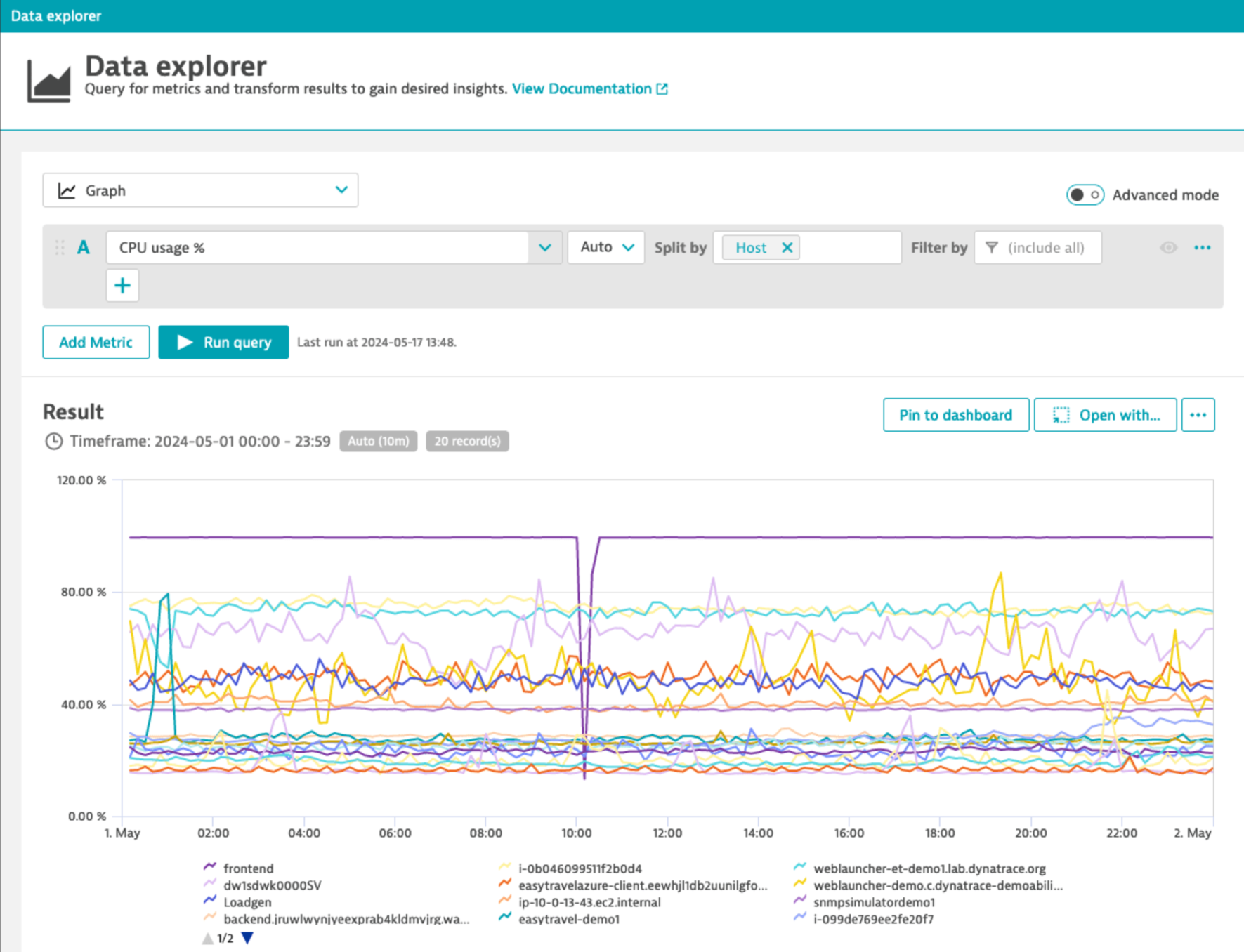


Рисунок 1.2 – Приклад дашборду «Dynatrace»

Переваги:

* **Автоматизація та AI** — повністю автоматичне виявлення та діагностика проблем без необхідності ручного налаштування;
* **Повна видимість** — єдина платформа для моніторингу від інфраструктури до користувацького досвіду;
* **Causation-Based AI** — визначення кореневих причин проблем, а не просто кореляцій між подіями;
* **Самоадаптація** — автоматична адаптація до змін в середовищі.

Недоліки:

* **Висока вартість ліцензії** — одне з найдорожчих рішень для моніторингу на ринку;
* **Складність впровадження** — потребує значного часу для повноцінного впровадження та налаштування;
* **Вимоги до ресурсів** — може потребувати значних ресурсів для роботи в великих середовищах;
* **Закрита екосистема** — менша гнучкість порівняно з відкритими рішеннями;
* **Складна крива навчання** — потребує часу для повного освоєння всіх можливостей;

## 1.1.3 Огляд стеку ELK.

ELK Stack представляє собою інтегроване рішення з відкритим вихідним кодом, спрямоване на комплексну роботу з логами: від їх збору та обробки до зберігання й візуалізації. Назва "ELK" походить від перших літер ключових компонентів: Elasticsearch, Logstash та Kibana. З розвиткомтехнології до стеку додалися легкі агенти Beats, що розширили його функціональність.

Рисунок 1.3 ілюструє архітектуру ELK Stack (Elasticsearch, Logstash, Kibana), доповнену компонентами Beats. Діаграма відображає повний потік даних у системі логування та моніторингу.

Зліва зображено сімейство Beats - легкі агенти для збору різних типів даних:

* Filebeat - для збору логів з файлів
* Winlogbeat - для збору логів подій Windows
* Metricbeat - для збору системних та прикладних метрик
* Packetbeat - для аналізу мережевого трафіку

Ці компоненти виконують функцію крайового збору даних (Edge Data Collection).

У центрі розташовано Logstash - систему обробки та трансформації даних, що представлена кількома робочими вузлами (Nodes) з увімкненими персистентними чергами (Persistent Queues Enabled). Logstash отримує дані від агентів Beats, обробляє їх та передає до Elasticsearch.

Справа розміщено два компоненти:

1. Elasticsearch - розподілене сховище для зберігання і пошуку даних, що складається з:
   * Master Nodes (3) - вузли управління кластером
   * Data Nodes - Hot (X) - вузли для зберігання активних даних
   * Data Nodes - Warm (X) - вузли для зберігання менш активних даних
2. Kibana - платформа візуалізації з кількома екземплярами (Instances (X))

Стрілки показують напрямок руху даних: від Beats до Logstash, і далі від Logstash до Elasticsearch, а Kibana підключається до Elasticsearch для відображення та аналізу зібраних даних.

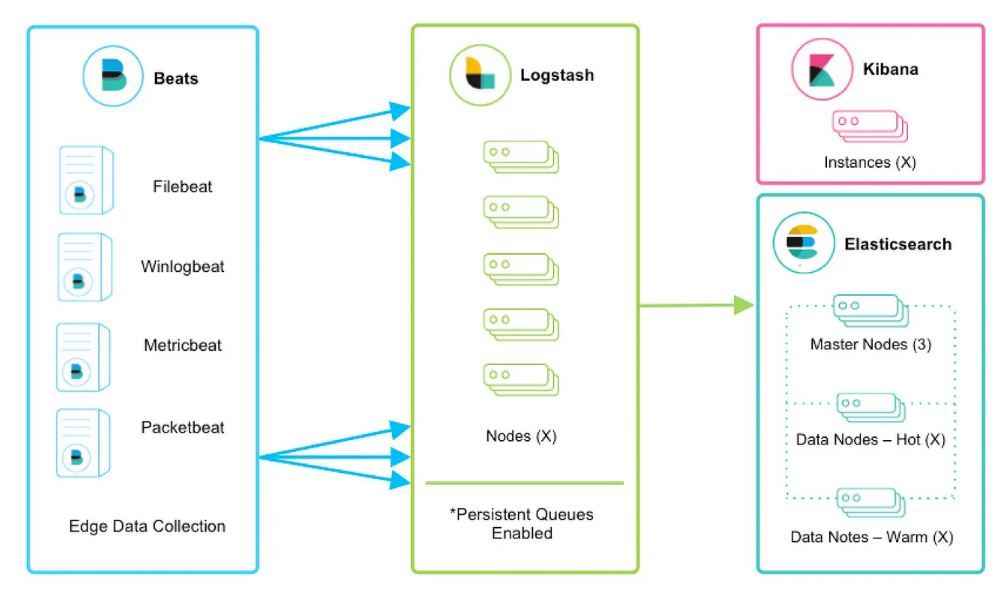


Рисунок 1.3 – ELK Stack

Переваги:

* **Прозорість реалізації**: можливість аудиту та розуміння внутрішньої роботи;
* **Спільнота розробників**: активна підтримка та розвиток компонентів;
* **Безкоштовне базове використання**: зниження витрат на впровадження;
* **Відсутність прив'язки до вендора**: незалежність від політики окремих компаній;
* **Можливість модифікації**: адаптація під специфічні потреби організації;

Недоліки:

* **Вимоги до ресурсів:** Одним з найсуттєвіших недоліків ELK Stack є високі вимоги до обчислювальних ресурсів;

### **Складність налаштування та підтримки:** ELK Stack вимагає значних зусиль для налаштування та підтримки;

### **Обмеження в роботі з метриками:** Незважаючи на можливість збору метрик через Metricbeat, ELK Stack має певні обмеження;

## 1.1.4 Огляд Prometheus Stack

Prometheus з Grafana формують потужний стек для моніторингу інфраструктури та додатків, який став стандартом де-факто для середовищ на базі Kubernetes. Цей стек пропонує спеціалізоване рішення для збору, зберігання, аналізу та візуалізації метрик.

## Архітектура та принципи роботи

Prometheus Stack побудований на кількох ключових компонентах:

* **Prometheus Server**: центральний компонент, що збирає та зберігає метрики як часові ряди
* **Exporters**: спеціалізовані агенти для надання метрик з різних систем
* **Alertmanager**: компонент для управління оповіщеннями
* **Pushgateway**: проміжний сервіс для короткоживучих завдань
* **Grafana**: платформа для візуалізації метрик у вигляді дашбордів

Архітектура стеку базується на моделі "pull", де Prometheus активно опитує (scrapes) цільові системи, а не чекає відправки даних, що забезпечує більшу надійність та контроль.

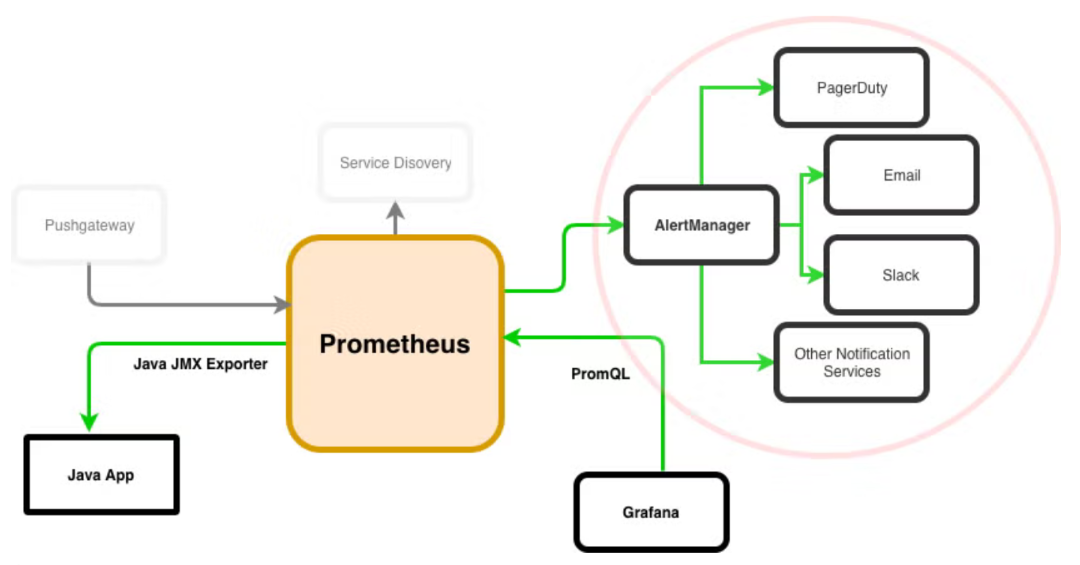


Рисунок 1.4 – Архітектура Prometheus

## 1.2.1 Принципи роботи Kubernetes.

Kubernetes (K8s) — це відкрита платформа для автоматизації розгортання, масштабування та управління контейнеризованими застосунками. Розроблена початково Google на основі внутрішньої системи Borg, Kubernetes стала стандартом де-факто для оркестрації контейнерів у промисловому масштабі.

#### Архітектура Kubernetes

Kubernetes має розподілену архітектуру типу "клієнт-сервер", що складається з наступних основних компонентів:

**Control Plane (Площина управління)**

Площина управління відповідає за глобальні рішення щодо кластера (наприклад, планування) та виявлення і реагування на події кластера. Компоненти площини управління включають:

1. **API Server (kube-apiserver)**: центральний компонент, що експонує API Kubernetes, обробляє запити та оновлює відповідні об'єкти в etcd. Всі взаємодії з кластером проходять через API Server.
2. **etcd**: розподілене сховище ключ-значення, яке зберігає всі дані кластера. Забезпечує надійне збереження конфігурації кластера та його стану.
3. **Scheduler (kube-scheduler)**: відповідає за відстеження новостворених подів, які не призначені до жодної ноди, та вибір ноди для їх запуску. Фактори для планування включають:
   * Вимоги до ресурсів
   * Обмеження апаратного/програмного забезпечення
   * Афініті та анти-афініті
   * Баланс навантаження між нодами
4. **Controller Manager (kube-controller-manager)**: запускає контролери, що обробляють рутинні завдання в кластері. Контролери включають:
   * Node Controller: моніторинг стану нод і реагування на їх відмови
   * Replication Controller: забезпечення правильної кількості подів
   * Endpoints Controller: зв'язування сервісів з подами
   * Service Account & Token Controllers: створення облікових записів і API токенів
5. **Cloud Controller Manager (cloud-controller-manager)**: інтегрує Kubernetes з API провайдерів хмарних послуг. Дозволяє основній кодовій базі Kubernetes залишатися незалежною від специфічних хмарних провайдерів.

**Worker Nodes (Робочі вузли)**

Worker ноди — це машини, які запускають контейнеризовані застосунки. Кожна нода містить:

1. **Kubelet**: агент, що запускається на кожній ноді. Забезпечує, що контейнери запускаються в поді. Kubelet:
   * Отримує специфікації подів від API сервера
   * Гарантує, що вказані контейнери працюють справно
   * Звітує про стан подів і ноди до API сервера
   * Запускає проби готовності та життєздатності
2. **Container Runtime**: програмне забезпечення, відповідальне за запуск контейнерів (containerd, CRI-O, Docker Engine та ін.)
3. **Kube-proxy**: мережевий проксі, що працює на кожній ноді. Підтримує мережеві правила та забезпечує зв'язок із подами. Реалізує частину концепції Kubernetes Service, керуючи перенаправленням з'єднань до відповідних подів.

#### Основні абстракції Kubernetes

Kubernetes оперує рядом абстракцій, що представляють стан системи:

**1. Pod (Под)**

Под — це найменша і найпростіша одиниця в Kubernetes, що представляє одиночний екземпляр застосунку. Под інкапсулює:

* Один або кілька контейнерів
* Спільний простір зберігання (томи)
* Специфічні налаштування мережі (IP-адреса, яку поділяють всі контейнери)
* Опції для запуску контейнерів

Поди, за своєю природою, є ефемерними (тимчасовими) і можуть бути замінені іншим подом при необхідності масштабування, оновлення або в разі збоїв.

**2. ReplicaSet**

ReplicaSet гарантує, що вказана кількість реплік подів запущена в будь-який момент часу. Забезпечує:

* Масштабованість застосунків
* Відмовостійкість через автоматичне відновлення подів
* Декларативне визначення бажаного стану

**3. Deployment**

Deployment — це вищий рівень абстракції над ReplicaSet, що надає декларативні оновлення для подів та ReplicaSets. Забезпечує:

* Стратегії оновлення (Rolling Update, Recreate)
* Відстеження історії розгортань
* Можливість відкату до попередніх версій
* Паузу та продовження розгортання

**4. Service**

Service — це абстракція, що визначає логічний набір подів і політику доступу до них. Сервіси дозволяють слабке зв'язування між залежними подами. Типи сервісів:

* ClusterIP: внутрішня IP-адреса, доступна лише всередині кластера
* NodePort: експонує сервіс на статичному порту кожної ноди
* LoadBalancer: використовує зовнішній балансувальник навантаження провайдера хмари
* ExternalName: повертає DNS CNAME запису

**5. Volume**

Volume вирішує дві проблеми: втрату даних, коли контейнер перезапускається, і спільне використання даних між контейнерами в поді. Типи томів:

* emptyDir: тимчасове порожнє сховище
* hostPath: монтування файлу або директорії з файлової системи ноди
* persistentVolume: постійне сховище, незалежне від життєвого циклу подів
* configMap/secret: монтування конфігурацій та секретів як файлів

**6. Namespace**

Namespace забезпечує віртуальне розділення кластера Kubernetes:

* Ізоляція ресурсів для різних команд, проектів або застосунків
* Керування доступом на рівні namespace
* Призначення квот ресурсів для окремих проектів

**7. StatefulSet**

StatefulSet керує розгортанням і масштабуванням набору подів з гарантіями щодо порядку розгортання та унікальності:

* Стабільні, унікальні імена мережі
* Стабільне, постійне сховище
* Упорядкований, плавний розгортання і масштабування
* Упорядковане оновлення та видалення

**8. DaemonSet**

DaemonSet забезпечує запуск копії поду на всіх (або деяких) нодах кластера:

* Ідеально для агентів моніторингу
* Збір логів на кожній ноді
* Агенти зберігання даних

**9. Job і CronJob**

Job створює один або кілька подів і гарантує, що вказана кількість успішно завершиться. CronJob створює Jobs за розкладом, подібно до crontab в Unix системах.

**10. ConfigMap і Secret**

ConfigMap зберігає конфігураційні дані у форматі ключ-значення для використання подами. Secret подібний до ConfigMap, але призначений для чутливих даних (паролі, токени, ключі).

# 2 ЗОВНІШНЄ І ВНУТРІШНЄ ПРОЕКТУВАННЯ

## 2.1 Зовнішнє проектування

## 2.1.1 Функціональне призначення

Функціональним призначенням програми є відточення навичків музикантів-початківців і любителів. Серед таких навичок: читання нот з нотного стану, тренування музичного слуху, навички зі знання тональностей.

## 2.1.2 Експлуатаційне призначення

Експлуатаційне призначення програми:

* роль музичного інструментарію;
* підвищення рівня базових знань у музиці;
* самоперевірка своїх знань;
* розвиток сприйняття нот на слух.

## 2.1.3 Функціональні вимоги

Дана музична програма має виконувати такі види вимог як:

* відтворювати тестові випробування з метою розвитку навичок для обраної користувачем категорії:
  + Note Identifier – ідентифікатор нот, навички читання з нотного стану;
  + Tonality Builder – побудова тональностей. Відточує навички зі знання різних тональностей, а також є інструментом з їх побудови;
* будувати візуальну схему із точок на віртуальному фортепіано у категорії Tonality Builder;
* кнопки режимів вибору налаштувань;
* відображати результати кожного циклу тестів;
* трекінг помилок кількості вірних відповідей та кількості тестів.

## 2.1.4 Вхідні та вихідні дані

## 2.1.4.1 Вхідні дані

Вхідними даними є:

1. isJustBeginned – змінна, що позначає чи виконується генерація тестових випробувань для користувача вперше у циклі. Дана змінна є типу bool та застосовується в обох функціоналах – Note Identifier та Tonality Builder;
2. note – змінна, що позначає ноту-відповідь від користувача під час режиму тренування у Note Identifier. Ця нота звіряється із нотою, згенерованою цим функціоналом. Дана змінна є типу перечислення Degree, що позначає основні ступені (C, D, E, F, G, A, B).
3. index – змінна, що використовується у методі визначення існування перечислення Degree за наданим індексом. Дана змінна є типу uint8\_t. Допустимі значення – від 0 до 255.
4. scaleRequestData – змінна, що застосовується у функціоналі Tonality Builder, та вміщує в собі загальні дані про тональність. Дана змінна є типу структури ScaleRequestData. Ця структура містить такі змінні: degree\_ типу Degree – ступінь ноти, що є тонікою тональності; alterationSign\_ типу AlterationSign – знак альтерації тоніки; key\_ типу Key – лад тональності (мажор чи мінор).
5. generatedScale – змінна, що позначає згенеровану тональність. Опираючись на цю змінну, також на інтерфейсі будується точкова схема на фортепіано для візуалізації тональності. Вона використовується під час генерації тональності для уникнення генерації одинакових тональностей. Дана змінна є типу структури Scale. Ця структура містить такі змінні: tonic\_ типу Note – тоніка тональності (зі ступенем degree\_ типу Degree, знаком альтерації alterationSign\_ типу AlterationSign, октавою octave\_ типу Octave); key\_ типу Key – лад тональності (мажор чи мінор); notes\_ типу NoteVec – вектор (динамічний масив) нот, що знаходяться в даній тональності; enharmonicScale\_ типу NoteVecOpt – вектор нот енгармонічної тональності, тобто такої, що по факту є такою самою тональністю і містить такі самі ноти, але просто називається по іншому. Opt позначає, що такої енгармонічної тональності може і не бути.
6. tonalityExists – змінна, що позначає існування згенерованої тональності. Вона допомагає уникнути генерації неіснуючих тональностей. Дана змінна є типу bool.
7. noteVec – змінна, що позначає вектор нот, за яким треба сформувати тональність. Використовується під час пошуку інформації про тональність за відомим переліком нот. Дана змінна є типу вектор нот Note.
8. volume – змінна, що відповідає за гучність нот, що програються через клас AudioEngine.

## 2.1.4.2 Вихідні дані

Вихідними даними є:

1. GenerateIdentifyNote – метод, що повертає згенеровану ноту для функціоналу Note Identify. Повертаєме значення є типу строка, що з боку інтерфейсу інтерпретується у вигляді розміщенного спрайту ноти на екрані.
2. IsCorrectAnswer – метод, що перевіряє надану користувачем відповідь на правильність. Цей метод використовується як у Note Identify, так і Tonality Builder функціоналі. Повертаєме значення є типу bool.
3. GetCorrectAnswers – метод, що повертає кількість вірних відповідей наданих користувачем. Цей метод використовується як у Note Identify, так і Tonality Builder функціоналі. Повертаєме значення є типу uint8\_t (від 0 до 255).
4. GetNotesGenerated – метод, що повертає кількість згенерованих нот у Note Identify функціоналі. Повертаєме значення є типу uint8\_t (від 0 до 255).
5. GenerateTonality – метод, що повертає згенеровану тональність для функціоналу TonalityBuilder. Повертаєме значення є типу Scale, який містить в собі згенеровану тональність та всі дані про неї.
6. BuildTonality – метод, що повертає побудовану тональність на основі даних, що надаються методу, про необхідну тональність. Повертаєме значення є типу Scale.
7. GetScalesGenerated – метод, що повертає кількість згенерованних тональностей у Tonality Builder функціоналі. Повертаєме значення є типу uint8\_t (від 0 до 255).
8. PlayNote – метод, що програє звук певної ноти на певній гучності. Повертаємого значення не має, а лише відтворює звук на персональному комп’ютері.
9. StartAndPlayQueue – метод, що програє звуки декількох нот у наданій черзі, послідовно відтворючи звуки на персональному комп’ютері. Повертаємого значення не має.

## 2.1.5 Опис зовнішнього інформаційного середовища

Програма, для повноцінного функціонування, потребує:

* Операційна система: Windows 7 або новіша;
* Наявність стандартних бібліотек Visual C++;
* Наявність динаміків для відтворення звуку нот;
* Наявність бібліотек QT 6.4.0 або новіше.

Нижче наведена діаграма прецедентів що зображує специфікацію функціональних вимог (рис. 2.1).

На ній можна побачити, що користувач має можливість використовувати функціонали нотного ідентифікатора (Note Identifier, тільки режим тренування), будувальника тональностей (Tonality Builder, обидва режими тренування та інструменту), транспонувальника (Transposer, обидва режими тренування та інструменту).

У кожному функціоналі користувач може програвати звуки нот під час роботи, але у нотному ідентифікаторі за замовчення ноти програються після генерації одразу. Цю функцію можна буде вимикати натисканням кнопки виключення звуку на інтерфейсі під час роботи.

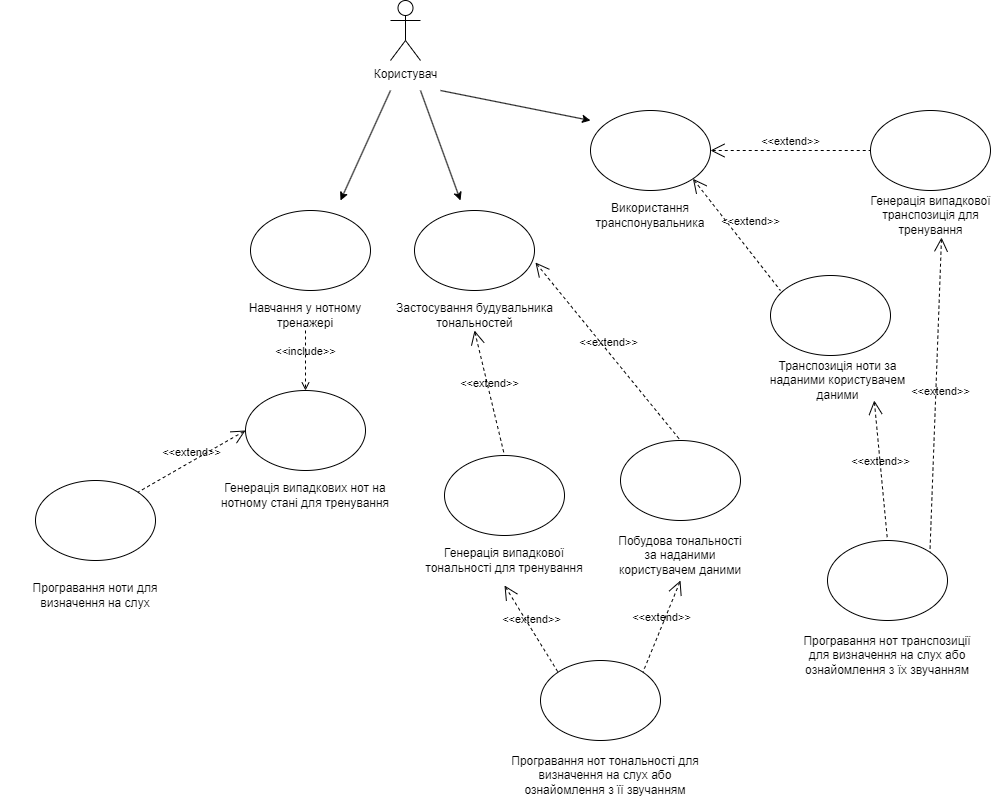


Рисунок 2.1 – Діаграма прецедентів

Транспонування у даній дипломній роботі розглядатися не буде, так як цю частину виконав студент Лисенко Максим (див. дипломну роботу студента Лисенка Максима).

## 2.2 Внутрішнє проектування

## 2.2.1 Аналіз зовнішніх специфікацій систем

## 2.2.1.1 Моделювання словника системи

Серед усіх сценаріїв взаємодї користувача можна виділити такі сутності як: Note Identifier (ідентифікатор нот), Tonality Builder (будувальник тональностей), Audio Engine (аудіо процессор), Note (сутність ноти), Scale (сутність тональності), Random (рандомізатор), Note Identifier Controller (допоміжна сутність зі сторони інтерфейсу для ідентифікатора нот), Tonality Builder Controller (допоміжна сутність зі сторони інтерфейсу для будувальника тональностей).

Опис обов’язків сутностей:

1. Note Identifier – реалізує функціонал системи вгадування нот, а саме логічну частину, обробку відповідей та генерації нот;
2. Tonality Builder – реалізує функціонал побудови тональностей, включаючи режим навчання та режим інструментарію. Виконує обробку відповідей користувача, обробку запитів на побудову тональності, генерацію тональностей для режиму навчання;
3. Audio Engine – відтворює звуки нот та нотних послідовностей (наприклад, ноти тональності), що дозволяє користувачу чути ноти під час генерації нот у Note Identifier, а також під час натискання на кнопки програвання у Tonality Builder;
4. Note – зберігає повну інформацію про ноту. Серед такої інформації:
   1. Ступінь (C, D, E, F, G, A, B);
   2. Знак альтерації (Дієз. Бемоль, бекар, дубль-дієз, дубль-бемоль або немає);
   3. Октава (суб-контроктава, контроктава, велика октава, мала октава, перша октава, друга октава, третя октава, четверта октава, п’ята октава, шоста октава);
5. Scale – зберігає повну інформацію про тональність. Серед такої інформації:
   1. Тоніка (ступінь, що відіграє роль першої ступені у тональності, від якої будується тональність);
   2. Знак альтерації тоніки;
   3. Лад;
   4. Перелік нот тональності;
   5. Перелік нот енгармонічної тональності – різною за назвою, але однаковою по факту звучання;
6. Random – керує рандомізацією чисельних значень, тобто генерацією випадкових чисел. Ці випадкові числа використовуються під час генерації нот та тональностей у режимі навчання.
7. Note Identifier Controller – оперує логікою Note Identifier зі сторони інтерфейсної частини користувача, обробляючи відповіді від Note Identifier і формуючи їх на екрані.
8. Tonality Builder Controller – оперує логікою Tonality Builder зі сторони інтерфейсної частини користувача, обробляючи відповіді від Tonality Builder і формуючи їх на екрані.

Детальні відповідні атрибути та операції, що стосуються кожного класу та є необхідними для виконання обов’язків наведено ниже, у табл. 2.1.

Таблиця 2.1 – Сутності, атрибути та методи

|  |  |  |
| --- | --- | --- |
| Сутність | Атрибути | Методи |
| 1 | 2 | 3 |
| Note Identifier | correctAnswers\_ – лічильник вірних відповідей;  generatedNote\_ – поточна згенерована випадкова нота;  notesGenerated\_ – лічильник згенерованих нот. | GenerateIdentifyNote – генерує випадкову ноту для подальшої ідентифікації користувачем під час випробування;  IsCorrectAnswer – перевіряє відповідь користувача на правильність;  GetCorrectAnswers – повертає кількість вірних відповідей, даних користувачем;  GetNotesGenerated – повертає загальну кількість генерацій нот; |

Продовження таблиці 2.1

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
|  |  | NullifyData – аннулює кількість генерацій та кількість правильних відповідей перед початком кожного нового випробування;  IsDegreeEnumIndex – визначає, чи існує за наданим індексом нотна ступінь. |
| Tonality Builder  (у коді ScaleBuilder) | correctAnswers\_ – лічильник вірних відповідей;  generatedScale\_ – поточна згенерована тональність для режиму тренування;  scalesGenerated\_ – лічильник згенерованих тональностей. | GenerateTonality – генерує випадкову тональність для режиму тренування;  BuildTonality – утворює тональність, опираючись на вхідні дані по необхідній тональності;  IsCorrectAnswer – перевіряє надану відповідь користувачем на правильність;  GetCorrectAnswers – повертає кількість вірних відповідей;  GetScalesGenerated – повертає кількість згенерованих тональностей; |

Продовження таблиці 2.1

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
|  |  | GetScaleRequestData – повертає опис тональності за наданим переліком нот;  GetScale – повертає тональність опираючись на вхідні дані, а саме опис тональності (тоніка, знак альтерації, лад);  GetEnharmonicScale – повертає енгармонічну тональність, а саме перелік нот цієї тональності, пов’язаної вхідними даними – описом тональності, за яким шукається енгармонічна тональність.  IsSuccessfulGeneration – перевіряє генерацію на успішність, звіряючи існування згенерованої тональності із записаними у базі функціоналу, тим самим запобігаючи помилкам генерації;  NullifyData – аннулює кількість генерацій та кількість вірних відповідей |

Продовження таблиці 2.1

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
|  |  | перед початком кожного нового випробування;  ValidateScaleRequestData – перевіряє запит що містить опис тональності, на коректність. Тобто, виконує валідацію. |
| Audio Engine | mediaPlayer\_ – клас, що відповідає за програвання аудіофайлу;  audioOutput\_ – клас, що відповідає за вивід звуку під час програвання аудіофайлу;  notesQueue\_ – утримує чергу з нот, звук яких потрібно програти послідовно;  lastVolumeLevel\_ – зберігає останній рівень гучності, який застосовувася останній раз (від 0.0 до 1.0);  audioDuration\_ – відповідає за тривалість програвання аудіофайлу (у секундах). | AudioEngine – конструктор, ініціалізує необхідні атрибути для роботи початковими значеннями;  PlayNote – програє задану ноту на заданій гучності.  StartAndPlayQueue – програє задану послідовну чергу з нот на заданій гучності;  StopAudio – зупиняє програвання будь-якого аудіо у програмі;  GetFilePathForNote – повертає шлях до аудіо файлу, що містить звук заданої ноти;  PlayAudioFile – програє аудіо файл за заданим шляхом на заданій гучності;  PlayNextAudio – програє |

Продовження таблиці 2.1

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
|  |  | наступний аудіо файл із черги. Виконує роль тріггера. |
| Note | degree\_ – ступінь ноти;  alterationSign\_ – знак альтерації;  octave\_ – октава. | GetNextDegree() – повертає наступну ступінь за висотою після поточної ступені даної ноти;  GetPreviousDegree() – повертає попередню ступінь за висотою перед поточною ступінню даної ноти;  GetNextOctave() – повертає наступну октаву за висотою після поточної октави даної ноти;  GetPreviousOctave() – повертає попередню октаву за висотою перед поточною октавою даної ноти;  IsSignGreater() – порівнює чи є перший вхідний знак альтерації більшим за другий вхідний знак альтерації;  IsSignLesser() – порівнює чи є перший вхідний знак альтерації більшим за |

Продовження таблиці 2.1

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
|  |  | другий вхідний знак альтерації. |
| Scale | tonic\_ – позначає ноту-тоніку тональності;  key\_ – лад (мінор чи мажор);  notes\_ – перелік нот в даній тональності;  enharmonicScale\_ – перелік нот енгармонічної тональності (якщо є). |  |
| Random |  | RandomizeInt32 – виконує генерацію випадкового цілого числа від першої вхідної межі до другої. |
| Note Identifier Controller | uiPtr\_ – слугує як смарт-вказівник на головне вікно програми;  audioEnginePtr\_ – слугує як смарт-вказівник на аудіо процессор Audio Engine;  isEndless\_ – позначає, чи користувач обрав нескінченний режим тренування; | NotesIdentifierController – конструктор, слугує ініціалізатором контроллера функціоналу;  BeginTheFlow – починає процес тренажеру ідентифікатору нот;  ProceedAnswer – оброблює відповідь користувача;  IsSoundOn – перевіряє, чи присутній звук у функціоналі та повертає стан; |

Продовження таблиці 2.1

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
|  | isSoundOn\_ – позначає, чи користувач увімкнув звук у даному функціоналі;  maxAmount\_ – зберігає значення кількості генерацій, обраної користувачем. | SetSoundState – змінює стан роботи звуку у функціоналі;  SetNotePosOnSheet – розміщує згенеровану ноту на нотному стані на інтерфейсі користувача;  SelectedIndexToNoteAmount – конвертує індекс комбо-боксу для вибору максимальної кількості нот. |
| Tonality Builder Controller | uiPtr\_ – слугує як смарт вказівник на головне вікно програми;  audioEnginePtr\_ – слугує як смарт-вказівник на аудіо процессор Audio Engine;  currentTonality\_ – поточна згенерована тональність.  isEndless\_ – позначає чи користувач обрав нескінченний режим тренування;  isToolMode\_ – позначає, чи зараз активований режим інструменту; | TonalityBuilderController – конструктор, слугує ініціалізатором контроллера функціоналу;  BeginLearnFlow – ініціалізує процес тренування у функціоналі;  BeginToolFlow – ініціалізує процес інструментарію у функціоналі;  ControlBuildNextButtonState – активує-деактивує кнопку побудови або кнопку продовження тренування;  IsToolMode – визначає, чи зараз активовано режим інструментарію та повертає |

Продовження таблиці 2.1

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
|  | maxAmount\_ – зберігає значення кількості генерацій, обраної користувачем; | відповідне значення;  ProceedAnswer – оброблює відповідь користувача у режимі тренування;  BuildTonality – будує тональність за запитом користувача у режимі тренування;  PlayBuiltTonality – програє побудовану тональність, програючи звуки нот, що входять у неї;  ControlPlayButtonState – активує-деактивує кнопку програвання звуків нот тональності;  SetTonalityDotsOnKeys – розміщує точки на віртуальному фортепіано, візуально зображуючи тональність для користувача;  HideDotsOnKeys – приховує  побудовані точки від користувача;  HideShowLearningModeElements – ховає або показує |

Закінчення таблиці 2.1

|  |  |  |
| --- | --- | --- |
|  |  | інтерфейсні елементи навчання (наприклад лічильники вірних відповідей)  SelectedIndexToTonalityAmount – конвертує індекс комбо-боксу кількості тональностей для визначення у режимі тренування;  ValidateTonality – корректизує тональність, а саме регуляє перехід на іншу октаву нот, що виходять за неї, і таким чином розміщує їх на віртуальному фортепіано коректно. |

#### 2.2.1.2 Моделювання розподілу обов’язків у системі

Нижче наведено перелік різних множин класів, що співпрацюють одне з одним задля виконання певних процесів:

* Note Identifier Controller, Note Identifier – передача згенерованої ноти на інтерфейс, прийняття та обробка відповідей від користувача, робота функціоналу ідентифікатора нот;
* Note Identifier Controller, Audio Engine – програвання звуку згенерованих нот;
* Note Identifier, Note – формування ноти;
* Note Identifier, Random – рандомізація ноти для генерації;
* Tonality Builder, Audio Engine – програвання звуку нот побудованих та згенерованих тональностей;
* Tonality Builder, Scale – формування тональності;
* Tonality Builder, Random – рандомізація тональності для генерації;
* Audio Engine, Note – програвання звуку конкретної ноти;
* Scale, Note – утворення послідовності нот у тональності;
* Tonality Builder Controller, Tonality Builder – передача згенерованої або побудованої тональності на інтерфейс, прийняття та обробка відповідей чи запитів побудови тональності від користувача, робота функціоналу будувальника тональностей;
* Tonality Builder Controller, Audio Engine – програвання звуку нот тональностей;

Примітивні типи будуть визначені під час побудови діаграми класів. Сам результат моделювання зв’язків різного виду наведено нижче у табл. 2.

Таблиця 2.2 – Моделювання залежностей

|  |  |  |
| --- | --- | --- |
| Клас, котрий зв’язується | Клас із котрим зв’язується | Тип зв’язку |
| 1 | 2 | 3 |
| Note Identifier Controller | Note Identifier | Реалізація |
| Audio Engine | Асоціація |
| Note Identifier | Note | Композиція |
| Random | Асоціація |
| Tonality Builder | Scale | Композиція |
| Random | Асоціація |
| Audio Engine | Note | Асоціація |
| Scale | Note | Композиція |
| Tonality Builder Controller | Tonality Builder | Реалізація |
| Audio Engine | Асоціація |
| Note | – | – |

Закінчення таблиці 2.2

|  |  |  |
| --- | --- | --- |
| Random | – | – |

#### 2.2.1.3 Визначення призначень об’єктів за допомогою CRC карток

Призначення об’єктів перелічено із застосуванням CRC (Class-Responsibility-Collaboration) карток нижче у таблицях 2.3 – 2.10. Завдяки таким карткам можна одразу побачити які класи походять від яких, які класи мають певних нащадків, а також

Таблиця 2.3 – CRC-картка для класу «Note Identifier Controller»

|  |  |
| --- | --- |
| Базовий клас | Похідні класи (нащадки) |
| відсутній | відсутні |
| Обов’язки | Зв’язки |
| Опрацьовує запити користувача під час роботи з Note Identifier, отримує результати логічних операцій від Note Identifier класу та інтерпретує їх на графічному інтерфейсі. | Note Identifier, Audio Engine |

Таблиця 2.4 – CRC-картка для класу «Note Identifier»

|  |  |
| --- | --- |
| Базовий клас | Похідні класи (нащадки) |
| відсутній | відсутні |
| Обов’язки | Зв’язки |
| Реалізує логічні операції щодо генерації нот, перевірку відповідей, що приходять від Note Identifier Controller. | Note, Random |

Таблиця 2.5 – CRC-картка для класу «Tonality Builder»

|  |  |
| --- | --- |
| Базовий клас | Похідні класи (нащадки) |
| відсутній | відсутні |

Закінчення таблиці 2.5

|  |  |
| --- | --- |
| Обов’язки | Зв’язки |
| Реалізує логічні операції щодо побудови тональностей, генерації тональностей, перевірку відповідей, що приходять від Tonality Builder Controller. | Scale, Random |

Таблиця 2.6 – CRC-картка для класу «Audio Engine»

|  |  |
| --- | --- |
| Базовий клас | Похідні класи (нащадки) |
| відсутній | відсутні |
| Обов’язки | Зв’язки |
| Відтворює звуки нот та послідовностей нот по черзі. | Note |

Таблиця 2.7 – CRC-картка для класу «Scale»

|  |  |
| --- | --- |
| Базовий клас | Похідні класи (нащадки) |
| відсутній | відсутні |
| Обов’язки | Зв’язки |
| Структуризація тональності як сутності | Note |

Таблиця 2.8 – CRC-картка для класу «Tonality Builder Controller»

|  |  |
| --- | --- |
| Базовий клас | Похідні класи (нащадки) |
| відсутній | відсутні |
| Обов’язки | Зв’язки |
| Опрацьовує запити користувача під час роботи з Tonality Builder, отримує результати логічних операцій від Tonality Builder класу та інтерпретує їх на графічному інтерфейсі, контролює режими | Tonality Builder, Audio Engine |

Закінчення таблиці 2.8

|  |  |
| --- | --- |
| Обов’язки | Зв’язки |
| тренування та інструментарію. |  |

Таблиця 2.9 – CRC-картка для класу «Note»

|  |  |
| --- | --- |
| Базовий клас | Похідні класи (нащадки) |
| відсутній | відсутні |
| Обов’язки | Зв’язки |
| Структуризація ноти як сутності | відсутні |

Таблиця 2.10 – CRC-картка для класу «Random»

|  |  |
| --- | --- |
| Базовий клас | Похідні класи (нащадки) |
| відсутній | відсутні |
| Обов’язки | Зв’язки |
| Рандомізація чисельних значень, що використовуються під час генерації у інших функціоналах програми | відсутні |

#### 2.2.1.4 Побудова об’єктної моделі (діаграми класів)

Як відомо, зазвичай об’єктна модель будується та демонструється через діаграму класів. Така діаграма класів позначає різноманітні об’єктні взаємозв’язки. Окрім цього, на діаграмі класів можна розгледіти як саме було виконано структуризацію класів. Серед цього можна відмітити види доступу до класових атрибутів та методів (приватний, публічний, або захищений), які позначаються відмітками +, - та #. Такі види доступу досягаються на практиці приписуванням у коді специфікаторів public, private, protected.

Важливо відмітити, що на діаграмі класів зазвичай вказуються різноманітні зв’язки між класами. Таких зв’язків на мові UML існує декілька, а саме серед них виділяють: композицію, агрегацію, асоціацію, залежність, реалізацію, спаадкування.

Узагальнити моделювання допоможе побудована діаграма класів (див рис.2.2.).

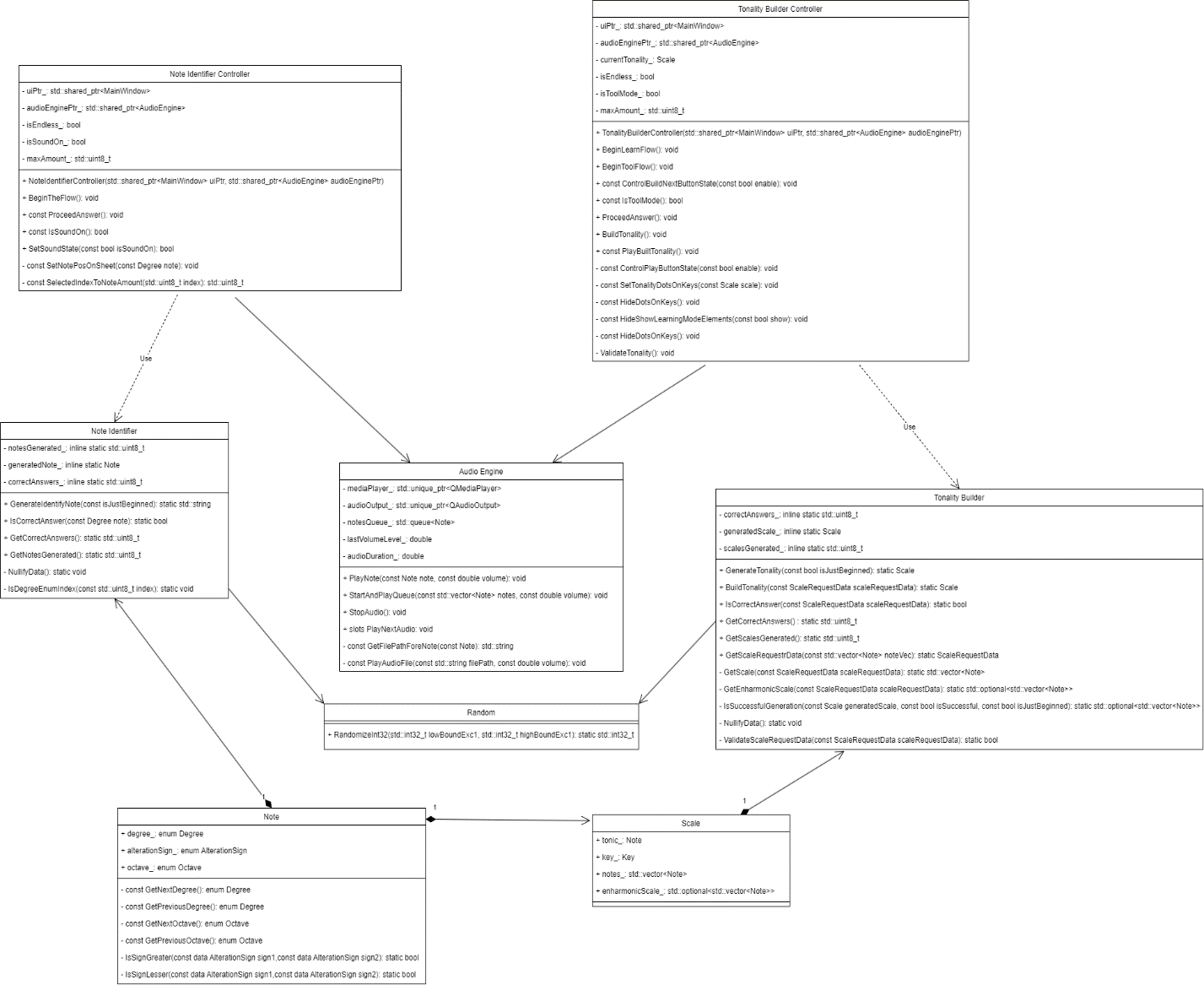


Рисунок 2.2 – Діаграма класів

Класи пов’язані асоціацією: Note Identifier Controller та Audio Engine, Note Identifier та Random, Audio Engine та Note, Tonality Builder та Random, Tonality Builder Controller та Audio Engine.

Класи пов’язані композицією: Note Identifier та Note, Tonality Builder та Scale, Scale та Note.

Класи пов’язані реалізацією: Note Identifier Controller та Note Identifier, Tonality Builder Controller та Tonality Builder.

#### 2.2.2 Проектування інтерфейсу користувача

#### 2.2.2.1 Створення ескізів форм

Спираючись на те, що проектування інтерфейсу користувача та створення ескізів форм було зроблено студентом Максимом Лисенко у цій роботі, їх можна побачити у його роботі у пункті «2.2.2.1. Створення ескізів форм».

#### 2.2.3 Проектування динаміки системи

У ролі об’єкту виступає користувач програмного забезпечення, який ініціює роботу класів через графічний інтерфейс. Самі звернення до інтерфейсу пропускаються, так як ця частина відноситься до роботи Лисенка Максима (див. дипломну роботу Лисенка Максима, пункт «2.2.3. Проектування динаміки системи). Розглядаються запити безпосередньо від контроллерів, через які здійснюється взаємодія між інтерфейсом там основною логічною складовою програми.

Опираючись на попередньо створену діаграму прецедентів [пункт 2.1.5], було розроблено дві діаграми послідовності для двох варіантів використання:

* Генерація випадкових нот на нотному стані для тренування (див. рис. 2.3)
* Генерація випадкової тональності для тренування (див. рис. 2.4)

Ці два прецеденти було обрано в зв’язку із тим, що найбільше логічної нагрузки на систему покладено саме у ціх прецедентах. «Побудова тональності за наданими користувачем даними» не була розглянута детально, так як практично виконує ті самі завдання, що й для генерації, але лише будує тональність за попередньо наданими даними.

На діаграмах не було ураховано сутності Scale та Note з тої причини, що вони виконують роль носїв інформації. Лише сутність Note має невелику кількість методів, що використовуються під час роботи застосунка, але ці методи виконують роль імітації операторів інкрементування, декрементування, порівняння «більше» та порівняння «менше». Сама сутність Scale повністю вміщує лише інформацію про тональність, не маючи жодних методів що могли б виконувати інкрементування або декрементування, операції порівняння «більше» та «менше», як це є у сутності Note. Окрім того присутній перелік ступеней, знаків альтерації та октав. Усі ці перелічення прив’язані до певних класів, тому вони також не розглядаються.

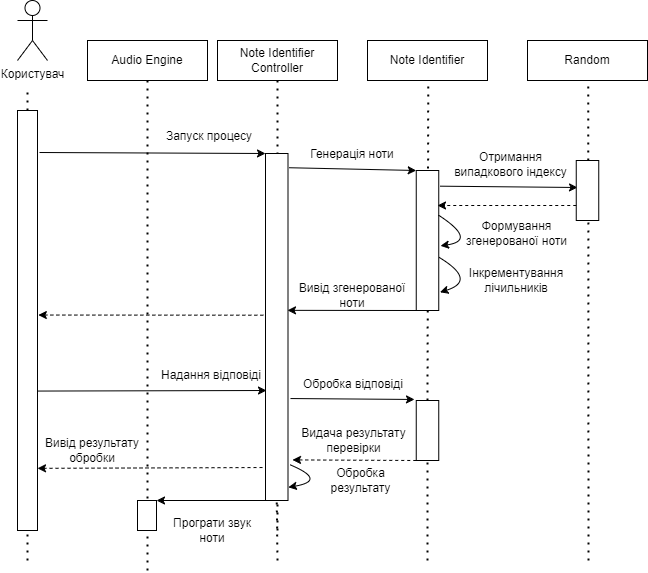


Рисунок 2.3 – Діаграма послідовності варіанту використання «Генерація випадкових нот на нотному стані для тренування»

Сценарій з рис. 2.3 пояснює наступні дії:

* Користувач посилає запит на запуск процесу тренажеру ідентифікатора нот;
* Запит оброблюється контролером та посилається до основного логічного модуля ідентифікатору нот;
* Логічний модуль звертається до рандомізатора з метою згенерувати індекс ноти, що буде згенеровано для користувача;
* За отриманим індексом формується нота, інкрементуються лічильники щодо згенерованої кількості нот;
* Контролер отримує відповідь від логічного модулю, формує її на екрані та передає на форму;
* Паралельно із цим програється звук ноти для користувача.

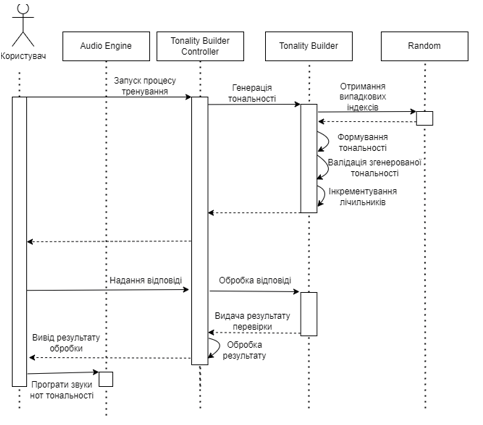


Рисунок 2.4 – Діаграма послідовності варіанту використання «Генерація випадкових тональностей для тренування»

Сценарій з рис. 2.4 пояснює наступні дії:

* Користувач посилає запит на запуск процесу тренажеру побудови тональності;
* Запит оброблюється контролером та посилається до основного логічного модуля побудови тональності;
* Логічний модуль звертається до рандомізатора з метою згенерувати індекси тональності (ступінь, знак альтерації, лад), за якими буде згенеровано саму тональність для користувача;
* За отриманими індексами формується тональність, яка надалі буде валідуватися перед видачею як відповідь на запит. Інкрементуються лічильники щодо згенерованої кількості тональностей;
* Контролер отримує відповідь від логічного модулю, формує її на екрані та передає на форму;
* За необхідності, користувач може програти звуки нот тональності.

#### 2.2.4 Вибір мови програмування та засобів спільної розробки

Для розробки продукту було вирішено використати мову програмування С++ тому, що дана мова дає можливість оперативно працювати із пам’яттю та надає гнучкість у розробці. Важливою складовою даної мови також є можливість програмувати за принципами ООП. Як відомо, ці принципи складають інкапсуляція, поліморфізм та наслідування, що С++ також має. Саме завдяки цім складовим розробка програми виявилась найбільш комфортною та ефективною.

Окрім самої мови програмування продукту, також застосовано автоматизовану систему збірки CMake. CMake слугує гнучкою системою збірки, завдяки якій можна розподіляти запускаємі частини програми на таргети – тобто об’єкти, які працюють по різному. Це є неймовірно корисним, як, наприклад, для багатосторонніх додатків на кштал серверів. У випадку з даним програмним продуктом, тут застосовується два таргети – таргет «програма» та таргет «юніт тести». У першому запускається сама програма, у другому – юніт тести до неї. Але на практиці запуск усієї програми відбувається з інтерфейсної частини фреймворку QT, що під’єднаний до логічної частини додатку.

Проект використовує додаткові бібліотеки, що встановлює пакетний менеджер Conan. Завдяки цьому пакетному менеджеру можна вільно встановити необхідні для проекту бібліотеки та підключати їх як модулі до необхідних частин коду. Conan завантажує специфіковані у файлі conanfile.txt бібліотеки та під’єднує їх до репозиторію.

Деякі компоненти було встановлено завдяки використанню MSYS2 – емулятора терміналу Unix-подібних систем, що працює на Windows. Він дозволяє оперувати у системі як наче це робиться у Linux, що дозволяє ставити різні бібліотеки, які зазвичай вільно постачаються для Linux. Серед таких слід відмітити Python бібліотеки, що необхідні для роботи пакетного менеджеру Conan.

Необхідно відмітити, що з боку інтерфейсної частини тут застосовано фреймворк QT. Цей фреймворк було обрано з тої причини, що він чудово співпрацює із мовою С++, так як безпосередньо написаний на ній. Надважливо також сказати, що цей фреймворк забезпечує роботу на різних платформах, таких як Windows, Linux та iOS. Це робить розробку продуктів на цьому фреймворці гнучкою, адже забезпечує кросплатформенність додатку. Фреймворк дозволяє працювати із мультимедіа, надаючи власні бібліотеки для цього, серед яких було застосовано QMultimedia.

Ключовим для проекту елементом є використання системи контролю версій Git. Дана система дозволяє зберігати виконану роботу у гілках – свого роду контрольних точках, на яких можна було продовжити роботу у інший час, якщо з’явилась більш термінова задача, або передати роботу іншому співучаснику на доопрацювання, або навіть проводити спільну роботу на одній і тій самій гілці, майже забезпечуючи спільну роботу у реальному часі. Це дозволяє не перекидувати кожного разу одне одному одні і ті самі файли під час виконання роботи над кодом засобами електронної пошти або іншими засобами, а також дозволяє постійно мати актуальний стан репозиторію, шляхом додавання власної роботи до спільного репозиторію та контролю над прогрессом.

Проектний контроль та керування подальшою співпрацею по проекту відбувалась із використанням Jira та хмарного репозиторію BitBucket. Це було зроблено задля забезпечення проекту підтримки актуальності різних проблемних моментів, багів, та подальших розробок різних функціоналів. Таким чином, у Jira – програмному забезпеченні відстеження задач, було створено безліч «тікетів», тобто задач, на які ми орієнтувалися під час спільної роботи над проектом. Завдяки цьому, ми мали змогу завжди бачити наші ключові задачі, мати актуальну необхідну інформацію по тікетам, а також відстежувати ті тікети, які є завершеними, та ті, які ще необхідно зробити. Jira також дозволяє розподіляти тікети не тільки між співучасниками проекту, а ще й відокремлювати тікети по спрінтам – короктим проміжкам часу , за які необхідно зробити певний обсяг робіт по проекту. Це надає можливість грамотно розподілити порядок задач та ефективно спланувати дії.

Використовуючи BitBucket, ми могли зберігати наш репозиторій по проекту у хмарному сховищі та завжди мати доступ до нього у випадку технічних несправностей на наших персональних комп’ютерах, таким чином нічого не втративши. Окрім цього, користуючись ресурсами BitBucket ми могли змогу проводити код-рев’ю по зробленим тікетам, створюючи пул-реквести – запити на додавання нового коду до репозиторію по певному тікету. Це могли бути або новий функціонал, або баг-фікс та інше.

#### Висновки до пункту 2

За отриманими проектними рішеннями можна розробляти програмний код для реалізації застосунку.

Найбільш ключовим рішенням, особливо як для групової (парної) роботи слід назвати застосування системи контролю версій Git, а також програмне забезпечення Jira, завдяки якому уся робота контролюється та чітко відображена у реальному часі. Саме із подібним підходом можна говорити про грамотну організацію роботи над проектом.

Насамперед, важливо зазначити те, що саме по собі проектне рішення має багато шляхів розвитку, так як сама програма не є фінальною версією, і має ще безліч ідей по реалізації різних аспектів музики – квінтове коло, розрішення акордів та нот та інших. Зокрема слід зазначити ще й можливість додавання теоретичного матеріалу прямо у додаток, що може допомогти користувачу зрозуміти всю суть музичної теорії напряму у програмі.

Також, після проектування та перших реалізацій проекту, з’являлось багато зауважень щодо власної роботи, а саме необхідність у рефакторінгу коду та деяких реструктурізацій коду, що можливо зробити у майбутньому. Ці зауваження були занесені до системи Jira, аби не загубити їх.

# 3 ТЕСТУВАННЯ ТА НАЛАГОДЖЕННЯ

#### 3.1 Тестування методом «білої скриньки»

Дана робота була виконана більше практично по бек-енд частині – створення логічних модулів, їх налагодження. Для тестування функціоналу було застосовано підхід використання юніт-тестів, що детально тестують кожен модуль окремо. Усього система налічує 20 одиниць юніт-тестів, з них – 3 тести створено студентом Лисенко Максимом (див. дипломну роботу студента Максима Лисенко, розділ «3. Тестування та налагодження»).

Серед юніт тестів виділяються наступні, подані нижче у таблиці 3.1.

Таблиця 3.1 – Перелік юніт тестів логічних модулів

|  |  |  |  |
| --- | --- | --- | --- |
| Назва тестового випадку | Короткий опис | Очікуваний результат | Статус |
| Обробка коректних нот  (ElegiaNotesIdentifierTest.CorrectNote) | Згенеровані ноти співпадають із запропонованими користувачем sвідповідями | Функція перевірки повертає true | Пройдено |
| Обробка некоректних нот  (ElegiaNotesIdentifierTest.WrongNote) | Згенерована нота не співпадає із запропоновою користувачем відповіддю | Функція перевірки повертає false | Пройдено |
| Перевірка гарантії неповторності ноти підряд  (ElegiaNotesIdentifierTest.WrongNote) | Зі 100 згенерованих нот | Кожні дві підряд не будуть повторюватись | Пройдено |
| Побудування тональності за | При побудові тональності | Побудована тональність | Пройдено |

Продовження таблиці 3.1

|  |  |  |  |
| --- | --- | --- | --- |
| параметрами тоніки, знака альтерації та ладу. |  | відповідає дійсній тональності |  |
| Генерація тональності і її вгадуванян з вірною відповіддю | Згнерована тональність співпадає із запропонованою користувачем відповіддю | Функція перевірки повертає true | Пройдено |
| Генерація тональності і її невгадування з невірною відповіддю | Згнерована тональність не співпадає із запропонованою користувачем відповіддю | Функція перевірки повертає false | Пройдено |
| Ввід енгармонічної тональності після генерації | Коли вводиться енгармонічна тоніальність | Відповідь також вважається вірною | Пройдено |
| Перевірка гарантії неповторності тональності підряд | Зі 100 згенерованих тональностей | Кожні дві підряд не будуть повторюватись | Пройдено |
| Виконання конвертацій із Degree enum до std::string та навпаки | Виконується конвертація із enum/std::string | Успішно повертається конвертований std::string/enum | Пройдено |
| Виконання конвертацій із Octave | Виконується конвертація із | Успішно повертається | Пройдено |

Продовження таблиці 3.1

|  |  |  |  |
| --- | --- | --- | --- |
| enum до std::string та навпаки | enum/std::string | конвертований std::string/enum |  |
| Виконання конвертацій із AlterationSign enum до std::string та навпаки | Виконується конвертація із enum/std::string | Успішно повертається конвертований std::string/enum | Пройдено |
| Виконання конвертацій із Note enum до std::string та навпаки | Виконується конвертація із enum/std::string | Успішно повертається конвертований std::string/enum | Пройдено |
| Виконання конвертацій із Key enum до std::string та навпаки | Виконується конвертація із enum/std::string | Успішно повертається конвертований std::string/enum | Пройдено |
| Інкрементація структури Note | Викликана інкрементація operator++ | Нота інкрементована корректно | Пройдено |
| Декрементація структури Note | Викликана декрементація operator-- | Нота декрементована корректно | Пройдено |
| Порівняння структури Note | Викликані оператори порівняння operator> та operator< | Ноти порівнюються вірно | Пройдено |
| Рандомізація цілого 32-бітного числа | Викликана функція рандомізації 10 разів | Результат знаходиться у заданих межах | Пройдено |

Тестування із використанням юніт-тестів показало надійність та коректність логічних процесів модулів бек-енду. Протестовано модулі функціоналу Note Identifier, Tonality Builder, Random, функції конвертації перечислень та перевантажені оператори Note.

#### 3.2 Тестування методом «чорної скриньки»

Так як дана робота реалізована практично лише по частині бек-енду, тестування методом чорної скриньки не інформативне з точки зору виявлення помилок. Тестування методом чорної скриньки використовується для тестування взаємодії інтерфейсу та користувача і перевірки виконання основних функціональних складових (див. у дипломній роботі Лисенка Максима у пункті «3.2. Тестування методом чорної скриньки»).

## Висновки до пункту 3

Невід’ємною частиною розробки програмних продуктів є їх тестування. На данному етапі, тестування було виконано шляхом написання юніт-тестів, а саме використовувались GTest (Google Test) тести, які допомагали протестувати продукт у різних сценаріях, зімітованих у самих тестах. Це дозволяє ще на етапі розробки виявити деякі баги та помилки під час розробки та ліквідувати їх.

# ВИСНОВОК

Сьогоді є безліч засобів з навчання музичній теорії та практиці. Це є як мобільні додатки, так і веб-ресурси. Кожен з них надає допомогу людині у тому, аби отримати певні навички у музиці. Тобто, музична освіта не обмежується одними книжками та підручниками, що робить процес навчання більш цікавим та захоплюючим.

Під час розробки даного програмного забезпечення, було прораховано потреби користувача у тому, аби візуалізовувати музичні тональності на клавішах фортепіано. Водночас було пропрацьовано можливість прослуховувати тональність, а саме ноти що знаходяться у ній.

Те саме стосується і ідентифікатора нот. Вміння читати ноти з нотного стану на листку є необхідним для більшості музикантів, а вміння робити це швидко – ще важливіше. Відточити ці навички допоможе саме ідентифікатор, що не тільки вимальовує ноту на нотному стані, а ще й програє її звук, що допомагає тренувати абсолютний слух користувачу.

Після розробки, на думку спало багато ідей для розвитку проекту. Наприклад, ті ж сами музичні тональності, що візуально зображуються на віртуальному піаніно, можна було б також відображувати на інших інструментах: гітара, бас-гітара, флейта, скрипка та інші. Це розширило б коло музикантів, які користувалися б програмою та мали б за основний інструмент перелічені інструменти. Ще потрібно відмітити те, що можна реалізувати квінтове коле, що допомагає музиканту створювати музичні твори, знаючи, на які ступені можна переносити твір під час програвання, і що буде звучати гармонічно. В тому числі, можна додати теоретичну частину прямо у додаток, завдяки якій користувач матиме ще більш розгорнуту інформацію по музичній теорії.

# ЛІТЕРАТУРА

1. І. В. Способін. Елементарна теорія музики [Текст] : підручник для музичних училищ / І. В. Способін. – М. : Державне музичне видавництво 1963. – 202 с.
2. Лисенко М. О. Розробка застосунку «Музична школа»: тренажер музичного слуху: дипломна робота на здобуття кваліфікаційного ступеня бакалавра : спец. 121 – Інженерія програмного забезпечення / наук. Керівник І. В. Шаповал ; Український державний університет науки та технологій – Дніпро, 2023. – 142 с.
3. Johan Telin, Jürgen Bocklage-Ryannel, Cyril Lorquet. QT Framework Documentation [Електронний ресурс] / Johan Telin, Jürgen Bocklage-Ryannel, Cyril Lorquet : документація – Режим доступу: URL: <https://doc.qt.io/>
4. Diego Rodriguez-Losada Gonzalez, Luis Martínez de Bartolomé. Conan C++ Package Manager Documentation [Електронний ресурс] / Diego Rodriguez-Losada Gonzalez, Luis Martínez de Bartolomé : документація – Режим доступу: URL: <https://docs.conan.io/2/>
5. Alexey Pavlov, Martell Malone, Ray Donnelly. MSYS2 Package Management Documentation [Електронний ресурс] / Alexey Pavlov, Martell Malone, Ray Donnelly : документація – Режим доступу: URL: <https://www.msys2.org/docs/package-management/>
6. Matias Capeletto. Boost C++ Library Documentation [Електронний ресурс] / Matias Capeletto : документація – Режим доступу: URL: <https://www.boost.org/doc/libs/>
7. Victor Zverovich. FMT C++ Library API [Електронний ресурс] / Victor Zverovich : документація – Режим доступу: URL: <https://fmt.dev/latest/api.html>
8. Edward Lowinsky. Tonality and atonality in sixteenth-century music [Текст] / Edward Lowinsky. – Berkeley : University of California Press 1961. – 101 с.

# ДОДАТКИ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | ЗАТВЕРДЖУЮ Проректор Українського державного університету науки і технологій \_\_\_\_\_ Анатолій РАДКЕВИЧ 07.12.2022 |  |  |
|  | |  |  |  |
| Розробка застосунку «Музична школа»: розробка інструментів для формування основних елементів музики | | |  |  |
| Текст програми ЛИСТ ЗАТВЕРДЖЕННЯ  1116130.01305-01 12 01-ЛЗ | | |  |  |
|  | |  |  |  |
|  | | Представники підприємства-розробника Завідувач кафедри КІТ 07.12.22 |  | |
|  | Керівник розробки 07.12.22 | | |  |
|  | Виконавець  07.12.22 | | |  |
|  | Нормконтролер \_\_\_\_\_ Світлана ВОЛКОВА 07.12.22 | | |  |

2023

# ДОДАТОК А – ТЕКСТ ПРОГРАМИ

ЗАТВЕРДЖЕНО

1116130.01305-01 12 01-ЛЗ

Розробка застосунку «Музична школа»: тренажер музичного слуху

Текст програми

1116130.01305-01 12 01

Листів 23

2023

# ТЕКСТ ПРОГРАМИ

backend/CMakeLists.txt:

cmake\_minimum\_required(VERSION 3.24.1)

project(ElegiaBE VERSION 0.1.0)

if (NOT EXISTS BACKEND\_FOLDER\_PATH)

    set(BACKEND\_FOLDER\_PATH "${CMAKE\_CURRENT\_SOURCE\_DIR}" CACHE PATH "Location of backend folder")

    message("Backend folder path is: ${BACKEND\_FOLDER\_PATH}")

endif()

list(APPEND CMAKE\_MODULE\_PATH "${CMAKE\_CURRENT\_SOURCE\_DIR}/cmake")

# Conan package manager (for libraries like fmt, boost, etc...)

set (CONAN\_CMAKE\_FILE\_PATH "${BACKEND\_FOLDER\_PATH}/cmake/conan.cmake")

message("Looking for conan.cmake file in ${CONAN\_CMAKE\_FILE\_PATH} ...")

# GTest

include(GoogleTest)

include(${CONAN\_CMAKE\_FILE\_PATH})

# Downloaded Conan libs

conan\_cmake\_run(CONANFILE conanfile.txt BASIC\_SETUP CMAKE\_TARGETS BUILD missing PROFILE default)

include(${BACKEND\_FOLDER\_PATH}/build/conanbuildinfo.cmake)

conan\_basic\_setup()

# Tests enable variable declaration

option(ENABLE\_TESTING "Unit tests building/configuration" ON)

# CMake GTest enabling if needed

if (ENABLE\_TESTING)

    add\_definitions(-DENABLE\_TESTING)

endif ()

add\_subdirectory(utils)

add\_subdirectory(data)

add\_subdirectory(elegia)

add\_subdirectory(app)

if (ENABLE\_TESTING)

    include(CTest)

    enable\_testing()

    add\_subdirectory(test)

endif()

backend/utils/CMakeLists.txt:

cmake\_minimum\_required(VERSION 3.24.1)

project(utils LANGUAGES CXX)

set(${PROJECT\_NAME}\_HEADERS

    include/Random.h

)

set(${PROJECT\_NAME}\_SOURCES

    src/Random.cpp

)

set(${PROJECT\_NAME}\_MISC

    src/pch.h

)

add\_library(${PROJECT\_NAME} STATIC

    ${${PROJECT\_NAME}\_SOURCES}

    ${${PROJECT\_NAME}\_HEADERS}

    ${${PROJECT\_NAME}\_MISC}

)

add\_library(elegia::utils ALIAS ${PROJECT\_NAME})

target\_compile\_features(${PROJECT\_NAME} PUBLIC cxx\_std\_17)

target\_include\_directories(${PROJECT\_NAME}

    PRIVATE

        ${CMAKE\_RUNTIME\_OUTPUT\_DIRECTORY}

        ${CMAKE\_CURRENT\_SOURCE\_DIR}/src

    PUBLIC

        ${CMAKE\_CURRENT\_SOURCE\_DIR}/include

)

target\_link\_libraries(${PROJECT\_NAME}

    PUBLIC

        ${CONAN\_LIBS}

)

# Precompiled header

target\_precompile\_headers(${PROJECT\_NAME} PRIVATE src/pch.h)

backend/test/CMakeLists.txt:

cmake\_minimum\_required(VERSION 3.24.1)

project(test LANGUAGES CXX)

set(SOURCES

    ElegiaNotesIdentifierTest.cpp

    ElegiaScaleBuilderTest.cpp

    ElegiaTranposerTest.cpp

    EnumConversionTest.cpp

    NoteTest.cpp

    RandomTest.cpp

)

set(MISC

    pch.h

)

include(FetchContent)

FetchContent\_Declare(

    googletest

    GIT\_REPOSITORY https://github.com/google/googletest.git

    GIT\_TAG        release-1.11.0

)

FetchContent\_MakeAvailable(googletest)

add\_library(GTest::GTest INTERFACE IMPORTED)

# Prevent overriding the parent project's compiler/linker

# settings on Windows

set(gtest\_force\_shared\_crt ON CACHE BOOL "" FORCE)

target\_link\_libraries(GTest::GTest INTERFACE gtest\_main gmock gmock\_main)

add\_executable(unit\_tests

    ${MISC}

    ${SOURCES}

    ${UTILS}

)

target\_include\_directories(unit\_tests

    PRIVATE

        ${CMAKE\_CURRENT\_SOURCE\_DIR}

)

target\_include\_directories(unit\_tests