НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра інженерії програмного забезпечення в енергетиці

«На правах рукопису»
УДК 004.4

ДО ЗАХИСТУ ДОПУЩЕНО
В.о. завідувача кафедри
_____Олександр КОВАЛЬ

……. ..... 2024 р.

Магістерська дисертація

на здобуття ступеня магістра
за освітньо-професійною програмою «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем в енергетиці»
зі спеціальності 121 Інженерія програмного забезпечення
на тему: «Модуль для створення віртуальних серверів та ресурсів для зберігання даних на основі хмарного середовища OpenStack»

Виконав студент 2 курсу, групи ТВ-21мп Лебедєв Артем Вячеславович _____
(шифр, прізвище, ім'я, по батькові) (підпис)

Науковий керівник професор кафедри ІПЗЕ, професор, д.е.н.
Сігайов Андрій Олександрович _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _____ _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2024

Національний технічний університет України
«Київський політехнічний інститут ім. Ігоря Сікорського»

Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці
Рівень вищої освіти другий, магістерський
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем в енергетиці»

ЗАТВЕРДЖУЮ
В.о.завідувача кафедри
_____ Олександр КОВАЛЬ
(підпис)
____ _____ 2024 р.

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ ДИСЕРТАЦІЮ СТУДЕНТУ

Лебедєв Артем Вячеславович
(прізвище, ім'я, по батькові)

1. Тема дисертації: Модуль для створення віртуальних серверів та ресурсів для зберігання даних на основі хмарного середовища OpenStack
Науковий керівник д.е.н., професор, професор кафедри ІПЗЕ Сігайов Андрій Олександрович
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 06 листопада 2023 року № 5152-с
2. Строк подання студентом дисертації: 10 січня 2024 року
3. Об'єкт дослідження: процес розробки системи управління віртуальними серверами та ресурсами.
4. Вихідні дані до роботи: модуль управління віртуальними серверами та ресурсами для зберігання даних з інтерфейсом користувача на основі хмарного середовища OpenStack.
5. Перелік питань, які потрібно розробити: опрацювати документацію хмарного середовища OpenStack; визначити метод взаємодії з OpenStack; створити варіанти можливих конфігурацій серверів з урахуванням потреб користувачів; проаналізувати підходи конкурентів, визначені їх слабкі та сильні сторони; реалізувати програмну систему, з урахуванням методу взаємодії з OpenStack та завчасно створених конфігурацій; визначити напрями розвитку та основні аспекти для подальшого удосконалення системи.

6. Орієнтований перелік ілюстративного матеріалу: діаграма архітектури системи, діаграма прецедентів, діаграма послідовності, діаграма моделей бази даних, огляд існуючих систем, структура проєктів, API документація, процес налаштування системи.
7. Дата видачі завдання: 01 листопада 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання магістерської дисертації	Строки виконання етапів магістерської дисертації	Примітка
1	Отримання завдання	01.11.2022	виконано
2	Дослідження предметної області	01.11.2022 - 01.12.2022	виконано
3	Постановка вимог до проєктування системи	01.12.2022 - 15.12.2022	виконано
4	Дослідження існуючих рішень	15.12.2022 - 03.01.2023	виконано
5	Розробка програмного продукту	03.03.2023 - 20.09.2023	виконано
6	Тестування	20.09.2023 - 15.10.2023	виконано
7	Захист програмного продукту	16.10-2023 – 20.10.2023	виконано
8	Підготовка магістерської дисертації	21.10.2023 – 17.11.2023	виконано
9	Передзахист	18.12.2023 - 22.12.2023	виконано
10	Захист	15.01.2024 – 19.01.2024	виконано

Студент _____ Лебедєв А. В.
(підпис) (прізвище та ініціали)

Науковий керівник _____ Сігайов А. О.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Структура та обсяг магістерської дисертації. Робота містить 94 сторінки, 26 рисунків, 2 додатки та 32 посилання. Магістерська дисертація складається зі вступу, 6 розділів («Постановка задачі створення модуля для роботи з віртуальними серверами», «Аналіз існуючих проблем», «Засоби розробки програмного забезпечення», «Опис програмної реалізації», «Розробка стартап-проєкту»), висновків до кожного з наявних розділів, загальних висновків та списку використаних джерел.

Актуальність теми. Одними з найпопулярніших технологій нашого часу є обробка великих даних, створення та підтримка блокчейн інфраструктури, навчання нейромереж та моделей машинного навчання. Усі вони потребують неабияких ресурсів для можливості якісної та швидкої роботи з ними. Хмарні технології та кластери є основними інфраструктурними інструментами для роботи з цими технологіями.

Наявні продукти для роботи з хмарними кластерами намагаються бути «універсальним інструментом» та надавати користувачам повний спектр всеможливих налаштувань, тим самим збільшуючи поріг легкості їх опанування.

Актуальність теми зумовлена необхідністю наявності сервісу, який балансує між платформами з мінімальними налаштуваннями та «універсальними інструментами», надаючи кінцевому користувачу інтуїтивно зрозумілий та легкий для опанування інтерфейс.

Метою роботи є розробка програмної платформи управління віртуальними серверами та ресурсами для зберігання даних на основі хмарного середовища OpenStack.

Для досягнення поставленої мети виконано такі завдання:

1) опрацьована документація хмарного середовища OpenStack;
2) визначений метод взаємодії з OpenStack;

3) створені варіанти можливих конфігурацій серверів з урахуванням потреб важких обрахунків;

4) проаналізовано підходи конкурентів, визначені їх слабкі та сильні сторони;

5) реалізовано програмну систему, з урахуванням методу взаємодії з OpenStack та завчасно створених конфігурацій;

6) визначено напрями розвитку та основні аспекти для подальшого удосконалення системи.

Об’єкт дослідження — процес розробки системи управління віртуальними серверами та ресурсами.

Предмет дослідження — модуль управління віртуальними серверами та ресурсами для зберігання даних з інтерфейсом користувача на основі хмарного середовища OpenStack.

Практичне значення одержаних результатів полягає в отриманні продукту з можливістю створення через WEB-інтерфейс віртуальних серверів на основі підготовлених конфігурацій для навчання чи роботи з великими даними, нейронними мережами або будь-якою іншою технологією. Результати є готовими для використання на кластері зі встановленим хмарним середовищем OpenStack.

Ключові слова: віртуальні сервери, ресурси для зберігання даних, OpenStack, вебсистема, хмарне середовище, кластер, ресурсозатратні операції.

ABSTRACT

The structure and scope of the master's thesis. The work contains 94 pages, 26 figures, 2 appendices and 32 references. The master's thesis consists of an introduction, 6 chapters («Setting the task of creating a module for working with virtual servers», «Analysis of existing problems», «Software development tools», «Description of software implementation», «Development of a startup project»), conclusions to each of the available chapters, general conclusions and a list of references.

Relevance of the topic. The most popular technologies of our time are big data processing, creation and support of blockchain infrastructure, and training of neural networks and machine learning models. All of them require considerable resources to work with them efficiently and quickly. Cloud technologies and clusters are the main infrastructure tools for working with these technologies.

Existing products for working with cloud clusters try to be a «universal tool» and provide users with a full range of various settings, thereby increasing the threshold for their ease of use.

The relevance of the topic is driven by the need for a service that balances between platforms with minimal customisation and «all-in-one tools», providing the end user with an intuitive and easy-to-use interface.

The research goal of the study is to develop a software platform for managing virtual servers and data storage resources based on the OpenStack cloud environment.

To achieve this goal, the following tasks were performed:

1) the documentation of the OpenStack cloud environment was studied;опрацьована документація хмарного середовища OpenStack;
2) a method of interaction with OpenStack was determined;
3) create variants of possible server configurations to meet the needs of heavy computing;

4) analysed competitors' approaches, identified their weaknesses and strengths;

5) a software system was implemented, taking into account the method of interaction with OpenStack and pre-created configurations;

6) the directions of development and the main aspects for further improvement of the system were identified.

The object of research is the process of creating a virtual server and resource management system.

The subject of research is a module for managing virtual servers and data storage resources with a user interface based on the OpenStack cloud environment.

The practical value of results is to obtain a product with the ability to create virtual servers through the WEB interface based on prepared configurations for training or working with big data, neural networks or any other technology. The results are ready for use on a cluster with an installed OpenStack cloud environment.

Keywords: virtual servers, data storage resources, OpenStack, web system, cloud environment, cluster, resource-intensive operations.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

REST	Representational State Transfer
API	Application Programming Interface
UI/UX	User Interface/User Experience
UML	Unified Modelling Language
ООП	Об'єктно-орієнтоване Програмування
SSH	Secure Shell
CLI	Command-line Interface
SSR	Server-Side Rendering
IP	Internet Protocol
SDLC	Software Development Lifecycle
SPA	Single Page Application
СУБД	Система Управління Базами Даних
DOM	Document Object Model
SCM	Software Configuration Management
YAML	Yet Another Markup Language
SQL	Structured Query Language
JSON	JavaScript Object Notation
SOA	Service-oriented Architecture
URI	Uniform Resource Identifier
DNS	Domain Name System

ВСТУП

У сучасному світі технології розвиваються з неабияким темпом, змінюючи світ та нашу взаємодію з ним за допомогою повної чи часткової автоматизації процесів. Особливого розвитку зазнають обробка великих даних, робота з нейронними мережами та штучним інтелектом, навчання машинних моделей. Усі ці технології дозволяють виконувати задачі, які кілька років тому здавалися, або неможливими, або дуже важкими у реалізації.

Багато стартапів та провідних організацій активно впроваджують ці технології в операційні процеси компаній. Це дозволяє їм скорити витрати на ресурси, тим самим отримавши переваги в конкурентному середовищі бізнесу. Однак для досягнення успіху у використанні новітніх технологій потрібні потужні комп'ютерні ресурси, які здатні витримувати сильні навантаження, гарантувати швидкість операції та цілісність даних. Адже це є важливими вимогами для якісного користувацького досвіду.

Стандартні фізичні сервери та дата-центри вимагають величезних коштів на їх підтримку, надають обмежену функціональність та є складними у налаштуванні. Також фізична серверна інфраструктура дуже важко адаптується під змінне навантаження та великі обсяги даних. Тим паче у разі використання даного підходу компанії зазнають значних додаткових витрат на постійне оновлення програмного забезпечення, обладнання, охорону, безперервне електропостачання, охолодження та спеціалізованих адміністраторів. Додатковою проблемою є ризики природних катастроф в місцях розміщення інфраструктури та важкість налаштування резервних даних на різних локаціях. Саме тому для більшості компаній хмарні технології є незамінними інструментами для створення серверної інфраструктури.

Хмарні технології дозволяють користувачу оперувати віртуальними серверами з гнучкими налаштуваннями, різними видами сховищ даних та надають

гнучку систему для роботи з ролями та дозволами користувачів. Вони мають ряд важливих переваг над стандартною фізичною інфраструктурою:

1) еластичність та масштабованість;
2) оплата лише за використання;
3) гнучкість та швидкість налаштування;
4) висока доступність і надійність;
5) зручне управління ресурсами;
6) безпека та захист даних;
7) віддалений доступ.

Попри усі ці переваги, наявні на ринку рішення мають свої недоліки, особливо для новачків та людей з інших сфер. Вони намагаються бути «універсальними інструментами» з можливістю налаштування будь-яких параметрів та всіх можливих конфігурацій. Тим самим забезпечуючи можливість створення інфраструктури під різноманітні повноцінні кінцеві проєкти користувачів, але додаючи складності в процес взаємодії з ними. Тобто головною їх проблемою є досить високий поріг входу, а їх використання потребує багато зусиль, досвіду, практичних і теоретичних навичок роботи з залізом, операційними системи та адмініструванням загалом. Подібною системою і є OpenStack.

Щобільше, навіть для досвідчених користувачів процес роботи з адміністративними панелями хмарних платформ може потребувати значних сил. Адже налаштування повноцінної інфраструктури з якісно розподіленими ресурсами навіть для тестових середовищ потребує розуміння предметної області системи та можливого навантаження на конкретні види ресурсів. Це призводить до зменшення продуктивність працааівників, які можуть витрачати багато часу на налаштування віртуального сервера замість фокусування на розробці та вдосконаленні продуктів.

Основною метою дипломної роботи є створення програмної платформи, яка спростить процес управління віртуальними серверами та ресурсами для

зберігання даних на основі хмарного середовища OpenStack. Актуальність теми зумовлена необхідністю розробки програмного забезпечення, яке дозволить ефективно створювати віртуальні сервери під різноманітні задачі за завчасно підготовленими конфігураціями. Програмний продукт розроблявся з урахуванням потреб користувачів у наявності спрощеного та інтуїтивно зрозумілого інтерфейсу для забезпечення легкості використання навіть без глибоких технічних знань.

1 ПОСТАНОВКА ЗАДАЧІ СТВОРЕННЯ МОДУЛЯ ДЛЯ РОБОТИ З ВІРТУАЛЬНИМИ СЕРВЕРАМИ

Мета роботи полягає у розробці програмної системи для створення та управління віртуальними серверами та ресурсами для зберігання даних на основі хмарного середовища OpenStack.

1.1 Опис предметної області

Предметна область роботи охоплює широкий спектр різних тем та аспектів, які пов'язані з розробкою, хмарними технологіями, віртуалізацією, безпекою й управлінням доступом, інтеграцією зі сторонніми системами та зберіганням даних.

Хмарні технології виконують ключову роль в автоматизації бізнес-процесів через надання гнучкості та ефективності в управлінні обчислювальними ресурсами. На ринку існує багато систем для роботи з хмарною інфраструктурою, починаючи від маленьких платформ з лімітованим доступом і закінчуючи цифровими гігантами, які надають свої послуги значній частці клієнтів по всьому світу. Однією з них є платформа OpenStack. OpenStack — це проєкт програмного забезпечення для інфраструктури хмарних обчислень з відкритим вихідним кодом, який входить до трійки найактивніших проєктів з відкритим вихідним кодом у світі [1].

Але майже усі наявні рішення намагаються запропонувати користувачеві свободу в діях та налаштуваннях, чим і ускладнюють процес взаємодії з ними. Щобільше, значна частка систем не надають готові рішення з конфігурацій ресурсів під різні типи та складності задач. Подібний підхід впливає на ефективність користування, оскільки вимагає додаткових знань щодо складових комп'ютерної інфраструктури.

1.2 Аналіз поставленої задачі

Тема роботи звучить як «Модуль для створення віртуальних серверів та ресурсів для зберігання даних на основі хмарного середовища OpenStack» та передбачає собою окрему частину адміністративної панелі для роботи з віртуальними серверами.

Адміністративна панель — це інструмент, який використовується для моніторингу та управління ресурсами хмарної інфраструктури, включаючи сервери, сховища та мережі. Інформаційна панель надає централізовану точку доступу до хмарних ресурсів, дозволяючи адміністраторам швидко і легко контролювати стан, продуктивність і якість використання своєї хмарної інфраструктури.

Віртуальний сервер — це тип програмного сервера, який можна створити, розбивши фізичний сервер, який часто називають хостом або металевим сервером, на менші, автономні сегменти. Віртуальний сервер може відтворювати функціональність будь-якого типу сервера, а також ділитися ресурсами з іншими типами віртуальних серверів [2].

За умовами модуль може становити собою WEB або REST API інтерфейс. Враховуючи потреби користувачів та можливу складність використання REST API, було вирішено використовувати підхід з WEB UI для взаємодії з кінцевим користувачем.

Для розробки та тестування описаної вище системи необхідно мати доступ до хмарного середовища OpenStack та налаштувати його відповідним чином.

1.3 Опис основних задач програмного продукту

Для отримання якісної платформи було поставлено ряд функціональних і нефункціональних вимог. Функціональні вимоги відповідають за задачі, які система має виконувати, а нефункціональні — якою саме вона має бути.

Таким чином, до основних функціональних вимог належать:

1) створення віртуальних серверів за заготовленими під різні види задач конфігураціями;
2) перегляд інформації та стану створених віртуальних серверів;
3) повне управління віртуальними серверами;
4) можливість підключення до сервера через веббраузер та консоль;
5) наявність інструкції з інформацією щодо користування платформою;
6) звіти з кількістю створених серверів та розподіл серверів за встановленими на нього додатками.

Нефункціональні вимоги створювалися на основі загальноприйнятих підходів та практик розробки вебсистеми, саме тому вони складаються з наступних пунктів:

1) можливість масштабування;
2) продуктивність та швидкодія;
3) безпека даних та використання системи;
4) економічність у споживанні ресурсів;
5) сумісність з різними операційними системами та веббраузером.

1.4 Призначення продукту та його користувачі

Потенційними користувачами системи можуть бути системні адміністратори, студенти, викладачі та взагалі будь-яка людина, яка має на меті створити віртуальний сервер для задач з різними потребами ресурсів.

Для різних видів користувачів у системі бути створено декілька ролей з різними дозволами:

– звичайний користувач;
– адміністратор.

Звичайний користувач має змогу створити, редагувати, видалити, змінити статус та під'єднатися до віртуальних серверів.

Відмінність адміністратора від звичайного користувача полягає у можливості редагування та видалення інших користувачів та їх віртуальних серверів.

1.5 Опис роботи продукту

Зазвичай розробка програмного забезпечення супроводжуються діаграмами, які описують загальну архітектуру системи, моделі баз даних чи процеси взаємодії користувачів з застосунком. Одним з найпопулярніших видів подібних діаграм є сімейство UML. Уніфікована мова моделювання — це сімейство графічних нотацій, підтримуваних єдиною метамоделлю, які допомагають описувати й проєктувати програмні системи, зокрема ті, які побудовані з використанням стилю ООП [3].

Для явної демонстрації процесів було створено 2 найбільш розповсюджені діаграми поведінки: прецедентів і послідовності.

1.5.1 Діаграма прецедентів

Діаграма прецедентів — це метод фіксації функціональних вимог до системи. Варіанти використання працюють, описуючи типові взаємодії між користувачами системи та самою системою, надаючи опис роботи системи [3].

На рисунку 1.1 зображена діаграма прецедентів, яка описує різні процеси під час взаємодії користувача з системою.

Вона складається з декількох компонентів:

– актори — усі сутності, з якими може взаємодіяти система;
– прецеденти — дії системи;
– межі системи.

Прецеденти можуть бути об'єднані між собою двома видами зв'язків: розширення — необов'язкова, але можлива взаємодія з системою, включення — необов'язкова дія. На даній діаграмі зображені лише зв'язки типу розширення.

Рисунок 1.1 — Діаграма прецедентів

З діаграми зрозуміло, що можливими діями користувачів можуть бути: реєстрація, логін, перегляд звітів, перегляд наявних конфігурацій, створення сервера, перегляд інформації про створені сервери, перегляд деталей сервера, підключення до сервера за допомогою SSH тунелю або вебконсолі та зміна стану серверу. На додаток, відповідно до зображених зв'язків бачимо, що реєстрація розширяється авторизацією, перегляд конфігурацій — створенням сервера, а огляд деталей — підключенням до сервера або зміною його стану.

1.5.2 Діаграма послідовності

Діаграма послідовності відображає поведінку системи в рамках виконання одного сценарію. На діаграмі показують декілька об'єктів і повідомлень, які передаються між цими об'єктами в межах варіанту використання [3].

Рисунок 1.2 демонструє послідовність дій під час створення сервера з точки зору системи.

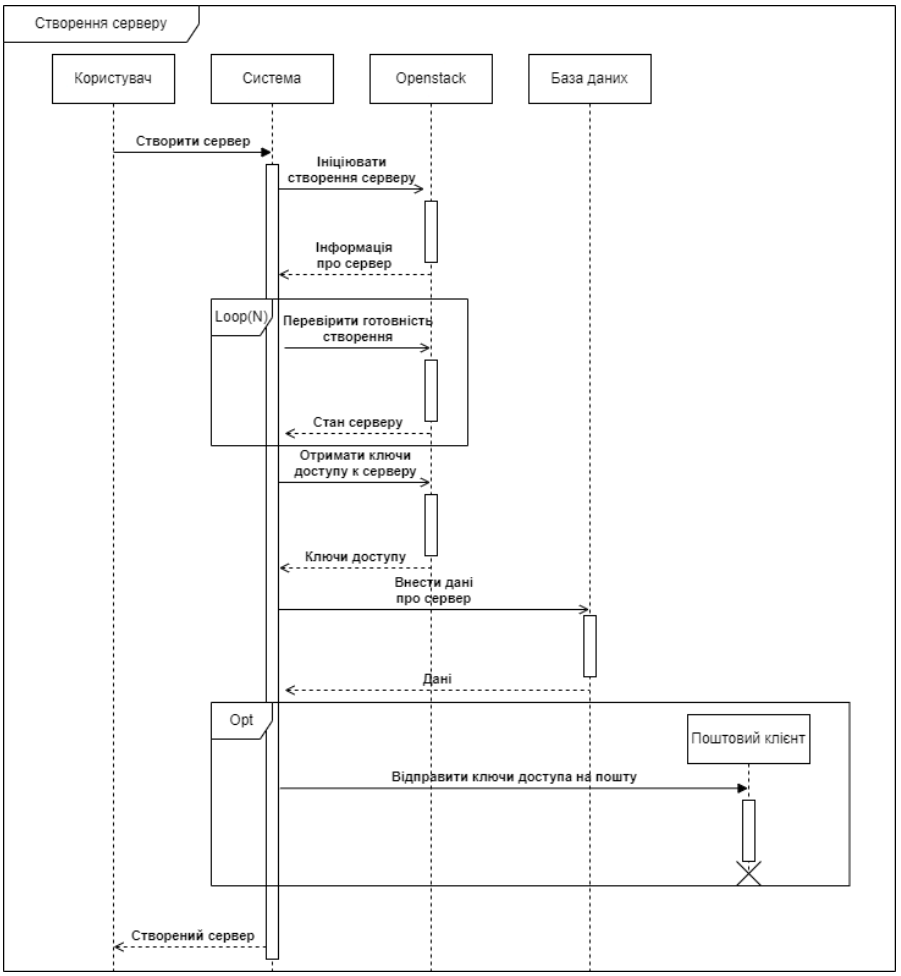


Рисунок 1.2 — Діаграма послідовності

На рисунку зображено чотири об'єкти з лініями життя, які описують тривалість їх існування під час створення сервера: користувач, система, OpenStack та поштовий клієнт.

Діаграма покроково описує дії необхідні для створення сервера:

1) користувач надсилає запит на створення сервера;

2) система ініціює процес створення сервера в OpenStack;

3) система перевіряє стан створення сервера в OpenStack з певною періодичністю;

4) після успішного створення відправляє запит на ключі доступу;

5) вносить відповідність сервера конкретному користувачеві з додатковою інформацією в локальну базу даних;

6) у випадку відсутності створених користувачем серверів раніше, система відправляє загальні ключі доступу користувачеві на пошту.

Кроки зображені верхньорівнево, щоб не вдаватися в деталі реалізації, але кожен з них може бути ще більше докомпонований та мати власну діаграму послідовності.

1.6 Організація процесу виконання задачі

Організація процесу розробки програмного забезпечення є дуже важливим етапом, адже передбачає наявність структурованого та ефективного алгоритму, спрямованого на досягнення визначених цілей у визначені строки.

Розбиття задачі на менші підзадачі, тобто її декомпозиція, дозволить розподілити навантаження на увесь доступний термін, отримати краще розуміння завдання та зменшити ризики й можливі помилки.

Декомпозиція передбачає розбиття складної проблеми або системи на менші частини, які є більш керованими й легшими для розуміння. Потім ці менші частини можна дослідити та вирішити або розробити окремо, оскільки з ними простіше працювати [4].

Задачу створення системи для управління віртуальними серверів та ресурсами для збереження даних було розділено на такі підзадачі:

1) створення базової структури backend частини;
2) інтеграція з OpenStack;
3) проєктування бази даних та її підключення до застосунку;
4) написання модуля авторизації;
5) розробка конфігурацій віртуальних серверів;
6) написання обробників для створення віртуальних серверів та отримання конфігурацій;
7) інтеграція поштового клієнта для інформування користувачів;

8) написання обробників для відображення списку та деталей створених віртуальних серверів;

9) написання обробників для оновлення, видалення та зміни стану віртуальних серверів;

10) написання обробників для підключення до віртуальних серверів;

11) розробка UI/UX дизайну;

12) створення базової структури frontend частини;

13) кодування сторінок авторизації;

14) кодування головної сторінки;

15) кодування сторінки з конфігураціями та створеними серверами;

16) написання форм для створення, оновлення та видалення серверів;

17) кодування сторінки деталей з можливістю підключення до віртуальних серверів через вебконсоль.

Не менш важливим аспектами є відстежування прогресу виконання та пріоритезація завдань. Під час написання проєкту для досягнення цих цілей було використано підхід Kanban.

Kanban — методологія управління проєктами. Kanban дошка — інструмент для візуалізації робочого процесу, де задачі розділені на картки, які переходять з одного статусу в інший. Kanban — це японське слово, яке буквально означає «сигнальна картка». Картки використовуються як сигнал для того, щоб вказати попередньому етапу процесу, що потрібно зробити [5].

На рисунку 1.3 зображена Kanban дошка під час розробки системи. Було виділено 3 основних статуси задач, які повністю описують шлях розробки від декомпозиції до отримання готової функціональної особливості системи: «зробити», «в роботі» та «готово». Звичайно під різні типи та вимоги проєктів статусів може бути більше або навіть менше, але обрані статуси є оптимальними для конкретної роботи.

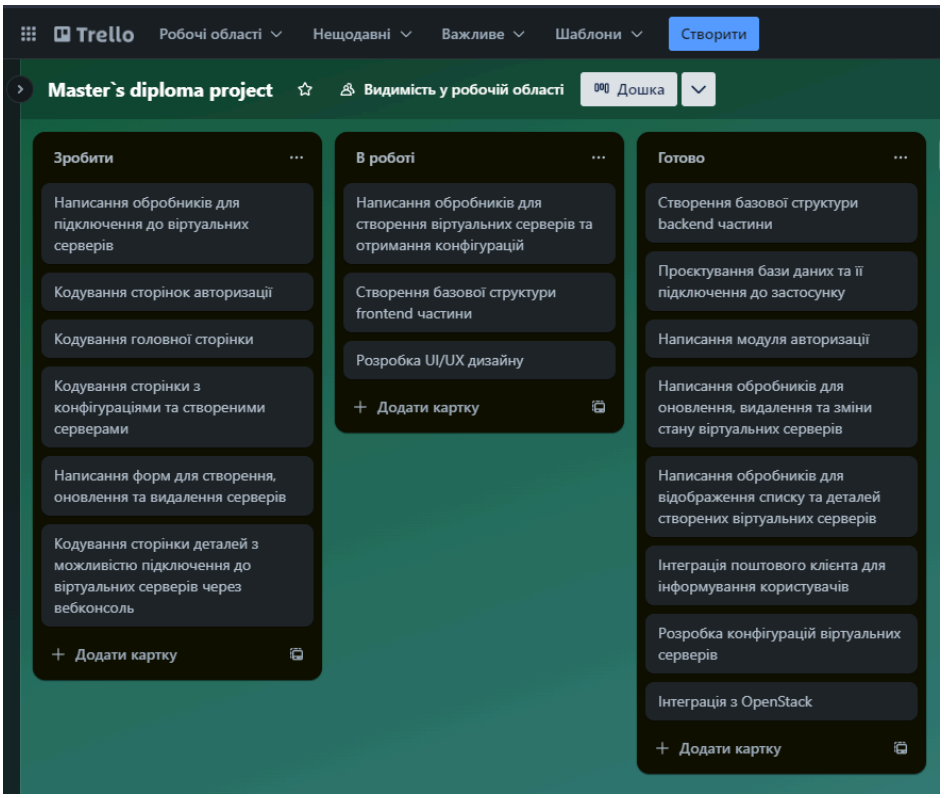


Рисунок 1.3 — Kanban дошка проєкту

За допомогою описаних вище підходів було повністю налагоджено процес створення системи, що своєю чергою посприяло якості та швидкості розробки.

Висновки до розділу 1

У цьому розділі були поставлені цілі та задачі роботи. Результатами планування стали:

– функціональні й нефункціональні вимоги;
– визначені призначення продукту та його цільові користувачі;
– створена діаграма загальної взаємодії з системою;
– створена діаграма послідовності дій під час створення сервера;
– проведена декомпозиція задачі;
– описані інструменти й підходи організації процесу.

Таким чином, завдяки якісному аналізу задачі були запобіженні можливі проблеми та ускладнення під час розробки системи.

2 АНАЛІЗ ІСНУЮЧИХ СИСТЕМ

На сучасному ринку існує декілька систем, які надають можливість управління віртуальними серверами чи ресурсами для зберігання даних з браузера. Кожна платформа надає свій унікальний користувацький інтерфейс, який реалізує процес взаємодії з хмарним середовищем. В цьому розділі буде представлений загальний опис декількох аналогів розроблюваній системі, описані їх основні особливості та плюси й мінуси. Серед розглянутих систем є ті, які взаємодіють виключно з OpenStack, так і провайдери власних рішень для управління хмарним середовищем як Amazon Web Services, Digital Ocean, Google Cloud. Ця інформація була використана під час розробки майбутнього продукту.

2.1 Horizon

Horizon — це реалізація інформаційної панелі OpenStack, що надає вебінтерфейс користувача та адміністратора для різних сервісів хмарної платформи [6]. Horizon за замовчуванням встановлюється разом з OpenStack. Модуль надає можливість взаємодіяти з різними додатками OpenStack без потреби в використання командного рядка або API.

До функціональних особливостей можна віднести можливість створення та керування віртуальними машинами, мережами, образами та іншими ресурсами хмарної інфраструктури. Користувачі можуть легко масштабувати свої ресурси, встановлювати політики доступу та моніторити характеристики своєї інфраструктури через інтуїтивний вебінтерфейс.

Основними плюсами є:

1) модульність, тобто додаток візуально розділений на модулі, які відповідають сервісам OpenStack;

2) підтримка ролей — адміністратори можуть налаштовувати доступи користувачів для певних ресурсів та проєктів;

3) підтримка різних проєктів, Horizon надає можливість користувачам оперувати тільки тими ресурсами, які належать їх проєкту.

Хоча Horizon і є гарним прикладом користувацького інтерфейсу для хмарного середовища, є декілька мінусів, які можна виділити:

1) обмежена функціональність — Horizon покриває більшу частину функціональності OpenStack, яка доступна через CLI, але все ще існують специфічні сценарії, які неможливо вирішити без використання API або CLI;

2) застарілий UI/UX, адже Horizon надає класичний інтерфейс адміністративної панелі, який є трохи перевантаженим для звичайного користувача;

3) підхід SSR (Server Side Rendering), при якому візуалізація сторінки відбувається на сервері та тільки потім відправляється клієнту. Як результат — кожна дія потребує перевантаження сторінки;

4) комплексність налаштувань, яка впливає на процеси створення та управління серверами для нового користувача та потребувати деякого часу на вивчення документації (рис. 2.1);

5) погана продуктивність застосунку, яка відображається навіть при невеликій кількості проєктів з десятками піднятих серверів відчувається велике використання ресурсів системи та затримка;

6) відсутність стандартних спеціалізованих конфігурацій серверів, що ускладнює процес створення повноцінно робочого серверу з публічною IP-адресою;

7) ускладнення процесу створення сервера, який складається з декількох кроків, кожен з яких вимагає конфігурації, що може бути важким та часозатратним для звичайного користувача (рис. 2.2).

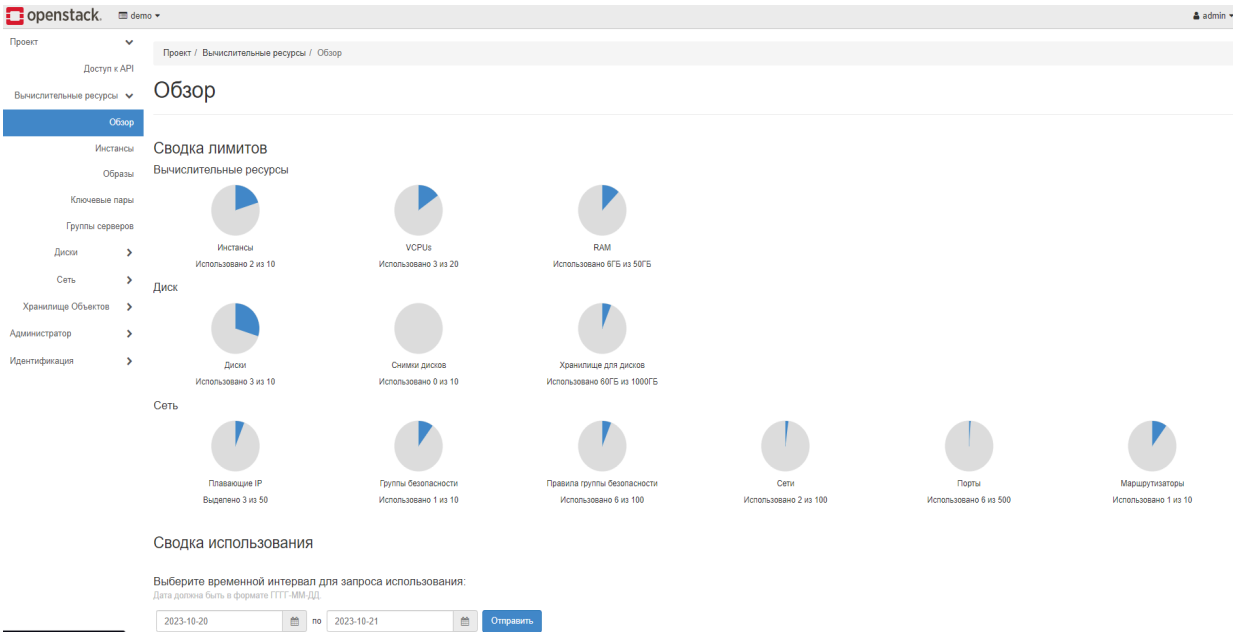


Рисунок 2.1 — Головна сторінка Horizon

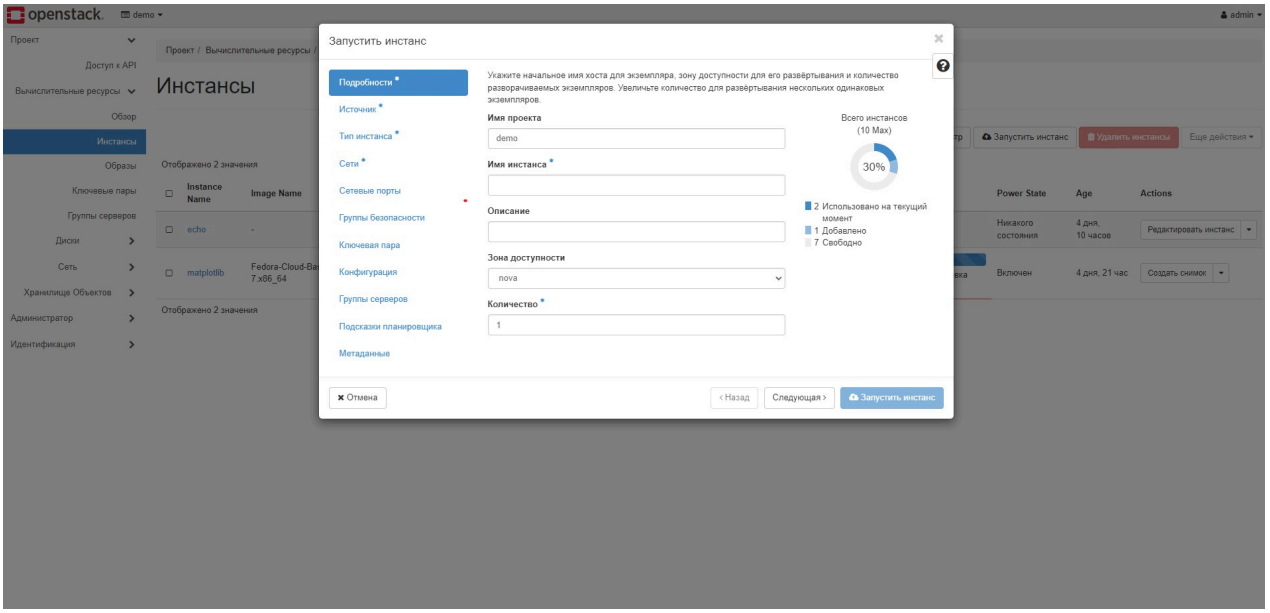


Рисунок 2.2 — Процес створення сервера у Horizon

Таким чином, можна зробити висновок, що Horizon є гарним прикладом системи для повного адміністрування та можливості різноманітної конфігурації хмарного середовища OpenStack. Проте звичайний користувач без наявності базових знань у предметній області та адміністрування інфраструктури можливо буде мати деякі труднощі під час його використання.

2.2 Skyline

Skyline є частиною OpenStack Modern Dashboard, яка надає вебінтерфейс користувача до сервісів OpenStack, включаючи Nova, Swift, Keystone тощо [7]. Він має сучасний технологічний стек та архітектуру, простіший для розробників та експлуатації користувачами, а також має вищу продуктивність.

Основні функціональні особливості Skyline повністю відповідають минулому прикладу, адже вони обидва є просто різними реалізаціями адміністративної панелі для OpenStack.

До плюсів системи можна віднести:

1) розподілена відповідальність, при якій клієнтська частина застосунку фокусується на дизайні та користувацькому досвіді, а серверна — на логіці даних;
2) проста архітектура, яка не інкапсулює непотрібну логіку та повністю відповідає наявному API сервісу OpenStack;
3) використання підпрограм для покращення паралельності, зменшення кількість посилань на виклики для підвищення продуктивності та швидкості роботи застосунку;
4) гарний дизайн та архітектура програмного забезпечення, орієнтовані на користувача;
5) складні процеси є більш оптимізованими під користувача шляхом покращення UX та наявності підказок.

Попри усі описані вище плюси, сервіс все ж таки має певні недоліки, які частково перетинатися з недоліками минулого прикладу:

1) додаток є відносно новим, тому досі має певну кількість помилок і знаходиться в доробці усього функціоналу;
2) хоча сервіс і є дуже зручним й зрозумілим для користувачів, він все ж таки надає розширені можливості для адміністрування, що вимагає від користувача певного досвіду або вивчення документації (рис. 2.3);

3) відсутність стандартних спеціалізованих конфігурацій серверів: як і в минулому прикладі, у системі відсутні стандартні конфігурації сервісів для роботи з великими даними чи штучним інтелектом (рис. 2.4).

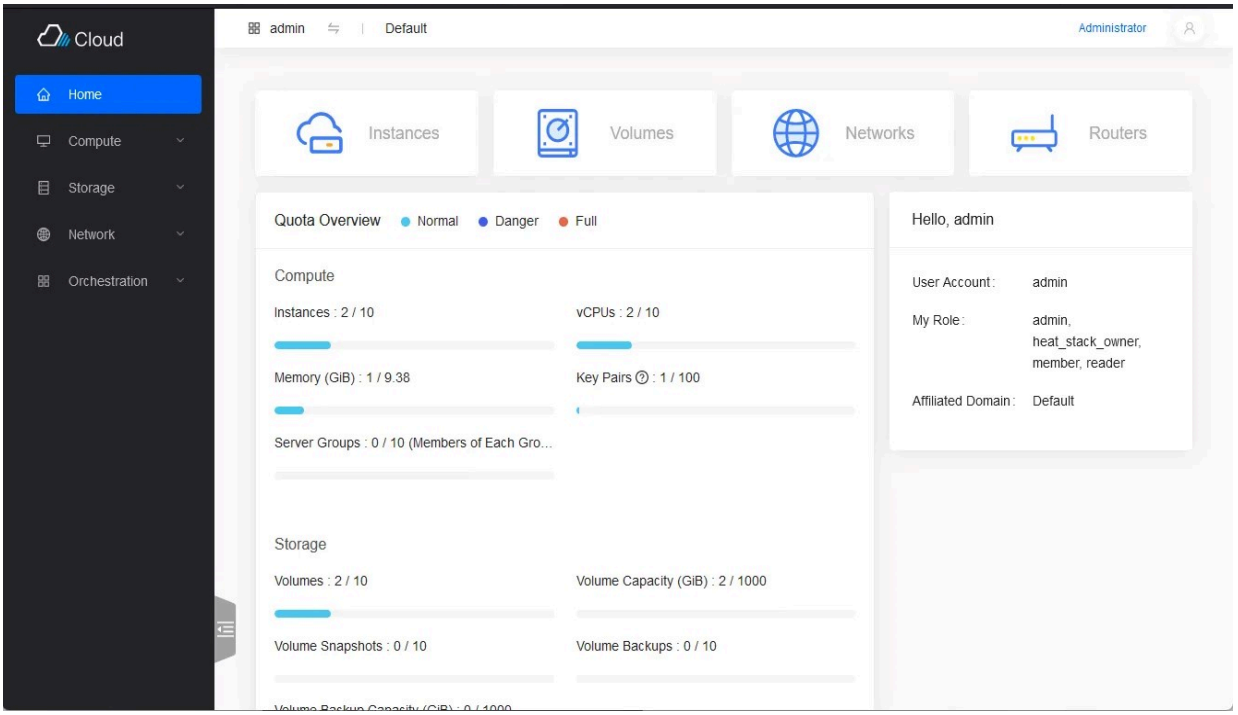


Рисунок 2.3 — Головна сторінка Skyline

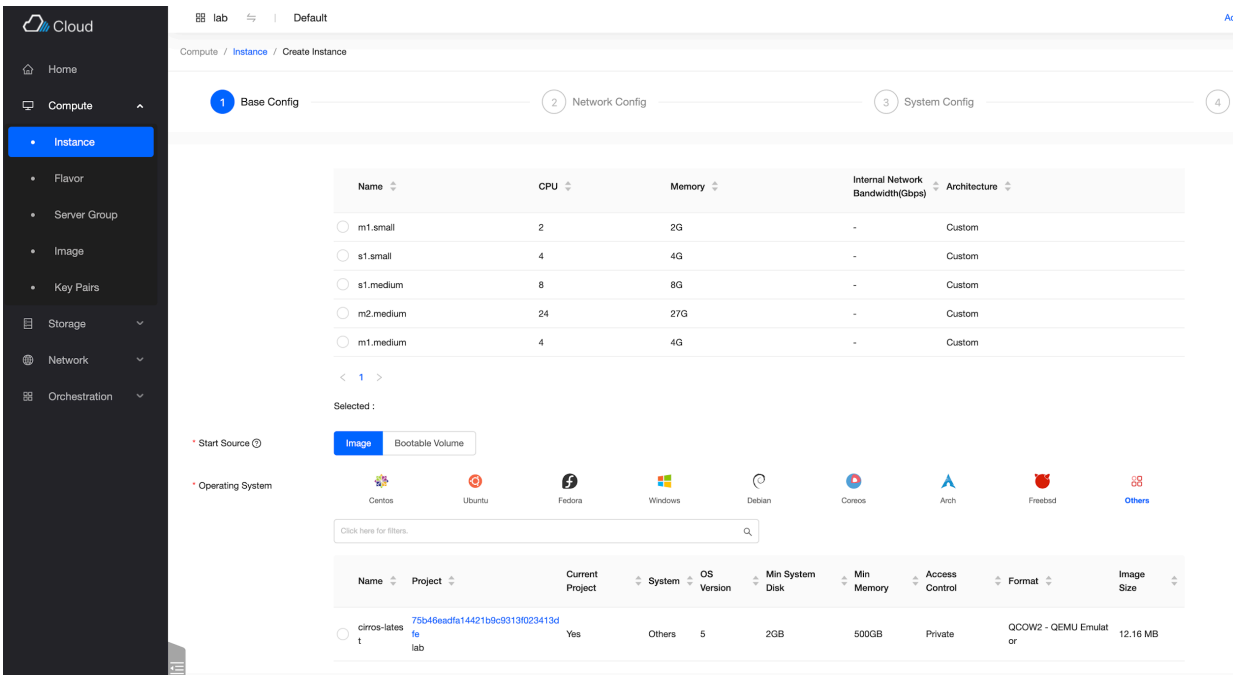


Рисунок 2.4 — Процес створення сервера у Skyline

Skyline є дійсно сучасним сервісом у всіх можливих аспектах, а конкретно у використанні популярних інструментів і загальноприйнятих підходів для веброзробки. Але все ж таки він є достатньо складним у використанні та повністю не покриває наявні функціональні вимоги до продукту.

2.3 Amazon Web Services

Amazon Web Services — це найповніша та найпоширеніша хмара у світі, що пропонує понад 200 повнофункціональних сервісів з центрів обробки даних по всьому світу [8]. AWS надає доступ до потужних обчислювальних ресурсів та різноманітних сервісів по всьому світу, дозволяючи користувачам виконувати різноманітні завдання без значних інвестицій у власні обладнання.

AWS є провайдером багатьох різноманітних сервісів, ось приклад декількох з них:

1) EC2 — обчислювальні послуги, які забезпечують масштабовані віртуальні сервери для запуску і виконання різноманітних завдань;
2) S3, EBS — служби сховища, що дозволяють зберігати та керувати даними в хмарі, забезпечуючи високий рівень надійності та доступності;
3) VPC, Route 53 — мережеві послуги з можливістю налаштування і керування мережами, DNS та іншими мережевими аспектами;
4) Athena, SageMaker — сервіси з аналітики та штучного інтелекту, які забезпечують інструменти для обробки та аналізу даних, включаючи машинне навчання;
5) AWS Lambda, CloudFormation — інструменти розробки та автоматизації розгортання інфраструктури;
6) Amazon RDS, DynamoDB — різні типи баз даних під різноманітні вимоги;
7) IAM, AWS WAF — сервіси для захисту доступу до ресурсів.

Це далеко не весь список доступних сервісів. Проте для аналізу та порівняння буде використовуватися сервіси EC2.

Amazon Elastic Compute Cloud — є сервісом, який надає широку обчислювальну платформу з понад 750 екземплярами та можливістю вибору різних процесорів, сховищ даних, мереж, операційних систем та моделей придбання для найкращого зіставлення з потребами користувачів у різному робочому навантаженню [9].

До основних функціональних особливостей належать наявність різних типів інстанцій, динамічне масштабування, різні види сховищ даних та можливість мережевого налаштування.

Плюсами даного сервісу є:

1) гнучкість у виборі конфігурацій інстанцій, операційної системи, мережевих параметрів та зберігання даних;
2) сучасний дизайн, який є адаптованим під користувача;
3) швидкість розгортання інстанцій;
4) мультирегіональність, тобто можливість створення віртуальних серверів у декількох регіонах;
5) можливість аварійного відновлення у випадку наявності аварійної ситуації у регіоні.

Проте також AWS EC2 має певні мінуси у використанні:

1) вартість EC2 може бути високою, особливо якщо немає правильної оптимізації ресурсів;
2) додаткові можливості, такі як моніторинг, резервне копіювання і забезпечення безпеки віртуального сервера, потребують окремого налаштування;
3) складне налаштовування через необхідність введення великої кількості параметрів (рис 2.5);
4) відсутність готових конфігурацій серверів аналогічно до минулих прикладів.

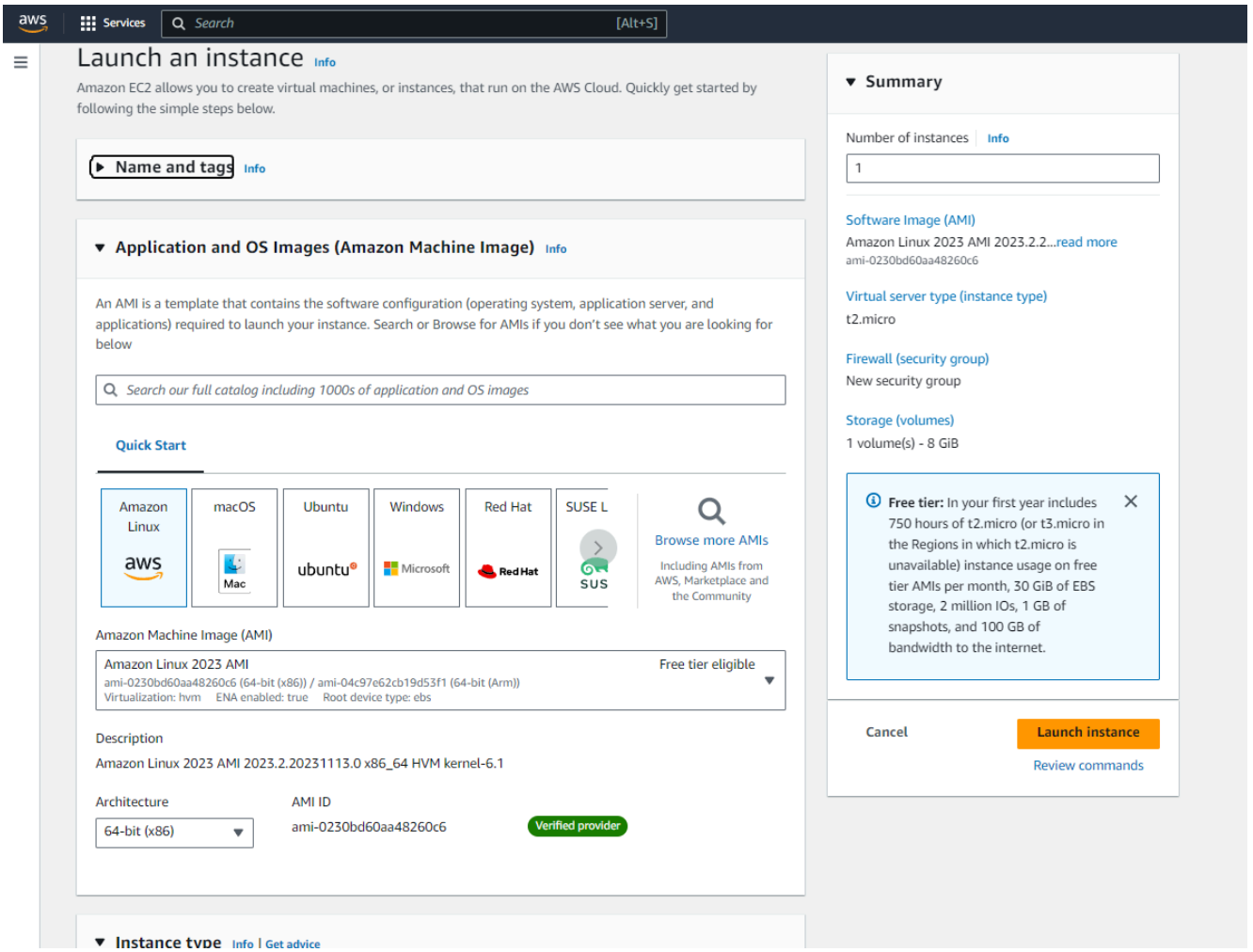


Рисунок 2.5 — Процес конфігурації сервера під час створення в AWS EC2

AWS EC2 надає користувачеві велику гнучкість у різноманітних налаштувань разом з унікальними функціональними можливостями як мультирегіональність та аварійне відновлення. Проте це й ускладнює процес взаємодії з сервісом через необхідність вивчення додаткових матеріалів по роботі з сервісом.

2.4 Digital Ocean

DigitalOcean — провайдер хмарних послуг, який став популярним завдяки своїй простоті використання та доступній ціновій політиці. Однією з ключових послуг, яку надає DigitalOcean, є Droplets — це сервіс для роботи з віртуальними

приватними серверами, який дозволяють користувачам швидко розгортати та керувати своїми інфраструктурними потребами в хмарі.

Серед функціональних особливостей виділяються наявність вибору різноманітних типів конфігурацій, легке масштабування, можливість роботи зі знімками файлової системи для швидкого відновлення або створення нових екземплярів.

До плюсів можна віднести:

1) має сучасний, інтуїтивно зрозумілий та легкий для використання інтерфейс;
2) висока швидкість розгортання серверів;
3) має оптимізовані під кінцевого користувача процеси;
4) привабливі та прозорі ціни, особливо для стартапів та невеликих проєктів;
5) високий рівень продуктивності віртуальних серверів для задач, які не вимагають великої кількості ресурсів;
6) наявність активної спільноти та інформативної документації, що сприяє легшому розв'язанню питань та проблем.

Основними мінусами є:

1) обмежений функціонал для великих або складних проєктів у порівнянні з іншими хмарними провайдерами;
2) порівнюючи з конкурентами, DigitalOcean може мати менше дата-центрів у різних регіонах світу;
3) відсутність додаткових функцій для серверів в арсеналі DigitalOcean, які пропонуються іншими провайдерами;
4) відсутність конфігурацій під важкі конкретні задачі;
5) вимагає додаткових заходів щодо дій та налаштувань для систем безпеки;
6) не повністю автоматизований процес створення серверів (рис 2.6).

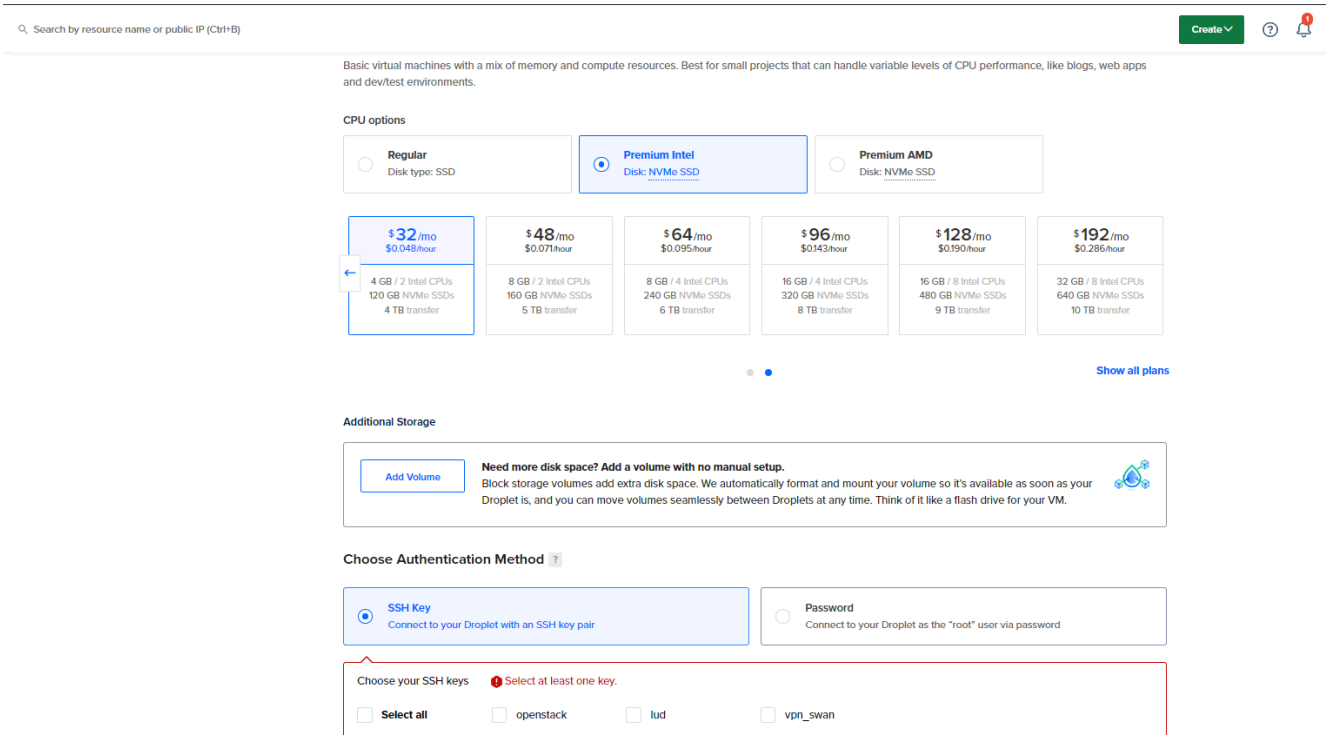


Рисунок 2.6 — Процес створення віртуального сервера у DigitalOcean

Отже, DigitalOcean дуже сильно спростив алгоритм створення віртуальних серверів для кінцевого користувача в порівнянні до описаних вище аналогів. Це вдалось досягти шляхом додавання підготовлених конфігурацій. Але все ще існують додаткові мануальні дії під час цього процесу.

2.5 Google Cloud Platform

Google Cloud Platform (GCP) — провайдер хмарних сервісів, який складається з набору фізичних ресурсів, таких як комп'ютери та жорсткі диски, і віртуальних ресурсів, таких як віртуальні машини, які містяться в центрах обробки даних по всьому світу [10].

Одним з базових сервісів GCP є Compute Engine. Compute Engine — це безпечний і настроюваний обчислювальний сервіс, який дозволяє створювати й запускати віртуальні машини на інфраструктурі Google [11].

До функціональних особливостей платформи можна віднести вибір різних типів віртуальних машин для різноманітних завдань, широкі мережеві

можливості, наявність засобів автоматизації для легкого розгортання та масштабування інфраструктури — Google Cloud Deployment Manager, мультирегіональність, групування інфраструктури по проєктах.

Плюсами є:

1) широка глобальна мережа дата-центрів, яка дозволяє користувачам обирати регіон для оптимальної продуктивності та доступності відповідно до їх потреб;

2) впровадження інноваційних технологій для швидкого виконання завдань машинного навчання;

3) широкий спектр послуг, включаючи BigQuery для аналізу даних, Kubernetes Engine для оркестрації контейнерів, Cloud Machine Learning для роботи з моделями машинного навчання;

4) можливість масштабування віртуальних машин в залежності від обсягу роботи;

5) гнучка та прозора система ціноутворення, яка може бути привабливою для користувачів з різними бюджетами.

До мінусів належать:

1) вища вартість користування в порівнянні з іншими хмарними провайдерами;

2) менша спільнота користувачів GCP, що може впливати на доступність ресурсів для розв'язання проблем або обміну досвідом;

3) наявність унікальні особливості та підходи, які потребують часу користувачів для засвоєння;

4) відсутність деяких класичних послуг, або їхні версії можуть бути менш розвиненими;

5) досить складний та не повністю інтуїтивно зрозумілий інтерфейс управління, який вимагає від користувачів складних дій для виконання простої задачі (рис 2.7).

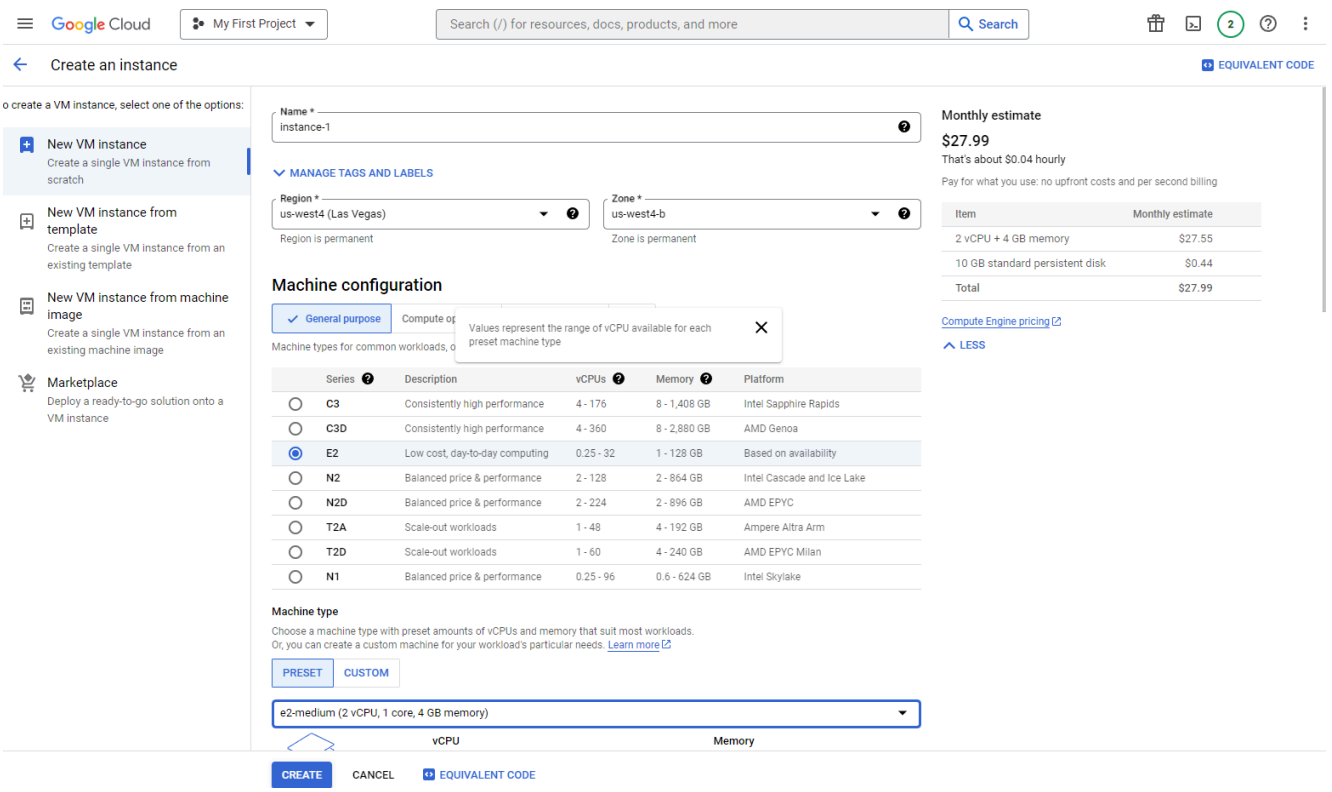


Рисунок 2.7 — Процес створення віртуального сервера у GCP

Таким чином, Google Cloud Platform пропонує користувачу ряд особливостей, які покривають функціональні потреби розроблюваного застосунку. Але ускладнений процес роботи з самою системою, досить висока ціна за використання, необхідність додаткових знань та умінь роблять GCP сильно впливають на час та якість взаємодії.

Висновки до розділу 2

У розділі були детально описані платформи, які виконують аналогічні функції з адміністрування віртуальних серверів, їх ключові особливості, плюси й мінуси. Проведений аналіз довів необхідність та можливу конкурентоспроможність розроблюваної системи, адже ні один з описаних додатків не задовольняє функціональні та нефункціональні вимоги, які були поставлені до продкту.

3 ЗАСОБИ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Важливим аспектом проєктування будь-якої системи є вибір правильних інструментів розробки — мов програмування, баз даних, протоколів комунікації тощо. Гарно підібрані інструменти мають прямий вплив на якість системи, швидкість реагування на дії користувача, швидкість і час необхідний для розробки. Саме тому цей етап має бути одним з першочергових у процесі SDLC.

Розроблювана система складається з декількох компонентів, під кожен з яких були підібрані інструменти, які відповідають сучасним підходам і практикам розробки вебсистем.

Для сервісу, який відповідає за комунікацію з хмарним середовищем OpenStack, використовується мова програмування Python, як гарний інструмент для комунікації з зовнішніми системами через API. Для спрощення написання вебсервера було використано вебфреймворк FastAPI, а для роботи з міграціями даних та початкового налаштування схеми бази — Alembic.

Фронтенд частина системи створена за допомогою мови програмування JavaScript та фреймворку Vue 3, який відповідає за реактивність та надає можливість реалізації SPA. Розмітка сторінки виконана завдяки HTML 5, а стильове оформлення за допомогою каскадних таблиць стилів CSS. Для UI-компонентів застосунку використовувався фреймворк Vuetify, а для керування станом — Pinia.

Також були застосовані інструменти, які спрощую процес безпосередньої розробки: Git та Docker в парі з Docker Compose. Git дозволяє відстежувати зміни в коді та працювати з віддаленим репозиторієм. Docker, своєю чергою, є незамінним у віртуалізації контейнерів для залежностей системи, а Docker Compose — для автоматизацій та можливості налаштування комунікацій між контейнерами та машиною.

Як сховище даних використовується об'єктно-реляційна база даних Postgres, яка показала себе як надійний та перевірений часом інструмент для збереження даних системи.

У створенні якісної та безпечної авторизації допоміг підхід з використанням JWT-токенів, який дозволяє зберігати інформацію про користувача у закодованому токені, чим і позбавляє необхідності використання додаткового сховища.

REST API виступає як основна архітектура комунікації у системі, адже вона показала себе як гарний вибір для написання вебдодатка.

Також важливим аспектом є середовища розробки, які мають безпосередній вплив на швидкість та якість написання коду. Тому на цьому етапі були підібрані інструменти компанії JetBrains: Pycharm для Python, Fleet для HTML5, CSS, JS та Vue 3.

3.1 Мова програмування Python

Python — це мова комп'ютерного програмування, яку часто використовують для створення вебсайтів і програмного забезпечення, автоматизації завдань та аналізу даних. Її зазвичай визначають як об'єктно орієнтовану мову сценаріїв — визначення, яке поєднує підтримку ООП із загальною орієнтацією на написання сценаріїв [12].

Python можна використовувати для створення різноманітних програм і вона не є спеціалізованою для вирішення якихось конкретних завдань. Ця універсальність в поєднанні з дружністю до початківців та велика спільнота зробили її однією з найпоширеніших мов програмування на сьогодні.

Особливостями цієї мови програмування є:

1) динамічна типізація — визначення типи даних під час виконання, а не під час компіляції;
2) мова є універсальним інструментом, який здатний підтримувати різні парадигми методів програмування, включаючи функціональне

програмування, процедурне програмування та об'єктно орієнтоване програмування;

3) велика стандартна бібліотека, яка надає можливість без додаткових встановлень мати доступ до файлових систем, роботи з вебзапитами та інших різноманітних завдань. Завдяки великій кількості готових інструментів, він дозволяє користувачам швидко та ефективно реалізовувати свої проєкти;

4) мова може похвалитися розширюваністю, що дає можливість інтегрувати інструменти та бібліотеки з інших мов програмування, зокрема C/C++;

5) у Python стандартно включено автоматичне керування пам'яттю та збирач сміття, які разом допомагають програмісту не думати про виділення та очищення пам'яті під змінні;

6) сумісність з різними операційними системами;

7) універсальність Python, яка дозволяє користувачам створювати вебдодатки, скрипти для автоматизації процесів, обчислювальні програми, код на рівні командного рядка, системне програмування та багато іншого.

3.2 Фреймворк FastAPI

FastAPI — це сучасний, швидкий, високопродуктивний вебфреймворк для створення API за допомогою Python 3.8 на основі стандартних підказок типів Python [13].

FastAPI дозволяє просто та оперативно написати невеликий REST API проєкт, не витративши на це багато зусиль на налаштування та імплементацію розповсюджених підходів і патернів.

Довкола FastAPI досить швидко розростається спільнота його шанувальників, мало не щодня з’являються нові бібліотеки. Тому деякі проєкти

вже поступово переходять на FastAPI з інших вебфреймворків мови програмування Python.

Серед найбільш значущих особливостей, можна виділити наступні:

1) швидкість, яка досягається завдяки Starlette і Pydantic, що робить FastAPI одним з найшвидших фреймворків Python на ринку. До того ж він може конкурувати по продуктивності навіть з мовами програмування NodeJS і Go;

2) завдяки чіткій типізації та автоматичної генерації документації, розробники можуть швидко та ефективно створювати застосунки, при цьому мінімізуючи потенційні помилки;

3) наявність інструментів, які роблять процес розробки простішим та більш інтуїтивно зрозумілим;

4) автогенерація API документації, що відповідає відкритому стандарту — OpenAPI, який раніше називався Swagger;

5) наявність вбудованих інструментів для роботи з аутентифікацією та авторизацією, включаючи підтримку OAuth2 та JWT;

6) підтримка асинхронного програмування з async і await, що дозволяє писати ефективний код для високопродуктивних операцій, особливо коли є необхідність у взаємодії з зовнішніми службами або базами даних;

7) наявність відкладених задач, які дозволяють віддати відповідь клієнту, запустивши задачі у фоні без необхідності налаштування Seleri чи подібних інструментів;

8) автоматична ін’єкція залежностей — створення та автоматичне передавання в обробнику запиту сутностей будь-яких залежностей за допомогою прописаних конструкторів.

FastAPI поєднує найкращі можливості інших вебфреймворків мови програмування Python, надаючи програмісту всі необхідні інструменти для зручної розробки.

3.3 Менеджер міграцій Alembic

Alembic — це легкий і безпечний інструмент міграції баз даних для використання з SQLAlchemy для Python [14].

Alembic надає можливість автоматизованого ведення версій схеми бази даних та керування міграціями даних. Основна ціль інструменту — дати можливість розробникам зручно відстежувати та створювати зміни в схемі бази даних, які можуть виникати під час розвитку програмного забезпечення.

До функціональних особливостей Alembic можна віднести:

1) міграції бази даних у вигляді Python-скриптів, які визначають зміни в структурі бази даних, такі як створення нових таблиць, зміни в наявних таблицях та інші модифікації;
2) автоматичне відстежування змін в моделях даних, що спрощує процес оновлення бази даних та ведення історії змін;
3) керування версіями бази даних зі збереженням історії міграцій, що дозволяє синхронізувати схеми бази даних між різними середовищами розробки;
4) підтримка різних систем управління базами даних (СУБД) — PostgreSQL, MySQL, SQLite та інші;
5) можливість повернутися на одну чи декілька міграцій назад за допомогою генерованого коду.

3.4 Мова програмування JavaScript

JavaScript — це високорівнева, інтерпретована мова програмування, відома своєю спроможністю додавати інтерактивність та динаміку вебсторінкам. JavaScript з'явився у 1995 році як спосіб додавання програм до вебсторінок у браузері Netscape Navigator. З того часу мова була прийнята всіма іншими основними графічними веббраузерами. Вона зробила можливими сучасні

вебдодатки, з якими користувачі взаємодіють безпосередньо, не перезавантажуючи сторінку для кожної дії [15].

JavaScript є мовою, яка стандартизована організацією Ecma International. Стандарт JavaScript має назву ECMAScript, і нові версії стандарту виходять регулярно. Наприклад, ECMAScript 2020 (або ES2020), випущений у 2020 році, вніс значні поліпшення та нововведення. Саме наявність цього стандарту забезпечує сумісність вебсторінкам у різних браузерах.

JavaScript має кілька потужних і цікавих особливостей, які й зробили її популярною мовою і монополістом у сфері браузерної розробки:

1) інтерпретована мова, яка виконується без попередньої компіляції;
2) можливість клієнтського та серверного програмування завдяки Node.js, що допомогло JavaScript широко розповсюдитися на стороні сервера, попри те, що спочатку мова була створена для використання в клієнтських браузерах;
3) підтримка концепції об'єктно орієнтованого програмування (ООП), але з допомогою використання прототипів для імплементації спадковості замість класичної реалізації;
4) використання зворотного виклику, обіцянок та очікування, які дозволили виконання асинхронних завдань, що особливо корисно для роботи з базами даних та мережевими операціями;
5) наявність динамічної типізації, яке дозволяє визначати типи даних під час виконання, створюючи середовище, яке сприяє гнучкості;
6) використання подієвої моделі, що дозволяє JavaScript розумно реагувати на численні взаємодії користувача та системні події, що відбуваються;
7) підтримка майже всіма сучасними веббраузерами;
8) наявність величезної кількості різноманітних бібліотек і фреймворків, більша частина яких підтримуються спільнотою.

JavaScript продовжує розвиватися, а нові версії стандарту ECMAScript та інновації у веброзробці роблять цю мову актуальною. Щобільше, зв'язка JS, HTML та CSS є найпопулярнішим вибором для створення сучасних вебдодатків.

3.5 Фреймворк Vue 3

Vue — це JavaScript-фреймворк для створення користувацьких інтерфейсів. Він базується на стандартних HTML, CSS та JavaScript і надає декларативну та компонентну модель програмування, яка допомагає ефективно розробляти користувацькі інтерфейси [16].

Фреймворк став популярним завдяки своїй простоті, гнучкості та високій продуктивності в порівнянні з іншими — Angular та React. Саме третя версія даного фреймворку відзначилась значними змінами та поліпшеннями.

Vue створювався спираючись на необхідності розробника під час створення додатка, саме тому він має наступні властивості:

1) реактивність, що дозволяє Vue реагувати на зміни в даних — одна з його найбільш визначних рис. DOM автоматично оновлюється при зміні вихідних даних, що робить реактивність помітною характеристикою Vue;
2) наявність Composition API, якою може похвалитися Vue 3, що забезпечує більшу гнучкість в організації та повторному використанні логіки компонентів;
3) ряд технічних особливостей, які призводять до значного підвищення продуктивності, включаючи оптимізований алгоритм відображення;
4) підтримка додатків різного призначення, такі як статичні сайти, рендеринг на стороні сервера та односторінкові додатки;
5) наявність модульної структури, яка пропонує легкий спосіб інтеграції лише необхідних компонентів фреймворку, що робить Vue чудовим вибором для модульних додатків;

6) легкість навчання — порівняно з іншими фреймворками, він часто відзначається своєю доступністю;

7) повна підтримка TypeScript за замовчуванням, що робить його більш популярним, враховуючи тренд на використання саме цього розширення JavaScript;

8) наявність профілю продуктивності, який надає інструменти для вимірювання продуктивності та відстеження змін в компонентах під час розробки.

Vue.js 3 продовжує зберігати свою простоту з минулих версій фреймворку, але з додаванням нових функцій та покращень, які дозволяють розробникам більш ефективно будувати високопродуктивні та масштабовані вебдодатки.

3.6 Бібліотека компонентів Vuetify

Vuetify — це колекція готових компонентів у поєднанні з потужними функціями, такими як динамічні теми, глобальні налаштування за замовчуванням, макети додатків та багато іншого [17].

Мета проєкту — надання розробникам усіх необхідних інструментів для створення багатого та гарного користувацького досвіду.

Завдяки своїм функціональним особливостям Vuetify набув неабиякої популярності серед програмістів, нижче описані декілька з них:

1) інтеграція з Material Design від Google, що забезпечує чіткий та консистентний дизайн додатків й сучасний вигляд компонентів;

2) велика кількість готових компонентів, таких як кнопки, карти, форми, таблиці й багато інших;

3) потужна система розміщення, яка дозволяє швидко розташовувати елементи на сторінці будь-яким необхідним способом;

4) можливість стилізації компонентів з використанням тем;

5) можливість зміни наявних в бібліотеці компонентів додатковими функціями чи стилями;

6) адаптивний дизайн, який дозволяє створювати інтерфейси з якісним відображенням на різних пристроях та розмірах екранів;

7) спеціалізовані компоненти, такі як календарі, діаграми, діалогові вікна і багато інших;

8) підтримка Vue версій 2 та 3;

9) Vuetify є безоплатним і випускається під ліцензією MIT;

10) велика популярність в спільноті розробників, що дозволяє отримувати швидку підтримку, а також велика кількість різноманітних ресурсів, таких як документація, форуми та блоги, для отримання релевантної інформації з використання;

11) Vuetify є проєктом з відкритим вихідним кодом, що дозволяє розробникам долучатися до його розвитку.

Загалом, Vuetify можна вважати достатньо потужною бібліотекою компонентів, яка полегшує розробку стильних та функціональних Vue.js додатків.

3.7 Фреймворк управління станом Pinia

Pinia — це бібліотека сховища для Vue, яка дозволяє ділитися станом між компонентами або сторінками. Інструмент був створений як експеримент, щоб переосмислити, як може виглядати магазин для Vue за допомогою Composition API приблизно в листопаді 2019 року. З того часу початкові принципи залишилися незмінними, але Pinia працює як для Vue 2, так і для Vue 3 і не вимагає використання Composition API [17].

Pinia розроблена з урахуванням сучасних підходів до керування станом та надає зручний декларативний спосіб управління даними в Vue.js додатках. Основна ідея полягає в покращенні роботи з управління стану, зробивши її зрозумілішою та простішою для розробників.

Pinia часто використовується у проєктах, які мають необхідність в збереженні стану застосунку, адже існують наступні функціональні особливості даного інструменту:

1) декларативний синтаксис для опису стану та методів, що робить код більш зрозумілим та легким для управління;
2) система реактивності, яка дозволяє слідкувати за змінами в стані та автоматично оновлювати компоненти при їх зміні;
3) підтримка Vue версій 2 та 3;
4) підтримка TypeScript для проєктів, де необхідна строга типізація для полегшення розробки та уникнення помилок;
5) модульна структура, яка дозволяє відокремити стан від логіки додатка на окремі модулі;
6) велика кількість плагінів для розширення стандартних можливостей Pinia;
7) підтримка Devtools для розробників;
8) підтримка гарячої заміни модулів для зберігання стану під час розробки;
9) наявність простого та легкого у використанні API, що дозволяє швидко налаштувати та використовувати керівні об'єкти в додатку;
10) можливість роботи з додатками з візуалізацією на стороні сервера — підхід SSR.

Таким чином, Pinia призначений для того, щоб полегшити розробку Vue.js додатків, зосереджуючись на простоті, декларативності та продуктивності.

3.8 Мова гіпертекстової розмітки HTML

HTML розшифровується як HyperText Markup Language (мова розмітки гіпертексту). Це стандартна мова розмітки для створення вебсторінки. Вона дозволяє створювати й структурувати розділи, абзаци та посилання за допомогою

елементів HTML, таких як теги й атрибути, які часто називають будівельними блоками вебсторінок. Мова HTML існує з 1989 року, а кілька років тому вона отримала значне оновлення у вигляді HTML5. Часто можна почути фразу, що HTML — це кістки, тобто структура, вебсторінки, а CSS — шкіра або ж зовнішній вигляд.

HTML документ складається з елементів, які називаються тегами та атрибутами тегів. Кожен тег складається з трьох елементів:

– тегу відкривання;
– контенту, те, що бачить користувач;
– тегу закриття.

Теги можуть бути вкладені один в одного, що й дозволяє будувати складні інтерфейсні композиції.

На основі цих елементів HTML будує дерево, яке часто називають об'єктною моделлю документа — DOM.

Важливою відмінністю HTML є толерантність до помилок. Коли теги, які повинні бути, відсутні, браузер реконструює їх сам. Спосіб, у який це робиться, стандартизовано, і ви можете покладатися на те, що всі сучасні браузери роблять це однаково [15].

Додатково HTML має ще декілька важливих особливостей, які також повпливали на його популярність у вебі:

1) можливість створювати гіпертекстові посилання, які вказують на інші сторінки, ресурси чи додатковий контент, за допомогою атрибуту href з прописаним шляхом або URL посиланням;
2) підтримка мультимедійних елементів, таких як зображення, аудіо та відео, що дозволяє розширити вміст сторінки різноманітними мультимедійними ресурсами;
3) можливість взаємодії з користувачами шляхом вводу даних у різноманітні форми, списки та поля;

4) метатеги, які існують для надання додаткової інформації про документ, такої як метаопис, кодування символів, властивості зображення та інші. Вони використовують пошуковими системами для індексування сторінок сайтів та браузерами для коректного відображення сторінок;

5) простота та легкість вивчення, що робить його ідеальним для початківців, адже синтаксис є інтуїтивно зрозумілим, а вивчення специфікації не вимагає значних зусиль.

HTML є фундаментально важливим інструментом у побудові вебсайтів. Щобільше, багато описаних вище інструментів в кінцевому етапі інтерпретуються в чистий HTML, адже він є монополістом у сфері розмітки вебсторінок.

3.9 Таблиця каскадних стилів CSS

CSS — каскадні таблиці стилів — мають велике значення для дизайну та верстки вебконтенту. CSS розширює можливості HTML, дозволяючи розробникам створювати естетично привабливий та гарний дизайн для вебсайтів. Його основна ціль — це розділити вміст і представлення. Таким чином, веброзробники можуть задати стилі, таким компонентам, як текст, зображення та посилання, відокремлюються від фактичного контенту. Таке розділення не лише полегшує розробку та підтримку вебсайту, але й покращує відображення на різних пристроях.

Загалом таблиця стилів — це набір правил для стилізації елементів у документі. Каскадність у назві означає, що декілька таких правил комбінуються для створення остаточного стилю елемента [15].

Правила можуть бути задані всередині тегу «<style>» або ж в будь-якому HTML тегу в атрибуті style або ж в окремому файлі з розширенням .css, який потім треба імпортувати в HTML розмітку.

Стилі можна описати для будь-якого тегу HTML з можливістю фільтрації по атрибутах, класах чи ідентифікаторах. Також можна використовувати складніші

селектори для опису стилів, наприклад «тег з конкретним класом, який знаходиться в документі після іншого тегу».

Ось декілька ключових особливостей CSS, які повпливали на його розповсюдженість:

1) каскадність та спадкоємність, які дозволяю визначати пріоритетність стилів в залежності від їх специфічності та можливість успадкування від батьківських елементів наступних рівнів вкладеності відповідно;

2) абстрактне розділення усіх HTML елементів на блокові, які займають всю ширину сторінки та починаються з нового рядка, та інлайнові, які займають лише необхідне місце без перенесення на новий рядок;

3) можливості позиціювання та вирівнювання елементів на розмітці сторінки;

4) адаптивний дизайн для забезпечення оптимального відображення вебсайт на різних пристроях та екранах;

5) можливість створювати анімації та переходи елементів для поліпшення користувацького досвіду;

6) модульність та перевикористання, що досягається шляхом декомпозиції одного файлу на декілька менших за розміром для забезпечення легкості в розумінні написаного;

7) підтримка багатьох різних стандартизованих видів розміток, таких як Grid та Flex, які не лише спрощують процес створення адаптивного дизайну, а й покращують та уніфікують розмітку в цілому;

8) спрощена робота з типографікою та кольористикою в порівнянні з імплементаціями за допомогою HTML.

Отже, CSS є важливим і розповсюдженим інструментом у веброзробці, який покращує UI компонент в парі UI/UX, роблячи інтерфейс більш приємним і зручним для кінцевого користувача.

3.10 Git

Git — це безплатна розподілена система контролю версій з відкритим вихідним кодом, призначена для швидкої та ефективної роботи з будь-якими проєктами, від невеликих до дуже великих [19]. Інструмент створений Лінусом Торвальдсом, розробником ядра сімейства операційних систем Linux, у 2005 році. Git є важливим інструментом для управління версіями кодової бази, забезпечуючи ефективну та спільну роботу над проєктами.

Контроль версій, також відомий як контроль вихідного коду, — це практика відстеження та управління змінами в програмному коді.

Система контролю версій — це програмний інструмент, який допомагає командам розробників керувати змінами у вихідному коді.

Враховуючи той факт, що прискорилися зміна кодової бази системи та процес розробки загалом, системи контролю версій допомагають командам розробників працювати швидше і якісніше. Ці системи є особливо корисними для команд DevOps, оскільки допомагають їм скоротити час налаштувань та збільшити кількість успішних розгортань [20].

Git простий у вивченні, займає мало місця та має блискавичну продуктивність. Він перевершує такі SCM-інструменти, як Subversion, CVS, Perforce та ClearCase завдяки своїм унікальним функціям [19].

Функції, які посприяли становленню Git, як найбільш успішного інструмента контролю версії:

1) розподілена природа, при якій кожен розробник має повну копію репозиторію на своєму локальному комп'ютері, що дозволяє працювати незалежно та навіть без доступу до мережі;
2) гнучкістю та швидкістю у використанні навіть при роботі з великими репозиторіями;
3) збереження повної історії змін проєкту для перегляду, відстежування та відновлювання попередніх версій коду;

4) підтримка віддалених репозиторіїв для можливості спільної праці над проєктами в режимі реального часу;

5) система гілкування, тобто створення відокремлених лінії розробки, що можуть існувати паралельно і незалежно від головної версії-гілки;

6) контрольований процес індексації, що сприяє вибору змін, які будуть включені до наступних версій.

3.11 Docker та Docker Compose

Docker — це програмна платформа, яка дозволяє швидко створювати, тестувати та розгортати додатки.

Docker пакує програмне забезпечення у стандартизовані одиниці, які називаються контейнерами, що містять все необхідне для запуску, включаючи бібліотеки, системні інструменти, код та середовище виконання [21].

Використовуючи Docker, розробники можуть забезпечити консистентність у роботі програмного забезпечення на різних середовищах, від локального комп'ютера до віртуального сервера у хмарній інфраструктурі.

Зазвичай Docker використовується для підняття та налаштування роботи зі сторонніми залежностями — сховищами даних, інструментами розробки та тестування. Docker може надавати широкий функціонал, залишаючись досить легким та швидким інструментом. Саме тому він прийшов на заміну віртуальним операційним системам, які раніше використовувалися для цих цілей.

Docker складається з трьох основних структурних елементів: image, container та engine. Image — це попередньо зібраний і налаштований програмний пакет, який містить усі необхідні залежності та інструкції для запуску певної програми чи сервісу. Container — це екземпляр пакета Image, створений під час виконання, з власним унікальним набором ресурсів і файловою системою. Engine — це програмне забезпечення, яке забезпечує створення і керування Image та Container.

Наступні особливості посприяли поширенню інструмента серед розробників:

1) концепція контейнеризації, при якій програмне забезпечення та його залежності упаковуються разом в ізольовані контейнери зі спільною операційною системою ядра;
2) портативність і можливість виконання на будь-якому комп'ютері;
3) спрощена система масштабування додатків, яка дозволяє запускати багато контейнерів на одній машині;
4) швидкість і легкість запуску, при якій контейнери можуть бути запущені у лічені секунди;
5) ізольованість контейнерів один від одного;
6) високий рівень безпеки, оскільки контейнер не може впливати на інші контейнери або основну систему;
7) ідеальне зіставлення з мікросервісною архітектурою, яка є однією з найпопулярніших у наш час;
8) широкий інтерфейс для управління контейнерами.

Docker Compose — це інструмент для визначення та запуску багатоконтейнерних Docker-додатків [22]. Він дозволяє описувати налаштування та конфігурації декількох контейнерів та можливого шляху їх взаємодії за допомогою YAML-файлів.

В той час як Docker Compose використовується для одночасного керування кількома контейнерами, що входять до складу програми.

3.12 База даних PostgreSQL

PostgreSQL — це потужна об'єктно-реляційна система баз даних з відкритим вихідним кодом, яка вже понад 35 років активно розвивається і завоювала міцну репутацію завдяки своїй надійності, функціональності та продуктивності [23].

PostgreSQL знаходить широке застосування у сфері веброзробки, бізнес-застосунків та галузей, де вимагається надійне та масштабоване зберігання даних. Ця СУБД привертає користувачів завдяки своїй ефективності, зручності в роботі та широкому спектру функціональностей.

PostgreSQL визначається високою ступенем сумісності зі стандартами SQL. Щобільше, ця СУБД надає можливість використовувати багато розширених типів даних і підтримує рівень оптимізації продуктивності, характерний для комерційних СУБД, таких як Oracle чи SQL Server.

PostgreSQL надає декілька важливих переваг в порівнянні з іншими СУБД:

1) об'єктно-реляційна модель даних, що дозволяє використовувати як реляційні, так і об'єктні конструкції. Це також охоплює можливість створення власних типів даних, функцій та операторів;

2) має вбудовану підтримку географічних об'єктів та географічних індексів, що ідеально підходить для роботи з вебдодатками чи геоінформаційними системами;

3) надає можливість реплікації, що надає безпеки високопродуктивним системам і не тільки;

4) можливість додавання розширень та власних функцій;

5) підтримка транзакцій в операціях;

6) широкий вибір різних видів індексів, включаючи як і звичайні B-tree індекси, так і специфічні — текстовий пошук;

7) підтримка JSON та JSONB (бінарний JSON) та різних видів операцій з ними;

8) підтримка зберігання та використання масивів, як окремого типу даних;

9) висока масштабованість, яка стосується не тільки об'єму даних, якими може керувати СУБД, а й кількості користувачів, які одночасно можуть працювати в ній.

PostgreSQL є важливим інструментом для будь-якого вебдодатка, який потребує наявність масштабованого, надійного, типізованого й безоплатного сховища даних.

3.13 JWT

JWT — вебтокени JSON — це відкритий галузевий стандарт RFC-7519 для безпечного обміну інформацією між двома сторонами [24].

В основному даний формат токенів використовується для автентифікації та передачі інформації про користувача між сторонами в безпечний спосіб. Інформація є в токені є правдивою, оскільки вона підписана цифровим підписом. JWT можуть бути підписані за допомогою секретного або відкритого/закритого ключів з використанням RSA або ECDSA.

Токен складається з трьох частин:

– заголовок;

– контент;

– підпис.

Заголовок містить системну інформацію і зазвичай складається з двох частин: типу токена, яким є JWT, і алгоритму підпису, наприклад, HMAC SHA256. Дані заголовки кодуються за допомогою алгоритму Base64.

Контент містить твердження про об'єкт, зазвичай про користувача, і додаткові дані. Існує три типи тверджень: зареєстровані, публічні та приватні. Контентна частина також кодуються з використанням алгоритму Base64.

Для створення підпису використовують закодовані заголовок та контент, а також секретний ключ для підпису. Алгоритм підпису аналогічний алгоритму в заголовках.

Усі три частини об’єднуються через крапку для генерації кінцевого токена.

Основні переваги JWT в порівнянні з іншими токенами для авторизації

включають:

1) незалежність від сервера, що досягається шляхом підписання секретним ключем на стороні сервера;
2) малий розмір та компактний формат, який дозволяє легко передавати та обробляти токени;
3) гнучкість та розширюваність через можливість додавати користувацьку інформацію в контентну частину токена;
4) цілісність даних через підписання сервером.

Існують й інші формати токенів, такі як OAuth-токени та SAML, які можуть використовуватись для авторизації при різних обставинах. Однак JWT стає популярним вибором завдяки своїм перевагам у простоті, безпеці та легкості використання.

3.14 REST API

REST — це програмна архітектура, яка накладає умови на те, як повинен працювати API. REST спочатку був створений як архітектурний стиль для управління зв'язком у складних системах, таких як мережа Інтернет. Архітектуру на основі REST використовується для підтримки високопродуктивного та надійного зв'язку в великих масштабах. API, які відповідають архітектурному стилю REST, називаються REST API або ж RESTful API [25].

Основною ідеєю архітектури є використання HTTP-протоколу для передачі та обміну даних між клієнтом і сервером.

Особливостями та перевагами архітектури REST, які виділяють його з поміж інших стилів є:

1) простота та легкість у розробці через використання стандартних HTTP-методів для взаємодії між компонентами системи;
2) незалежність від мови програмування та платформи використання;

3) уніфікований інтерфейс, тобто компоненти передають один одному інформацію в стандартизованому форматі;

4) уніфікація помилок через використання HTTP статус кодів;

5) незалежність від стану компонента, тобто сервер обробляє кожен запит клієнта незалежно від усіх його попередніх запитів;

6) можливість кешування ресурсів на стороні клієнта або сервера для покращення швидкості передачі даних та зниженню навантаження на мережу;

7) багаторівнева архітектура системи, при якій клієнт може підключатися до інших авторизованих посередників між клієнтом і сервером, і він все одно буде отримувати відповіді від початкового сервера;

8) розширення та додаткове налаштування клієнтської функціональності, передаючи програмний код клієнту.

REST API активно використовується у веброзробці, мобільних додатках та інших сценаріях, де простота та стандартизація є ключовими факторами.

Висновки до розділу 3

Даний розділ описує основні інструменти та засоби розробки, які були використані під час створення системи. Усі технології обирались спираючись на функціональні потреби застосунку та їх популярність на ринку. Щобільше, усі інструменти є актуальними та перевіреними часом й спільнотою.

4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

Для розроблюваної системи необхідно було створити якісний та швидкий застосунок з інтуїтивно зрозумілим і простим для кінцевого користувача інтерфейсом. Саме тому в процесі розробки були використані найкращі та найсучасніші практики вебсистеми — оптимізація продуктивності, безпека, адаптивний дизайн, SPA та інтеграція з API. Усі вони мали неабиякий вплив під час проєктування архітектури системи, яка дозволяє швидко обробляти запити, реагувати на дії користувача, забезпечує гарну швидкість роботи та ефективне споживання ресурсів машин.

4.1 Загальна архітектура системи

Процес проєктування архітектури системи є значущим етапом, адже саме правильна архітектура дозволить застосунку бути стійким до великого навантаження, швидким, безпечним та дієздатним. Щобільше, на великих масштабах правильна архітектура сприяє працездатності системи в цілому.

Існує багато різних можливих архітектур систем, починаючи від монолітної та закінчуючи подієво-орієнтованою. Кожна з них підходить під конкретні задачі. Перед розробкою розглядалися та порівнювалися лише два типи: монолітна та мікросервісна.

Монолітна архітектура передбачає, що весь додаток має бути побудований як єдиний, автономний блок [26]. Усі компоненти та функціональні особливості розташовані в межах одного програмного модуля, є тісно пов'язаними між собою і розгортаються разом. Даний тип архітектури є відносно простим і підходить для невеликих додатків, але в залежності від реалізації його може бути складно підтримувати й масштабувати в міру зростання системи.

Архітектура мікросервісів є розширенням концепції SOA, де система поділяється на набір невеликих, незалежних сервісів [26]. Кожен сервіс відповідає за конкретну функціональність та може бути розроблений, тестований, та розгорнутий незалежно від інших. Щобільше, кожен з них представляє певну бізнес-можливість і взаємодіє з іншими сервісами за допомогою легких протоколів, таких як HTTP або RPC. Мікросервіси забезпечують незалежну масштабованість, простоту розгортання та ізоляцію несправностей.

В таблиці 4.1 наведено порівняння двох видів архітектур, яке було використане під час вибору для системи.

Таблиця 4.1 — Порівняння монолітної та мікросервісної архітектур

Характеристика	Монолітна архітектура	Мікросервісна архітектура
1	2	3
Розмір та складність проєкту	Єдина кодова база	Невеликі, незалежні мікросервіси
Масштабованість	Складно масштабувати великі проєкти, особливо вертикально	Легше масштабувати, можливість горизонтального та вертикального масштабування
Гнучкість розробки	Може бути менш гнучкою при збільшенні кодової бази та великому розміру команди	Більш гнучка, оскільки окремі сервіси можуть розроблятися та оновлюватися незалежно
Залежність технологій	Одна технологія та версія для всього додатку	Різні технології та версії можуть використовуватися для різних мікросервісів
Тестування	Легкий процес тестування, оскільки весь додаток об'єднаний	Більш ускладнений процес тестування, через необхідність тестувати взаємодію між сервісами

Продовження таблиці 4.1

1	2	3
Витрати на розгортання та інфраструктуру	Менше витрат, оскільки всі компоненти розгортуються як єдиний блок	Більше витрат, оскільки потрібні ресурси для кожного окремого сервісу
Складність управління	Менш складне управління, оскільки всі компоненти об'єднані	Складніше управління, особливо при великій кількості мікросервісів
Проблеми мережі	Менше мережевих проблем, оскільки всі компоненти на одному рівні	Більша кількість проблем з мережею через необхідність постійної взаємодії між сервісами
Цілісність даних	Сховище даних зазвичай централізоване, що дозволяє використовувати транзакції	Управління цілісністю даних ускладнено через розділену природу системи

Спираючись на проведене порівняння двох архітектур було вирішено використовувати розширення стандартної модульної архітектури, а саме модульний моноліт. Модульно-монолітна архітектура поєднує переваги модульного дизайну з простотою монолітної архітектури. Вона передбачає поділ системи на набір слабко пов'язаних між собою модулів, кожен з яких має чітко визначені межі та явні залежності від інших модулів.

Модульно-монолітна архітектура розділяє логіку додатка на модулі, і кожен модуль є незалежним, ізольованим, має власну бізнес-логіку та власну взаємодію з API.

Система складається з декількох компонентів: клієнт, сервер, СУБД Postgres та хмарне середовище OpenStack. Таким чином на рівні конкретного компонента реалізується модульно-монолітна архітектура, а на системному рівні — один з принципів SOLID, а саме Single Responsibility Principle. Тобто кожен з

компонентів має власну зону відповідальності й не залежить від інших, що дозволяє в майбутньому розширювати систему додатковими компонентами або замінювати їх на інші. Системну архітектуру системи показано на рисунку 4.1.

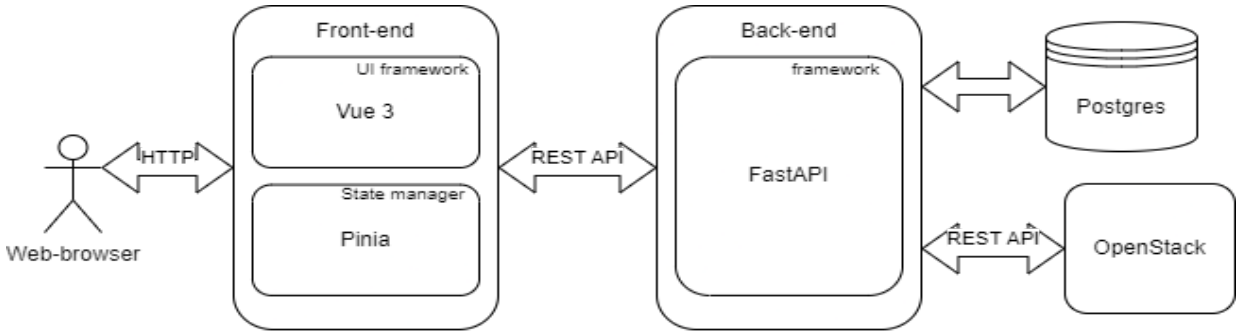


Рисунок 4.1 — Архітектура системи

Як видно з рисунка 4.1 комунікація між браузером та фронтендом відбувається шляхом використання протоколу HTTP — протокол передачі даних, що використовується в комп'ютерних мережах [27]. А внутрішня комунікація між компонентами заснована на використанні підходу архітектури мережевих протоколів REST API. Це набір правил, які визначають, як програми або пристрої можуть підключатися та взаємодіяти один з одним [28].

Відповідно до архітектури модульного моноліту серверну частину було розділено на модулі:

1) модуль авторизації;
2) модуль управління серверами;
3) модуль роботи з API OpenStack.

Разом з тим, клієнтська частина також була побудована з врахуванням підібраної архітектури й складається з таких модулів:

1) модуль авторизація;
2) модуль звітів;
3) модуль інструкцій;
4) модуль управління серверами;
5) робота з серверним API.

Вибір мов програмування ґрунтувався на ряді факторів, включаючи

специфіку проєкту, наявність відповідних бібліотек та інструментів, вимоги до продуктивності та ресурсів, досвід роботи з певною мовою, а також переваги з точки зору вартості, підтримки та масштабованості.

4.2 Модель бази даних

На рисунку 4.2 можна побачити логічну модель бази даних. В системі наявна невелика кількість таблиць з описом основних доменних моделей, інші дані зберігаються на самому OpenStack відповідно. Загалом існують 4 таблиці: user, server_config, server та alembic_version.

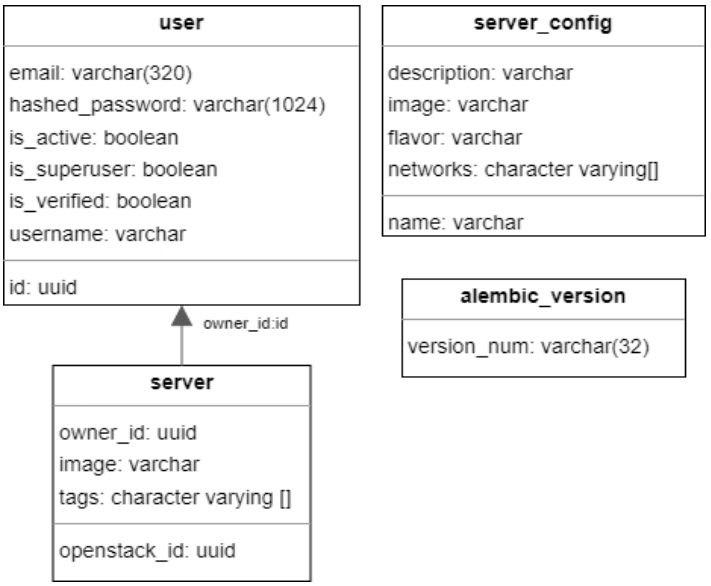


Рисунок 4.2 — Логічна модель бази даних системи

Таблиця user описує користувача системи й складається з 7 полів: id — ідентифікатор користувача, email — поштова адреса, hashed_password — хеш пароля, is_active — прапорець активності користувача, is_superuser — прапорець адміністратора, is_verified — прапорець верифікації, username — логін.

Таблиця server має зв’язок «багато до одного» з таблицею user та складається з декількох полів: owner_id — ідентифікатор користувача, image — образ, який використовувався для створення сервера, openstack_id —

ідентифікатор сервера в хмарному середовищі OpenStack та tags — список станів віртуального сервера та додатків, які на ньому завантажені.

Щодо таблиці server_config, то вона зберігає дані про можливі спеціалізовані конфігурації серверів у системі. Вона має наступні поля: name — назва конфігурації, description — опис, image — назва образу, flavor — назва групи характеристик віртуального обладнання та networks — список назв мереж, які будуть підключені до сервера.

Остання таблиця alembic_version є системною та відповідає за міграції бази даних.

4.3 Структура серверної частини

На рисунку 4.3 зображена структура REST API сервісу, який відповідає за створення та управління віртуальними серверами на основі підготовлених конфігурацій. Тобто саме тут описана основна бізнес-логіка системи, процес взаємодії з API OpenStack та з базою даних.

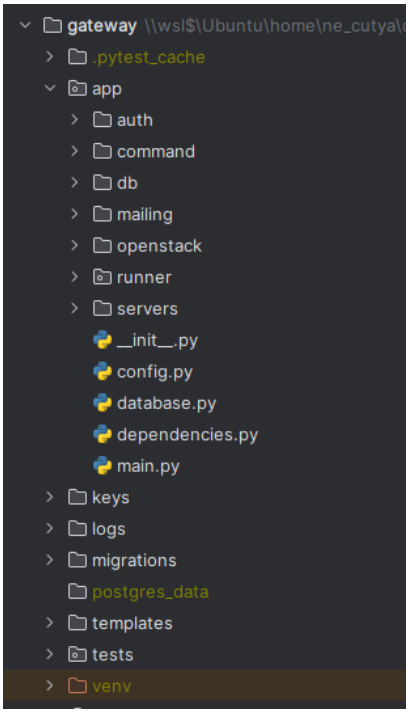


Рисунок 4.3 — Структура REST API сервісу

Основною текою у проєкті на основі фреймворку FastAPI є app, де знаходяться модулі. Вони розподіленні відповідно до своїх функціональних особливостей:

1) auth — авторизація та автентифікації користувача;
2) command — опис команд для запуску на серверах;
3) db — взаємодія з базою даних;
4) mailing — робота з поштовим клієнтом;
5) openstack — взаємодія з OpenStack API;
6) runner — запуск команд на серверах;
7) servers — управління серверами.

Також є додаткові теки для роботи з шаблонами, журналом запису дій та помилок системи, міграціями бази даних та встановленими зовнішніми модулями для Python.

До того ж під час розробки сервісу використовувалися наступні шаблони проєктування:

1) ін’єкція залежностей (dependency injection) — це стиль налаштування об'єкта, коли поля об'єкта задаються зовнішньою сутністю;
2) одинак (singleton pattern) — це патерн, при якому клас може гарантувати, що жоден інший екземпляр не може бути створений, перехоплюючи запити на створення нових об'єктів, а також надає спосіб доступу до екземпляра [29];
3) декоратор (decorator pattern) — це структурний патерн проєктування, яки дозволяє динамічно додавати об’єктам класу нову функціональність, загортаючи їх в обгортки;
4) фабрика (factory pattern) — це патерн, який надає інтерфейс для створення сімейств пов'язаних або залежних об'єктів без вказівки їх конкретних класів [29];
5) фасад (facade pattern) — структурний патерн проєктування, який надає простий інтерфейс до складної підсистеми.

Застосування даних патернів дозволило не лише створити гнучкий застосунок, який з легкістю піддається змінам та модульному й інтеграційному тестуванням, а й спростити процес розробки системи.

4.4 API серверної частини

Серверна частина реалізує REST API для взаємодії з системою з клієнтської сторони.

Рисунок 4.4 демонструє згенеровану за допомогою фреймворку FastAPI документацію OpenAPI.

API надає можливість авторизації у систему, управління обліковим записом та повний список можливих дій з віртуальними серверами, починаючи від створення та закінчуючи можливістю підключення до сервера через вебконсоль.

Рисунок 4.4 — OpenAPI документація з описом API

У всіх запитах використовується формат даних JSON для взаємодії, а також HTTP-заголовок Authorization для передачі JWT-токена. Це здійснюється з метою забезпечення безпеки та ідентифікації користувачів у процесі обміну інформацією, що передається між клієнтами та сервером.

4.5 Структура клієнтської частини

Структура клієнтського компонента зображена на рисунку 4.5.

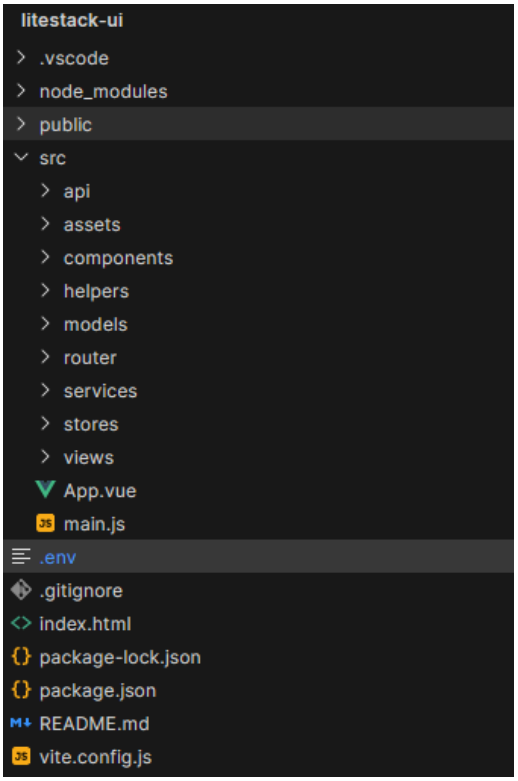


Рисунок 4.5 — Структура frontend сервісу

Тека src є головною та відповідає за усю бізнес-логіку клієнтської системи, node_modules описує встановлені модулі у системі, а public містить опис та стилів, зображень і інших публічних файлів.

Головна тека складає з наступних функціональних модулів:

1) api — робота з API серверної частини;
2) assets — статичні файли, логотипи за зображення;
3) components — спільні компоненти для сторінок;

4) helpers — обгортки для простішої взаємодії з сервером;

5) models — моделі системи;

6) router — опис відповідностей сторінок до їх URI-шляхів;

7) services — сервіси з бізнес-логікою;

8) stores — модуль для роботи зі збереженням стану компонентів;

9) views — вебсторінки.

Також використовувалися додаткові інструменти, які розширювали стандартні можливості фреймворку Vue та спрощували процес розробки. Такими інструментами є:

1) vue-router — відповідає маршрутизацію користувача у системі;

2) vee-validate та yup — бібліотеки, які допомагають з валідацією форм;

3) vue-chartjs — бібліотека для відображення графіків та діаграм, яка заснована на бібліотеці chartjs;

4) vuetify — фреймворк для UI-компонентів, який надає семантичні та багаторазові компоненти для спрощення процесу створення додатка;

5) pinia — бібліотека управління станом екосистеми Vue, основна ціль якої допомогти в керуванні реактивними даними та станом компонентів програми.

4.6 Підбір конфігурацій віртуальних серверів

Для реалізації необхідного функціоналу у системі важливо підібрати правильні конфігурації серверів, адже вони мають пряму залежність на ефективність та продуктивність віртуальних серверів, вартість підтримки кластера та ліміти хмарного середовища.

Під різні конкретні вимоги та потреби користувачів було створено чотири конфігурації. Вони дозволяють користувачам обирати оптимальний баланс між продуктивністю, вартістю та ресурсами залежно від їхніх конкретних потреб та

завдань. Конфігурації мають назви «Базова», «Легка», «Покращена» та «Професійна» та детально описані таблиці 4.2.

Таблиця 4.2 — Конфігурації серверів

Назва	Image	Flavor	Volumes	Networks
1	2	3	4	5
Базова	Fedora-Cloud-Base-37-1.7.x86_64	m1.small	50GB SSD	Private, Public
Легка	Fedora-Cloud-Base-37-1.7.x86_64	m1.medium	100GB SSD + 1TB HDD	Private, Public
Покращена	Fedora-Cloud-Base-37-1.7.x86_64	m1.large	250GB SSD + 2TB HDD	Private, Public
Професійна	Fedora-Cloud-Base-37-1.7.x86_64	m1.xlarge	500GB SSD + 4TB HDD	Private, Public

4.6.1 Опис параметрів конфігурацій

Кожна конфігурація складається з 4 основотворчих параметрів — це image, flavor, volumes, networks.

Колонка Image відповідає за образ операційної системи або програмного забезпечення, який буде встановлено на віртуальну машину. Образ — це файл, який містить завантажувальну операційну систему та можливі завчасно встановлені додатки з усіма необхідними залежностями. В нашому випадку, це Fedora-Cloud-Base-37-1.7.x86_64.

Flavor — це умовне позначення, яке використовується хмарним середовищем OpenStack для опису конфігурація ресурсів віртуальної машини.

Вона визначає кількість CPU, RAM та інші характеристики. У таблиці 4.3 описані характеристики відповідно до наведених вище назв.

Таблиця 4.3 — Конфігурації ресурсів віртуальної машини

Назва	Віртуальні CPU	Розмір диска (у Гб)	Розмір RAM (у Мб)
1	2	3	4
m1.small	1	20	2048
m1.medium	2	40	4096
m1.large	4	80	8192
m1.xlarge	8	160	16384

Volumes описує тип (SSD чи HDD) та об'єм віртуального диска для зберігання даних, а Networks — мережеві інтерфейси, до яких підключена віртуальна машина. Мережеві інтерфейси можуть бути приватними (Private) або публічними (Public).

4.6.2 Опис конфігурацій

Кожна з конфігурацій створена з урахуванням усіх можливих потреб та вимог користувачів у створенні віртуальних серверів під різні типи завдань та з певними обчислювальними здібностями. Опишемо кожну з них для розуміння випадків використання.

Базова конфігурація призначена для початкового рівня користувачів, які тільки опановують роботу зі складними процесами. Використовується m1.small для економії ресурсів, але з достатнім обсягом SSD для швидкості. Підходить для початківців, які мають необхідність у легких застосунках.

Легка конфігурація має збільшений об'єм диска та ресурсів, що дозволяє швидше та ефективніше обробляти більше даних. Є гарним варіантом для людей, які тільки вивчають обробку великих даних чи штучний інтелект.

Покращена версія, своєю чергою, ідеально підходить для великих наборів даних та важких завдань. Вон має значний об'єм диска та велику кількість обчислювальних ресурсів, що дозволяє якісно та швидко обробляти достатньо велику кількість даних.

Професійна — найбільш потужна з наявних конфігурацій для найбільших завдань з даними. Є оптимальним вибором для великих команд даних чи проєктів, які працюють з дійсно великими об'ємами даних.

4.7 Налаштування хмарного середовища OpenStack

Окрім стандартного процесу встановлення хмарного середовища OpenStack на кластер є необхідність додаткових налаштувань, які впливають на інтеграцію з системою та роботу з віртуальними серверами.

4.7.1 Створення нового проєкту

У першу чергу, необхідно створити новий проєкт, який буде інтегруватися з системою. Для створення проєктів чи облікових записів нових користувачів необхідно бути адміністратором OpenStack та використати для цього OpenStack Dasboard (рис. 4.6) або OpenStack CLI.

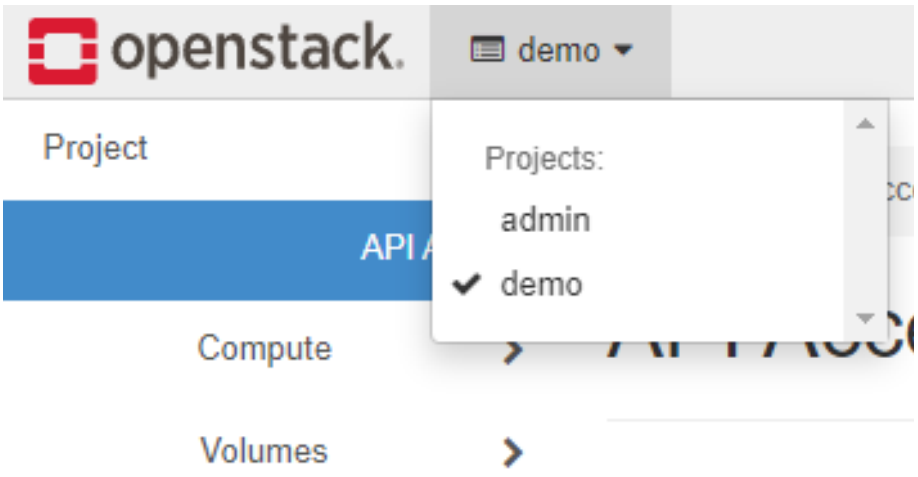


Рисунок 4.6 — Демонстрація створеного проєкту

Також на рисунку 4.6 зображений новостворений проєкт — demo, проєкт admin є стандартним в OpenStack.

4.7.2 Налаштування топології мережі

Наступним важливим кроком є створення топології мережі, яка б дозволяла віртуальним серверам мати доступ до зовнішнього світу. На рисунку 4.7 зображена топологія, яка складається з декількох основних компонентів:

1) маршрутизатор, який надає прохід між внутрішніми та зовнішніми мережами. Маршрутизатор може мати як IPv4, так і IPv6 адреси;

2) публічна мережа, яка підключена до Інтернету. Це дозволяє віртуальним машинам в OpenStack отримувати доступ до зовнішнього світу. Адреси в цій мережі можуть бути як IPv4, так і IPv6;

3) спільна мережа, до якої можуть підключатися різні користувачі або проєкти в OpenStack;

4) приватна мережа, яка призначена для комунікації між віртуальними машинами в межах одного проєкту або користувача. Ця мережа ізольована від інших мереж, що забезпечує безпеку та приватність.

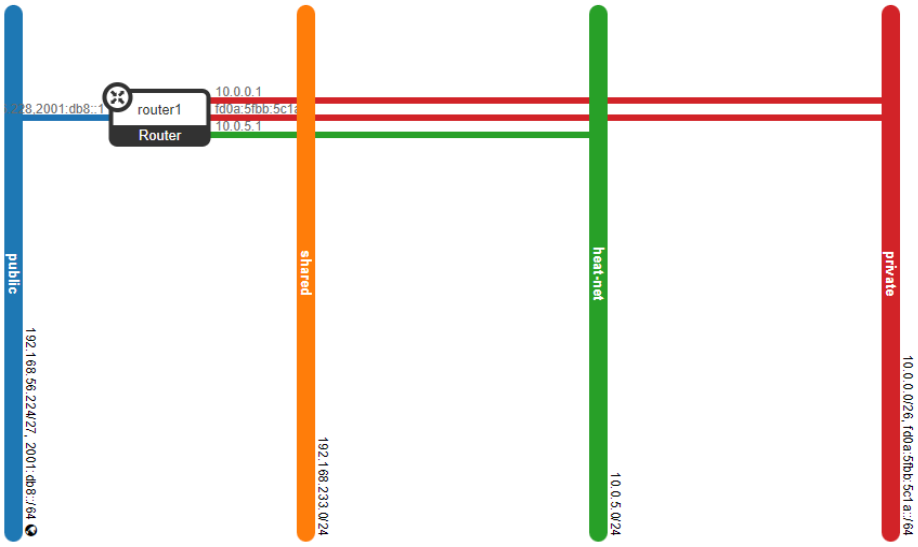


Рисунок 4.7 — Топологія мережі

Описана топологія дозволяє гнучко керувати мережевими ресурсами в OpenStack, забезпечуючи ізоляцію, безпеку і масштабованість для розгортання різних додатків і сервісів.

4.7.3 Додавання правил до групи безпеки

Також обов'язково треба додати правила до груп безпеки, які дозволяють під'єднуватися до віртуальних серверів по протоколу SSH та перевіряти їх робочий стан за допомогою утиліти ping.

Рисунок 4.8 демонструє наявні правила груп безпеки для новоствореного проєкту.

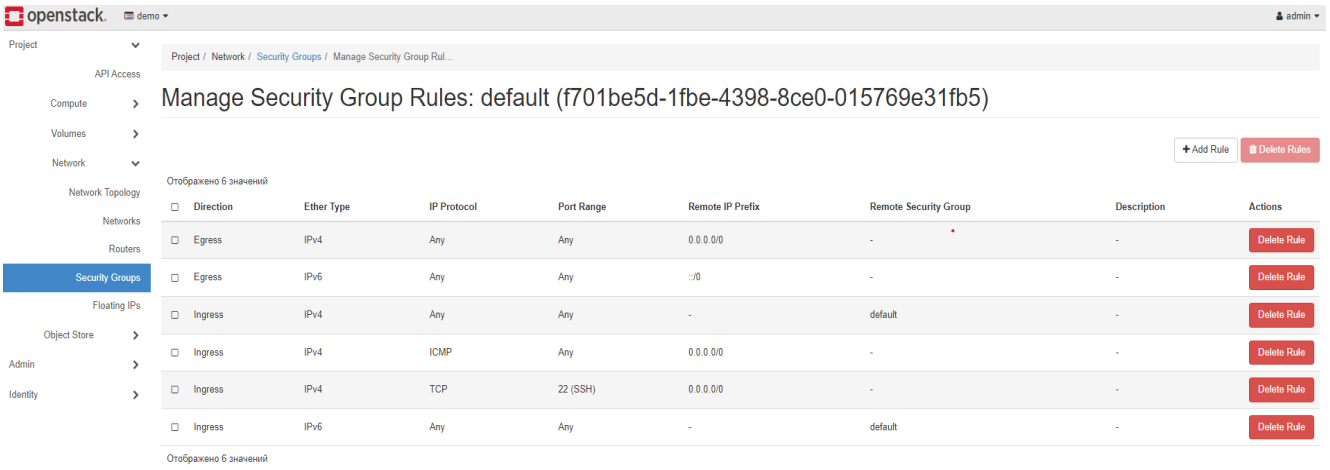


Рисунок 4.8 — Правила групи безпеки

Група безпеки — це набори правил IP-фільтрів, які застосовуються до всіх екземплярів проєкту і визначають мережевий доступ до сервера [30]. Ці правила призначені для того, щоб забезпечити більшу безпеку хмарного середовища. Для управління групами безпеки в OpenStack зазвичай використовується OpenStack CLI або через вебінтерфейс OpenStack Dashboard — Horizon.

4.7.4 Налаштування DNS

Останнім важливим кроком налаштування є додавання DNS серверу до підмережі (рис. 4.9).

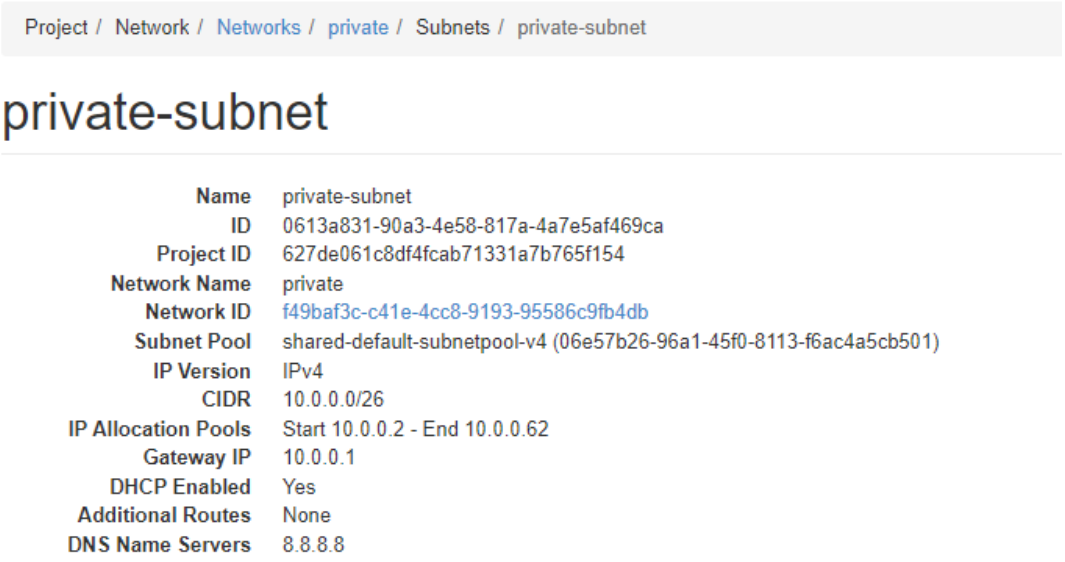


Рисунок 4.9 — Підмережа з налаштуванням DNS серверу

Це дозволить серверам мати доступ до Інтернету по доменному імені, а не лише за IP-адресою. В нашому випадку, використовується DNS сервер від Google, який має IP-адресу 8.8.8.8.

4.8 Робота користувача з системою

Для роботи з системою користувачу треба мати підключення до мережі Інтернет та наявність пристрою з будь-яким сучасним інтернет-браузером, який підтримує мову програмування JavaScript. Також бажано мати електронну пошту та будь-який SSH-клієнт.

Після запуску вебдодатка користувачу необхідно авторизуватися у системі, якщо він має обліковий запис. Форма авторизації показана на рисунку 4.10. Якщо дані користувача не будуть існувати в базі даних, він отримає відповідну помилку.

Якщо користувач не має облікового запису у системі чи бажає зареєструвати ще один, йому необхідно пройти реєстрацію. Важливо вказувати правильні дані, особливо пошту, адже вони будуть використовуватися у майбутньому при роботі з

віртуальними серверами. На рисунку 4.11 продемонстрована форма реєстрації з усіма необхідними полями.

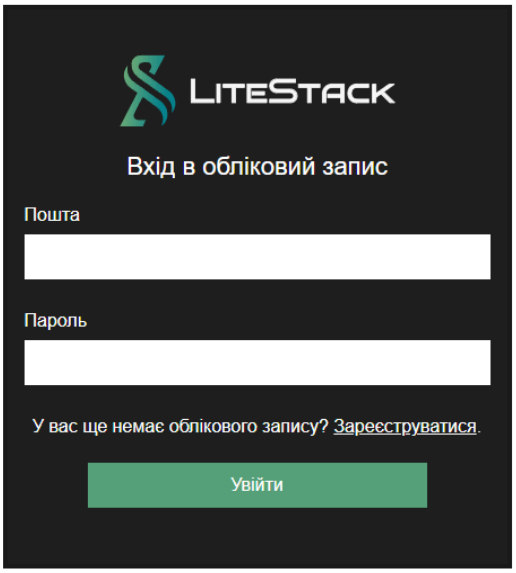


Рисунок 4.10 — Форма авторизації

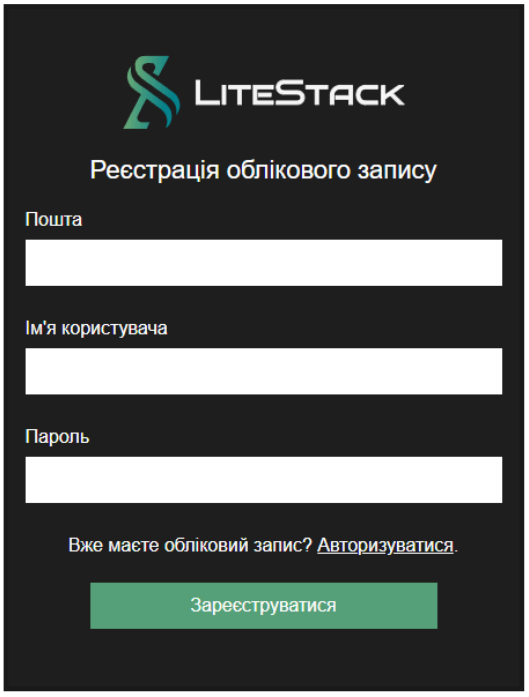


Рисунок 4.11 — Форма реєстрації

Після процесу авторизації користувач потрапляє на головну сторінку, на якій розміщені графіки з кількістю доступних та наявних віртуальних серверів і розподіл серверів по тегах (рис. 4.12).

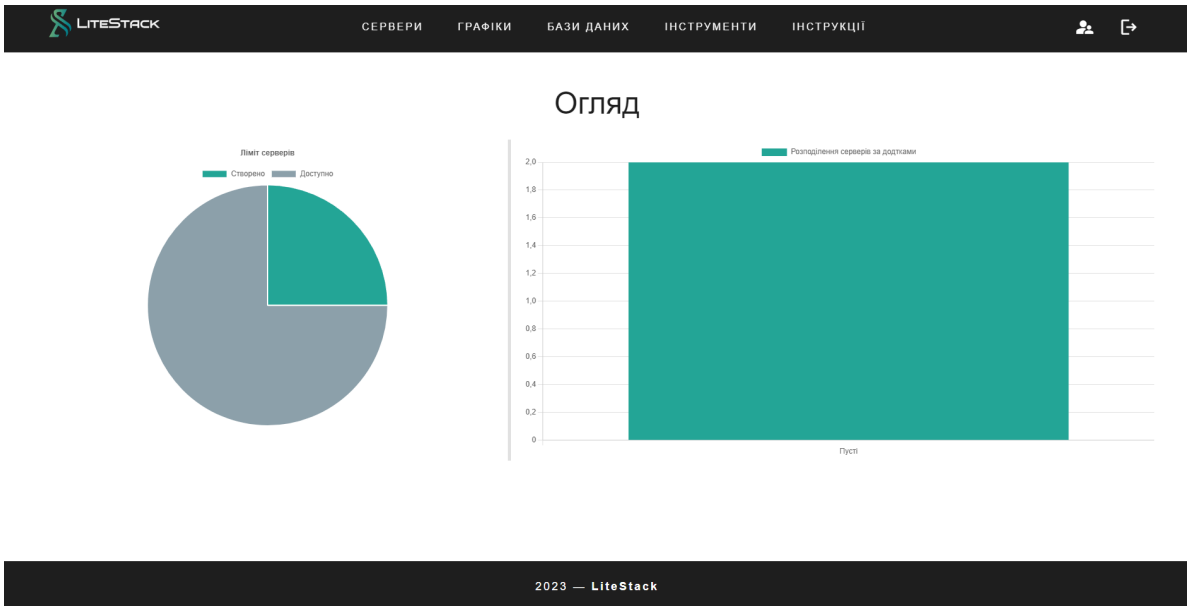


Рисунок 4.12 – Головна сторінка

При переході на сторінку «Сервери», користувачу відображаються наявні у системі конфігурації, список створених віртуальних серверів та дії, які можна з ними зробити – редагування і видалення (рис. 4.13).

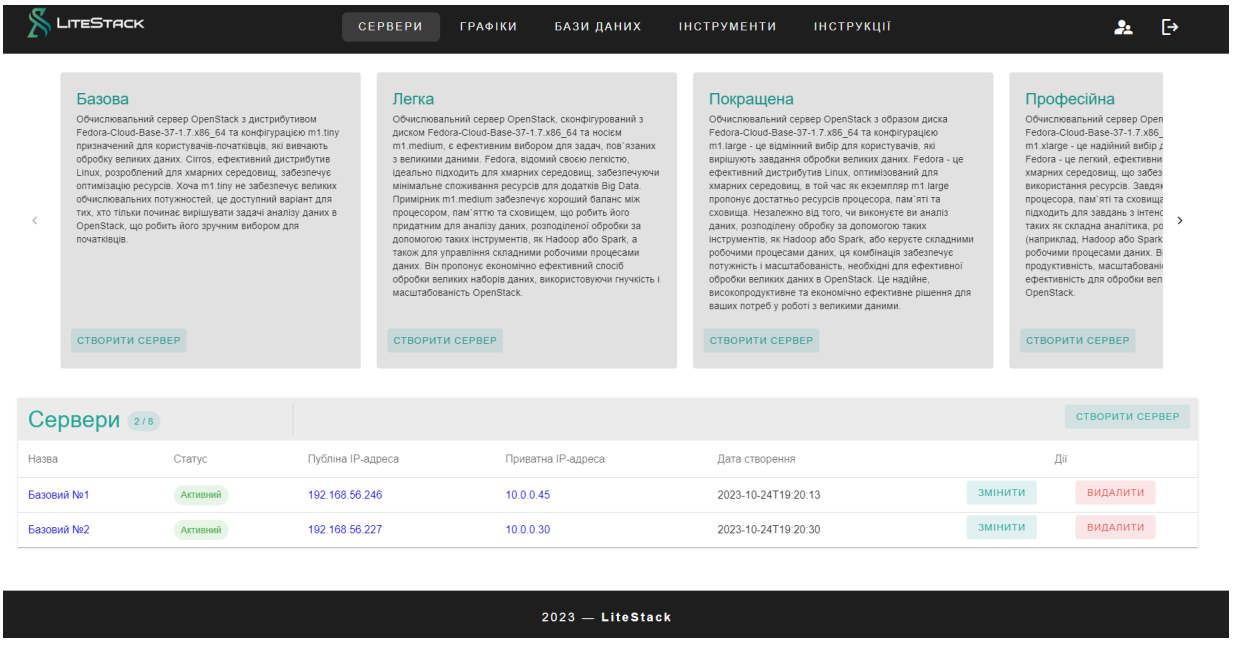


Рисунок 4.13 — Сторінка «Сервери»

Також користувачу пропонується створити сервер за однією з конфігурації, що і зображено на рисунку 4.14.

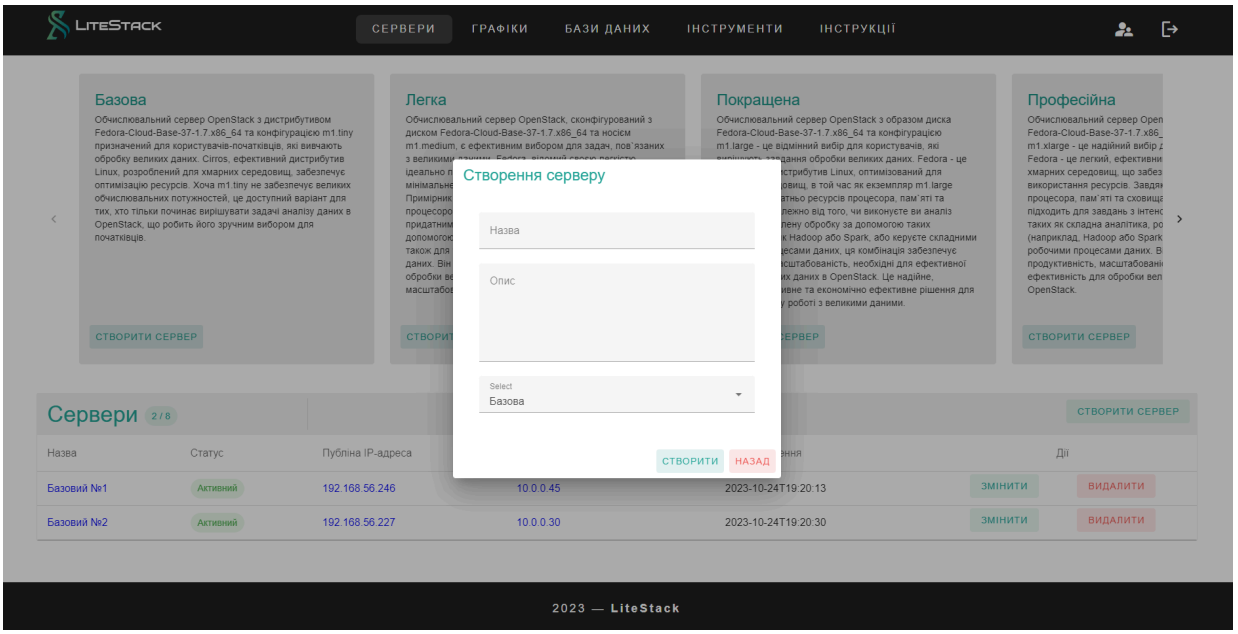


Рисунок 4.14 — Форма створення сервера

Після успішного створення сервера користувачу на пошту, по якій зареєстрований його обліковий запис, прийде повідомлення з SSH-ключами для під’єднання до сервера та інструкцією, щодо їх використання (рис. 4.15).

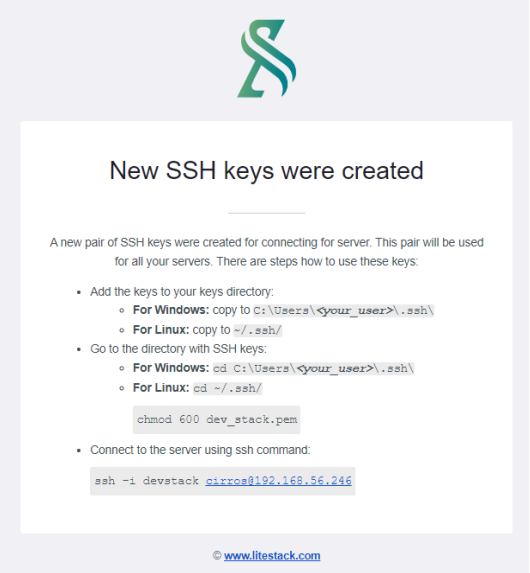


Рисунок 4.15 – Повідомлення на пошті користувача

Також, є можливість переглянути детальну інформацію, щодо сервера та зробити додаткові дії, а саме: призупинити, відновити, запустити, зупинити та перезавантажити сервер (рис. 4.16).

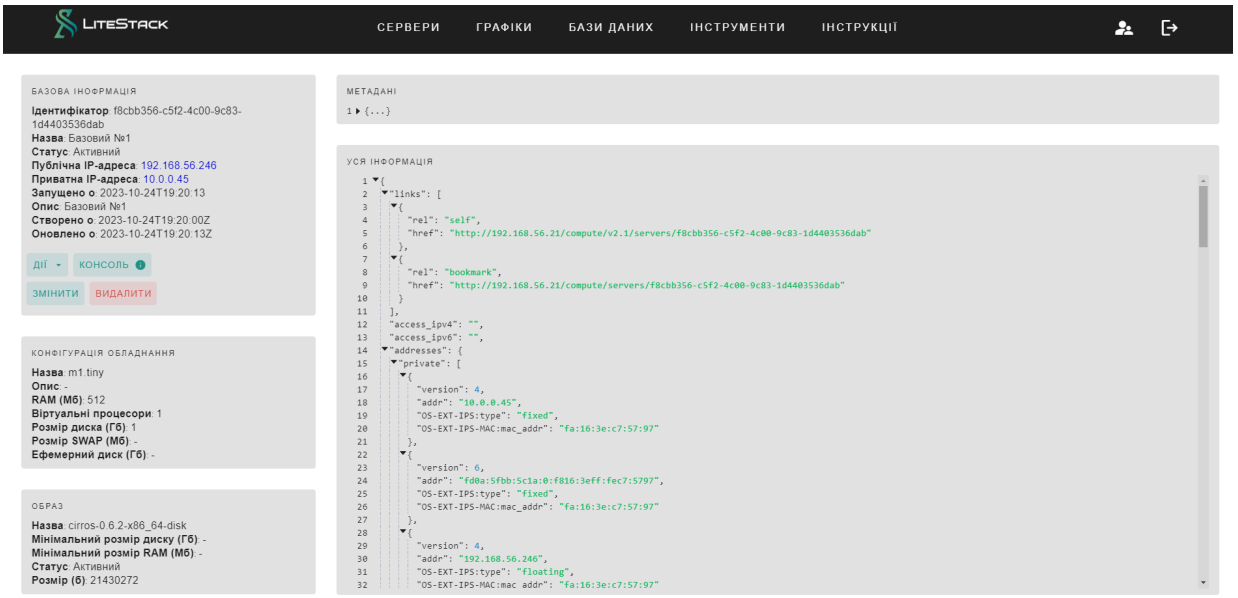


Рисунок 4.16 – Екран детальної інформації про сервер

У деталях можна знайти базову інформацію про сервер, характеристики обладнання, характеристики образу, метадані та розділ для технічно орієнтованих користувачів, який надає детальну інформацію про сервер у форматі JSON.

На додаток, користувач має змогу під'єднатися до сервера через браузер або будь-який SSH-клієнт. Вказівки, щодо використання даного функціоналу знаходиться у відповідному розділі з інструкціями.

Висновки до розділу 4

У цьому розділі детально висвітлено ключові аспекти створення системи. Була описана загальна архітектура системи для розуміння взаємодії між компонентами. Перелік кінцевих точок в API документації надає розуміння процесі взаємодії клієнта та сервера. Ілюстрація логічної моделі бази даних визначає схему та усі залежності моделей даних.

Щобільше, були детально продемонстровані процеси оптимального підбору конфігурацій віртуальних серверів для користувачів та налаштування хмарного середовища OpenStack. Також був описаний процес взаємодії користувача з системою.

5 РОЗРОБКА СТАРТАП-ПРОЄКТУ

В даному розділі будуть висвітлені особливості маркетингового аналізу продукту для оцінювання його можливостей та формування заходів щодо ринкового впровадження. Буде зроблений опис основної ідеї проєкту і проведений аудит технологічного стека. Щобільше, будуть описані шляхи ринкової реалізації та визначений можливий напрям розвитку цієї реалізації.

5.1 Опис ідеї проєкту

Детальний опис і аналіз ідеї проєкту складається з його змісту, напрямків застосування системи, вигоди використання продукту для користувачів та порівняльної характеристики з подібними або аналогічними на ринку продуктами.

Зміст ідеї полягає у розробці й впровадженні системи, яка через взаємодію з OpenStack надає користувачу легкий та зручний інтерфейс для створення й управління віртуальними серверами та ресурсами для зберігання даних з можливістю авторизації та адміністрування.

В таблиці 5.1 описані напрямки застосування та вигоди використання у цих напрямках.

Таблиця 5.1 — Опис ідеї стартап-проєкту

№	Напрямки застосування	Вигоди використання
1	2	3
1	Навчальний процес	Надання платформи для запуску програм учням та студентам, які вивчають технології з необхідністю у важких обчисленнях чи у великій кількості ресурсів

Продовження таблиці 5.1

1	2	3
2	Науково-дослідний процес	Надання можливості науковцям робити ресурсозатратні досліди, які важко запускати на власній машині, надання можливості хостингу сайтів на віртуальних серверах
3	Сучасний UI для OpenStack з відкритим вихідним кодом	Підвищення ефективності роботи технічних спеціалістів при необхідності легкої взаємодії з OpenStack, а саме для створення віртуальних серверів

Для успішного впровадження стартапу критично важливим є проведення аналізу ринку: провести оцінку наявних схожих продуктів та дослідити здатність проєкту конкурувати з іншими гравцями на ринку та мати попит серед користувачів.

Звичайно стартап продукт не може конкурувати з великими гравцями на ринку, які розроблялися десятками років та сотнями технічних спеціалістів, адже вони фактично є монополістами у своїй сфері. Але розроблена система може бути, не лише корисна для користувача при виконанні конкретної специфічної частини роботи, а займати лідерські позиції серед невеликих продуктів з аналогічним функціоналом.

Таблиця 5.2 описує порівняльний аналіз техніко-економічних показників задля визначення гірших, кращих та нейтральних характеристик та властивостей ідеї продукту. Ці дані фактично є підґрунтям для розуміння його конкурентоспроможності, що буде впливати на розробку ринкової стратегії та маркетингової програми.

Таблиця 5.2 — Визначення сильних, слабких та нейтральних характеристик ідеї проєкту

№	Технічно економічні характеристики ідеї	Товари конкурентів		
		Мій проєкт	Конкурент 1	Конкурент 2
1	2	3	4	5
1	Швидкість роботи системи	Нижча в порівнянні з конкурентами через взаємодію з OpenStack	Краща через використання більш низькорівневих технологій та можливість налаштування хмари	Краща через використання більш низькорівневих технологій та можливість налаштування хмари
2	Зручність використання	Використання більш зручне через легкий інтерфейс та багато автоматизацій поза оком користувача	Зручне, але має пересичений інтерфейс	Не зручне через спробу переосмислити стандартні процеси та перенасичення кінцевого інтерфейсу
3	Ціна за використання	Безплатна	Ціна залежить від часу роботу віртуального сервера, кількості серверів та взаємодій з ним	Ціна залежить від кількості серверів та взаємодій
4	Складність процесу створення віртуального сервера	Досить проста завдяки автоматизації багатьох процесів	Простий та зрозумілий процес	Досить складно через можливість налаштування багатьох параметрів та мануальної роботи

Продовження таблиця 5.2

1	2	3	4	5
5	Наявність заготовлених конфігурацій	Присутні конфігурації під різні типи задач та вимоги користувачів	Присутня деяка кількість конфігурацій, але без прив'язки під типи задач, які можуть якісно ці конфігурації вирішувати	Відсутні конфігурації
6	Механізм сповіщення поза межами системи	Присутній механізм сповіщення при створенні сервера для відправлення ключів доступу	Присутній механізм сповіщення при створенні, редагуванні, видалені чи будь-яких інших діях з віртуальними серверами	Присутній механізм сповіщення при створенні сервера для відправлення ключів доступу
7	Кастомізація віртуальних серверів	Майже відсутня через автоматизацію та спрощення процесу взаємодії користувача з системою	Присутні деякі способи впровадження налаштувань	Велика кількість всеможливих налаштувань

Як бачимо з таблиці сильними сторонами системи відносно конкурентів є зручність взаємодії, плата за використання, наявність заготовлених конфігурацій під різні типи задач. Нейтральні характеристики — складність процесу створення й управління віртуальними серверами та механізм сповіщення поза межами системи. Слабка характеристика це швидкість роботи системи.

Таким чином, присутні аспекти, які роблять продукт краще відносно інших, проте також є й такі, які послаблюють позиції в порівнянні з конкурентами. Але в цілому, порівняльний аналіз показав, що продукт має право на існування й може конкурувати з аналогами.

5.2 Технологічний аудит ідеї проєкту

Технологічний аудит проводиться для розуміння технологічного стека продукту: яка саме технологія буде використана для компонентів системи, їх наявність та доступність цієї технології розробникам. В таблиці 5.3 продемонстрований аналіз технологічної здійсненності ідеї проєкту.

Таблиця 5.3 — Технологічний аудит ідеї проєкту

№	Ідея проєкту	Технології реалізації	Наявність технологій	Доступність технологій
1	2	3	4	5
1	Хмара для взаємодії	OpenStack	+	Технології доступні безплатно
2	Спосіб взаємодії з хмарою	OpensStack API та OpenStackSDK	+	Технології доступні безплатно
3	Вебсервер	FastAPI	+	Технології доступні безплатно
4	Бізнес-логіка застосунку	Python 3.10	+	Технології доступні безплатно
5	База даних	PostgreSQL	+	Технології доступні безплатно

Продовження таблиця 5.3

1	2	3	4	5
6	Вебінтерфейс користувача	Vue3, HTML, CSS	+	Технології доступні безплатно

Проведений аналіз показав наявність усіх необхідних технологій та інструментів для реалізації продукту. Також можна стверджувати про відсутність необхідності створювати чи дороблювати технології, адже усі вони існують та не потребують плати за їх використання. Тобто, проєкт є здійсненним з технічної точки зору.

5.3.Аналіз ринкових можливостей стартап-проєкту

Аналіз ринкових можливостей стартап-проєкту допомагає визначити ситуацію на ринку, характеристики проєкту та потенційних користувачів системи, дає розуміння можливих загроз та дозволяє спланувати напрями розвитку проєкт з урахуванням стану ринкового середовища.

Таблиця 5.4 демонструє попередні характеристики потенційного ринку стартап-проєкту.

Таблиця 5.4 — Попередня характеристика потенційного ринку

№	Показники стану ринку (найменування)	Характеристика
1	2	3
1	Кількість головних гравців, од	10
2	Динаміка ринку (якісна оцінка)	Зростає

Продовження таблиця 5.4

1	2	3
3	Наявність обмежень для входу і їх характер	Відсутні
4	Специфічні вимоги до стандартизації та сертифікації	Відсутні
5	Середня норма рентабельності в галузі (або по ринку), %	21%

Спираючись на проведений аналіз характеристик ринку, можна дійти висновку, що ринок є відносно привабливим для входу, проте має велику кількість основних гравців, багато з яких є цифровими гігантами в технологічній сфері.

У таблиці 5.5 наведена характеристика потенційних клієнтів стартап-проекту. Враховуючи поставлені кафедрою задачі та специфічність застосунку, основним клієнтом є заклади освіти, невеликі дослідницькі компанії та технічні спеціалісти.

Таблиця 5.5 — Характеристика потенційних клієнтів стартап-проекту

Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару
1	2	3	4
Модуль для роботи з віртуальними серверами	Заклади освіти	Використання для наукових робіт або у процесі навчання студентів різних видів операцій	Простота, зручний інтерфейс, швидкість налаштування, наявність конфігурацій серверів

Продовження таблиця 5.5

1	2	3	4
Модуль для роботи з віртуальними серверами	Невеликі дослідницькі компанії	Використання для швидкого підняття серверів та перевірок гіпотез та теорій	Зручність використання, наявність конфігурацій серверів, можливість адміністрування
Модуль для роботи з віртуальними серверами	Технічні спеціалісти	Використання для швидкого підняття серверів	Зручність використання, наявність конфігурацій

Не менш важливим етапом є оцінка даних, які можуть перешкоджати або сприяти впровадженню проєкту на ринку, а саме — фактори загроз (таблиця 5.6) та фактори можливостей (таблиця 5.7) відповідно.

Таблиця 5.6 — Фактори загроз

№	Фактор	Зміст загрози	Можлива реакція компанії
1	2	3	4
1	Поява нових конкурентів	Конкуренти з аналогічним функціоналом	Покращення продукту та збільшення рекламної компанії
2	Збільшення ціни розробки та підтримки системи	Збільшення витрат на розробку та підтримку функціонала продукту	Перехід продукту на гнучкіші технології
3	Зміна тенденцій ринку	Нові необхідності цільової аудиторії	Адаптація до тенденцій ринку
4	Необхідність у нових конфігураціях віртуальних серверів	Користувачам продукту потрібні нові конфігурації	Впровадження нових заготовлених конфігурацій

Таблиця 5.7 — Фактори можливостей

№	Фактор	Зміст можливості	Можлива реакція компанії
1	2	3	4
1	Збільшення попиту на продукт	Поява більшої кількості клієнтів з необхідністю у функціоналу продукту	Впровадження нового та покращення наявного функціонала
2	Зменшення кількості конкурентів на ринку проєкту	Припинення робіт або вихід конкурентів з ринку	Перехват клієнтів конкурентів
3	Наявність унікальних функціональних особливостей	Удосконалення проєкту шляхом впровадження нового та покращення наявного функціонала	Збільшення кількості клієнтів зі складнішими вимогами

Наступним пунктом є аналіз конкуренції на ринку (таблиця 5.8), який надає розуміння загальних рис та особливостей конкурентного середовища.

Таблиця 5.8 — Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	Факт прояву характеристики	Вплив на діяльність підприємства
1	2	3
Тип конкуренції — олігополія	Домінує невелика кількість продуктів, які конкурують між собою і виробляють значну частку продукції галузі	Покращення проєкту по всім можливим параметрам та впровадження агресивнішої маркетингової компанії
Рівень конкурентної боротьби — міжнародне	Конкуренція ведеться без урахування географічних або національних ознак	Локалізація продукту під конкретний регіон

Продовження таблиці 5.8

1	2	3
Галузева ознака — міжгалузева	Продукт використовують для можливості роботи з задачами з інших галузей	Покращення функціональності системи
Конкуренція за видом товару — товаро-родова	Конкуренція ведеться на рівні технології	Покращення та впровадження загальноприйнятих практик та вимог
Характер конкурентних переваг — нецінова	Увага користувачів в першу чергу приділяється функціональним можливостям системи	Впровадження нового функціоналу
Інтенсивність — не марочна	Користувачі приділяють увагу можливостям системи, а не бренду	Впровадження нового та покращення наявного функціонала

На основі проведеному аналізу конкуренції та описаних факторів можливостей і загроз, можна назвати та обґрунтувати фактори конкурентоспроможності, які описані в таблиці 5.9.

Таблиця 5.9 — Обґрунтування факторів конкурентоспроможності

№	Фактор конкурентоспроможності	Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проєктів значущим)
1	2	3
1	Складність системи	Звичайний користувач не хоче витрачати багато часу на вивчення документації по взаємодії з продуктом, проте наявність невеликих інструкцій може бути приємним бонусом

Продовження таблиці 5.9

1	2	3
2	Зручність використання	Користувацький інтерфейс та його зручність неабияк впливають на досвід користувача у взаємодії з продуктом
3	Плата за використання	Для багатьох користувачів плата є досить таки вагомим фактором, тому підхід з певними обмеженнями по ресурсах може бути більш приємним для багатьох
4	Автоматизація ручних задач	Під час створення та управління серверами користувачі виконують багато повторювальних дій, які вимагають певних знань, тому їх автоматизація є гарним фактором для конкурентоспроможності системи
5	Наявність заготовлених конфігурацій	Користувачі без досвіду та певних знань роботи з залізом можуть мати труднощі при налаштуванні

Беручи за основу описані фактори конкурентоспроможності можна зробити порівняльний аналіз сильних та слабких сторін проєкту, під час якого продукт зіставляється з конкурентами в описаних факторах (таблиця 5.10).

Таблиця 5.10 — Порівняльний аналіз сильних та слабких сторін проєкту

№	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні з розробленим проєктом						
			-3	-2	-1	0	1	2	3
1	2	3	4						
1	Складність системи	16			К2		К1	П	

Продовження таблиці 5.10

1	2	3	4						
2	Зручність використання	15			К1	К2	П		
3	Плата за використання	14		К1			К2	П	
4	Автоматизація ручних задач	18				К2	К1		
5	Наявність заготовлених конфігурацій	17				К2	К1	П	

Аналіз сильних та слабких сторін показав, що продукт має можливості бути конкурентоспроможним по функціональних особливостях порівнюючи його до систем, які надають схожі можливості. Найбільше з усіх виділяються фактори автоматизації мануальних задач та наявність заготовлених конфігурацій, які є дуже важливими аспектами для користувачів.

5.4 Розроблення ринкової стратегії продукту

Ринкова стратегія — це ефективний інструмент маркетингу, що включає практику доведення до споживачів необхідної для підприємства інформації та розглядається як процес управління рухом товарів на всіх етапах. Ринкова стратегія є життєво необхідним джерелом існування будь-якого продукту, оскільки вона є основою для всіх сфер ринкової діяльності [31].

Процес розробки ринкової стратегії складається з декількох етапів: опис цільових груп потенційних споживачів, формування базової стратегії розвитку продукту та визначення базової стратегії конкурентної поведінки. Інколи ще додатково визначають стратегії позиціювання продукту, як висновок та уніфікація інформації з попередніх етапів.

У таблиці 5.11 проведений опис цільових груп потенційних споживачів, який допоможе оцінити та зрозуміти основний робочий сегмент продукту.

Таблиця 5.11 — Вибір цільових груп потенційних споживачів

Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в сегменті	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
1	2	3	4	5
Навчальні заклади	Потребують	Попит присутній	Незначна	Просто
Невеликі дослідницькі компанії	Потребують	Попит присутній	Помірна	Помірно
Технічні спеціалісти	Потребують	Попит присутній	Значна	Важко

Враховуючи проведений аналіз була визначена стратегія охоплення ринку — диференційованого маркетингу. Дана стратегія працює з групами та розробляє для кожної з них власну програму. У таблиці 5.12 детально визначена базова стратегія розвитку.

Таблиця 5.12 — Визначення базової стратегії розвитку

Обрана альтернатива розвитку проєкту	Стратегія охоплення ринку	Ключові позиції відповідно до конкурентів	Базова стратегія розвитку
1	2	3	4
Орієнтація поточної моделі під різні типи ринків	Диференційований маркетинг	Зручність використання безоплатність, можливість автоматизація ручних задач та наявність заготовлених конфігурацій	Стратегія диференціації

Останнім етапом у процесі розробки ринкової стратегії є створення базової стратегії конкурентної поведінки, що наведено у таблиці 5.13.

Таблиця 5.13 — Визначення базової конкурентної поведінки

Чи є проєкт новатором на ринку?	Чи буде компанія шукати нових споживачів, або забирати у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента?	Стратегія конкурентної поведінки
1	2	3	4
Ні	Так	Так, але з адаптацією під клієнтів	Стратегія заняття конкурентної ніші

Результатами проведеного аналізу є визначені базові принципи ринкової стратегії: метод охоплення ринку, базовий метод розвитку та вид конкурентної поведінки. Ці принципи мають неабиякий влив на можливість існування та розвитку проєкту в конкурентному середовищі, адже визначає як компанія буде взаємодіяти з конкурентами та підлаштовуватися під вимоги користувачів продукту.

5.5 Розроблення маркетингової програми стартап-проєкту

Маркетингова програма — це намічений для планомірного здійснення, об'єднаний єдиною метою та залежний від певних строків комплекс взаємопов'язаних завдань, спрямованих на розв'язання комплексних проблем [32].

Розроблення маркетингової програми стартап-проєкту складається з декількох кроків: формування маркетингової концепції товару (таблиця 5.14), визначення меж встановлення ціни та концепція маркетингових комунікацій

(таблиця 5.15). Пункт визначення меж встановлення ціни не є релевантним для розроблюваного продукту.

Таблиця 5.14 — Визначення переваг концепції потенційного товару

Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами
1	2	3
Зручність використання	Зручний, легкий та інтуїтивно зрозумілий користувацький інтерфейс	Спрощення складних процесів
Безоплатність	Безоплатність використання	Безоплатність використання
Автоматизація ручних задач	Автоматизація більшої частини мануальних процесів	Автоматизація більшої частини мануальних процесів
Наявність заготовлених конфігурацій	Присутня певна кількість заготовлених конфігурацій	Заготовлена певна кількість конфігурацій під певні типи задача та їх складність

Таблиця 5.15 — Концепція маркетингових комунікацій

Специфіка поведінки цільових клієнтів	Канали комунікацій цільових клієнтів	Ключові позиції, обрані для позиціювання	Завдання рекламного повідомлення	Концепція рекламного звернення
1	2	3	4	5
Клієнт отримує інформацію з реклами та зі статей на технічних сайтах	Електронна пошта, месенджери та соціальні мережі	Унікальні функціональні особливості, гнучкість системи та підтримка клієнтів	Донести інформацію про продукт та переваги над аналогами	Стаття на експертних сайтах та платформах з описом технічних особливостей

В результаті отримано маркетингову програму, яка охоплює концепції товару та маркетингових комунікацій з цільовою аудиторією.

Висновки до розділу 5

Розділ описує особливості та стадії побудови й аналізу стартап-проєкту. Він надає інформацію щодо ідеї проєкту, її технологічного аудиту, аналізу ринкових можливостей для впровадження продукту, розробки рекламної та маркетингової компаній. Усе це дозволяє зрозуміти можливі перспективи й бар'єри стартап-проєкту та доцільність його подальшої імплементації.

ВИСНОВКИ

Результатом виконання роботи є розроблена програмна платформа для створення та управління віртуальними серверами на основі хмарного середовища OpenStack. У ході розробки використані такі інструменти: мови програмування Python та JavaScript, фреймворки FastAPI та Vue, мова гіпертекстової розмітки HTML 5, таблиці каскадних стилів CSS 3 та система управління базами даних Postgres.

Розробка спеціальної архітектури допомогла зробити застосунок не лише швидким, а й гнучким для розширення та інтеграції додаткових функціональних особливостей. Сучасні й популярні інструменти, які використовувалися для розробки, допомогли створити систему з інтуїтивно зрозумілим та приємним для кінцевого користувача інтерфейсом.

Аналіз схожих продуктів допоміг зрозуміти актуальність завдання та факт нестачі подібних систем на ринку. Враховуючи це, можна дійти висновку, що розроблена система може бути досить конкурентоспроможною, адже аналоги не покривають поставлені вимоги на сто відсотків.

Проведений аналіз ідеї проєкту показав сильні та слабкі сторони розроблюваної системи. Технологічний аудит доказав правильність вибраних на минулих етапах інструментів. Розроблені ринкова та маркетингова стратегії сприяли якісній оцінці поточній ситуації на ринку та створенню плану маркетингового просування продукту.

Розроблена система має гарний потенціал для майбутнього розвитку, додаючи можливі конфігурації або дії з серверами. Тобто у перспективі продукт може конкурувати з більш серйозними гравцями ринку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. OpenStack docs. OpenStack Docs. URL: https://docs.openstack.org/latest/ (date of access: 01.12.2023).
2. What is a virtual server? | IBM. IBM in Deutschland, Österreich und der Schweiz | IBM. URL: https://www.ibm.com/topics/virtual-server (date of access: 02.12.2023).
3. Fowler M. UML distilled: a brief guide to the standard object modeling language (3rd edition) (the addison-wesley object technology series). 3rd ed. Addison-Wesley Professional, 2003. 208 p.
4. What is decomposition? - decomposition - KS3 computer science revision - BBC bitesize. BBC - Home. URL: https://www.bbc.co.uk/bitesize/guides/zqqfyrd/revision/1 (date of access: 03.12.2023).
5. Kanban: Successful Evolutionary Change for Your Technology Business. Sequim , Washington : Blue hole press, 2010. 261 p.
6. Horizon: The OpenStack Dashboard Project – horizon 23.4.0.dev76 documentation. OpenStack Docs: 2023.2. URL: https://docs.openstack.org/horizon/latest/ (date of access: 04.12.2023).
7. Skyline - OpenStack. OpenStack. URL: https://wiki.openstack.org/wiki/Skyline (date of access: 05.12.2023).
8. What is AWS? - Cloud Computing with AWS - Amazon Web Services. Amazon Web Services, Inc. URL: https://aws.amazon.com/what-is-aws/ (date of access: 06.12.2023).
9. Cloud Compute Capacity - Amazon EC2 - AWS. Amazon Web Services, Inc. URL: https://aws.amazon.com/ec2/ (date of access: 07.12.2023).
10. Google Cloud overview | Overview. Google Cloud. URL: https://cloud.google.com/docs/overview (date of access: 08.12.2023).

11. Compute Engine | Google Cloud. Google Cloud. URL: https://cloud.google.com/compute (date of access: 08.12.2023).
12. Lutz M. Learning Python. 5th ed. 2013. 1540 p.
13. FastAPI. FastAPI. URL: https://fastapi.tiangolo.com/ (date of access: 09.12.2023).
14. Welcome to Alembic's documentation! – Alembic 1.13.1 documentation. Welcome to Alembic's documentation! – Alembic 1.13.1 documentation. URL: https://alembic.sqlalchemy.org/en/latest/ (date of access: 22.12.2023).
15. Haverbeke M. Eloquent JavaScript: A Modern Introduction to Programming. 3rd ed. San Francisco, California, United States of America : No Starch Press, 2018. 472 p.
16. Introduction | Vue.js. Vue.js - The Progressive JavaScript Framework | Vue.js. URL: https://vuejs.org/guide/introduction.html (date of access: 10.12.2023).
17. Why you should be using Vuetify – Vuetify. Vuetify. URL: https://vuetifyjs.com/en/introduction/why-vuetify/ (date of access: 11.12.2023).
18. Introduction | Pinia. Pinia | The intuitive store for Vue.js. URL: https://pinia.vuejs.org/introduction.html (date of access: 12.12.2023).
19. Git. Git. URL: https://git-scm.com/ (date of access: 13.12.2023).
20. What is version control | Atlassian Git Tutorial. Atlassian. URL: https://www.atlassian.com/git/tutorials/what-is-version-control (date of access: 14.12.2023).
21. What is Docker? | AWS. Amazon Web Services, Inc. URL: https://aws.amazon.com/docker/?nc1=h_ls (date of access: 15.12.2023).
22. Docker Compose overview. Docker Documentation. URL: https://docs.docker.com/compose/ (date of access: 15.12.2023).
23. PostgreSQL. PostgreSQL. URL: https://www.postgresql.org/ (date of access: 16.12.2023).

24. JWT.IO. JSON Web Tokens - jwt.io. URL: https://jwt.io/ (date of access: 16.12.2023).
25. What is RESTful API? - RESTful API Explained - AWS. Amazon Web Services, Inc. URL: https://aws.amazon.com/what-is/restful-api/?nc1=h_ls (date of access: 17.12.2023).
26. Microservices vs. monolithic architecture | Atlassian. Atlassian. URL: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith (date of access: 18.12.2023).
27. Учасники проектів Вікімедіа. HTTP – Вікіпедія. Вікіпедія. URL: https://uk.wikipedia.org/wiki/HTTP (дата звернення: 19.12.2023).
28. What is a REST API? | IBM. IBM in Deutschland, Österreich und der Schweiz | IBM. URL: https://www.ibm.com/topics/rest-apis (date of access: 19.12.2023).
29. Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing Series) / R. Helm et al. Addison-Wesley Professional, 1995. 395 p.
30. OpenStack Docs: Manage project security. OpenStack Docs: 2023.2. URL: https://docs.openstack.org/nova/queens/admin/security-groups.html (date of access: 20.12.2023).
31. Гулик Т. В. Сутність поняття ринкової стратегії та її місце в системі міжнародного маркетингу : Стаття. Мукачів, 2018. 8 с.
32. Біловодська О. А. Маркетинговий менеджмент : навч. посіб. Київ : Знання, 2010. 332 с.

ДОДАТКИ

ДОДАТОК А Лістинг розробленої програми

```
import uuid
from fastapi import APIRouter, Depends, HTTPException, BackgroundTasks
from openstack import connection
from openstack.exceptions import ConflictException, ResourceNotFound, DuplicateResource
from openstack.exceptions import  HttpException as OpenstackHTTPException
from sqlalchemy.ext.asyncio import AsyncSession
import app.openstack.compute_service as openstack
import app.servers.db_service as db
from app.auth.config import fastapi_users
from app.auth.models import User
from app.command.echo import EchoCommand
from app.config import settings
from app.dependencies import get_openstack_connection, get_async_session
from app.command.command import ExecAction
from app.command.grafana import GrafanaCommand
from app.command.matplotlib import MatplotlibCommand
from app.command.mongo import MongoCommand
from app.command.postgres import PostgresCommand
from app.command.pytorch import TorchCommand
from app.command.tensorflow import TensorflowCommand
from app.servers.schemas import (
    Server as ServerSchema,
    ServerDetailed as ServerDetailedSchema,
    ServerStateActionUpdate,
        ServerStateActionEnum,  ServerConfiguration,  ServeCreate,  ServeUpdate,  ServerCommand,
ServerCommandEnum
)
from app.servers.service import send_keypair_email
from app.openstack.models import get_os_default_user, get_server_public_ip
from app.runner.service import CommandRunner
from fastapi.concurrency import run_in_threadpool

current_user = fastapi_users.current_user()
```

```
router = APIRouter(
    prefix="/servers",
    tags=["servers"]
)


@router.get("/configurations", response_model=list[ServerConfiguration])
async def get_server_configurations(
        _: User = Depends(current_user),
        session: AsyncSession = Depends(get_async_session),
):
    try:
        server_configurations = await db.get_server_configurations(session)
        server_configurations.sort(key=lambda item: item.name)
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Failed to list server configurations: {e}")

    return server_configurations


@router.get("/limit")
async def get_server_limit(
        conn: connection.Connection = Depends(get_openstack_connection),
        _: User = Depends(current_user),
):
    return {'limit': openstack.get_instance_limit(conn)}


@router.get("", response_model=list[ServerSchema])
async def get_user_servers_list(
        user: User = Depends(current_user),
        conn: connection.Connection = Depends(get_openstack_connection),
        session: AsyncSession = Depends(get_async_session)
):
    try:
        if user.is_superuser:
            user_servers = await db.get_all_servers(session)
```

```
        else:
            user_servers = await db.get_user_servers(user, session)

        if len(user_servers) == 0:
            return []
        server_ids = [server.openstack_id for server in user_servers]
        servers = openstack.get_servers_by_ids(conn, server_ids)

        user_server_map={str(server.openstack_id): server for server in user_servers}

    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Failed to list server: {e}")

                                return [ServerSchema.create_from_openstack_server(user.id, server,
user_server_map.get(server.id, {})) for server in servers]


@router.get("/instruments", response_model=list[ServerSchema])
async def get_configurable_user_servers_list(
        user: User = Depends(current_user),
        conn: connection.Connection = Depends(get_openstack_connection),
        session: AsyncSession = Depends(get_async_session)
):
    try:
        if user.is_superuser:
            user_servers = await db.get_all_servers(session)
        else:
            user_servers = await db.get_user_servers(user, session)

        if len(user_servers) == 0:
            return []

        server_ids = []
        for server in user_servers:
            if 'cirros' not in server.image:
                server_ids.append(server.openstack_id)

        servers = openstack.get_servers_by_ids(conn, server_ids)
```

```
        user_server_map={str(server.openstack_id): server for server in user_servers}

    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Failed to list server: {e}")

                                    return [ServerSchema.create_from_openstack_server(user.id, server,
user_server_map.get(server.id, {})) for server in servers]

@router.post("/{server_id}/command", status_code=200)
async def run_command_on_server(
        background_tasks: BackgroundTasks,
        server_id: uuid.UUID,
        server_command: ServerCommand,
        user: User = Depends(current_user),
        conn: connection.Connection = Depends(get_openstack_connection),
        session: AsyncSession = Depends(get_async_session),
):
    try:
        if not await db.is_user_server(server_id, user, session):
            return HTTPException(status_code=404, detail="Server not found")

        server = await db.get_user_server(user, str(server_id), session)
        openstack_server = openstack.get_server(conn, str(server_id))
        executor = None
        match server_command.command:
            case ServerCommandEnum.install_torch :
                executor = TorchCommand(ExecAction.INSTALL)
            case ServerCommandEnum.delete_torch:
                executor = TorchCommand(ExecAction.DELETE)
            case ServerCommandEnum.install_tensorflow:
                executor = TensorflowCommand(ExecAction.INSTALL)
            case ServerCommandEnum.delete_tensorflow:
                executor = TensorflowCommand(ExecAction.DELETE)
            case ServerCommandEnum.install_grafana:
                executor = GrafanaCommand(ExecAction.INSTALL)
            case ServerCommandEnum.delete_grafana:
                executor = GrafanaCommand(ExecAction.DELETE)
```

```
            case ServerCommandEnum.install_matplotlib:
                executor = MatplotlibCommand(ExecAction.INSTALL)
            case ServerCommandEnum.delete_matplotlib:
                executor = MatplotlibCommand(ExecAction.DELETE)
            case ServerCommandEnum.install_postgres:
                executor = PostgresCommand(ExecAction.INSTALL)
            case ServerCommandEnum.delete_postgres:
                executor = PostgresCommand(ExecAction.DELETE)
            case ServerCommandEnum.install_mongo:
                executor = MongoCommand(ExecAction.INSTALL)
            case ServerCommandEnum.delete_mongo:
                executor = MongoCommand(ExecAction.DELETE)
            case ServerCommandEnum.echo:
                executor = EchoCommand()

        if executor:
                                        background_tasks.add_task(CommandRunner(server,    executor,
get_server_public_ip(openstack_server)).run)

    except ResourceNotFound as e:
        raise HTTPException(status_code=404, detail=str(e))
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Failed to run command: {e}")

@router.get("/{server_id}", response_model=ServerDetailedSchema)
async def get_user_server(
        server_id: uuid.UUID,
        user: User = Depends(current_user),
        conn: connection.Connection = Depends(get_openstack_connection),
        session: AsyncSession = Depends(get_async_session)
):
    if not await db.is_user_server(server_id, user, session):
        raise HTTPException(status_code=404, detail="Server not found")
    try:
        image=None
        openstack_server = openstack.get_server(conn, str(server_id))
        if openstack_server.image.id:
            image = openstack.get_image(conn, openstack_server.image.id)
```

```
        server = await db.get_user_server(user, str(server_id), session)
    except ResourceNotFound:
        raise HTTPException(status_code=404, detail=f"Server not found")
    except DuplicateResource:
        raise HTTPException(status_code=409, detail=f"Multiple servers found with ID {server_id}")
    except OpenstackHTTPException as e:
        raise HTTPException(status_code=e.status_code, detail=e.details)
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Failed to get server: {e}")

        return  ServerDetailedSchema.create_from_openstack_server(user.id,server,  openstack_server,
image)

@router.get("/{server_id}/console-url")
async def get_user_server_console_url(
        server_id: uuid.UUID,
        user: User = Depends(current_user),
        conn: connection.Connection = Depends(get_openstack_connection),
        session: AsyncSession = Depends(get_async_session)
):
    if not await db.is_user_server(server_id, user, session):
        raise HTTPException(status_code=404, detail="Server not found")
    try:
        console = openstack.create_server_console(conn, str(server_id))
    except ResourceNotFound:
        raise HTTPException(status_code=404, detail=f"Server not found")
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Failed to create server console: {e}")

    url = console.get('url', '')
    if url == '':
        raise HTTPException(status_code=500, detail=f"Failed to create server console: no url")

    return {'url': url}

@router.post("/from_configuration", response_model=ServerDetailedSchema)
async def create_user_server(
        background_tasks: BackgroundTasks,
```

```
    req: ServeCreate,
    user: User = Depends(current_user),
    conn: connection.Connection = Depends(get_openstack_connection),
    session: AsyncSession = Depends(get_async_session),
):
  try:
    servers = await db.get_user_servers(user, session)
    if len(servers) > openstack.get_instance_limit(conn):
      raise HTTPException(status_code=409, detail="Too many servers")

    server_config = await db.get_server_configuration(req.configuration_name, session)

    openstack_server, key_pair, floating_ip = openstack.create_server(
      conn,
      str(user.id),
      req.name,
      req.description,
      server_config,
    )

    public_ip_address = floating_ip.floating_ip_address
    if key_pair.private_key and settings.mail_username != "" and public_ip_address != "":
                    await send_keypair_email(background_tasks, user, key_pair, public_ip_address,
get_os_default_user(server_config.image))

    await db.insert_user_server(openstack_server.id, user.id, server_config.image, session)
    server = await db.get_user_server(user, str(openstack_server.id), session)
  except ResourceNotFound as e:
    raise HTTPException(status_code=404, detail=f"Resource not found: {e}")
  except Exception as e:
    raise HTTPException(status_code=500, detail=f"Failed to create server: {e}")

  return ServerDetailedSchema.create_from_openstack_server(user.id, server, openstack_server)


@router.delete("/{server_id}", status_code=204)
async def delete_user_server(
    server_id: uuid.UUID,
```

```
    user: User = Depends(current_user),
    conn: connection.Connection = Depends(get_openstack_connection),
    session: AsyncSession = Depends(get_async_session)
):
  try:
    if not await db.is_user_server(server_id, user, session):
      raise HTTPException(status_code=404, detail="Server not found")

    openstack.delete_server(conn, str(server_id))
    await db.delete_user_server(server_id, session)

  except ResourceNotFound:
    raise HTTPException(status_code=404, detail=f"Server not found")
  except Exception as e:
    raise HTTPException(status_code=500, detail=f"Failed to get server: {e}")

  return


@router.patch("/{server_id}", status_code=200)
async def update_server(
    server_id: uuid.UUID,
    req: ServeUpdate,
    user: User = Depends(current_user),
    conn: connection.Connection = Depends(get_openstack_connection),
    session: AsyncSession = Depends(get_async_session)
):
  try:
    if not await db.is_user_server(server_id, user, session):
      raise HTTPException(status_code=404, detail="Server not found")

    openstack.update_server(conn, str(server_id), req.name, req.description)

  except ResourceNotFound as e:
    raise HTTPException(status_code=404, detail=str(e))
  except ConflictException as e:
    raise HTTPException(status_code=409, detail=str(e))
  except Exception as e:
```

```
        raise HTTPException(status_code=500, detail=f"Failed to update server: {e}")


@router.patch("/{server_id}/state", status_code=200)
async def set_state_action(
        server_id: uuid.UUID,
        state_action: ServerStateActionUpdate,
        user: User = Depends(current_user),
        conn: connection.Connection = Depends(get_openstack_connection),
        session: AsyncSession = Depends(get_async_session)
):
    try:
        if not await db.is_user_server(server_id, user, session):
            raise HTTPException(status_code=404, detail="Server not found")

        match state_action.action:
            case ServerStateActionEnum.pause:
                openstack.pause_server(conn, str(server_id))
            case ServerStateActionEnum.unpause:
                openstack.unpause_server(conn, str(server_id))
            case ServerStateActionEnum.start:
                openstack.start_server(conn, str(server_id))
            case ServerStateActionEnum.stop:
                openstack.stop_server(conn, str(server_id))
            case ServerStateActionEnum.reboot:
                openstack.reboot_server(conn, str(server_id))
    except ResourceNotFound as e:
        raise HTTPException(status_code=404, detail=str(e))
    except ConflictException as e:
        raise HTTPException(status_code=409, detail='Invalid status')
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Failed to set server state: {e}")

import time

from openstack import connection
from openstack.compute.v2.flavor import Flavor as OpenStackFlavor
from openstack.compute.v2.image import Image as OpenStackImage
```

```
from openstack.compute.v2.keypair import Keypair as OpenStackKeypair
from openstack.compute.v2.server import Server as OpenStackServer
from openstack.network.v2.floating_ip import FloatingIP as OpenstackFloatingIP

import app.openstack.network_service as network_service
from app.config import settings
from app.openstack.block_service import create_volume, get_block_device_mapping
from app.servers.models import ServerConfig


def create_server(
        conn: connection.Connection,
        user_id: str,
        name: str,
        description: str,
        server_config: ServerConfig,
) -> (OpenStackServer, OpenStackKeypair, OpenstackFloatingIP):
    block_device_mapping = []

    # if image is not equal to cirros
    if "cirros" not in server_config.image.split("-"):
        volume = create_volume(conn, server_config.image, name + "_disk")
        block_device_mapping.append(get_block_device_mapping(volume))

    image = conn.image.find_image(server_config.image)
    flavor = conn.compute.find_flavor(server_config.flavor)

    networks = []
    for network in server_config.networks:
        openstack_network = network_service.get_network(conn, network)
        networks.append({"uuid": openstack_network.id})

    keypair = create_keypair(conn, user_id)

    server = conn.compute.create_server(
        name=name,
        description=description,
        admin_password=name,
```

```
        image_id=image.id,
        flavor_id=flavor.id,
        networks=networks,
        key_name=keypair.name,
        block_device_mapping_v2=block_device_mapping,
    )
    conn.compute.wait_for_server(server)
    ip_address = network_service.create_floating_ip_and_assign_to_server(conn, server)
    return server, keypair, ip_address


def add_floating_ip_to_server(conn: connection.Connection, server: OpenStackServer, floating_ip:
OpenstackFloatingIP):
    conn.compute.add_floating_ip_to_server(server, floating_ip)


def create_keypair(conn: connection.Connection, name) -> OpenStackKeypair:
    keypair = conn.compute.find_keypair(name)

    if not keypair:
        keypair = conn.compute.create_keypair(name=name)

    return keypair


def create_floating_ip(conn: connection.Connection, network_id: str) -> OpenstackFloatingIP:
    return conn.network.create_ip(floating_network_id=network_id)


def get_all_servers(conn: connection.Connection) -> list[OpenStackServer]:
    return conn.compute.servers(all_projects=True)


def get_servers_by_ids(conn: connection.Connection, ids: list[str]) -> list[OpenStackServer]:
    servers = []
    for server_id in ids:
        server = conn.compute.find_server(server_id)
        if server:
```

```
            servers.append(server)
        else:
            print(f"Server with ID {server_id} not found.")

    return servers


def get_server(conn: connection.Connection, server_id: str) -> OpenStackServer:
    return conn.compute.find_server(server_id)


def create_server_console(conn, server_id: str) -> dict[str]:
    return conn.compute.create_console(server_id, "novnc")


def update_server(
        conn: connection.Connection,
        server_id: str,
        name: str,
        description: str,
):
    data = {}
    if name:
        data["name"] = name

    if description:
        data["description"] = description

    conn.compute.update_server(server_id, **data)


def delete_server(conn: connection.Connection, server_id: str):
    conn.compute.delete_server(server_id, ignore_missing=True)


def pause_server(conn: connection.Connection, server_id: str):
    conn.compute.pause_server(server_id)
```

```
def unpause_server(conn: connection.Connection, server_id: str):
    conn.compute.unpause_server(server_id)


def start_server(conn: connection.Connection, server_id: str):
    conn.compute.start_server(server_id)


def stop_server(conn: connection.Connection, server_id: str):
    conn.compute.stop_server(server_id)


def reboot_server(conn: connection.Connection, server_id: str):
    conn.compute.reboot_server(server_id, reboot_type="HARD")


def get_flavors(conn: connection.Connection) -> list[OpenStackFlavor]:
    return conn.compute.flavors()


def get_images(conn: connection.Connection) -> list[OpenStackImage]:
    return conn.compute.images()


def get_image(conn: connection.Connection, image_id: str) -> list[OpenStackImage]:
    return conn.compute.find_image(image_id)


def get_instance_limit(conn: connection.Connection) -> int:
    limits = conn.compute.get_limits()
    max_server_limit = limits.absolute['maxTotalInstances']
    return min(max_server_limit, settings.max_server_limit)
```

ДОДАТОК Б Презентація

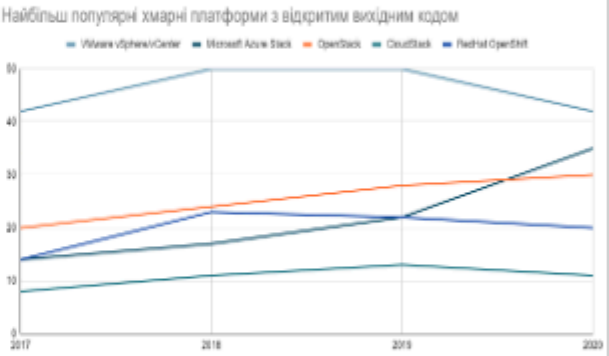
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
Кафедра Інженерії програмного забезпечення в енергетиці.

Модуль для створення віртуальних серверів та ресурсів для зберігання даних на основі хмарного середовища OpenStack

Виконав: студент магістерського рівня 2-го року навчання, групи ТВ-21мп
Лебедєв Артем Вячеславович
Керівник: д.е.н., професор, професор кафедри ІПЗЕ Сігайов А.О.

Актуальність роботи

- Популярність технологій, які потребують великої кількості ресурсів.
- Популярність хмарних технологій на ринку, в особливості OpenStack.
- Складність управління та необхідність в додаткових знаннях для взаємодії з хмарою.
- Мала кількість продуктів, яка пропонує створення віртуальних серверів за заданими конфігураціями.
- Складні користувацькі інтерфейси систем, які взаємодіють з OpenStack.

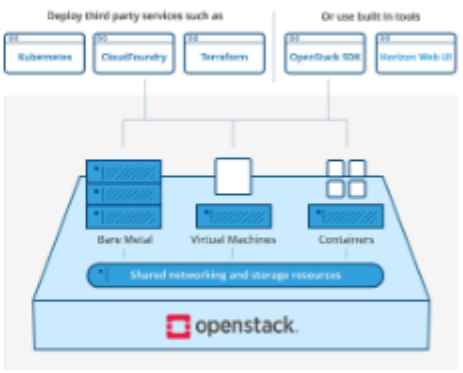


Мета, об'єкт та предмет дослідження

- **Мета роботи** — розробка програмної платформи управління віртуальними серверами та ресурсами для зберігання даних на основі хмарного середовища OpenStack.
- **Об'єкт дослідження** — процес розробки системи управління віртуальними серверами та ресурсами.
- **Предмет дослідження** — модуль управління віртуальними серверами та ресурсами для зберігання даних з інтерфейсом користувача на основі хмарного середовища OpenStack.

Основні завдання

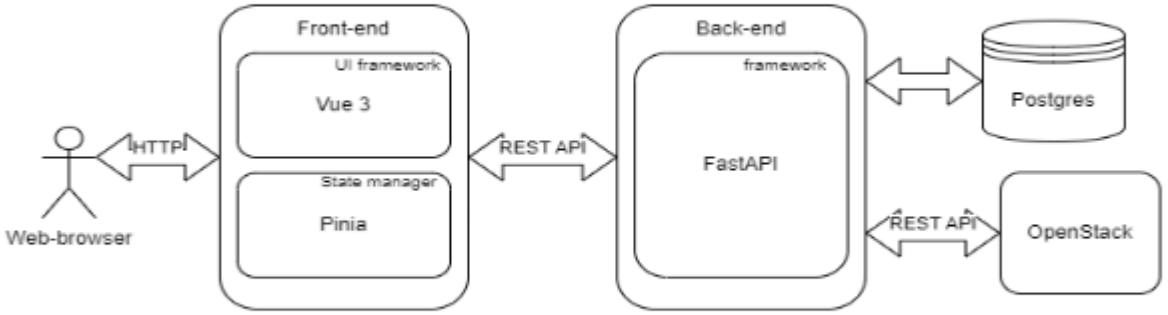
- Опрацювати документацію хмарного середовища OpenStack.
- Визначити метод взаємодії з OpenStack.
- Створити варіанти можливих конфігурацій серверів з урахуванням потреб користувачів.
- Проаналізувати підходи конкурентів, визначені їх слабкі та сильні сторони.
- Реалізувати програмну систему, з урахуванням методу взаємодії з OpenStack та завчасно створених конфігурацій.
- Визначити напрями розвитку та основні аспекти для подальшого удосконалення системи.



Порівняння з аналогами

Апаратні методи вирішення поставленої задачі

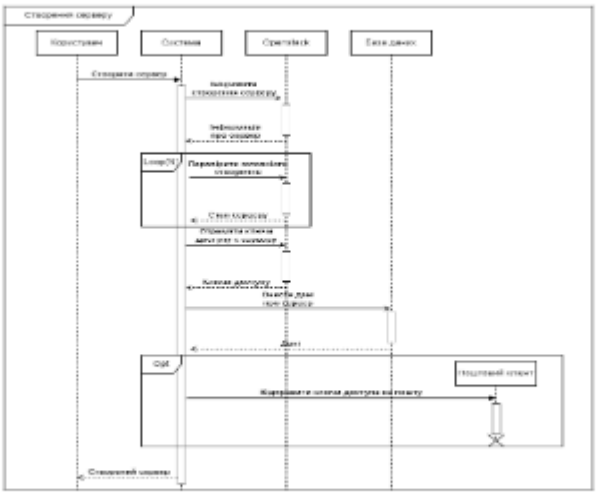
Архітектура системи



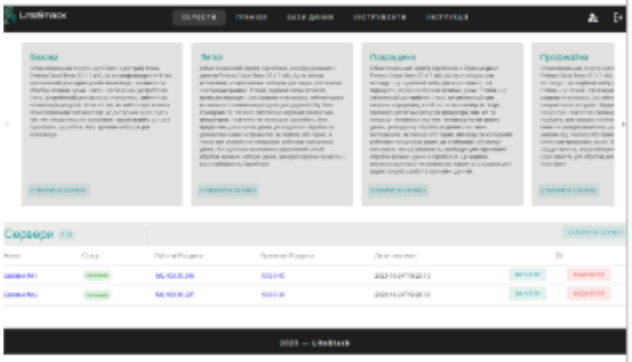
Підготовлені конфігурації віртуальних серверів

Назва	Image	Flavor	Volumes	Networks
Базова	Fedora-Cloud-Base-37-1.7.x86_64	m1.small	50GB SSD	Private, Public
Легка	Fedora-Cloud-Base-37-1.7.x86_64	m1.medium	100GB SSD + 1TB HDD	Private, Public
Покращена	Fedora-Cloud-Base-37-1.7.x86_64	m1.large	250GB SSD + 2TB HDD	Private, Public
Професійна	Fedora-Cloud-Base-37-1.7.x86_64	m1.xlarge	500GB SSD + 4TB HDD	Private, Public

Діаграма послідовності створення сервера

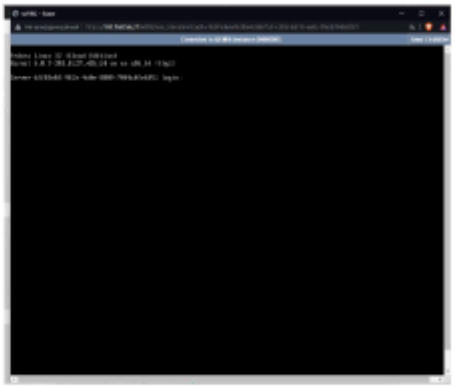


Результати роботи



Результати роботи

Результати роботи



Висновки

В результаті виконання дипломної роботи була розроблена **програмна платформа для створення та управління віртуальними серверами на основі хмарного середовища OpenStack**.

Основні **факти про створену систему**:

- Використання сучасних та популярних інструментів пришвидшило процес розробки приблизно на 20% у порівнянні з іншими, внаслідок наявності SDK й бібліотек, та підвищило якість системи.
- Спроєктована архітектура з використанням загальноприйнятих принципів і практик допомогла зробити якісний застосунок.
- Система має гарний потенціал для майбутнього розвитку.
- Розроблена система може бути досить конкурентоспроможною, особливо в порівнянні зі стандартним UI OpenStack.

Національний технічний університет України
«Київський політехнічний інститутімені Ігоря Сікорського»

National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute"