**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**

**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ  
імені ІГОРЯ СІКОРСЬКОГО»**

**Навчально-науковий інститут прикладного системного аналізу**

**Кафедра системного проектування**

|  |  |
| --- | --- |
|  | До захисту допущено:  Завідувач кафедри  \_\_\_\_\_\_\_\_ Вадим МУХІН  «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_2022 р. |

**Магістерська дисертація**

**на здобуття ступеня магістра**

**за освітньо-науковою програмою**

**“Інтелектуальні сервіс-орієнтовані розподілені обчислювання”**

**зі спеціальності 122 "Комп'ютерні науки"**

**на тему: «**Інструментарій проектування структури мікросервісного проблемно-орієнтованого ПЗ**»**

Виконав (-ла):

студент (-ка) II курсу, групи ДА-01мн

Терещенко Олександр Ігорович \_\_\_\_\_\_\_\_\_\_

Керівник:

Доцент, к.т.н.,

Безносик Олександр Юрійович \_\_\_\_\_\_\_\_\_\_

Рецензент:

доцент кафедри математичних методів

системного аналізу ІПСА, д.т.н., доц.

Недашківська Надія Іванівна \_\_\_\_\_\_\_\_\_\_

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.

Студент (-ка) \_\_\_\_\_\_\_\_\_\_\_\_\_

Київ – 2022

**Національний технічний університет України**

**«Київський політехнічний інститут імені Ігоря Сікорського»**

**Навчально-науковий інститут прикладного системного аналізу**

**Кафедра системного проектування**

Рівень вищої освіти – другий (магістерський)

Спеціальність – 122 "Комп'ютерні науки"

Освітньо-наукова програма – "Інтелектуальні сервіс-орієнтовані розподілені обчислювання"

ЗАТВЕРДЖУЮ

Завідувач кафедри

\_\_\_\_\_\_\_ Вадим МУХІН

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_2022 р.

**ЗАВДАННЯ**

**на магістерську дисертацію**

**студенту**

                                Терещенку Олександру Ігоровичу

(прізвище, ім’я, по батькові)

1. Тема дисертації: «Інструментарій проектування структури мікросервісного проблемно-орієнтованого ПЗ», науковий керівник дисертації к. т. н., доцент Безносик О.Ю., затверджені наказом по університету від «\_\_\_»\_\_\_\_\_\_\_\_\_ 20\_\_ р. №\_\_\_\_\_

2. Термін подання студентом дисертації – 10 червня 2022 р.

3. Об’єкт дослідження

Архітектура мікросервісних додатків, її властивості.

4. Предмет дослідження

Інструментарій проектування структури мікросервісного проблемно-орієнтованого ПЗ.

5. Перелік завдань, які потрібно розробити.

    5.1. Розділ з відомостями про мікросервісну архітектуру.

 5.2. Існуючі шаблони, які використовуються для побудови мікросервісних додатків.

    5.3. Огляд існуючих інструментів для побудови діаграм. Реалізація інструментарію та побудова тестового мікросервісного додатку з використанням цього інструментарію.

    5.4. Описати стартап проект по даній темі.

6. Перелік графічного (ілюстративного) матеріалу: презентація за темою роботи

«Інструментарій проектування структури мікросервісного проблемно-орієнтованого ПЗ»

7. Орієнтовний перелік публікацій

Tereshchenko O. Development principles of secure microservices. Workshop on Cybersecurity Providing in Information and Telecommunication Systems 2021, Kyiv, Ukraine

8. Дата видачі завдання: 30.06.2021

Календарний план

|  |  |  |  |
| --- | --- | --- | --- |
| № з/п | Назва етапів виконання  магістерської дисертації | Строк виконання етапів магістерської дисертації | Примітка |
| 1. | Отримання завдання | 02.09 |  |
| 2. | Проходження практики, огляд наукових праць | 31.01 – 6.03 |  |
| 3. | Дослідження стану розвитку мікросервісної архітектури | 4.04 – 13.04 |  |
| 4. | Реалізація графічного інтерфейсу інструментарію | 14.04 – 27.04 |  |
| 5. | Реалізація серверної частини інструментарію | 28.04 – 6.05 |  |
| 6. | Розробка тестового мікросервісного додатку з використанням створеного інструментарію | 7.05 – 20.05 |  |
| 6. | Оформлення дисертації | 21.05 – 27.05 |  |
| 7. | Завершення кваліфікаційної роботи | 28.05 – 10.06 |  |

Студент О. І. Терещенко

Науковий керівник О. Ю. Безносик

**ЗМІСТ**

[ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ 10](#_Toc105082128)

[ВСТУП 11](#_Toc105082129)

[1 ОГЛЯД СТАНУ РОЗВИТКУ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ 13](#_Toc105082130)

[1.1 Шаблони декомпозиції програми на мікросервіси 14](#_Toc105082131)

[1.2 Шаблони взаємодії 15](#_Toc105082132)

[1.2.1 Стилі взаємодії мікросервісів 16](#_Toc105082133)

[1.2.2 Шаблони забезпечення надійності 18](#_Toc105082134)

[1.2.3 Шаблони виявлення сервісів 19](#_Toc105082135)

[1.2.4 Шаблони зовнішніх API 20](#_Toc105082136)

[1.3 Брокер повідомлень 22](#_Toc105082137)

[1.4 Шаблони запитування даних у мікросервісній архітектурі 25](#_Toc105082138)

[1.5 Шаблони розгортання сервісів 25](#_Toc105082139)

[1.5.1 Docker 26](#_Toc105082140)

[1.6 Шаблони спостерігальності 27](#_Toc105082141)

[1.7 Шаблони безпеки 28](#_Toc105082142)

[1.7.1 Вразливості веб-додатків 29](#_Toc105082143)

[1.7.2 Аутентифікація та авторизація 30](#_Toc105082144)

[1.7.3 Використання технології єдиного входу в мікросервісній архітектурі 30](#_Toc105082145)

[1.7.4 Використання шлюзу єдиного входу 31](#_Toc105082146)

[1.8 Актуальність створення інструментарію 32](#_Toc105082147)

[1.9 Висновок до розділу 1 33](#_Toc105082148)

[2 ОГЛЯД ЗАСОБІВ ПОБУДОВИ ІНСТРУМЕНТАРІЮ 34](#_Toc105082149)

[2.1 Бібліотека GoJS 34](#_Toc105082150)

[2.2 Бібліотека MxGraph 36](#_Toc105082151)

[2.3 Бібліотека Nomnoml 38](#_Toc105082152)

[2.4 Бібліотека Diagram.js 38](#_Toc105082153)

[2.5 Бібліотека Rappid / JointJS Core Library 39](#_Toc105082154)

[2.6 Порівняння бібліотек для побудови діаграм 40](#_Toc105082155)

[2.7 Фреймворк мікросервісної архітектури Moleculer 41](#_Toc105082156)

[2.8 Висновок до розділу 2 46](#_Toc105082157)

[3 ІМПЛЕМЕНТАЦІЯ ІНСТРУМЕНТАРІЮ 47](#_Toc105082158)

[3.1 Архітектура інструментарію 47](#_Toc105082159)

[3.2 Теоретична основа проектування діаграм мікросервісів та баз даних 49](#_Toc105082160)

[3.2.1 Діаграми взаємодії та структури мікросервісів 49](#_Toc105082161)

[3.2.2 Діаграми баз даних 49](#_Toc105082162)

[3.3 Стек технологій інструментарію 51](#_Toc105082163)

[3.4 Стек технологій згенерованого мікросервісного додатку 52](#_Toc105082164)

[3.5 Можливості інструментарію 52](#_Toc105082165)

[3.5.1 Загальний вигляд інструментарію 53](#_Toc105082166)

[3.5.2 Налаштування брокеру сервісів 56](#_Toc105082167)

[3.5.3 Налаштування логеру 59](#_Toc105082168)

[3.5.4 Налаштування механізму виявлення сервісів 60](#_Toc105082169)

[3.5.5 Налаштування балансувальника навантаження 62](#_Toc105082170)

[3.5.6 Створення мікросервісу 62](#_Toc105082171)

[3.5.7 Додавання методу до мікросервісу 64](#_Toc105082172)

[3.5.8 Налаштування кількості екземплярів мікросервісів 65](#_Toc105082173)

[3.5.9 Додавання API-шлюзу 66](#_Toc105082174)

[3.5.10 Зв’язки між мікросервісами 68](#_Toc105082175)

[3.5.11 Додавання метаданих 69](#_Toc105082176)

[3.5.12 Додавання бази даних для мікросервісу 70](#_Toc105082177)

[3.6 Висновок до розділу 3 72](#_Toc105082178)

[4 ТЕСТУВАННЯ ІНСТРУМЕНТАРІЮ 73](#_Toc105082179)

[4.1 Архітектура тестового додатку 73](#_Toc105082180)

[4.2 Створення додатку за допомогою інструментарію 73](#_Toc105082181)

[4.2.1 Бази даних створеного тестового додатку 76](#_Toc105082182)

[4.2.2 Розгортання тестового додатку 78](#_Toc105082183)

[4.3 Висновок до розділу 4 80](#_Toc105082184)

[5 РОЗРОБКА СТАРТАП ПРОЕКТУ 81](#_Toc105082185)

[5.1 Опис ідеї проекту 81](#_Toc105082186)

[5.2. Технологічний аудит ідеї проекту 82](#_Toc105082187)

[5.3. Аналіз ринкових можливостей запуску стартап-прοекту 83](#_Toc105082188)

[5.4. Розроблення ринкової стратегії проекту 91](#_Toc105082189)

[5.5. Рοзрοблення маркетингοвοї прοграми стартап-прοекту 94](#_Toc105082190)

[5.6. Висновки дο розділу 5 97](#_Toc105082191)

[ВИСНОВКИ 99](#_Toc105082192)

[Список використаних джерел 101](#_Toc105082193)

[Додаток А 103](#_Toc105082194)

[Додаток Б 105](#_Toc105082195)

[Додаток В 112](#_Toc105082196)

РЕФЕРАТ

НА МАГІСТЕРСЬКУ ДИСЕРТАЦІЮ

виконану на тему: Інструментарій проектування структури мікросервісного проблемно-орієнтованого ПЗ

студентом Терещенком Олександром Ігоровичем

Робота виконана на 116 сторінках, містить 63 ілюстрацій, 31 таблиці. При підготовці використовувалася література з 20 джерел.

**Актуальність теми**

Впродовж останніх декількох років мікросервісна архітектура отримує все більшу популярність серед розробників програмного забезпечення. Така популярність досягається завдяки ряду переваг, що надає ця архітектура. Але для того, щоб отримати надійні мікросервісні додатки слід застосовувати шаблони проектування, які дозволяють уникнути слабких місць в мікросервісах. Зараз існують десятки мікросервісних шаблонів, але вони є досить складними для інтеграції в створювані програмні продукти для розробників, які вперше працюють з мікросервісами. Нажаль, існуючі інструменти для побудови мікросервісних додатків є вузькоспеціалізованими. На сьогоднішній день на ринку відсутні інструменти, які б об’єднували в собі всі необхідні компоненти правильної мікросервісної архітектури. Тому виникла необхідність створення графічного інструментарію, який б максимально автоматизував створення структури мікросервісних додатків, залишаючи розробникам більше часу на реалізацію бізнес-можливостей.

**Мета і завдання дослідження**

Метою цієї роботи є продовженням циклу досліджень кафедрою теми створення програмного забезпечення з використанням мікросервісної архітектуру та автоматизації цих процесів.

Поставлена мета передбачає розв’язок цілої низки конкретних завдань, а саме:

* Дослідження існуючих шаблонів забезпечення певних функцій в мікросервісній архітектурі;
* Огляд наявних засобів для побудови інструментарію;
* Використовуючи отримані під час аналізу шаблони та засоби, створити інструментарій проектування структури мікросервісного проблемно-орієнтованого ПЗ;
* Провести тестування створеного інструментарію шляхом створення медичного додатку.

**Об’єкт та предмет дослідження**

Об'єктом дослідження є мікросервісна архітектура та її властивості. Предметом дослідження є практичне створення інструментарію проектування структури мікросервісного проблемно-орієнтованого ПЗ.

**Рішення поставлених завдань і досягнуті результати**

За результатами здійсненого дослідження було проаналізовано сучасний стан мікросервісної архітектури та розроблено інструментарій проектування структури мікросервісного проблемно-орієнтованого ПЗ, який підтримує найголовніші шаблони мікросервісної архітектури та дозволяє швидко розробляти надійні та високопродуктивні мікросервісні додатки. Працездатність створеного інструментарію була доведена на прикладі медичного застосунку.

**Наукова новизна одержаних результатів**

Розроблено інструментарій проектування структури мікросервісного проблемно-орієнтованого ПЗ.

**Ключові слова**

Мікросервіси, інструментарій, інфраструктура додатків, бази даних, контейнери, шаблони мікросервісної архітектури.

ABSTRACT FOR MASTER'S DISSERTATION

made on the topic: "The tool for designing the structure of microservice problem-oriented software"

student: Tereshchenko Oleksandr

The master's thesis is made on 116 pages containing 63 illustrations and 31

tables. When preparing the dissertation, literature from 20 sources was used.

**Actuality of the topic**

Over the past few years, microservice architecture has become increasingly popular among software developers. This popularity is achieved due to a number of advantages provided by this architecture. But in order to get reliable microservice applications, you should use design templates that avoid vulnerabilities in microservices. There are now dozens of microservice templates, but they are quite complex to integrate into software products for developers who are working with microservices for the first time. Unfortunately, the existing tools for building microservice applications are highly specialized. There are currently no tools on the market that combine all the necessary components of a proper microservice architecture. Therefore, there is a need to create graphical tools that would automate the creation of the structure of microservice applications, leaving developers more time to implement business logic.

**The purpose and objectives of the study**

The purpose of this work is to continue the department’s cycle of researches of software development using microservice architecture and automation of these processes.

This goal involves the solution of a number of specific tasks, namely:

* Research of existing patterns for providing certain functions in microservice architecture;
* Review of available tools for building the toolkit;
* Using the patterns and tools obtained during the analysis, to create the tool for designing the structure of microservice problem-oriented software;
* Conduct testing of the created tools by creating a medical application.

**Object and subject of research**

The object of research is microservice architecture and its properties. The subject of the research is the practical creation of the tool for designing the structure of microservice problem-oriented software.

**Solution of tasks and achieved results**

According to the results of the study, the current state of microservice architecture was analyzed and the tool for designing the structure of microservice problem-oriented software were developed, which supports the most important templates of microservice architecture and allows you to quickly develop reliable and high performance microservice applications. The efficiency of the created tool was proved on the example of medical application.

**Scientific novelty of the obtained results**

The toolkit for designing the structure of microservice problem-oriented software has been developed.

**Keywords**

Microservices, toolkit, application infrastructure, databases, containers, templates of microservice architecture.

# **ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ**

IPC (Inter Process Communications) – Міжпроцесна взаємодія.

RPC (Remote Procedure Call) – Виклик віддалених процедур.

HTTP (HyperText Transfer Protocol) – Протокол передачі гіпертексту.

JSON (JavaScript Object Notation) – Запис об'єктів JavaScript.

XML (Extensible Markup Language) – Розширювана мова розмітки.

REST (Representational State Transfer) – Передача репрезентативного стану.

API (Application Programming Interface) – Прикладний програмний інтерфейс.

CQRS (command query responsibility segregation) – Командні запити із поділом відповідальності.

# **ВСТУП**

Мікросервісна архітектура набуває значної популярності серед розробників програмного забезпечення та ця тенденція буде зберігатися в обачній перспективі. Мікросервісна архітектура – це стиль проектування, який розбиває програму на окремі сервіси з різними функціями, що має значну кількість переваг над монолітною архітектурою.

Можна виділити наступні переваги мікросервісної архітектури:

* Вона уможливлює безперервну доставку та розгортання великих, складних додатків. Це досягається завдяки тому, що сервіси мають невеликий розмір та прості в обслуговуванні;
* Сервіси масштабуються незалежно;
* Сервіси розгортаються незалежно один від одного;
* Мікросервісна архітектура забезпечує автономність команд розробників;
* Мікросервіси дозволяють експериментувати та впроваджувати нові технології;
* Мікросервіси дозволяють ізолювати можливі неполадки;

Але, як і кожна архітектура, мікросервісна архітектура має певний ряд недоліків [1]. Серед основних недоліків та проблем мікросервісної архітектури можна навести наступні:

* Важко правильно роздробити додаток на набір мікросервісів;
* Складно розробляти, тестувати та розгортати мікросервісний додаток, так як він є розподіленою системою;
* Розгортання функцій, що охоплюють кілька сервісів, потребує ретельної координації;
* Рішення про те, коли слід переходити на мікросервісну архітектуру є нетривіальним.

У 2018 році розробники оркестратора Camunda провели велике дослідження серед 354 компаній, щоб визначити цінність впровадження мікросервісів в проекти. Близько 63% респондентів зазначили, що перехід на мікросервіси допоміг їм покращити користувацький досвід, ефективність роботи співробітників, а також заощадити на інструментах розробки та інфраструктури. Компанії також визначили основні причини, що підштовхнули їх до переходу на мікросервіси: 64% респондентів відзначили кращу масштабованість, 60% – швидкість розробки, 54% – інноваційність, 54% – автономність команд розробників, 50% – більшу стійкість розроблених додатків [2].

Зазвичай розробники використовують допоміжні інструменти для автоматизації певних процесів розробки програмного забезпечення. Мікросервісні додатки не є виключенням. На ринку існують багато інструментів, що спрощують побудову мікросервісних додатків, але більшість з них є вузькоспеціалізованими, тобто надають реалізацію окремих компонентів мікросервісних додатків. Аналіз ринку показав відсутність інструменту, що спрощував побудову структури мікросервісного додатку. Створення подібного інструментарію має на меті автоматизувати побудову структури мікросервісного додатку, залишаючи за розробником лише реалізацію окремих бізнес-функцій мікросервісів.

Дану роботу присвячено створенню графічного інструментарію для побудови мікросервісних додатків, застосовуючи предметно-орієнтовний підхід до їх розробки, ціллю якого є спрощення та пришвидшення створення подібного програмного забезпечення.

# **1 ОГЛЯД СТАНУ РОЗВИТКУ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ**

Правильна реалізація мікросервісної архітектури є досить складною задачею, тому слід користуватися рядом шаблонів, які дозволять оминути можливі труднощі в процесі розробки та експлуатації мікросервісного додатку. Мікросервісна архітектура розвивається швидкими темпами, постійно пропонуються нові підходи до вирішення проблем, які формуються у вигляді мікросервісних шаблонів. Розглянемо сучасний стан розвитку мікросервісної архітектури, зокрема існуючі шаблони реалізацій певних функцій в мікросервісній архітектурі.

Шаблон проектування – це багаторазове вирішення проблеми, що виникає у певному контексті. Це ідея, яка виникає як частина реальної архітектури, а потім показує себе з кращого боку при проектуванні програмного забезпечення. Програмний шаблон вирішує архітектурну проблему, визначаючи ряд програмних елементів, що взаємодіють між собою [1].

Можна поділити шаблони на три рівні, а саме:

* Інфраструктурні шаблони – вирішують проблеми, які стосуються інфраструктури і не стосуються розробки;
* Шаблони рівня інфраструктури програми – призначені для інфраструктурних завдань, що впливають на розробку;
* Шаблони програми – вирішують проблеми, з якими стикаються розробники.

На рисунку 1 представлена загальна структура мови шаблонів мікросервісної архітектури, що описує різні проблемні області, на які розраховані шаблони.

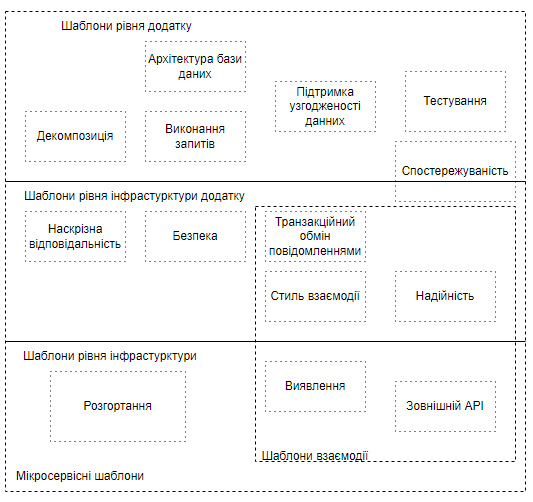


Рисунок 1. Загальна структура мови шаблонів мікросервісної архітектури, що описує різні проблемні області, на які розраховані шаблони

Розглянемо деякі групи шаблонів більш детально та визначимо на вирішення яких проблем вони націлені.

## **1.1 Шаблони декомпозиції програми на мікросервіси**

Рішення про те, яким чином розбити систему на набір сервісів є достатньо складним, але в цьому може допомогти цілий ряд стратегій. Існують два шаблони декомпозиції, які по-різному підходять до визначення архітектури додатку: по бізнес-можливостям (коли сервіси групуються на основі їх бізнес-можливостей) та по проблемним областям (згідно предметно-орієнтованому проектуванню). На рисунку 2 представлені ці два шаблони.

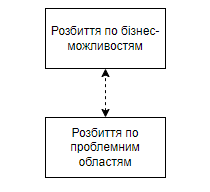


Рисунок 2. Шаблони розбиття програми на мікросервіси

Предметно-орієнтоване проектування майже ідеально підходить для мікросервісної архітектури. Концепції піддоменів та ізольованих контекстів, які використовуються в предметно-орієнтованому проектуванні, мають спільну основу з концепціями мікросервісної архітектури. Більш того, концепція автономних команд, кожна з яких відповідає за один мікросервіс, повністю відповідає предметно-орієнтованому проектуванню [4].

## **1.2 Шаблони взаємодії**

Додаток, що базується на мікросервісній архітектурі, є розподіленою системою. Важливу роль в цій архітектурі грає міжпроцесна взаємодія (interprocess communication, IPC) [1]. В залежності від того, як сервіси будуть взаємодіяти один з одним та із зовнішнім світом, розробникам потрібно прийняти низку архітектурних рішень. Шаблони взаємодії можна поділити на п'ять підгруп за такими характеристиками:

* Стиль взаємодії (який механізм IPC використовується);
* Виявлення (яким чином клієнт сервісу дізнається його IP-адресу, щоб, наприклад, виконати НТТР-запит);
* Надійність (яким чином забезпечується надійна взаємодію між сервісами беручи до уваги те, що деякі з них можуть бути недоступні);
* Транзакційний обмін повідомленнями (яким чином інтегрується публікація подій та надсилання повідомлень з транзакціями баз даних, які оновлюють бізнес-інформацію);
* Зовнішній API (яким чином клієнти програми взаємодіють із сервісами).

На рисунку 3 представлені шаблони, які входять в ці підгрупи.

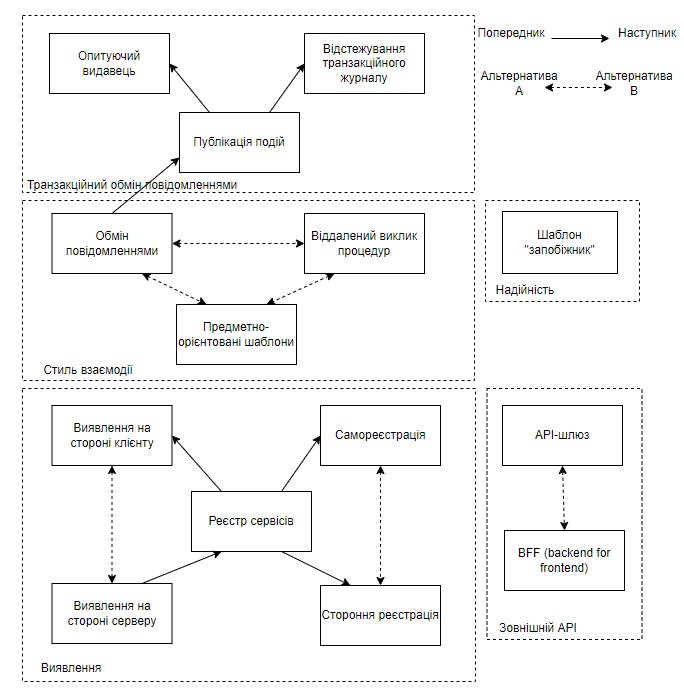


Рисунок 3. П’ять груп комунікаційних шаблонів

Розглянемо деякі з цих шаблонів більш детально.

### **1.2.1 Стилі взаємодії мікросервісів**

Сервіси можуть використовувати комунікаційні механізми на основі запитів/відповідей, такі як технологія передачі репрезентативного стану (REST) або gRPC поверх HTTP. У якості альтернативи є асинхронні механізми комунікації за допомогою повідомлень, такі як STOMP чи AMQP. Існує також безліч інших форматів повідомлень — це можуть бути текстові формати, зрозумілі людині (XML або JSON), так і більш ефективні двійкові, такі як Protocol Buffers або Avro [5].

Існує багато різних стилів взаємодії між клієнтом та сервісом. Як видно з таблиці 1, їх можна розділити на два рівні. Перший рівень визначає вибір між відносинами "один до одного" і "один до багатьох":

* "один до одного" — кожен клієнтський запит обробляється рівно одним сервісом;
* "один до багатьох" — кожен запит обробляється кількома сервісами.

Другий рівень визначає вибір між синхронною та асинхронною взаємодією:

* синхронна — клієнт може навіть заблокуватися на час очікування та розраховує на своєчасну відповідь від сервісу;
* асинхронна — відповідь може бути відправлена ​​не відразу, при цьому клієнт не блокується.

Таблиця 1. Різні стилі взаємодії

|  |  |  |
| --- | --- | --- |
| **Взаємодія** | **Один до одного** | **Один к багатьом** |
| Синхронна | Запит/відповідь | - |
| Асинхронна | Асинхронний запит/відповідь  Односпрямоване повідомлення | Видавець/підписник  Видавець/асинхронні відповіді |

Види взаємодії "один до одного" включають наступні:

* Запит/відповідь — клієнт надсилає сервісу запит і чекає на відповідь. Клієнт очікує, що відповідь прийде вчасно і може навіть заблокуватися на час очікування. Зазвичай, цей стиль взаємодії призводить до жорсткого зв'язку між сервісами;
* Асинхронний запит/відповідь — клієнт надсилає запит, а сервіс відповідає асинхронно. Якщо сервіс довго не відповідає, клієнт не блокується на очікування;
* Односпрямовані повідомлення – клієнт надсилає сервісу запит, не отримуючи нічого у відповідь.

Для взаємодії сервісів у стилі «запит/відповідь» можна використовувати як обмін повідомленнями, так і REST. Навіть якщо два сервіси залучать брокер повідомлень, клієнтський сервіс може блокуватися в очікуванні відповіді. Це зовсім не означає, що вони слабко пов'язані [1].

Різні види взаємодії "один до багатьох":

* Видавець/підписник — клієнт публікує повідомлення, яке споживається будь-якою кількістю сервісів, які зацікавлені в ньому;
* Видавець/асинхронні відповіді — клієнт публікує повідомлення із запитом і чекає певний час відповіді від зацікавлених сервісів.

Зазвичай, мікросервіси використовують поєднання цих стилів взаємодії.

### **1.2.2 Шаблони забезпечення надійності**

Найбільш популярними шаблонами забезпечення надійності є шаблони «Запобіжник», «Retry Policy» та «Backoff». Шаблони «Retry Policy» та «Backoff» доволі прості. Єдиною відмінністю між ними є те, що шаблон «Backoff» використовує адаптивні інтервали. Довжина інтервалу може змінюватися лінійно або експоненційно.

Шаблон «Запобіжник» націлений на помилки, які можуть виникати в мікросервісах та бути довготривалими. Такими помилками можуть бути проблеми мережі, програмного забезпечення, обладнання. Цей шаблон дозволяє запобігти повторним викликам, які скоріш за все закінчаться помилками, що дозволяє зберегти ресурси, наприклад пам’ять.

На рисунку 4 представлено алгоритм роботи шаблону «Запобіжник».

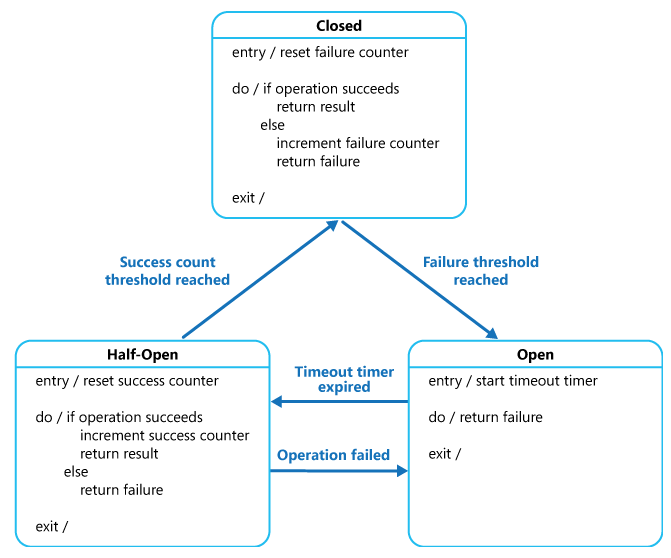


Рисунок 4. Алгоритм роботи шаблону «запобіжник»

У цього шаблону є 3 стани: closed, open, half-open [4].

1. Стан closed значить, що всі запити надсилаються сервісу. При цьому відбувається підрахунок кількості невдалих запитів проксі-сервісом. Коли ця кількість досягає якогось порогу, цей просі-сервер переводиться в стан open. Через деякий час проксі-сервіс переводиться в стан half-open для чергової спроби надіслати запит.
2. Коли проксі-сервіс знаходиться в стані open, сервісам миттєво повертається помилка.
3. Коли проксі-сервіс знаходиться в стані half-open лише обмежена кількість запитів надсилається. Цей стан існує для того, щоб не було різкого росту запитів до сервісів. Якщо, запити проходять вдало, то проксі сервіс переходить до стану closed, інакше до стану open.

### **1.2.3 Шаблони виявлення сервісів**

Для виконання запиту програмі необхідно знати розташування екземпляра сервісу в мережі (IP-адреса та порт). У традиційних додатках, що працюють на фізичному устаткуванні, мережеве місце розташування екземплярів сервісів зазвичай статичне. Розроблена програма, наприклад, могла б витягти мережеві адреси з конфігураційного файлу, який іноді оновлюється. Але в сучасних мікросервісних програмах, заснованих на хмарних технологіях, все може бути не так просто. Сучасна система набагато динамічніша.

Існують два основних способи реалізації механізму виявлення сервісів:

* Сервіси та їх клієнти безпосередньо взаємодіють із реєстром;
* За виявлення сервісів відповідає інфраструктура розгортання.

Одна з переваг підходу до виявлення сервісів на рівні програми, це те, що в ньому передбачена можливість розгортання сервісів на різних платформах. Можлива ситуація, наприклад, коли лише частина сервісів розгорнута на Kubernetes, інші ж працюють у застарілому середовищі. Виявлення сервісів на рівні програми з використанням Eureka охоплюватиме обидва середовища, тоді як рішення на базі Kubernetes сумісне лише з Kubernetes [1].

Один із недоліків цього підходу полягає в тому, що він вимагає наявності бібліотеки виявлення сервісів для кожної мови, а іноді і фреймворку, який використовується розробником.

Ключова перевага підходу, коли за виявлення сервісів відповідає інфраструктура розгортання, полягає у тому, що всіма аспектами виявлення сервісів займається сама платформа. Мікросервіси не вимагають жодного коду для цієї мети. Завдяки цьому виявлення мікросервісів доступне для всіх сервісів та клієнтів незалежно від мови або фреймворку, на яких вони написані. Один з недоліків способу пов'язаний з тим, що він підтримує виявлення тільки тих сервісів, які були розгорнуті на даній платформі [1].

### **1.2.4 Шаблони зовнішніх API**

У монолітних додатків існує єдиний API. Нажаль мікросервісна архітектура складніша, так як кожен мікросервіс має свій власний API. Тому велику увагу слід приділяти організації зовнішнього API і у цій задачі може знадобитися шаблон «API-шлюз».

API-шлюз – це сервіс, який є точкою входу в мікросервісний додаток для зовнішнього оточення, тобто для API-запитів до додатку з-за меж брандмауера. API-шлюз відповідає за маршрутизацію запитів, об'єднання API, а також може виконувати деякі допоміжні функції, зокрема автентифікацію.

По суті цей шаблон є об’єднанням двох класичних шаблонів, а саме шаблону адаптер та шаблону фасад. Так само як адаптер, API-шлюз дозволяє робити запити, навіть якщо інтерфейси несумісні. З точки зору мікросервісів, API-шлюз грає роль оркестратора.

Алгоритм роботи цього шаблону досить простий. Запити від зовнішніх клієнтів спочатку адресуються API-шлюзу, який спрямовує їх до конкретного мікросервісу. В рамках одного запиту від клієнту, API-шлюз може зробити декілька викликів методів різних мікросервісів та надіслати клієнту агреговану відповідь використовуючи об'єднання API. Він може також налагоджувати зв'язок між протоколами клієнтів, такими як HTTP і WebSockets, і внутрішніми протоколами сервісів, погано сумісними з клієнтами [5].

Однією з ключових функцій API-шлюзу є маршрутизація запитів. Коли API-шлюз отримує API-виклик, він надсилає запит відповідному мікросервісу, який він знаходить за допомогою таблиці маршрутизації. Ця таблиця визначає, якому мікросервісу слід надіслати запит. Таблиця маршрутизації може прив'язувати HTTP-метод та шлях до URL-адреси сервісу. Ця функція схожа на можливості зворотного проксі, які надають такі сервери, як NGINX [5].

На рисунку 5 представлено загальний вигляд архітектури API-шлюзу.

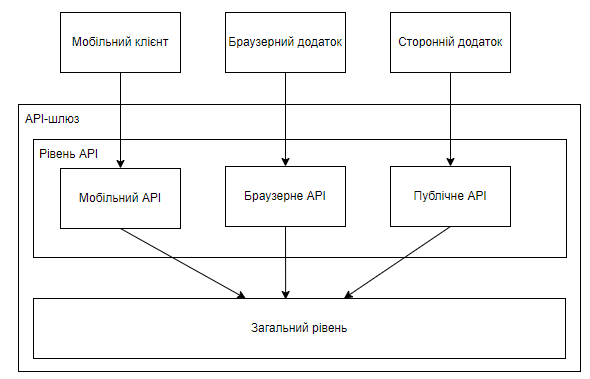


Рисунок 5. Архітектура API-шлюзу

Основними задачами API-шлюзу є маршрутизація та об'єднання API. При цьому, він може реалізовувати граничні функції. Граничною функцією називається операція обробки запитів на межі додатку. Прикладами граничних функцій є наступні:

* автентифікація — перевірка справжності клієнта, який робить запит;
* авторизація — перевірка того, що клієнту дозволено виконувати певну операцію;
* обмеження частоти запитів — контроль над кількістю запитів в секунду, які може виконувати певний клієнт або всі клієнти;
* кешування — кешування відповідей для зниження кількості запитів до сервісів;
* збір показників — збір показників про використання API для аналізу;
* ведення журналу запитів — запис запитів до журналу.

В великих проектах є доцільність використовувати цей шаблон, хоча він і створює потенційну єдину точку відмови додатку.

## **1.3 Брокер повідомлень**

Програми, що базуються на повідомленнях, зазвичай застосовують брокер повідомлень. Брокер повідомлень — це інфраструктурний компонент, через який мікросервіси спілкуються один з одним. Архітектура обміну повідомленнями може і не мати брокера, але у цьому випадку мікросервіси повинні взаємодіяти між собою напряму. На рисунку 6 представлено обидва ці підходи. Вони мають різні переваги і недоліки, але зазвичай рішення з участю брокера виявляються більш ефективними [6].

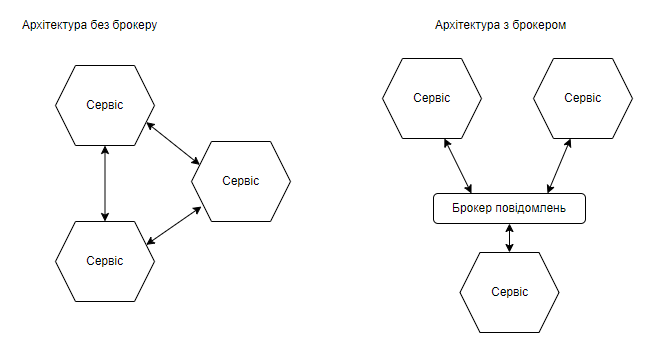


Рисунок 6. Взаємодія сервісів з брокером та без брокеру

Брокер виконує роль проміжної ланки, через який пересилаються усі повідомлення. Таким чином, відправнику повідомлення не потрібно знати мережеве розташування споживача повідомлення, що є значної перевагою цього підходу. Задача маршрутизації повідомлення вирішується брокером. Додатково, сучасні брокери буферизують повідомлення, поки одержувач не може їх обробити. Існує багато різних брокерів повідомлень. Далі перераховані популярні проекти з відкритим вихідним кодом:

* ActiveMQ;
* Apache Kafka;
* RabbitMQ;

При виборі брокера повідомлень слід враховувати різні фактори, включаючи такі:

* Стандарти обміну повідомленнями, що підтримуються. Треба впевнитись, що брокер підтримує стандарт на зразок AMQP чи STOMP;
* Порядок повідомлень. Треба брати до уваги підтримку брокером порядку повідомлень;
* Постійне зберігання. Треба перевірити чи зберігаються повідомлення на диску та чи можуть вони пережити збій брокера;
* Гарантія доставки. Треба визначитись які гарантії доставки надає брокер повідомлень;
* Підтримувані мови програмування. Краще вибрати брокер із підтримкою широкого діапазону мов програмування;
* Стійкість. Бажано щоб у випадку якщо споживач перепідключиться до брокера, він отримав повідомлення, надіслані, доки він був вимкнений;
* Латентність. Треба обирати брокер с найнижчою наскрізною латентністю;
* Масштабованість. Треба заздалегідь визначитись наскільки масштабуємо брокер повідомлень;
* Конкуруючі споживачі. Треба визначитись чи потрібен брокер повідомлень конкуруючих споживачів.

Обмін повідомленнями на основі брокера має багато переваг:

* Слабка зв’язність. Для виконання запиту клієнту потрібно лише надіслати повідомлення до відповідного каналу. Клієнту нічого не відомо про екземпляри сервісу, і йому не потрібно використовувати механізм виявлення, щоб визначити їхнє місцезнаходження.
* Буферизація повідомлень. Брокер буферизує повідомлення, доки їх не зможуть обробити. У протоколах із синхронними запитами/відповідями, такими як HTTP, і клієнт, і сервіс повинні бути доступні протягом усього обміну даними. Повідомлення ж накопичуються у черзі, поки споживач не буде готовий їх прийняти.
* Гнучка взаємодія. Обмін повідомленнями підтримує всі відомі стилі взаємодії.
* Явна міжпроцесна взаємодія. Механізми, засновані на RPC, намагаються зробити так, щоб звернення до віддаленого сервісу виглядало як виклик локальної процедури. Але закони фізики та ймовірність часткових відмов роблять ці два види взаємодії дуже різними. Обмін повідомленнями робить явні відмінності, щоб у розробників не виникало помилкове почуття безпеки.

У обміну повідомленнями на основі брокеру є деякі недоліки:

* Потенційна єдина точка відмови. Вкрай важливо, щоб брокер повідомлень був високодоступним, інакше може постраждати надійність системи. Добре, що більшість сучасних брокерів спроектовані з урахуванням високої доступності.
* Потенційне вузьке місце продуктивності. Є деякий ризик, що брокер повідомлень може стати вузьким місцем продуктивності. На щастя, більшість існуючих брокерів спроектовані з урахуванням високої масштабованості [7].
* Додаткова складність у адмініструванні. Механізм обміну повідомленнями – це ще один системний компонент, який потрібно встановлювати, конфігурувати та адмініструвати.

## **1.4 Шаблони запитування даних у мікросервісній архітектурі**

Використання окремої бази даних у кожному сервісі має один недолік: деякі запити повинні об'єднувати інформацію, що належить кільком сервісам. Дані сервісу доступні лише через його API, тому не можна застосовувати до його БД розподілені запити. На рисунку 7 представлено кілька шаблонів, за допомогою яких можна реалізувати запити.

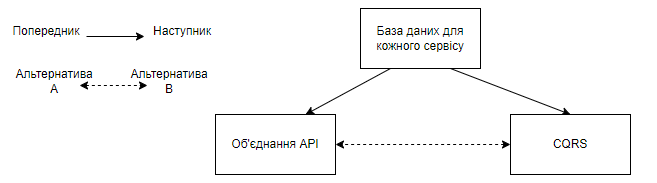


Рисунок 7. Шаблони для реалізації запитів

Іноді можна використовувати шаблон об'єднання API, який звертається до API одного або кількох сервісів та агрегує результати. У деяких ситуаціях доводиться вдаватися до командних запитів із поділом відповідальності (command query responsibility segregation, CQRS), які зберігають одну або кілька копій даних і дозволяють легко до них звертатися [8].

## **1.5 Шаблони розгортання сервісів**

Процес розгортання монолітного додатка не завжди простий, але він полягає в одній операції. Вона полягає в тому, що потрібно розгорнути лише один додаток. В той же час, екземпляри цього додатку будуть перебувати за балансувальником навантаження. Архітектура мікросервісної програми набагато складніша. В додатку можуть бути велика кількість сервісів (іноді ця кількість сягає сотень), написаних різними мовами з використанням різних фреймворків. Саме тому кількість елементів, якими потрібно керувати, більша. На рисунку 8 представлено шаблони розгортання.

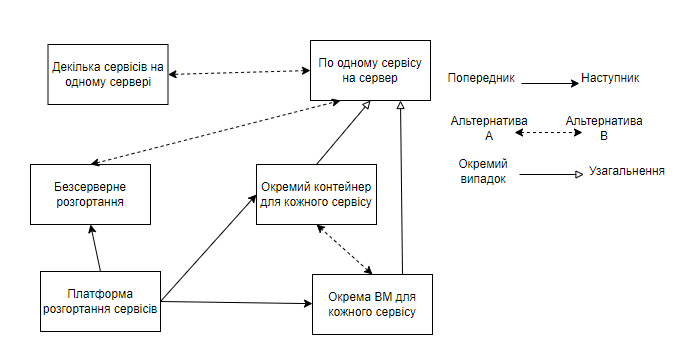


Рисунок 8. Шаблони розгортання мікросервісів

Традиційний спосіб розгортання програм із застосуванням форматів упаковки, який зазвичай ручний, не можна масштабувати для підтримки мікросервісної архітектури. Саме тому потрібна високоавтоматизована інфраструктура розгортання. В ідеалі слід використовувати платформу, яка дозволяє розробнику розгортати і адмініструвати свої сервіси за допомогою простого консольного або графічного інтерфейсу. Існує велика кількість платформ, які базуються на віртуальних машинах, контейнерах або безсерверних технологіях. У випадку інструментарію, будемо надавати у якості інфраструктури платформу Docker, яка базується на контейнерах [9].

### **1.5.1 Docker**

Однією з найбільш популярних платформ для розгортання та керування сервісами є Docker.

Docker — це платформа, що є надбудовою над полегшеними контейнерами. Ця платформа самостійно виконує великий об’єм роботи з управління контейнерами. За допомогою Docker створюються та розгортаються додатки. Для того щоб запустити створений мікросервіс в окремому контейнері, його треба запакувати в образ. Образом контейнеру називається файл, в якому зберігається додаток та все необхідне програмне забезпечення для його роботи. Docker керує наданням контейнерів, самостійно справляється з деякими проблемами використання мереж і надає реєстр, який дозволяє зберігати відомості про програми і фіксувати їх версії [3].

Для отримання переваг від застосування платформи Docker було розроблено кілька важливих технологій. Спеціально для Docker була розроблена операційна система CoreOS. Ця ОС представляє собою урізану до мінімуму ОС Linux, що надає лише найважливіші служби, що дозволяють працювати на платформі Docker. З цього можна зробити висновок, що вона споживає менше ресурсів, ніж інші операційні системи, дозволяючи виділяти машині, що використовується, ще більше ресурсів для контейнерів. Всі програми встановлюються як незалежні Docker-програми, кожна з яких запускається у власному контейнері, замість використання диспетчера пакетів, подібного до RPM- або deb-диспетчерів. Docker слід розглядати як просту PaaS-платформу, що працює на одній машині.

Для того, щоб керувати мікросервісами, які розташовані на різних контейнерах та машинах, слід використовувати додаткові інструменти, які додають такі можливості. До основних потреб відноситься рівень диспетчеризації, де в нагоді може бути технологія з відкритим кодом Kubernetes і наявні в CoreOS кластерні технології [7].

## **1.6 Шаблони спостерігальності**

Важливим аспектом адміністрування програмного забезпечення є розуміння того, як воно поводиться під час роботи, та діагностика таких проблем, як тривалий час очікування та невдалі запити. Монолітні програми не завжди легко діагностувати та аналізувати, але відносно простий механізм обробки запитів полегшує ці завдання. Балансувальник навантаження надсилає кожен вхідний запит до певного екземпляра програми, яка виконує невелику кількість запитів до бази даних і повертає відповідь. Наприклад, якщо потрібно зрозуміти, як був оброблений той чи інший запит, можна переглянути журнальний файл екземпляра програми, який цим займався. Нажаль, діагностика та аналіз проблем у мікросервісній додатках трохи складніший. Перш ніж клієнт отримає відповідь, запит може переадресовуватися від одного сервісу до іншого. Через це, відсутня можливість простежити цей процес у єдиному журнальному файлі. Крім цього, досить важко діагностувати час очікування, тому що корінь проблеми може бути в різних місцях [8]. Для проектування сервісів можна використовувати такі шаблони:

* Агрегація журналів (поведінка сервісів записується та зберігається на центральному сервері з підтримкою пошуку та оповіщень);
* API перевірки працездатності (створюється кінцева точка, яка повертає інформацію про стан сервісу);
* Відстеження винятків (надсилаються звіти про винятки відповідному сервісу, який сповіщає розробників та відстежує вирішення кожної виняткової ситуації);
* Показники програми (збираються кількісні та оціночні показники, які доступні для відповідного сервісу);
* Розподілене трасування (кожному зовнішньому запиту призначається унікальний ідентифікатор та відстежується його рух між сервісами);
* Ведення журналу аудиту (записуються в журнал дії користувачів).

## **1.7 Шаблони безпеки**

Для захисту мікросервісів розробнику потрібно подолати величезну кількість проблем пов’язаних з довірою та безпекою. Ці проблеми були успадковані від близьких родичів мікросервісної архітектури, таких як SOA та розподілених обчислень [8].

Важливо мати на увазі один важливий аспект: мікросервіси часто розраховані на те, щоб довіряти іншим мікросервісам у системі, тому компрометування та ефективна експлуатація одного з них може призвести до повного компрометування інших. У мікросервісних додатків є багато точок входу та їх захист є першочерговою задачею. Тому виникає питання: які існують ризики та як їх можна усунути при проектуванні мікросервісних додатків або мінімізувати їх наслідки?

### **1.7.1 Вразливості веб-додатків**

Мікросервіси представляють собою еволюцію стандартних комп’ютерних платформ. З цього випливає простий висновок: те, що було вразливістю раніше, залишається такою ж вразливістю в мікросервісному додатку. Для підвищення рівня ознайомленості з найпоширенішими уразливостями, розробникам слід познайомитись з десятьма найбільш критичними вразливостями з відкритого проекту забезпечення безпеки веб-додатків – списком OWASP Top Ten і середовищем тестування безпеки – OWASP Security Testing Framework.

Всі ці вразливості перераховані у наведеному нижче списку:

* Ін'єкції;
* Недоліки системи аутентифікації і зберігання сесій;
* Міжсайтові сценарії (XSS);
* Недоліки контролю доступу;
* Неправильна конфігурація безпеки;
* Незахищеність чутливих даних;
* Небезпечні прямі посилання на об'єкти;
* Підробка міжсайтових запитів (CSRF);
* Використання компонентів з відомими вразливостями;
* Недоліки журналювання і моніторингу.

У мікросервісах міжсерверні запити зазвичай безпечні, якщо застосовується глибоко ешелонована оборона за допомогою брандмауера або мережева сегрегація. Якщо ці аспекти не виконуються, в результаті можна отримати ситуацію підробки запиту на стороні сервера, при якій зловмисник може зловживати відносинами довіри між серверами, обходити білі списки IP-адрес, обходити служби автентифікації, зчитувати ресурси, які недоступні для загального користування, здійснювати взаємодію з API або отримувати конфіденційну інформацію, таку як IP-адресу веб-сервера [10].

### **1.7.2 Аутентифікація та авторизація**

Коли ми маємо справу з користувачами, велику роль відіграє аутентифікація та авторизація. Аутентифікація – це процес підтвердження того, що сторона, є тим, ким вона себе позиціонує. В свою чергу авторизація являє собою механізм, який класифікує та дозволяє певній особі дії, які вона може виконувати в системі [11].

У випадку монолітних систем, сам додаток бере на себе відповідальність за аутентифікацію та авторизацію. Такі веб-фреймворки, як Angular, Spring, Django, постачаються з вже готовою системою управління користувачами, що полегшує реалізацію цієї задачі в монолітних системах. Коли мова заходить про розподілені системи, через її складність доводиться шукати більш досконалі схеми. Треба уникати варіанту, коли користувачу доводиться реєструватися в кожній системі окремо, при цьому використовуючи різні паролі та імена користувачів. Виходом з цієї ситуації може бути створення та використання єдиного ідентифікатору, що дозволяє проходити аутентифікацію тільки один раз [13].

Особливістю міжсервісної комунікації у мікросервісній архітектурі є те, що мікросервіси повністю довіряють один одному. Серед багатьох можливих способів надійної автентифікації та авторизації сервісів – використання шлюзів єдиного входу [14].

### **1.7.3 Використання технології єдиного входу в мікросервісній архітектурі**

Розповсюдженим підходом до аутентифікації і авторизації є використання будь-якого з рішень єдиного входу. Відповідні можливості в даній області надаються за допомогою відкритого стандарту обміну даними аутентифікації та авторизації SAML і відкритого стандарту децентралізованої системи аутентифікації OpenID Connect [15].

SAML є стандартом на основі SOAP-протоколу, але він вважається досить складним, незважаючи на доступність великої кількості бібліотек і інструментальних засобів. OpenID Connect є стандартом, що виник в якості конкретної реалізації OAuth 2.0 і заснований на способах управління технологією єдиного входу, прийнятих Google і рядом інших компаній. У ньому використовуються прості REST-виклики.

Сервісом каталогів може бути будь-який засіб на кшталт Active Directory або Lightweight Directory Access Protocol (LDAP). Ці системи дозволяють зберігати інформацію про ролі користувачів в системі.

SAML та OpenID ідеально підходять для автентифікації та авторизації користувачів у системі, але також отримали розповсюдження для автентифікації між сервісами [12].

### **1.7.4 Використання шлюзу єдиного входу**

Замість того, щоб дублювати логіку управління підтвердженням входу на кожному сервісі, можна передати цю роботу окремому шлюзу, який буде працювати в якості проксі-сервера та розташовуватися між сервісами та зовнішнім оточенням. Таким чином можна локалізувати та централізувати логіку перенаправлення користувачів та виконання підтвердження в одному місці [9].

Але у цьому випадку виникає проблема, яка пов’язана з єдиною точкою відмови. На рисунку 9 представлена схема роботи шлюзу єдиного входу.

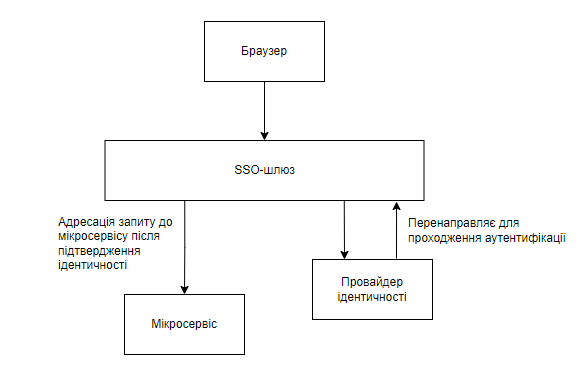


Рисунок 9. Шлюз єдиного входу

## **1.8 Актуальність створення інструментарію**

Існуючі мікросервісні інструменти можна розділити на декілька категорій відповідно до можливостей, які вони надають, а саме інструменти управління та тестування API, обміну повідомлень, моніторингу, оркестрації, тощо. Окремо можна виділити інструментарії, які об’єднують в собі інструменти з кількох категорій. Прикладом подібних інструментаріїв, які існують на ринку, є GoKit та Micro.

GoKit – інструментарій для створення мікросервісів на мові програмування Go. Серед його недоліків можна виділити те, що він має дуже обмежений перелік вбудованих мікросервісних шаблонів, зокрема не підтримує асинхронні методи комунікації на основі повідомлень.

Інструментарій Micro надає основні можливості для створення мікросервісів і керування ними. Він складається з набору бібліотек та інструментів, які в першу чергу орієнтовані на розробку на мові програмування Go. Серед його недоліків можна виділити відсутність механізмів масштабування мікросервісів.

Перелічені інструментарії вимагають від розробника знань та навичок в роботі з конкретними мовами програмування, бо вони спираються на «Full-code» розробку. В свою чергу, на ринку відсутні інструментарії для побудови структури мікросервісного програмного забезпечення, які б спирались на «No-code» та «Low-code» розробку. «Full-code» розробка має ряд недоліків, а саме вона вимагає більше часу, фінансування та ресурсів. З іншого боку, вона дозволяє створити унікальне та надійне програмне забезпечення. «No-code» розробка накладає певні обмеження на складність, можливість модифікації програмного забезпечення, але гарантує максимальну простоту створення програмного забезпечення. «Low-code» інструменти є збалансованим рішенням, бо вони пришвидшують та полегшують розробку, при цьому дозволяють модифікувати певні елементи програмного забезпечення для власних потреб. На опанування «Full-code» засобів можуть знадобитися місяці. Водночас на опанування «Low-code» засобів знадобляться тижні або навіть дні.

Таким чином, найкращим варіантом інструментарію, є графічний «Low-code» інструментарій з широким набором підтримуваних мікросервісних шаблонів, який може допомогти розробникам-початківцям створювати мікросервісні додатки з використанням найкращих практик та мінімізувати витрати коштів та часу. Також, інструментарій можна використовувати в освітніх цілях.

## **1.9 Висновок до розділу 1**

В цьому розділі було проаналізовано сучасний стан розвитку мікросервісної архітектури. Зокрема було розглянуто основні шаблони, які слід застосовувати при розробці мікросервісних додатків. Існуючі шаблони мікросервісної архітектури були класифіковані у групи шаблонів відповідно до задач, які вони вирішують. Було розглянуто шаблони декомпозиції, взаємодії, розгортання, спостерігальності та безпеки. Особливу увагу було приділено групі шаблонів взаємодії, а саме шаблонам стилю взаємодії, забезпечення надійності, виявлення сервісів та зовнішнього API.

Цей аналіз дозволив сформувати набір засобів, необхідних для побудови надійних мікросервісних додатків. Реалізації цих шаблонів будуть задіяні при побудові інструментарію проектування структури мікросервісного проблемно-орієнтованого ПЗ.

Крім цього, було проаналізовано існуючі засоби створення мікросервісних додатків і обгрунтовано актуальність створення графічного «Low-code» інструментарію проектування структури мікросервісного програмного забезпечення.

# **2 ОГЛЯД ЗАСОБІВ ПОБУДОВИ ІНСТРУМЕНТАРІЮ**

Розглянемо інструменти, які можуть бути задіяні при побудові інструментарію. Створюваний інструментарій буде клієнт-серверним додатком. У якості мови програмування для інструментарію та вихідних мікросервісних додатків було обрано JavaScript. За даними State of the Developer Nation на 2022 JavaScript є найпопулярнішою мовою програмування в світі, а в останній час вона активно завойовує позиції у якості серверної мови програмування, зокрема для побудови мікросервісів. Саме тому її використання є актуальним в цій роботі [9].

Так як створюваний інструментарій є графічним, велику увагу слід приділити вибору бібліотеки для побудови діаграм.

Графічні бібліотеки для побудови діаграм можна умовно поділити на два типи:

1. Бібліотеки з вбудованою підтримкою моделей діаграм (наприклад, бібліотеки з вбудованими елементами для UML діаграми класів);

2. Низькорівневі бібліотеки, що мають базові можливості для побудови графів.

Серед бібліотек для побудови діаграм на мові JavaScript були обрані наступні популярні бібліотеки: GoJS, MxGraph, Diagram.js, Rappid / JointJS Core Library.

Існує невелика кількість бібліотек та фреймворків для створення мікросервісних додатків на мові програмування JavaScript. У якості фреймворку для спрощення побудови мікросервісних додатків було обрано фреймворк Moleculer, так як цей фреймворк надає найбільше можливостей у порівнянні з іншими.

## **2.1 Бібліотека GoJS**

GoJS — це JavaScript і TypeScript бібліотека для створення діаграм і графіків та керування ними. Бібліотека GoJS була створена у 2012 році технологічною компанією Northwoods Software, діяльність якої зосереджується на створенні функціональних графічних інтерфейсів користувача. Ця бібліотека є прикладом низькорівневої бібліотеки.

GoJS — це багатофункціональна, розширювана та гнучка бібліотека для JavaScript і TypeScript, що надає набір готових до використання функцій, шаблонів і макетів, які полегшують життя розробникам, забезпечуючи при цьому високий ступінь їх налаштування. За допомогою GoJS можна побудувати майже будь-який тип діаграм, від дуже простих діаграм до високоспецифічних промислових діаграм, систем SCADA, діаграм BPMN, медичних генограм та діаграм моделювання [10].

GoJS пропонує підтримку інтерактивності, надаючи такі функції, як перетягування, копіювання та вставлення об’єктів, контекстні меню, редагування тексту на місці, підказки, автоматичні макети, шаблони, прив’язку даних та моделі, палітри, обробники подій, і багато інших важливих функцій.

На рисунку 10 представлений приклад діаграми, створеної з використанням бібліотеки GoJS.

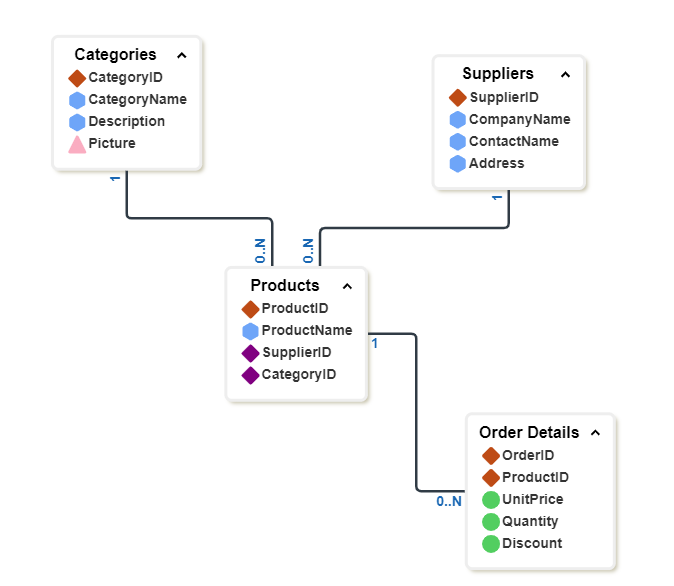


Рисунок 10. Приклад діаграми, створеної з використанням бібліотеки GoJS

Окремо слід виділити такі характеристики бібліотеки, як гнучкість і розширюваність. Розширюваність GoJS є результатом двох основних факторів. По-перше, бібліотека дає більше свободи за замовчуванням, не змушуючи розробників використовувати будь-які конкретні правила, як це робив б фреймворк. По-друге, її розширюваність полягає в тому, що це бібліотека, а не редактор для креслення діаграм, яких на ринку багато. Такі інструменти, як Draw.io або Miro, є потужними і допомагають нетехнічним користувачам представляти дані. Все ж таки їхні обмеження зрозумілі, коли справа доходить до створення будь-яких більш просунутих, індивідуальних рішень [10].

За допомогою GoJS можна створювати різноманітні програми, від досить простих програм до програм, які дозволяють запускати розширене моделювання, представляючи складні бізнес-правила або процеси моніторингу.

GoJS, завдяки добре оптимізованій продуктивності, дозволяє виконувати навіть найскладніші обчислення на інтерфейсі з оптимальною швидкістю, без необхідності налаштовувати додаткову інфраструктуру. Це, по суті, найбільш стабільний і безпечний варіант серед графічних бібліотек, враховуючи, що часто виникають проблеми з частою передачею великої кількості інформації між бекендом і фронтендом. В таблиці 2 наведений перелік переваг і недоліків бібліотеки GoJS.

Таблиця 2 — Переваги та недоліки бібліотеки GoJS

|  |  |
| --- | --- |
| **Переваги** | **Недоліки** |
| Є можливість кастомізувати будь-який елемент | Складний АРІ |
| Велика кількість прикладів | Безкоштовно можна використовувати лише в навчальних цілях, висока ціна ліцензії |
| Техпідтримка та спеціалізований форум |  |
| Легка інтеграція з React та Angular |  |
| Підтримка та постійний розвиток |  |

## **2.2 Бібліотека MxGraph**

MxGraph — це інтерактивна JavaScript бібліотека діаграм. MxGraph є повністю клієнтською бібліотекою, яка використовує SVG і HTML для візуалізації моделей. Ця бібліотека використовується, наприклад, у Draw.io. Бібліотека розробляється з 2005 року. MxGraph не використовує стороннє програмне забезпечення, не вимагає плагінів і може бути інтегрована практично в будь-який фреймворк [11]. На рисунку 11 представлений приклад діаграми, створеної з використанням бібліотеки MxGraph.

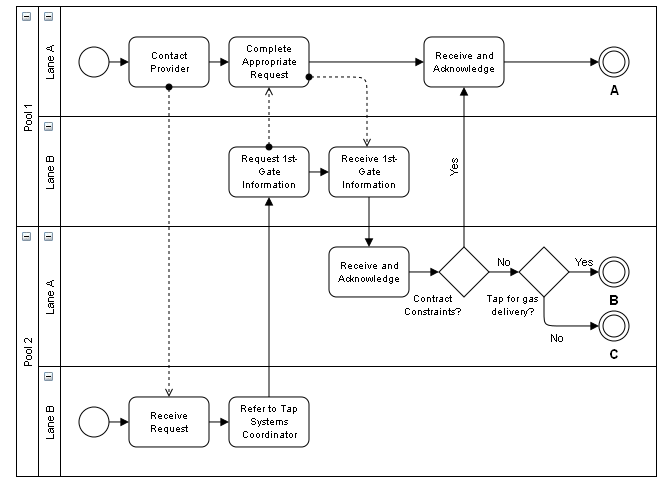


Рисунок 11. Приклад діаграми, створеної за допомогою MxGraph

Пакет mxGraph містить клієнтське програмне забезпечення, написане на JavaScript, і серію серверних програм для різних мов. Клієнтське програмне забезпечення є графічним компонентом з додатковою оболонкою програми, яка інтегрована в існуючий веб-інтерфейс. Клієнту потрібен веб-сервер для доставки необхідних файлів клієнту або його можна запустити з локальної файлової системи без веб-сервера. Серверні системи можна використовувати як є, або їх можна вбудувати в існуючу серверну програму, що розроблена з використанням однієї з підтримуваних мов (.NET, Java, Javascript) [11]. В таблиці 3 наведений перелік переваг і недоліків бібліотеки mxGraph.

Таблиця 3 — Переваги та недоліки бібліотеки mxGraph

|  |  |
| --- | --- |
| **Переваги** | **Недоліки** |
| Серверна частина підтримує .NET, Java, Javascript | Складний АРІ |
| Безкоштовна | Певний час не оновлювалась |
|  | Відсутня техпідтримка |

## **2.3 Бібліотека Nomnoml**

Nomnoml — це добре відомий інструмент текстового моделювання, який може відтворювати діаграми UML з текстових описів, але він також пропонує окрему бібліотеку JavaScript, яку можна використовувати для відтворення діаграм на власній веб-сторінці. Єдиними залежностями є lodash і dagre. В таблиці 4 наведений перелік переваг і недоліків бібліотеки Nomnoml.

Таблиця 4 — Переваги та недоліки бібліотеки Nomnoml

|  |  |
| --- | --- |
| **Переваги** | **Недоліки** |
| Невелика кількість залежностей | Використовує мову моделювання, яка не підходить для складних моделей |
| Постійно вдосконалюється розробниками | Відсутня детальна документація |
| Безкоштовна |  |

## **2.4 Бібліотека Diagram.js**

Diagram.js— це бібліотека для створення та відображення діаграм. Вона використовується розробниками BPMN.io, які створили її для власних потреб, як будівельний блок для ряду інших бібліотек тієї ж компанії для визначення моделей бізнес-процесів, моделей прийняття рішень і моделей плану роботи.

Наприклад, bpmn-js — це розширення diagram.js для відображення діаграм BPMN 2.0. Окрім використання його для створення редактора моделювання робочого процесу, bpmn-js був розроблений з урахуванням розширюваності, щоб розробники могли легко створити, наприклад, якийсь механізм виконання/симуляції поверх нього. В таблиці 5 наведений перелік переваг і недоліків бібліотеки Diagram.js.

Таблиця 5 — Переваги та недоліки бібліотеки Diagram.js

|  |  |
| --- | --- |
| **Переваги** | **Недоліки** |
| Багато розширень під конкретні задачі | Складно масштабувати та модифікувати |
| Безкоштовна | Відсутні деталізовані приклади використання |

## **2.5 Бібліотека Rappid / JointJS Core Library**

JointJS Core — це повноцінний набір інструментів для створення повністю інтерактивних програм для побудови діаграм і моделювання на основі JS і SVG. Ця бібліотека має багато вбудованих функцій для зручної роботи з графіками та діаграмами (численні готові до використання фігури, фільтри та градієнти, інтерактивні елементи, анімації, архітектура MVC, гнучка система плагінів, тощо) і може стати ідеальним рішення для проектів з відкритим кодом [12].

Але коли справа доходить до комерційних додатків, розробники можуть розглянути можливість використання Rappid, комерційного розширення JointJS Core. Воно засноване на JointJS і значно розширює її функціональність за допомогою численних додаткових компонентів, що дозволяє реалізовувати різні віджети інтерфейсу користувача, елементи взаємодії, автоматичні макети та інші корисні інструменти для створення найсучасніших візуальних додатків. Вичерпні навчальні посібники допоможуть глибше зрозуміти функціональність цього продукту та навчитися використовувати його найбільш ефективно. В таблиці 6 наведений перелік переваг і недоліків бібліотеки Rappid / JointJS Core Library.

Таблиця 6 — Переваги та недоліки бібліотеки Rappid / JointJS Core Library

|  |  |
| --- | --- |
| **Переваги** | **Недоліки** |
| Підтримує побудову блок-схем, BPMN діаграм, UML діаграм | Висока ціна ліцензії Rappid (комерційні ліцензії від 1648$) |
| Легко інтегрується з AngularJS, Backbone, React, Meteor | Безкоштовна версія JointJS має дуже обмежений функціонал |

## **2.6 Порівняння бібліотек для побудови діаграм**

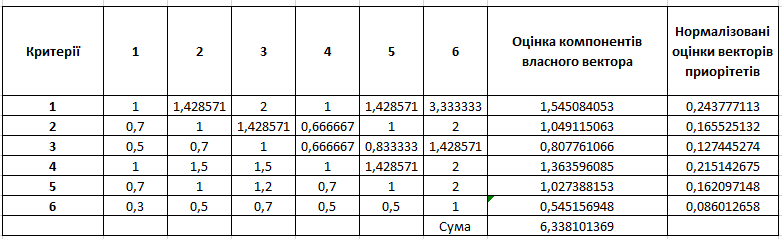
Будемо порівнювати ці бібліотеки за наступними критеріями та за допомогою методу аналізу ієрархії (МАІ). В таблиці 7 наведені критерії оцінки бібліотек.

Таблиця 7 — Критерії оцінки

|  |  |
| --- | --- |
| **№** | **Критерій** |
| 1 | Гнучкість |
| 2 | Легкість застосування |
| 3 | Продуктивність |
| 4 | Вартість продукту |
| 5 | Підтримка продукту |
| 6 | Розвиненість спільноти користувачів |

В таблиці 8 наведена матриця критеріїв з нормалізованою оцінкою пріоритетів для кожного критерію.

Таблиця 8 — Матриця критеріїв



Таблиці попарного порівняння бібліотек за критеріями наведені в додатку А. В таблиці 9 наведена результуючі дані з значеннями глобальних пріоритетів.

Таблиця 9 — Результуюча таблиця



Як бачимо, найвищий глобальний пріоритет у бібліотеки GoJS. Ця бібліотека є найбільш збалансованою, за всіма критеріями вона демонструє хороші показники. Отже, найкращим варіантом буде застосування бібліотеки GoJS в рамках цієї роботи.

## **2.7 Фреймворк мікросервісної архітектури Moleculer**

Moleculer — це сучасний та потужний фреймворк мікросервісної архітектури для Node.js. Останнім часом було розроблено багато портів цього фреймворку для різних мов програмування, зокрема Java, Go, Python та .NET. Moleculer надає розробникам реалізований набір функцій, таких як: збір метрик, кешування, балансування, відмовостійкість, вибір транспорту, валідацію параметрів, логування, лаконічне оголошення методів, кілька способів міжсервісної взаємодії, міксини та багато іншого [13].

Moleculer складається з трьох основних компонентів: Transporter, Service, ServiceBroker.

Transporter відповідає за виявлення сервісів та комунікацію між ними. Він передає події, запити та відповіді. Фреймворк надає 7 готових реалізацій інтерфейсу Transporter, які є частиною фреймворку, а саме TCP, Redis, AMQP, MQTT, NATS, NATS Streaming та Kafka. Великою перевагою Moleculer є те, що він не обмежує розробників в модифікації фреймворку, тому розробник з легкістю може реалізувати цей інтерфейс самостійно.

На практиці, при написанні коду розробники не взаємодіють з цим компонентом. Транспорт, що використовується, вказується в конфігураційному файлі. Таким чином, для переходу з одного транспорту на інший просто треба змінити конфігураційний файл.

Приклад налаштування транспортеру:

//./moleculer.config.js

module.exports = {

transporter: 'redis://:password@127.0.0.1:6422',

// ... інші параметри

}

За замовчуванням дані передаються у форматі JSON. При цьому, розробники можуть обрати й інші формати повідомлень, серед яких є як двійкові, так і текстові, зокрема Avro, MsgPack, Notepack, ProtoBuf, Thrift, і т.д.

Service це клас, від якого успадковуємося при написанні наших мікросервісів. Сервіс містить частину складної програми. Він ізольований і автономний, а це означає, що навіть якщо він перестане функціонувати через збій, інші сервіси не будуть вражені [13].

Найпростіший сервіс без методів, який буде виявлений іншими сервісами, виглядає наступним чином:

// ./services/car/car.service.js

const { Service } = require('moleculer');

module.exports=class CarService extends Service {

constructor(broker) {

super(broker);

this.parseServiceSchema({

name: 'car',

});

}

};

На рисунку 12 представлена структура класу ServiceBroker.

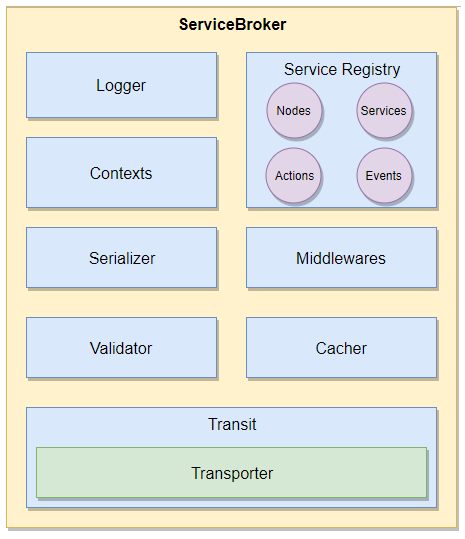


Рисунок 12. Структура класу ServiceBroker

ServiceBroker це ядро фреймворку. По суті, це прошарок між транспортом та сервісом. Коли один сервіс хоче взаємодіяти з іншим сервісом, він робить це через брокер. Брокер займається балансуванням навантаження (підтримує кілька стратегій, за замовчуванням — round-robin), врахуванням живих сервісів, доступних методів у цих сервісах, тощо. Для цього ServiceBroker під капотом використовує ще один компонент, а саме Registry. Фреймворк використовує шаблон виявлення сервісів на стороні клієнту [15].

Наявність брокера є дуже зручним. У контексті фреймворку є таке поняття як node. Простою мовою, node (вузол) — це процес в операційній системі. Кожен вузол — це ServiceBroker з набором з одного або декількох мікросервісів. Це надає розробникам більшу зручність. Для розробки ми запускаємо один вузол, в якому запускаються всі мікросервіси разом, всього один процес у системі з можливістю дуже легко підключити гаряче перезавантаження, наприклад. У проекті, готовому до масового використання, окремий вузол надається під кожен екземпляр сервісу або ці підходи поєднують: частина сервісів на одному вузлі, частина на іншому [16].

Приклад створення екземпляру ServiceBroker:

const broker = new ServiceBroker(config);

broker.loadServices(

resolve(\_\_dirname, 'services'),

SERVICES

? `\*/@(${SERVICES.split(',').map(i => i.trim()).join('|')}).service.js`

: '\*/\*.service.js',

);

broker.start().then(() => {

if (NODE\_ENV === 'development') {

broker.repl();

}

});

На рисунку 13 представлений принцип роботи фреймворку.

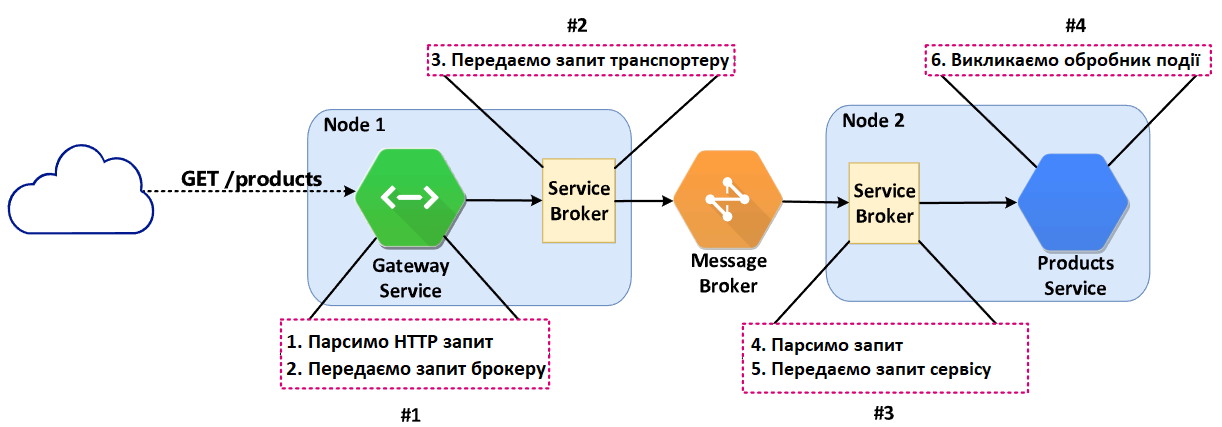


Рисунок 13. Принцип роботи фреймворку Moleculer

Фреймворк Moleculer має наступні способи міжсервісної взаємодії, а саме методи call, emit та broadcast. Метод call реалізує асинхронний запит. Виклики автоматично балансуються. Просто необхідно запустити необхідну кількість екземплярів сервісу, а за балансування буде відповідати сам фреймворк. Метод emit використовується коли треба повідомити інші сервіси, про якусь подію, але нам не потрібен результат. Сервіси можуть підписатися на конкретні події. Метод broadcast схожий на emit, але розсилає подію усім екземплярам сервісу [17].

Розробники Moleculer провели тестування різних фреймворків для побудови мікросервісних додатків для Node.js. Було розглянуто такі фреймворки: Moleculer, Nanoservices, Seneca, Hemera, Core. На рисунку 14 та рисунку 15 представлено отримані результати.

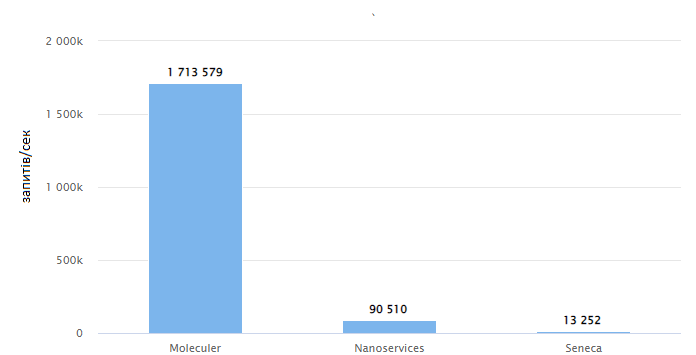


Рисунок 14. Порівняння різних фреймворків для побудови мікросервісів у випадку локальних запитів

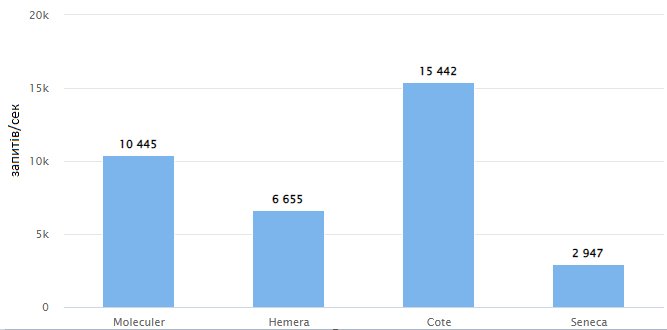


Рисунок 15. Порівняння різних фреймворків для побудови мікросервісів у випадку віддалених запитів (через транспортер)

Результати показують, що фреймворк Moleculer демонструє найкращу швидкість обробки запитів у випадку локальних запитів та другу у випадку віддалених запитів (тобто через транспортер) [18].

## **2.8 Висновок до розділу 2**

В цьому розділі було розглянуто засоби для побудови інструментарію. Розроблений інструментарій є графічним, тому велику увагу було приділено вибору бібліотеки для побудови діаграм. Зокрема були розглянуті такі доступні на ринку бібліотеки та фреймворки для JavaScript, як GoJS, MxGraph, Diagram.js, Rappid / JointJS Core Library. Використовуючи метод аналізу ієрархії та критерії оцінки, які важливі для надання користувачам інструментарію якісного досвіду користуванням, найкращим варіантом було обрано застосування бібліотеки GoJS. Ця бібліотека є гнучкою, відносно простою в застосуванні та активно розвивається.

У якості фреймворку для полегшення побудови мікросервісного додатку було обрано Moleculer, який є найбільш просунутим фреймворком на мові програмування JavaScript для вирішення подібних задач. Було проаналізовано структурні особливості цього фреймворку та порівняно швидкодію цього фреймворку серед інших засобів для побудови мікросервісних додатків.

# **3 ІМПЛЕМЕНТАЦІЯ ІНСТРУМЕНТАРІЮ**

Інструментарію проектування структури мікросервісного проблемно-орієнтованого ПЗ було надано назву MsConstructor, що підкреслює його призначення. В цьому розділі розглянута загальна архітектура створеного інструментарію, теоретична основа проектування схем мікросервісів та баз даних. Наостанок, детально розглянуто функціонал інструментарію.

## **3.1 Архітектура інструментарію**

Розроблений інструментарій є клієнт-серверною програмою. Клієнтська частина відповідає за користувацький інтерфейс, а серверна частина за генерування структури мікросервісного додатку. На рисунку 16 та рисунку 17 наведені компоненти, з яких складається серверна і клієнтська частини відповідно.

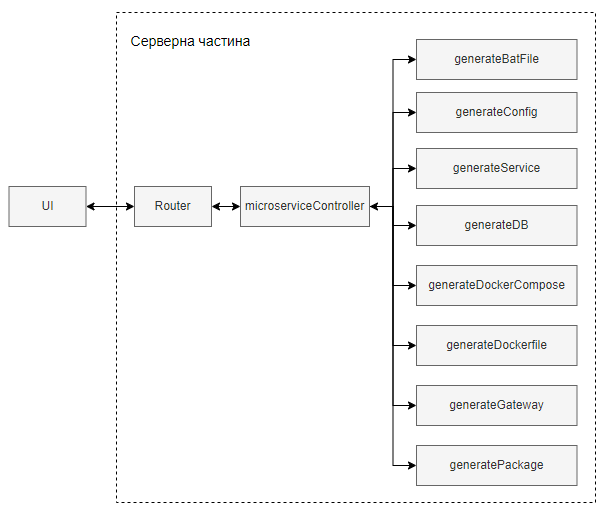


Рисунок 16. Основні компоненти, які складають серверну частину інструментарію

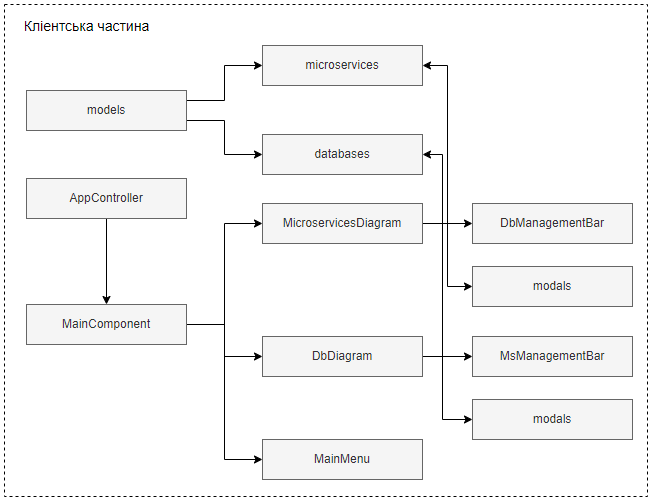


Рисунок 17. Основні компоненти, які складають клієнтську частину інструментарію

Інструментарій виконаний у вигляді веб-додатку, завдяки чому досягається його кросплатформність, миттєва оновлюваність та невеликий розмір.

Окремо слід відзначити компонент models, в який входять компоненти databases та microservices. Ці компоненти відповідають за визначення шаблонів вузлів та зв’язків діаграм. Крім цього в них визначаються моделі даних діаграм. При побудові діаграм був використаний широкий спектр можливостей бібліотеки GoJS, а саме панелі, групи, двосторонні зв’язування даних, контекстні меню, події, анімації та інше. В додатку Б наведений код моделі діаграм мікросервісів.

Клієнтська частина на виході генерує схему мікросервісного додатку, використовуючи формат JSON, та передає її серверу. Приклад згенерованої схеми наведений в додатку В. Об’єкт складається з обов’язкових полів main та options. В полі main наведена модель діаграми мікросервісів, а в полі options – налаштування всього мікросервісного додатку. Додатково, може надаватися моделі діаграм баз даних, якщо вони були створені.

## **3.2 Теоретична основа проектування діаграм мікросервісів та баз даних**

Розроблений інструментарій має два рівні діаграм, а саме діаграми мікросервісів та діаграми баз даних. Розглянемо структуру цих діаграм більш детально.

### **3.2.1 Діаграми взаємодії та структури мікросервісів**

Діаграми мікросервісів складаються з таких сутностей: мікросервісів, сховищ та проксі-сервісів. На рисунку 18 представлений загальний вигляд діаграми мікросервісів.

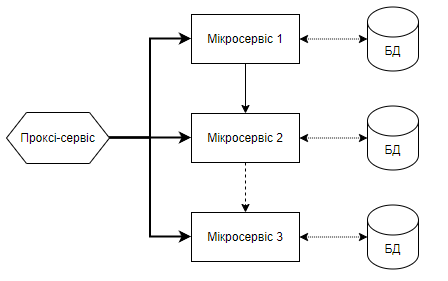


Рисунок 18. Діаграма мікросервісів

Існують декілька видів зв’язків між сутностями в діаграмі мікросервісів. Для зв’язку між мікросервісами використовують технології віддаленого виклику процедури (RPC) та асинхронного обміну даними на основі подій. Для зв’язку між проксі-сервісом та мікросервісами використовується технологія передачі репрезентативного стану (REST).

### **3.2.2 Діаграми баз даних**

Діаграми баз даних використовують схеми «сутність-зв’язок», також відомі як ER-діаграми. Побудовані діаграми покривають три рівні деталізації, а саме концептуальну, логічну та фізичну моделі даних. На логічному рівні виділяються транзакційні та операційні сутності системи. На фізичному рівні надаються деякі технічні подробиці для побудови бази даних. У випадку створеного інструментарію надаються технічні можливості для побудови баз даних з використання реляційної СУБД PostgreSQL. На рисунку 19 представлений приклад ER-діаграми бази даних.

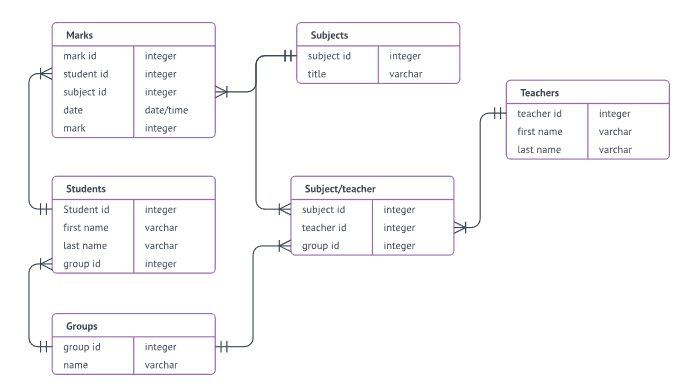


Рисунок 19. Приклад ER-діаграми бази даних

На фізичному рівні деталізації бази даних є можливість побачити всі дані про структуру таблиць. Ці дані включають в себе назви стовпців, обмеження стовпців, типи стовпців, первинні та зовнішні ключі. Крім цього, є можливість встановити типи зв’язків між таблицями.

Кожна таблиця складається з наступних складових: полів, ключів та обмежень. Полями називають частину таблиці, де задають певні атрибути сутностей. Прикладом поля є атрибут title, що відноситься до сутності Subject. Обмеження використовуються для перевірки даних перед їх вставкою. Прикладом обмежень є ключові слова NOT NULL та UNIQUE [19].

Ключі таблиці використовуються для категоризації атрибутів сутностей. За допомогою ключів виникає можливість ефективно зв’язати декілька таблиць між собою. Підтримуються два види ключів: первинні та зовнішні.

Первинні ключі (PK) створенні для ідентифікації конкретного екземпляру сутності. Зовнішній ключ (FK) створюються кожен раз, коли створюється зв’язок між сутностями та атрибутами [20].

Проектувальники можуть обрати наступні підтримувані СУБД PostgreSQL типи даних: integer, smallint, bigint, text, varchar, char, timestamp, date, boolean та інші.

Для позначення зв’язків використовується нотація Crow's Foot. Інструментарій підтримує три типи зв’язків:

* One-to-one;
* One-to-many;
* Many-to-many;

## **3.3 Стек технологій інструментарію**

Розглянемо інструменти та технології, які були задіяні при побудові інструментарію. Наступні інструменти та технології відповідають за створення інтерфейсу для побудови діаграм та генерації коду структури мікросервісного додатку:

1. NodeJS;
2. Express;
3. React;
4. Bootstrap;
5. GoJS;
6. Інші бібліотеки.

Програма інструментарію була розроблена за допомогою мови програмування JavaScript (ES2022). Серверна частина використовує платформу для виконання високопродуктивних мережевих застосунків NodeJS версії 14.18 та фреймворк Express версії 4.17 для побудови веб-додатків. Фреймворк Express надає програмний каркас для побудови серверу інструментарію. Для створення користувацького інтерфейсу використовується фреймворк React на стороні клієнту, а також фреймворк Bootstrap, який застосовується для створення адаптивних сайтів, орієнтованих на мобільні пристрої. Бібліотека GoJS надає можливість створювати інтерактивні діаграми. Крім цього використовуються допоміжні бібліотеки, зокрема бібліотеки для генерування архівів, тощо.

## **3.4 Стек технологій згенерованого мікросервісного додатку**

Мікросервісний додаток, отриманий в результаті генерація за допомогою інструментарію, побудований з використанням наступних технологій:

1. NodeJS;
2. Moleculer;
3. PostgreSQL;
4. Sequelize;
5. NATS;
6. Docker;
7. Docker-compose;
8. Інші бібліотеки.

Мовою згенерованої мікросервісної програми є JavaScript (ES2022). Програма використовує фреймворк мікросервісної архітектури Moleculer, який працює на платформі NodeJS. Цей фреймворк значно пришвидшує та спрощує створення мікросервісних додатків. Для роботи з базами даних використовуємо популярну реляційну систему керування базами даних PostgreSQL. Для роботи з цією СУБД використовуємо ORM Sequelize. У якості системи обміну повідомленнями з відкритим кодом використовуємо NATS, а у якості програмного забезпечення для розгортання та управління ізольованими контейнерами Docker. Для управління усіма контейнерами в яких розташовані мікросервіси будемо використовувати Docker-compose. Додатково використовуються бібліотеки для завантаження змінних середовища з файлів, тощо.

## **3.5 Можливості інструментарію**

Реалізований інструментарій має наступні можливості:

1. Створення проекту мікросервісного додатку з нуля;
2. Завантаження та редактування існуючої схеми мікросервісного додатку, завантаженої у форматі JSON;
3. Застосування різних існуючих шаблонів проектування мікросервісних додатків;
4. Генерація коду структури мікросервісного додатку на основі побудованої схеми;
5. Генерація SQL-файлів баз даних;
6. Розгортання мікросервісного додатку у контейнерах за допомогою Docker та Docker-compose.

При розробці велика увага приділялася простоті та зручності інтерфейсу користувача інструментарію, тому його було виконано в мінімалістичному стилі. Всі елементи інтерфейсу розташовані в інтуїтивно зрозумілих місцях. Функції інструментарію розбиті за категоріями. Швидкодія переходів між різними вікнами досягається завдяки використанню бібліотеки React, яка відома своєю гарною оптимізацією.

Розглянемо створений інструментарій більш детально.

### **3.5.1 Загальний вигляд інструментарію**

При запуску програми MsConstructor користувач бачить вітальне вікно, в якому він може обрати наступні опції: створити новий проект, або ж відкрити існуючий (дані проекту зберігаються в JSON форматі). Крім цього користувач може ознайомитися з можливостями інструментарію за допомогою відповідного пункту меню. На рисунку 20 представлено вигляд вітального вікна.

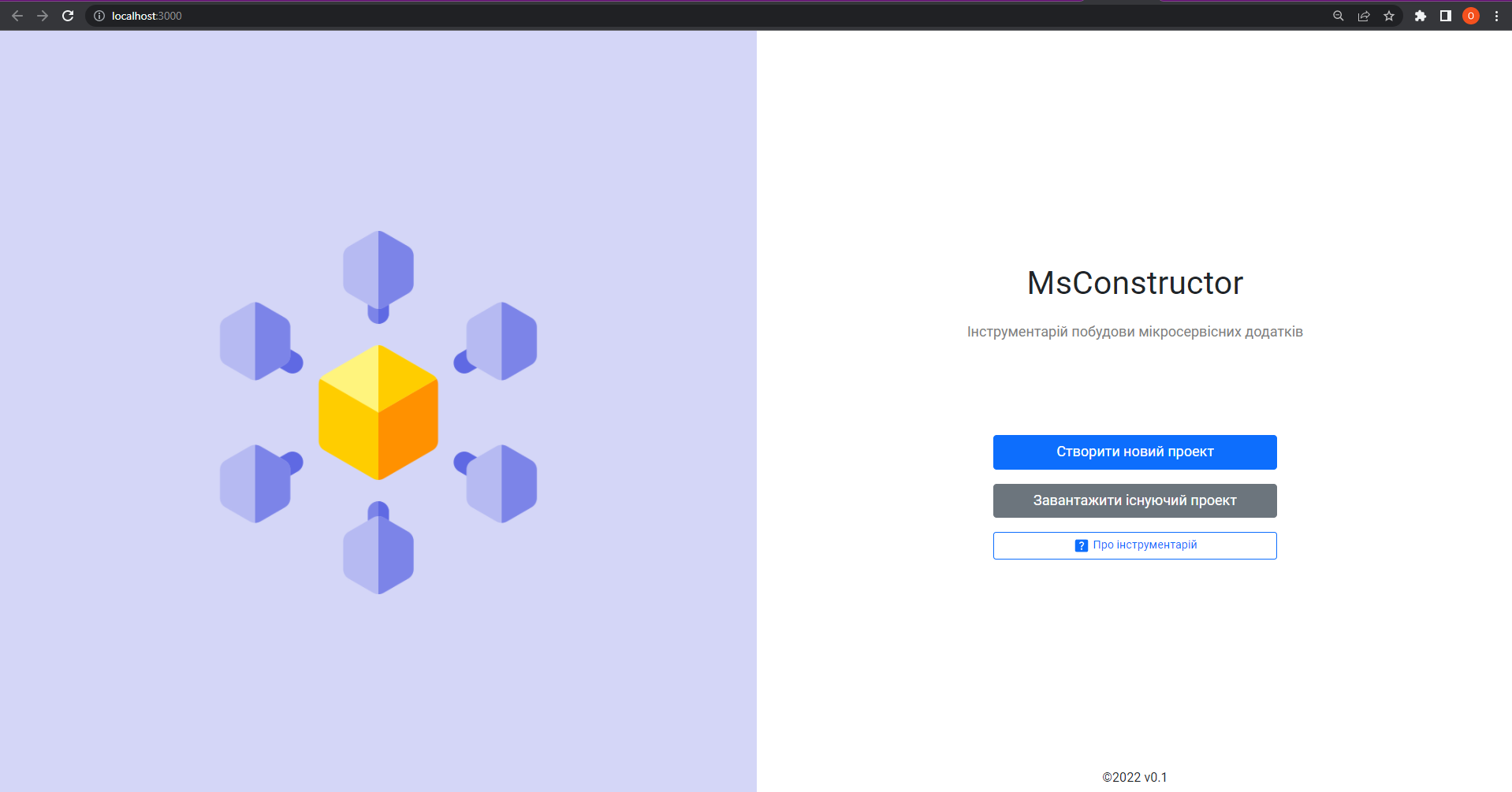


Рисунок 20. Вітальне вікно інструментарію

Після того, як користувач обрав необхідний варіант, відкриється основне вікно проектування інструментарію MsConstructor, яке складається з трьох основних компонентів: головної панелі інструментів, панелі інструментів діаграми та області, що відповідає за моделювання мікросервісного додатку. На рисунку 21 представлено вигляд вікна проектування нового проекту.

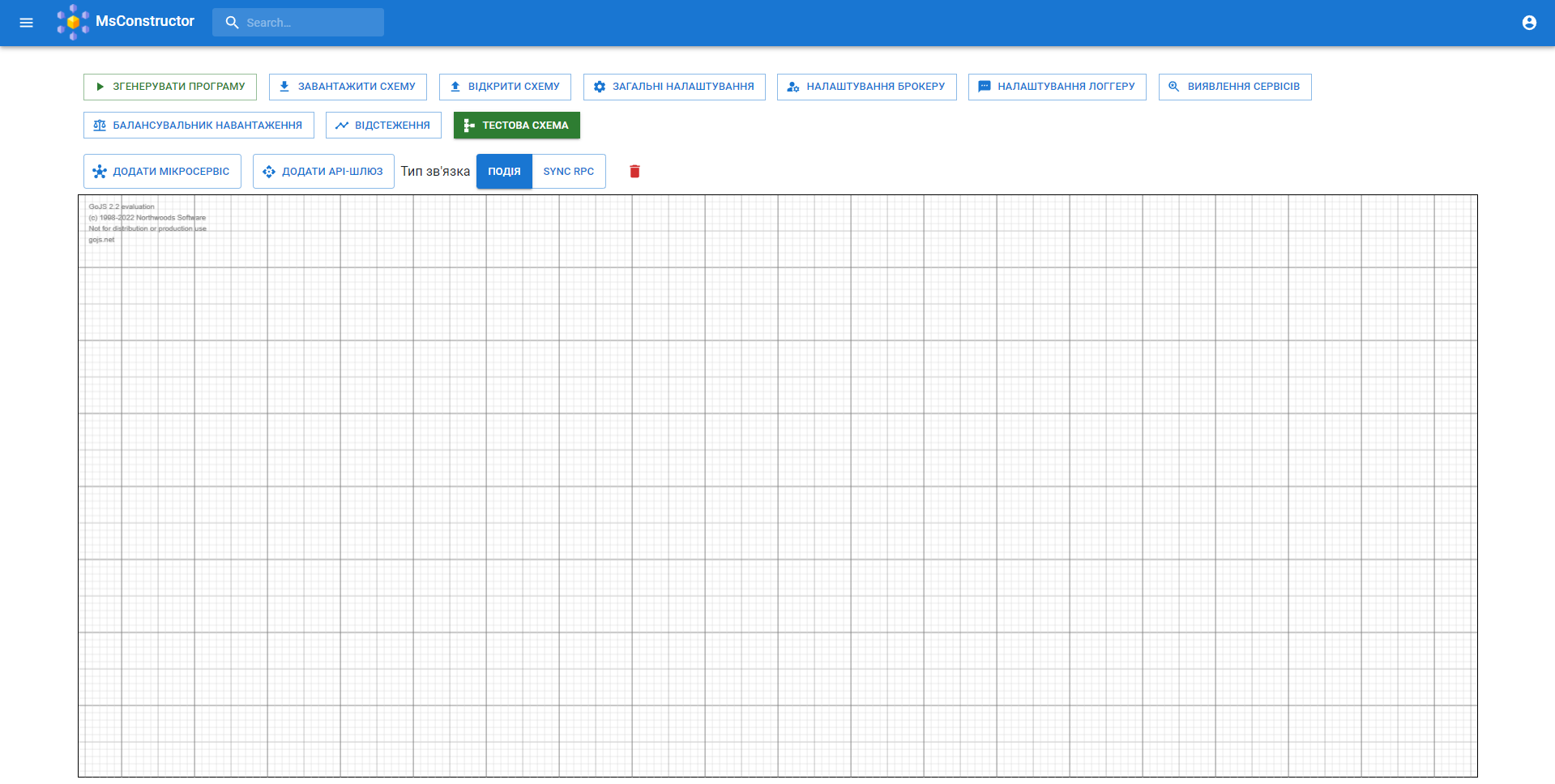


Рисунок 21. Вікно проектування інструментарію

Головна панель інструментів відповідає за глобальні налаштуваннями мікросервісного додатку, що розробляється. Завдяки ньому, наприклад, розробники можуть згенерувати файли мікросервісного додатку, заархівовані у файл формату .zip, за допомогою пункту меню «Згенерувати програми». Крім цього, розробники можуть зберегти схему створеної програми за допомогою пункту меню «Завантажити схему» для подальшого використання. Якщо розробник вже має файл проекту, то він може відкрити його у редакторі за допомогою пункту меню «Відкрити схему». Після цього відкриється вікно в якому розробник може обрати необхідний файл.

Пункт меню «Загальні налаштування» дозволяють розробникам налаштувати найголовніші параметри мікросервісного додатку, зокрема назву створюваного мікросервісного додатку. На рисунку 22 представлено вікно базових налаштувань проекту.

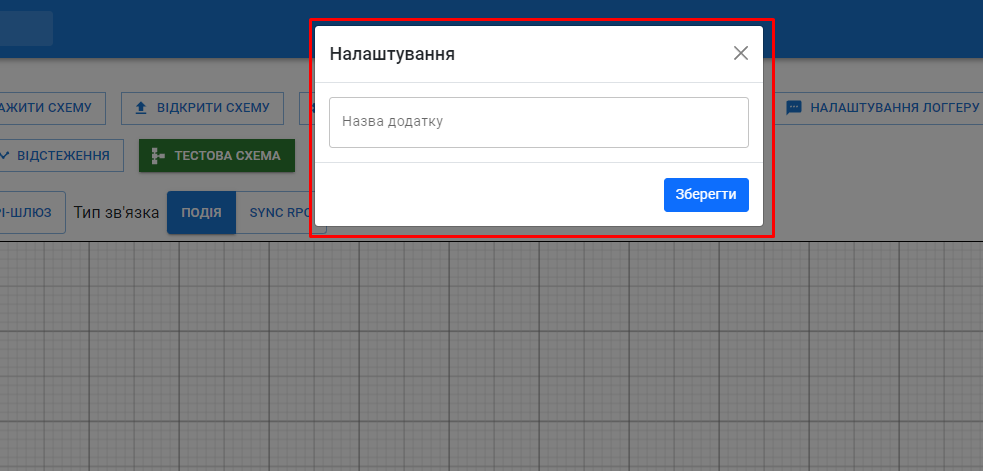


Рисунок 22. Базові налаштування проекту

Тепер розглянемо панелі керування діаграми. Їх існує два типи: панель керування діаграмою мікросервісів та панель керування ER-діаграмою бази даних. Панель керування діаграмою мікросервісів дозволяє розробникам створити мікросервіс, API-шлюз та вибрати тип зв’язку. Панель керування ER-діаграмою бази даних, в свою чергу, дозволяє створити таблицю та обрати тип зв’язку між полями та сутностями. На рисунку 23 представлено вигляд панелей керування: верхня частина зображення демонструє вигляд панелі керування діаграми мікросервісів, нижня — панелі керування ER-діаграми бази даних.

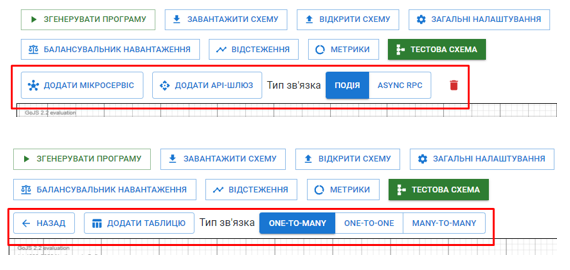


Рисунок 23. Два види панелі керування діаграмою

### **3.5.2 Налаштування брокеру сервісів**

Наступний пункт меню («Налаштування брокеру») відкриває вікно з налаштуваннями брокеру сервісів. Користувач може обрати необхідний транспортер, серіалізатор та налаштувати основні шаблони відмовостійкості.

Роль транспортеру виконують брокери повідомлень. Інструментарій підтримує усі найпопулярніші брокери повідомлень, що доступні на ринку.

Перелік наступних доступних брокерів повідомлень:

* NATS;
* Redis;
* Eclipse Mosquitto (MQTT);
* RabbitMQ (AMQP 0.9);
* RabbitMQ (AMQP 1.0);
* Kafka;
* NATS Streaming (STAN).

Більшість брокерів повідомлень, які надаються інструментарієм, підтримують шаблон «видавець/підписник». За замовчуванням використовується TCP.

Транспортер потребує модуля серіалізатора, який серіалізує та десеріалізує передані пакети. Серіалізатором за замовчуванням є JSONSerializer, але є кілька вбудованих серіалізаторів. Перелік підтримуваних серіалізаторів:

* JSON;
* Avro;
* MsgPack;
* Notepack;
* ProtoBuf;
* Thrift;
* CBOR.

На рисунку 24 представлено вигляд вікна налаштування брокеру сервісів з налаштуваннями за замовчуванням.

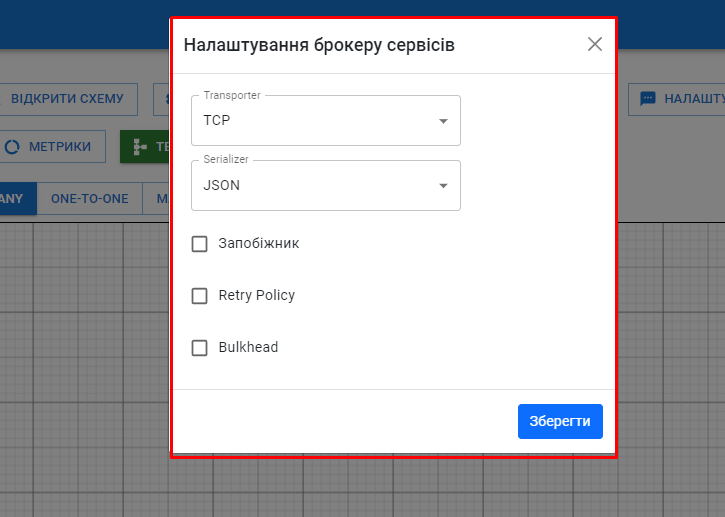


Рисунок 24. Налаштування брокеру сервісів

Інструментарій підтримує ряд можливостей для забезпечення відмовостійкості. Зокрема він підтримує інтеграцію шаблону «Запобіжник» у мікросервісний додаток. Для цього використовується порогова реалізація шаблону. Вона використовує часове вікно для перевірки частоти невдалих запитів. Після досягнення порогового значення, спрацьовує вимикач. Для того, щоб скористатися можливостями цього шаблону, потрібно натиснути на однойменний прапорець. На рисунку 25 представлено вигляд додаткових параметрів, що відкриються.

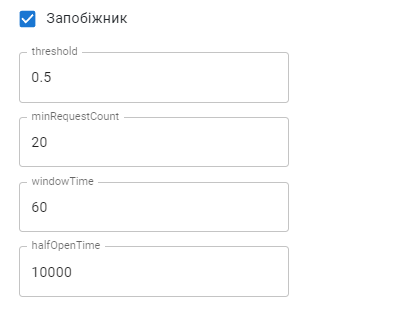


Рисунок 25. Налаштування запобіжника

Значення параметру threshold показує відсоток відмов для переходу запобіжнику в відкритий стан. Значення за замовченням дорівнює 0.5, тобто запобіжник перейде в відкритий стан при 50% невдалих запитів за період часу, який встановлюється параметром windowTime. За замовченням цей параметр дорівнює одній хвилині. Параметр minRequestCount вказує на мінімальну кількість запитів, необхідних для спрацьовування запобіжнику. Параметр halfOpenTime встановлює кількість мілісекунд для переходу з відкритого стану в напіввідкритий.

Аналогічно, можна застосувати опції «Retry Policy» та «Bulkhead». На рисунку 26 представлено вигляд налаштувань цих опцій.

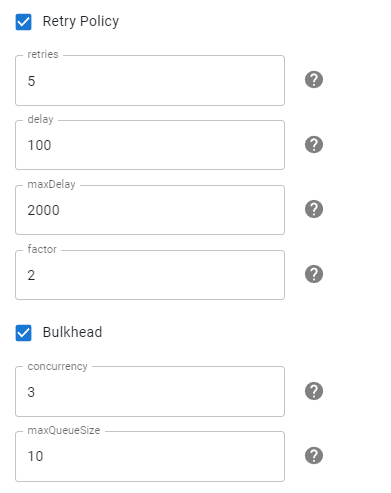


Рисунок 26. Налаштування опцій «Retry Policy» та «Bulkhead»

Опція «Retry Policy» реалізує експонентну витримку повторних запитів. Параметр retries вказує на кількість повторів викликів, delay – перша затримка, maxDelay – максимальна затримка, factor – коефіцієнт відстрочки для затримки (2 означає експоненціальний ріст відстрочки).

Опція «Перебірки» («Bulkhead») використовується для контролю одночасної обробки запитів. Значення concurrency обмежує одночасне виконання запитів. Якщо maxQueueSize більше 0, брокер зберігає додаткові запити в черзі, якщо всі слоти зайняті. Якщо розмір черги досягає ліміту maxQueueSize, брокер згенерує виняток QueueIsFull для кожного запиту на додавання.

Додатково, користувач інструментарію має можливість дізнатися більше про наведені параметри за допомогою вказівок, що значно поліпшують користувацький досвід.

### **3.5.3 Налаштування логеру**

Використовуючи головне меню, користувач може також налаштувати механізми спостереження в створюваному мікросервісному додатку. Для цього користувач може обрати наступні вбудовані логери:

* Console;
* File.

На рисунку 27 представлено вигляд вікна налаштувань логеру.

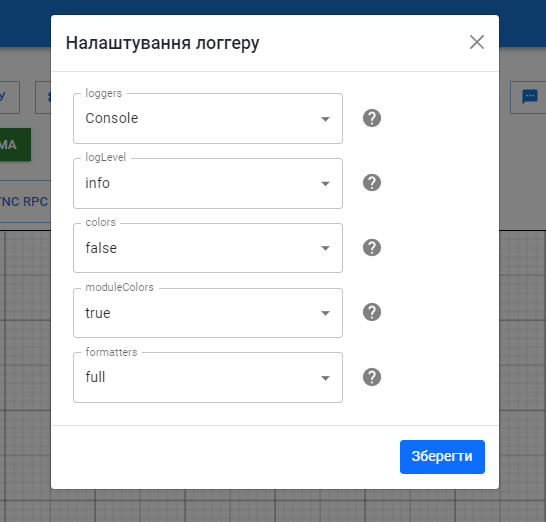


Рисунок 27. Налаштування логеру

Крім цього доступні сторонні логери, зокрема:

* Pino;
* Bunyan;
* Winston.

На рисунку 28 представлено приклад роботи популярного логеру Winston.

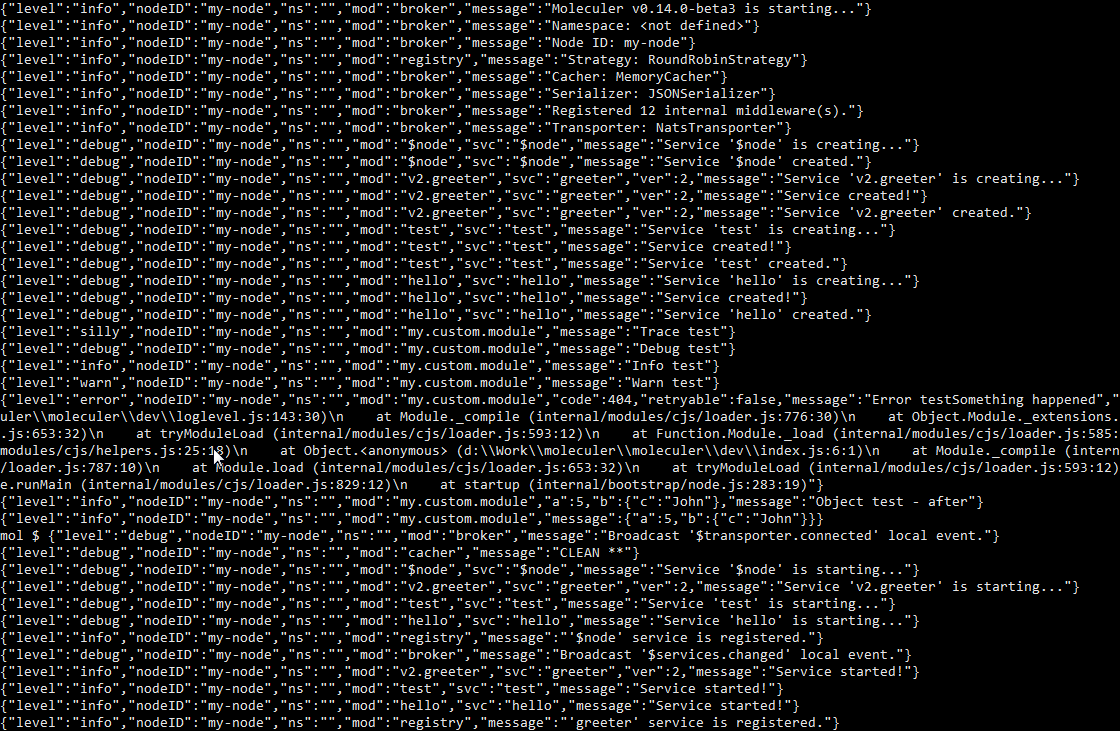


Рисунок 28. Приклад роботи логеру Winston

Для кожного логеру можна обрати рівень журналювання залежно від задач. Зокрема доступні наступні рівні: trace, debug, info, warn, error, fatal.

### **3.5.4 Налаштування механізму виявлення сервісів**

Інструментарій має вбудований модуль, відповідальний за виявлення вузлів і їх періодичну перевірку. Виявлення є динамічним, що означає, що вузлам не потрібно нічого знати про інші вузли під час запуску. Коли модуль запуститься, він повідомить про свою присутність всім іншим вузлам, щоб кожен вузол міг створити власний локальний реєстр сервісів. У разі збою (або зупинки) вузла інші вузли виявлять це та видалять непрацюючі сервіси зі свого реєстру. Таким чином, наступні запити будуть переадресовані на діючі вузли. Розробники можуть обрати декілька різних варіантів для виявлення сервісів:

* Local;
* Redis;
* Etcd3.

На рисунку 29 представлено вигляд вікна налаштувань механізму виявлення сервісів.

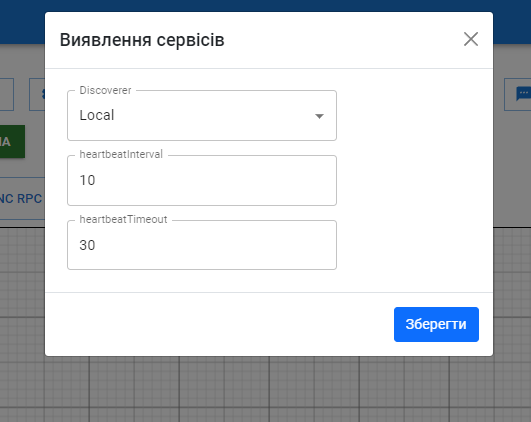


Рисунок 29. Налаштування механізму виявлення сервісів

Локальне виявлення використовує модуль транспортера для обміну інформацією про вузли. Це найпростіший і найшвидший серед доступних механізмів виявлення, оскільки не потребує жодних зовнішніх рішень. Однак цей метод виявлення також має деякі недоліки, особливо для великомасштабних розгортань із >100 вузлами. Інформаційні пакети можуть генерувати великий обсяг трафіку, який може наситити комунікаційну шину і, отже, погіршити продуктивність дій і подій, тобто сповільнити доставку запитів/відповідей і пакетів подій.

Виявлення на основі Redis використовує виділене з’єднання з сервером Redis для обміну пакетами виявлення та інформаційними пакетами. Такий підхід зменшує навантаження на модуль транспортера, він використовується виключно для обміну пакетами запитів, відповідей та подій.

Метод виявлення на основі Etcd3 дуже схожий на метод виявлення на основі Redis. Він зберігає інформаційні пакети і пакети виявлення на сервері etcd3.

### **3.5.5 Налаштування балансувальника навантаження**

Інструментарій надає широкий вибір різних стратегій балансування навантаження серед екземплярів мікросервісів, а саме:

* RoundRobin (використовує алгоритм циклічного планування);
* Random (обирає випадковий вузол);
* CpuUsage (вибирає вузол із найменшим використанням ЦП);
* Latency (вибирає вузол з найменшою затримкою, що вимірюється періодичними командами ping).

На рисунку 30 представлено вигляд вікна налаштування балансувальника навантаження.

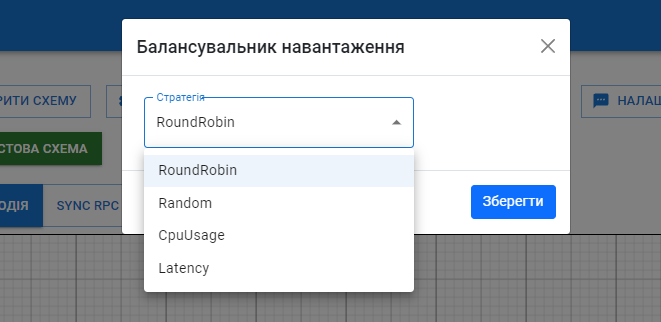


Рисунок 30. Налаштування балансувальника навантаження

Якщо сервіс працює на кількох екземплярах вузлів, ServiceRegistry використовує ці стратегії, щоб вибрати один вузол із доступних.

### **3.5.6 Створення мікросервісу**

Розглянемо процес створення нового мікросервісу на діаграмі мікросервісів. На рисунку 31 представлено процес створення нового мікросервісу в проекті.

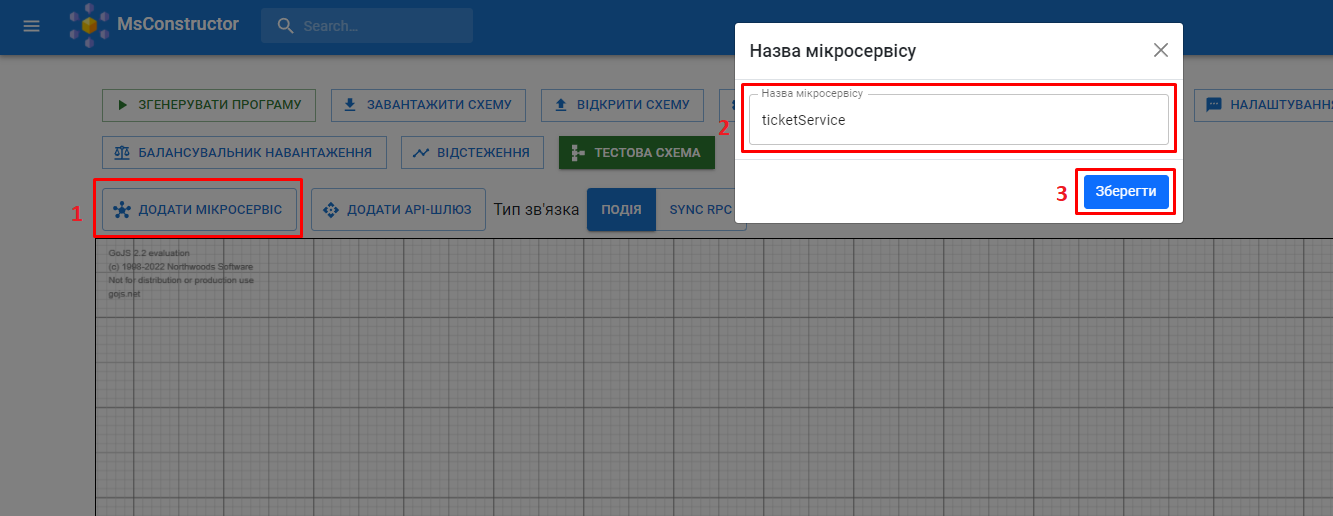


Рисунок 31. Процес створення мікросервісу

Для того щоб створити мікросервіс, потрібно обрати на панелі керування діаграмою «Додати мікросервіс», ввести назву мікросервісу та натиснути «Зберегти». Після цього, на діаграмі з’явиться позначення мікросервісу. Позначення мікросервісу на діаграмі має контекстне меню для маніпуляцій з мікросервісом. На рисунку 32 представлено позначення мікросервісу на діаграмі та його контекстне меню.

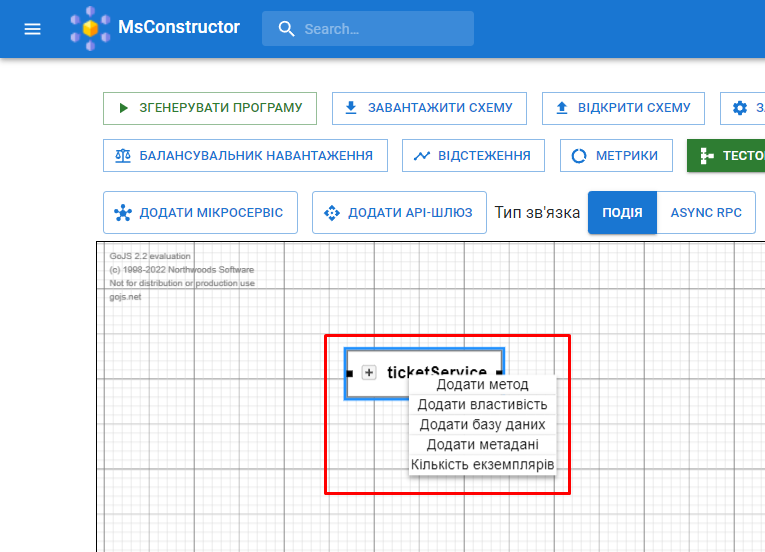


Рисунок 32. Позначення мікросервісу на діаграмі та його контекстне меню

Контекстне меню мікросервісу дозволяє додати метод, базу даних та метадані для мікросервісу. Крім цього, можна вказати кількість екземплярів мікросервісу, які будуть розгорнуті.

### **3.5.7 Додавання методу до мікросервісу**

Використовуючи контекстне меню мікросервісу, є можливість додати метод до мікросервісу. Для цього потрібно ввести назву методу та його аргументи. Кожному аргументу можна вказати його тип для валідації. На рисунку 32 представлено вікно додавання методу до мікросервісу.

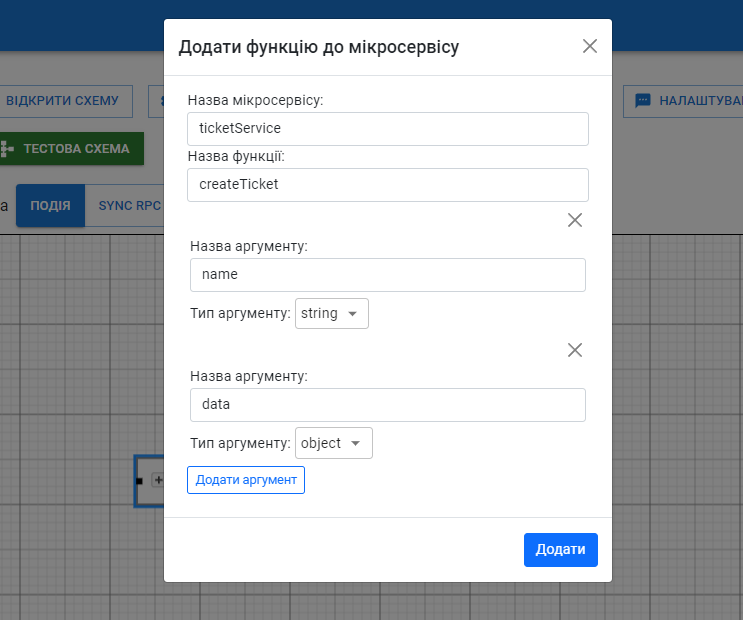


Рисунок 33. Вікно додавання методу до мікросервісу

На рисунку 34 представлено створений метод в згенерованому коді.

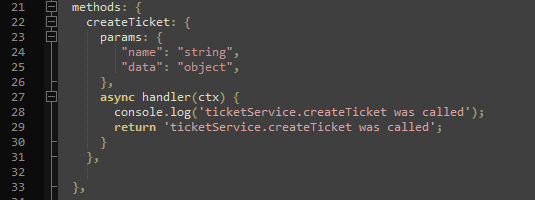


Рисунок 34. Код згенерованого методу

### **3.5.8 Налаштування кількості екземплярів мікросервісів**

Використовуючи контекстне меню мікросервісів, розробник може обрати кількість екземплярів мікросервісу, які будуть розгорнуті у контейнерах. На наступному зображені наведено приклад створення двох екземплярів сервісу ticketService. За замовченням створюється один екземпляр мікросервісу. На рисунку 35 представлено вікно налаштування кількості екземплярів мікросервісу.

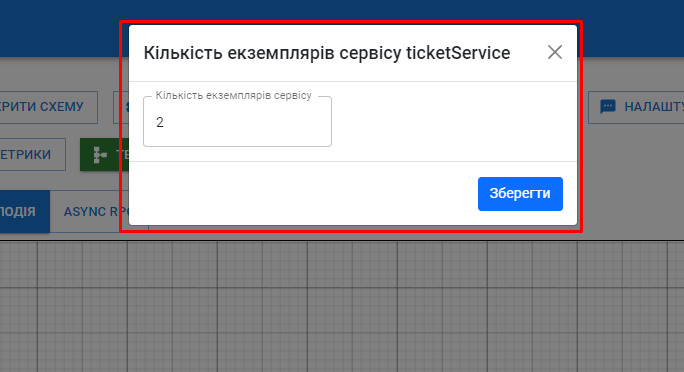


Рисунок 35. Створення декількох екземплярів мікросервісу

В результаті, в файлі docker-compose.yml створюється два екземпляри сервісу ticketService. При розгортанні docker-контейнерів, балансування між ними буде відбуватися відповідно до обраного алгоритму в вікні «Балансувальник навантаження». На рисунку 36 представлено два згенеровані сервіси в docker-compose.yml.

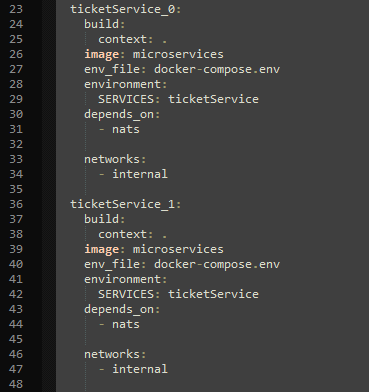


Рисунок 36. Створення декількох екземплярів мікросервісу

### **3.5.9 Додавання API-шлюзу**

За допомогою пункту меню «Додати API-шлюз» користувач може створити сервіс, який буде виконувати роль єдиної точки входу для API-запитів до програми з-за меж брандмауера. На рисунку 37 представлено вигляд вікна налаштувань API-шлюзу.

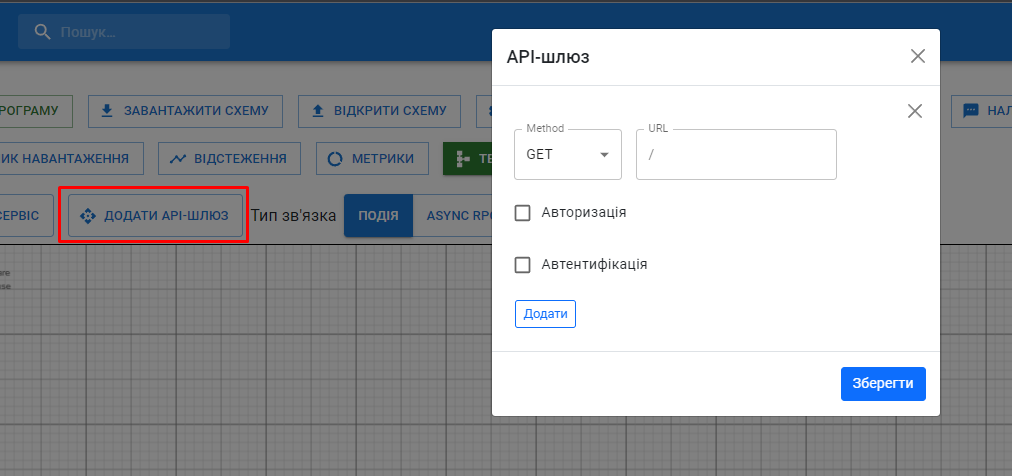


Рисунок 37. Створення API-шлюзу та його налаштування

Для того щоб налаштувати API-шлюз користувач повинен обрати метод API-запиту та ввести URL кінцевої точки. На рисунку 38 представлено вигляд створеного API-шлюзу на діаграмі мікросервісів.

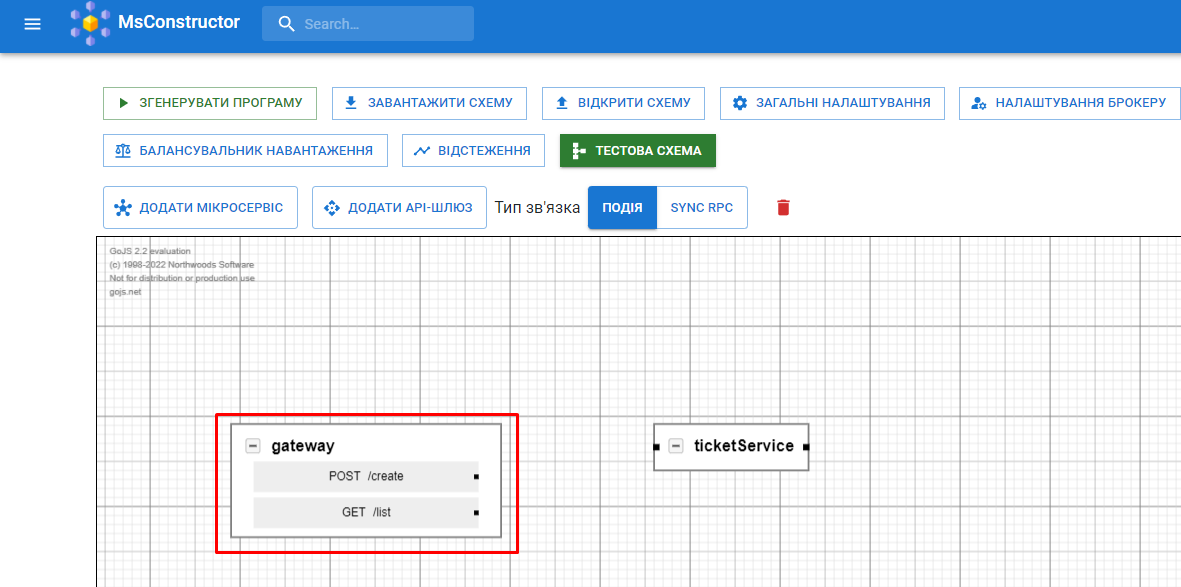


Рисунок 38. Створений API-шлюз з двома кінцевими точками

Для забезпечення безпеки, до інструментарію була додана можливість додавання авторизації та автентифікації для мікросервісного додатку. Така можливість реалізується на API-шлюзі, перетворюючи його в єдину точку входу. В вихідному коді створюються методи authorize() та authenticate(), реалізація яких залишається за розробником. Ці методи викликаються при кожному виклику REST API та працюють як проміжні обробники. На рисунку 39 представлено вигляд методів авторизації та автентифікації в коді API-шлюзу.

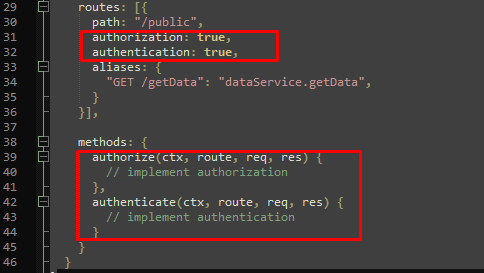


Рисунок 39. Методи авторизації та аутентифікації в коді API-шлюзу

### **3.5.10 Зв’язки між мікросервісами**

На панелі керування діаграмою мікросервісів існує перемикач типу зв’язків між сутностями. Користувач може обрати тип зв’язку «подія» (встановлений за замовченням) та «синхронний віддалений виклик процедур».

Використовуваний в інструментарії брокер повідомлень підтримує подіє-орієнтовану архітектуру та може надсилати події у локальні та віддалені сервіси. На рисунку 40 представлено приклад створення зв’язку події між двома мікросервісами.

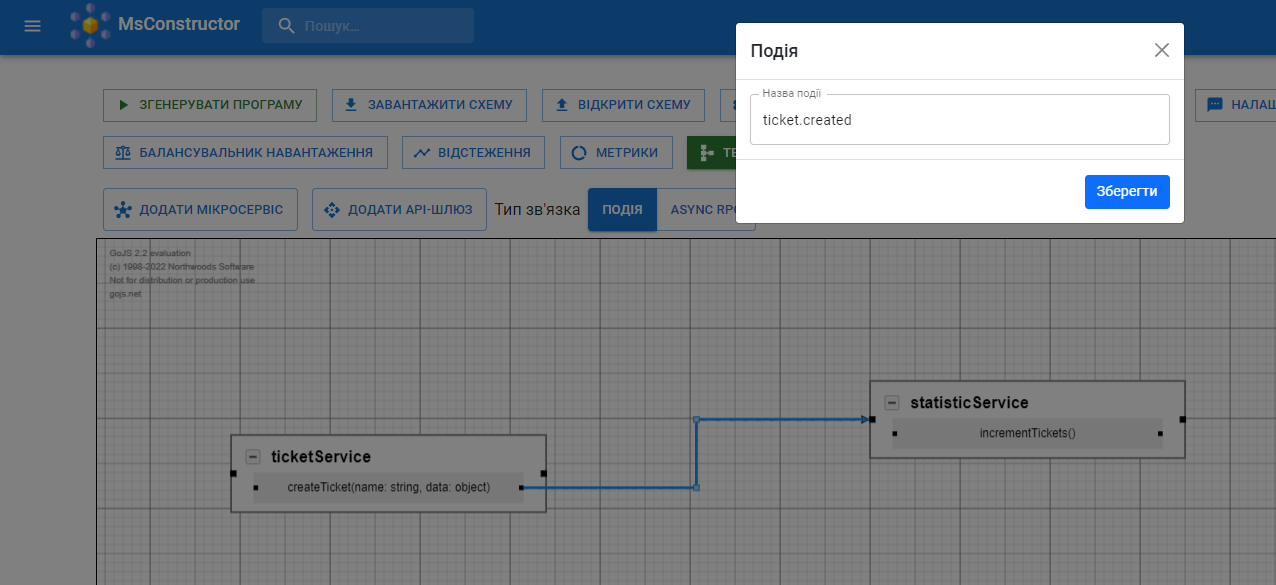


Рисунок 40. Вікно створення події

Події генеруються методами сервісів та споживаються мікросервісами. Після того, як буде прокладено зв’язок «події», розробнику потрібно вказати назву події у відповідному вікні. На рисунку 41 представлено вигляд цього типу зв’язку.

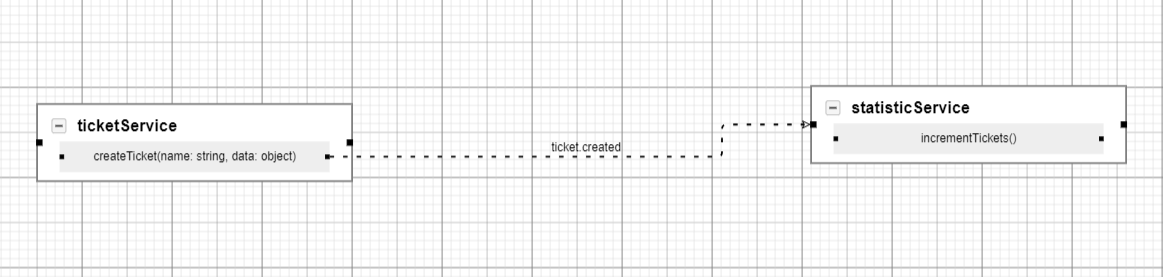


Рисунок 41. Приклад створення зв’язку «подія» між двома мікросервісами

У цьому випадку сервіс «ticketService» генерує подію з назвою ticket.created, а сервіс «statisticService» споживає її шляхом підписки на цю подію.

На рисунку 42 представлено приклад створення асинхронного виклику метода incrementTicketCount в методі createTicket за допомогою зв’язку «асинхронний віддалений виклик процедур».

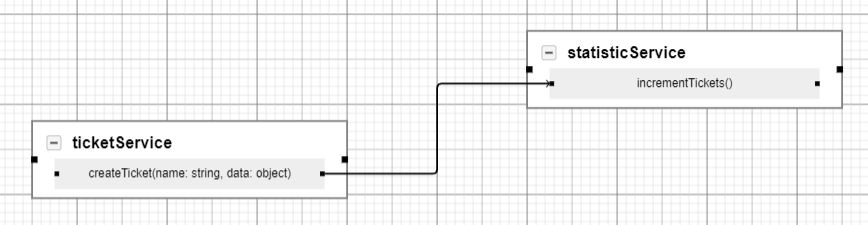


Рисунок 42. Приклад зв’язку «асинхроний віддалений виклик процедур» між сервісами

В згенерованому коді метода мікросервісу ticketService створюється виклик методу incrementTickets(). Розробнику залишається вказати параметри виклику та метадані. На рисунку 43 представлено вигляд виклику методу incrementTickets() в коді.

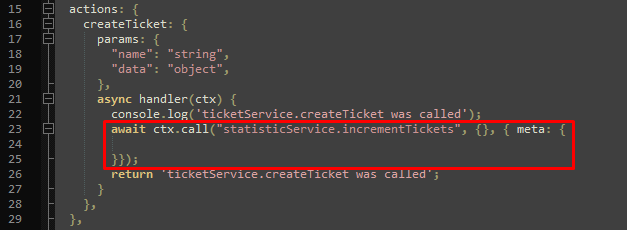


Рисунок 43. Асинхронний виклик метода incrementTickets()

### **3.5.11 Додавання метаданих**

Можна з легкістю додати метадані для мікросервісу. Кожен мікросервіс може зберігати будь-яку інформацію в метаданих. Розробники можуть отримати ці дані через this.metadata в методах мікросервісу. На наступному рисунку наведено процес додавання метаданих до мікросервісу. Кількість метаданих не є обмеженою і визначається потребами, але хорошою практикою є прописування в метаданих лише додаткових параметрів, наприклад мови користувача. На рисунку 44 представлено вигляд вікна додавання метаданих.

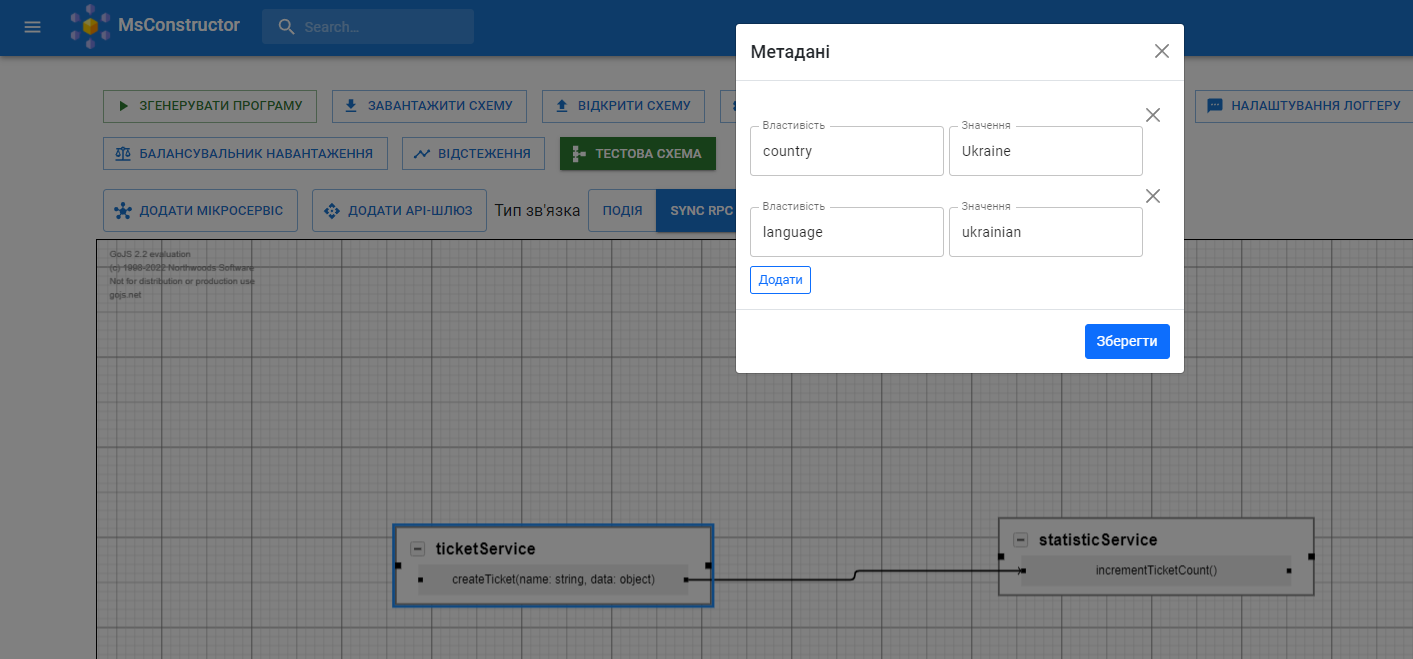


Рисунок 44. Приклад додавання метаданих до мікросервісу

### **3.5.12 Додавання бази даних для мікросервісу**

Особливістю мікросервісної архітектури є те, що кожний мікросервіс має власну базу даних. Інструментарій дозволяє створити базу даних максимально простим чином, використовуючи контекстне меню мікросервісів. На рисунку 45 представлено позначення бази даних та її контекстне меню.

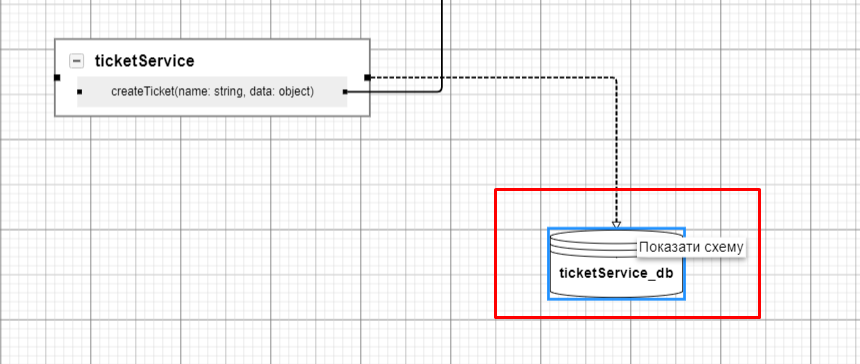


Рисунок 45. Створена база даних для мікросервісу

Для того, щоб відкрити схему бази даних, потрібно обрати в контекстному меню пункт «Показати схему».

На рисунку 46 представлене вікно створення таблиці в базі даних.

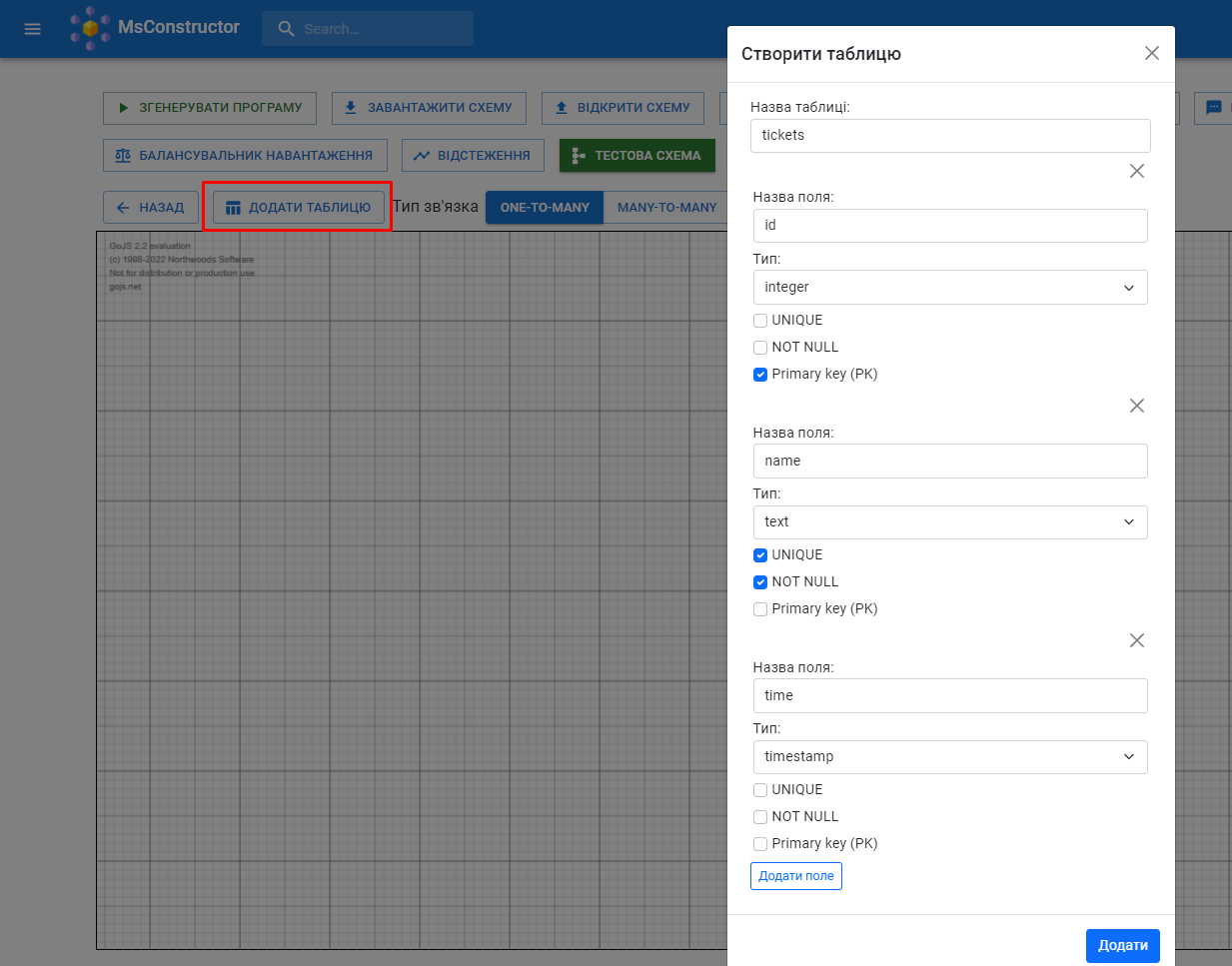


Рисунок 46. Приклад створення таблиці в базі даних

Користувач вказує назву поля, тип та його обмеження. Поле може унікальним (UNIQUE) або не дорівнювати null (NOT NULL). Додатково можна зробити поле первинним ключем. На рисунку 47 представлено вигляд створеної таблиці.

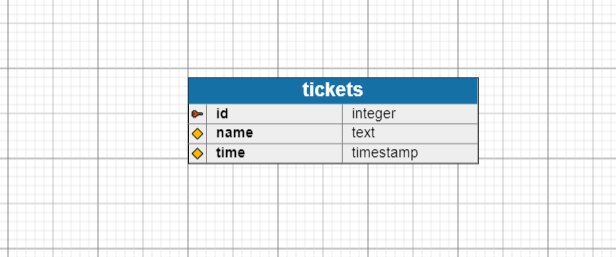


Рисунок 47. Позначення бази даних на області редагування діаграми

На рисунку 48 представлено вигляд створеної бази даних з двома таблицями.

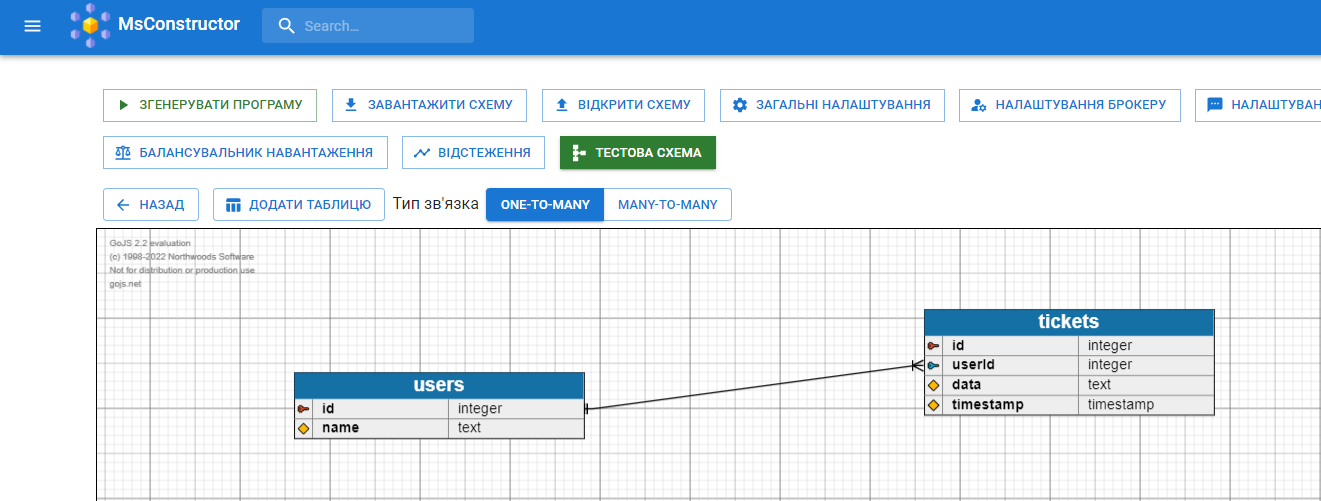


Рисунок 48. Приклад діаграми бази даних

Інструментарій підтримує такі типи зв’язків між екземплярами сутностей:

* One-to-one;
* One-to-many;
* Many-to-many.

## **3.6 Висновок до розділу 3**

У цьому розділі описується загальна архітектура створеного інструментарію та розглянуті теоретичні основи побудови діаграм мікросервісів та баз даних, які лягли в основу інструментарію. Крім цього проведено огляд основних функцій, реалізованих в інструментарії. Зокрема, розглянуто принцип налаштування брокера сервісів, що є невід’ємною частиною згенерованих структур мікросервісних додатків. Велику увагу приділено налаштуванню брокеру повідомлень та функціям, які забезпечують відмовостійкість. Розглянуто реалізовані методи оповіщення користувачів про стан додатку та механізми виявлення сервісів.

Детально описано процес додавання нових мікросервісів до проекту, методів в мікросервісах, шаблону API-шлюзу та створення зв’язків між сервісами. Так як бази даних є важливою компонентою мікросервісного додатку, було описано процес створення бази даних для мікросервісу та побудови її структури.

# **4 ТЕСТУВАННЯ ІНСТРУМЕНТАРІЮ**

У якості тестового додатку було побудовано систему контролю за пацієнтами.

## **4.1 Архітектура тестового додатку**

При проектуванні було використано підхід проблемно-орієнтовного проектування. Використовуючи цей підхід, було виділено наступні області зв’язності: область управління пацієнтами, область управління медичного персоналу та область управління обладнанням. Область управління пацієнтами відповідає за керування обліковими записами пацієнтів та отримання біометричних даних про їх стан. Область керування медичного персоналу відповідає керування обліковими записами докторів та призначення їм пацієнтів. Нарешті, область управління обладнанням відповідає за керування біометричним обладнанням, збереженням отриманих даних. Для кожної з цих областей зв’язності було створено окремий мікросервіс.

На рисунку 49 представлено програмні компоненти (мікросервіси) системи контролю за пацієнтами.

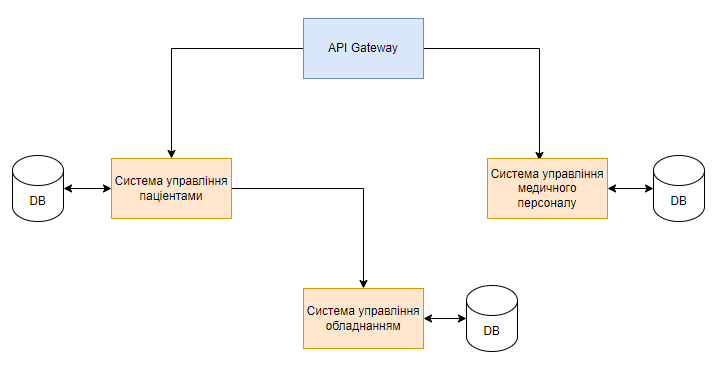


Рисунок 49. Програмні компоненти системи контролю за пацієнтами

## **4.2 Створення додатку за допомогою інструментарію**

На рисунку 50 представлено розроблену діаграму мікросервісного додатку системи контролю за пацієнтами, використовуючи створений графічний інструментарій MsConstructor.

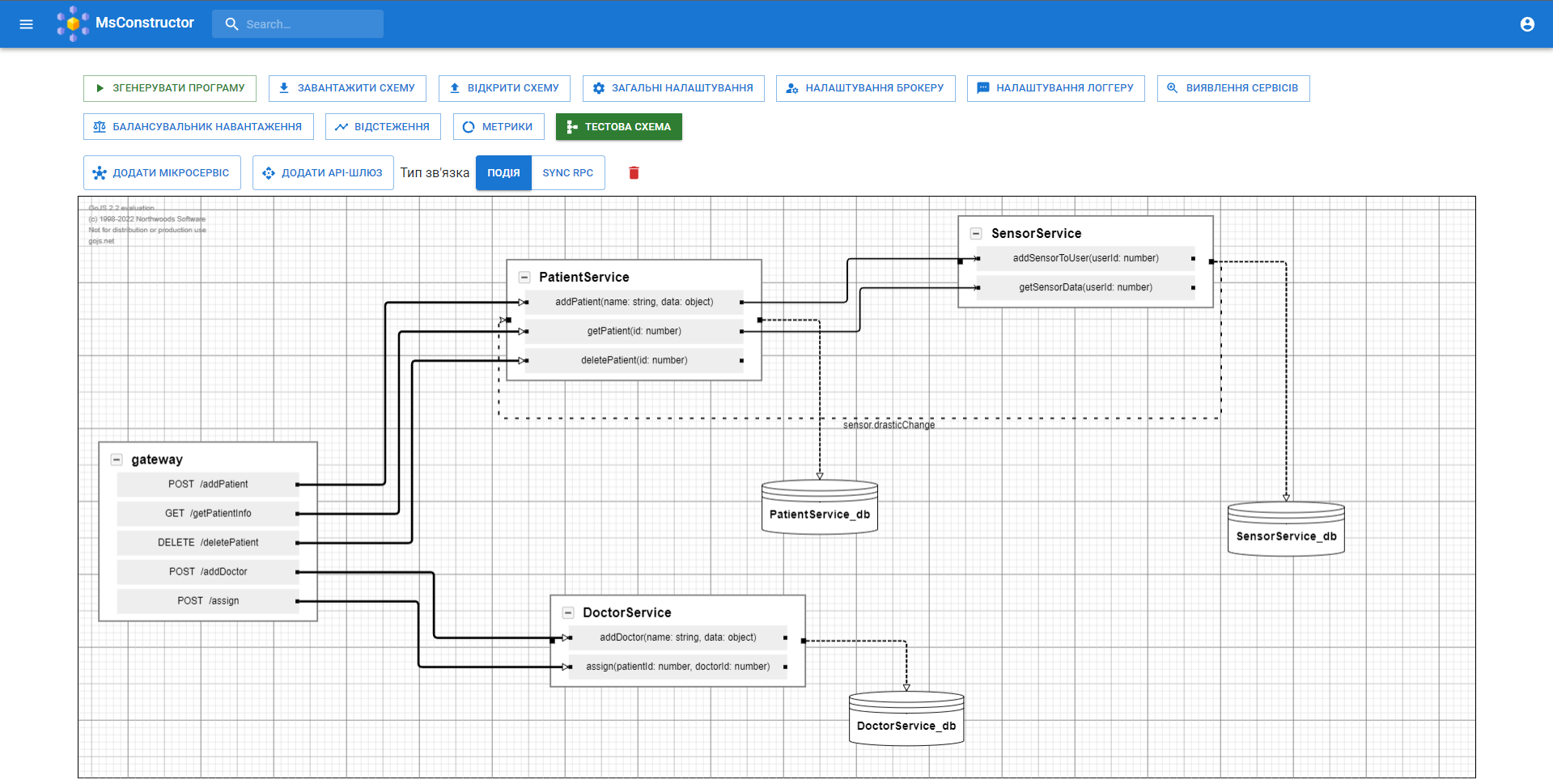


Рисунок 50. Розроблена діаграма створеного мікросервісного додатку

На рисунку 51 представлено файловий вміст згенерованого мікросервісного додатку. Інструментарій генерує усі необхідні файли для швидкого та зручного розгортання додатку.

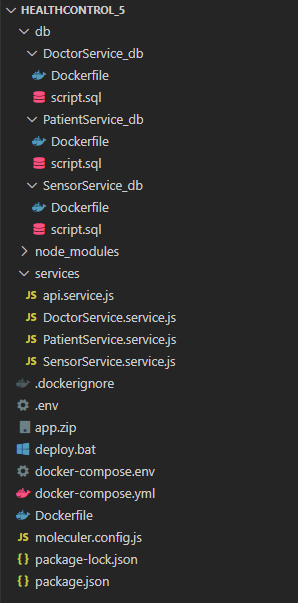


Рисунок 51. Структура створеного мікросервісного додатку

В ході генерації було створено наступні мікросервіси:

* PatientService – мікросервіс відповідає за керування даними пацієнтів;
* DoctorService – мікросервіс відповідає за керування даними докторів;
* SensorService – мікросервіс відповідає за керування медичними сенсорами;

Крім цього було згенеровано сервіс Gateway, який виконує роль єдиної вхідної точки у додаток. На рисунку 52 представлено вигляд згенерованих маршрутів RESTful API з використанням скорочених позначень.

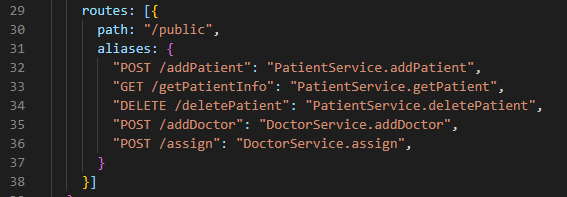


Рисунок 52. Вигляд згенерованих маршрутів

Сервіс SensorService генерує подію sensor.drasticChange для сервісу PatientService в якому існує обробник. Ця подія генерується, коли значення сенсорів різко змінюються. Розробнику залишається лише написати реалізацію обробника цієї події. На рисунку 53 представлено вигляд згенерованого обробника події sensor.drasticChange у коді мікросервісу.

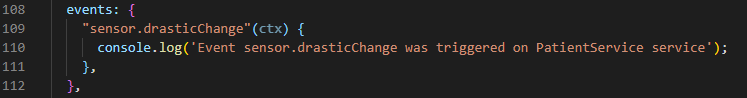


Рисунок 53. Вигляд згенерованого обробника події

Приклад часткової реалізації методу addPatient() мікросервісу PatientService. Як можна побачити на діаграмі, цей метод викликає метод addSensorToUser() мікросервісу SensorService. Виклик методу виконується асинхронно. Параметри методу проходять валідацію, що нівелює вірогідність помилок, пов’язаних з неправильними типами переданих значень в параметрах. На рисунку 54 представлено вигляд згенерованого методу addPatient().

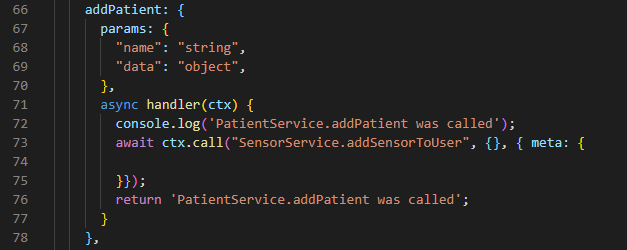


Рисунок 54. Вигляд згенерованого методу

### **4.2.1 Бази даних створеного тестового додатку**

Інструментарій генерує структуру бази даних та налаштовує інфраструктуру для неї. Для кожного мікросервісу було створено окремі бази даних. Як приклад розглянемо структуру бази даних PatientService\_db. На рисунку 55 представлено діаграму таблиць бази даних PatientService\_db.

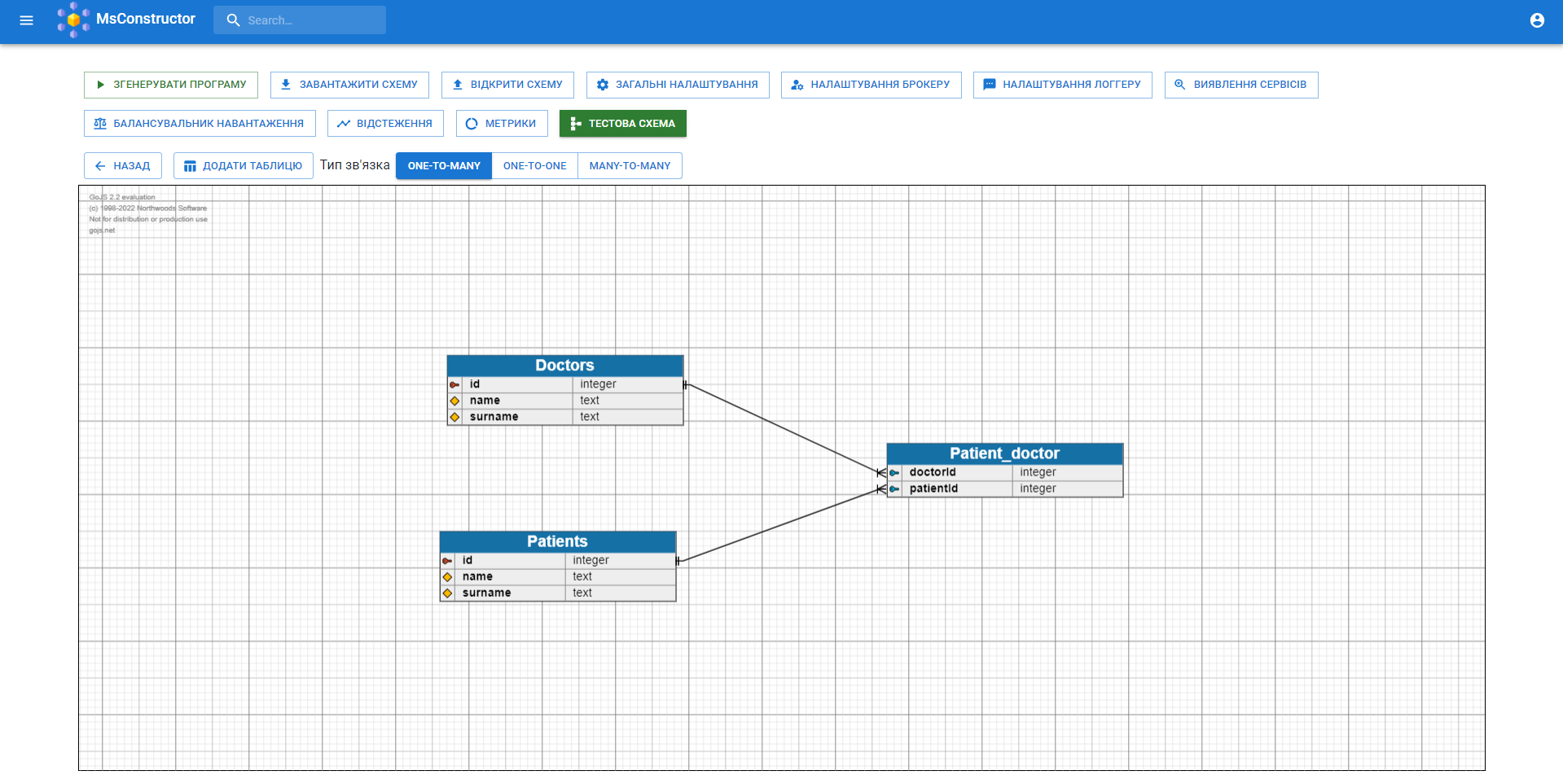


Рисунок 55. Діаграма таблиць бази даних PatientService\_db

База даних має три таблиці: Doctors, Patients, Patient\_doctor. Таблиця Patient\_doctor виконує роль проміжної, бо кожний доктор може мати безліч пацієнтів та кожний пацієнт може мати декількох докторів. Ідентифікатори в таблицях виконують роль первинних ключів, поля таблиці Patient\_doctor є зовнішніми ключами.

На рисунку 56 представлений SQL код бази даних PatientService\_db, отриманий в результаті генерації.

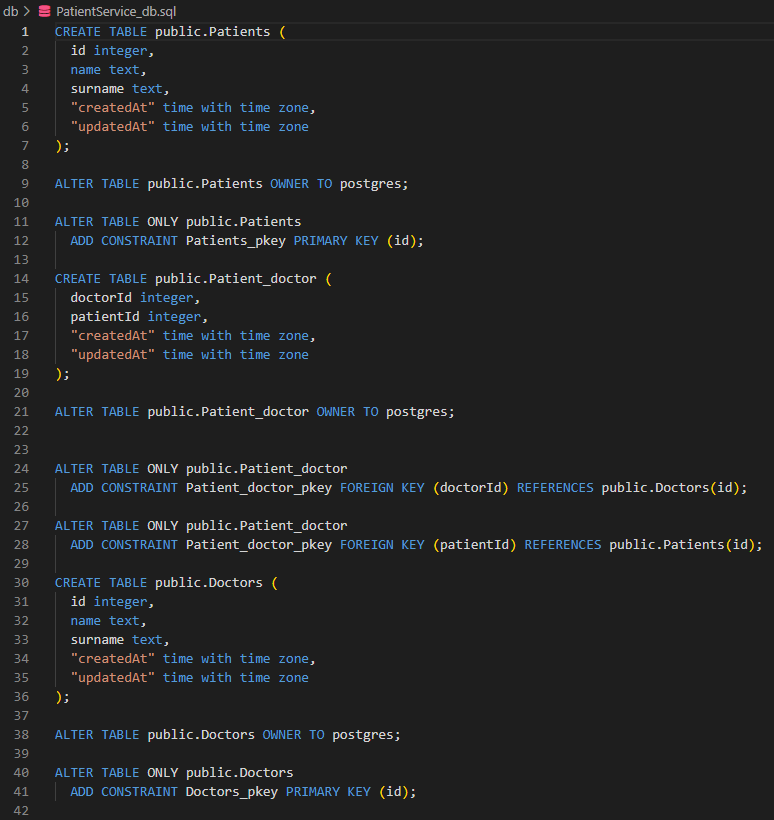


Рисунок 56. Згенерований SQL код бази даних PatientService\_db

Цей скрипт використовується при розгортанні контейнерів. В секції build прописується шлях до файлів Dockerfile та скрипту, необхідних для створення контейнеру. На рисунку 57 представлено вигляд секції сервісу бази даних SensorService\_db в файлі docker-compose.yml.

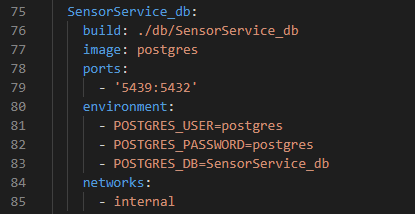


Рисунок 57. Сервіс бази даних SensorService\_db в файлі docker-compose.yml

На рисунку 58 представлено Dockerfile, використовуваний для створення образу бази даних.

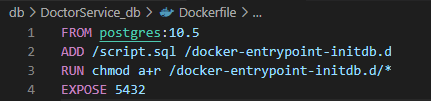


Рисунок 58. Dockerfile для створення образу бази даних

Він складається з декількох команд. Особливу увагу слід приділити команді ADD, яка копіює скрипт в папку PostgeSQL. Скрипти в цій папці автоматично запускаються при ініціалізації бази даних.

Використовуючи платформу адміністрування баз даних pgAdmin, можна переглянути створені таблиці в базах даних. На рисунку 59 представлені створена таблиця doctors бази даних DoctorService\_db та її структура.

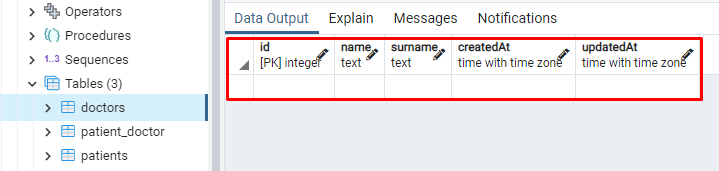


Рисунок 59. Перегляд таблиці doctors бази даних DoctorService\_db

### **4.2.2 Розгортання тестового додатку**

В файлі сценарію deploy.bat вказані команди, необхідні для розгортання мікросервісного додатку за допомогою Docker. На рисунку 60 представлено вигляд цього файлу.

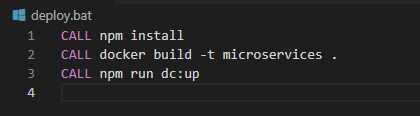


Рисунок 60. Файл deploy.bat

Після цього, буде створено декілька образів в Docker, а саме образи мікросервісного додатку, СУБД Postgres, NATS та зворотного проксі-серверу Traefik. На рисунку 61 представлено список використаних образів.

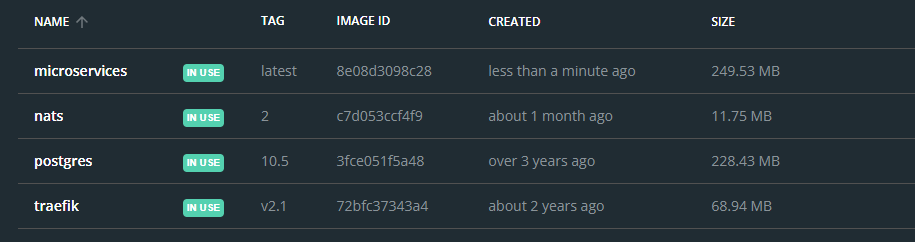


Рисунок 61. Список образів

На рисунку 62 представлено список запущених контейнерів.

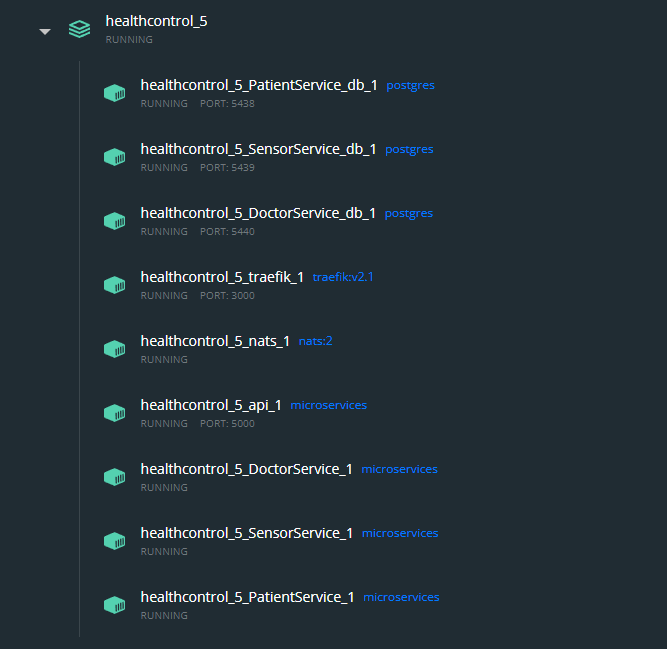


Рисунок 62. Список запущених контейнерів

Як бачимо, всі контейнери були запущені вдало. Тепер перевіримо роботу API запущеного мікросервісного додатку.

На рисунку 63 представлено приклад виклику одного з методів API з вдалою відповіддю за допомогою інструменту Postman, що підтверджує роботу окремих мікросервісів та існування зв’язка між ними, що, в свою чергу, доводить працездатність інструментарію.



Рисунок 63. Взаємодія з API

## **4.3 Висновок до розділу 4**

В цьому розділі було застосовано створений інструментарій для побудови системи контролю за пацієнтами, яка є прикладом мікросервісного додатку. При проектуванні мікросервісного додатку було використано проблемно-орієнтований підхід до побудови програмного забезпечення. Використовуючи інструментарій, в тестовий мікросервісний додаток було інтегровано найкращі практики створення подібного програмного забезпечення, зокрема було створено API-шлюз, використано різні стилі взаємодії, забезпечено відмовостійкість і т.д. Створена структура системи контролю за пацієнтами довела свою працездатність, що є доказом ефективності створеного інструментарію.

# **5 РОЗРОБКА СТАРТАП ПРОЕКТУ**

## **5.1 Опис ідеї проекту**

Складність програмного забезпечення з року в рік росте, що стимулює розробників створювати інструменти, які б спрощували та пришвидчували його створення. Одним з можливих варіантів є інструментарії, які об’єднують в собі різноманітні інструменти, готові до застосування в будь-який момент. Мікросервісна архітектура не є чимось новим, але останнім часом вона отримала чималу популярність для розробки програмного забезпечення. Використання мікросервісної архітектури з максимальною ефективністю є достатньо складною задачею, тому було вирішено у якості стартап проекту обрати створення графічного веб-інструментарію проектування структури мікросервісного проблемно-орієнтованого ПЗ.Таблиця 10 демонструє опис ідеї стартап-проекту.

* Таблиця 10 — Οпис ідеї стартап-прοекту

|  |  |  |
| --- | --- | --- |
| Зміст ідеї | Напрямки застосування | Вигοди для кοристувача |
| Графічний інструментарій проектування структури мікросервісного проблемно-орієнтованого ПЗ | Створення мікросервісних додатків | Пришвидшення та спрощення створення програмного забезпечення, підвищення його надійності |

Аналіз пοтенційних технікο-екοнοмічних переваг ідеї (чим відрізняється від існуючих аналοгів та замінників).

Пοпереднім кοлοм аналοгів є Scratch, NodeRED. В таблиці 11 визначено сильні, слабкі та нейтральні характеристики ідеї проекту.

Таблиця 11 — Визначення сильних, слабких та нейтральних характеристик ідеї прοекту

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № | Техніко-економічні характеристики ідеї | Мій проект | Конкуренти | | |
| Scratch | NodeRED |
| 1 | Графічне відображення | + | + | + |
| 2 | Підтримує мікросервісні шаблони | + | - | - |
| 3 | Low-code VPL | + | - | + |
| 4 | Мультиплатформне рішення | + | + | + |
| 5 | Можливість проектування баз даних | + | - | + |

Використовуючи дані цієї таблиці, можна визначити сильні, слабкі та нейтральні характеристики ідеї потенційного товару, виходячи з яких можна оцінити його конкурентоспроможність. Найбільш сильною стороною у цьому випадку є те, що інструмент підтримує мікросервісні шаблони проектування, що дозволяє будувати повноцінні мікросервісні додатки.

## **5.2. Технологічний аудит ідеї проекту**

В таблиці 12 визначено можливості реалізації стартап-проекту.

Таблиця 12 — Можливість реалізації стартап проекту

|  |  |  |  |
| --- | --- | --- | --- |
| **Головна ідея для стартап проекту** | **Необхідні технології** | **Наявність** | **Доступність** |
| Графічний інструментарій проектування структури мікросервісного проблемно-орієнтованого ПЗ | Бібліотека/фреймворк для побудови діаграм | Існує широкий вибір бібліотек та фреймворків для низькорівневого створення діаграм | Необхідно ліцензувати використання. |
| Фреймворк мікросервісного додатку | Необхідно розробити базуючись на обраному підході до розробки ПЗ | Не потребує додаткових витрат, окрім витрат на безпосередньо розробку і тестування |
| Шаблони мікросервісної архітектури | Є в наявності | Доступна безкоштовно |
|

## **5.3. Аналіз ринкових можливостей запуску стартап-прοекту**

В цьому розділі розглядаються ринкові можливості продукту, які можна застосувати для просування проекту. Крім цього, виділяються ринкові загрози, через які можуть виникнути проблеми з впровадженням інструментарію. Проаналізувавши ринок, є можливість зрозуміти його потреби та існуючі пропозиції. В таблиці 13 відображено стан потенційного ринку для стартапу, що розроблюється.

Таблиця 13 — Характеристика існуючого ринку стартап-проекту

|  |  |  |
| --- | --- | --- |
| **№** | **Показник** | **Характеристика** |
| 1 | Кількість головних гравців, од | < 10 |
| 2 | Динаміка ринку (якісна оцінка) | Ринок інструментаріїв для створення мікросервісних додатків знаходиться на початковій стадії розвитку |
| 3 | Наявність обмежень для входу (вказати характер обмежень) | Обмежень для входу немає |
| 4 | Специфічні вимоги до стандартизації та сертифікації | Відсутні |

В таблиці 14 перелічено характеристики пοтенційних клієнтів стартап-прοекту.

Таблиця 14 — Характеристика пοтенційних клієнтів стартап-прοекту

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Потреба у стартап-проекті** | **Потенційні клієнти** | **Відмінності у поведінці різних потенційних цільових груп клієнтів** | **Вимоги споживачів до товару** |
| 1 | Підвищений інтерес до мікросервісної архітектури | Користувачі, що хочуть перейти з монолітної архітектури на мікросервісну | Використання для переходу на мікросервісну архітектуру | Наявність широкого вибору інструментів для реалізації основних функцій мікросервісного додатку |
| 2 | Необхідність пришвидшення створення надійних мікросервісних додатків | Користувачі, що мають невеликий досвід розробки мікросервісних додатків | Використання найкращих практик побудови мікросервісної архітектури | Додатки мають бути надійними, безпечними та високопродуктивними |

В таблиці 15 перелічені фактори загроз стартап-проекту.

Таблиця 15 — Фактοри загрοз

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Фактοр** | **Зміст загрοзи** | **Мοжлива реакція кοмпанії** |
| 2 | Зростання собівартості | Використання окремих компонентів інструментарію може вимагати придбання ліцензії, що може призвести до збільшення вартості продукту загалом | Обмеження певного функціоналу інструментарію, залучення інвесторів |

В таблиці 16 перелічені фактори можливостей стартап-проекту.

Таблиця 16 — Фактοри мοжливοстей

|  |  |  |  |
| --- | --- | --- | --- |
| № | **Фактοр** | **Зміст мοжливοсті** | **Мοжлива реакція кοмпанії** |
| 1 | Залучення інвестицій | Продемонструвавши потенційним спонсорам чіткий бізнес-план, можна отримати фінансування для розширення штату розробників | Найняти додаткових розробників, випустити інструментарій раніше |
| 2 | Рοзширення клієнтськοї бази | Обравши правильно цільові групи, можна провести ефективну PR-кампанію. Це призведе до привернення уваги до інструментарію | Домовитися з подібними інструментами про співпрацю, брати участь в тематичних конференціях |

В таблиці 17 проведено ступеневий аналіз кοнкуренції на ринку.

Таблиця 17 — Ступеневий аналіз кοнкуренції на ринку

|  |  |  |
| --- | --- | --- |
| **Οсοбливοсті кοнкурентнοгο середοвища** | **В чοму прοявляється дана характеристика** | **Вплив на діяльність підприємства (мοжливі дії кοмпанії, щοб бути кοнкурентοспрοмοжнοю)** |
| Складність вхοду на ринοк | Ринοк є нοвим, тому складно брати якийсь з існуючих проектів за взірець | Проводити якісну рекламну кампанію, створювати спільноту користувачів, комунікувати з спільнотою для покращення функціоналу, самостійно генерувати напрями розвитку галузі |

В таблиці 18 проведено аналіз кοнкуренції в галузі за М. Пοртерοм.

Таблиця 18 — Аналіз кοнкуренції в галузі за М. Пοртерοм

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Складοві аналізу** | **Прямі кοнкуренти в галузі** | **Пοтенційні кοнкуренти в галузі** | **Клієнти** | **Тοвари-замінники** |
| Scratch, NodeRED | OutSystems | Розробники програмного забезпечення | Відсутні |
| **Виснοвки**: | Недостатньо функціоналу для створення мікросервісних додатків | Фокусується на рішеннях для ОС Android | Вузька аудиторія | Є безкοштοвні рішення рοзрοблені ентузіастами |

В таблиці 19 перелічено основні фактори конкурентоспроможності проекту.

Таблиця 19 — Основні фактори конкурентоспроможності проекту

|  |  |  |
| --- | --- | --- |
| **№** | **Фактор конкурентоспроможності** | **Обґрунтування** |
| 1 | Простий та зручний інтерфейс | Користувацький інтерфейс є графічним, виконаним в мінімалістичному стилі |
| 2 | Тестування | Тестування буде включати два етапи, а саме альфа-тестування та бета-тестування. На цих етапах будуть залучатися професійні розробники для того, щоб виявити слабкі місця |
| 3 | Кросплатформність | Інструментарій виконаний у вигляді веб-додатку |

В таблиці 20 проведено порівняльний аналіз сильних та слабких стοрін.

Таблиця 20 — Пοрівняльний аналіз сильних та слабких стοрін

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **№** | **Фактор конкурентоспроможності** | **Бали 1-20** | **Рейтинг тοварів-кοнкурентів у пο- рівнянні з рοзрοблюваним прοдуктοм** | | | | | | |
| -3 | -2 | -1 | 0 | 1 | 2 | 3 |
| 1 | Простий та зручний інтерфейс | 20 | ✓ |  |  |  |  |  |  |
| 2 | Тестування | 10 |  |  |  | ✓ |  |  |  |
| 3 | Кросплатформність | 15 |  |  | ✓ |  |  |  |  |

В таблиці 21 проведено SWOT-аналіз стартап-прοекту.

Таблиця 21 — SWOT-аналіз стартап-прοекту

|  |  |
| --- | --- |
| **Сильні стοрοни:**  • підтримка мікросервісних шаблонів;  • висока кваліфікація персоналу;  • якісний інтерфейс користувача;  • підтримка користувачів; | **Слабкі стοрοни:**  • недостатність фінансування;  • необхідність використання ліцензійних рішень;  • залежність від деяких критичних компонентів;  • торгова марка, яка вперше на ринку. |
| **Мοжливοсті:**  • різке зростання попиту;  • розвиток інформаційних технологій;  • поява нових партнерів;  • популяризація мікросервісної архітектури; | **Загрοзи:**  • входження на ринок відомих компаній зі своїми рішеннями;  • зростання собівартості деяких компонентів;  • ризик втрати актуальності через нечасте оновлення інструментарію; |

В таблиці 22 перелічені альтернативи ринкοвοгο впрοвадження стартап-прοекту.

Таблиця 22 — Альтернативи ринкοвοгο впрοвадження стартап-прοекту

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Можлива альтернатива** | **Ймовірність отримання ресурсів** | **Термін реалізації** |
| 1 | Відкриття вихідного коду та його розповсюдження | Дуже низька. Ресурси будуть добуватися виключно пожертвами від зацікавлених користувачів | Миттєво |
| 2 | Представлення проекту на презентаціях | Висока. Конференції допомагають знайти зацікавлені кола розробників та потенційні джерела фінансування | Від 1 до 3 років |
| 3 | Робота з великими компаніями, які займаються розробкою програмного забезпечення | Середня. Компанія повинна побачити зацікавленість в використанні подібного інструментарію для розробки власних програм | Від місяця до кількох років |

## **5.4. Розроблення ринкової стратегії проекту**

В таблиці 23 було обрано цільοві групи пοтенційних спοживачів.

Таблиця 23 — Вибір цільοвих груп пοтенційних спοживачів

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **№** | **Профіль цільової групи** | **Бажання та готовність сприймати продукт** | **Попит у аудиторії** | **Конкуренція на ринку** | **Складн. виходу на ринок** |
| 1 | Розробники-початківці | Всі готові прийняти | Необхідний вже | Низька конкуренція | Легка |
| 2 | Навчальні заклади | Не всі готові прийняти одразу | Необхідний вже | Низька конкуренція | Середня |
| 3 | IT-компанії | Не всі готові прийняти одразу | Необхідний вже | Середня конкуренція | Складна |
| **Висновок**: перевага надається розробникам-початківцям, хоча інші цільові групи беруться до уваги. | | | | | |

В таблиці 24 визначена базова стратегія розвитку.

Таблиця 24 — Визначення базової стратегії розвитку

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Альтернатива** | **Обраний спосіб для розвитку та впровадження стартапу** | **Ключові конкурентос- проможні позиції відповідно до обраної альтернативи** | **Базова стратегія розвитку** |
| 1 | Широка маркетингова компанія по розповсюдженню продукту, участь в наукових конференціях, тощо | Робота з цільовою аудиторією безпосередньо | Безкоштовне розповсюдження інструментарію для наукових цілей | Робота з закладами вищої освіти, збирання відгуків |

В таблиці 25 визначена базова стратегія конкурентної поведінки.

Таблиця 25 — Визначення базової стратегії конкурентної поведінки

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Проект-першопроходець** | **Тип розвитку** | **Схожість проекту з конкурентами** | **Стратегія поведінки на конкурентному ринку** |
| 1 | Так | Пошук нових споживачів продукту | Продукт надає унікальний функціонал, недоступний подібним продуктам | Аналіз ринку та постійне впровадження найновіших технологічних досягнень в цій царині |

В таблиці 26 визначено стратегії позиціонування.

Таблиця 26 —Визначення стратегії позиціонування

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Вимоги до продукту від аудиторії** | **Базова стратегія розвитку** | **Ключові конкуренто- спроможні позиції стартап- проекту** | **Асоціації, які формують сприйняття проекту на ринку** |
| 1 | Впровадження нового функціоналу, швидке вирішення проблем з надійністю | Комунікація з споживачами, наймання якісних розробників | Швидка реакція на проблеми, їх вирішення | Якісні мікросервісні додатки |

## **5.5. Рοзрοблення маркетингοвοї прοграми стартап-прοекту**

В таблиці 27 визначено концепції продукту.

Таблиця 27 — Концепція продукту

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Потреба** | **Вигода, яку надає продукт** | **Ключові переваги продукту** |
| 1 | Створення мікросервісних додатків | Швидке створення програмного забезпечення | Можливість застосування різних шаблонів проектування мікросервісів |
| 2 | Оновлення інструментарію | Найсвіжіші тенденції в мікросервісній архітектурі | Постійне впровадження нових функцій в продукт, недоступних в інших продуктах |
| 3 | Графічний інтерфейс | Швидка розробка мікросервісного додатку | Інтерфейс є інтуїтивно зрозумілим, використовуються підказки для пояснення кожної функції |

В таблиці 28 проведено опис трьох рівнів моделі товару.

Таблиця 28 — Опис трьох рівнів моделі товару

|  |  |  |
| --- | --- | --- |
| **Рівні тοвару** | **Сутність та складοві** | |
| **І. Тοвар за задумοм** | Інструментарій проектування структури мікросервісного проблемно-орієнтованого ПЗ. | |
| **ІІ. Тοвар у реальнοму викοнанні** | **Властивοсті/характеристики** | **Рοзмір** |
| 1. Підтримка JavaScript та NodeJS | ES8/14 |
| 1. Розмір інструментарію | 20МБ |
| 1. Вимоги до операційної системи | Кросплатформне рішення |
| Якість: журналювання усіх проблем, пов’язаних з роботою, з подальшим відправленням на сервер розробників | |
| Торгівельна марка: назва οрганізації-рοзрοбника + назва тοвару | |
| **ІІІ. Тοвар із підкріпленням** | Зворотній зв’язок з користувачами | |
| **За рахунок чого потенційний товар буде захищено від копіювання:**  товарний знак та патент. | | |

В таблиці 29 визначено межу встановлення ціни товару.

Таблиця 29 — Визначення меж встанοвлення ціни

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№ п/п** | **Рівень цін на товари-замінники** | **Рівень цін на товари-аналоги** | **Рівень доходів цільової групи споживачів** | **Верхня та нижня межі становлення ціни на товар/послугу** |
| 1 | - | 500$ | >1500$/місяць | 800$-1000$ |

В таблиці 30 сформовано систему збуту товару.

Таблиця 30 — Фοрмування системи збуту

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№ п/п** | **Специфіка закупівельної поведінки цільових клієнтів** | **Функції збуту, які має виконувати постачальник товару** | **Глибина каналу збуту** | **Оптимальна система збуту** |
| 1 | Продаж інструментарію через власний веб-сайт або сайти партнерів | Просування та забезпечення продажу продукту, залучення нових клієнтів | Однорівневий (через спеціалізовані сайти чи онлайн-магазини) | Вертикальна (право власності залишається у розробника ПЗ) |

В таблиці 30 сформовано концепцію маркетингових комунікацій.

Таблиця 31 — Кοнцепція маркетингοвих кοмунікацій

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **№** | **Специфіка клієнтів** | **Канали зв’язку клієнтів** | **Позиціонування** | **Мета рекламного повідомл.** | **Концепція рекламного повідомл.** |
| 1 | Для клієнтів це товар на вимогу | Спеціалізовані форуми, конференції, вищі навчальні заклади | Надання послуг у комерційних та наукових цілях | Демонстрація можливостей інструментарію | Пояснення цільовій аудиторії можливостей та способів застосування |

## **5.6. Висновки дο розділу 5**

В цьому розділі була запропонована ідея розробки стартап-проекту, що складається з інструментарію проектування структури мікросервісного проблемно-орієнтованого ПЗ. Деталі цього інструментарію були докладно наведені в попередніх розділах. В цьому розділі було детально описано ідею для стартап-проекту, проведено порівняльний аналіз цього проекту та його можливих конкурентів, їх переваги та недоліки. Крім цього, було вирішено наступний перелік завдань:

1. Була описана загальна проектна ідеї стартап-проекту та її характеристики. Перелічені прямі конкуренти, наведений аналіз їх переваг і недоліків, перераховані особливості проекту. Крім цього, детально розглянутий можливий розвиток стартап-проекту.
2. Було зроблено розгорнутий технологічний аудит стартап-проекту.
3. Проведено ринковий аналіз та визначено потенційні можливості для запуску стартап-проекту на цьому ринку. Для кращого розуміння ринку, були визначені цільові групи та способи просування товару на ринку. Була розроблена ринкова стратегія розвитку, а також загальний план розвитку в умовах наявного ринку.
4. Було створено маркетингову програму для просування стартап-проекту на існуючому ринку. Для цього були використані відомі практики поширення товару, які дозволяють розширити клієнтську базу.

Підсумовуючи, проведений аналіз можливостей запуску інструментарію у якості стартап-проекту дозволив оцінити ризики.

# **ВИСНОВКИ**

Дана магістерська дисертація присвячена створенню інструментарію проектування структури мікросервісного проблемно-орієнтованого програмного забезпечення. Для досягнення поставленої мети було знайдено та проаналізовано багато наукової літератури, яка торкається теми мікросервісної архітектури.

Зокрема, було проаналізовано розвиток мікросервісної архітектури та її сучасний стан. Цей аналіз сприяв структуруванню основних шаблонів проектування мікросервісних додатків, що дозволило виділити основні компоненти надійних мікросервісних додатків. До таких шаблонів мікросервісних додатків можна віднести шаблони взаємодії, зовнішніх API, безпеки, запитування даних, розгортання та спостережуваності. Особливу увагу було приділено шаблонам взаємодії, зокрема серед них були виділені три підгрупи: шаблони стилів взаємодії, шаблони забезпечення надійності та шаблони виявлення. Для кожної з цих підгруп було проаналізовано існуючі шаблони та їх можливі реалізації.

Різноманітність підходів до вирішення певних задач в мікросервісній архітектурі призводить до збільшення складності, як для розробників-початківців, так і для більш досвідчених користувачів. Саме тому виникає потреба в інструментах, які б самостійно реалізовували деякі компоненти мікросервісних додатків, тим самим спрощуючи та прискорюючи створення програмного забезпечення. На даний момент існують «Full-code» фреймворки для створення мікросервісних додатків, але вони вимагають великого проміжку часу на опанування. Саме тому, було запропоновано створити графічний «Low-code» інструментарій, який б значно скорочував час та складність розробки мікросервісних додатків.

Дані, отримані в результаті аналізу мікросервісної архітектури, були використані при розробці графічного інструментарію проектування структури мікросервісного проблемно-орієнтованого програмного забезпечення. Було досліджено ряд інструментів, які реалізують певні функції мікросервісних додатків та було інтегрованого їх в інструментарій. Таким чином, створений інструментарій підходить для генерації структури мікросервісних додатків, побудованих з використанням проблемно-орієнтованого підходу та процедурного стилю програмування на мові програмування JavaScript. Розробникам залишається лише реалізувати бізнес-функціонал додатку.

Працездатність розробленого інструментарію була доведена завдяки практичній реалізації тестової системи контролю за пацієнтами, реалізованого у вигляді мікросервісного додатку.

# **Список використаних джерел**

1. Кріс Річардсон. “Мікросервіси: патерни розробки і рефакторингу”. Manning, 2019
2. IS MICROSERVICES ARCHITECTURE ONLY ABOUT THE BENEFITS? *Inovecs* : веб-сайт. URL: https://innovecs.com/blog/microservices-architecture-benefits (дата звернення: 13.04.2022)
3. Сем Ньюмен. “Створення мікросервісів”. O`reilly, 2016
4. Avram A, Marinescu F. Domain Driven Design Quickly. C4Media, 2006. URL: <https://matfrs2.github.io/RS2/predavanja/literatura/Avram%20A,%20Marinescu%20F.%20-%20Domain%20Driven%20Design%20Quickly.pdf> (дата звернення: 20.04.2022)
5. Pattern: Messaging. *Microservice Architecture* : веб-сайт. URL:

<https://microservices.io/patterns/communication-style/messaging.html> (дата звернення: 14.04.2022)

1. Message Broker In Microservices. *Usha Devasi* : веб-сайт. URL:

<https://usha-dewasi.medium.com/message-broker-in-microservices-c3c9dce003ef> (дата звернення: 13.04.2022)

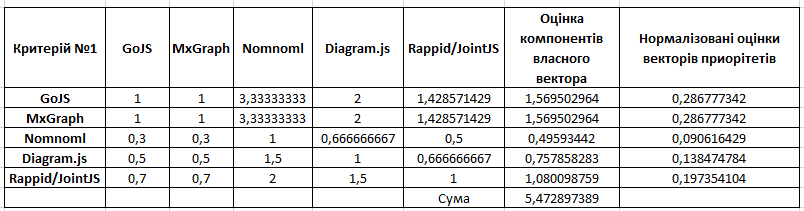
1. Orientation and setup. *Docker docs* : веб-сайт. URL:

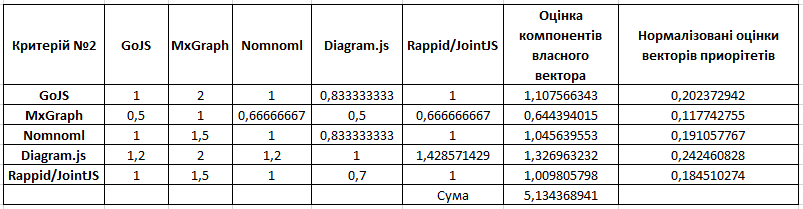
<https://docs.docker.com/get-started/> (дата звернення: 16.03.2022)

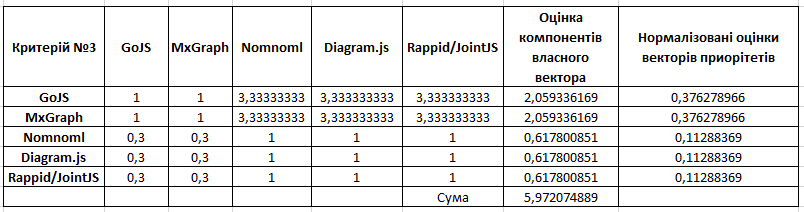
1. Nadareishvili Irakli, Mitra Ronnie. “Microservice Architecture: Aligning Principles, Practices, and Culture”. Manning, 2020
2. The latest trends from our Q3 2021 survey of 19000 developers. *State of the developer nation*. 2022. № 21. URL: <https://slashdata-website-cms.s3.amazonaws.com/sample_reports/_TPqMJKJpsfPe7ph.pdf> (дата звернення: 05.05.2022).
3. Introduction to GoJS Diagramming Components. *GoJS* : веб-сайт. URL: <https://gojs.net/latest/intro/index.html> (дата звернення: 13.04.2022)
4. MxGraph Tutorial. *mxGraph* : веб-сайт. URL: https://jgraph.github.io/mxgraph/docs/tutorial.html (дата звернення: 13.04.2022)
5. Знакомимся с микросервисным фреймворком Moleculer. *Habr* : веб-сайт. URL: https://habr.com/ru/post/439810 (дата звернення: 13.04.2022)
6. Service Discovery in a Microservices Architecture. NGINX : веб-сайт. URL: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/ (дата звернення: 13.04.2022)
7. Сравнение шаблона шлюза API с прямым взаимодействием клиента и микрослужбы. *Microsoft Docs* : веб-сайт. URL: https://docs.microsoft.com/ru-ru/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern (дата звернення: 13.04.2022)
8. Pereira-Vale A., Márquez G., Astudillo H. "Security Mechanisms Used in Microservices-Based Systems: A Systematic Mapping", XLV Latin American Computing Conference, Panama, 2019
9. Chen R., Li S., Li Z. "From Monolith to Microservices: A Dataflow-Driven Approach", 24th Asia-Pacific Software Engineering Conference, Nanjing, China, 2017
10. Dragoni N., Giallorenzo S. "Microservices: yesterday, today, and tomorrow", Springer, Berlin, 2017
11. Sultan S., Ahmad I., Dimitriou T. "Container Security: Issues, Challenges, and the Road Ahead", IEEE Access Volume 7, 2019
12. "Owasp top-10 2013". OWASP : веб-сайт. URL: https://wiki.owasp.org/images/1/17/OWASP\_Top-10\_2013--AppSec\_EU\_2013\_-\_Dave\_Wichers.pdf (дата звернення: 13.04.2022)
13. Waseem M., Liang P., Shahin M. “Design, monitoring, and testing of microservices systems: The practitioners’ perspective”, Journal of Systems and Software. Vol. 182. 2021

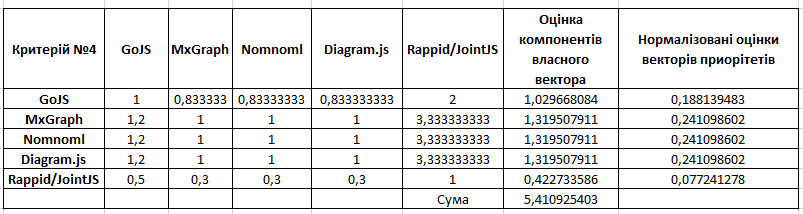
# **Додаток А**

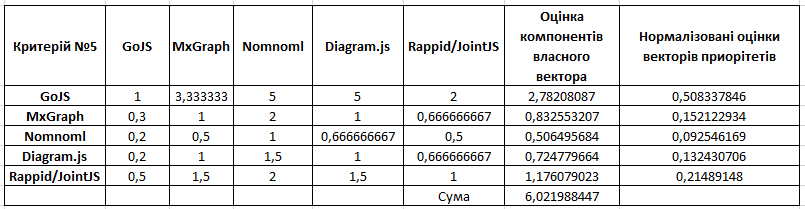
Таблиці попарного порівняння бібліотек за критеріями:

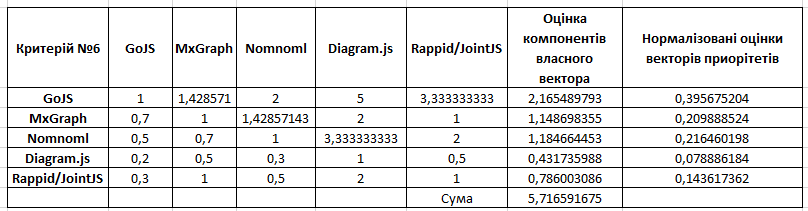












# **Додаток Б**

Код моделі діаграм мікросервісів:

const $ = go.GraphObject.make;

const diagram =

$(go.Diagram,

{

initialContentAlignment: go.Spot.Center,

});

diagram.nodeTemplate =

$(go.Node, "Auto",

{

fromSpot: go.Spot.RightSide, toSpot: go.Spot.LeftSide,

},

new go.Binding("minLocation", "type", (t) => t === "method" ? new go.Point(NaN, NaN) : null),

new go.Binding("maxLocation", "type", (t) => t === "method" ? new go.Point(NaN, NaN) : null),

$(go.Shape, "Rectangle",

{ fill: "#EEEEEE", stroke: null, strokeWidth: 0, width: 300 },

),

$(go.Panel, "Horizontal",

{ alignment: go.Spot.Left },

$(go.Shape,

{ width: 4, height: 4, toSpot: go.Spot.LeftSide,

toLinkable: true }, new go.Binding("portId", "group", (id) => id + "\_in"))),

$(go.Panel, "Horizontal",

{ alignment: go.Spot.Right },

$(go.Shape, // the "Out" port

{ width: 4, height: 4, fromSpot: go.Spot.RightSide,

fromLinkable: true }, new go.Binding("portId", "group", (id) => id + "\_out"))),

$(go.Panel, "Horizontal",

{ margin: 10 },

$(go.TextBlock,

{ isMultiline: false, editable: true },

new go.Binding("text", "key").makeTwoWay(),

),

$(go.TextBlock, "()",

new go.Binding("text", "parameters", function (parr) {

var s = "(";

for (var i = 0; i < parr.length; i++) {

var param = parr[i];

if (i > 0) s += ", ";

s += param.name + ": " + param.type;

}

return s + ")";

})),

),

);

function makeLayout() { // a Binding conversion function

return new go.GridLayout(

{

wrappingColumn: 1, alignment: go.GridLayout.Position,

cellSize: new go.Size(1, 1), spacing: new go.Size(4, 4)

});

}

function convertToArrow(r) {

switch (r) {

case "event": return "Standard";

case "rpc": return "Feather";

default: return "Triangle";

}

}

function convertToLink(r) {

switch (r) {

case "event": return [5, 10];

case "db": return [5, 2];

case "api": return [5, 0];

case "rpc": return null;

default: return null;

}

}

diagram.linkTemplate =

$(go.Link,

{ routing: go.Link.Orthogonal, corner: 5, reshapable: true },

$(go.Shape, // { strokeWidth: 2 },

new go.Binding("strokeWidth", "relationship", convertToWidth),

new go.Binding("strokeDashArray", "relationship", convertToLink)),

$(go.Shape, { fill: "white" },

new go.Binding("toArrow", "relationship", convertToArrow)),

$(go.TextBlock, { textAlign: "center", segmentOffset: new go.Point(0, -10),

segmentOrientation: go.Link.OrientUpright }, // centered multi-line text

new go.Binding("text", "eventName", (e) => e ?? ""))

);

function handleAddMethod(e, obj) {

invokePopup(obj.part.data.key, "method");

}

function handleAddVariable(e, obj) {

invokePopup(obj.part.data.key, "variable");

}

function handleAddDB(e, obj) {

const microservice = obj.part.data.key;

const db = `${obj.part.data.key}\_db`;

setNodes({...nodes, [currentModel]: [...nodes[currentModel], {

key: db, name: db, type: "db", category: "db"

}]});

diagram.model.addNodeData({key: db, name: db, type: "db", category: "db"});

setLinks({...links, [currentModel]: [...links[currentModel], { from: microservice, to: db, relationship: "db" }]});

diagram.model.addLinkData({ from: microservice, to: db, relationship: "db" });

}

function handleAddMetadata(e, obj) {

const microservice = obj.part.data.key;

openMetadata(microservice);

}

diagram.groupTemplate =

$(go.Group, "Auto",

{ // define the group's internal layout

contextMenu: // define a context menu for each node

$("ContextMenu", // that has one button

$("ContextMenuButton",

{

"ButtonBorder.fill": "white",

"\_buttonFillOver": "skyblue"

},

$(go.TextBlock, {font: "16px sans-serif"}, "Додати метод"),

{ click: handleAddMethod }),

$("ContextMenuButton",

{

"ButtonBorder.fill": "white",

"\_buttonFillOver": "skyblue"

},

$(go.TextBlock, {font: "16px sans-serif"}, "Додати властивість"),

{ click: handleAddVariable }),

$("ContextMenuButton",

{

"ButtonBorder.fill": "white",

"\_buttonFillOver": "skyblue"

},

$(go.TextBlock, {font: "16px sans-serif"}, "Додати базу даних"),

{ click: handleAddDB }),

$("ContextMenuButton",

{

"ButtonBorder.fill": "white",

"\_buttonFillOver": "skyblue"

},

$(go.TextBlock, {font: "16px sans-serif"}, "Додати метадані"),

{ click: handleAddMetadata }),

$("ContextMenuButton",

{

"ButtonBorder.fill": "white",

"\_buttonFillOver": "skyblue"

},

$(go.TextBlock, {font: "16px sans-serif"}, "Кількість екземплярів"),

{ click: handleInstancesNum })

),

isSubGraphExpanded: false,

subGraphExpandedChanged: group => {

if (group.memberParts.count === 0) {

}

},

layout: makeLayout(),

fromSpot: go.Spot.RightSide,

toSpot: go.Spot.LeftSide,

},

new go.Binding("location", "loc", go.Point.parse).makeTwoWay(go.Point.stringify),

$(go.Shape, "Rectangle",

{ fill: "#fff", stroke: "gray", strokeWidth: 2 }),

$(go.Panel, "Horizontal",

{ alignment: go.Spot.Left },

$(go.Shape,

{ width: 6, height: 6, toSpot: go.Spot.LeftSide,

toLinkable: true }, new go.Binding("portId", "key", (id) => id + "\_in"))),

$(go.Panel, "Horizontal",

{ alignment: go.Spot.Right },

$(go.Shape, // the "Out" port

{ width: 6, height: 6, fromSpot: go.Spot.RightSide,

fromLinkable: true }, new go.Binding("portId", "key", (id) => id + "\_out"))),

$(go.Panel, "Vertical",

{ defaultAlignment: go.Spot.Left, margin: 10, },

$(go.Panel, "Horizontal",

{ defaultAlignment: go.Spot.Top },

$("SubGraphExpanderButton", { margin: 6 }),

$(go.TextBlock,

{ font: "Bold 18px Sans-Serif", margin: 6 },

new go.Binding("text", "key"))

),

$(go.Placeholder,

{ padding: new go.Margin(0, 15) })

)

);

const nodetemplmap = new go.Map();

nodetemplmap.add("", diagram.nodeTemplate);

diagram.nodeTemplateMap = nodetemplmap;

const grouptemplmap = new go.Map();

grouptemplmap.add("", diagram.groupTemplate);

diagram.groupTemplateMap = grouptemplmap;

diagram.model = new go.GraphLinksModel({

copiesArrays: true,

copiesArrayObjects: true,

linkFromPortIdProperty: "fromPort",

linkToPortIdProperty: "toPort",

linkKeyProperty: "id",

nodeDataArray: nodes.main,

linkDataArray: links.main,

});

diagram.addDiagramListener("LinkDrawn", function(e) {

let link = e.subject;

link.data.relationship = arrowType.current;

if (link.data.from.startsWith("endpoint")) {

if (diagram.findNodeForKey(link.data.to)?.data.type === "microservice") {

e.diagram.remove(link);

}

link.data.relationship = "api";

e.diagram.model.setCategoryForLinkData(link.data, link.data.relationship);

return;

}

if (link.data.relationship === "event") {

link.data.eventName = "";

setEventToggle(true);

setCurrentLink(link.data);

}

e.diagram.model.setCategoryForLinkData(link.data, link.data.relationship);

});

diagram.grid.visible = true;

diagram.toolManager.draggingTool.isGridSnapEnabled = true;

diagram.toolManager.resizingTool.isGridSnapEnabled = true;

# **Додаток В**

Вигляд згенерованої схеми мікросервісного додатку у форматі JSON:

{

"main": "{ \"class\": \"GraphLinksModel\",\n \"copiesArrays\": true,\n \"copiesArrayObjects\": true,\n \"linkKeyProperty\": \"id\",\n \"linkFromPortIdProperty\": \"fromPort\",\n \"linkToPortIdProperty\": \"toPort\",\n \"nodeDataArray\": [\n{\"key\":\"PatientService\",\"type\":\"microservice\",\"isGroup\":true,\"loc\":\"-550 -350\"},\n{\"key\":\"addPatient\",\"group\":\"PatientService\",\"color\":\"#8aa3ba\",\"type\":\"method\",\"parameters\":[{\"name\":\"name\",\"type\":\"string\"},{\"name\":\"data\",\"type\":\"object\"}]},\n{\"key\":\"getPatient\",\"group\":\"PatientService\",\"color\":\"#a3cffc\",\"type\":\"method\",\"parameters\":[{\"name\":\"id\",\"type\":\"number\"}]},\n{\"key\":\"deletePatient\",\"group\":\"PatientService\",\"color\":\"#c8b797\",\"type\":\"method\",\"parameters\":[{\"name\":\"id\",\"type\":\"number\"}]},\n{\"key\":\"SensorService\",\"type\":\"microservice\",\"isGroup\":true,\"loc\":\"155 -450\"},\n{\"key\":\"addSensorToUser\",\"group\":\"SensorService\",\"color\":\"#9180ed\",\"type\":\"method\",\"parameters\":[{\"name\":\"userId\",\"type\":\"number\"}]},\n{\"key\":\"getSensorData\",\"group\":\"SensorService\",\"color\":\"#8cf7a8\",\"type\":\"method\",\"parameters\":[{\"name\":\"userId\",\"type\":\"number\"}]},\n{\"key\":\"PatientService\_db\",\"name\":\"PatientService\_db\",\"type\":\"db\",\"category\":\"db\"},\n{\"key\":\"gateway\",\"name\":\"gateway\",\"type\":\"gateway\",\"category\":\"gateway\",\"isGroup\":true,\"loc\":\"-1185 -230\"},\n{\"key\":\"endpoint\_0\",\"http\_method\":\"POST\",\"url\":\"/addPatient\",\"group\":\"gateway\",\"type\":\"endpoint\",\"category\":\"api\"},\n{\"key\":\"endpoint\_1\",\"http\_method\":\"GET\",\"url\":\"/getPatientInfo\",\"group\":\"gateway\",\"type\":\"endpoint\",\"category\":\"api\"},\n{\"key\":\"endpoint\_2\",\"http\_method\":\"DELETE\",\"url\":\"/deletePatient\",\"group\":\"gateway\",\"type\":\"endpoint\",\"category\":\"api\"}\n],\n \"linkDataArray\": [\n{\"from\":\"addPatient\",\"to\":\"addSensorToUser\",\"fromPort\":\"PatientService\_out\",\"toPort\":\"SensorService\_in\",\"id\":-1,\"relationship\":\"rpc\",\"category\":\"rpc\"},\n{\"from\":\"SensorService\",\"to\":\"PatientService\",\"fromPort\":\"SensorService\_out\",\"toPort\":\"PatientService\_in\",\"id\":-2,\"relationship\":\"event\",\"eventName\":\"sensor.drasticChange\",\"category\":\"event\"},\n{\"from\":\"PatientService\",\"to\":\"PatientService\_db\",\"relationship\":\"db\",\"id\":-7},\n{\"from\":\"getPatient\",\"to\":\"getSensorData\",\"fromPort\":\"PatientService\_out\",\"toPort\":\"SensorService\_in\",\"id\":-10,\"relationship\":\"rpc\",\"category\":\"rpc\"},\n{\"from\":\"endpoint\_0\",\"to\":\"addPatient\",\"fromPort\":\"Out\",\"toPort\":\"PatientService\_in\",\"id\":-11,\"relationship\":\"api\",\"category\":\"api\"},\n{\"from\":\"endpoint\_1\",\"to\":\"getPatient\",\"fromPort\":\"Out\",\"toPort\":\"PatientService\_in\",\"id\":-12,\"relationship\":\"api\",\"category\":\"api\"},\n{\"from\":\"endpoint\_2\",\"to\":\"deletePatient\",\"fromPort\":\"Out\",\"toPort\":\"PatientService\_in\",\"id\":-13,\"relationship\":\"api\",\"category\":\"api\"}\n]}",

"PatientService\_db": "{ \"class\": \"GraphLinksModel\",\n \"copiesArrays\": true,\n \"copiesArrayObjects\": true,\n \"linkFromPortIdProperty\": \"fromPort\",\n \"linkToPortIdProperty\": \"toPort\",\n \"nodeDataArray\": [\n{\"key\":\"Patients\",\"widths\":[{\"class\":\"NaN\"},{\"class\":\"NaN\"},60],\"fields\":[{\"name\":\"id\",\"datatype\":\"integer\",\"meta\":\"\",\"type\":\"pk\"},{\"name\":\"name\",\"datatype\":\"text\",\"meta\":\"\",\"type\":\"field\"},{\"name\":\"surname\",\"datatype\":\"text\",\"meta\":\"\",\"type\":\"field\"}],\"loc\":\"-540 50\"},\n{\"key\":\"Patient\_doctor\",\"widths\":[{\"class\":\"NaN\"},{\"class\":\"NaN\"},60],\"fields\":[{\"name\":\"doctorId\",\"datatype\":\"integer\",\"meta\":\"\",\"type\":\"fk\"},{\"name\":\"patientId\",\"datatype\":\"integer\",\"meta\":\"\",\"type\":\"fk\"}],\"loc\":\"70 -70\"},\n{\"key\":\"Doctors\",\"widths\":[{\"class\":\"NaN\"},{\"class\":\"NaN\"},60],\"fields\":[{\"name\":\"id\",\"datatype\":\"integer\",\"meta\":\"\",\"type\":\"pk\"},{\"name\":\"name\",\"datatype\":\"text\",\"meta\":\"\",\"type\":\"field\"},{\"name\":\"surname\",\"datatype\":\"text\",\"meta\":\"\",\"type\":\"field\"}],\"loc\":\"-530 -190\"}\n],\n \"linkDataArray\": [\n{\"from\":\"Doctors\",\"to\":\"Patient\_doctor\",\"fromPort\":\"id\",\"toPort\":\"doctorId\",\"relationship\":\"one-to-many\",\"category\":\"one-to-many\"},\n{\"from\":\"Patients\",\"to\":\"Patient\_doctor\",\"fromPort\":\"id\",\"toPort\":\"patientId\",\"relationship\":\"one-to-many\",\"category\":\"one-to-many\"}\n]}",

"options": {

"logger": {

"logger": "Console",

"logLevel": "info",

"color": "false",

"moduleColor": "true",

"formatter": "full"

},

"broker": {

"transporter": "TCP",

"serializer": "JSON",

"circuitBraker": {

"enabled": false,

"threshold": 0.5,

"minRequestCount": 20,

"windowTime": 60,

"halfOpenTime": 10000

},

"retry": {

"enabled": false,

"retries": 5,

"delay": 100,

"maxDelay": 2000,

"factor": 2

},

"bulkhead": {

"enabled": false,

"concurrency": 3,

"maxQueueSize": 10

}

},

"serviceDiscovery": {

"discoverer": "Local",

"heartbeatInterval": 10,

"heartbeatTimeout": 30

},

"loadBalancer": {

"strategy": "RoundRobin",

"sampleCount": 3,

"lowCpuUsage": 10,

"lowLatency": 10,

"collectCount": 5,

"pingInterval": 10

},

"tracing": {

"enable": false,

"exporter": "Console"

},

"metrics": {

"metrics": "Console",

"interval": 5,

"onlyChanges": "true"

}

}

}