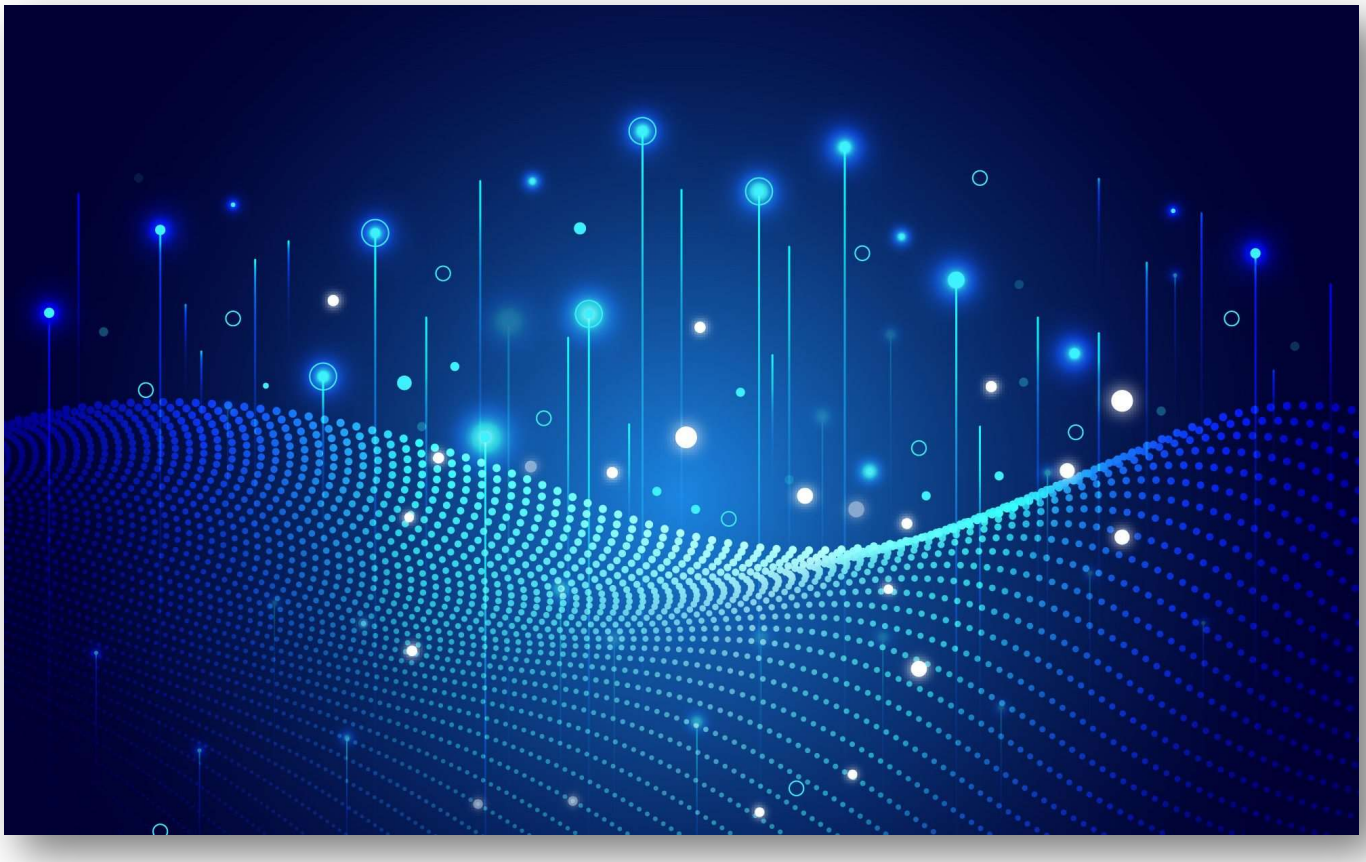
NoSQL Database Management System
Atypon Internship

"The only way to do great work is to love what you do."

Steve Jobs



Yazan Yousef

Table of Contents

Introduction

Our project is a complete NoSQL document-based database management system that incorporates a schema. The system is distributed, utilizing sharding to distribute data across multiple nodes, enhancing reliability and scalability.

The system uses replication to guarantee data availability and is built to accommodate numerous users, such as bank employees who require frequent access to the database for read and write operations.

Docker was used to run the system, which consists of three nodes (workers) and a bootstrapping node to launch the cluster.

All queries were performed using RESTful APIs.

What is NoSQL?

NoSQL refers to a type of database that stores data in a non-relational format, different from the traditional tabular format of relational databases. NoSQL databases can use various data models, such as key-value, document or graph. Our project adopts a document-based approach, it’s also known as "not only SQL."

What does "Document-based" mean?

"The term "document-based" refers to a type of database that stores data in a document format, typically in JSON (JavaScript Object Notation) format. This means that data is stored as a collection of self-contained documents, each containing all the relevant data, his allows for more flexible and dynamic data modeling.

What is a Schema?

A schema is a blueprint that defines the structure of the data we intend to store in the database. To ensure consistency and accuracy, we have implemented a JSON schema that validates every JSON object before storing it in the database.

In MongoDb we can create collections without schemas, but for simplicity we will ask for schema with collection that we create.

User Manual

To run the project, you must have Docker installed on your machine or the machine where you want to run the project. To start the database, use the following command:

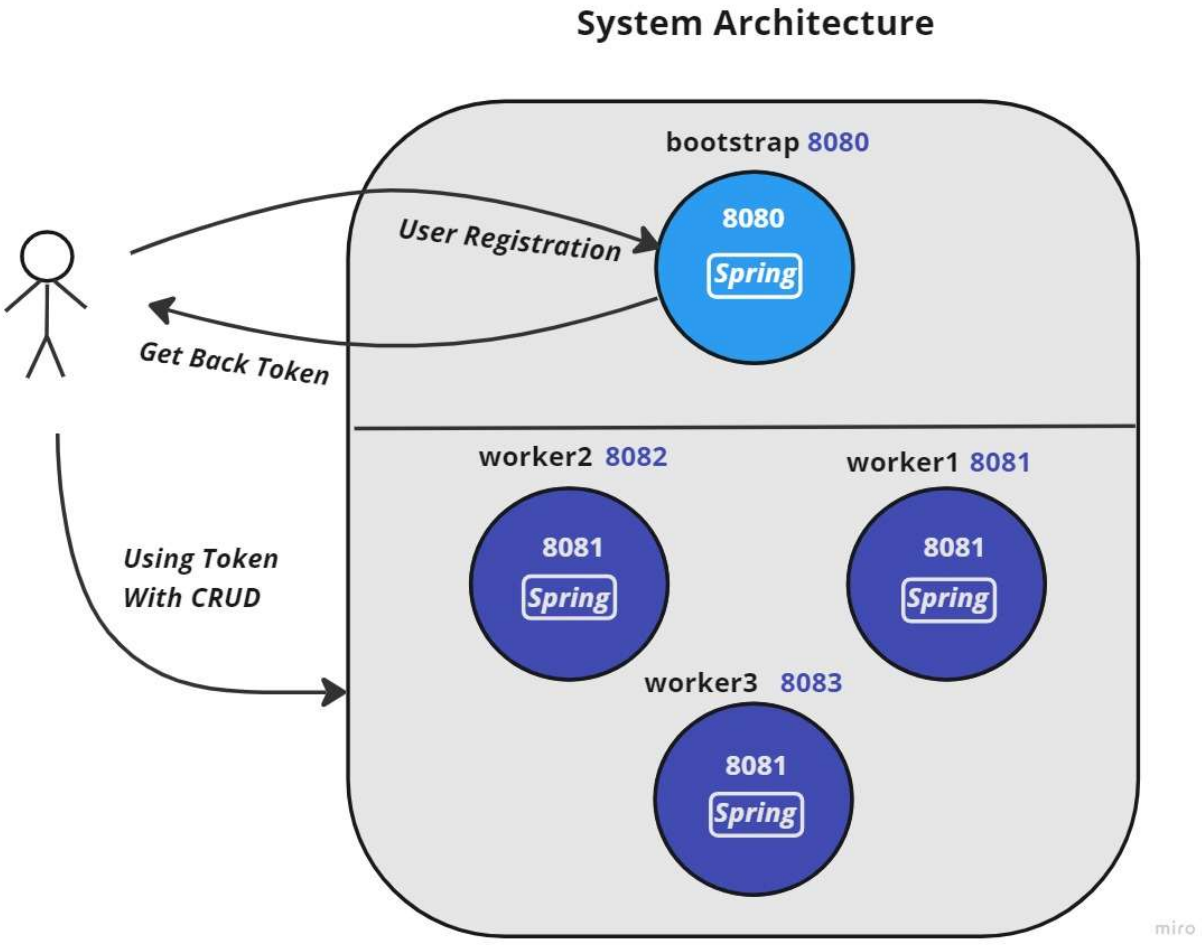
docker-compose up -d

After starting the database, you can run the Banking System application using IntelliJ IDE, any other Java IDE.

After running the Banking Web Application, please enter the IP address for the Database, it will be “localhost” if you’re running the DB on your local machine.

- If you want to join the admins system, consider using this credentials:
  - bootstrappingNode
  - @321bootstrappingNode123@

System Architecture



The system is composed of 3 workers and a bootstrapping node, each running in a separate Docker container. The bootstrapping node's responsibilities include registering new users, assigning them unique tokens, storing them in the database, and providing login credentials to a designated worker.

To ensure communication between nodes, the system relies on REST API calls. Additionally, data replication is

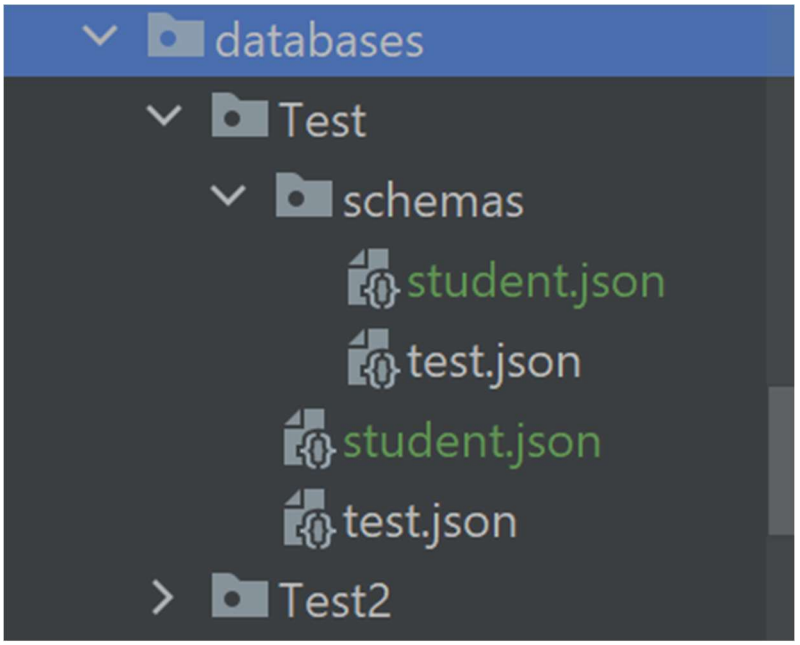
employed to maintain system reliability, meaning that each node has a copy of the data.

- **Database schema**

The file system structure or the database schema is as follows:

Database > Collection > Document.

Inside of the databases directory there is databases such as “Test” and “Test2”.



Each database contains:

1) Schemas directory.
2) All collections inside that DB.

So when we send a request to create a “student” collection inside the “Test” database, two “student.json” files will be created, one under the schema which stores the schema file, and one under the DB directly representing the collection.

Each collection is a JSON Array such that it contains multiple documents with the same schema inside it.

Database Implementation

Used Technologies

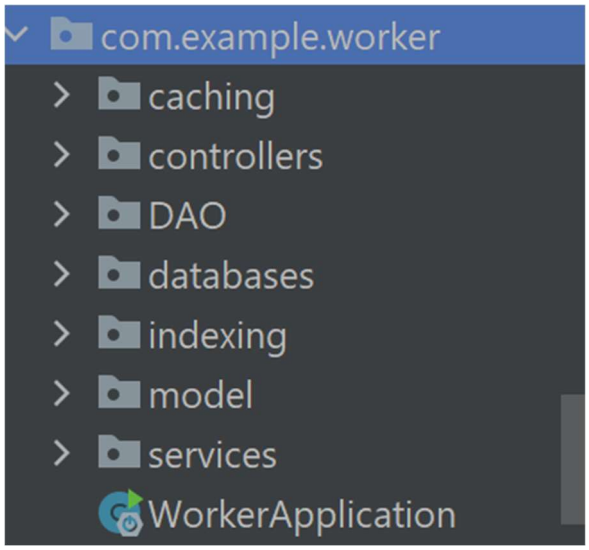
The following technologies were used in our project:

- Spring Boot was utilized to build both the worker bootstrapping module and the demo application. It provided us with an efficient and powerful framework to develop our system.
- Docker was used to run the system on multiple containers. We have four containers running the system, three workers, and one bootstrapping node. Docker ensured that our system is easy to deploy and maintain while ensuring consistency across different environments.

- Maven was used as a build tool and for dependency management. It simplified the process of building and packaging our project, as well as managing our project's dependencies.

Worker Module Structure

We have organized the worker module into several packages:



1) **database**:

This package manages the data storage into JSON files and directories.

2) **DAO**:

This package follows the DAO design pattern and is responsible for all data access to the database.

3) **controller**:
This package contains several controllers for handling HTTP requests in the system:

- **DocumentController:** Responsible for handling requests related to documents, such as adding a new document, getting a document by its ID, and more.

- **DatabaseController:** Responsible for adding new databases, which involves creating a new directory for the database and a “schemas” directory inside it, as discussed in the database schema section.

- **CollectionController:** Responsible for any requests that is related to collections, e.g. filtering documents in a collection by a specific field, retrieving all documents in a collection, and more.

- **CommunicationController:** Responsible for handling communication between all nodes, including messages between workers themselves and messages between the bootstrapping node and workers. Examples of tasks performed by this controller include adding a new user for the worker (received by the bootstrapping node when a new user is registered), retrieving the name of the current worker, and setting the worker as an affinity for the next "write" that happens on the DB in general.

4) **Model:**

This package contains the schema class used to represent the schema of a collection. It also includes an essential class, the "APIResponse" class, used to represent the response of any request in the system. The class contains the status of the request (e.g., 200 for success and 500 for failure) and the response body as a string. This class ensures consistency in receiving response bodies for all requests.

5) **Indexing:**

Indexing is crucial in this project and is discussed in detail later. This package manages the indexing of documents in the system. It includes the PropertyIndex class, representing a single index (dbName, collectionName, propertyName, propertyValue), which is used to retrieve all documents with the same "index." The PropertyIndexManager is responsible for the indexing process.

6) Caching:

Caching is a useful technique to improve system performance. This package implements a simple caching system that follows the LRU (Least Recently Used) algorithm. We'll discuss it in detail later in the report.

7) Services:

This package includes the authentication service sub-package, which stores and manages authenticated users for the current worker. Additionally, it includes the affinity service sub-package, which stores and retrieves affinity data for each document (inside a JSON file) in the entire database, and this data is propagated to all three workers.

Bootstrap Module Structure:

The Bootstrap Module Structure consists of five classes that play a significant role in the functioning of the system:

- Controller class: It handles the incoming HTTP requests in the bootstrapping node, with only three endpoints available: registering a new user, retrieving all active users in the system, and removing a user from the system.

- NetworkManager class: This class sends messages (HTTP Requests) to communicate with the workers in the system. For instance, sending a message to the first worker to set it as the affinity for the next write request in the system.

- LoadBalancer class: Load balancing is crucial in this project since we have three workers. To avoid overloading any worker, this class balances the load between them. Load balancing will be explained in detail later.

- TokensManager class: It handles the tokens of the users or their accounts. It creates new tokens and expires the old ones. Every token is active for only **three hours** for security purposes. This class has a scheduled method that runs every **30 minutes** to check for any expired tokens.

-APIResponse class: It is the same class mentioned earlier in the worker module. This class represents the response of any request in the system, including the status of the request (e.g., 200 for success, 500 for failure) and the response body as a string. It ensures consistency in receiving the response body across all requests.

RESTful API

REST (Representational State Transfer) is an architectural style used for developing web services. A RESTful API is a web service that follows the principles of REST architecture. It provides an interface for clients to communicate with a server over HTTP, using standard HTTP methods such as GET, POST, PUT, DELETE, etc. RESTful APIs use URLs (Uniform Resource Locators) to identify resources, which can be accessed and manipulated using the HTTP methods.

Endpoints in the system:

- Note: All requests require the username and token to be in the header.

Bootstrapping node

- POST api/register/{username} - registers a new user to the system and returns the user's information in a JSON format, including “userName”, “token”, “workerName”.

- GET api/getAllUsers - returns a JSON array of all the users that are currently registered to the system.

- GET api/removeUser/{token} - removes the user with the given token from the system.

Database Controller

- POST api/createDB/{name} - creates a new database with the given name.

- DELETE api/deleteDB/{name} - deletes the database with the given name.

- GET api/listDB - returns a list of all the databases in the system (as a string)

Collection Controller

- POST api/createCol/{db_name}/{collection_name} - creates a new collection with the given name in the given database. This request requires the schema for the collection in the body of the request.

- DELETE api/deleteCol/{db_name}/{collection_name} - deletes the collection with the given name in the given database.

- GET api/filter/{db_name}/{collectionName}?attributeName={name}&attributeValue={value} - returns all the documents that matches the value of given property (in the form of a JSON Array).

- GET api/getCollections/{db_name} - returns a list of all the collections in the given database(as a string)

Document Controller

- POST api/insertOne/{db_name}/{collection_name} - inserts a new document in the given collection in the given database. This request requires the document in the body of the request that matches the schema of the specified collection.

- GET api/getAllDocs/{db_name}/{collection_name} - returns all the documents in the given collection.

- GET api/getDoc/{db_name}/{collection_name}/{doc_id} - returns the document with the given id in the given collection(in the form of JSON object)

- DELETE api/deleteDoc/{db_name}/{collection_name}/{doc_id} - deletes the document with the given id in the given collection.

- POST api/updateDoc/{db_name}/{collection_name}/{doc_id}/{property_name}/{new_value}- updates the value of the given property in the given document.

Communication Controller

- This set of endpoints is related to the system itself it’s not used in the client application.

- POST api/addAffinityData/{id}/{workerName} – is we discussed earlier to keep the load balancing we should have an affinity worker for each document in the DB and it should be known among all workers, so this endpoint is responsible for sharing such information.

- POST api/addAuthenticatedUser/{username}/{token} - adds an authenticated user to the current node. The bootstrap node should receive this request when a new user registers.

- POST api/addAdmin/{newAdminame}/{newAdminToken} - adds a new admin to the current node. The bootstrap node should receive this request when a new admin registers.

- DELETE api/removeAuthenticatedUser/{username}/{token} - removes the authenticated user from the current node.

- DELETE api/removeAdmin/{adminName}/{adminToken} - removes the admin from the current node.

- POST api/setAffinity - marks the current worker as the affinity for the next document that will be added.

- POST api/unsetAffinity - removes the affinity from the current worker.

- GET api/isAffinity - checks whether the current worker is the affinity or not.

- POST api/setCurrentWorkerName/{workerName} - sets the given string as the current worker name.

- GET api/getCurrentWorkerName - returns the current worker name.

- GET api/isAdmin/{username}/{token} - checks whether these credentials belong to one of the admins.

- GET api/isUser/{username}/{token} - checks whether the given credentials belong to an authenticated user or not.

Data Consistency

To ensure data consistency across all workers in the system, a mechanism was implemented using the API. When a write operation is performed on one worker, the same request is broadcasted to all other workers. To prevent endless propagation of requests, a boolean variable called **"X-Propagated-Request"** is included in the header of each write query which is set to be false by default.

If the "X-Propagated-Request" variable is set to false, it indicates that the request should be propagated to all

other workers after the write operation is completed. Once the request is propagated, the variable is set to true to prevent it from being propagated again. If the variable is already set to true, it indicates that the request has already been propagated and should not be propagated again. This mechanism ensures that all workers have consistent data at all times.

Logging

Logging is an essential aspect of any system as it helps with understanding how the system is working and to

debug any potential issues. In this project, the log4j2 library was utilized to log all requests made in the system

or any important actions that were taken.

The Logger class from this library was used to provide various logging levels, including info, warn, and error.

By implementing logging in this project, it was easier to track down errors and understand the flow of requests through

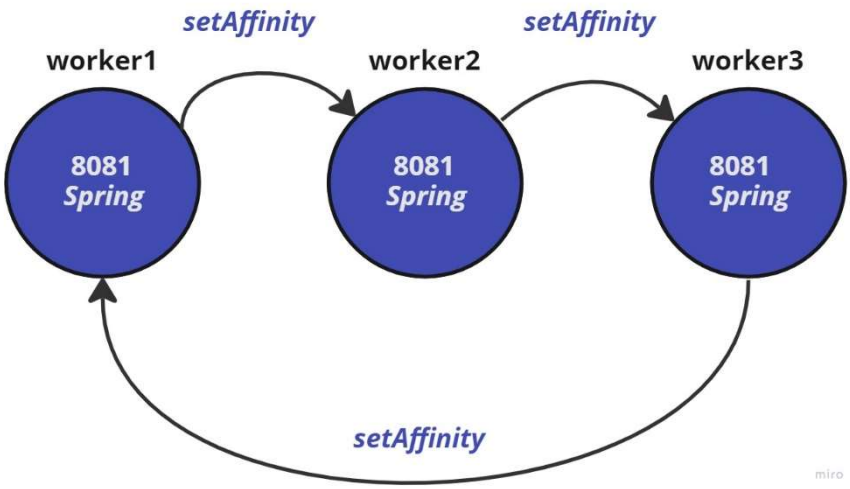
the system. This allowed for more efficient development and a better overall user experience.

Load Balancing:

There were a lot of areas where load balancing was needed in this project:

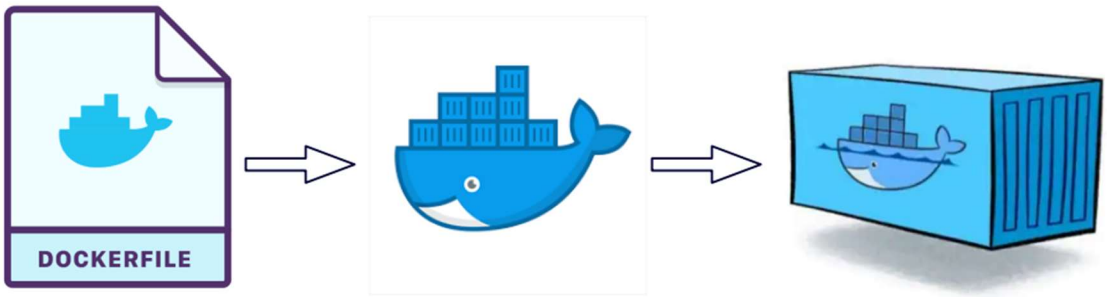
1) **Worker assignment for users** was implemented in the bootstrapping node. Upon user registration, a simple **round-robin** load balancing algorithm was employed to assign workers. Each worker was given an equal number of users to handle. This was achieved through a logical implementation within the Load Balancer class, whereby the next worker in line was identified by incrementing the index and wrapping it around when it reached the end of the list.

2) Assigning affinity worker for the next write request



The three endpoints in each worker, namely **isAffinity**, **setAffinity**, and **unsetAffinity**, play a crucial role in the load balancing algorithm. The bootstrapping node sends a message to the first worker to set it as affinity for the next write request. Once the write request is completed, the first worker sends a request to the second worker to set it as affinity and unset itself as affinity. This process repeats, ensuring that the workload is balanced among the workers, leading to improved efficiency and performance.

Docker



Docker file | Docker Image | Docker Container

Docker was used in the project to containerize the application and its dependencies, ensuring that it can be run consistently on different environments. This allowed for easier deployment and scalability of the application, as well as better isolation and security. Docker images were created for each component of the system, which is the worker and bootstrapping node.

Docker Files:

```
FROM openjdk:17

WORKDIR /app

COPY . .

EXPOSE 8081

CMD ["java", "-jar", "target/Worker-0.0.1-SNAPSHOT.jar"]
```

This is how Dockerfiles looks like, one important thing to keep in mind, is that we should copy all the project to the container, to guarantee that the paths for files are accessed correctly.

Docker-Compose

It will run all container on the intended ports as shown in the sketching at the System Architecture Section

This is for running the **second worker**:

```
worker2:
  build: ./Worker
  restart: always
  container_name: worker2
  ports:
    - "8082:8081"
```

And this part is for running the **bootstrap**:

```
bootstrap:
  build: ./Bootstrap
  restart: always
  container_name: bootstrap
  ports:
    - "8080:8080"
  depends_on:
    - worker1
    - worker2
    - worker3
```

We can see that it depends on

Three of the workers, because it should wait all of them to run in order to give the needed configurations for each of them.

Note: inside the code workers and bootstrap each communicate other nodes by using the container name as a reference.

Security and Authentication

When the user register to the bootstrapping node and gets his own token, he will register using it for one of the three workers, he should include his credentials in the headers of any subsequent requests to one of the three workers, if the credentials is not authorized inside the worker the request will be return 401 Unauthorized user.

This approach of including credentials in the headers is more secure than using path variables.

HTTPSession were used in the demo application to save the credentials for the user once he registers to the system.

The tokens will expire after **3 hours** from generating it, so if someone takes your account he can’t do that much

with it in this time and the admins may expire the token manually from the admins system.

Indexing

Indexing plays a crucial role in this project as it enables efficient searching of documents with common properties.

To implement indexing, I used a HashMap where the key is PropertyIndex and the value is a list of documents that share the same index. During startup, an initial indexing process takes place, and each new document added to the system triggers the indexing process for all its properties.

Whenever a document or a collection is deleted, the indexing process is triggered again to remove the indexed documents from the index. While B trees are more efficient for indexing than HashMaps, they are more complicated to implement and are not as fast (O(log n) time complexity).

Caching

Caching was also applied in this project, to improve the performance of the system.

I used a simple LRU (Least Recently Used) algorithm to implement the caching system.

LRU **(Least Recently Used)** is a caching algorithm that is commonly used in computer science to manage cache memory efficiently. The idea behind LRU is to keep track of the most recently used items in the cache and to remove the least recently used item when the cache is full and a new item needs to be added.

The LRU algorithm works by maintaining a doubly linked list of cache entries, with the most recently used entry at the head of the list and the least recently used entry at the tail. When a cache miss occurs and a new entry needs to be added to the cache, the algorithm checks to see if there is room in the cache. If the cache is not full, the new entry is added to the head of the list. If the cache is full, the least recently used entry at the tail of the list is removed, and the new entry is added to the head of the list.

Whenever an entry is accessed or added to the cache, it is moved to the head of the list, so that the most recently used entries are always at the front of the list. This ensures that the least recently used entries are always at the back of the list and are the first to be removed when the cache is full.

There is also other options for example the LFU algorithm.

Data Structures

I used a HashMap (ConcurrentHashMap) whenever there is a key-value storage that doesn't require the map to be sorted, for example in the indexing and caching.

A doubly linked list was also used in the LRU Caching System.

Dynamic arrays i.e. ArrayList was used in a lot.

Multithreading & Concurrency

Since the system is a RESTful API built on Spring Boot, it is important to note that it is multithreaded by default,

which means that race conditions may occur. To ensure that the system is thread-safe, I implemented the synchronized keyword to restrict access to critical sections of code where data is edited.

I fully synchronized some methods, such as the method below:

```
public synchronized void clearCollectionIndexing(String dbName, String collectionName) {
    List<PropertyIndex> toBeRemoved = new ArrayList<>();

    for (PropertyIndex propertyIndex : propertyIndexMap.keySet()) {
        if (propertyIndex.getDbName().equals(dbName.toLowerCase()) &&
                propertyIndex.getCollectionName().equals(collectionName.toLowerCase())
        ) {
            toBeRemoved.add(propertyIndex);
        }
    }

    for (PropertyIndex propertyIndex : toBeRemoved) {
        propertyIndexMap.remove(propertyIndex);
    }
}
```

While partially synchronizing others:

```
jsonObject.put("username", username);
jsonObject.put("token", token);
jsonArray.put(jsonObject);

try {

    FileWriter fileWriter = new FileWriter(usersCollection);
    synchronized (lock) {
        fileWriter.write(jsonArray.toString());
    }
    fileWriter.close();

} catch (IOException e) {
    throw new RuntimeException(e);
}
```

I used locks with synchronized blocks to manage concurrent access to shared resources.

In addition, I used concurrent data structures, such as ConcurrentHashMap, instead of normal HashMap to ensure thread safety.

Optimistic Locking

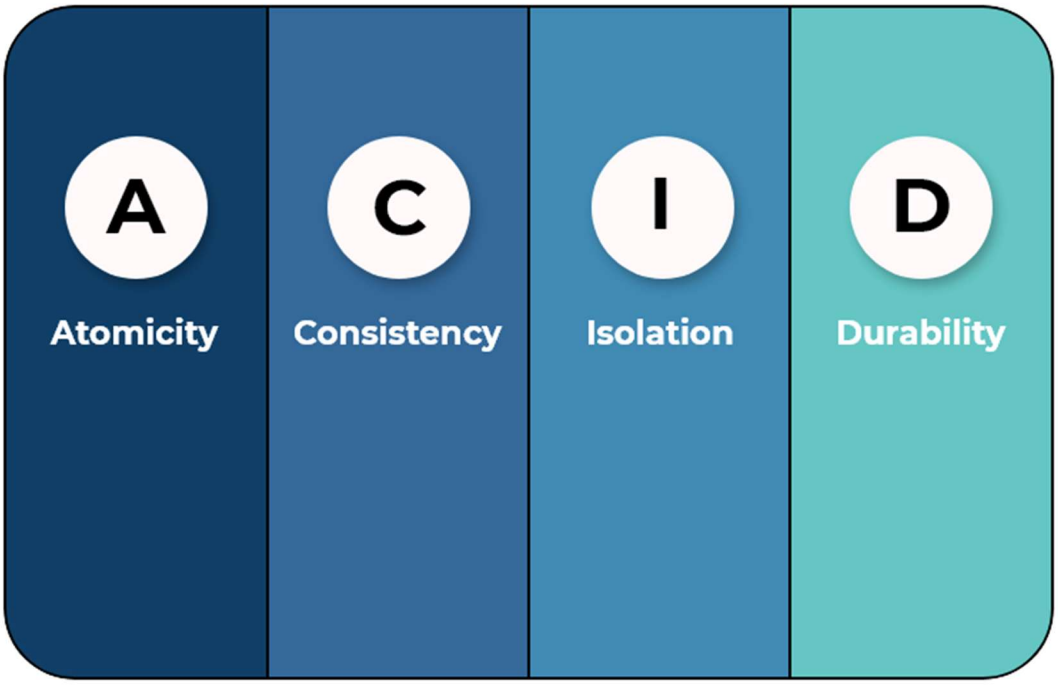
Optimistic locking is an interesting race condition that can occur in this scenario:

"Two workers attempt to update a particular property within the same document at the same time, and neither of them are the affinity for the updated document"

What should happen is that the affinity node will update the JSON property only if the current version of the data matches its own version.

So when doing the update we have to take the old version of the document and compare it with the current version to decide whether to apply the update request or not.

ACID



ACID is an acronym used to describe the fundamental characteristics of a strongly consistent database.

The four components of ACID are:

Atomicity: This ensures that if any part of a transaction fails, the entire operation will roll back. To achieve atomicity, the system uses exception handling and multiple validation levels. For example, if there is a request to update a document inside a collection and that

collection is not found in the file system, the system will return an error to the user and rollback the transaction.

Consistency: This ensures that the database remains structurally sound with every transaction. To guarantee consistency, the system uses an Object Schema to validate the data before storing it in the file system. If there is any problem with the transaction, the system will rollback without touching the file system.

Isolation: This ensures that each transaction is independent of other transactions. To achieve isolation, the system makes all transactions different from each other and avoids any dependency between them. This means that if one transaction fails, it will not affect the other transactions.

Durability: This ensures that all transaction results are permanently preserved. To achieve durability, the system stores the transaction in the file system. In the event of a system crash or application failure, all the transactions will be stored in the file system, and can be recovered once the system is back up and running.

Clean Code

Clean code is code that is easy to understand, maintain, test and easy to extend.

Naming Conventions and Guidelines:

The names of classes, methods, and variables were chosen to be self-explanatory and easy to understand. It helped to ensure that the code was organized and easy to read.

Comments:

In general the code is self-descriptive, however, I tried to provide useful comments to clarify some parts of the code.

Avoiding Returning Null: I tried to avoid returning null by returning empty lists instead or empty strings, and also by throwing exceptions instead of returning null.

Avoiding Magic Numbers: instead of having hard coded numbers I used constants to make the code more readable, for example the files paths instead of writing it again and again I used a constant.

It's also useful if I want to change the path of the files, I just have to change the constant.

Method parameters & Flag Arguments: I tried to minimize the number of parameters in each method, by using helper methods and by encapsulating the parameters in objects, for example, instead of identifiers for the index I used a class called **PropertyIndex** object to encapsulate the identifiers, additionally, I tried to avoid using flag arguments.

Avoiding Long Methods: I tried to keep methods short as possible.

Conditions: Instead of having very big conditions that is hard to understand, I divided that conditions into smaller parts using helper functions.

Fail Fast & Return Early: In every method, I prioritize the validation of input and conditions before executing the main logic.

This helps to catch potential issues early on and prevent unexpected behavior later in the code.

Exception handling: In order to make the code more reliable and efficient, almost every exception is handled in the code.

Coupling: To make the code more maintainable and modular, I attempted to minimize coupling between classes as much as possible. Design patterns were used to decouple certain components.

Cohesion: I ensured that each class only included attributes and methods related to its own functionality, which made the code easier to understand, debug, and maintain. Additionally, I organized the code into cohesive packages, further improving its readability and maintainability.

Principle of Proximity: I followed the principle of proximity by organizing the attributes and functions in a way that related components are placed close to each other.

This helps to reduce the cognitive load on the developer, as they don't have to search through different parts of the code to understand how a specific component works.

Use pronounceable names:

Instead of “ua” we use “usersArray”.

Code Smells

Redundant Comment:

A comment is redundant if it describes something that adequately describes itself.

Dead Function:

Dead functions are functions they never call from any piece of code.

Duplicated code

It is when two or more pieces of code have identical or very similar logic. This can lead to problems with maintainability and can make debugging more difficult.

Data Clumps:

Data clumps occur when two or more pieces of data are frequently used together throughout the codebase.

This can lead to redundant code and can make it harder to maintain the codebase.

It's better to group related data together in a class or structure, rather than scattering it throughout the codebase.

SOLID Principles

I used the SOLID principles to design the system, and I tried to apply them as much as possible.

S: Single Responsibility Principle

"A class should have one and only responsibility over a single part of the functionality provided by the software."

All classes and methods I wrote, all of them apply this principle, so no method nor class break this principle, for example the **tokensManager** class is responsible for managing the tokens, storing them and expiring them, it is not responsible for making HTTP requests inside it, instead it would take advantage **NetworkManager** class to make HTTP requests.

O: Open / Closed Principle: "Entities should be open for extension, but closed for modification"

This means that new functionality can be added without changing the existing code,

L: Liskov substitution principle

"Subtypes must be substitutable for their base types, Derived types must becompletely substitutable for their base types"

I : Interface Segregation Principle

"One fat interface need to be split to many smaller and relevant interfaces"

D : Dependency Inversion Principle

"Depend upon abstraction (interfaces) not upon concrete classes

Abstractions should not depend on the detailed whereas the details should depend on abstractions

High-level modules should not depend on low level modules"

Design Patterns

I used the following design patterns in my code:

Data Access Object Pattern: this is structural pattern, it is used to separate the data access logic from the business logic, which provides better separation of concerns.

I found it really useful in my code, as when I decided to add the indexing and caching features, I just had to add the indexing and caching logic in the DAO class, and the business logic will not be affected at all.

Singleton Pattern: this is a creational pattern, it is used to ensure that only one instance of a class is created, and it is accessible to all other classes, so it's shared among all project components, I used it in the TokensManager class, NetworkManager class, and the DAO class.

Facade Pattern: this is a structural pattern, it was used in the NetworkManager class to provide a simple interface for the developer send http requests so it provides a set of useful methods and the requests is done inside it, hiding the complexity of the system.

Effective Java

Here are some of the items that were used in my project:

Enforce non-instantiability with a private constructor

Of course for Singleton classes, as well as the classes that contains only static methods.

Avoid creating unnecessary objects

All objects were created for a particular reason, and if there is an ability to reuse the same object more than once it, doing this makes the code faster and more efficient.

Always Override hashCode when you override equals.

As we could see in the PropertyIndex class when I wanted to put in the HashMap it’s required to override equals method as well as the HashCode which helps the HashMap utility.

Eliminate obsolete object references.

Always override toString.

Which is used to print the object, so it's useful for debugging and logging.

Minimize the accessibility of classes and members.

In public classes, use accessor methods, not public fields.

Favour composition over inheritance

Favour static member classes over non-static

Prefer lists to arrays

Check parameters for validity

Design method signatures carefully

Return empty arrays or collections, not nulls

Prefer for-each loops to traditional for loops.

Know and use libraries

Avoid float and double if exact answer are required

Beware the performance of string concatenation

Adhere to generally accepted naming conventions

Avoid unnecessary use of checked exceptions

Avoid excessive synchronization

Testing

When it comes to testing I like to test every new feature that I add to the code, I always think of edge cases and handle them.

Specifically, as the database is using REST API, I was doing a lot of requests using IntelliJ and test the responses, and also check for the server logs as discussed earlier, it was really useful to see the server logs and see what's going on in the server side.

And after creating the Banking Web Application, I intended to use every endpoint in the database, so

I resolved some issues that appeared in the web application due to the database, and I fixed them.

So I minimized the number of potential bugs in the database, and I made sure that the database is working as expected.

Demo Web App – BankingSystem

I built a web application that uses the database, which is built on a specific database schema.

The DB included 2 collections which is the customers and transactions collections with a predefined schema for each that will be followed by demo HTML forms.

It's a simple banking system, the users is meant to be bank employees, they can register and they can take tokens to access the system, and they can add new customers, and add new transactions for the customers, (adding or deleting money from the customer account balance),

Updating customer’s information, remove a customer from the system, showing customers transactions, and so on.

The web app contains an admins system that allows the admin to expire tokens for users and to take a look at all active authenticated users in the system.