F3

Faculty of Electrical Engineering
Department of Computer Science

Bachelor's Thesis

Research on distributed in-memory databases and cache implementation in SpringBoot application

Roman Danilchenko
Open Informatics

January 2024
Supervisor: Mgr. Jakub Maxa

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Danilchenko Roman** Personal ID number: **499005**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Software**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Research on distributed in-memory databases and cache implementation in SpringBoot application

Bachelor’s thesis title in Czech:

Výzkum distribuovaných in-memory databází a implementace cache ve SpringBoot aplikaci

Guidelines:

Bibliography / sources:

• https://hazelcast.com/
• https://redis.io/
• https://ignite.apache.org/
• Spring Start Here by Laurentiu Spilca, 2021, Manning Publications,ISBN:9781617298691
• https://kotlinlang.org/
• https://scholar.google.com/

Name and workplace of bachelor’s thesis supervisor:

Mgr. Jakub Maxa Trask solutions a.s.

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **09.02.2023** Deadline for bachelor thesis submission: **09.01.2024**

Assignment valid until: **22.09.2024**

Mgr. Jakub Maxa Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

. | Date of assignment receipt | Student’s signature

Acknowledgement / Declaration

I extend my deepest gratitude to my supervisor, Mgr. Jakub Maxa, for giving me such an opportunity to work on this interesting and beneficial project. To my family and friends, your constant support and belief in my abilities have been my driving force.

I hereby declare that I have completed this thesis independently and quoted all the sources of information used in accordance with methodological instructions on ethical principles for writing an academic article.

In Prague 09.01.2024

.....

Abstrakt / Abstract

Mezi cíle této bakalářské práce patří výzkum ukládání dat do mezipaměti v distribuovaných systémech a zkoumání platforem, které tuto funkci nabízejí. Následně je cílem implementovat aplikaci integrovanou s jednou z těchto platforem. Práce poskytuje srovnávací analýzu vybraných technologií, nastiňuje návrh architektury systému a prezentuje výsledky získané při testování implementované aplikace. Závěr navíc obsahuje návrhy na budoucí zlepšení.

Klíčová slova: cachování, distribuované systémy, webové aplikace, Spring Boot, Hazelcast.

The objectives of this bachelor thesis include researching data caching in distributed systems and exploring platforms that offer this functionality. Afterwards, the aim is to implement an application integrated with one of these platforms. The thesis provides a comparative analysis of the chosen technologies, outlines the system architecture design, and presents the results obtained from testing the implemented application. Additionally, the conclusion includes suggestions for future improvements.

Keywords: caching, distributed systems, web application, Spring Boot, Hazelcast.

Contents /

1 Introduction **1**
1.1 Requirements for modern software applications 1
1.2 Objectives 2
1.3 Motivation 2

2 Theoretical part **3**
2.1 Concept of caching 3
2.1.1 What is caching? 3
2.1.2 How does caching work? . . . 3
2.1.3 When to use caching? 4
2.1.4 Where caching can be used? 4
2.1.5 Eviction policies 5
2.1.6 Cache invalidation 5
2.1.7 Types of caches 5
2.1.8 Caching strategies 6
2.2 In-Memory Computing 8
2.2.1 In-Memory Database 8
2.2.2 In-Memory Data Grid 9
2.3 Caching platforms 9
2.3.1 Apache Ignite 9
2.3.2 Redis 11
2.3.3 Hazelcast 12
2.3.4 Aerospike 14
2.3.5 Analysis conclusion 15
2.4 Architectural approaches 15
2.4.1 Monolithic architecture . . 16
2.4.2 Microservice architecture . 16
2.5 Event streaming 17
2.5.1 Related definitions 17
2.6 Change Data Capture 18
2.7 Application containerization . . 18
2.7.1 What is a containerized application? 18
2.7.2 Advantages 18
2.7.3 Disadvantages 18

3 Application design **19**
3.1 System requirements 19
3.1.1 Functional requirements . . 19
3.1.2 Nonfunctional requirements 19
3.2 Architecture concept 19
3.3 Cache design 20
3.3.1 General information 20
3.3.2 Caching method 21
3.4 Data model 21
3.4.1 Database data model . . . 21
3.4.2 Cache data model 21

4 Practical part **23**
4.1 Technology stack 23
4.2 Technologies introduction . . . 23
4.2.1 Oracle Database 23
4.2.2 Apache Kafka 23
4.2.3 Kafka Connect 25
4.2.4 Debezium 25
4.2.5 Spring Boot 25
4.3 Infrastructure setup 25
4.3.1 Prerequisites 25
4.3.2 Setup steps 26
4.4 Cache server 29
4.4.1 Spring boot project initialization 29
4.4.2 Overview 29
4.4.3 Hazelcast configuration . . 31
4.4.4 Docker image 32
4.5 Cache client 33
4.5.1 Spring boot project initialization 33
4.5.2 Overview 33
4.5.3 App configuration 35
4.5.4 Hazelcast configuration . . 36
4.5.5 Documentation 37
4.5.6 Docker image 37

5 Testing **38**
5.1 Testing approach 38
5.2 Performance 38
5.2.1 First set of test cases . . . 38
5.2.2 Second set of test cases . . 39
5.2.3 Third set of test cases . . . 40
5.2.4 Conclusion 41
5.3 Data consistency 41
5.3.1 Client deletion 41
5.3.2 Update product 41
5.3.3 Delete product 42
5.3.4 Create relation 42
5.3.5 Delete relation 42
5.3.6 Update product key in relation 42
5.3.7 Update client key in relation 42

5.3.8 Update both keys in relation 42

6 Conclusion **43**

6.1 Areas for improvement 43

References **44**

A Acronyms and symbols **47**

A.1 List of acronyms 47

Tables / Figures

3.1 Clients map structure visualisation. 21
3.2 Products map structure visualisation. 22
3.3 Relations map structure visualisation. 22

2.1 Basic cache work, first request ..3
2.2 Basic cache work, second request 4
2.3 Example of external cache. 6
2.4 Cache-Aside pattern visualisation. 7
2.5 Write-Through pattern visualisation. 7
2.6 Read-Through pattern visualisation. 7
2.7 Write-Behind pattern visualisation. 8
2.8 CAP theorem visualisation. ... 10
2.9 SQL and Redis Stack example. 12
2.10 Hazelcast embedded topology. . 13
2.11 Hazelcast client-server topology. 13
2.12 Aerospike Monitoring Stack example. 15
2.13 Monolithic architecture example. 16
2.14 Microservice architecture example. 17
3.1 Architecture concept visualisation. 20
3.2 Database data model visualisation. 21
4.1 Kafka topic structure visualisation. 24
4.2 Command to run Oracle Database in container. 26
4.3 PDB schema creation. 26
4.4 Oracle configuration for Debezium. 27
4.5 Data model realization. 27
4.6 Dockerfile for custom Kafka Connect image creation. 28
4.7 Docker Compose file contents. . 28
4.8 Request payload for connector registration. 29
4.9 OpenAPI documentation example. 37
5.1 Cache cluster structure after application startup. 38

5.2 5000 clients uploaded to the database. 38
5.3 The first request for the client speed. 39
5.4 The second request for the client speed. 39
5.5 Performance difference for client right for product verification service. 39
5.6 The first request speed for getting a list of products for a client. 39
5.7 The second request speed for getting a list of products for a client. 39
5.8 Number of rows in each table. . 40
5.9 Request for client existence verification. 40
5.10 Client right for product verification request. 40
5.11 Available product list for client request. 40
5.12 Number of rows in each table. . 40
5.13 Request for client existence verification. 41
5.14 Client right for product verification request. 41
5.15 Available product list for client request. 41

Chapter 1
Introduction

In modern world, software applications play a big role in many different areas of our lives. Industries such as online commerce, social media, healthcare systems, e-learning, enterprise resource planning and many others rely on digital infrastructure. Software increases productivity, work efficiency and makes our lives easier, more comfortable and convenient. This implies a constant increase in user expectations for seamless experience and demand for high-performance and scalable solutions.

1.1 Requirements for modern software applications

Speed is crucial for user experience. Slow-loading applications frustrate users and can lead to abandonment, especially for web and mobile applications. Based on Google's research, a one-second delay in page load time can lead to a 7% drop in conversions. For mobile apps, a delay of 0.1 second can result in a 20% decrease in user engagement [1].

As the business grows, the user base expands and the load increases accordingly, the software must be able to handle the increase in traffic without sacrificing performance. This is especially important for enterprise applications with a large number of simultaneous users, e-commerce sites during peak times or seasons, and social networking sites during various large events. It was the scalability of the application that helped Zoom to ensure its application ran seamlessly during the COVID-19 pandemic. The company scaled its infrastructure multiple times and provided communications to 300 million users every day [2].

High availability is equally crucial for modern applications. This process ensures that users have access to critical software as much as possible and guarantees proper operation in situations such as power outages, server failures, etc. Downtimes can lead to significant financial losses, reputational damage, and customer dissatisfaction. Enterprises in vertical markets such as banking and finance, stock exchanges, communications/media, insurance, healthcare, manufacturing, retail and transportation, whose businesses are based on intensive data transactions, can lose millions if service is interrupted for two, five, 10 or 30 minutes (Information Technology Intelligence Consulting, 2016, p.8) [3].

Additionally, recent years have witnessed an unprecedented surge in the volume of data processed by applications across various businesses. According to a report by Deloitte Touche Tohmatsu Limited, which is considered one of the Big Four accounting firms , the global data volume is expected to reach 175 zettabytes[1] by year 2025 [4]. As organizations grapple with this exponential growth, optimizing data access and retrieval has become paramount to sustaining competitive advantages.

[1] One zettabyte is equal to one sextillion bytes.

1.2 Objectives

The purpose of this work is to research and use distributed caching platforms as a tool to improve performance and scalability of modern software applications. The main attention is paid to the study of its capabilities, advantages and practical application, as well as the implementation and testing of distributed cache in the environment of multi-server applications.

1.3 Motivation

The main motivation for this work was to expand my knowledge, improve my professional skills in web-application development and then apply the learned technologies and the proposed architectural concept in a real-world enterprise system.

Chapter 2
Theoretical part

This chapter introduces the concept of caching, contains a comparative analysis of the Apache Ignite, Redis, Hazelcast, and Aerospike platforms, and provides theoretical overview for the subsequent implementation of the practical part of the work.

2.1 Concept of caching

2.1.1 What is caching?

Caching is the process of storing data in a cache, which is a temporary storage area that facilitates faster access to data with the goal of improving application and system performance [5]. However, caching brings certain complexities to the system, some of the most challenging tasks when integrating caches into applications are keeping data up-to-date and efficiently managing limited amount of memory. These will be described later.

2.1.2 How does caching work?

Cache can be either software or hardware component and the data stored in it is a copy of the main storage data. Logically, a cache is a storage in the form of key-value pairs. Each such pair is persisted for a certain time, which is called Time to live (or TTL).

The following diagrams describe a basic example of how caching works.

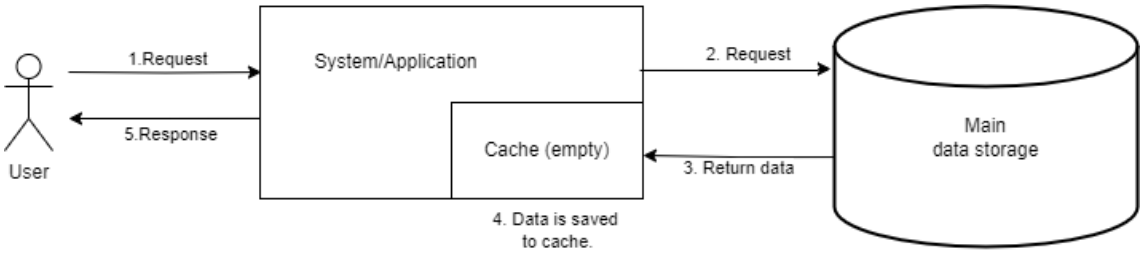

Figure 2.1. First request.

Figure 2.1. shows the order of events in the system when the first request for data occurs:

1. User sends a request for data.
2. As the cache is empty, application sends a request to the main data storage for data retrieval.
3. Data is returned from the main storage.
4. The cache is populated with retrieved data.
5. Application sends a response to user with requested data.

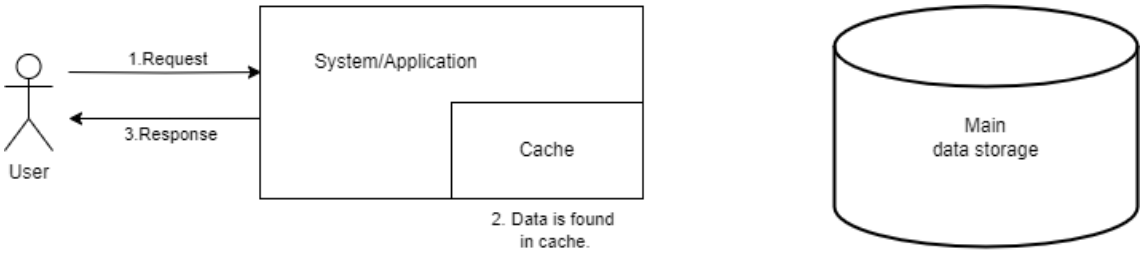

Figure 2.2. Second request.

Figure 2.2. shows the order of events in the system when the second request for data occurs:

1. User sends a request for data.
2. Due to the fact that the cache contains the necessary data, there is no need to query the main data storage.
3. Application sends a response to user with requested data.

2.1.3 When to use caching?

The way caching is used always depends on the requirements of the specific application. In general, any data can be cached, but the following 2 groups of data are particularly suitable for storing in temporary storage with fast access:

- Static data that almost never changes. For example, DNS Information, geolocation Data, API Responses.
- Data that changes within hours or days. For example, event schedules, news articles, available products on websites and information about them.

Most often it makes no sense to cache data that changes with each new request. In such cases, keeping stored information actual and managing the cache may outweigh the gain in system performance.

2.1.4 Where caching can be used?

As caching is a general concept, it can be employed at various levels of the system architecture, e.g.:

- Central Processing Unit has an attached hardware component called L1 cache, it stores small amounts of data so that future requests for that data can be served faster [6].
- Database contains an integrated cache. It is managed within the database engine and when the data stored in the database table is changed, cache is updated automatically [7].
- Application typically can have an in-memory store system on its server-side to hold frequently accessed or expensive to compute data that are not cached in other caching levels [8]. The main focus of this work is directed at this level of caching.
- Content delivery network for quicker delivery of such content as images, webpages and videos caches data in proxy servers [1] that are located closer to end users than origin servers [9].

[1] A proxy server is a server that receives requests from clients and passes them along to other servers.

2.1.5 Eviction policies

Since the size of the cache is limited and usually significantly smaller than the size of the main storage at some point it is necessary to decide which stored data will be replaced by newer data. Special eviction strategies exist for this purpose:

- LRU (Least Recently Used) - an eviction policy that removes the least recently accessed cache entries first, based on the assumption that items accessed lately are more likely to be needed soon [10].
- LFU (Least Frequently Used) - a policy that evicts the cache entries that are least frequently accessed first, assuming that items accessed infrequently are less likely to be needed in the near future [10].
- MRU (Most Recenly Used) - an eviction policy that removes the most recently used items from cache first.
- FIFO (First-In-First-Out) - an eviction strategy that uses queueing logic when storing elements, when the size of the cache is exceeded elements at the front of the queue are expunged.
- LIFO (Last-In-First-Out) - a policy that is similar to FIFO, but it removes elements from the end of the queue when the temporary storage size limits are reached.
- RR (Random Replacement) - an eviction policy that removes random cache entries.
- TTL - an eviction strategy that removes data from the cache based on a special timestamp that is assigned to it when added. Once this time limit is reached, the data will be deleted.

2.1.6 Cache invalidation

Cache invalidation is the process of removing data from the cache or marking it as invalid. This process ensures that information stored in cache is up-to-date. Users are exposed to the risk of seeing irrelevant information if the cache is not invalidated, which might be confusing or even violate their privacy. The process of cache invalidation entails coordinating numerous copies of data across several system tiers, including a database and web or application server, in order to guarantee that the data that is cached is reliable and accurate [11]. Also, as stated in article on cache invalidation topic by Redis [11], there are different types of cache invalidation:

- Time-based - cache entries are removed based on TTL.
- Event-based - cache entries are invalidated when a specific event occurs in the system.
- Command-based - a user triggers a predefined command which results in an invalidation ID. Then all cached objects that contain dependency ID [2] matching invalidation ID are deleted from cache.
- Group based - takes place when the cache is invalidated based on a specific group or a category of objects. This kind of invalidation is useful when the requirement is to invalidate a larger group of data at once.

To efficiently perform cache invalidation, the eviction algorithms discussed above can be used.

2.1.7 Types of caches

From the architectonic point of view we can distinguish the following types of caches:

[2] `Dependency ID is typically generated when the object is added to cache.`

- Embedded - in this type of caching frequently accessed data from database is stored in the application memory, which implies that communication with cache is fast. Figure 2.1 shows an example of an application with an embedded cache.
- External - this variant assumes an exposition of an independent, standalone cache server which is shared among all running application instances. Communication between the application and the cache is performed by means of network calls [12].

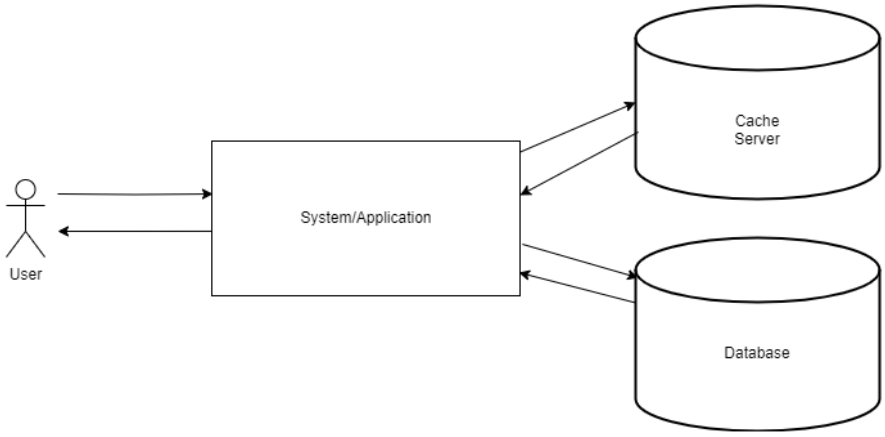

Figure 2.3. External cache example

It is also important to note that when using external caching, the application is more flexible for scaling due to the fact that it is possible to separately scale the cache and the application itself, whereas with embedded caching only joint scaling of the cache and the application is supported, i.e. the number of instances of the application and cache always equals each other. Both of these types of caching can be run in the following ways [13]:

- Node cache (or Cache per member) - which means that each server instance has its own life cycle. It is comparatively easy for implementation. However, this approach brings such problems as different data on each node and non-synchronous invalidation cycles between nodes.
- Cluster cache - in this case, the application is not accessing a specific node but the entire cluster. The cluster contains multiple nodes and acts like a single cache.

Cluster cache can be distributed and/or replicated.

In replicated cache every node stores a complete copy of data. It means that any node can serve a request for data, besides the case of a node failure, other nodes still have the whole dataset. However, replicated cache may be limited from scaling perspective as each node requires a full copy of data.

In distributed cache data is also spread across multiple nodes, but compared to replicated cache nodes, they store only subsets of data, therefore while accepting a data request, the cache system first identifies the specific node that holds the information and then retrieves the data from it. Distributed caches provide a broad opportunity for horizontal scaling, allowing to increase cluster size and efficiency, but if a particular node fails, there is a risk of losing a part data from the entire cluster.

2.1.8 Caching strategies

There is a variety of caching patterns that describe the interactions between the application, cache, and database.

- Cache-Aside is the most common caching strategy [14]. In this strategy, the application is responsible for managing the cache [15].

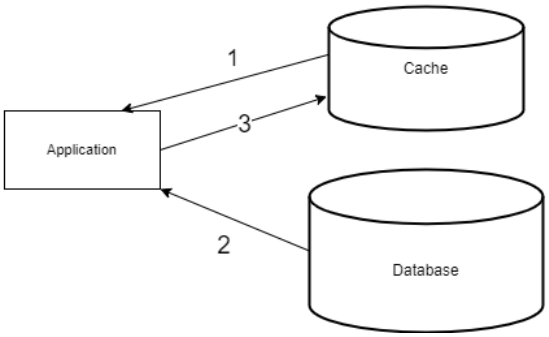

Figure 2.4. Cache-Aside pattern visualisation.

1. When application needs to read data from the database, it checks the cache first to determine whether the data is available
2. If a cache hit [3] occurs, the cache data is returned to the caller.
3. If a cache miss [4] occurs, application queries the database for the needed data. After that cache is populated with the retrieved data, and then it is returned to the caller.

When using this pattern, cache contains only data that the application requests, but it is saved to the cache only after a cache miss, which adds additional overhead to the initial response time [14].

- Write-Through is a caching strategy in which cache and database are updated almost simultaneously.

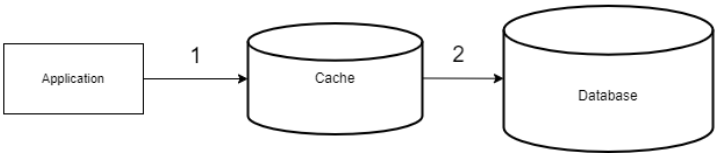

Figure 2.5. Write-Through pattern visualisation.

1. Data is written to cache.
2. Data is written to database.

The use of this strategy helps guarantee that data is consistent between the cache and the database [16]. However, this approach requires the full data to be uploaded to cache, introduces higher latency on the write operations, and higher write load on the cache system [17].

- Read-Through is a caching strategy that uses the cache as the primary data source.

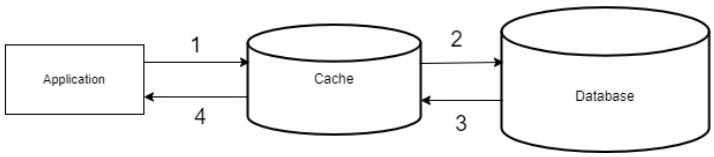

Figure 2.6. Read-Through pattern visualisation.

[3] Cache hit means that required data is found in cache.
[4] Cache miss means that required data is not found in cache.

1. Read the data from cache.
2. In the case of the cache miss query the database.
3. Return the data and populate the cache.
4. Return the data to the caller.

The main advantage is that this strategy significantly reduces the read latency for frequently accessed data. The biggest disadvantages are that in case of a cache failure, application loses access to the data, and an increased risk of stale data if information in the database is frequently updated.

- Write-Behind is a caching strategy in which the cache is updated first, and then the database is updated after a set period of time.

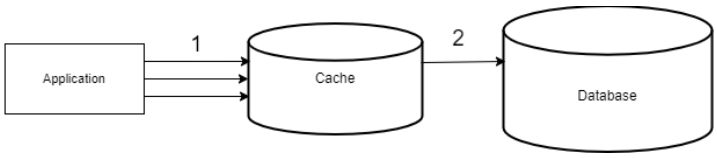

Figure 2.7. Write-Behind pattern visualisation.

1. Write the data to cache.
2. Write the data to database after some time.

Write-Behind strategy is more useful when many cache updates are expected, as the user does not need to wait for changes to be saved to the database [16]. Nevertheless, in case of a system failure there is a risk that data may not have been saved to the database yet.

- Refresh-Ahead is a caching pattern that proactively refreshes data in the cache before it is requested by the application [18]. This strategy is useful when the application needs the most recent data and refreshing the data is less expensive than retrieving it from the backend system [18]. But this can result in a strain on the backend system.

2.2 In-Memory Computing

In-Memory Computing (or IMC) is a technology that allows to keep and process data in the internal computer memory, also called random access memory (or RAM), in real-time [19]. Provides performance many times faster than disk-based systems which is the greatest advantage of this approach. On the other hand, the volatility of RAM brings a risk of losing data in case of an unforeseen work interruption of IMC based system. This technique is often implemented in the form of middleware software and is used in a clustered environment pooling the RAM of all nodes together [20].

2.2.1 In-Memory Database

An in-memory database (or IMDB) is a data storage which holds all its information in the computer RAM, thus achieving a significant acceleration of data access compared to traditional databases, which use more reliable and long-term memory storage solutions, such as solid-state drives or hard disk drives [21].

Mostly, IMBDs provide similar functionalities as classic relational database management systems (or RDBMS), e.g. SQL support and may be a good replacement for existing RDBMS with minimal effort and changes [22].

Since In-Memory databases are commonly run in a cluster to increase the amount of used RAM, they provide complex features such as distributed joins, which are rather useful for developers, but prevent flexible horizontal scalability [22].

In-memory data stores are employed and beneficial in cases where the system needs fast writing and reading of data from storage [23]. Some real-world use cases of IMDBs are:

- E-commerce websites need to store such components as shopping carts, a list of offered products and information about them to improve user experience [24].
- IoT devices that produce and transfer large streams of data might need to store it in some quick storages [24].
- In banking applications, IMBDs are used to analyze customer's shopping patterns and help detect fraud when suspicious transactions are made [24].

2.2.2 In-Memory Data Grid

An In-Memory data grid (or IMDG) is an advanced distributed cache, which stores data in the combined random access memory of the computers in the cluster [25]. It is integrated with the underlying database to keep stored information relevant and consistent [26].

IMDG is a key-value store that provides high flexibility in data storage, since keys and values can be any domain object, unlike IMDB [27]. Furthermore, IMDGs distribute data in a cluster such that a particular node stores a specific portion of data [27].

It is worth noting that integration of IMDG into an existing application implies changes to the application itself, but does not require changes to the RDBMS [22]. The benefits of In-Memory data grids are faster data access and comparatively easy further horizontal scaling.

Example use cases are:

- Payment processing when a number of calculations should be conducted in a limited time window [26].
- Fraud detection [26].
- Large-scale simulations that take into account a variety of variables to help create a clearer picture of potential future events [26].

2.3 Caching platforms

There is a range of platforms that provide a ready made caching solution. Below is an overview and comparison of the most popular of these.

2.3.1 Apache Ignite

Apache Ignite is an open source IMDG, distributed database, caching and high perfomance computing platform [28].

Apache Ignite memory architecture is designed in such a way that data is stored both in the RAM of the nodes in the cluster and on disk [29]. It is important to note that the format of data storage in both places is the same, which saves time when moving information between them [30]. This architecture divides data into special blocks - pages, each of which has a unique identifier and is stored in RAM and on disk in a special hierarchy using arrays and B+ trees [29].

Ignite supports integration with such popular databases as Oracle, MySQL, Microsoft SQL server, PostgreSQL, MongoDB and others. It is able to generate a domain model according to the schema of the connected database [31]. Ignite monitors the consistency of data in the main storage and in memory. When a new transaction is made,

Ignite controls its success in the database, then propagates the changes to memory. In addition, it is possible to use distributed SQL with ANSI-99 syntax, for which a special driver, such as JDBC, should be connected [29, 31].

Ignite realises LRU eviction policy.

Another very important thing to note is the support of ACID transactions, depending on the requests the developer can choose the strategy of transaction execution - pessimistic or optimistic. Each strategy has its own nuances, with pessimistic strategy the speed of transaction execution is lower, but the guarantee of data consistency is higher, with optimistic strategy it is vice versa [31]. It is because of the ACID transaction support Ignite can be classified as a CP system [32]. This means that consistency and partition-tolerance take precedence over ensuring availability.

ACID is an acronym which defines a set of desired properties of transaction management [33].

- Atomicity - partial execution is not allowed
- Consistency - every transaction will bring database from one valid state to another.
- Isolation - transaction executed in parallel do not interfere or affect one another.
- Durability - changes made to data within a successful transaction are saved even in the case of system failure.

CAP theorem states that only two of the three characteristics depicted in the figure below can be guaranteed in a distributed system.

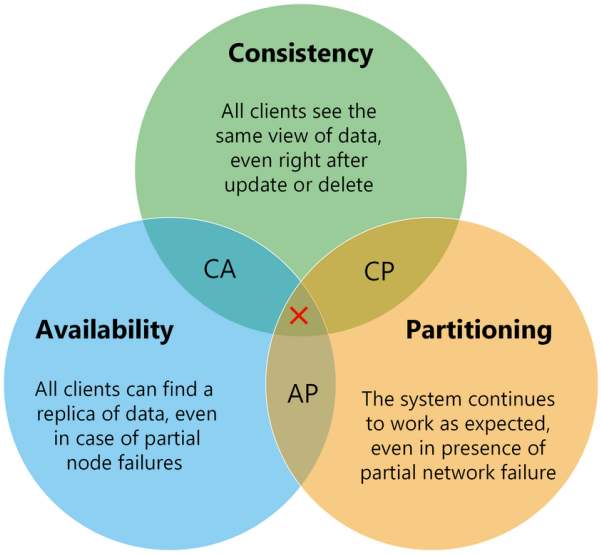

Figure 2.8. CAP theorem visualisation [5].

As stated in the official documentation [34], Ignite has two main discovery mechanisms:

- TCP/IP discovery, which employs TCP/IP to detect nodes in the cluster, and can be configured to use multicast or static IP addresses. It uses the ring topology, which may entail an increase in the time of adding a new node or detecting a node that has failed when there are a large number of them in the cluster.

[5] Picture was downloaded from https://www.researchgate.net/figure/Visualization-of-CAP-theorem_fig2_282679529

- ZooKeeper [6] discovery organizes nodes into a star topology and uses ZooKeeper cluster for adding new nodes and detecting failures. As ZooKeeper is also a distributed system, it can be challenging to implement this approach.

It is also possible for the developer to perform actions depending on the state of the nodes. Each node has four states [29]:

- Before the startup
- Just after the startup
- Just before the shutdown
- After the shutdown

Ignite API provides such structures as [34]: distributed set and queue, atomic long and reference, distributed semaphore, distributed id generator, distributed locks.

Ignite API is available for many programming languages - Java, C#/.NET, C++, Python, NodeJS and others [34].

The official documentation describes how to work with various tools for monitoring and controlling the cluster, both from the publisher itself - Control Script, and from third parties - GridGain Control Center, Tableau and so on. In addition, it is possible to download a special script for measuring the efficiency of Ignite components.

Ignite has a rather large community of professionals around the world, with over 100 developers daily allowing the project to grow and expand [7].

2.3.2 Redis

The information provided in this section was taken from official documentation unless otherwise stated [35].

Redis is an open source, in-memory data structure store used as a database, cache, message broker, and streaming engine .

Redis provides the ability to save data to disk. This is realized in several ways: Redis Database (RDB), Append Only File (AOF) and a combination of these. RDB is a compact file that stores data from Redis that was created at some point in time using a snapshot. RDB is a good choice for recovering data after an application failure. RDB also improves performance because a separate process is responsible for saving data to durable store. Also, reboots with large amounts of data are faster with RDB than with AOF. However, RDB has disadvantages as well. For example, this strategy does not minimize the chances of data loss in case of an outage. Typically, data snapshots are created every 5 minutes, so last-minute information that has not yet been saved will be lost. In addition, as mentioned earlier, RDB creates a separate process to save data, which can take some time and even cause Redis to stop processing requests for up to 1 second. AOF, unlike RDB, stores information about each write operation that Redis receives. This allows to restore the sequence of operations when the server is restarted. The file to which AOF writes data is an append-only log. Redis handles hypothetical problems that might occur with this file, e.g., in situations when the disk runs out of space or power supply is interrupted. Additionally, at times when the file becomes too large, Redis creates a new file containing the minimum number of operations to reproduce the current data set, and then starts writing newly received operations only there. Furthermore, since the AOF file is stored in a format that is easy to understand and parse, it is possible to export it. However, these files are usually larger than RDB files storing equivalent information.

[6] This technology will be introduced later.
[7] https://ignite.apache.org/our-community.html

Redis does not provide native SQL support, but has its own tool for data selection, projection and aggregation - Redis Stack. The table below shows examples of using these two technologies.

Type	SQL	Redis Stack
Selection	`SELECT * FROM bicycles WHERE price >= 1000`	`FT.SEARCH idx:bicycle "@price:[1000 +inf]"`
Simple projection	`SELECT id, price FROM bicycles`	`FT.SEARCH idx:bicycle "*" RETURN 2 __key, price`
Calculated projection	`SELECT id, price-price*0.1 AS discounted FROM bicycles`	`FT.AGGREGATE idx:bicycle "*" LOAD 2 __key price APPLY "@price-@price*0.1" AS discounted`
Aggregation	`SELECT condition, AVG(price) AS avg_price FROM bicycles GROUP BY condition`	`FT.AGGREGATE idx:bicycle "*" GROUPBY 1 @condition REDUCE AVG 1 @price AS avg_price`

Figure 2.9. SQL and Redis Stack example.

Redis also supports transactions, though which are not ACID. They only guarantee that all operations within a transaction are executed in the order in which they were entered. With AOF, if the server fails during the execution of a transaction, there is a chance that only a portion of the operations will be executed and logged. Moreover, Redis does not support rollbacks because they affect the efficiency of the whole system.

Redis Cluster is a project that enables Redis to run in a cluster. Redis cluster automatically splits data between nodes and keeps operations going when some of them fail. TCP/IP is used to discover the cluster and the elements inside. Data is distributed among the nodes according to hash slots. An entire cluster has 16384 slots which are shared among the nodes according to their quantity. When a new entry is saved, the hash slot of the node that will store this information is calculated from the entry key. It is also possible to create replicas for each node, which will store the same hash slots as the main node. This is done in order to ensure that in case of failure in the main node, the cluster can continue its work. Redis Cluster does not guarantee strong consistency due to the fact that data from master node is propagated asynchronously to replicas, i.e. the master node does not receive confirmation from replicas.

The eviction strategies that are implemented in Redis are LRU, LFU, Random and TTL.

Redis provides such data structures as strings, lists, sets, sorted sets and hash maps.

Supported programming languages are C/C++, Java, Go, JavaScript and others.

Logs or integration with third-party systems such as Prometheus, Uptrace, and Nagios can be used to monitor and control the Redis cluster.

Redis has a relatively large community, provides various resources for learning about its features and for troubleshooting problems [8].

2.3.3 Hazelcast

The information provided in this section was taken from official documentation [36].

Hazelcast IMDG is an open-source distributed in-memory object store. It is implemented in Java language and has clients for Java, C++, .NET, REST, Python, Go and Node.js.

[8] `https://redis.io/community/`

Hazelcast IMDG is highly scalable, a cluster can easily support hundreds or even thousands of members. When a new node is added, it automatically discovers the cluster, which adds performance and increases the amount of available memory. Communication between nodes is established using TCP/IP. In terms of functionality, all members have the same role, the first node added to the cluster distributes data between the ones added later.

The memory segments in Hazelcast are called partitions, and their size is limited by the hardware parameters of the system. By default, Hazelcast creates 271 partitions and divides them among the nodes in the cluster. All partitions have identifiers. These are stored in the Partition Table along with the address of the members in the cluster so that all members know where the data is located. Also, a copy is created for each partition, which is called a backup. Backups are stored by members that do not keep the original of this data. When a new object is added to the cluster, its key is translated into a data array, hashed and divided by the number of partitions, the resulting remainder after the division is the ID of partition where the object will be stored.

Hazelcast IMDG supports both embedded and client-server topology.

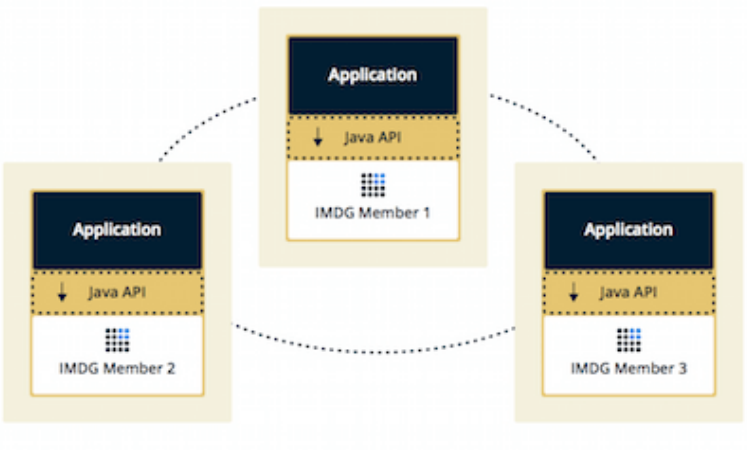

Figure 2.10. Hazelcast embedded topology.

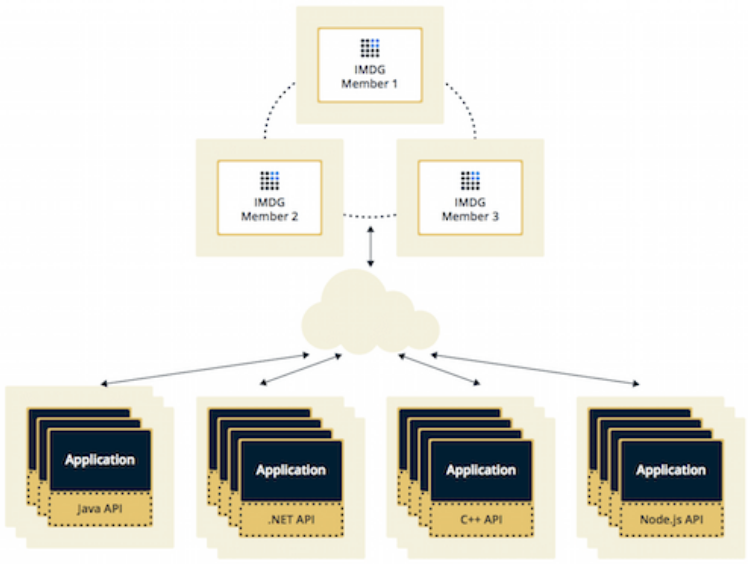

Figure 2.11. Hazelcast client-server topology.

Hazelcast provides a large number of distributed structures - Map, Queue, Priority Queue, Set, List, MultiMap, atomic long, Semaphore, Lock, ID generator and others.

It is worth noting that Hazelcast IMDG supports distributed queries. Each node in the cluster receives the requested predicate, processes it and filters data, then returns it to the sender, which merges all the obtained results. The big advantage is speed of execution and scalability, the more members in the cluster, the faster the query will be processed. Distributed queries can be utilized using Criteria API and SQL-like predicates.

Hazelcast has the capability to use distributed SQL. This functionality is in beta and is compatible with distributed Map data structure.

Hazelcast supports two types of transactions, one-phase and two-phase. If there is a conflict during a commit, a one-phase transaction can lead the system to a non-consistent state because there is no preparatory phase. The two-phase transaction has a preparation and a commit, so the second phase will not be performed if there is a conflict in the first one. If the speed is a more important system requirement, it is better to employ one-phase transaction. If it is necessary to guarantee data consistency, then the performance is sacrificed and two-phase transaction is chosen.

Eviction policies realised in Hazelcast are TTL, LRU and LFU.

To monitor a cluster Hazelcast IMDG provides various mechanisms:

- Script for the health check.
- A health monitor that periodically writes out information to console.
- Management Center enables to analyze and browse used data structures, update configurations and get members' memory information.

Additionally, Hazelcast has a large community and a variety of resources that help educate users and assist in problem solving.

2.3.4 Aerospike

The information provided in this section was taken from official documentation [37].

Aerospike is a distributed NoSQL key-value database.

There are three levels in the architecture of Aerospike:

- Client Layer consists of open source client libraries that track nodes, implement Aerospike APIs, and are aware of the cluster's data location.
- Clustering and Data Distribution Layer controls connections inside the cluster and automates data movement, intelligent rebalancing, replication, and failover.
- Data Storage Layer stores data in RAM or on SSDs.

The Aerospike database does not require a traditional RDBMS schema. Data is organized using bins and namespaces, which are similar to RDBMS columns and databases. Each bin supports the following data types and structures: integer, string, float, list, map, binary objects and others. Every namespace has 4096 partitions, which are evenly distributed between cluster nodes. Aerospike performs a hashing process that maps a newly added record to a single partition. Partition are also replicated to one or more nodes, then a single node conducts read and write operations for a partition, while others act like read-only replicas for the partition. The replication factor can be also set to 1, meaning that there is only one copy of data within the database. It is important to note that Aerospike's data rebalancing mechanism makes a node failure seamless, the system stays continuously available.

Aerospike does not provide ACID transactions. However, it uses Strong Consistency algorithm, which disallows any potentially conflicting writes in the cluster. Furthermore, by committing write operations to a number of physical servers with different hardware components Aerospike ensures durability.

Aerospike has a shared-nothing architecture, where every node is identical, there is no master node and no single point of failure.

Aerospike does not support SQL, but offers its own tool for browsing the data - AQL. Despite the similarity in names, there are syntactic differences between these languages.

The only eviction policy implemented in Aerospike is TTL.

Aerospike also provides a special monitoring tool - Aerospike Monitoring Stack. It supports integration with third-party monitoring tools and can be configured in various ways. An example of a configuration is shown in the figure below.

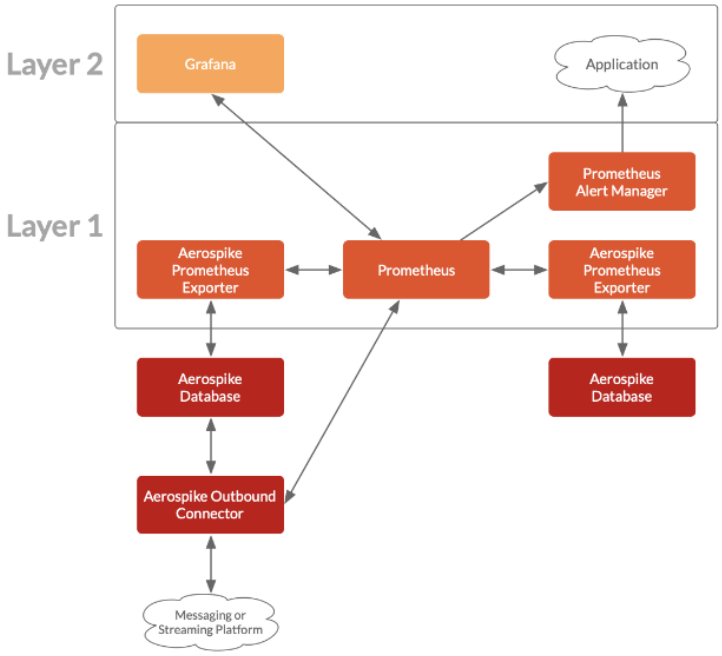

Figure 2.12. Aerospike Monitoring Stack example.

Among the programming languages that can be used with Aerospike are C#/.NET, C/C++, Java, Go, Python.

2.3.5 Analysis conclusion

All technologies described above provide very similar functionality. They are high performant, scalable, able to maintain data consistency, offer APIs for a large number of programming languages and each of them has its own advantages and disadvantages. I would like to note that from my point of view, when choosing platform from this list, one should rely on the specific requirements of the system.

2.4 Architectural approaches

In terms of organizing the structure of an application, there are two main architectural styles - monolithic and microservice.

2.4.1 Monolithic architecture

A monolithic application is a single indivisible unit into which all the components are combined, including client user interface, server application, and database. All functions of this application are managed in one place. This option makes development from scratch simpler and, due to shared code and memory, provides faster communications between system components. However, over time, the monolith code base turns unwieldy, posing challenges in terms of navigation and modification. Difficulties may also emerge when incorporating new technologies, often necessitating extensive rewrites of significant parts of the application, resulting in both cost and time implications. The limited flexibility of monoliths is another disadvantage, for example, if a bug occurs in one component of the application, it affects the entire system. In addition, every implementation change requires re-deployment of the entire application, which can slow down development significantly.

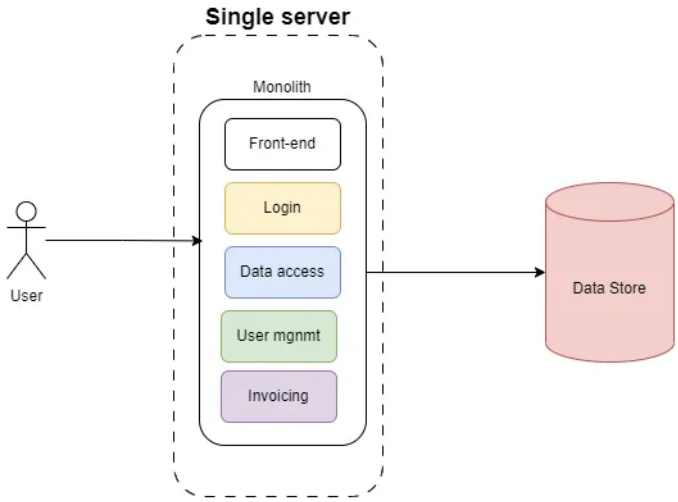

Figure 2.13. Monolithic architecture example [9].

2.4.2 Microservice architecture

Microservices represent individual system components, each functioning as a small independent application with its dedicated database, and offering APIs for seamless communication with other services. These components might employ varied technologies and could potentially be implemented using diverse programming languages. There are various alternatives for realizing communication between microservices, such as network requests or messages. This architectural approach allows system units to be scaled independently and offers flexibility for implementing changes. Even if an error occurs in one of the components, the application continues to operate without a loss of functionality. A clear advantage of this approach is the capability for independent development by multiple teams of programmers. When modifications are made to the application code, it is necessary to deploy only the specific component where those changes took place. Nevertheless, this strategy introduces specific challenges, including complexities in management, maintaining data integrity, and conducting thorough testing.

[9] Diagram was downloaded from https://camunda.com/blog/2023/08/monolith-vs-microservice-architecture-comparison

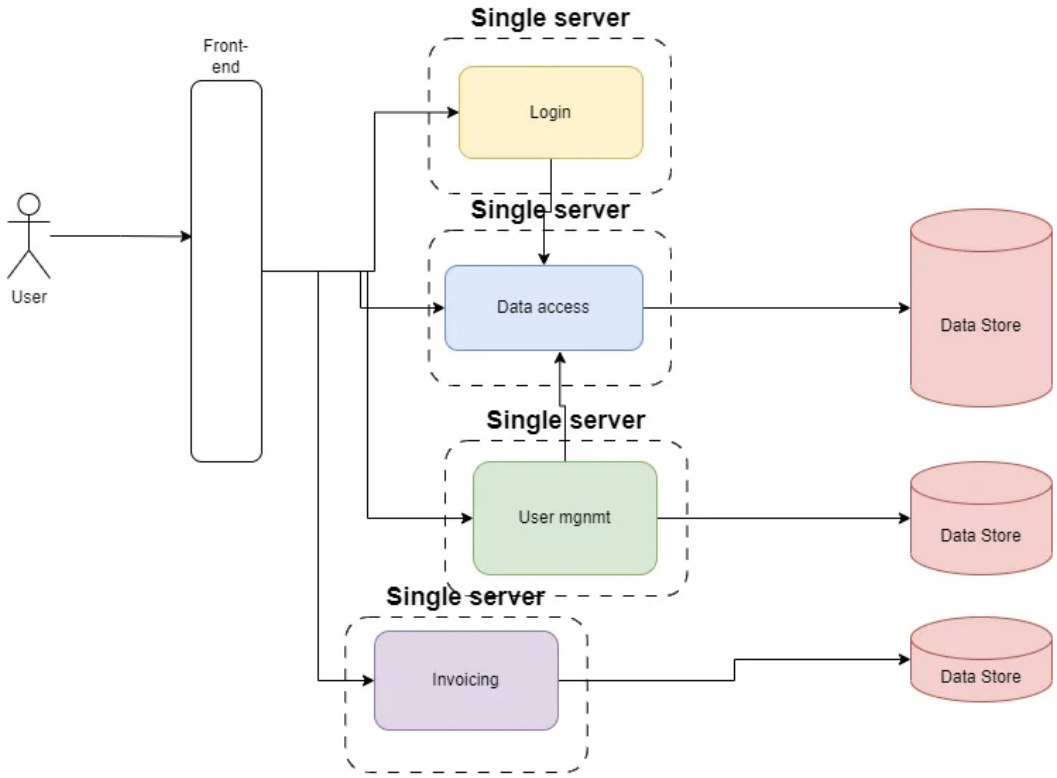

Figure 2.14. Microservice architecture example [10].

2.5 Event streaming

In the context of a software system, events are significant occurrences or changes in state. Event streaming is a practice of capturing and processing these event in real-time. This enables efficient and scalable transmission of information among different components or systems.

2.5.1 Related definitions

Following this, a list of key elements associated with event streaming will be provided.

- Event producers are components or systems that generate events.
- Event consumers refer to components or systems that subscribe to and handle events.
- Event brokers receive events from producers and distribute them to the relevant consumers. Message queues [11] and publish-subscribe patterns [12] are commonly used to implement the events distribution.
- Event-driven architecture is an architectural style where components interact by exchanging events. It fosters loose coupling among various parts of a system, enhancing flexibility, scalability, and ease of maintenance.

Several technologies and platforms provide event-streaming functionality, some of them are Apache Kafka, RabbitMQ and Redis.

[10] Diagram was downloaded from https://camunda.com/blog/2023/08/monolith-vs-microservice-architecture-comparison

[11] Message queues represent a point-to-point communication system where messages are directed to a single receiver, ensuring ordered and reliable delivery.

[12] Publishers send messages to specific topics, and subscribers receive messages from the message broker based on their topic subscriptions.

2.6 Change Data Capture

Change Data Capture (CDC) is a technique employed to recognize and capture alterations made to data within a database. Often these modifications are propagated to other systems. CDC is mainly used in scenarios where it is essential to keep multiple data sources consistent and up-to-date.

Examples of platforms that provide CDC functionality are Debezium, Oracle Golden Gate, Keboola and IBM InfoSphere Change Data Capture.

2.7 Application containerization

2.7.1 What is a containerized application?

Containerized applications are programs executed within isolated packages of code known as containers. These containers encapsulate all the necessary dependencies for an application to run on any host operating system, incorporating libraries, binaries, configuration files, and frameworks into a unified, lightweight executable [38]. The containerized application consists of multiple elements such as app components, forming the container image. This image, serving as the architecture of the container system, is subsequently executed by the container engine.

2.7.2 Advantages

- As containers are platform-independent, they can be used on nearly any environment that has a support for the container runtime.
- Containers are lighter and more efficient than other virtualization methods, because they carry a minimum information needed.
- It is comparatively easy to add more container instances to scale up the application if there is such a demand.
- Each container has its dependencies isolated and runs in private environment. Which means that there is a possibility of running multiple applications on the same host.
- Due to the fact that containers are lightweight and share the host system's resources, they are resource efficient.

2.7.3 Disadvantages

- Since containers share the same resources, it can bring security issues, when one container affects others.
- It is a non-trivial task to manage containers in large-scale environments.
- Containers operate within a designated container runtime.
- Often containers do not store any data or state.

Chapter 3
Application design

Since the practical part of this work involves the implementation of the application, this chapter introduces its requirements, architectural design, and data model.

3.1 System requirements

3.1.1 Functional requirements

Functional requirements define what the system will enable users to do.

- The system will allow to check if a particular customer exists.
- The system will allow to check if a particular customer is eligible for a particular product.
- The system will provide a list of available products for a specific customer.

3.1.2 Nonfunctional requirements

Nonfunctional requirements are used to specify various system qualities and attributes that are not directly related to their functionality. The design of the application is often guided by them.

- The system will be easily scalable.
- The system will provide relevant data.
- The system will be available at any time.
- The system will be highly efficient.

3.2 Architecture concept

Based on the application requirements, an architectural concept was created consisting of the following components.

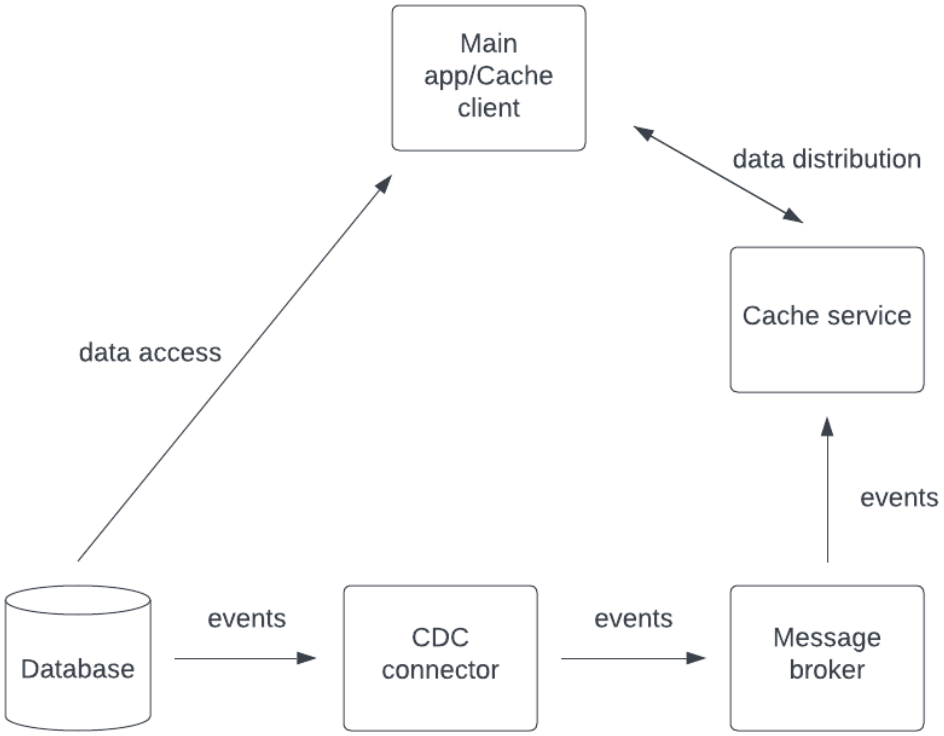

Figure 3.1. Architecture concept visualisation.

- Database - main repository of information.
- Because the data in the main storage can be changed by third parties, we need to keep it up to date in the cache. It is the CDC connector that will track changes in tables, generate events with information about them and send them to the message broker.
- The Message broker will receive events generated by the CDC and deliver them to the cache client.
- Based on the received events, the cache client will make appropriate changes to the data it keeps.
- The main application will be a cache client. In other words, it will process requests received from outside and check the cache. In case of a cache hit the data will be returned immediately, in case of a cache miss this component will query the database, write the data to the cache and return the corresponding result.

3.3 Cache design

3.3.1 General information

The cache will be implemented in a client-server topology using IMDG technology. This will ensure that the system can maintain high efficiency, data consistency among cache cluster nodes and seamless scalability for future development. Since the application should have access to the relevant information to the maximum extent possible, the Cache-Aside in combination with Refresh Ahead pattern was chosen for the implementation. That is, even if the cache fails, the main application will be able to serve the request and return up-to-date data from the database.

3.3.2 Caching method

Two alternatives for caching data were considered when creating the application design:

- At system startup, all data stored in the database are automatically written to the cache. After that, the cache becomes the main resource of information in the system, i.e. all subsequent operations with data are propagated to the cache. In this case, the cache client has a read-only access to the cache. This approach provides very fast access to data, but consumes more memory because data is stored in several places at once - in the database, in the cache and in the broker. In addition, this method results in the fact that the cache stores unnecessary information, i.e. information that has not been requested yet and probably will never be requested. Also, the cache server processes and conducts as many operations as occurred with the data in the database, which can lead to increased load on the cluster.
- At system startup, the cache remains empty and is populated by cache client only when a request for specific data is received. And the combination of CDC, message broker and cache server is only responsible for keeping the cache up to date. This approach gives less performance, because the result of each first request will be a cache miss. However, in this case the cluster can consist of fewer nodes, the cache stores less information and only that which has already been requested, and the cache client performs fewer cache operations in the cluster.

After comparing the advantages and disadvantages of each approach, I decided to utilize the second option.

3.4 Data model

3.4.1 Database data model

The diagram below shows the relation between the entities described in the assignment.

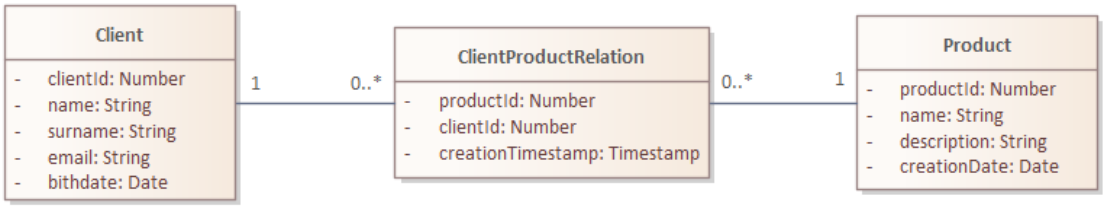

Figure 3.2. Database data model visualisation.

3.4.2 Cache data model

According to the fact that the cache is a key-value store the following data model was chosen.

Key	Value
clientId	object of (clientId, name, surname, email, birthdate)

Table 3.1. Clients map structure visualisation.

Key	Value
productId	object of (productId, name, description, creationDate)

Table 3.2. Products map structure visualisation.

Key	Value
clientId	set of (object of (productId, creationTimestamp))

Table 3.3. Relations map structure visualisation.

Chapter 4
Practical part

This chapter provides an overview of the technologies used to create the application, examples of its configurations and main components.

4.1 Technology stack

The technology stack selected to implement the architecture outlined in the preceding chapter includes:

- Oracle Database
- Debezium
- Kafka Connect
- Apache Kafka
- Hazelcast IMDG
- Java
- Spring Boot framework
- Docker

4.2 Technologies introduction

In this section, I aim to introduce technologies utilized in the implementation that were not previously described.

4.2.1 Oracle Database

Oracle Database is a powerful and widely used relational database management system developed by Oracle Corporation.

To understand the configuration of the implemented system, it is necessary to have a general idea of the Oracle Database structure. As stated in the official documentation, starting in Oracle Database 21c, which was employed in the system implementation, a multitenant container database is the only supported architecture [39]. Container Database (CDB) is the main container that holds one or more Pluggable Databases (PDB). CDB contains the Oracle system data, common users and resources shared by all PDBs. PDB is an individual, separate database within CDB. It has its own data, tablespaces, schemas, isolated from other PDBs.

4.2.2 Apache Kafka

Kafka provides a robust and scalable solution for managing data flows between different applications and components in a distributed environment. Kafka organizes data using two key concepts: topics and partitions.

- Topics serve as channels through which data is transmitted. They can be conceptualized as logical channels that aggregate related data. Each topic may have from zero to an unlimited number of producers and consumers. Events stored in topics persist even after being read and the duration of data retention in a topic is user-configurable. It's important to note that in Kafka, events are immutable and cannot be altered once they have been added to a topic.
- Partitions are physical separate logs belonging to a topic. When a message is received in Kafka, it is appended to the end of the respective partition. Each topic can be divided into multiple partitions. This concept plays a significant role in facilitating horizontal scaling and enhancing performance, as partitions can be processed independently.

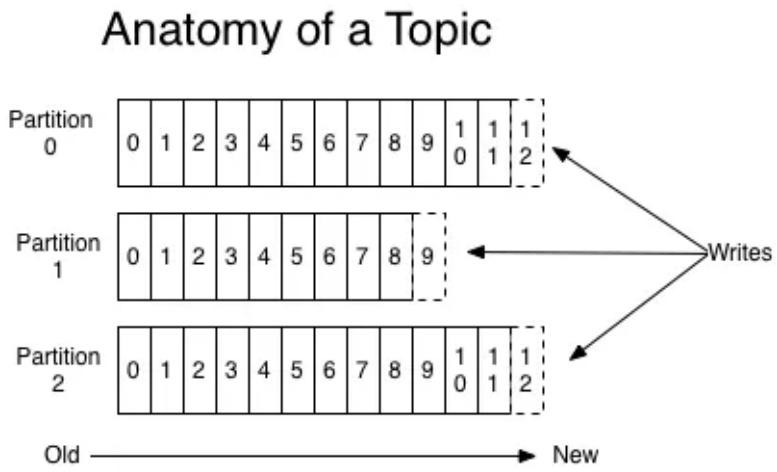

Figure 4.1. Kafka topic structure visualisation [1].

An important note is that Kafka guarantees the order of messages in a partition, but not in a topic. For example, if Kafka writes two events to different partitions, there is a chance that consumer will read these events in the wrong order. However, this problem is solved by adding a key to the message. All events with the same key are added to the same partition. This ensures that the consumer receives messages with the same key in the order they were produced.

Consumer group is a mechanism for organizing and coordinating the distribution of data processing among several consumers for a single topic. Each consumer group is identified by a unique ID. When a new consumer joins a group, the consumer group ID is specified, and Kafka utilizes it to organize and distribute partitions among consumers within the group. Each partition can only be processed by one consumer in the group. Additionally, consumers within a group operate in parallel, handling data from different partitions, thereby enhancing the throughput of data processing.

Another important component in the Kafka ecosystem is Apache ZooKeeper, which provides the coordination and state management in a distributed environment necessary for Kafka to run efficiently. Kafka uses ZooKeeper for:

- Storing broker configuration information and guaranteeing atomicity and consistency when it changes.

[1] `Diagram was downloaded from https://medium.com/javarevisited/kafka-partitions-and-consumer-groups-in-6-mins-9e0e336c6c00`

- Tracking changes in broker structure, availability, and partition reallocation in case of failures.

4.2.3 Kafka Connect

Kafka Connect is a free, open-source component of Apache Kafka that serves as a centralized data hub for simple data integration between databases, key-value stores, search indexes, and file systems [40]. The core concept of Kafka Connect is connectors, which encapsulate the logic for interfacing with external systems and facilitating the transfer of data between Kafka and these systems. Connectors offer the capability to both write data to Kafka and retrieve data from it. Kafka Connect is specifically designed to operate in a distributed environment, enabling the concurrent execution of multiple connector instances to manage substantial data volumes. It further provides a REST API to manage, monitor the status and customize the connectors.

4.2.4 Debezium

Debezium is a collection of distributed services used to capture changes occurring in databases, enabling applications to detect and respond to these changes [41]. Debezium captures all row-level changes within each database table and organizes them into a change event stream. Applications then consume these streams to access the change events in the exact sequence in which they occurred. Debezium is built on top of Apache Kafka and offers a number of Kafka Connect compatible connectors, each of them works with a specific database management system. Connectors record the history of data changes in the DBMS by detecting changes in real-time, generate events and stream them to a Kafka topic. It is important to note that, by default, Debezium uses a primary key from the database as the key for the change event.

4.2.5 Spring Boot

Spring Boot is a popular, open-source framework for building Java-based applications. It provides a variety of tools that reduce the need for boilerplate code, enable fast project bootstrap with pre-configured setups for common use cases, automatically configure the application, suit for building scalable and modular microservice-based architectures and enhance the development experience. Spring Boot also has a large community with comprehensive documentation and support.

4.3 Infrastructure setup

The initial phase of application development involved configuring the infrastructure components.

4.3.1 Prerequisites

- Docker or any tool to work with it, for example Docker Desktop [2]. However, I operated with Docker using Windows Subsystem for Linux [3] console.
- Oracle Database image. It can be installed from Oracle Container Registry website [4]. Oracle Database Enterprise Edition 21c (version 21.3.0.0) was employed in the implementation.

[2] `https://www.docker.com/products/docker-desktop/`
[3] `https://learn.microsoft.com/en-us/windows/wsl/install`
[4] `https://container-registry.oracle.com`

- ZooKeeper image [5].
- Kafka image [6].
- Kafka Connect image [7].
- Oracle Instant Client for Linux [8].

4.3.2 Setup steps

- The first step is to start Docker daemon process using command `dockerd`.
- The next step is the database setup. To execute a Docker container, we need to use the following command.

```
docker run -d --name orcl-bach
-p 1521:1521
-p 5500:5500
-e ORACLE_SID=DEVCDB
-e ORACLE_PDB=DBZ
-e ORACLE_PWD=orclpwd
-e ENABLE_ARCHIVELOG=true
container-registry.oracle.com/database/enterprise:21.3.0.0
```

Figure 4.2. Command to run Oracle Database in container.

This command tells Docker to create and run a Docker container with name `orcl-bach`, map internal ports 1521 and 5500 to external ports 1521 and 5500, create CDB with name `DEVCDB` and set its password to `orclpwd`, create PDB with name `DBZ` and enable archive log mode for future CDC mechanism utilization.

After the container has been successfully started, it is crucial to perform two actions: create schema in PDB, where data model from chapter 4 will be realized, and conduct the appropriate configuration of database to prepare it for integration with Debezium. All configuration steps are provided in Debezium documentation [9] and can be found in db-config.sql script and in figures below.

```
--docker exec -it orcl-bach sqlplus sys/orclpwd@DEVCDB as sysdba
ALTER SESSION SET CONTAINER=DBZ;
CREATE USER BACHDEV IDENTIFIED BY bachdev;
GRANT DBA TO BACHDEV;
COMMIT;
```

Figure 4.3. Create PDB schema with name `BACHDEV` and password `bachdev` with database administrator privileges.

[5] https://hub.docker.com/r/debezium/zookeeper
[6] https://hub.docker.com/r/debezium/kafka
[7] https://hub.docker.com/r/debezium/connect
[8] https://www.oracle.com/database/technologies/instant-client/linux-x86-64-downloads.html
[9] https://debezium.io/documentation/reference/2.4/index.html

```
--docker exec -it orcl-bach sqlplus sys/orclpwd@DEVCDB as sysdba
CREATE TABLESPACE logminer_tbs DATAFILE '/opt/oracle/oradata/DEVCDB/logminer_tbs.dbf' SIZE 25M REUSE AUTOEXTEND ON MAXSIZE UNLIMITED;
COMMIT;
--exit;

--docker exec -it orcl-bach sqlplus BACHDEV/bachdev@DBZ;
CREATE TABLESPACE logminer_tbs DATAFILE '/opt/oracle/oradata/DEVCDB/DBZ/logminer_tbs.dbf' SIZE 25M REUSE AUTOEXTEND ON MAXSIZE UNLIMITED;
COMMIT;
--exit;

--docker exec -it orcl-bach sqlplus sys/orclpwd@DEVCDB as sysdba
CREATE USER c##dbzuser IDENTIFIED BY dbz DEFAULT TABLESPACE logminer_tbs QUOTA UNLIMITED ON logminer_tbs CONTAINER=ALL;
GRANT CREATE SESSION TO c##dbzuser CONTAINER=ALL;
GRANT SET CONTAINER TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$DATABASE to c##dbzuser CONTAINER=ALL;
GRANT FLASHBACK ANY TABLE TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ANY TABLE TO c##dbzuser CONTAINER=ALL;
GRANT SELECT_CATALOG_ROLE TO c##dbzuser CONTAINER=ALL;
GRANT EXECUTE_CATALOG_ROLE TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ANY TRANSACTION TO c##dbzuser CONTAINER=ALL;
GRANT LOGMINING TO c##dbzuser CONTAINER=ALL;
GRANT CREATE TABLE TO c##dbzuser CONTAINER=ALL;
GRANT LOCK ANY TABLE TO c##dbzuser CONTAINER=ALL;
GRANT CREATE SEQUENCE TO c##dbzuser CONTAINER=ALL;
GRANT EXECUTE ON DBMS_LOGMNR TO c##dbzuser CONTAINER=ALL;
GRANT EXECUTE ON DBMS_LOGMNR_D TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$LOG TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$LOG_HISTORY TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$LOGMNR_LOGS TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$LOGMNR_CONTENTS TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$LOGMNR_PARAMETERS TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$LOGFILE TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$ARCHIVED_LOG TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$ARCHIVE_DEST_STATUS TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$TRANSACTION TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$MYSTAT TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$STATNAME TO c##dbzuser CONTAINER=ALL;
COMMIT;
--exit;
```

Figure 4.4. Oracle configuration for Debezium.

The final step in configuring the database involves implementing the data model from the previous chapter. To accomplish this, next SQL queries were created.

```
--docker exec -it orcl-bach sqlplus BACHDEV/bachdev@DBZ;
CREATE TABLE CLIENT (
    ID NUMBER(19, 0) GENERATED ALWAYS AS IDENTITY (START WITH 1 INCREMENT BY 1) PRIMARY KEY,
    NAME VARCHAR2(50) NOT NULL,
    SURNAME VARCHAR2(50),
    EMAIL VARCHAR2(100) UNIQUE NOT NULL,
    BIRTHDATE DATE NOT NULL);

CREATE TABLE PRODUCT (
    ID NUMBER(19, 0) GENERATED ALWAYS AS IDENTITY (START WITH 1 INCREMENT BY 1) PRIMARY KEY,
    NAME VARCHAR2(50) UNIQUE NOT NULL,
    DESCRIPTION VARCHAR2(255) UNIQUE NOT NULL,
    CREATION_DATE DATE NOT NULL);

CREATE TABLE CLIENT_PRODUCT_RELATION (
    ID NUMBER(19, 0) GENERATED ALWAYS AS IDENTITY (START WITH 1 INCREMENT BY 1) PRIMARY KEY,
    CLIENT_ID NUMBER(19, 0),
    PRODUCT_ID NUMBER(19, 0),
    CREATION_DATE TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP AT TIME ZONE 'UTC' NOT NULL,
    CONSTRAINT FK_CLIENT FOREIGN KEY (CLIENT_ID) REFERENCES CLIENT(ID) ON DELETE CASCADE,
    CONSTRAINT FK_PRODUCT FOREIGN KEY (PRODUCT_ID) REFERENCES PRODUCT(ID) ON DELETE CASCADE,
    CONSTRAINT FK_UNIQUE_RELATION UNIQUE (CLIENT_ID, PRODUCT_ID) );

GRANT SELECT ON CLIENT to c##dbzuser;
ALTER TABLE CLIENT ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;
GRANT SELECT ON PRODUCT to c##dbzuser;
ALTER TABLE PRODUCT ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;
GRANT SELECT ON CLIENT_PRODUCT_RELATION to c##dbzuser;
ALTER TABLE CLIENT_PRODUCT_RELATION ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;

COMMIT;
-- exit;
```

Figure 4.5. SQL queries for data model realization.

Here, I would like to emphasize a few key points. IDs are limited to 19 digits, aligning with the maximum size of the Long data type in Java, and are generated automatically when data is added to the table, incrementing by 1 each time. `C##DBZUSER`

must have privileges to select data from created tables and supplemental logging should be enabled for proper Debezium functioning.

- The last step includes the setup of ZooKeeper, Kafka and Kafka Connect with Debezium Connector for Oracle. However, there is one nuance to using `debezium/connect` image [10] with Oracle: due to licensing restrictions some of the dependencies are not a part of the image, so a custom image with essential dependencies should be created from the installed one. For that purpose, we need to create a following Dockerfile and build an image from it using command `docker build -t connect-orcl:latest`, which gives custom image a name `connect-orcl` and version `latest`.

```
FROM debezium/connect:latest
ENV KAFKA_CONNECT_JDBC_DIR=$KAFKA_CONNECT_PLUGINS_DIR/kafka-connect-jdbc

ENV MAVEN_DEP_DESTINATION=$KAFKA_HOME/libs \
    ORACLE_JDBC_REPO="com/oracle/database/jdbc" \
    ORACLE_JDBC_GROUP="ojdbc8" \
    ORACLE_JDBC_VERSION="21.6.0.0" \
    ORACLE_JDBC_MD5=312e6f4ec9932bbf74a4461669970c4b

RUN docker-maven-download central "$ORACLE_JDBC_REPO" "$ORACLE_JDBC_GROUP" "$ORACLE_JDBC_VERSION" "$ORACLE_JDBC_MD5"

USER kafka
```

Figure 4.6. Dockerfile for custom Kafka Connect image creation [11].

After that, we use the Docker Compose file, which will allow us to start all the containers with a command `docker-compose -f docker-compose.yaml up --build`.

```
version: '2'
services:
  zookeeper:
    image: debezium/zookeeper
    ports:
      - 2181:2181
      - 2888:2888
      - 3888:3888

  kafka1:
    image: debezium/kafka
    links:
      - zookeeper
    ports:
      - 9092:9092
    environment:
      BROKER_ID: 1
      ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_LISTENERS: INTERNAL://:29092,EXTERNAL://:9092
      KAFKA_ADVERTISED_LISTENERS: INTERNAL://kafka1:29092,EXTERNAL://localhost:9092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL

  kconnect:
    image: connect-orcl:latest
    build:
      context: /root/bachelors/zkk/docker_steps/connect-orcl
    ports:
      - 8083:8083
    environment:
      CONFIG_STORAGE_TOPIC: my-connect-configs
      OFFSET_STORAGE_TOPIC: my-connect-offsets
      STATUS_STORAGE_TOPIC: my-connect-statuses
      BOOTSTRAP_SERVERS: kafka1:29092
      LD_LIBRARY_PATH: /mnt/c/oic/instantclient-basic-linux.x64-21.12.0.0.0dbru/instantclient_21_12
    links:
      - zookeeper
    depends_on:
      - kafka1
      - zookeeper
```

Figure 4.7. Docker Compose file contents.

[10] `https://hub.docker.com/r/debezium/connect`

[11] `Dockerfile was taken from https://github.com/debezium/debezium-examples/blob/main/tutorial/debezium-with-oracle-jdbc/Dockerfile`

The last action to be performed immediately after successful launch of all containers is registration of Debezium Connector for Oracle. This can be done using a POST request to the REST endpoint `/connectors` exposed by Kafka Connect on port 8083. Payload of the request will be as follows:

```
{
    "name": "client-connector",
    "config": {
        "connector.class": "io.debezium.connector.oracle.OracleConnector",
        "tasks.max": "1",
        "topic.prefix": "server1",
        "topic.creation.default.partitions": "5",
        "topic.creation.default.replication.factor": "1",
        "database.hostname": "172.17.0.1",
        "database.port": "1521",
        "database.user": "c##dbzuser",
        "database.password": "dbz",
        "database.dbname": "DEVCDB",
        "database.pdb.name": "DBZ",
        "schema.history.internal.kafka.bootstrap.servers": "kafka1:29092",
        "schema.history.internal.kafka.topic": "schema-changes-top",
        "table.include.list": "BACHDEV.CLIENT, BACHDEV.PRODUCT, BACHDEV.CLIENT_PRODUCT_RELATION",
        "database.oracle.version": "21.3.0.0",
        "database.server.name": "server1"
    }
}
```


Figure 4.8. Request payload for connector registration.

As soon as Kafka Connect processes the request, Debezium will automatically create Kafka topics based on `table.include.list` property specified in request payload. Debezium names topics using `databaseServerName.schemaName.tableName` pattern. These topics will be utilized for storing events that occurred in corresponding tables in database.

4.4 Cache server

4.4.1 Spring boot project initialization

Cache server is a Spring Boot application that was initialized as a Maven project using the Spring Initializr tool, incorporating the following components:

- Spring Kafka - provides abstractions and integration support for working with Apache Kafka.
- Spring Boot DevTools - provides features to enhance the development experience.
- Lombok - provides annotations to reduce boilerplate code in Java classes.
- Spring Boot Starter Web - simplifies the setup for developing web application in Spring Boot.
- Jackson Databind library - used for processing JSON data.
- Hazelcast - used to integrate Hazelcast IMDG with Java application.
- Spring Boot Starter Test - provides support for testing Spring Boot applications.

4.4.2 Overview

Cache server is divided into 3 main components:

- Kafka Consumers - consume events generated by Debezium and send them to event processor facade.

  Example of a Kafka consumer implementation for processing messages from `server1.BACHDEV.CLIENT` topic:

```
@KafkaListener(topics = KafkaTopics.CLIENT_TOPIC,
               groupId = KafkaTopics.GROUP_1)
    public void consumeClient(
            @Header(KafkaHeaders.RECEIVED_KEY) String key,
            @Payload(required = false) String payload
    ) {
        if (ObjectUtils.isEmpty(key)) {
            log.warn("kafka message key is null");
        } else if (ObjectUtils.isEmpty(payload)) {
            log.warn("kafka message value is null");
        } else {
            eventProcessorFacade.processClientEvent(key, payload);
        }
    }
```

- Event processor facade - serves as an orchestrator. Based on received operation type and message, this component decides what cache service method should be invoked.

  Example of an event processor method:

```
@Override
public void processClientEvent(String messageKey, String messageValue)
{
    String changedRowId =
        JsonMessageKeyDeserializer.getChangedRowId(messageKey);
    IMap<String, Client> clientsMap =
        hazelcastInstance.getMap(CacheMapNames.CLIENTS_MAP);

    if (clientsMap.containsKey(changedRowId)) {
        CacheOperations cacheOperation =
            JsonMessageValueDeserializer.
                getOperationType(messageValue);

        switch (cacheOperation) {
            case CREATE:
                break;
            case UPDATE:
                Client client = JsonMessageValueDeserializer.
                        getClient(messageValue);
                cacheService.
                    updateClient(changedRowId, client, clientsMap);
                break;
            case DELETE:
                IMap<String, Set<ClientProductRelationCacheValue>>
                relationsMap =
                    hazelcastInstance.
                        getMap(CacheMapNames.RELATIONS_MAP);
                cacheService.deleteClient(
                        changedRowId,
                        clientsMap,
                        relationsMap);
                break;
```

```
            default:
                log.warn(String.format(
                    "Operation \%s ignored " +
                        "while processing client event.",
                    cacheOperation.getOperation()));
        }
    }
}
```

- Cache service - the component that uses Hazelcast API to operate with the distributed key-value store. All Hazelcast API methods used in code guarantee atomicity of operations.

  Example of an update operation:

```
@Override
public void updateClient(
    String changedRowId,
    Client client,
    IMap<String, Client> clientsMap) {
    clientsMap.replace(changedRowId, client);
}
```

4.4.3 Hazelcast configuration

Hazelcast IMDG is configured in hazelcast.yml file. The configuration of cluster and distributed maps is as follows:

```
hazelcast:
  phone:
    home:
      enabled: false

server:
  cluster-name: bachelors-hz-cluster
  properties:
    logging-type: slf4j
  network:
    port:
      auto-increment: true
      port: 5701

    join:
      auto-detection:
        enabled: true
      multicast:
        enabled: true
        multicast-timeout-seconds: 15
        multicast-time-to-live: 32

  map:
    clients:
```

```
      backup-count: 1
      time-to-live-seconds: 86400 #1 day
      max-idle-seconds: 0
      eviction:
        size: 2147483647
        max-size-policy: PER_NODE
        eviction-policy: LRU
      merkle-tree:
        #Data structure used for
        #efficient comparison of the
        #difference in the contents
        #of large data structures
        enabled: true
        depth: 10
        #precision of comparison is
        #defined by depth

    products:
      backup-count: 1
      time-to-live-seconds: 86400 #1 day
      max-idle-seconds: 0
      eviction:
        size: 2147483647
        max-size-policy: PER_NODE
        eviction-policy: LRU
      merkle-tree:
        enabled: true
        depth: 10

    relations:
      backup-count: 1
      time-to-live-seconds: 86400 #1 day
      max-idle-seconds: 0
      eviction:
        size: 2147483647
        max-size-policy: PER_NODE
        eviction-policy: LRU
      merkle-tree:
        enabled: true
        depth: 10
```

4.4.4 Docker image

Despite the fact that during testing the cache server was executed on localhost, it can also be containerized using the following Dockerfile.

```
FROM amazoncorretto:17-alpine3.17-jdk
VOLUME /tmp
ARG JAR_FILE=target/hz-server-side-develop.jar
COPY \${JAR_FILE} cache-server.jar
EXPOSE 8080
```

```
ENTRYPOINT exec java -jar cache-server.jar
```

4.5 Cache client

4.5.1 Spring boot project initialization

Cache client is a Spring Boot application that was initialized as a Maven project using the Spring Initializr tool, incorporating the following components:

- Spring Boot Starter Actuator - allows to monitor the application, exposes endpoint which show information about app's health, metrics, environment and more.
- Spring Boot Starter Data JPA - allows to interact with data stores with the help of Java Persistence API.
- Oracle JDBC driver - necessary for establishing connection between Java application and Oracle Database.
- MapStruct - generates code for mapping between Java beans.
- MapStruct Lombok Binding - provides integration between MapStruct and Lombok.
- Spring Doc OpenAPI Starter WebMVC UI - allows to integrate OpenAPI with Spring Boot. OpenAPI generates documentation for APIs in app.
- Spring Boot Starter Web - was introduced previously.
- Hazelcast - was introduced previously.
- Spring Boot Dev Tools - was introduced previously.
- Lombok - was introduced previously.
- Spring Boot Starter Test - was introduced previously.

4.5.2 Overview

Cache client has a structure of a classic MVC application, it includes following layers:

- Entity layer contains Java classes that are mapped to Database Tables. Example of a client entity:

```
@Entity
@NoArgsConstructor
@AllArgsConstructor
@Data
@ToString
@Table(name = "CLIENT")
public class ClientEntity {

    public static final String PK = "ID";

    @Id
    @Column(name = PK,
            unique = true,
            updatable = false,
            insertable = false)
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "NAME", length = 50, nullable = false)
```

```
    private String name;

    @Column(name = "SURNAME", length = 50)
    private String surname;

    @Column(name = "EMAIL",
            length = 100,
            unique = true,
            nullable = false)
    private String email;

    @Column(name = "BIRTHDATE", nullable = false)
    private Date birthDate;

    @ToString.Exclude
    @EqualsAndHashCode.Exclude
    @OneToMany(
            mappedBy = "client",
            fetch = FetchType.LAZY,
            cascade = {CascadeType.REFRESH, CascadeType.REMOVE})
    private List<ClientProductRelationEntity> relationEntityList;

}
```

- Data Access Object - layer which serves for querying the database. Example of a method to get client entity by id from database, using Criteria API:

```
@Override
    public ClientEntity getClient(Long id) {
        HibernateCriteriaBuilder cb =
            session.getCriteriaBuilder();
        JpaCriteriaQuery<ClientEntity> q =
            cb.createQuery(ClientEntity.class);
        Root<ClientEntity> root =
            q.from(ClientEntity.class);

        q.select(root)
                .where(
                        cb.equal(root.get("id"), id)
                );

        Query<ClientEntity> query = session.createQuery(q);
        return query.getSingleResultOrNull();
    }
```

- Resource access (RA) layer - layer which communicates with data resources. RA has access to DAO and handles all caching logic in the application. Example of a method for client retrieval:

```
@Override
    public boolean clientExists(Long id) {
        IMap<String, Client> clientsMap =
```

```
            hazelcastInstance.getMap(CacheMapNames.CLIENTS_MAP);
        //if key is in cache - return true
        if (clientsMap.containsKey(id.toString())) {
            return true;
        }

        //if key is not there, go to database
        ClientEntity clientEntity = clientDAO.getClient(id);

        //if nothing is found in db, return false
        if (clientEntity == null) {
            return false;
        }

        //if client is found in db, write it to cache
        Client foundClient =
            clientRaMapper.mapClientEntityToDomain(clientEntity);
        clientsMap.put(foundClient.getId().toString(),
                       foundClient);

        return true;
    }
```

- Service layer - the role of this layer is to implement business logic and communicate with RA.
- Online manager - the layer where all REST controllers are realized. Example of a REST controller method:

```
@GetMapping("/client")
public ResponseEntity<Boolean> clientExists(
        @RequestParam("clientId") @NotNull Long clientId
) {
    boolean clientExists = clientService.clientExists(clientId);
    return ResponseEntity.ok(clientExists);
}
```

4.5.3 App configuration

Next snippet provides an overall app configuration and most importantly the datasource configuration.

```
server:
  port: 8081

springdoc:
  show-actuator: true
  swagger-ui:
    path: /main-app/swagger-ui.html
    enabled: true

main-app:
```

```
  appInfo:
    appName: "Bachelor's Main App"
    description: "Main and Hazelcast client-side app"
    version: "develop"
  datasource:
    url: "jdbc:oracle:thin:@//localhost:1521/DBZ"
    driver: "oracle.jdbc.driver.OracleDriver"
    schema: "BACHDEV"
    username: "BACHDEV"
    password: "bachdev"
```

4.5.4 Hazelcast configuration

Hazelcast client configuration looks almost the same as the Hazelcast server configuration.

```
hazelcast:
  phone:
    home:
      enabled: false

client:
  cluster-name: bachelors-hz-cluster
  properties:
    logging-type: slf4j
  connection-strategy:
    connection-retry:
      cluster-connect-timeout-millis: -1

  map:
    clients:
      backup-count: 1
      time-to-live-seconds: 86400 #1 day
      max-idle-seconds: 0
      eviction:
        size: 2147483647
        max-size-policy: PER_NODE
        eviction-policy: LRU
      merkle-tree:
        enabled: true
        depth: 10

    products:
      backup-count: 1
      time-to-live-seconds: 86400 #1 day
      max-idle-seconds: 0
      eviction:
        size: 2147483647
        max-size-policy: PER_NODE
        eviction-policy: LRU
      merkle-tree:
```

```
        enabled: true
        depth: 10

    relations:
      backup-count: 1
      time-to-live-seconds: 86400 #1 day
      max-idle-seconds: 0
      eviction:
        size: 2147483647
        max-size-policy: PER_NODE
        eviction-policy: LRU
      merkle-tree:
        enabled: true
        depth: 10
```

4.5.5 Documentation

As OpenAPI was employed for documenting REST endpoints, here is an example of the generated documentation, which can be accessed from the web browser on address `/main-app/swagger-ui.html`.

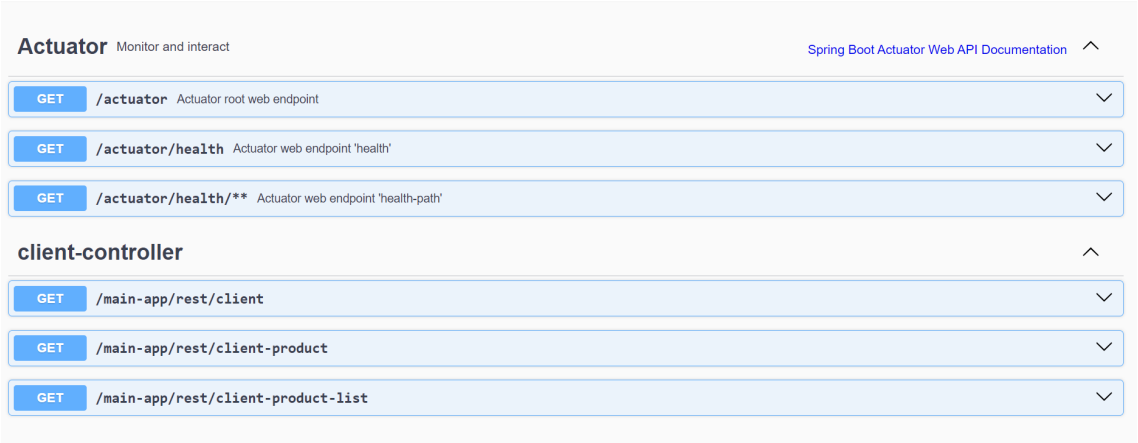

Figure 4.9. OpenAPI documentation example.

4.5.6 Docker image

Despite the fact that during testing the cache client was executed on localhost, it can also be containerized using the following Dockerfile.

```
FROM amazoncorretto:17-alpine3.17-jdk
VOLUME /tmp
ARG JAR_FILE=target/main-app-develop.jar
COPY \${JAR_FILE} cache-client.jar
COPY src/main/resources/application.DEV.yml /app/config/
EXPOSE 8080
ENTRYPOINT exec java -jar cache-client.jar
--spring.config.location=/app/config/application.DEV.yml
```

Chapter 5
Testing

This chapter covers the methodology for testing of the developed application, introduces various test scenarios, and presents the results of executed test cases.

5.1 Testing approach

Since the goal of the practical part of the work is the implementation of a distributed cache, the testing will focus on this system component. The cache should be evaluated for its impact on enhancing system performance and supporting data consistency with the database.

5.2 Performance

In order to understand the impact of the cache on data retrieval speed, a comparison of query times for fetching information from both the database and the cache is essential. I suppose that the difference in data retrieval speed depends on the amount of data in the storage, so measurements will be taken at 5000, 10000 and 15000 records. All test cases assume that application is executed with 2 cache server and 4 cache client instances.

```
Members {size:6, ver:8} [
    Member [10.1.48.50]:5701 - 8b3c55e2-52a8-4dd4-9642-e2e89cb0e0be
    Member [10.1.48.50]:5702 - 1455c510-f424-4685-8be6-ba49a5741bba this
    Member [10.1.48.50]:5703 - 9b27c2ef-852b-42cb-be97-b52b2960cb87
    Member [10.1.48.50]:5704 - a9cbf3b4-655e-4a3a-8c0a-f468a66556a1
    Member [10.1.48.50]:5705 - c8257180-75c1-477f-b8e7-3795ac6226eb
    Member [10.1.48.50]:5706 - e8ffa1c6-dea0-4c74-ada3-410c78cd5374
]
```

Figure 5.1. Cache cluster structure after application startup.

5.2.1 First set of test cases

- Client existence verification.
  This test case requires 5000 clients to be uploaded to the database.

```
SQL> select count(*) from client;

  COUNT(*)
----------
      5000
```

Figure 5.2. 5000 clients uploaded to the database.

As the cache is empty just after the execution, the first request shows the time of retrieval of a client entry from the database. The second request for the same client will show the duration of retrieval from the cache.

Figure 5.3. The first request for the client speed.

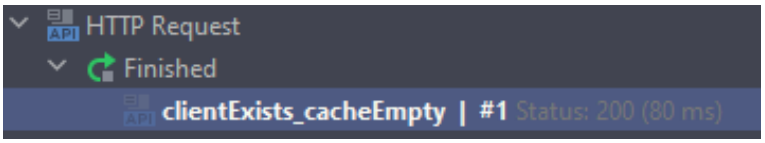

Figure 5.4. The second request for the client speed.

As we can see, the time difference between requests is tremendous. However, after adding about a 1500 entries to cache and performing multiple tests, on this level of data volume the average duration of data access in both stores does not differ much and equals to 65 milliseconds per request.

- Client right for product verification.

  After conducting a number of tests, I came to the conclusion that the cache speeds up the processing of data requests in this operation by almost 2 times. The average performance of this service when accessing the database is 150 milliseconds, and when accessing the cache is 80 milliseconds. This test assumes that `CLIENT_PRODUCT_RELATION` table contains around 5500 entries.

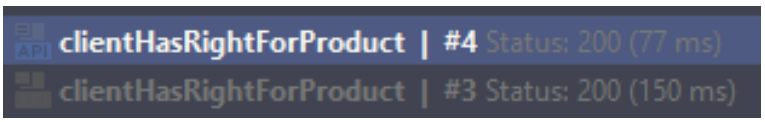

Figure 5.5. Performance difference for client right for product verification service. The second line is a database request and the first one is a cache request.

- Available products for client service.

  The testing of this service also showed that cache brought a two-fold increase in data access efficiency.

Figure 5.6. The first request speed for getting a list of products for a client.

Figure 5.7. The second request speed for getting a list of products for a client.

5.2.2 Second set of test cases

The second set of test cases assumes that each table contains around 10000 entries.

```
SQL> select count(*) from client;

  COUNT(*)
----------
     10001

SQL> select count(*) from product;

  COUNT(*)
----------
     10000

SQL> select count(*) from client_product_relation;

  COUNT(*)
----------
     10000
```

Figure 5.8. Number of rows in each table.

- Client existence verification.

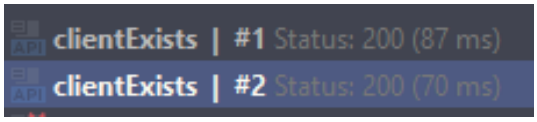

Figure 5.9. Performance statistics for the database and cache query.

- Client right for product verification.

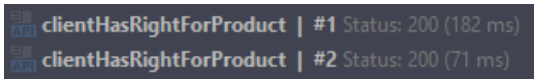

Figure 5.10. Performance statistics for the database and cache query.

- Available products for client.

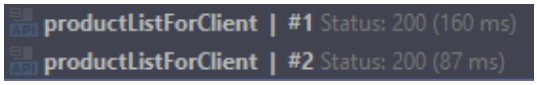

Figure 5.11. Performance statistics for the database and cache query.

5.2.3 Third set of test cases

The third set of test cases assumes that each table contains around 15000 entries.

```
SQL> select count(*) from client;

  COUNT(*)
----------
     15000

SQL> select count(*) from product;

  COUNT(*)
----------
     15000

SQL> select count(*) from client_product_relation;

  COUNT(*)
----------
     15000
```

Figure 5.12. Number of rows in each table.

- Client existence verification.

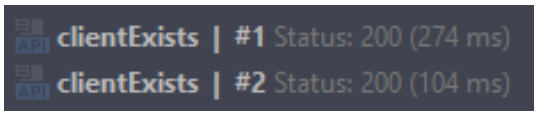

Figure 5.13. Performance statistics for the database and cache query.

- Client right for product verification.

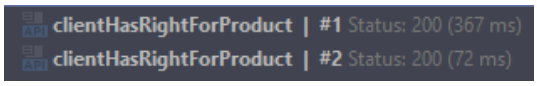

Figure 5.14. Performance statistics for the database and cache query.

- Available products for client.

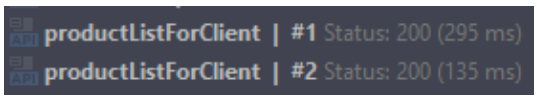

Figure 5.15. Performance statistics for the database and cache query.

5.2.4 Conclusion

The result of the performance testing is that Hazelcast IMDG has indeed accelerated the data access process. The improvement, however, is more noticeable with large amounts of stored information.

5.3 Data consistency

Since third parties have access to the database and can modify the data, it is necessary to test the mechanism for keeping the information in cache consistent, implemented in the application.

The following subsections introduce performed test cases.

5.3.1 Client deletion

- Request for an existing client and his product assignments. By doing that, the corresponding information will be added to cache.
- Delete the client from database.
- Expected result: a repeated request for the client returns `false`. A repeated request for products assigned to him returns an empty list. It means that information was removed from the cache.

5.3.2 Update product

- Request for an existing client product assignments list. After that, all relevant products will be loaded to cache.
- Choose any product from assignments list and update one or more columns of the product entry in database.
- Expected result: a repeated request for the client's product list returns up-to-date information. It means that the product data was successfully updated in cache.

5.3.3 Delete product

- Request for existing product assignment lists related to multiple clients, which will populate cache.
- Delete some product that is contained in cache.
- Expected result: the product is not a part of any cached assignment list. It means that the product was deleted from all values in `RelationsMap` and `ProductsMap`.

5.3.4 Create relation

- Request for an existing products list assigned to a client. After that, cache will be populated with the corresponding information.
- Create a new relation between the client and a product in database.
- Expected result: a repeated request for products assigned to the customer returns up-to-date information. That means, the `RelationsMap` was populated with the new relation.

5.3.5 Delete relation

- Request for an existing products list assigned to a client. After that, cache will be populated with the corresponding information.
- Delete some of the cached relations in database.
- Expected result: a repeated request for products assigned to the customer returns up-to-date information. That means that the product from deleted relation was removed from the `RelationsMap`.

5.3.6 Update product key in relation

- Request for an existing products list assigned to a client. After that, cache will be populated with the corresponding information.
- Choose a cached relation and change its product key value in database.
- Expected result: a repeated request for products assigned to the customer returns up-to-date information, meaning that a product ID was replaced in `RelationsMap` for a corresponding client.

5.3.7 Update client key in relation

- Request for a products list for a Client A and Client B. After that, cache will be populated with the corresponding information.
- Choose cached relations and change its client key value from Client A to Client B in database.
- Expected result: a repeated request for products lists returns up-to-date information, meaning that product ID was removed from Client A products set and added to Client B products set in `RelationsMap`.

5.3.8 Update both keys in relation

It is a combination of workflows described in two previous subsections.

Chapter 6
Conclusion

The main goals of this project were comparative analysis of platforms supporting distributed caching and further implementation of Java Spring Boot web application with distributed cache.

The first step was a research of in-memory distributed caching platforms, which dealt with the main concepts, advantages and disadvantages of Apache Ignite, Redis, Hazelcast and Aerospike.

The next step was to present the design of the future application based on the analysis of system requirements.

The third step involved the selection of technologies and the implementation of the application. The choice of technologies was influenced by my professional experience.

Following implementation, testing was conducted to demonstrate that the cache substantially accelerates the application's data access. Additionally, the testing aimed to verify the implemented mechanism for maintaining data consistency between the database and the cache.

I find this work to be exceptionally beneficial for me. Through it, I gained insights into the concepts of distributed caching, Change Data Capture, and event streaming. Moreover, I acquainted myself with and deepened my understanding of technologies such as Kafka, Kafka Connect, Debezium, and Docker.

In my view, all tasks associated with this work have been successfully accomplished.

6.1 Areas for improvement

There are a few basic ideas for improving the app that have not been implemented due to time constraints.

- Load Balancer implementation. I would like to add another microservice to the application, its role would be to distribute requests between client cache nodes. Spring Cloud Load Balancer library can be utilized for the realization.
- Adding authentication and authorization to cache client microservice. It can be implemented using Spring Security library.
- Connecting security in Kafka for safe change events transmission from the database.
- Using Schema Registry. It is a service that manages the schemas for messages in Kafka. The main benefits of Schema Registry include its capability to handle and supervise changes to message schemas effectively, along with the utilization of more efficient formats during the serialization of messages.

References

[1] Google. *Marketing Strategies. App Mobile.* https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-conversion-data/.

[2] Bloomberg. *Zoom daily users surge to 300M despite privacy woes.* https://www.bnnbloomberg.ca/zoom-daily-users-surge-to-300-million-despite-privacy-woes-1.1425522.

[3] ITIC. *Global Server Hardware, Server OS Reliability Report*. 2016. https://www.winpro.com.sg/wp-content/uploads/2017/07/itic_reliability_2H_2016_wp.pdf.

[4] Deloitte. *Data: a small four-letter word which has grown exponentially to such a big value.* https://www2.deloitte.com/cy/en/pages/technology/articles/data-grown-big-value.html.

[5] Robert Sheldon. *Caching*. 2023. https://www.techtarget.com/whatis/definition/caching.

[6] Pavel Píša, Richard Šusta, Michal Štepanovský, and Miroslav Šnorek. *Computer Architectures. Fast and/or Large Memory – Cache and Memory Hierarchy.* https://cw.fel.cvut.cz/b202/_media/courses/b35apo/en/lectures/04/b35apo_lecture04-cache-en.pdf.

[7] Amazon. *Database caching.* https://aws.amazon.com/caching/database-caching/.

[8] Jhonny Mertz, and Ingrid Nunes. *A Qualitative Study of Application-Level Caching.* 2017. https://ieeexplore.ieee.org/document/7762909.

[9] Cloudflare. *What is caching?* https://www.cloudflare.com/learning/cdn/what-is-caching/.

[10] John Noonan. *Cache Eviction Strategies Every Redis Developer Should Know.* 2023. https://redis.com/blog/cache-eviction-strategies/.

[11] Redis. *Cache Invalidation.* https://redis.com/glossary/cache-invalidation/.

[12] Ashish Teotia. *Types of Caching in Microservices*. 2021. https://medium.com/@ashishteotia/types-of-caching-in-microservices-a68455ba8c45.

[13] Martin Tomasek. *Software Applications Design. Cache.* https://cw.fel.cvut.cz/b222/_media/courses/b6b36nss/prednasky/cache_v2.pdf.

[14] Amazon. *Caching patterns.* https://docs.aws.amazon.com/whitepapers/latest/database-caching-strategies-using-redis/caching-patterns.html.

[15] Moshe Biniel. *Cache strategies*. 2023. https://medium.com/@mmoshikoo/cache-strategies-996e91c80303.

[16] Nikita Koksharov. *What are write-through and write-behind caching?* 2022. https://redisson.org/glossary/write-through-and-write-behind-caching.html.

[17] Kislay Verma. *Architecture Patterns: Caching (Part-1).* 2022. https://kislayverma.com/software-architecture/architecture-patterns-caching-part-1.

[18] AdvanceWorks. *Caching Patterns: Boosting Your Application's Performance and Scalability*. 2023. https://www.linkedin.com/pulse/caching-patterns-boosting-your-applications-performance/.

[19] Nikita Ivanov. *What Is In-Memory Computing?* 2023. https://www.gridgain.com/resources/blog/what-is-in-memory-computing.

[20] Hazelcast. *What Is In-Memory Computation?* https://hazelcast.com/glossary/in-memory-computation/.

[21] MongoDB. *In-Memory Databases Explained.* https://www.mongodb.com/databases/in-memory-database.

[22] Nikita Ivanov. *In-Memory Database vs In-Memory Data Grid: Revisited*. 2023. https://www.gridgain.com/resources/blog/in-memory-database-vs-in-memory-data-grid-revisited.

[23] Hazelcast. *In-Memory Database.* https://hazelcast.com/glossary/in-memory-database/.

[24] Arun Singh. *In-memory databases: use cases and pros-cons*. 2022. https://arunininfosys.medium.com/in-memory-databases-use-cases-and-pros-cons-f68cbee572c0.

[25] Apache Ignite. *IN-MEMORY DATA GRID OVERVIEW*. https://ignite.apache.org/use-cases/in-memory-data-grid.html.

[26] Hazelcast. *What Is an In-Memory Data Grid?* https://hazelcast.com/glossary/in-memory-data-grid/.

[27] Nikita Ivanov. *In-Memory Data Grid: Explained...* 2023. https://www.gridgain.com/resources/blog/in-memory-data-grid-explained.

[28] Sujoy Acharya. *Apache Ignite Quick Start Guide*. Packt Publishing Ltd., 2018. ISBN 978-1-78934-753-1.

[29] Baeldung. *A Guide to Apache Ignite*. 2018. https://www.baeldung.com/apache-ignite.

[30] Neetesh Mehrotra. *Apache Ignite: An Overview*. 2023. https://www.opensourceforu.com/2023/09/apache-ignite-an-overview/.

[31] GridGain. *Introduction to Apache Ignite.* https://www.gridgain.com/resources/papers/introducing-apache-ignite.

[32] Tiago (CERN) Marques Oliveira, Matthias (CERN) Bräger, Brice (CERN) Copy, Szymon (CERN) Halastra, Daniel (CERN) Martin Anido, and Alexander Papa-

georgiou Koufidis. *Distributed Caching at Cloud Scale with Apache Ignite for the C2MON Framework*. 2021.
https://cds.cern.ch/record/2809586/files/document.pdf.

[33] Tomáš Skopal, and Irena Holubová. *B0B36DBS: Database Systems. Lecture 6. Database Transactions.* 2021.
https://cw.fel.cvut.cz/b212/_media/courses/b0b36dbs/lecture-06-database-transactions.pdf.

[34] Apache Ignite. *Apache Ignite Documentation*. 2023.
https://ignite.apache.org/docs/latest/.

[35] Redis. *Documentation*.
https://redis.io/docs/.

[36] Hazelcast. *Hazelcast IMDG Reference Manual 4.2.8*.
https://docs.hazelcast.com/imdg/4.2/.

[37] Aerospike. *Aerospike Technical Documentation*.
https://aerospike.com/docs/.

[38] Google. *Containerized Applications*.
https://cloud.google.com/discover/what-are-containerized-applications.

[39] Oracle. *Oracle Database. Release 21. Database Concepts*.
https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/.

[40] Confluent. *Kafka Connect*.
https://docs.confluent.io/platform/current/connect/index.html.

[41] Debezium. *Debezium Documentation*.
https://debezium.io/documentation/reference/2.4/index.html.

Appendix A
Acronyms and symbols

A.1 List of acronyms

ACID	Atomicity Consistency Integrity Durability
TTL	Time to live
DNS	Domain Name System
API	Application Programming Interface
LRU	Least Recently Used
LFU	Least Frequently Used
MRU	Most Recently Used
FIFO	First-In-First-Out
LIFO	Last-In-First-Out
RR	Random Replacement
ID	Identification
IMC	In-Memory Computing
RAM	Random Access Memory
IMDB	In-Memory Database
IMDG	In-Memory Datagrid
JDBC	Java Database Connectivity
CAP	Consistency Availability Partition Tolerance
TCP/IP	Transmission Control Protocol/Internet Protocol
RDB	Redis Database
AOF	Append Only File
SQL	Structured Query Language
SSD	Solid-state Drive
CDC	Change Data Capture
CDB	Container Database
PDB	Pluggable Database
REST	Representational State Transfer
DBMS	Database Management System
JSON	JavaScript Object Notation
MVC	Model View Controller
UI	User Interface
DAO	Data Access Object
RA	Resource Access