

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЗАПОРІЗЬКА ПОЛІТЕХНІКА»

Факультет комп'ютерних наук та технологій
Кафедра комп'ютерних систем та мереж

Пояснювальна записка

до дипломного проєкту (роботи)

магістра

(ступінь вищої освіти)

на тему **ВИСОКОПРОДУКТИВНА СИСТЕМА КЕШУВАННЯ ДАНИХ В
РОЗГАЛУЖЕНІЙ МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ**

Виконав: студент 2 курсу, групи КНТ-512м
спеціальності _____

123 Комп'ютерна інженерія

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Комп'ютерні системи та мережі

БРАТЧИКОВ В.В.

(ПРИЗВИЩЕ та ініціали)

Керівник КИРИЧЕК Г.Г.

(ПРИЗВИЩЕ та ініціали)

Рецензент КОЗІНА Г.Л.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет Комп'ютерних наук і технологій

Кафедра «Комп'ютерні системи та мережі»

Ступінь вищої освіти магістерський

Спеціальність 123 Комп'ютерна інженерія

(код і найменування)

Освітня програма (спеціалізація) Комп'ютерні системи та мережі

(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Зав. кафедри Кудерметов Р.К.



“ ” 2023 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА

БРАТЧИКОВА Володимира Володимировича

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Високопродуктивна система кешування даних в розгалуженій мікросервісній архітектурі

керівник проєкту (роботи) КИРИЧЕК Галина Григорівна к. т. н., доцент,

(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом вищого навчального закладу від “ 27 ” жовтня 2023 року №

2. Строк подання студентом проєкту (роботи) 10 грудня 2023 року

3. Вихідні дані до проєкту (роботи) архітектура системи кешування даних і доведення її переваги перед класичним централізованим варіантом

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1) Аналіз технічного завдання, дослідження наявних систем кешування даних;

2) Технології, методи та алгоритми роботи системи;

3) Реалізація основних модулів системи;

4) Тестування системи та дослідження функціональності її параметрів.

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

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада Консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4	КИРИЧЕК Г. Г., к. т. н., доцент		
нормоконтроль	ПОЛЬСЬКА О.В., ст. викл.		

7. Дата видачі завдання 01.10.2023 р.

КАЛЕНДАРНИЙ ПЛАН

	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Аналіз наявних систем для кешування даних, перегляд можливостей конфігурування, виділення недоліків та переваг	10.10.2023 р.	
2	Реалізація тестового стенду	15.10.2023 р.	
3	Реалізація архітектури системи	20.10.2023 р.	
4	Реалізація системи з синхронізацією кеш нод	05.11.2023 р.	
5	Дослідження системи асинхронного розподіленого кешування	20.11.2023 р.	
6	Оформлення отриманих результатів у ПЗ	25.11.2023 р.	
7	Оформлення графічного матеріалу	01.12.2023 р.	
8	Оформлення допоміжного матеріалу	10.12.2023 р.	

Студент  **Володимир БРАТЧИКОВ**
(підпис) (ім'я, ПРИЗВИЩЕ)

Керівник проєкту (роботи)  **Галина КИРИЧЕК**
(підпис) (ім'я, ПРИЗВИЩЕ)

РЕФЕРАТ

ПЗ: 76 с.; 21 рис; 9 табл; 29 джерел; 4 додатки.

DOCKER, MEMCACHED, MAVEN, REDIS, SPRING, TOKEN RING, КЕШУВАННЯ, КЛАСТЕР, ОПТИМІЗАЦІЯ

Об'єктом дослідження є процес реалізації високопродуктивної системи кешування даних для розгалуженої мікросервісної архітектури з використанням архітектури token ring та інструментів синхронізації між нодами. Предметом дослідження є моделі, методи та засоби автоматизації кешування даних у розгалуженій мікросервісній архітектурі.

Мета роботи дослідження методів та засобів автоматизації кешування даних у розгалуженій мікросервісній архітектурі та реалізація системи автоматизованого кешування для веб-платформ з метою впровадження етапів кешування даних в розгалуженій мікросервісній архітектурі. Система реалізована з використанням архітектури token ring та має механізми синхронізації між вузлами.

Для створення моделі системи та реалізації кешування обрано фреймворк Spring Boot та інструмент Maven у середовищі IntelliJ IDEA. Процес розгалуженого кешування виконаний за допомогою Maven, Docker та із використанням власного реалізованого механізму кешування та синхронізації даних для розгалуженої мікросервісної архітектури.

Проведено дослідження продуктивності системи, в порівнянні з існуючими рішеннями на основі аналізу швидкісних характеристик використання механізму кешування. Реалізована високопродуктивна система кешування даних в розгалуженій мікросервісній архітектурі істотно підвищує якість послуг, які дозволяють зберігати та отримувати доступ до даних швидше, зменшуючи навантаження на сервери та оптимізуючи ресурси.

ABSTRACT

Explanatory note to the master's work: 76 p., 21 figures; 9 tables; 29 sources; 4 appendices

CACHING, CLUSTER, DOCKER, MEMCACHED, OPTIMIZATION, REDIS, TOKEN RING

The object of investigation in this project is the implementation of a high-performance data caching system for a branched microservices architecture using the token ring architecture and synchronization tools between nodes. The subject of the study is the methods and means of automating data caching in a branched microservices architecture.

The goal of this diploma project is to design and implement an automated caching system for a web platform to study all aspects of data caching in a branched microservices architecture. The system is implemented using the token ring architecture and has synchronization mechanisms between nodes.

To create the project structure and implement caching, the Spring Boot framework and Maven tool in the IntelliJ IDEA environment were chosen. The process of branched caching is performed using Maven, Docker, and the implementation of a custom caching and data synchronization mechanism for a branched microservices architecture.

The performance of the system was investigated by comparing it with existing solutions based on an analysis of the speed characteristics of caching mechanism usage. The research results provide a significant contribution to understanding the advantages and limitations of using the implemented data caching system in a branched microservices architecture.

ЗМІСТ

Вступ.....	7
1 Аналіз технічного завдання та дослідження наявних методів кешування	9
1.1 Основні поняття та визначення систем збереження та кешування даних	9
1.2 Аналіз існуючих систем та архітектур в області кешування даних	10
1.3 Постановка завдань	15
2 Технології, методи та алгоритми роботи системи.....	16
2.1 Апаратні та програмні вимоги до системи.....	16
2.2 Технології та методи	18
2.3 Вибір мови програмування та бібліотек.....	23
3 Реалізація основних модулів системи	29
3.1 Логічна модель системи.....	29
3.2 Опис програмних модулів реалізованої системи	31
3.3 Опис програмних модулів для тестування реалізованої системи.....	43
3.4 Опис реалізованої інфраструктури	44
4 Тестування системи та дослідження функціональності її параметрів	52
4.1 Дослідження результатів виконання тестів	52
4.2 Аналіз результатів	60
Висновки.....	65
Перелік джерел посилання	67
Додаток А Зміст pom.xml.....	69
Додаток Б Зміст класу CacheClientServiceImpl.java	71
Додаток В Зміст класу CacheClientController.java.....	73
Додаток Г Зміст класу CacheServiceImpl.java.....	75

ВСТУП

Сучасний світ інформаційних технологій відзначається надзвичайною динамікою розвитку та нескінченним потоком даних, що потребують обробки та надійного збереження. Переваги мікросервісної архітектури, яка дозволяє будувати розподілені та масштабовані системи, незаперечні. Проте разом із зростанням масштабів додатків і величезним потоком даних виникає необхідність в ефективному управлінні цими даними та доступі до них. Важливою складовою для забезпечення високої продуктивності в мікросервісній архітектурі є система кешування даних.

Системи кешування даних визначаються як критичний аспект в забезпеченні швидкодії мікросервісних додатків, особливо в умовах високого навантаження та надзвичайно швидкозмінних даних. Кешування дозволяє зберігати та доступатися до даних швидше, зменшуючи навантаження на сервери та оптимізуючи ресурси.

Мета даної дипломної роботи полягає у вивченні, аналізі та реалізації високопродуктивної системи кешування даних, спеціально призначеної для використання в розгалуженій мікросервісній архітектурі. В рамках цього дослідження буде розглянуто основні принципи та підходи до кешування даних, реалізовано та налаштовано систему кешування, та проведено тестування її продуктивності в реальних умовах.

У результаті цієї роботи очікується створення ефективної системи кешування даних, яка допоможе підвищити продуктивність та масштабованість мікросервісних додатків, що стане важливим внеском у розвиток сучасних інформаційних систем.

У першому розділі досліджено та проаналізовано технічні вимоги та існуючі методи кешування даних для отримання повного розуміння предметної області..

У другому розділі розглянуто технологічний стек та методи, які будуть використовуватися для впровадження системи кешування даних в розгалуженій мікросервісній архітектурі..

У третьому розділі предсталено деталі реалізації ключових компонентів системи кешування даних та її логічну модель.

У четвертому розділі наведені результати дослідження системи та її переваги та недоліки у порівнянні з вже існуючими фреймворками.

1 АНАЛІЗ ТЕХНІЧНОГО ЗАВДАННЯ ТА ДОСЛІДЖЕННЯ НАЯВНИХ МЕТОДІВ КЕШУВАННЯ ДАНИХ

1.1 Основні поняття та визначення систем збереження та кешування

Системи збереження та кешування даних відіграють важливу роль у поліпшенні продуктивності та швидкості доступу до інформації [1].

Ми оглянемо базові терміни, які використовуються в цьому контексті, розглянемо ключові особливості кожної з цих технологій, і надаємо загальний огляд того, як вони допомагають вирішити проблеми швидкодії та доступу до даних. Зрозуміння цих понять є важливим для розробників та архітекторів, які прагнуть вдосконалити свої додатки та системи, забезпечити їх ефективність і надійність.

Розглянемо базові терміни кешування в мікросервісних архітектурах [2].

Кеш - це тимчасове сховище для даних, яке зберігається в оперативній пам'яті або на швидкодіючих носіях. Головною метою кешування є зменшення часу доступу до даних, забезпечення швидкодії та зменшення навантаження на основну систему збереження даних.

Ключ-значення (Key-Value) - це структура збереження даних, де кожен об'єкт (зазвичай називається ключем) має відповідне значення. Доступ до даних відбувається за ключем, що дозволяє швидко знаходити та оновлювати інформацію.

Термін дії (TTL - Time To Live) - вказує на термін, протягом якого дані можуть бути збережені в кеші. Після закінчення терміну дії, дані автоматично видаляються з кешу.

Розподілене сховище даних (Distributed Data Store) - це система, яка розподіляє дані по кількох вузлах чи серверах. Це забезпечує високу доступність, стійкість до відмов та можливість горизонтального масштабування для забезпечення швидкодії.

Інтерактивні запити та аналітика - це можливість виконувати складні запити до даних, включаючи запити SQL-подібного формату та аналіз даних для отримання корисної інформації.

Масштабованість (Scalability) - вказує на здатність системи рости зі збільшенням навантаження. Це може бути горизонтальна масштабованість, коли додаються нові сервери, або вертикальна масштабованість, коли покращують продуктивність на існуючих серверах [3].

Типова схема роботи кеш систем показана на рисунку 1.1.

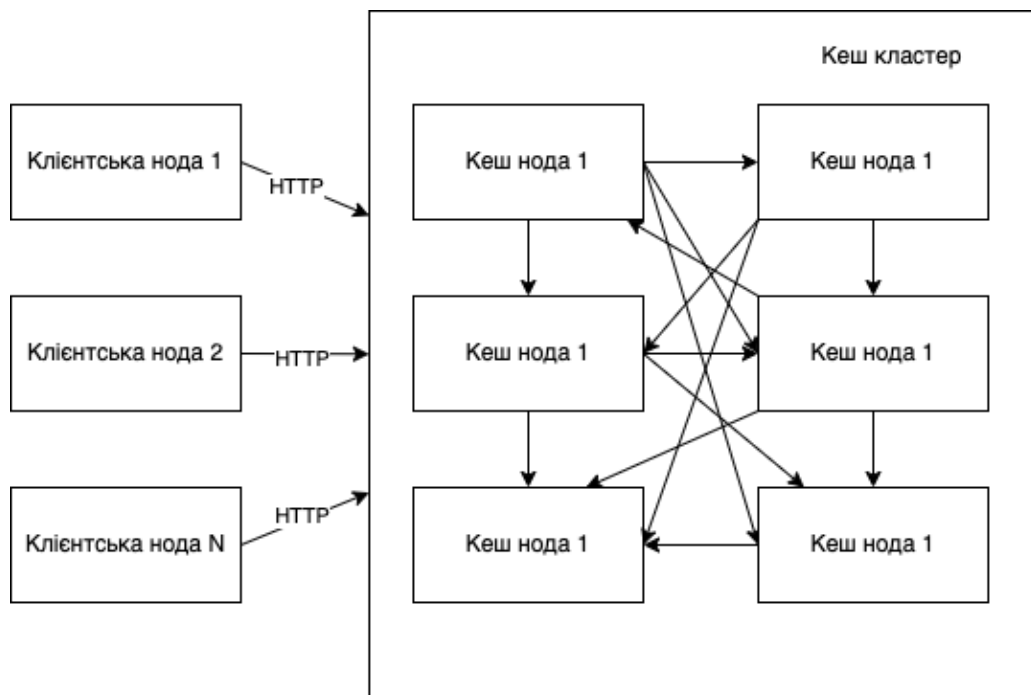


Рисунок 1.1 – Типова схема роботи кеш системи

1.2 Аналіз існуючих систем та архітектур в області кешування даних

В цьому розділі ми розглянемо основні поняття та визначення, пов'язані із такими системами, включаючи Redis, Apache Ignite та Memcached.

Redis (Remote Dictionary Server) є дистриб'ютованою системою кешування та збереження даних у пам'яті, яка використовує ключ-значення для збереження

та доступу до інформації [4]. Redis - це високопродуктивна система збереження даних, яка використовується як ключ-значення сховище. Redis відомий своєю високою швидкістю та простотою використання. Опис архітектури роботи Redis представлено нижче.

Ін-Memory Сховище Redis - це система, яка зберігає дані в оперативній пам'яті. Це робить Redis надзвичайно швидкодіючою, оскільки доступ до даних відбувається без необхідності читати їх з диска.

Redis використовує клієнт-серверну архітектуру. Redis-сервер - це окремий процес, який слухає запити від клієнтів та виконує їх. Клієнти можуть бути написані на різних мовах програмування та взаємодіяти з Redis через мережу.

Ключ-Значення Сховище Redis зберігає дані у форматі ключ-значення, де кожен ключ пов'язаний з певним значенням. Це дозволяє клієнтам швидко знаходити та оновлювати дані.

Redis підтримує різноманітні типи даних, включаючи рядки, хеші (асоціативні масиви), списки, множини та сортовані множини. Це дозволяє зберігати та обробляти різноманітні дані в Redis.

Ефективне Кешування Даних. Redis часто використовується для кешування даних. Це означає, що дані зберігаються в Redis на певний термін, після чого вони можуть автоматично видалятися. Це допомагає покращити продуктивність та зменшити навантаження на основну систему збереження даних.

Розподілені Операції [5]. Redis підтримує ряд розподілених операцій, таких як інкремент, декремент, блокування та багато інших. Це робить Redis потужним інструментом для реалізації різних додатків та систем.

Система Підписки та Повідомлення. Redis має підтримку публікації/підписки (Pub/Sub), що дозволяє клієнтам слідкувати за подіями та сповіщеннями, що відбуваються в системі.

Горизонтальне Масштабування. Redis підтримує горизонтальне масштабування шляхом розподілу даних на кілька серверів. Це дозволяє

обробляти великі обсяги даних та забезпечувати високу доступність та безпеку. Основні поняття Redis включають:

- ключ-значення (key-value) сховище: Redis використовує структуру ключ-значення для збереження даних у швидкодіючій пам'яті. Кожен ключ пов'язаний з певним значенням, яке може бути рядком, числом або іншими типами даних;
- типи даних: Redis підтримує різноманітні типи даних, включаючи рядки, хеші, списки, множини та сортовані множини, що дозволяє зберігати та обробляти різноманітні дані;
- експірація даних: в Redis можна встановити термін дії для кожного ключа, що дозволяє автоматично видаляти застарілі дані з кешу.

Apache Ignite є розподіленою системою для збереження даних, яка поєднує в собі можливості кешування та обчислення. Apache Ignite використовується для покращення швидкодії, доступності даних та розподілених обчислень.

Кешування даних. Apache Ignite використовує ін-меморі кеш для зберігання часто використовуваних даних в оперативній пам'яті. Це допомагає покращити швидкодію доступу до даних та зменшити навантаження на основну систему збереження даних, таку як бази даних.

Розподілене сховище даних. Apache Ignite може слугувати як розподілене сховище даних. Дані розподіляються по вузлах кластера, що забезпечує високу доступність та стійкість до відмов. Дані можуть бути збережені як ключ-значення, де ключ - це унікальний ідентифікатор для доступу до даних.

Сервер-клієнт архітектура. Apache Ignite використовує архітектуру клієнт-сервер, де клієнти взаємодіють з серверами для доступу до даних та виконання розподілених операцій.

Розподілені операції та обчислення. Apache Ignite надає можливість виконувати розподілені операції та обчислення. Це означає, що ви можете виконувати операції над даними, розподіленими по кластеру, та навіть запускати розподілені обчислення, які можуть бути використані для аналізу даних або інших завдань.

Підтримка SQL-подібних запитів. Apache Ignite підтримує SQL-подібні запити для доступу до даних, що зроблює його досить зручним для розробників, звиклих до реляційних баз даних.

Інтерактивні запити та аналітика. Apache Ignite дозволяє виконувати складні запити та аналітику над даними, розподіленими по кластеру.

Масштабованість: Apache Ignite дозволяє горизонтально масштабувати систему шляхом додавання нових вузлів до кластера. Це дозволяє обробляти великі обсяги даних та забезпечувати високу доступність.

Архітектура Apache Ignite [6] робить його потужним інструментом для реалізації додатків, які вимагають швидкодії доступу до даних, розподілених операцій та аналітики. Вона дозволяє покращити продуктивність та надійність додатків, що обробляють великі обсяги даних.

Основні поняття Apache Ignite включають: розподілені збережені дані: Apache Ignite дозволяє розподіляти дані по кластеру, забезпечуючи високу доступність та стійкість до відмов:

- кешування даних: завдяки кешуванню даних в пам'яті, Apache Ignite покращує швидкодію доступу до інформації та використовується для прискорення операцій;
- інтерактивні запити та аналітика: Apache Ignite дозволяє виконувати складні SQL-запити та аналітику над даними, що розподілені по кластеру.

Memcached (Memory Cache Daemon) [7] - це система кешування даних, яка базується на розподіленій пам'яті. Архітектура роботи Memcached - це проста та ефективна система кешування даних, яка використовується для прискорення доступу до інформації. Memcached відомий своєю швидкістю та легкістю використання. Наведемо опис архітектури роботи Memcached.

Ін-Memory кеш. Memcached - це ін-меморі кеш, що означає, що дані зберігаються в оперативній пам'яті. Це дозволяє Memcached надзвичайно швидко доступатися до даних, оскільки немає необхідності читати дані з диска.

Сервер-клієнт архітектура. Memcached використовує клієнт-серверну архітектуру. Memcached-сервер - це окремий процес, який слухає запити від

клієнтів та виконує їх. Клієнти можуть бути написані на різних мовах програмування та взаємодіяти з Memcached через мережу.

Ключ-значення сховище. Memcached зберігає дані у форматі ключ-значення, де кожен ключ пов'язаний з певним значенням. Це дозволяє клієнтам швидко знаходити та оновлювати дані.

Ефективне кешування даних. Головна функція Memcached - це кешування даних. Дані зберігаються в Memcached на певний термін, після чого вони можуть автоматично видалятися. Це допомагає покращити продуктивність та зменшити навантаження на основну систему збереження даних.

Розподілена система. Memcached може бути розподіленою системою, що дозволяє горизонтально масштабувати кеш для більшої продуктивності. Кілька серверів Memcached можуть працювати разом, і клієнти можуть автоматично вибирати сервер для збереження та отримання даних.

Термін дії (TTL - Time To Live): Memcached дозволяє встановити термін дії для кожного ключа, що вказує, скільки часу дані будуть зберігатися в кеші перед автоматичним видаленням.

Горизонтальне масштабування. Для підвищення продуктивності та доступності можна додавати нові сервери Memcached та розподіляти навантаження між ними. Основні поняття Memcached включають:

- кешування в оперативній пам'яті. Memcached зберігає дані в оперативній пам'яті, що дозволяє швидко доступатися до них без необхідності обчислення чи запиту до основного сховища даних;
- ключ-значення сховище. Як і Redis, Memcached використовує ключ-значення для збереження даних. Ключем може бути якийсь текст. Значенням може бути будь-який формат даних але серіалізований в текст, наприклад JSON, XML або файли представлені у форматі Base64;
- розподілена система. Memcached може бути розподіленою системою, що дозволяє горизонтально масштабувати кеш для більшої продуктивності.

1.3 Постановка завдань

Для реалізації системи кешування найкраще використовувати перевірені та надійні мови програмування та фреймворки а також використовувати досвід існуючих систем.

Для реалізації подібної системи автор повинен добре розумітись в побудові архітектур та знати декілька мов програмування, наприклад Java, Javascript, а також вміти конфігурувати кластер з серверів, наприклад використовуючи docker-compose.

Виділимо основні задачі для реалізації:

- реалізація логічної та функціональної схем системи;
- аналіз наявних систем для кешування даних;
- реалізація тестового стенду;
- реалізація архітектури системи;
- реалізація системи з синхронізацією кеш нод;
- дослідження переваг та недоліків реалізованої системи.

Основними перевагами систем, що впроваджуються, в порівнянні з наявними на ринку продуктами повинні бути:

- зменшення затримки в операціях синхронізації, підвищення швидкості доступу до даних;
- підтримка інструменту. Оскільки підтримка здійснюється внутрішнім персоналом компанії, це забезпечуватиме необхідну оперативність у вирішенні проблем та у здійсненні необхідних опрацювань;
- відкритість. Планується реалізація роботи з вихідними кодами на засадах вільного програмного забезпечення;
- мова програмування та підтримка ОС. Мовою розробки фреймворку обрано мову Java, як найбільш поширену на сьогодні по ряду індексів. Таким чином, вирішується проблема зі складністю потенційного пошуку нових

розробників. Аналогічно це впливає на потенційну кількість розробників/тестувальників, які будуть користувачами фреймворку.

2 ТЕХНОЛОГІЇ, МЕТОДИ ТА АЛГОРИТМИ РОБОТИ СИСТЕМИ

2.1 Апаратні та програмні вимоги до системи

Для запуску системи знадобиться мінімум 3 ком'ютери або сервери з мінімальною конфігурацією або один сервер з налаштованими віртуальними машинами та віртуальною мережею з можливістю керування затримками. Кожна операційна система повинна мати JVM та підключення до локальної мережі.

Мінімальна необхідна конфігурація для Windows:

- процесор – Intel Core I3 або аналоги до наявності інструкцій SSE2;
- оперативна пам'ять – від 4 Гб (рекомендується 8 Гб);
- наявність місця на диску – мінімум 2 Гб (рекомендується щонайменше 10 Гб за необхідності тривалого зберігання звітів, логів, додатків, скріншотів та відеозаписів).

Мінімальна необхідна конфігурація для Mac OS:

- Mac OS X 10.5 та вище;
- оперативна пам'ять – від 2 Гб (рекомендується 4 Гб);
- наявність місця на диску – мінімум 2 Гб та не менше 1 Гб для кешу.

Мінімальна необхідна конфігурація для Mac Linux OS:

- GNOME или KDE;
- оперативна пам'ять – від 2 Гб (рекомендується 4 Гб);
- наявність місця на диску – мінімум 2 Гб та не менше 1 Гб для кешу.

Для реалізації роботи було використано середовище IntelliJ IDEA [8].

Воно є одним з найрозповсюдженіших у світі. Сьогодні IntelliJ IDEA має гарний та інтуїтивний інтерфейс, який дозволяє розібратися навіть новачкові. Насамперед команда сконцентрувалася на створенні засобів для рефакторингу та

аналізу. Розвиваючись у цьому напрямі, девелопери змогли реалізувати товар, у якому код пишеться частково вами, а частково машинним інтелектом. А все починалося з того, щоб забезпечити утилітарні функції: комфортне перейменування класів, методів, автоматичне визначення та інші. Інтерфейс програми зображений на рисунку 2.1.

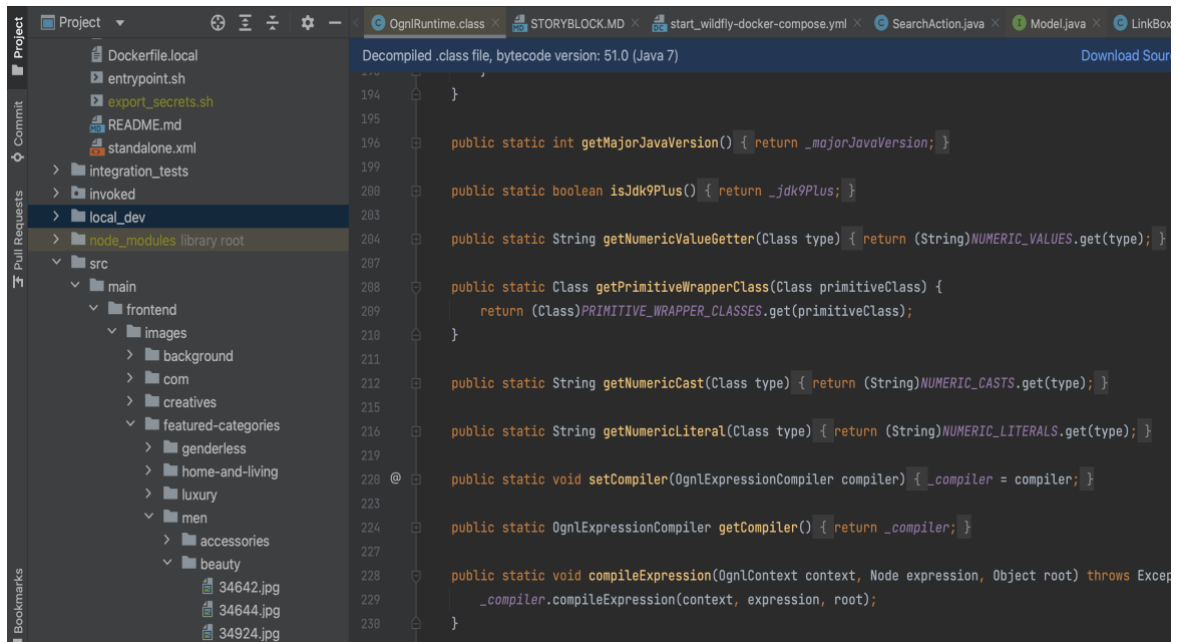


Рисунок 2.1 – Зображення інтерфейсу IntelliJ IDEA

IntelliJ IDEA, під час реалізації та виконання коду, може аналізувати його, виявляти помилки та запропонувати належні рішення. Наприклад, IDEA може будувати синтаксичне дерево навіть тоді, коли ви ще лише набираєте код. Аналізуючи посилання та шляхи виконання програми, IDE пропонує варіанти того, як код може бути доповнений. Після того, як ви адаптуєтеся до такого ритму, стає досить важко повертатися до безкоштовних програм.

Звичайно, автодоповнення - не унікальна функція. Подібні можливості доступні практично в кожній безкоштовній програмі. Причому вони реалізовані на досить високому рівні. Однак інтелект, як завжди, робить IntelliJ IDEA відмінним. Наприклад, ви пишете код, який система аналізує і готова надати варіанти для автодоповнення. Там, де звичайна програма запропонує кілька

варіантів, IntelliJ IDEA видає лише один, але вірний. Це може зекономити лише декілька секунд, але разом це робить процес реалізації більш оптимізованим

Інший важливий інструмент - це графічний редактор для створення інтерфейсів. Це набагато зручніше, ніж писати шаблонний код вручну. Девелопер може просто перетягувати елементи інтерфейсу на робочу область, в то час як код генерується автоматично. Деякі досвідчені програмісти можуть вважати це підходом для новачків, але фактично це ефективний спосіб заощадити багато часу, при цьому якість коду залишається високою. Тому візуальні редактори використовують не лише початківці, але і ті, хто просто прагне оптимізувати свою роботу.

2.2 Технології та методи

Технологія кешування даних є ключовою в інформаційних системах для забезпечення швидкодії та ефективності. Ця техніка включає в себе збереження копій даних у швидкодійних сховищах, таких як оперативна пам'ять, для подальшого доступу до них без необхідності повторного запитування первинних джерел інформації.

Існує кілька методів кешування даних, які можуть бути використані в залежності від конкретних потреб та характеристик системи [9].

Кешування в пам'яті. Дані зберігаються безпосередньо в оперативній пам'яті, що забезпечує найшвидший доступ до них. Ідеально для швидкодії кешування, особливо для часто використовуваних або обчислювально витратних даних.

Дискове кешування. Дані зберігаються на диску, забезпечуючи персистентність, але з меншою швидкістю доступу порівняно з кешуванням в пам'яті. Зручно для обсягових даних або тих, які не часто використовуються.

Кешування в базі даних. Деякі бази даних підтримують вбудовані механізми кешування для покращення доступу до даних. Для оптимізації частого доступу до даних з бази даних.

Кешування на рівні звернення до API. Дані кешуються на рівні викликів до зовнішніх API, зменшуючи кількість фактичних звернень. Ефективно для складних запитів або зовнішніх викликів, що можуть бути витратними.

Кешування CDN. Використовується для кешування статичних та динамічних вмісту на рівні Content Delivery Network (CDN). Забезпечує швидкий доступ до контенту для користувачів у різних регіонах світу.

Кешування на рівні браузера. Деякі дані можуть бути кешовані на рівні самого браузера користувача для зменшення часу завантаження сторінок. Підходить для ресурсів, які можна кешувати на бічному браузері, такі як зображення, стилі або скрипти.

Кешування на рівні сервера. Деякі сервери можуть використовувати внутрішні механізми кешування для покращення продуктивності. Для швидкодії доступу до обчислень або результатів обробки на рівні сервера.

В мікросервісних архітектурах існує необхідність синхронізувати дані між нодами. Існує декілька базових алгоритмів.

Алгоритм Лівайна (Leveldb) для LSM-дерев. Використовується в розподілених системах, де дані зберігаються у вигляді LSM-дерев (Log-Structured Merge Tree). Алгоритм використовує підходи мультиверсійності для оптимізації читання та письма в розподіленому середовищі. Основні переваги - висока продуктивність. LSM-дерева забезпечують швидкий доступ до даних, що робить цей алгоритм ефективним для розподілених систем з великим обсягом даних. Використання мультиверсійності дозволяє оптимізувати одночасне читання та запис, знижуючи конфлікти. Основні недоліки - складність реалізації та потреба пам'яті. Реалізація LSM-дерев вимагає додаткових зусиль через їхню складність порівняно з іншими структурами даних. LSM-дерева можуть використовувати значну кількість оперативної пам'яті для ефективної роботи.

Алгоритм Шутдайна (CRDT). Conflict-Free Replicated Data Types (CRDT) є класом алгоритмів, які дозволяють синхронізувати дані без необхідності вирішення конфліктів. Це особливо корисно для систем, де можуть виникати асинхронні зміни. Основні переваги - відсутність конфліктів. CRDT дозволяє уникнути конфліктів при синхронізації даних, що полегшує реалізацію та управління розподіленими системами. Недоліки - обмеженість видів даних, розмір даних. Деякі типи даних можуть бути складні для вираження у вигляді CRDT, що обмежує їхню універсальність. Деякі CRDT можуть потребувати зберігання додаткової метаінформації, що може збільшувати обсяг даних.

Алгоритм Vector Clocks. Використовується для визначення порядку подій в розподіленому середовищі. Кожна нода має свій вектор часу, який використовується для визначення, які зміни вже були враховані в системі. З переваг - це визначення порядку подій та зменшення конфліктів. Vector Clocks дозволяють чітко визначати порядок подій в розподіленій системі. Дозволяє уникати конфліктів при визначенні часового порядку подій. А це все призводить до зростання розміру векторів. Зі збільшенням кількості нод у системі зростає розмір векторів, що може призводити до великих обсягів даних та збільшення навантаження на мережу. Ще один недолік - складність в реалізації, вимагає точного управління часом та синхронізації між нодами.

Алгоритм PACELC. Заснований на теорії CAP (Consistency, Availability, Partition Tolerance). PACELC ставить акцент на те, як система поводить себе під час виникнення розбиття мережі (Partition) і в яких умовах вона обирає між доступністю і консистентністю. Використовує концепцію кворуму для синхронізації даних. При записі або оновленні даних потрібно досягти кворуму нод для підтвердження операції. Надає можливість вибору між доступністю, консистентністю та обмеженістю у випадку розбиття мережі. Система може адаптуватися до змін у стані мережі та середовища. Недоліки: вибір між компромісами, складність прийняття рішень. Вибір між доступністю, консистентністю та обмеженістю залежить від конкретних умов, що може

викликати складнощі в управлінні системою. Рішення вибору між компромісами може бути складним і залежить від конкретних вимог системи.

Алгоритм Gossip Protocol. Використовується для обміну інформацією між нодами в мережі. Кожна нода періодично обмінюється даними з іншими нодами, щоб усунути розбіжності в стані кешу. Основна перевага - автоматична розповсюдження інформації.

Протокол Gossip автоматично розповсюджує інформацію між нодами, що полегшує синхронізацію. Основний недолік – затримка. Може виникати затримка в поширенні інформації, оскільки вона залежить від інтервалів оновлення.

Ці переваги і недоліки можуть варіюватися в залежності від конкретних вимог та контексту використання системи.

Кешування на рівні мережі. Деякі кешувальні пристрої можуть бути розташовані на рівні мережі для зменшення трафіку та покращення продуктивності. Зручно для покращення ефективності обміну даними між мікросервісами.

Технічно, система кешування даних може бути побудована за різними архітектурними підходами. Існують централізовані та розподілені системи кешування. У розподіленому кешуванні, розподілені ноди кешуються, що дозволяє зберігати кеш на багатьох серверах для підвищення доступності та розділення навантаження.

Для спілкування між додатками та системою кешування використовуються спеціалізовані протоколи, такі як Redis Protocol, Memcached Protocol, або RESTful APIs [10]. Ці протоколи дозволяють додаткам здійснювати операції читання, запису, оновлення та видалення даних у кеші.

Важливою частиною системи кешування є стратегії валідації даних. Наприклад, час життя даних у кеші, механізми валідації та оновлення, а також управління обсягом кешованих даних. Технологія кешування також може включати в себе можливості реплікації, для забезпечення високої доступності, або розподілення кешованих даних між серверами.

Алгоритм роботи між клієнтською ногою і кешем, може бути розглянутий наступним чином.

Клієнтська нода встановлює TCP/IP з'єднання [11] з кеш-сервером за допомогою IP адреси та порта кеш-сервера. Клієнтська нода та кеш-сервер обмінюються командами та даними через певний протокол. Найпоширенішими протоколами є Redis Protocol або Memcached Protocol. Клієнтська нода відправляє запити до кеш-сервера, що можуть бути такими:

- GET key. Клієнт запитує дані для конкретного ключа;
- SET key value. Клієнт записує дані з визначеним ключем та значенням;
- DEL key. Клієнт видаляє дані для конкретного ключа;
- інші команди для роботи зі структурами: хеші, списки, множини.

Кеш-сервер отримує запити та виконує їх. Якщо запит GET [12], то сервер перевіряє, чи існують дані для вказаного ключа у своєму кеші. Якщо дані є та актуальні, вони повертаються клієнту. Якщо дані відсутні або застарілі, сервер [13] повертає відповідну відсутність даних або оновлює кеш і повертає нові дані.

Інші команди виконують відповідні операції на даних. Кеш-сервер відправляє результати обробки запитів назад клієнтській ноді. Це може бути дані, відсутність даних або інша відповідь.

Кеш-сервер може включати в себе механізми для встановлення та керування термінами дії кешованих даних, автоматичного оновлення даних, механізми валідації, інвалідації та реплікації для забезпечення актуальності та доступності даних.

Після виконання запитів та отримання результатів, клієнтська нода може закрити з'єднання [14] з кеш-сервером.

Усі ці технічні аспекти дозволяють створювати потужні системи кешування даних, які використовуються для підвищення продуктивності та зниження навантаження [15] на інші джерела даних, і вони є ключовими в сучасних інформаційних системах, включаючи мікросервісні архітектури.

2.3 Вибір мови програмування та бібліотек

Для реалізації системи було проаналізовано 3 популярних мови програмування: Java, Python, C#. До уваги були взяті 4 основні критерії: швидкодія, екосистема, підтримка розподілених систем, масштабованість та підтримка операцій.

Швидкодія:

- java відома своєю високою швидкістю завдяки компіляції в байт-код та виконанню на віртуальній машині Java (JVM). Це дозволяє отримати високий рівень продуктивності;
- python, як інтерпретована мова, зазвичай менш швидка в порівнянні з Java. Проте існують JIT-компілятори, такі як PyPy, які можуть поліпшити швидкість;
- C# використовує виконання на платформі .NET, що зазвичай забезпечує високу швидкість. Іноді C# може бути швидшим за Java завдяки оптимізаціям .NET.

Екосистема:

- java має розгалужену екосистему з багатьма фреймворками та бібліотеками для розподілених систем, обчислень, кешування даних та багато інших завдань;
- python володіє розширеною бібліотекою сторонніх модулів, але вона може бути менш розгалуженою, коли мова йде про розподілені системи;
- C# має велику екосистему на платформі .NET для реалізації різноманітних додатків, включаючи серверні рішення та кешування.

Підтримка розподілених систем:

- java має багато фреймворків та інструментів для реалізації розподілених систем, таких як Apache ZooKeeper, Apache Kafka, та інші;
- python також має бібліотеки для розподілених систем, але їхні можливості можуть бути меншими в порівнянні з Java;

- C# може використовувати багато фреймворків .NET, які добре підходять для реалізації розподілених систем.

Масштабованість та підтримка операцій:

- java має високий рівень масштабованості та підтримку для обчислень у багатьох потоках. Можна використовувати бібліотеки, такі як Akka, для підтримки акторного моделю обчислень;

- python також підтримує багато потоків, але може бути менш підходящим для масштабованих систем;

- C# має хорошу підтримку многозадачності та паралелізму завдяки багатьом інструментам на платформі .NET.

Враховуючи переваги в швидкості можливості масштабування та наявність великої кількості бібліотек була обрана мова програмування Java. Додаткові особливості, переваги Java [16] і деталі її архітектури наведені нижче.

Віртуальна машина Java (JVM). Java використовує віртуальну машину (JVM) для виконання свого коду. JVM відповідає за трансляцію байт-коду Java (поточна мова програми після компіляції) в нативний машинний код конкретної операційної системи. Ця процедура, відома як JIT (Just-In-Time) компіляція, дозволяє забезпечити високу продуктивність та швидкодію Java-додатків.

Автоматичне управління пам'яттю. Java використовує систему автоматичного управління пам'яттю, відому як "збірка сміття" (Garbage Collection) [17]. Замість того, щоб девелопери вручну виділяти та звільняти пам'ять для об'єктів, JVM автоматично визначає та видаляє об'єкти, які більше не використовуються. Це допомагає уникнути витоку пам'яті та полегшує роботу з пам'яттю.

Багатопотоковість. Java має вбудовану підтримку багатопотоковості, що дозволяє одночасно виконувати багато задач в межах одного додатку. Це досягається завдяки пакету `java.util.concurrent`, який надає девелоперам інструменти для створення та керування потоками, а також синхронізації їх роботи.

Об'єктно-орієнтований підхід. Java базується на об'єктно-орієнтованому підході до програмування, де все є об'єктами. Кожен об'єкт має свої властивості та методи, що робить код більш структурованим та легко зрозумілим. ООП дозволяє створювати більш абстрактний та модульний код.

Велика стандартна бібліотека. Java має велику стандартну бібліотеку, включаючи пакети для роботи з мережами, базами даних, графікою, безпекою, XML-парсингу, роботи з файлами та іншими аспектами реалізації. Девелопери можуть використовувати ці бібліотеки для створення програм з різними функціональностями, що значно скорочує час реалізації та полегшує інтеграцію з різними технологіями.

Платформонезалежність. Однією з ключових переваг Java є платформонезалежність. Байт-код Java може виконуватися на будь-якій платформі, яка має відповідну реалізацію JVM. Це робить Java ідеальною мовою для впровадження крос-платформових додатків. Реалізації JVM існують для багатьох операційних систем, що дає можливість запускати Java-додатки практично скрізь.

Велика спільнота і підтримка. Java має велику та активну спільноту девелоперів, а також широку підтримку від компаній та організацій. Це забезпечує наявність різних ресурсів, бібліотек та інструментів для девелоперів, а також оновлення і поліпшення мови та інфраструктури.

Робота з мережами. Java надає потужні інструменти для роботи з мережами [18]. Бібліотека `java.net` містить класи для створення мережових з'єднань, обміну даними через протоколи, такі як HTTP, TCP та UDP, та для створення мережових серверів. Наприклад, ви можете легко реалізувати HTTP-клієнт або сервер за допомогою класів `URLConnection` або `HttpServer`. Java також підтримує мережеві операції на низькому рівні за допомогою класів, які надають доступ до сокетів. Це дозволяє розробникам створювати мережеві додатки, які взаємодіють з іншими додатками через мережу, передачу даних та комунікацію з веб-серверами. Зокрема, бібліотека `java.net` містить класи для роботи з URL (Uniform Resource Locator), які дозволяють створювати, зчитувати та взаємодіяти з

ресурсами в Інтернеті. Завдяки цьому, Java є популярним інструментом для реалізації мережевих додатків, включаючи браузер, поштові клієнти та веб-служби.

Безпека. Питання безпеки в мережевих додатках має велике значення. Java надає багато інструментів для захисту мережевих додатків, включаючи підтримку шифрування даних та аутентифікації. Бібліотеки, такі як Java Cryptography Extension (JCE) та Java Authentication and Authorization Service (JAAS), дозволяють розробникам забезпечувати високий рівень безпеки у мережевих додатках. Java також має вбудовану підтримку SSL (Secure Sockets Layer) та TLS (Transport Layer Security) для захисту мережевих з'єднань. Це важливо для забезпечення конфіденційності та цілісності даних, що передаються через мережу.

Реалізація веб-додатків. Java є популярною мовою для реалізації веб-додатків, особливо за допомогою технологій, таких як JavaServer Pages (JSP) та Java Servlets. Ці технології дозволяють створювати динамічні веб-сторінки та оброблювати HTTP-запити на сервері. Крім того, існують різноманітні фреймворки для реалізації веб-додатків на Java, такі як Spring, JavaServer Faces (JSF), та Play Framework.

Ці фреймворки полегшують створення веб-додатків та надають розробникам потужні інструменти для роботи з мережами. Усі ці можливості роблять Java потужним і відомим інструментом для реалізації мережевих додатків, незалежно від їхнього обсягу і складності. Додатки, зроблені на Java, можуть легко взаємодіяти з іншими системами через мережу, надійно передавати дані та забезпечувати безпеку комунікацій.

RESTful веб-служби. Java також широко використовується для реалізації RESTful веб-служб, що є популярним підходом до створення веб-додатків, що взаємодіють через HTTP-протокол. Розробники можуть використовувати бібліотеки, такі як JAX-RS (Java API for RESTful Web Services) для створення і обробки REST-запитів та відповідей. Це дозволяє створювати масштабовані [19] та легко розширювані веб-служби.

Мережеві фреймворки. Java має численні мережеві фреймворки, які допомагають розробникам створювати мережеві додатки з високою продуктивністю та надійністю. Деякі приклади цих фреймворків включають Netty, Grizzly та Apache MINA. Вони надають різноманітні інструменти для роботи з сокетами, обробки мережевих подій та засоби створення мережевих серверів [19].

Мережеві безпекові аспекти. Java також має багато засобів для забезпечення безпеки мережевих додатків, включаючи валідацію та фільтрацію вхідних даних, обробку аутентифікації та авторизації користувачів, захист від атак на мережеві протоколи та багато іншого. Безпека мережевих додатків - важлива складова реалізації в сучасному інтернет-середовищі, і Java надає розробникам інструменти для створення безпечних мережевих рішень [20]. В цілому, Java має багато можливостей для роботи з мережами, починаючи від створення мережевих додатків до реалізації веб-служб та забезпечення їх безпеки. Ці технічні можливості додають до універсальності та потужності Java як мови програмування для мережевих додатків.

Архітектура Java наділена безліччю технічних особливостей, які роблять її потужним і універсальним інструментом для реалізації програмного забезпечення.

В якості основного фреймворку обраний Spring Boot [21]. Нижче приведене порівняння Spring Boot з конкурентами.

Порівняння Spring Boot з Java EE (Jakarta EE):

- spring boot надає велику кількість зручних інструментів та бібліотек для реалізації Java-додатків. Він спрощує розгортання та конфігурацію додатків та надає багато функцій, таких як інверсія керування (IoC), аспектно-орієнтоване програмування (AOP) та інші;
- java EE (раніше відома як J2EE) - це стандарт для реалізації підприємних додатків на Java. Вона також надає різноманітні специфікації та інструменти для реалізації великих підприємних додатків. Проте Java EE може бути складнішим у використанні та громіздким у порівнянні з Spring Boot.

Порівняння Spring Boot з Micronaut:

- spring boot - це зрілий фреймворк, який надає велику кількість готових рішень та розширень. Він має велику спільноту і активно підтримується;
- micronaut - це новий фреймворк, який спеціалізується на мікросервісах та інтернеті речей (IoT). Він славиться низьким споживанням пам'яті та високою швидкістю завдяки анотаційному процесу компіляції.

Порівняння Spring Boot з Quarkus:

- spring boot добре підходить для створення традиційних Java-додатків та має велику кількість інтеграцій з іншими технологіями;
- quarkus - це фреймворк, спрямований на оптимізацію для гібридних та хмарних додатків. Він славиться низьким часом завантаження та високою швидкістю завдяки компіляції в нативний код.

Нижче наведено більше технічних деталей щодо архітектури Spring Framework.

Шари Spring Framework [22]:

- шар презентації (Presentation Layer): у цьому слої розташовані веб-контролери, які обробляють HTTP-запити та генерують відповіді. Spring MVC є одним із ключових модулів для роботи з цим шаром;
- шар бізнес-логіки (Business Logic Layer): цей слой містить бізнес-логіку додатку. Spring дозволяє вам визначити біни (компоненти) бізнес-логіки та управляти їх життєвим циклом. У цьому контексті Spring Core (Spring IoC) використовується для впровадження залежностей і управління бінами;
- шар доступу до даних (Data Access Layer): Spring Framework надає підтримку для взаємодії з різними системами управління базами даних (JDBC, Hibernate, JPA), а також для роботи з транзакціями. Тут ключовими є модулі Spring Data та Spring JDBC;
- шар інфраструктури (Infrastructure Layer): у цьому слої розташовані різноманітні служби та інфраструктурні функції. Наприклад, Spring AOP (Aspect-Oriented Programming) дозволяє застосовувати аспекти до коду для реалізації різних функцій, включаючи безпеку та журналювання.

IoC контейнер - це серце Spring Framework. Він відповідає за управління компонентами додатку та їх життєвим циклом. Spring IoC реалізує принцип інверсії управління, що означає, що не об'єкти запитують та створюють свої залежності, а контейнер програмного забезпечення надає їм ці залежності.

Spring використовує внедрення залежностей для надання об'єктам їх необхідних залежностей. Це полегшує керування залежностями між об'єктами і дозволяє змінювати компоненти без необхідності модифікації коду.

Spring Framework підтримує модульність додатків, дозволяючи розділити його на декілька підпроектів, кожний зі своєю функціональністю. Конфігурація може бути виконана за допомогою XML-файлів, анотацій або Java-коду.

Spring Framework надає підтримку аспектно-орієнтованого програмування (AOP), що дозволяє вам визначати аспекти, які можна застосовувати до декількох об'єктів. Це особливо корисно для реалізації аспектів безпеки, журналювання та транзакцій.

Враховуючи популярність фреймворку Spring Boot, його документацію, ком'юніті, відносну простоту інтеграції, обрано саме цей фреймворк.

3 РЕАЛІЗАЦІЯ ОСНОВНИХ МОДУЛІВ СИСТЕМИ

3.1 Логічна модель системи

Система кешування даних між мікросервісами використовує вдосконалений алгоритм синхронізації для ефективного управління і підтримки актуальності даних між різними кеш-нодами. Основна мета цього алгоритму полягає в зниженні затримок, пов'язаних із мережею, і забезпеченні консистентності інформації, розподіленої між різними складовими системи. Алгоритм реалізований таким чином, щоб забезпечити ефективне керування синхронізацією даних і уникнути надмірної завантаженості мережі. Це сприяє підвищенню ефективності системи кешування та забезпечує мінімізацію

затримок у витягуванні актуальних даних. Такий підхід дозволяє системі оперативно реагувати на зміни в даних та забезпечує їх узгодженість в усьому середовищі мікросервісної архітектури [23]. Основні кроки алгоритму включають такі дії:

- визначення сусідніх нод. Кожна кеш-нода в системі знає, які ноди є її сусідами в топології token ring. Це дозволяє кожній ноді знаходити своїх безпосередніх сусідів для обміну даними та оновлень. Інформація про сусідні ноди зберігається в налаштуваннях ноди.;
- ініціалізація синхронізації. Кожна кеш-нода періодично ініціює процес синхронізації зі своїми сусідніми нодами. Це може відбуватися за певним розкладом чи після певної кількості запитів на оновлення;
- обмін даними. Кеш-нода відправляє запит на синхронізацію своїх даних сусідній ноді. Запит містить інформацію про оновлені дані та час останньої синхронізації. Також в запиті знаходиться версія кешованого об'єкта що являє собою порядковий номер, наприклад при створенні нового запису версія буде 0, при наступному оновленні версія буде 1 і т д.;
- підтвердження та оновлення. Сусідня нода приймає запит на синхронізацію, перевіряє дані та порівнює їх з власними. Якщо виявляються розходження, сусідня нода надсилає оновлені дані та підтвердження, що призведе до процесу синхронізації;
- оновлення та актуалізація. Кеш-нода оновлює свої дані на основі отриманих від сусідньої ноди. Цей процес допомагає підтримувати актуальність даних на кожній кеш-ноді та забезпечує їхню консистентність. Після оновлення даних на одній кеш-ноді, вони можуть бути розповсюджені через мережу за топологією token ring. Це означає, що дані поступово синхронізуються між всіма нодами в системі. Процес синхронізації повторюється для кожної кеш-ноди з її сусідніми нодами. Це дозволяє зменшити затримку мережі та підтримувати актуальність даних між усіма нодами в системі.

Логічна схема системи зображена на рисунку 3.1

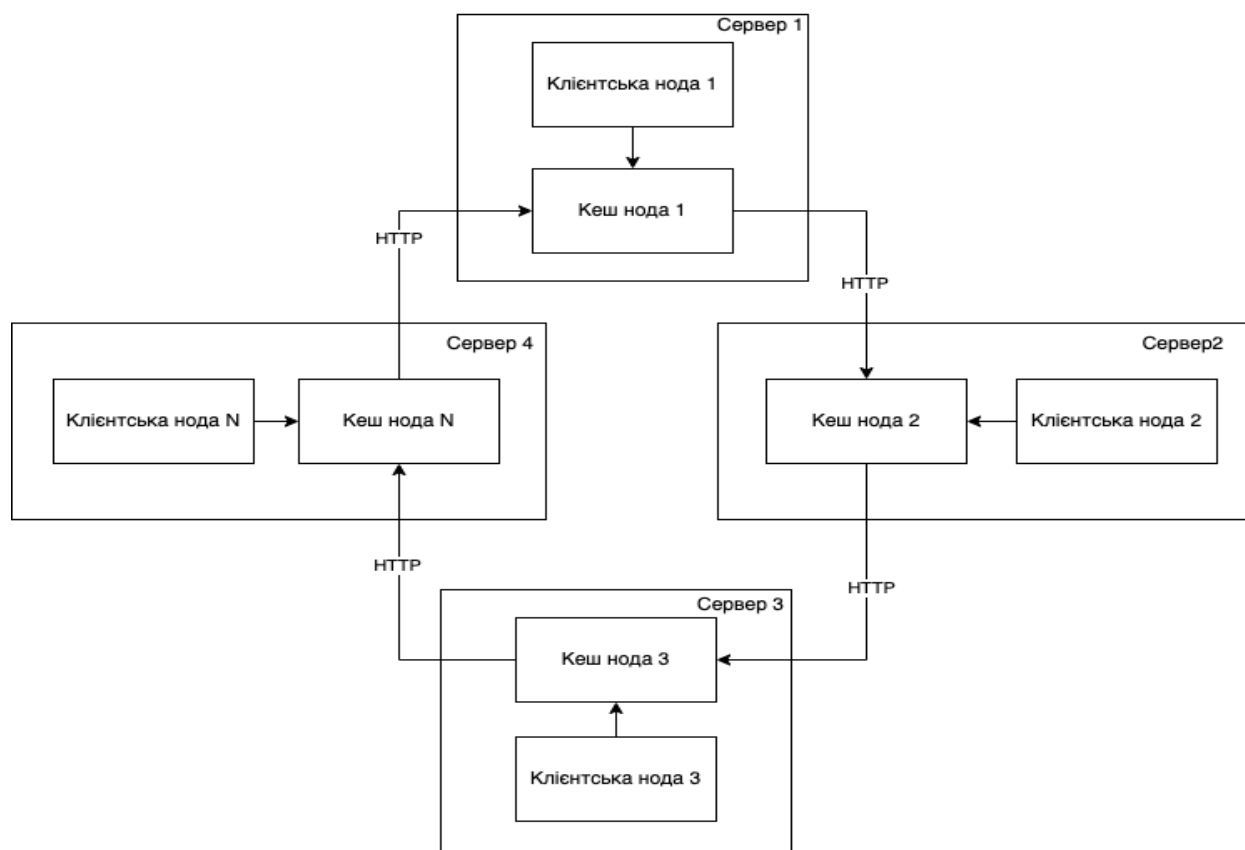


Рисунок 3.1 – Логічна схема системи

3.2 Опис програмних модулів реалізованої системи

Оскільки робота будується на основі Java, Spring Boot та використовує архітектуру MVC, нижче наведений приклад структури модулів роботи. Схема модулів зображена на рисунку 3.2.

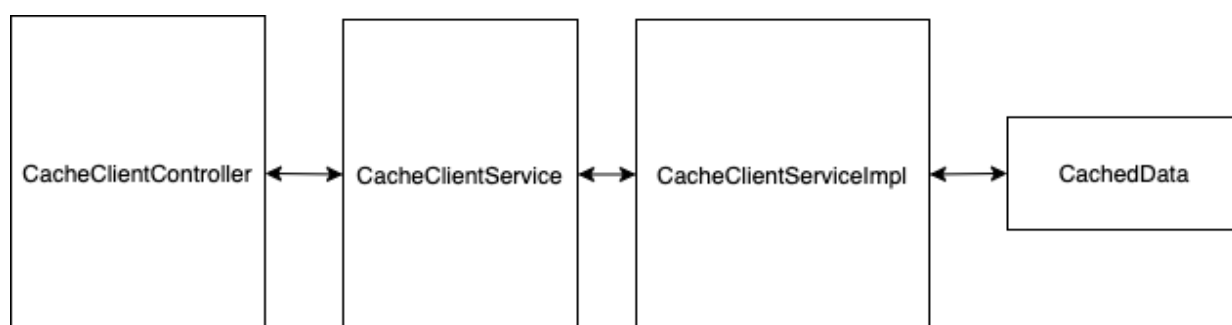


Рисунок 3.2 – Логічна схема модулів підсистеми 'клієнтська нода'

CacheClientController.java: представляє собою клас контролера у контексті веб-застосунку, відповідальний за обробку HTTP-запитів та взаємодію з клієнтами системи. Цей клас становить важливий елемент управління кешем і відповідає за ефективне отримання, збереження та оновлення кешованих даних.

Основна функціональність CacheController включає обробку запитів, які надходять від клієнтів системи через протокол HTTP. Зокрема, контролер може обробляти запити на отримання (GET), збереження (POST) та оновлення (PUT) кешованих даних. Це робить його ключовим компонентом для взаємодії між клієнтами та кеш-системою.

В контексті отримання даних, контролер приймає GET-запити від клієнтів, спрямовуючи їх до відповідного сервісу чи компонента системи, який відповідає за роботу з кешем. Відповідно до логіки програми, в контексті POST-запитів, контролер відповідає за збереження нових даних у кеші. Щодо PUT-запитів, він координує оновлення кешованих даних, забезпечуючи актуальність інформації для клієнтів.

Ключова відповідальність контролера полягає в перехопленні HTTP-запитів та перенаправленні їх до відповідних сервісів чи компонентів, відповідальних за роботу з кешем. Це дозволяє контролеру бути посередником між клієнтами та системою кешування, забезпечуючи коректний обмін даними та оптимальну роботу кеш-механізму (лістинг 3.1).

Лістинг 3.1 - Лістинг реалізації

```
Public class CacheClientController

@RestController
@RequiredArgsConstructor
@RequestMapping("client")
public class CacheClientController {

    private final CacheClientService cacheClientService;
```

```

@Value("${cache.node.force.delay}")
private Long forceDelayTime;

@GetMapping
public Collection<CachedData> getAll() {
    return cacheClientService.getAll();
}

@sneakyThrows
@GetMapping("/{id}")
public CachedData get(@PathVariable String id,
                      @RequestParam(name="useDelay",
defaultValue = "false") boolean useDelay) {
    if(useDelay) {
        sleep(forceDelayTime);
    }
    return cacheClientService.get(id);
}

@sneakyThrows
@PostMapping
public String post(@RequestBody String data,
                  @RequestParam(name="useDelay",
defaultValue = "false") boolean useDelay) {
    if(useDelay) {
        sleep(forceDelayTime);
    }
    return cacheClientService.save(data);
}

@sneakyThrows
@PutMapping("/{id}")
public String put(@RequestBody String data, @PathVariable
String id, @RequestParam(name="useDelay", defaultValue = "false")
boolean useDelay) {

```

```

        if(useDelay) {
            sleep(forceDelayTime);
        }
        return cacheClientService.update(id, data);
    }
}

```

CachedData.java є класом, що визначає модель даних, призначену для кешування. У своїй суті, цей клас виступає в ролі структури даних, яка визначає, які конкретні елементи можуть бути збережені та отримані з кешу. Одна з ключових ролей CachedData - це надання зручного інтерфейсу для представлення даних у кеші, включаючи всі необхідні атрибути та їх значення (лістинг 3.2).

Цей клас містить необхідні поля для зберігання даних, таких як ключі та відповідні їм значення. Наприклад, він може включати поля, що ідентифікують конкретний запис у кеші (наприклад, унікальний ідентифікатор чи ключ), а також самі дані, які мають бути збережені.

Лістинг 3.2 - Лістинг реалізації CachedData

```

@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class CachedData {

    public static final CachedData EMPTY
        = new CachedData(null, null, null, 0);
    private String id;
    private String data;
    private LocalDateTime lastUpdateTime;
    private int versionID;

}

```

CacheClientService.java представляє собою інтерфейс, який визначає контракт для сервісу, відповідального за взаємодію з кешем в системі. Цей інтерфейс містить методи, які будь-який конкретний клас або сервіс, що реалізує його, повинен виконувати. Основна мета інтерфейсу - надати стандартні методи для роботи з кешованими даними та визначити загальний інтерфейс для всіх класів, які працюють з кешем.

Цей інтерфейс містить методи, які включають в себе операції отримання, збереження та оновлення кешованих даних.

Отримання (Get): метод для отримання даних із кешу на підставі вказаного ключа чи іншого ідентифікатора. Це дозволяє швидко та ефективно отримувати збережені дані без необхідності звертатися до основного джерела даних.

Збереження (Save): метод для збереження нових даних у кеші. Використовується для додавання нового запису чи оновлення існуючого в кеші.

Оновлення (Update): метод для оновлення існуючих даних в кеші. Важливий для того, щоб забезпечити актуальність та консистентність даних в кеші.

Інтерфейс дозволяє використовувати поліморфізм і взаємозамінювати конкретні реалізації CacheClientService без необхідності зміни коду, який його викликає (лістинг 3.3). Це сприяє гнучкості та розширюваності системи кешування, оскільки нові реалізації можуть легко додаватися або замінятися без впливу на викликаючий код.

Лістинг 3.3 - Лістинг реалізації CacheClientService

```
public interface CacheClientService {  
    Collection<CachedData> getAll();  
    CachedData get(String id);  
    String save(String data);  
    String update(String id, String data);  
}
```

CacheClientServiceImpl.java: є класом, що реалізує інтерфейс CacheService, і відповідає за конкретну реалізацію операцій з кешуванням даних в системі. Його важливою відповідальністю є забезпечення функціональності, описаної в інтерфейсі, та реалізація логіки, необхідної для взаємодії з кеш-нодами та управління кешованими даними.

Реалізація інтерфейсу CacheClientService: клас повинен надавати конкретні реалізації всіх методів, визначених у інтерфейсі CacheClientService, таких як методи для отримання, збереження та оновлення кешованих даних.

Взаємодія з кеш-нодами: клас взаємодіє з фізичними кеш-нодами або їх репрезентаціями, що може включати в себе логіку розподілених операцій, синхронізації між нодами та керуванням реплікацією даних для забезпечення стійкості та доступності системи.

Логіка кешування даних: клас містить логіку для ефективного зберігання та управління кешованими даними. Це включає в себе механізми очищення кешу, стратегії збереження та видалення даних, а також механізми керування часом життя даних в кеші.

Синхронізація та керування доступом: клас може включати в себе логіку синхронізації для уникнення колізій та забезпечення атомарності операцій над кешованими даними. Керування доступом до даних гарантує, що вони залишаються консистентними та вірними.

Інші допоміжні операції: можливі інші функції, такі як моніторинг, логування та обробка помилок, які покращують функціональність та надійність кеш-сервісу.

Загалом, CacheServiceImpl є ключовим компонентом для забезпечення ефективного та надійного кешування даних у системі, і його реалізація визначає, як система взаємодіє з кешем та забезпечує доступ до актуальних даних (лістинг

Для створення кеш-ноди, яка вміє передавати дані по запиту клієнта та синхронізуватися з сусідньою ногою за принципом token-ring, для цього потрібно створити модулі для кеш-ноди.

Схема модулів зображена на рисунку 3.3.

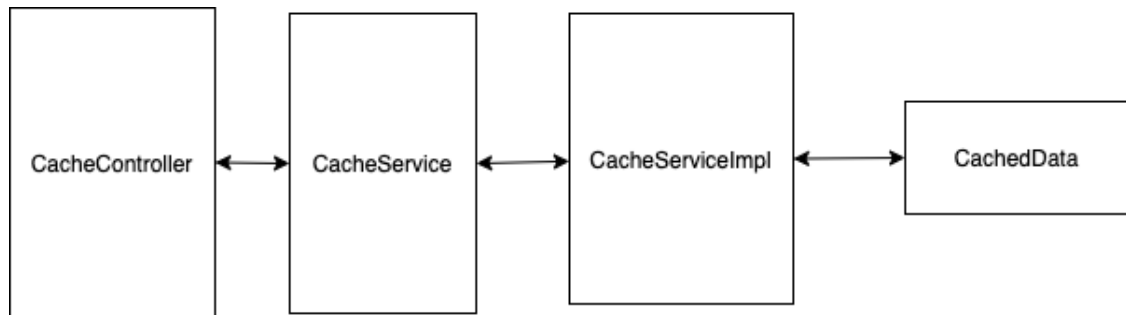


Рисунок 3.3 – Логічна схема модулів підсистеми ‘кеш нода’

Лістинг 3.4 - Лістинг реалізації CacheClientServiceImpl

```

@Service
@RequiredArgsConstructor
public class CacheClientServiceImpl implements
CacheClientService {

    private final CacheStorageService cacheStorageService;
    private final CacheService cacheService;

    @Override
    public Collection<CachedData> getAll() {
        return cacheStorageService.getData().values();
    }

    @Override
    public CachedData get(String id) {
        return cacheStorageService.getData().getOrDefault(id,
CachedData.EMPTY);
    }

    @Override
    public String save(String data) {
        String key = UUID.randomUUID().toString();
        CachedData value = CachedData.builder()
            .id(key)

```

```

        .data(data)
        .lastUpdateTime(LocalDateTime.now())
        .versionID(1)
        .build();
cacheStorageService.getData().put(key, value);
cacheService.sync(null, value);
return value.getId();
}

@Override
public String update(String id, String data) {
    CachedData cachedData
        = cacheStorageService.getData().get(id);
    if(cachedData == null) {
        throw new RuntimeException("ID " + id
+ "doesn't exists");
    }
    CachedData value = CachedData.builder()
        .id(id)
        .data(data)
        .lastUpdateTime(LocalDateTime.now())
        .versionID(cachedData.getVersionID() + 1)
        .build();
    cacheStorageService.getData().put(id, value);
    cacheService.sync(null, value);
    return value.getId();
}
}

```

CacheController.java, клас контролера відповідає за обробку HTTP-запитів, які надходять від клієнтів, та за відповідь на них. (лістинг 3.5) Він також може містити логіку для обміну даними з сусідніми нодами за принципом token-ring. Загалом, CacheController виступає як посередник між клієнтами та кеш-сервісом, забезпечуючи обробку та відповідь на HTTP-запити.

Лістинг 3.5 - Лістинг реалізації CacheControlle

```

@RestController
@RequestMapping("node")
@RequiredArgsConstructor
public class CacheController {

    private final CacheService cacheService;

    @Value("${cache.node.force.delay}")
    private Long forceDelayTime;

    @SneakyThrows
    @PostMapping("{initiator}")
    public ResponseEntity<Object> sync(@RequestBody CachedData
data, @PathVariable String initiator) {
        sleep(forceDelayTime);
        cacheService.update(initiator, data);
        return ResponseEntity.ok("");
    }
}

```

CacheStorageService.java, клас зберігає локальні дані кешу в оперативній пам'яті та надає доступ до цього сховища (лістинг 3.6).

Лістинг 3.6 - Лістинг реалізації CacheStorageService

```

@Service
@Getter
public class CacheStorageService {

    private final Map<String, CachedData> DATA
        = new HashMap<>();
}

```

CacheService.java - інтерфейс визначає методи для роботи з кешем, включаючи збереження, отримання та оновлення даних. Також, в ньому може бути визначена логіка для синхронізації з сусідніми нодами (лістинг 3.7). Інтерфейс виступає як контракт, який будь-яка конкретна реалізація повинна виконувати для правильної роботи з кешованими даними та, при необхідності, для взаємодії з іншими нодами у розподіленій системі.

Лістинг 3.7 - Лістинг реалізації CacheService

```
public interface CacheService {
    void sync(String initiator, CachedData data);
    void update(String initiator, CachedData data);
}
```

CacheServiceImpl.java, клас реалізує інтерфейс 'CacheService' та містить логіку для роботи з кешем, включаючи логіку синхронізації з сусідніми нодами за принципом token-ring (лістинг 3.8). Цей клас виступає як центральний компонент, який реалізує логіку кеш-сервісу та забезпечує правильну синхронізацію та управління даними у розподіленому середовищі.

Лістинг 3.8 - Лістинг реалізації CacheService

```
@Service
@RequiredArgsConstructor
@Slf4j
public class CacheServiceImpl implements CacheService {

    private final RestTemplate restTemplate;
    private final CacheStorageService cacheStorageService;
    @Value("${cache.node.current.host}")
    String currentHost;

    @Value("${cache.node.next.host}")
```

```

String nextHost;
@Override
public void sync(String initiator, CachedData data) {
    if(nextHost.isEmpty()) {
        return;
    }
    String url = "http://" + nextHost + ":8080" + "/node/"
+ (initiator == null ? currentHost : initiator);
    log.info("Try to sync {} with {}. Data: {}",
url, nextHost, data);
    restTemplate.postForObject(url,
data, CachedData.class);
}
@Override
public void update(String initiator, CachedData data) {
    log.info("Try to update value {}, in {}. Initiator: {}"
,data, currentHost, initiator);
    if(initiator.equals(currentHost)) {
        return;
    }
    cacheStorageService.getData().put(data.getId(), data);
    this.sync(initiator, data);
}
}

```

Для формування дистрибутиву та керування залежностями використовується фреймворк Maven (лістинг 3.9).

Лістинг 3.9 - Лістинг Maven конфігурації

```

<?xml version="1.0" encoding="UTF-8"?>
<project          xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">

```

```

<modelVersion>4.0.0</modelVersion>
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.0</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.example</groupId>
<artifactId>cache-node</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>cache-node</name>
<description>cache-node</description>
<properties>
    <java.version>17</java.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>

```

```

        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-
plugin</artifactId>

        <configuration>
            <excludes>
                <exclude>
                    <groupId>org.projectlombok</groupId>
                    <artifactId>lombok</artifactId>
                </exclude>
            </excludes>
        </configuration>
    </plugin>
</plugins>
</build>
</project>

```

3.3 Опис програмних модулів для тестування реалізованої системи

Оскільки робота будується на основі Java, Spring Boot та використовує архітектуру MVC, то і модуль для тестування та аналізу результатів також будується на таких самих компонентах, щоб зменшити ціну реалізації. Нижче наведений приклад структури модулів роботи. Схема модулів зображена на рисунку 3.4.

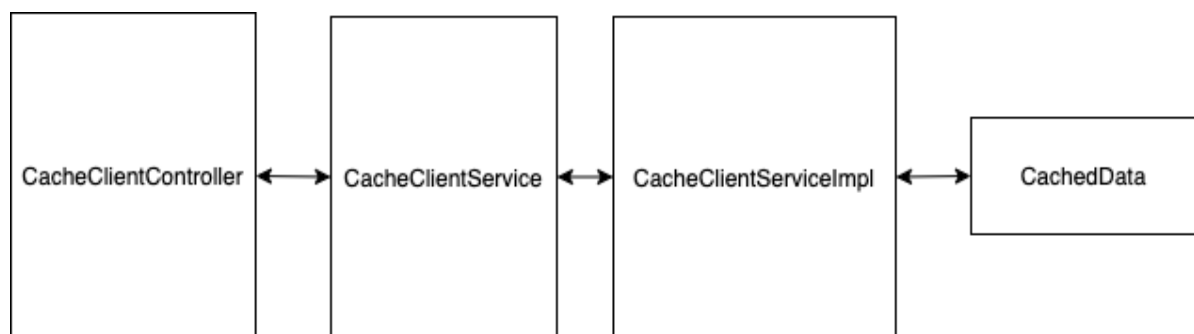


Рисунок 3.4 – Логічна схема модулів підсистеми тестування

TestApplication.java, базовий клас, з необхідними конфігураціями. CachedData.java, є класом, що визначає модель даних, призначену для кешування. У своїй суті, цей клас виступає в ролі структури даних, яка визначає, які конкретні елементи можуть бути збережені та отримані з кешу. Одна з ключових ролей CachedData - це надання зручного інтерфейсу для представлення даних у кеші, включаючи всі необхідні атрибути та їх значення.

Цей клас містить необхідні поля для зберігання даних, таких як ключі та відповідні їм значення. Наприклад, він може включати поля, що ідентифікують конкретний запис у кеші (наприклад, унікальний ідентифікатор чи ключ), а також самі дані, які мають бути збережені. Врешті решт цей клас є повною копією класа з впроваджених модулів, щоб не робити залежність модуля тестування від реалізованих модулів.

TestController.java: представляє собою контроллер для керування процесом тестування та даними та визначити загальний інтерфейс для всіх класів, які працюють з кешем.

Модуль тестування даних запускається окремо і налаштовується для кожної ноди. Цей модуль здатен проводити тестування з навантаженням, створювати багато нових записів у кеші та зчитувати їх, а також показувати середній час операцій. Для більшої гнучкості тестування та можливості аналізу даних додані спеціальні параметри для симуляції затримки мережі (ping, jitter).

3.4 Опис реалізованої інфраструктури

Для комплексної перевірки був побудований тестовий стенд, на базі docker контейнерів [24]. Всі докер контейнери об'єднанні у спільну мережу за допомогою технології docker-compose (лістинг 3.10).

Схема стенду зображена на рисунку 3.5.

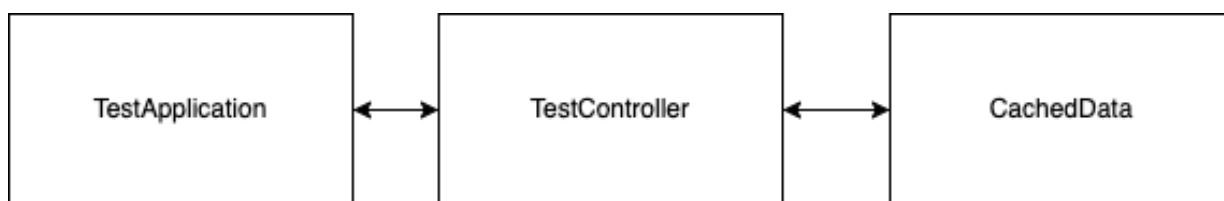


Рисунок 3.5 – схема тестового стенду

Лістинг 3.10 - Лістинг TestController

```

@RestController
@RequiredArgsConstructor
@Slf4j
public class TestController {

    private final RestTemplate restTemplate;

    @Value("${destination.host}")
    private String destHost;

    @SneakyThrows
    @GetMapping("/{type}/{count}")
    public String test(@PathVariable String type,
                      @PathVariable int count,
                      @RequestParam(name = "useDelay",
                                   defaultValue = "false")
                      boolean useDelay) {
        CountDownLatch countDownLatch=new CountDownLatch(count);
        switch (type) {

            case "GET" -> {
                List<Long> res = new ArrayList<>();
                ExecutorService service = Executors
                    .newFixedThreadPool(10);

                ResponseEntity<List<CachedData>>
                rateResponse=restTemplate.exchange(destHost"/client",

```

```

HttpMethod.GET, null,
new ParameterizedTypeReference<List<CachedData>>() {}));
List<CachedData> lis= rateResponse.getBody();
for (int i = 0; i < count; i++) {
    CachedData cachedData = list.get(i);
    if (cachedData == null) break;
    service.submit(() -> {
        String url = destHost + "/client/" + cachedData.getId()
            + (useDelay ? "?useDelay=true" : "");
        long time = System.currentTimeMillis();
        CachedData data = restTemplate
            .getForObject(url, CachedData.class);
        time=System.currentTimeMillis()- time;
        res.add(time);
        countdownLatch.countDown();
    });
}
countdownLatch.await();
return "Avg time GET = " + res.stream()
    .mapToLong(val -> val)
    .average().orElse(0.0);
}

case "POST" -> {
    List<Long> res = new ArrayList<>();
    ExecutorService service = Executors
        .newFixedThreadPool(10);
    for (int i = 0; i < count; i++) {
        String data = "New data " + i;
        service.submit(() -> {
            String url = destHost + "/client"
                + (useDelay ? "?useDelay=true" : "");
            long time= System.currentTimeMillis();
            String s = restTemplate
                .postForObject(url, data, String.class);
            time=System.currentTimeMillis()- time;

```

```

        res.add(time);
        countDownLatch.countDown();
    });
}
countDownLatch.await();
return "Avg time POST = "
        + res.stream()
            .mapToLong(val -> val)
            .average().orElse(0.0);
    }

case "PUT" -> {
    List<Long> res = new ArrayList<>();
    ExecutorService service = Executors
        .newFixedThreadPool(10);

    ResponseEntity<List<CachedData>>
        rateResponse =
            restTemplate.exchange(destHost
+ "/client", HttpMethod.GET, null,
            new ParameterizedTypeReference
<List<CachedData>>() {});

    List<CachedData> list= rateResponse.getBody();

    for (int i = 0; i < count; i++) {
        CachedData cachedData = list.get(i);
        if (cachedData == null) break;
        service.submit(() -> {
            String url = destHost + "/client/"
                + cachedData.getId()
                + (useDelay ? "?useDelay=true" : "");
            cachedData.setData("new data");
            long time= System.currentTimeMillis();
            restTemplate.put(url, cachedData);
            time=System.currentTimeMillis()- time;

```

```

        res.add(time);
        countDownLatch.countDown();
    });
}
countDownLatch.await();
return "Avg time PUT = " + res.stream()
    .mapToLong(val -> val)
    .average().orElse(0.0);
}
}
return null;
}
}

```

Конфігурація docker-compose [25] для кеш нод зображена на рисунку 3.6.

```

services:
  cachenode-1:
    build:
      context: .
      dockerfile: Dockerfile
    image: cachenode:latest
    environment:
      CACHE_NODE_CURRENT_HOST: cachenode-1
      CACHE_NODE_NEXT_HOST: cachenode-2
    ports:
      - 8081:8080
  cachenode-2:
    build:
      context: .
      dockerfile: Dockerfile
    image: cachenode:latest
    environment:
      CACHE_NODE_CURRENT_HOST: cachenode-2
      CACHE_NODE_NEXT_HOST: cachenode-3
    ports:
      - 8082:8080
  cachenode-3:
    build:
      context: .
      dockerfile: Dockerfile
    image: cachenode:latest
    environment:
      CACHE_NODE_CURRENT_HOST: cachenode-3
      CACHE_NODE_NEXT_HOST: cachenode-1
    ports:
      - 8083:8080

```

Рисунок 3.6 – Частина конфігурації docker-compose для кеш нод.

Конфігурація docker-compose (лістинги 3.11, 3.12) для кеш тестових модулів зображена на рисунку 3.7.

```

test-1:
  build:
    context: ./test
    dockerfile: Dockerfile
  image: cachetest:latest
  environment:
    DESTINATION_HOST: http://cachenode-1:8080
    SERVER_PORT: 8080
  ports:
    - 8091:8080
test-2:
  build:
    context: ./test
    dockerfile: Dockerfile
  image: cachetest:latest
  environment:
    DESTINATION_HOST: http://cachenode-2:8080
    SERVER_PORT: 8080
  ports:
    - 8092:8080
test-3:
  build:
    context: ./test
    dockerfile: Dockerfile
  image: cachetest:latest
  environment:
    DESTINATION_HOST: http://cachenode-3:8080
    SERVER_PORT: 8080
  ports:
    - 8093:8080

```

Рисунок 3.7 – Частина конфігурації docker-compose для тестових модулів.

Лістинг 3.11 - Лістинг docker-compose.yml

```

version: '2'
services:

  cachenode-1:

    build:
      context: .
      dockerfile: Dockerfile
    image: cachenode:latest
    host: node1

```

```
environment:
  CACHE_NODE_CURRENT_HOST: cachenode-1
  CACHE_NODE_NEXT_HOST: cachenode-2
ports:
  - 8081:8080
```

cachenode-2:

```
build:
  context: .
  dockerfile: Dockerfile
image: cachenode:latest
host: node2
environment:
  CACHE_NODE_CURRENT_HOST: cachenode-2
  CACHE_NODE_NEXT_HOST: cachenode-3
ports:
  - 8082:8080
```

cachenode-3:

```
build:
  context: .
  dockerfile: Dockerfile
image: cachenode:latest
host: node3
environment:
  CACHE_NODE_CURRENT_HOST: cachenode-3
  CACHE_NODE_NEXT_HOST: cachenode-1
ports:
  - 8083:8080
```

test-1:

```
build:
  context: ./test
```

```
    dockerfile: Dockerfile
  image: cachetest:latest
  host: test1
  environment:
    DESTINATION_HOST: http://cachenode-1:8080
    SERVER_PORT: 8080
  ports:
    - 8091:8080
```

test-2:

```
  build:
    context: ./test
    dockerfile: Dockerfile
  image: cachetest:latest
  host: test2
  environment:
    DESTINATION_HOST: http://cachenode-2:8080
    SERVER_PORT: 8080
  ports:
    - 8092:8080
```

test-3:

```
  build:
    context: ./test
    dockerfile: Dockerfile
  image: cachetest:latest
  host: test3
  environment:
    DESTINATION_HOST: http://cachenode-3:8080
    SERVER_PORT: 8080
  ports:
    - 8093:8080
```

Лістинг 3.12 - Лістинг Dockerfile

```
FROM amazoncorretto:17
MAINTAINER Volodymyr Bratchykov
COPY target/test-0.0.1-SNAPSHOT.jar test-0.0.1-SNAPSHOT.jar
ENTRYPOINT ["java","-jar","/test-0.0.1-SNAPSHOT.jar"]
```

4 ТЕСТУВАННЯ СИСТЕМИ ТА ДОСЛІДЖЕННЯ ФУНКЦІОНАЛЬНОСТІ ЇЇ ПАРАМЕТРІВ

4.1 Дослідження результатів виконання тестів

Для дослідження результатів швидкості роботи реалізованої системи виконаємо наступні тести [26]: тестування запису 1000 генерованих даних, тестування запису 1000 генерованих даних з штучною затримкою, зчитування 1000 тестових даних, зчитування 1000 тестових даних з штучною затримкою впровадженою в конфігурації.

Запуск буде виконуватися на кожній ноді з використанням спеціального інстансу тестового модуля. Всі тести [27] будуть запуснені на кожній ноді по 3 рази. Запуск тесту виконується за допомогою REST API. Реалізовані наступні REST запити:

- GET `http://{host}:{port}/POST/1000`, для запису 1000 генерованих даних;
- GET `http://{host}:{port}/POST/1000?useDelay=true`, для запису 1000 генерованих даних з штучною затримкою;
- GET `http://{host}:{port}/GET/1000`, для зчитування 1000 тестових даних;
- GET `http://{host}:{port}/GET /1000?useDelay=true`, для зчитування 1000 тестових даних з штучною затримкою.

Розроблені ендпоінти в якості результату повертають рзрахований середній час для обробки однієї одиниці. Розрахунок виконується за рахунок середнього арифметичного значення в мілісекундах. Ці результати будуть використовуватися для загального порівняння метрік [28]. зі швидкості доступу до даних кеш ноди [29].

На рисунку 4.1 зображено скріншот запуску кластеру.

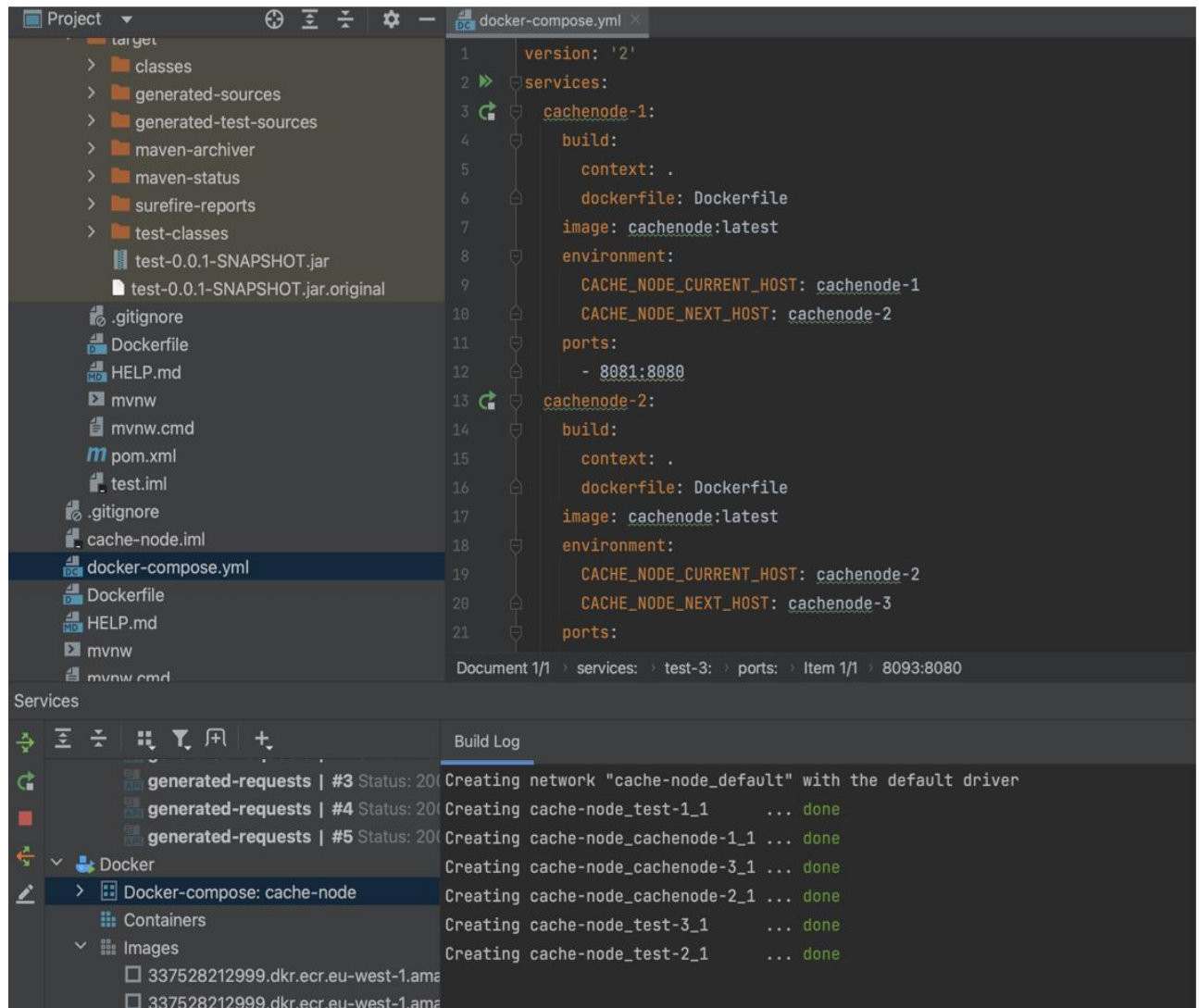


Рисунок 4.1 – Запуск кластеру використовуючи docker-compose

Після запуску кластеру стає доступною REST API. Приклад використання методів GET і POST, для запису нових даних в кеш та зчитування даних з кешу, в розробленому REST API показаний на рисунку 4.2.

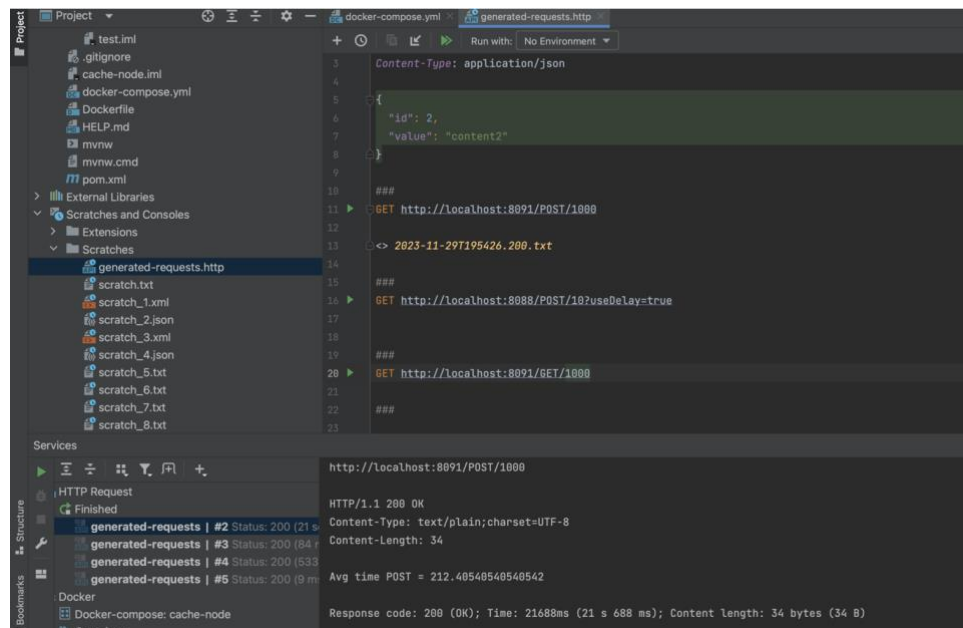


Рисунок 4.2 – Приклад використання REST API

Розберемо виконання методів для класу TestController. Схема методів наведена на рисунку 4.3.

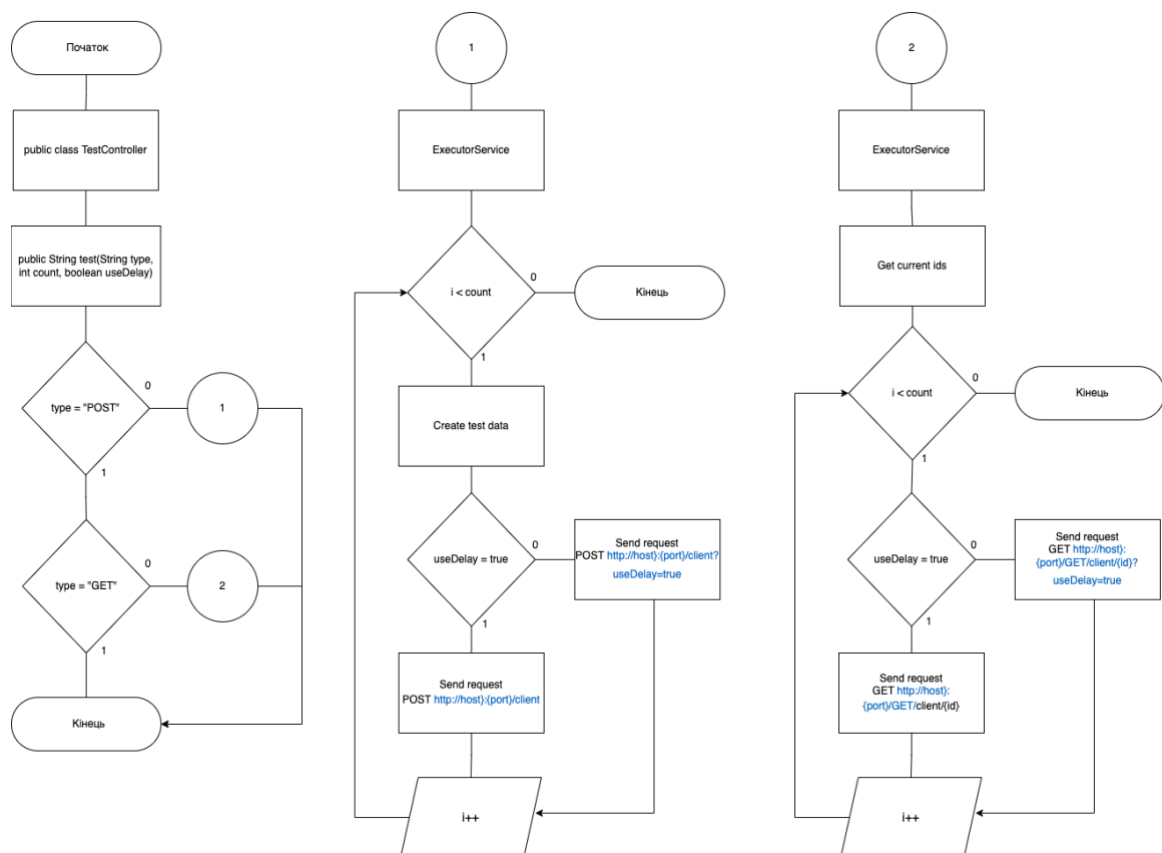


Рисунок 4.3 – Схема виконання методів тестування для класу TestController

У цьому класі для виконуються усі команди для маніпулювання тестовими даними та проведення замірів. Розберемо виконання методів для класу `CacheClientServiceImpl`. Схема методів наведена на рисунку 4.4.

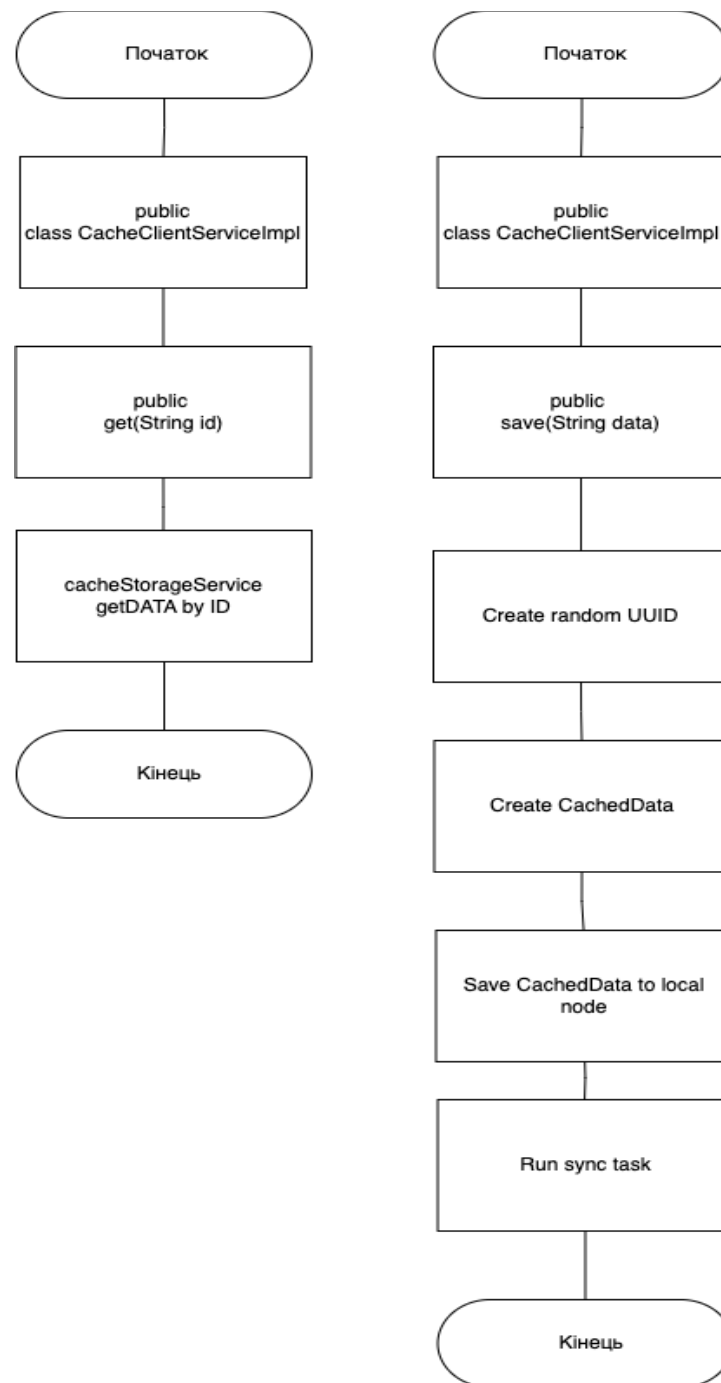


Рисунок 4.4 – Схема виконання методів тестування для класу `CacheClientServiceImpl`

Схема тестового стенду зображена на рисунку 4.5.

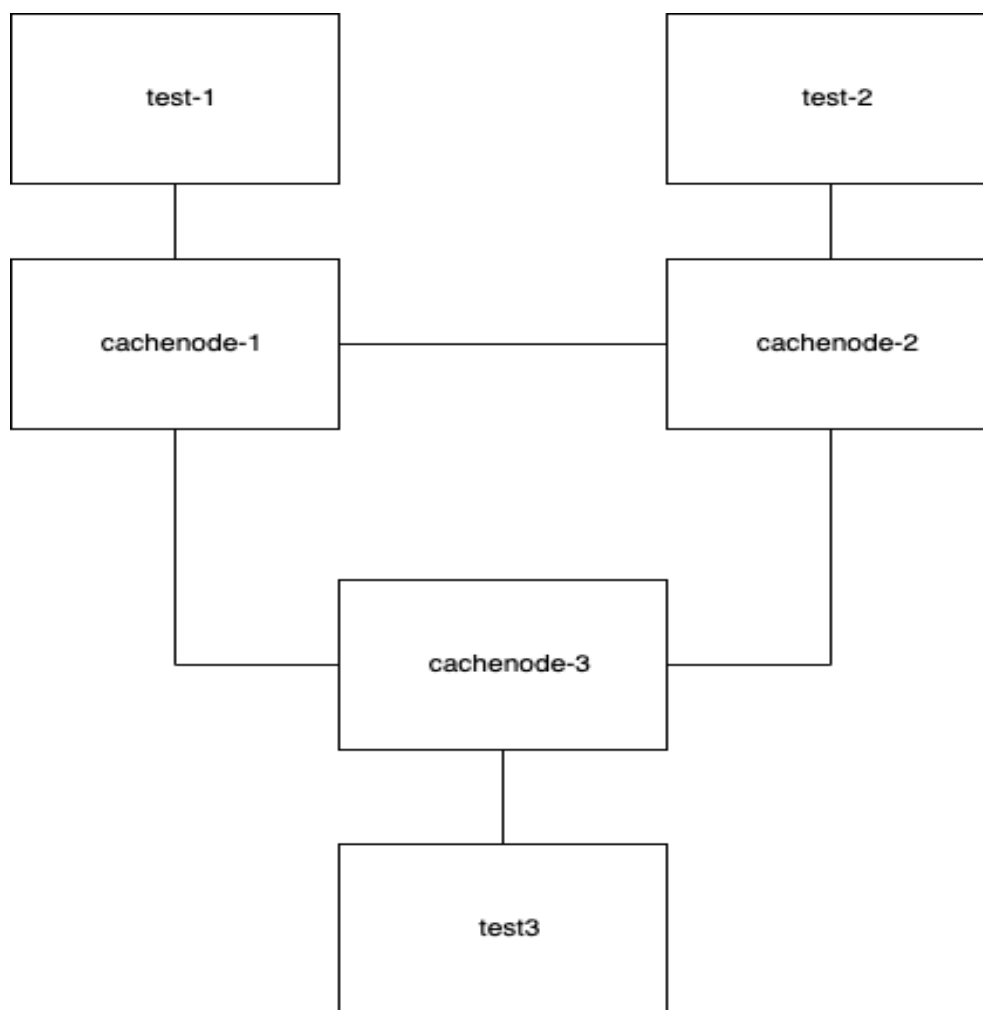


Рисунок 4.5 – Схема тестового стенду

Результат виконання команди створення 1000 випадкових записів в кеш кластері. Скріншот результату приведений на рисунку 4.6.

```
http://localhost:8091/POST/1000

HTTP/1.1 200 OK
Content-Type: text/plain;charset=UTF-8
Content-Length: 23

Avg time POST = 199.574

Response code: 200 (OK); Time: 19998ms (19 s 998 ms); Content length: 23 bytes (23 B)
```

Рисунок 4.6 – Скріншот результату виконання команди створення 1000 випадкових записів

Результат виконання команди створення одного випадкового запису в кеш кластері. Скріншот з прикладом виконання запиту та виведення результату з HTTP статусом та середнім часом операції додавання в мілісекундах приведений на рисунку 4.7.

```
http://localhost:8091/POST/1

HTTP/1.1 200 OK
Content-Type: text/plain;charset=UTF-8
Content-Length: 21

Avg time POST = 193.0

Response code: 200 (OK); Time: 201ms (201 ms); Content length: 21 bytes (21 B)
|
```

Рисунок 4.7 – Скріншот результату виконання команди створення одного випадкового запису

Треба урахувати що при виконанні операції створення виконується додаткова синхронізація даних з іншими кеш нодами. Додатково, для симуляції реальної мережі для кожної операції синхронізації додана фіксована затримка 60мс. Таким чином формула (4.1) сумарної затримки виглядає:

$$t = n * p, \quad (4.1)$$

де t – сумарна затримка, n – кількість кеш нод, p – затримка між кеш ногою i та між кеш ногою $i+1$. Також слід врахувати що в реалізації представлений в цій версії використовується синхронна операція `sync`, це означає, що операція `POST` буде завершена лише коли буде оновлена остання нода в кільці. Цей процес може бути покращений, за рахунок додавання асинхронної синхронізації нод, а також виконання агрегованих операцій. Результат виконання команди завантаження 1000 випадкових записів з кеш кластеру. Скріншот результату приведений на рисунку 4.8.

```

http://localhost:8091/GET/1000

HTTP/1.1 200 OK
Content-Type: text/plain;charset=UTF-8
Content-Length: 20

Avg time GET = 4.617

Response code: 200 (OK); Time: 811ms (811 ms); Content length: 20 bytes (20 B)

```

Рисунок 4.8 – Результат виконання команди завантаження 1000 випадкових записів

Результат виконання команди завантаження 1 випадкового запису з кеш кластеру. Скріншот результату приведений на рисунку 4.9.

```

http://localhost:8091/GET/1

HTTP/1.1 200 OK
Content-Type: text/plain;charset=UTF-8
Content-Length: 18

Avg time GET = 2.0

Response code: 200 (OK); Time: 54ms (54 ms); Content length: 18 bytes (18 B)

```

Рисунок 4.9 – Результат виконання команди завантаження 1 випадкового запису

Перевагою впровадженої системи є відсутність додаткових витрат на взаємодію з мережею при операціях зчитування. Таким чином для систем де кількість операцій зчитування значно перевищують кількість операцій додавання нових, це може бути вагомим чинником для швидкості обробки даних. Наприклад це може бути досить статичний словник, як словник країн, міст, вулиць, тощо, який створюється при ініціалізації системи але потім змінюється досить рідко.

Після виконання тестових операцій ми маємо дані для проведення дослідження швидкості взаємодії з кеш нодами, які були використані для дослідження у наступній главі.

Додатково, для порівняння результатів реалізованої системи з результатами типової системи, впроваджено реалізацію замінено технологією Redis. Схема тестового стенду на базі Redis показано на рисунку 4.10.

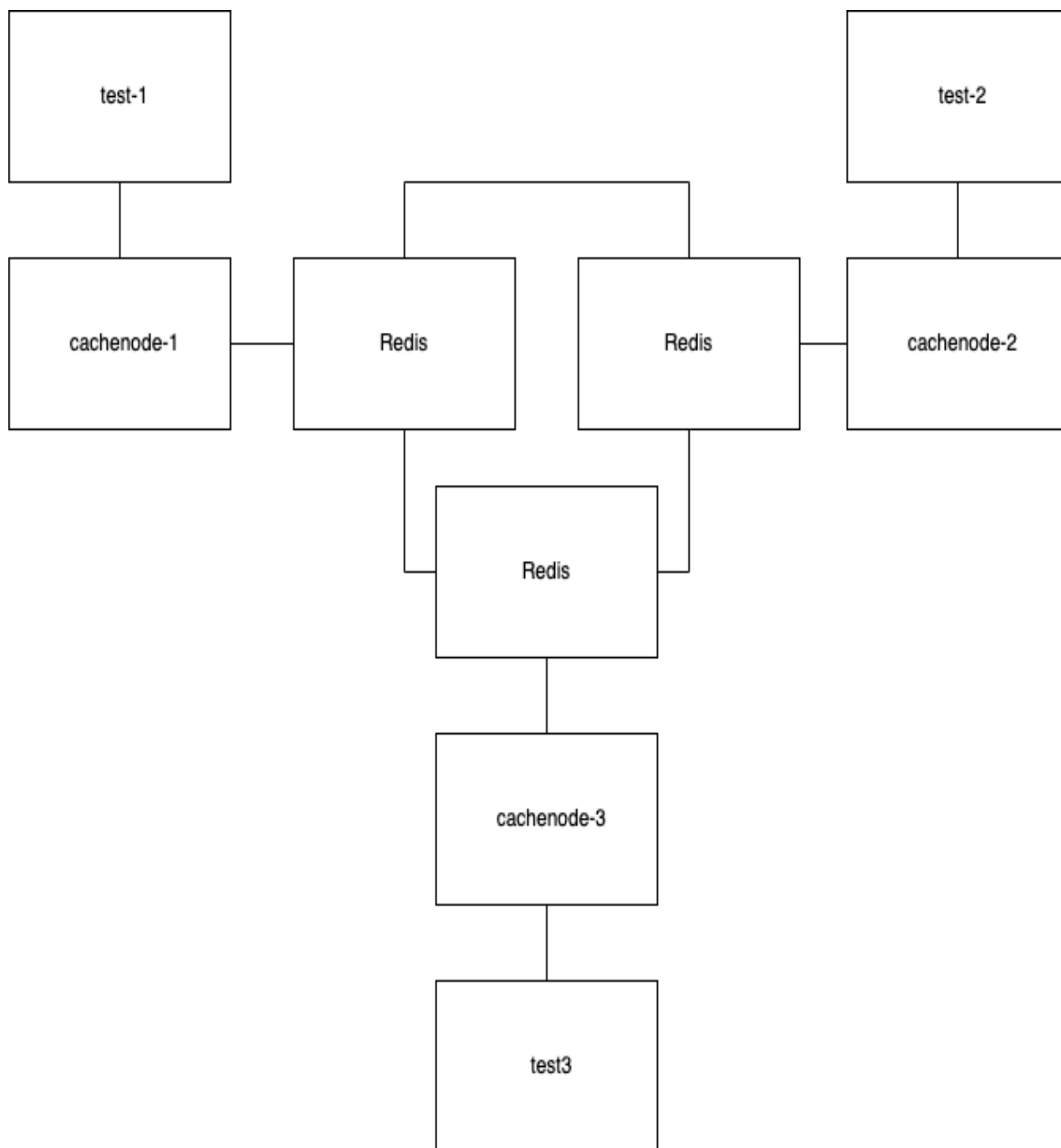


Рисунок 4.10 – Схема тестового стенду на базі Redis

4.2 Аналіз результатів

На основі проведених у попередньому розділі тестувань узагальнимо швидкісні показники систем у вигляді таблиці. Для цього проведено 3 тестування запису 1000 генерованих даних для кожної з 3х нод, 3 тестування запису 1000 генерованих даних з штучною затримкою для кожної з 3х нод, 3 зчитування 1000 тестових даних для кожної з 3х нод та 3 зчитування 1000 тестових даних з штучною затримкою для кожної з 3х нод (табл.4.1, 4.2, 4.3, 4.4).

Таблиця 4.1 – Таблиця проведення результатів з тестування запису 1000 генерованих даних

№	Нода	Середній час обробки 1 запису, мс
1	Нода 1	186
2	Нода 1	182
3	Нода 1	182
4	Нода 2	184
5	Нода 2	188
6	Нода 2	185
7	Нода 3	182
8	Нода 3	184
9	Нода 3	182

Таблиця 4.2 – Таблиця проведення результатів з тестування запису 1000 генерованих даних з штучною затримкою.

№	Нода	Штучна затримка, мс	Середній час обробки 1 запису, мс
1	Нода 1	60	241
2	Нода 1	60	244
3	Нода 1	60	250
4	Нода 2	60	243
5	Нода 2	60	246
6	Нода 2	60	244
7	Нода 3	60	342
8	Нода 3	60	346
9	Нода 3	60	248

Таблиця 4.3 – Таблиця проведення результатів з тестування зчитування 1000 тестових даних

№	Нода	Середній час обробки 1 запису, мс
1	Нода 1	2
2	Нода 1	2
3	Нода 1	3
4	Нода 2	4
5	Нода 2	2
6	Нода 2	2
7	Нода 3	3
8	Нода 3	2
9	Нода 3	4

Таблиця 4.4 – Таблиця проведення результатів з тестування зчитування 1000 тестових даних з штучною затримкою.

№	Нода	Штучна затримка, мс	Середній час обробки 1 запису, мс
1	Нода 1	60	62
2	Нода 1	60	62
3	Нода 1	60	63
4	Нода 2	60	62
5	Нода 2	60	64
6	Нода 2	60	62
7	Нода 3	60	64
8	Нода 3	60	66
9	Нода 3	60	62

Після отримання результатів впровадженої системи стенд змінено на стенд на базі Redis і автором проведені аналогічні тести.

Результати тестів кеш системи на базі Redis наведено в таблицях 4.5, 4.6, 4.7 та 4.8.

Таблиця 4.5 – Таблиця проведення результатів з тестування запису 1000 генерованих даних на базі Redis.

№	Нода	Середній час обробки 1 запису, мс
1	Нода 1	24
2	Нода 1	30
3	Нода 1	28

Продовження таблиці 4.8

4	Нода 2	40
5	Нода 2	36
6	Нода 2	38
7	Нода 3	42
8	Нода 3	48
9	Нода 3	48

Таблиця 4.6 – Таблиця проведення результатів з тестування запису 1000 генерованих даних з штучною затримкою на базі Redis.

№	Нода	Штучна затримка, мс	Середній час обробки 1 запису, мс
1	Нода 1	60	90
2	Нода 1	60	88
3	Нода 1	60	96
4	Нода 2	60	102
5	Нода 2	60	94
6	Нода 2	60	88
7	Нода 3	60	92
8	Нода 3	60	94
9	Нода 3	60	100

Таблиця 4.7 – Таблиця проведення результатів з тестування зчитування 1000 тестових даних на базі Redis.

№	Нода	Середній час обробки 1 запису, мс
1	Нода 1	10
2	Нода 1	12
3	Нода 1	8
4	Нода 2	16
5	Нода 2	14
6	Нода 2	10
7	Нода 3	16

Таблиця 4.8 – Таблиця проведення результатів з тестування зчитування 1000 тестових даних з штучною затримкою на базі Redis.

№	Нода	Штучна затримка, мс	Середній час обробки 1 запису, мс
1	Нода 1	60	78
2	Нода 1	60	72

Продовження таблиці 4.8

3	Нода 1	60	76
4	Нода 2	60	76
5	Нода 2	60	72
6	Нода 2	60	80
7	Нода 3	60	76
8	Нода 3	60	78
9	Нода 3	60	78

З метою агрегації результатів і наведення висновків, середні значення по всім операціям наведені у таблиці 4.9.

Таблиця 4.9 – Таблиця зі зведеним результатом вимірювань

№	Нода	Тип операції	Середній час (реалізація), мс	Середній час (Redis), мс
1	Нода 1	Запис	215	47
2	Нода 2	Запис	216	45
3	Нода 3	Запис	212	48
1	Нода 1	Зчитування	32	40
2	Нода 2	Зчитування	34	44
3	Нода 3	Зчитування	32	46

На рисунках 4.11 та 4.12, наведені графіки порівняння результатів запису та результатів зчитування.

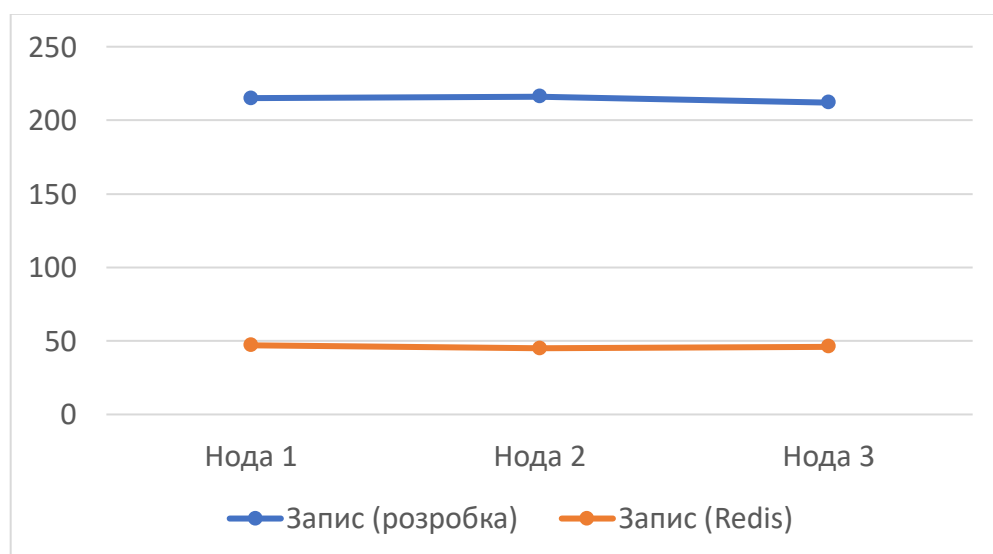


Рисунок 4.11 – Графік порівняння результатів запису

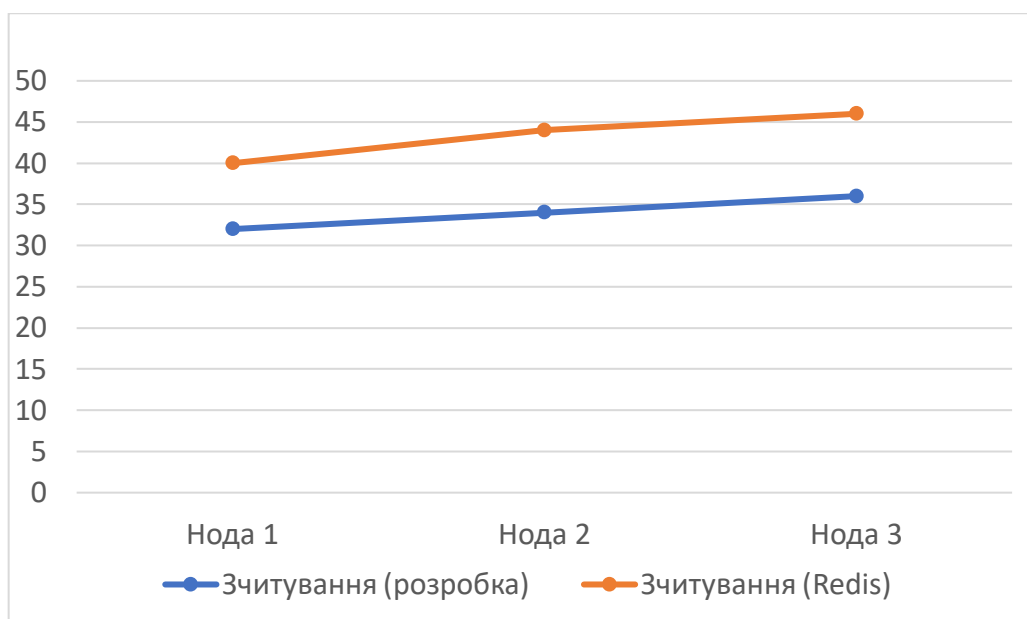


Рисунок 4.12 – Графік порівняння результатів зчитування

Наведені результати вимірювань дозволяють зробити висновки, що попри деградації швидкості запису нових даних в впровадженій системі досягається приріст в швидкості зчитування даних близько 25%, що може бути вагомою перевагою для систем з високим навантаженням.

ВИСНОВКИ

У процесі дослідження та впровадження системи успішно створено функціональну та високопродуктивну систему, спрямовану на вирішення конкретних завдань та задоволення потреб користувачів.

Система ефективно задовольняє вимоги, враховуючи результати аналізу існуючих систем та мов програмування. Використання технологій Redis, Apache Ignite та Memcached дозволило побудувати надійне та продуктивне рішення для зберігання та доступу до даних. Дослідження та аналіз цих технологій допомогли зрозуміти їхні переваги та недоліки в контексті роботи.

Синхронізація в системі реалізована на основі архітектури token ring та працює надійно. Ця архітектура дозволяє підтримувати актуальність даних та забезпечувати консистентність системи навіть в розподіленому середовищі.

Під час аналізу існуючих систем та мов програмування були враховані найкращі практики та рекомендації з використання Redis, Apache Ignite та Memcached. Це дозволило побудувати оптимізовану та високопродуктивну систему, яка враховує сучасні стандарти та потреби користувачів.

Загалом, реалізована система є результатом вивчення та аналізу існуючих технологій та систем у сфері кешування даних. Вона відповідає вимогам, забезпечує надійність та продуктивність, і може бути цінним інструментом для різних застосувань в розгалуженій мікросервісній архітектурі.

В результаті роботи створена система, що дозволяє отримати швидкий доступ на зчитування даних з кешу, так як дані зберігаються локально в кожній ноді і немає необхідності витрачати ресурси на звернення до іншого ресурсу по мережі.

В процесі виконання роботи проведено тестування системи та дослідження швидкісних характеристик реалізованої системи у порівнянні з іншим аналогом. Отримані такі часові характеристики у результаті тестування: для Запису даних - $t = 214\text{мс}$ для реалізованої системи та $t = 45\text{мс}$ для системи на базі Redis, а також

для Зчитування даних - $t = 32\text{мс}$ для реалізованої системи та $t = 44\text{мс}$ для системи на базі Redis.

Таким чином, впроваджена система має як ряд недоліків так і ряд переваг. Основні переваги – це швидкий доступ до даних на локальній ноді, не використовуючи запит даних через мережу, а також зберігання повної копії даних на кожній ноді. З іншого боку недоліки: високі вимоги до оперативної пам'яті та велика затримка при додаванні нових даних, хоча цей процес може бути оптимізований. Реалізована система ідеально підійде для випадків коли в розподіленій високонавантаженій системі є спільні словники які є досить статичними, тобто кількість запитів на зчитування значно перевищують запити на додавання чи оновлення даних.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

- 1 Tibshirani R. et al. Clustering methods for the analysis of DNA microarray data //Dept. Statist., Stanford Univ., Stanford, CA, Tech. Rep. 1999.
- 2 Bazylevych R., Burtnyk R. Algorithms for software clustering and modularization //2015 Xth International Scientific and Technical Conference" Computer Sciences and Information Technologies"(CSIT). IEEE, 2015. C. 30-33.
- 3 Velepucha V., Flores P. A survey on microservices architecture: Principles, patterns and migration challenges //IEEE Access. – 2023.
- 4 Carlson J. Redis in action. Simon and Schuster, 2013.
- 5 Da Silva M. D., Tavares H. L. Redis Essentials. Packt Publishing Ltd, 2015.
- 6 Acharya S. Apache Ignite Quick Start Guide: Distributed data caching and processing made easy. Packt Publishing Ltd, 2018.
- 7 Fitzpatrick B. Distributed caching with memcached //Linux journal. 2004. T. 2004. №. 124. C. 5.
- 8 Krochmalski J. IntelliJ IDEA essentials. Packt Publishing Ltd, 2014.
- 9 Li L., Zhao G., Blum R. S. A survey of caching techniques in cellular networks: Research issues and challenges in content placement and delivery strategies //IEEE Communications Surveys & Tutorials. 2018. T. 20. №. 3. C. 1710-1732.
- 10 Fielding R. et al. Hypertext transfer protocol--HTTP/1.1. 1999. №. rfc2616.
- 11 Fall K. R., Stevens W. R. Tcp/ip illustrated. Addison-Wesley Professional, 2012. T. 1.
- 12 Seufert M. et al. A survey on quality of experience of HTTP adaptive streaming //IEEE Communications Surveys & Tutorials. 2014. T. 17. №. 1. C. 469-492.
- 13 Tanenbaum A. et al. Computer networks title: Computer networking: A top-down approach //Instructor. 2019. T. 201901.

- 14 Peterson L. L., Davie B. S. Computer networks: a systems approach. Elsevier, 2007.
- 15 Stevens W. R., Narten T. UNIX network programming //ACM SIGCOMM Computer Communication Review. 1990. T. 20. №. 2. C. 8-9.
- 16 Horstmann C. S. Core Java SE 9 for the impatient. Addison-Wesley Professional, 2017.
- 17 Bloch J. Effective java. Addison-Wesley Professional, 2017.
- 18 Bloch J., Gafter N. Java puzzlers: traps, pitfalls, and corner cases. Pearson Education, 2005.
- 19 Richardson C. Microservices patterns: with examples in Java. Simon and Schuster, 2018.
- 20 Balalaie A. et al. Microservices migration patterns //Software: Practice and Experience. 2018. T. 48. №. 11. C. 2019-2042.
- 21 Walls C. Spring in action. Simon and Schuster, 2022.
- 22 Walls C. Spring Boot in action. Simon and Schuster, 2015.
- 23 Ньюмен С. Побудова мікросервісів. O'Reilly Media, 216. 326 с.
- 24 Nickoloff J., Kuenzli S. Docker in action. Simon and Schuster, 2019.
- 25 Miell I., Sayers A. Docker in practice. – Simon and Schuster, 2019.
- 26 Kirichek G., Skrupsky S., Tiahunova M., Timenko A. Implementation of Web System Optimization Method. In CMIS-2020: The Third International Workshop on Computer Modeling and Intelligent Systems, Zaporizhzhia, Ukraine, April 27-May1, 2020, pp. 199-210.
- 27 Киричек Г.Г, Фалькевич В.Г. Оптимізація інтерактивної системи. Міжнародна науково-практична конференція «Технічні науки: історія, сучасність, майбутнє, досвід ЄС», Влоцлавек, Польща, 27–28 вересня 2019, pp.30-32.
- 28 ANSI/IEEE standard 610.12-1990: Glossary of SE Terminology. NY: IEEE [Електронний ресурс] URL: <https://ieeexplore.ieee.org/document/159342>
- 29 Koomen T., and Pol M., “Test process improvement: a practical step-by-step guide to structured testing”, Addison Wesley, 1999.

ДОДАТОК А

ЗМІСТ POM.XML

Лістинг А.1 – Зміст pom.xml файлу

```

<?xml version="1.0" encoding="UTF-8"?>
<project                xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.2.0</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>cache-node</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>cache-node</name>
    <description>cache-node</description>
    <properties>
        <java.version>17</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>

```

```

        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-
plugin</artifactId>
            <configuration>
                <excludes>
                    <exclude>

<groupId>org.projectlombok</groupId>
                        <artifactId>lombok</artifactId>
                    </exclude>
                </excludes>
            </configuration>
        </plugin>
    </plugins>
</build>

</project>

```

ДОДАТОК Б

ЗМІСТ КЛАСУ CACHECLIENTSERVICEIMPL.JAVA

Лістинг Б.1 – Зміст CacheClientServiceImpl.java файлу

```
package com.example.cachenode.client;

import com.example.cachenode.data.CacheStorageService;
import com.example.cachenode.data.CachedData;
import com.example.cachenode.node.CacheService;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import java.time.LocalDateTime;
import java.util.Collection;
import java.util.UUID;

@Service
@RequiredArgsConstructor
public class CacheClientServiceImpl implements
CacheClientService {
    private final CacheStorageService cacheStorageService;
    private final CacheService cacheService;

    @Override
    public Collection<CachedData> getAll() {
        return cacheStorageService.getData().values();
    }

    @Override
    public CachedData get(String id) {
        return cacheStorageService.getData().getOrDefault(id,
CachedData.EMPTY);
    }
}
```

```

    }

    @Override
    public String save(String data) {
        String key = UUID.randomUUID().toString();
        CachedData value = CachedData.builder()
            .id(key)
            .data(data)
            .lastUpdateTime(LocalDateTime.now())
            .versionID(1)
            .build();
        cacheStorageService.getData().put(key, value);
        cacheService.sync(null, value);
        return value.getId();
    }

    @Override
    public String update(String id, String data) {
        CachedData
cachedData =cacheStorageService.getData().get(id);
        if(cachedData == null) {
            throw new RuntimeException("ID " + id + "doesn't
exists");
        }
        CachedData value = CachedData.builder()
            .id(id)
            .data(data)
            .lastUpdateTime(LocalDateTime.now())
            .versionID(cachedData.getVersionID() + 1)
            .build();
        cacheStorageService.getData().put(id, value);
        cacheService.sync(null, value);
        return value.getId();
    }
}

```

ДОДАТОК В

ЗМІСТ КЛАСУ CACHECLIENTSERVICEIMPL.JAVA

Лістинг В.1 – Зміст CacheClientController.java файлу

```
package com.example.cachenode.client;

import com.example.cachenode.data.CachedData;
import lombok.RequiredArgsConstructor;
import lombok.SneakyThrows;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.*;

import java.util.Collection;

import static java.lang.Thread.sleep;

@RestController
@RequiredArgsConstructor
@RequestMapping("client")
public class CacheClientController {

    private final CacheClientService cacheClientService;

    @Value("${cache.node.force.delay}")
    private Long forceDelayTime;

    @GetMapping
    public Collection<CachedData> getAll() {
        return cacheClientService.getAll();
    }

    @SneakyThrows
    @GetMapping("{id}")
```

```

        public CachedData get(@PathVariable String id,
                               @RequestParam(name = "useDelay",
defaultValue = "false") boolean useDelay) {
            if(useDelay) {
                sleep(forceDelayTime);
            }
            return cacheClientService.get(id);
        }

        @SneakyThrows
        @PostMapping
        public String post(@RequestBody String data,
                           @RequestParam(name = "useDelay",
defaultValue = "false") boolean useDelay) {
            if(useDelay) {
                sleep(forceDelayTime);
            }
            return cacheClientService.save(data);
        }

        @SneakyThrows
        @PutMapping("{id}")
        public String put(@RequestBody String data, @PathVariable
String id,
                           @RequestParam(name = "useDelay",
defaultValue = "false") boolean useDelay) {
            if(useDelay) {
                sleep(forceDelayTime);
            }
            return cacheClientService.update(id, data);
        }
    }
}

```

ДОДАТОК Г

ЗМІСТ КЛАСУ CACHESERVICEIMPL.JAVA

Лістинг Г.1 – Зміст CacheServiceImpl.java файлу

```
package com.example.cachenode.node;

import com.example.cachenode.data.CacheStorageService;
import com.example.cachenode.data.CachedData;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
@RequiredArgsConstructor
@Slf4j
public class CacheServiceImpl implements CacheService {

    private final RestTemplate restTemplate;

    private final CacheStorageService cacheStorageService;

    @Value("${cache.node.current.host}")
    String currentHost;

    @Value("${cache.node.next.host}")
    String nextHost;

    @Override
    public void sync(String initiator, CachedData data) {
        if(nextHost.isEmpty()) {
```

```

        return;
    }
    String url = "http://" + nextHost + ":8080" + "/node/"
+ (initiator == null ? currentHost : initiator);
    log.info("Try to sync {} with {}. Data: {}", url,
nextHost, data);
    restTemplate.postForObject(url, data,
CachedData.class);
}

@Override
public void update(String initiator, CachedData data) {
    log.info("Try to update value {}, in {}. Initiator:
{}",data, currentHost, initiator);
    if(initiator.equals(currentHost)) {
        return;
    }
    cacheStorageService.getData().put(data.getId(), data);
    this.sync(initiator, data);
}
}

```