

ATYPON

Capstone Project

Done By:

Hamza Hassan

Decentralized Cluster-Based NoSQL DB System

ABSTRACT

This project presents a Java-based Decentralized Cluster-Based NoSQL DB System, where users interact with nodes in a decentralized NoSQL database cluster. The system handling JSON-based document databases, supporting DB and document operations, and addressing load balancing, data consistency, security, and efficient node communication. Key challenges like document-to-node affinity, optimistic locking, and custom indexing are resolved.

Table of Contents

Introduction.....	3
System Design	5
Database Implementation	7
?How Can use the database queries	11
Database queries.....	12
queries Collection.....	12
Document queries.....	13
Communication protocols between nodes	15
Data Consistency issues.....	18
load balancing.....	20
:Document-to-node affinity load balanced.....	20
:bootstrap node load balanced.....	21
Data structures	22
Cache	23
Indexing	24
Multithreading and locks	27
Security issues.....	28
Code testing	29
Clean code.....	30
Effective Java.....	33
SOLD principles	34
Design patterns	36
DevOps Practices	37
DOCKER	38
Maven.....	42
Demo.....	43

Introduction

Our project will talk about Decentralized Cluster-Based NoSQL DB System

we can notes in the title we have multiple terminology we must talk about it and we can see the objective of this project its to build the database system not using a download famous database system and using it

but build the system from scratch using pure java

now let's talk about the second term of the title "NoSQL"

NoSQL mean “not only SQL” we already know about SQL that used for managing and manipulating relational databases. However, NoSQL refers to a new type of database that stores data in a non-relational format like document, graph or key-value.

after getting a quick look about NoSQL let's talk about the third term Cluster-Based

A Cluster-Based, is a configuration in which multiple individual Node are connected and work together as a single unit. The purpose of clustering is to enhance availability, reliability, and performance of applications and Nodes.

What does decentralization mean? And what are its benefits?

Decentralized means the absence of a central authority or control, with decision-making and operations distributed across a network or system, reducing single points of failure.

In this project the database is a document-based database that uses JSON objects to store documents
the database will contain multiple of collections and each collection will contain multiple of documents
each document within a DB has a JSON schema that belongs to the DB schema

A schema is serves as a template outlining the organization and format of the data we plan to store within the database

System Design

This system consists of 3 main and important components:

1. Users
2. Bootstrap Node:

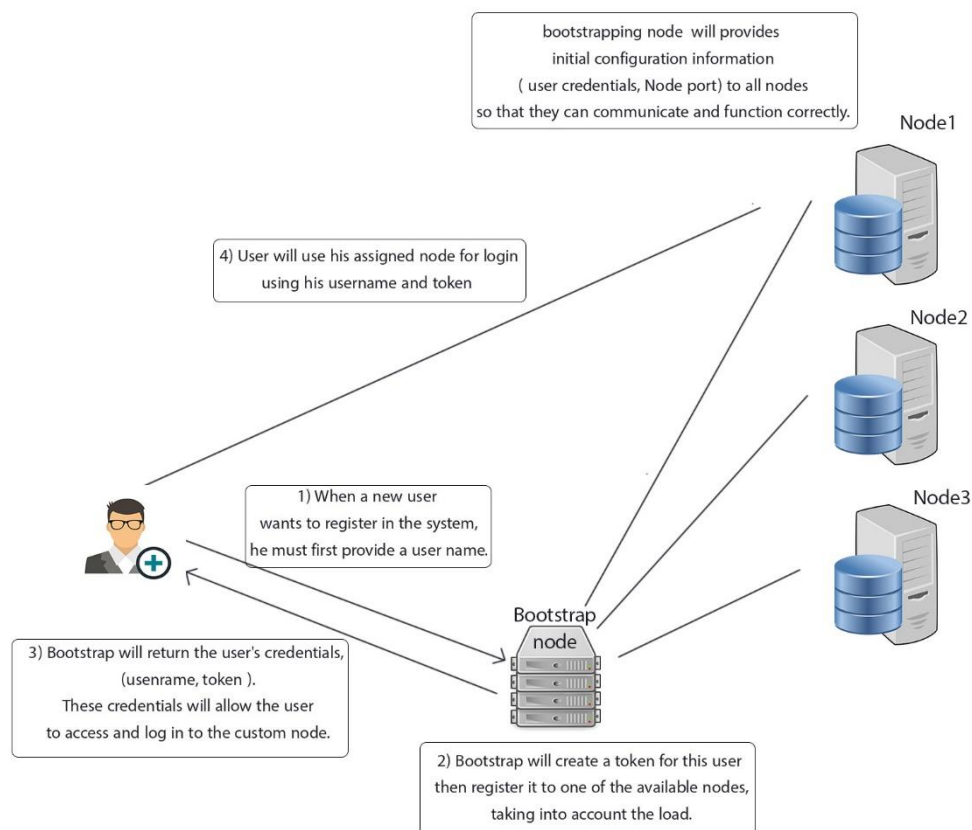
The bootstrapping node has two main jobs:

- a. starting up the cluster and initiating all nodes
- b. signing up new users, and give them unique tokens, saving their information in the database, also map existing users (each user to specific node) in a load-balanced manner.

3. Decentralized Cluster Nodes (VMs):

As the system is designed to be decentralized, every node operates independently, yet ultimately, all nodes are duplicates of each other. Each node has identical code, and each node holds the same data as the rest of the nodes. This system will consist of three nodes each running in a separate Docker container.

The picture below explains the process better.



In this system, write queries are tied to a specific node, which we call "node affinity." This means that when you want to write data to a particular document, you can only do so in the node where it has an affinity or connection.

If write query goes to a node where it no affinity, that node will redirect the query to the right node that's has affinity for that document.

Once a write operation is completed, the node with affinity must inform the other nodes about the change. During this notification process, if any read requests occur on other nodes, they will still access the old version of the data. However, whenever a node gets updated, subsequent read requests must return the most up-to-date version of the data

Database Implementation

As we mentioned previously, to build our own document-based database that uses JSON objects to store documents

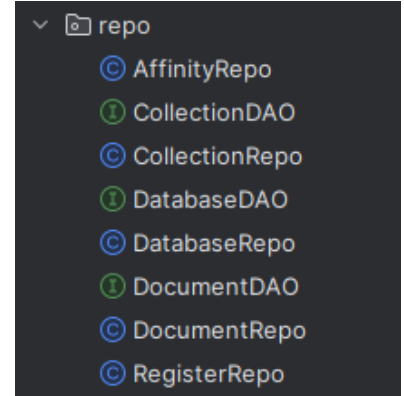
We will need to deal with 4 main elements.

1. Database
2. Collections: The data base will contain several collections that will represent the schema
3. Documents: The collection contains many documents
4. Affinity Node : We'll talk about it later

Overview :

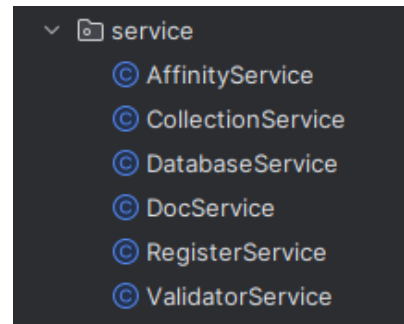
Repository (Repo) package:

contain multiple of Repo class in each of them was responsible for dealing with Json's external files in this a document-based database system to do important operation like creation, deletion, update, etc..(delete or writing data on files, and reading from the files)



Service package:

contain multiple of Server class The purpose of them to act as an bridge between the repository and the controller. Its primary responsibilities included:



Data Validation:

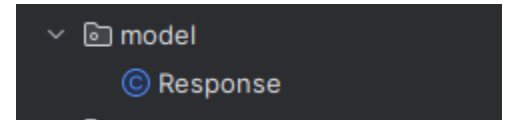
Ensuring the data received from the controller was valid and met the required criteria before forwarding it to the repository for processing.

Data Handle Broadcasting:

Broadcasting the data to the appropriate destination.

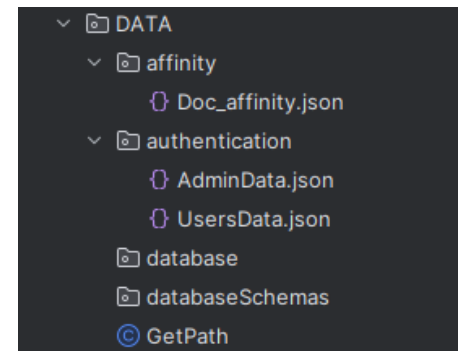
Model:

The "Response" class model defines a data structure for encapsulating HTTP response status and message for communication within a Java application.



Data package:

I prepared this package to be the path in which the node will store its data like database ,database schema , and Document-to-node affinity data , and authentication data

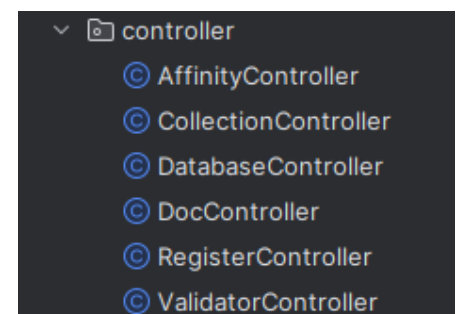


Broadcast package:

This package includes the class that manages data sharing among all nodes. Its role is to ensure that all nodes stay updated about system events so that all nodes have the same data.

Controller package:

As I mentioned earlier, I use the Spring Framework, and controllers are essential to define the API for handling database queries and managing HTTP requests in the system.
To achieve this, I utilize REST Spring in these classes.

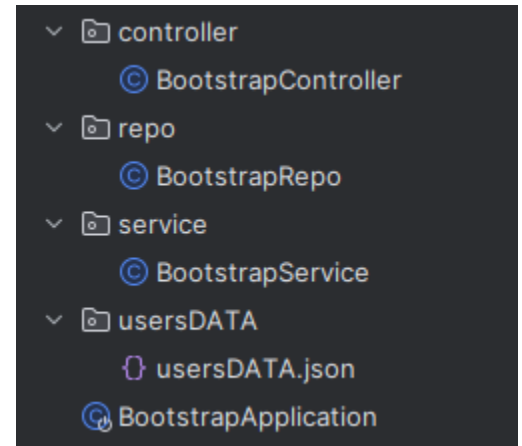


Bootstrap Implementation

The structure will be similar to what we've seen in the Database Implementation, and we'll keep using the Spring Framework. We'll follow the same division approach

Repository (Repo):

This component will handle tasks such as user registration, generating unique tokens for users, storing data in the Bootstrap's database (inside file userData.json), user deletion, and various other functions.



Service:

The service layer will manage communication with nodes, registering users with specific nodes, and ensuring load balancing across the system.

Controller:

The controller will serve as a RESTful API, taking care of HTTP requests within the system.

After we know how build our system in general and knowing that the API for this system will be a RESTful API

What are the queries available to us in this system and how to use them?

How Can use the database queries?

Using RESTful API and inside Controller class will find the way deal with queries using URL mapping :

Example to create database:

POST database/createDB/{ DBName }

```
#creat database DONE
# curl -H "username: root" -H "token: root123" http://localhost:8081/api/createDB/firstDB
▶ POST http://localhost:4001/database/createDB/RegistrationSystem
  username: root
  token: root123
  content-type: application/json
```

As shown in the picture, the database name will be included as a path variable in the URL.

Additionally, it's important to add the headers, specifically the username and token. These headers are important for security, ensuring that only authorized users can send data to prevent any potential data issues or damage. So, before executing any database queries, we need to confirm that the user has the proper authorization to perform these operations.

Now after we know how to use it and that we must put username and token to headers, Let's quickly review the queries that we can use to build our database

Database queries

create database:

POST database/createDB/{ DBName }

delete database:

DELETE database /deleteDB/{ DBName }

show all existing database:

GET database /showAllDatabases

Collection queries

To create a collection, the first step is to define its schema and then insert it using a RequestBody.

the schema is important because it defines the structure of the documents that will be added to the collection later.

create collection:

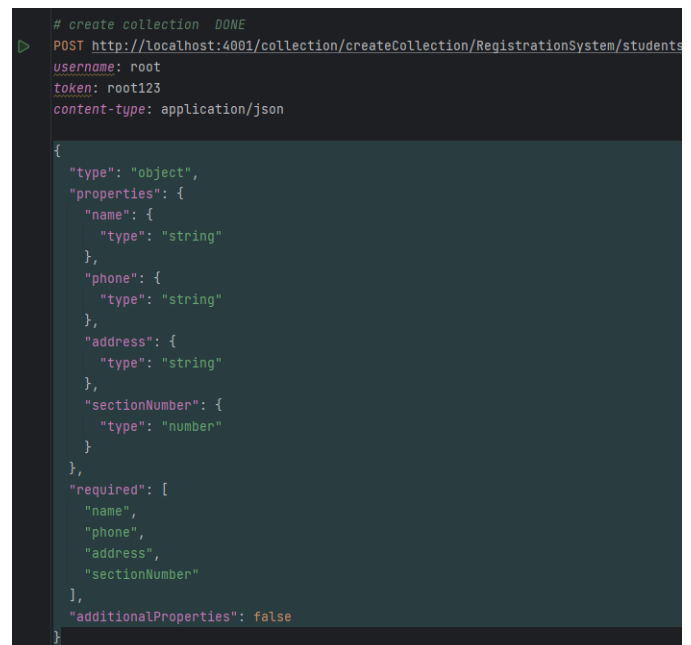
POST collection/createCollection/{DBName}/{ collectionName }

Delete collection:

DELETE collection/deleteCollection/{ DBName }/{collectionName}

show all existing database:

GET collection /showAllCollections



```
# create collection DONE
POST http://localhost:4001/collection/createCollection/RegistrationSystem/students
username: root
token: root123
content-type: application/json

{
  "type": "object",
  "properties": {
    "name": {
      "type": "string"
    },
    "phone": {
      "type": "string"
    },
    "address": {
      "type": "string"
    },
    "sectionNumber": {
      "type": "number"
    }
  },
  "required": [
    "name",
    "phone",
    "address",
    "sectionNumber"
  ],
  "additionalProperties": false
}
```

Document queries

add document:

POST document/addDoc/{DBName}/{collectionName}

Delete document:

DELETE document/deleteDoc/{DBName}/{collectionName}/{docID}

update document property value:

POST

document/updateDoc/{DBName}/{collectionName}/{docID}/{propertyName}/{newValue}

Get specific document:

GET document/getDoc /{DBName}/{collectionName}/{docID}

Reading Specific Properties:

GET

document/readingSpecificProperties/{DBName}/{collectionName}/{docID}/{propertyName}

Document filtered based on the property value:

GET

document/filterDoc/{DBName}/{collectionName}/{propertyName}/{propertyValue}

show all existing document:

GET document / getAllDocuments / {DBName} / {collectionName}

Some of RESTful API URL important for system to work correctly

RegisterController

it's important when communicating with bootstrap

Register specific user to specific node:

POST register / addNewUserToNode /{username}/{token}

remove specific user from specific node:

DELETE register / removeUserFromNode /{username}/{token}

Add new admin to the system:

POST register /addNewAdmin/{newAdminName}/{newAdminToken}

ValidatorController

important to check validation of the users and admin when using the system

Make sure that he is an authorized Admin:

GET validation /isAdmin/{username}/{token}

Make sure that he is an authorized User:

GET validation /isValidUser/{username}/{token}

What about affinity controller? why would we need it?

Before talking about this topic, we must talk about Communication protocols between nodes and Data Consistency issues in the DB.

Communication protocols between nodes

The communication protocols between cluster nodes will be done by Spring boot framework , Because each node will run on its own Docker container, therefore it will have its own port number that we will use to communicate with this node, so that communication can occur between the nodes, sharing documents between them, and redirecting the document to its affinity node.

We will use this process extensively when discussing load balancing, which I will explain in detail in the load balancing section.

It is a good decision now to talk about node affinity

Node affinity is a way to ensure that specific nodes are responsible for performing write operations to a specific document, Not all nodes have the authority to do this, mean that there is one node has responsible to write or update or delete to it . If the user sends an update request to a node that doesn't have affinity for the document they want to update, that node will pass the request to the node that has affinity to this document.

After you make a change to a document, the affinity node will broadcast to the other nodes about the change. While this is happening, if you try to read the same document from a different node, you might get the old version. But once a server gets updated, it will give you the latest version when you read the document.

It's really important that we spread out the work of handling documents across all the nodes in a fair way so that no node gets overloaded, so we need a way to make the nodes can communication with each other.

To enable nodes to communicate with each other and determine which node has affinity or will take the next turn to become an affinity node, we will explore this further in the load balancing section.

AffinityRepo: to read or write the data about Document-to-node affinity to file(database)

AfiinityService: it will provide important services.

AffinityController: will provide RESTful API to can node talk with each other.

will use this URL mapping to save the Document-to-node affinity data in database :

POST affinity / addDocToNodeAffinity / {id} / {nodeName}

To marks the current node as the affinity for the next document that will be added:

GET affinity / setAffinityNode

To removes the affinity from current node:

GET affinity / clearAffinityNode

To checks whether the current node is the affinity or not:

GET affinity / isAffinity

This URL using in bootstrap stage to setup the node and give it the name :

POST affinity / setCurrentNode / {nodeName}

use to get the named of current node :

GET affinity / getCurrentNode

To get all document-to-node data that saved in database:

GET affinity / getAllDocToNodeAffinity

Data Consistency issues

To ensure consistency between all parts of the cluster, when any of the nodes makes any modification or addition, we must publish this change to all nodes to ensure consistency between them and that they all have exactly the same data.

For this reason, I created a Broadcast Class, which will be responsible for sending any modification, addition, or deletion and broadcasting it to all nodes so that they update their data to the latest version.

```
8 usages
public void broadcast(String endPoint, String method, String requestBody) {
    for (String node : nodes) {
        String url = "http://" + node + ":4001/" + endPoint;
        HttpHeaders headers = new HttpHeaders();
        headers.set("username", "root");
        headers.set("token", "root123");
        headers.set("broadcast", "true");

        HttpMethod httpMethod = switch (method) {
            case "GET" -> HttpMethod.GET;
            case "DELETE" -> HttpMethod.DELETE;
            case "POST" -> HttpMethod.POST;
            default -> HttpMethod.GET;
        };

        RestTemplate restTemplate = new RestTemplate();
        HttpEntity<String> requestEntity = new HttpEntity<>(requestBody, headers);
        restTemplate.exchange(url, httpMethod, requestEntity, Void.class);
    }
}
```

it will rotate through the nodes one by one to send the given data to

the end point

You will need to include an endpoint, which will represent the action that all nodes will perform

In addition to the inclusion of username and token to header, as we mentioned previously, Also must add addition new header called broadcast , it is false by default, must set it to true when performing the broadcast process

So that all nodes knows that this request has been published and there is no need to do the broadcasting process again.

You must choose the HttpMethod to be used.

In addition to sending a request body to be added if it exists

How can Document-to-node affinity to be load balanced? How do we determine which is the best node to be the affinity node to specific document?

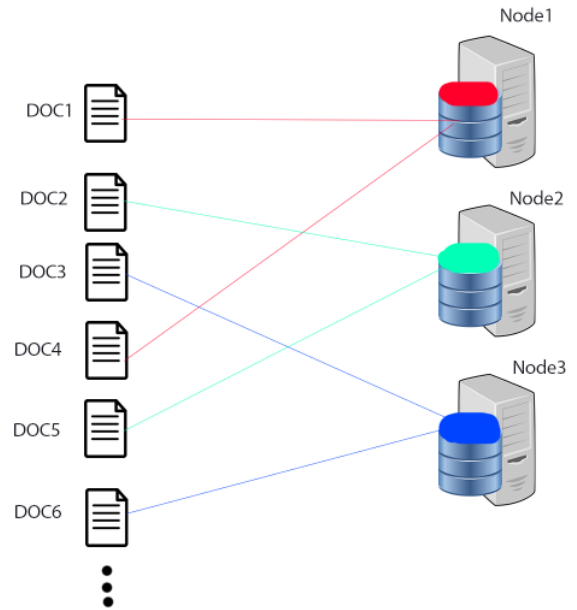
To get the answers, follow Section load balancing.

load balancing

Document-to-node affinity load balanced:

To achieve load balance between nodes to choose affinity node to specific document , I decided to apply the cyclic approach between nodes

****Remember all documents will be present in all nodes, But the picture is only to illustrate which nodes will have affinity to modify the document ****



This will distribute the documents on the nodes almost evenly and keep the load balanced. It is a simple and effective solution.

By utilizing the `setAffinityNode`, `clearAffinityNode`, and `isAffinity` URLs mapping, nodes can establish communication and emulate this cyclic approach.

After the node with the affinity adds a new document, it will clear its affinity and pass it to the next node in the cycle using method called `cycleAffinityNode`

```

1 usage
public void cycleAffinityNode() {
    clearAffinityNode();
    Map<String, String> nodeMapping = new HashMap<>();
    nodeMapping.put("node1", "node2");
    nodeMapping.put("node2", "node3");
    nodeMapping.put("node3", "node1");

    String currentNode = getCurrentNode();
    String nextNode = nodeMapping.get(currentNode);

    String url = "http://" + nextNode + ":8081/affinity/setAffinityNode";
    HttpEntity<String> requestEntity = new HttpEntity<>("body: ");
    RestTemplate restTemplate = new RestTemplate();
    restTemplate.exchange(url, HttpMethod.GET, requestEntity, Void.class);
}

```

bootstrap node load balanced:

We will also need load balancing in the bootstrap node, as it is responsible for connecting users to node

I also used the cyclic approach here, It is simple and effective and will ensure that users are evenly distributed among the nodes

```

1 usage
public String getNodeByLoadBalance(){
    int index = nodeIndex++ % nodes.size();
    return nodes.get(index);
}

```

Data structures

ConcurrentHashMap

I used ConcurrentHashMap because it's great for managing shared data in multi-threaded programs.

It allows multiple threads to work with a map at the same time without causing problems. It's designed smartly, using locks at a smaller level, which makes it faster and ensures thread safety. It's flexible, can handle a lot of data, and doesn't block reading, making it a solid choice for handling data in Java programs when multiple things are happening at once.

LinkedListHashMap

Where I used it inside the cache and method called `removeOldestEntries` in the Indexing class, where its power is shown by:

1. Maintains entries in the order in which they were accessed (access order).
2. Provides an efficient way to remove the oldest entry when the maximum size is reached.

3. ensuring that the most recent entries are at the end. It ensures a predictable recurrence order, which simplifies the implementation of the LRU eviction policy for the index map and cache.

HashMap

Array List

Cache

In cache uses a LinkedHashMap to store key-value pairs. It has a maximum capacity of 10,000 items and automatically removes the least recently used item when the limit is reached to make space for new entries. To ensure data consistency during concurrent access, it employs a ReadWriteLock, allowing multiple threads to read data and ensuring that only one thread can modify the cache at a time. This makes it safe for use in multi-threaded applications.

```
cache = new LinkedHashMap<>(capacity, 0.75f, true):
```

In this line of code, specified initial capacity (capacity) and a load factor of 0.75 (0.75f). The true parameter at the end enables access order, meaning the LinkedHashMap will maintain the order of entries based on their access pattern. When the size of the cache exceeds 75% of its capacity, it will start removing the least recently accessed entries due to the removeEldestEntry method override.

Indexing

Indexing in NoSQL databases is a technique used to optimize data retrieval by creating data structures that allow for efficient searching and querying.

Unlike traditional relational databases, NoSQL databases often use various indexing methods, such as B-trees, hash indexes, or full-text indexes, to quickly locate and access data based on different criteria like property.

This helps improve query performance, What we seek in this project is to create indexes on one single json property.

After reading about data structures that are used for indexing, I found many methods, I choose to do it hash indexes. I know it's not the best option for indexing, But it is good and serves what we need in this project, and it is faster and less complicated than other methods

In hashing, we use a "key" and a "value." Our main goal is to quickly find data within a JSON file based on a specific property.

To do this, we first need to know which database contains the JSON file. Then, we figure out which collection the file is in within that database. Finally, we use the "key" (which matches the JSON property we're interested in) to locate the exact piece of information we want.

It's like following a map: we start with the database, then the collection, and finally use the key to pinpoint our data.

```
Map<Map<String, Map<String, Map<String, String>>>, JSONArray> indexingMap ;
```

Diagram labels: DBName (green), collection (red), collectionName (red), propertyName (yellow), propertyValue (yellow), DB (green). A bracket under the first Map is labeled DB. A bracket under the inner Map is labeled collection. A bracket under the innermost Map is labeled propertyName. A bracket under the String value is labeled propertyValue.

But to make it easier and less complicated, I built a model called indexingModel to put all this information inside

```
Map<IndexingModel, JSONArray> indexingMap
```

Because we are working in a multi-threaded system, I decided to use ConcurrentHashMap to ensure work safety.

The time complexity to get the data from the indexing will be $O(1)$

To maintain the size of the index and prevent it from exceeding the available memory, create the removeOldestEntries method is responsible for maintaining the size of the indexingMap within a specified limit (MAX_INDEX_SIZE) by evicting the least recently used entries. It does this by temporarily transferring the existing entries to a LinkedHashMap called lruMap, which automatically removes the oldest entries when the map size exceeds the limit, ensuring that the indexingMap stays within its defined boundaries. This mechanism helps manage memory usage and ensures the

efficient operation of the indexing system.

```
1 usage
private void removeOldestEntries() {
    LinkedHashMap<IndexingModel, JSONArray> lruMap = new LinkedHashMap<>(MAX_INDEX_SIZE, loadFactor: 0.75f, accessOrder: true)
    2 usages
    @Override
    protected boolean removeEldestEntry(Map.Entry<IndexingModel, JSONArray> eldest) {
        // Remove the eldest entry when the map size exceeds the maximum
        return size() > MAX_INDEX_SIZE;
    }
};

// Copy the existing indexing map into the LinkedHashMap to maintain access order
lruMap.putAll(indexingMap);

// Update the indexing map with the LinkedHashMap (it will contain the most recently accessed entries)
indexingMap = lruMap;
}
```

Multithreading and locks

Given that the system operates as a RESTful API within a Spring Boot framework, it inherently supports multithreading.

To ensure that multiple threads can work safely together, I've used the “synchronized” keyword at key points in the code to coordinate their actions.

In my code, some methods are completely synchronized, while others are partially synchronized. I also use synchronized blocks with locks to control how multiple threads access shared resources.

I have used concurrent data structures like `ConcurrentHashMap` to enhance thread safety, also concurrent locks like `ReadWriteLock`

An interesting race condition scenario when Two nodes attempt to update a particular property within the same document at the same time, and neither of them are the affinity node 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 data from the node want to update and compare it with the current data inside the node has affinity of this document to decide whether to apply the update request or not.

Security issues

In this section I will discuss the security challenges that I've faced and how I solve them.

The Challenges

- Someone is trying to send a request to add or delete documents using the URL and he is not authorized to do so.
- Someone tries to connect without getting authenticated by the bootstrap node.
- Someone attempts to establish a connection with a node they are not authorized to connect to

To solve all these problems and ensure the confidentiality of information, I added the username and token as a requirement header.

Which must be submitted by any user who will use any end point for the system

There I will check with this user whether it is registered with the nodes it is trying to access or not

some actions, like setting up all nodes, are restricted to the admin, and they also require the admin's username and token during the bootstrap process.

Code testing

When it comes to testing, I make sure to check every new thing I add to the code. I always consider different cases and handle them. Specifically, for the database that uses a REST API, I send lots of requests using postman and IntelliJ to see how it responds. Moreover, I've added print statements and a logger mechanism to the code to assist with debugging and identifying potential errors.

After creating the School Registration System web app, I made sure to use every part of the database. This helped me find and fix some problems that were causing issues in the web app. So, I reduced the chances of having bugs in the database and made sure it works as it should.

Clean code

In the world of software development, code is not just lines of instructions; it's a communication tool that connects developers, machines, and the future. Clean code is the art of writing software that is not only functional but also easy to understand, maintain, and extend. It's the craft of transforming complex logic into a readable, elegant, and efficient form.

Meaningful Variable and Method Names:

- **Avoid Mental Mapping:**
the names in the code are clear and others can easily understand what they mean.
- **Avoid Disinformation:**
Using names that obscure the meaning of the code was totally avoided.
- The name of the implemented variables, methods, or classes in the system tell us why they exist and what they do.

- We made sure to avoid using duplicate names for methods or variables by adding a number at the end if needed, like A1 or A2.

Clean Formatting:

Make the code look organized by using the same spacing style. It helps others read the code.

Keep Methods and Classes Small:

Breaking down large methods and classes into smaller, focused units makes the code easier to understand and maintain. The Single Responsibility Principle (SRP) suggests that a method or class should have only one reason to change.

Low Coupling:

Low coupling means that modules are independent and changes in one module don't impact others. This isolation reduces the chances of errors spreading and makes debugging easier.

High Cohesion:

High cohesion means each module has a clear purpose, making the code more understandable for current and future developers. It also

simplifies the system by breaking it into smaller making it easier to design and work with.

Error Handling:

- When an error occurs, ensure the exception message provides users with enough details.
- Avoid returning null values.
- Do not pass null as an argument to methods; it's better to handle null values gracefully within the method itself.

Effective Java

Efficient Java by Joshua Bloch is an essential book for Java developers, providing concise, practical advice on writing efficient, readable, and maintainable Java code.

I used some of the tips from this book when I wrote the code

Use Descriptive Variable Names

Minimize the Scope of Local Variables

Avoid Synchronization Where Possible

Avoid creating unnecessary objects

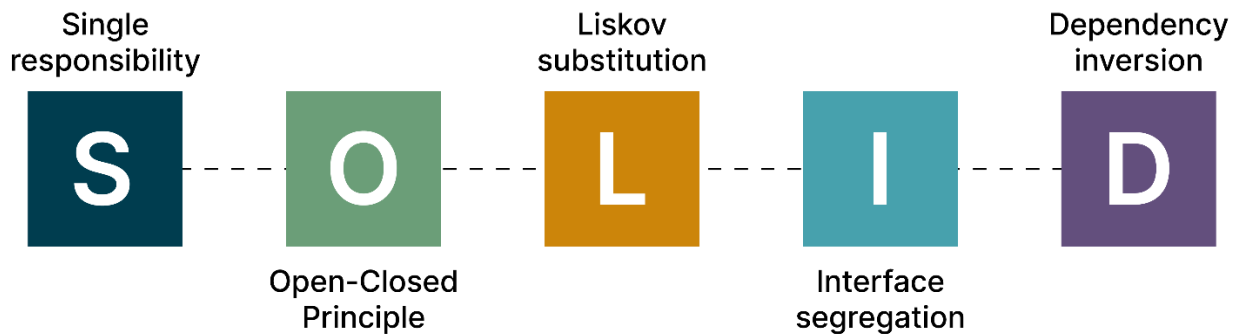
Document Your Exceptions

Return Empty Arrays or Collections, Not null

Don't Ignore Exceptions

SOLID principles

The SOLID principles provide guidance on organizing functions and data structures into classes, as well as defining the relationships between these classes. These principles offer a framework for creating software that is easier to maintain and extend as the project evolves. Additionally, they help in preventing code issues, supporting code refactoring.



Single Responsibility Principle (SRP)

Mostly, all implemented Classes have one reason to change, Meaning that a class has only one job to do.

for example, DocumentRepo class Its only function is to deal with documents by writing, modifying, or deleting them from a file on the device

Open-Closed Principle (OCP)

Entities should be open for extension but closed for modification. In other words, you should be able to add new functionality to a system without altering its existing code.

Liskov Substitution Principle (LSP)

states that you should be able to use a child class wherever you use the parent class, and the program should still produce the correct results. In my code, you will observe the application of this principle when implementing the DAO design pattern.

Interface Segregation Principle

classes should not be forced to deal with things they don't need. It suggests using small and specific interfaces instead of big, one-size-fits-all interfaces.

In my code, you will observe the application of this principle when implementing the DAO interface.

Dependency Inversion Principle (DIP)

that high-level modules should not depend on low-level modules, both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions.

Design patterns

Singleton pattern

Singleton pattern is a design pattern that restricts the instantiation of a class to a single instance and provides a global access point to that instance across the application.

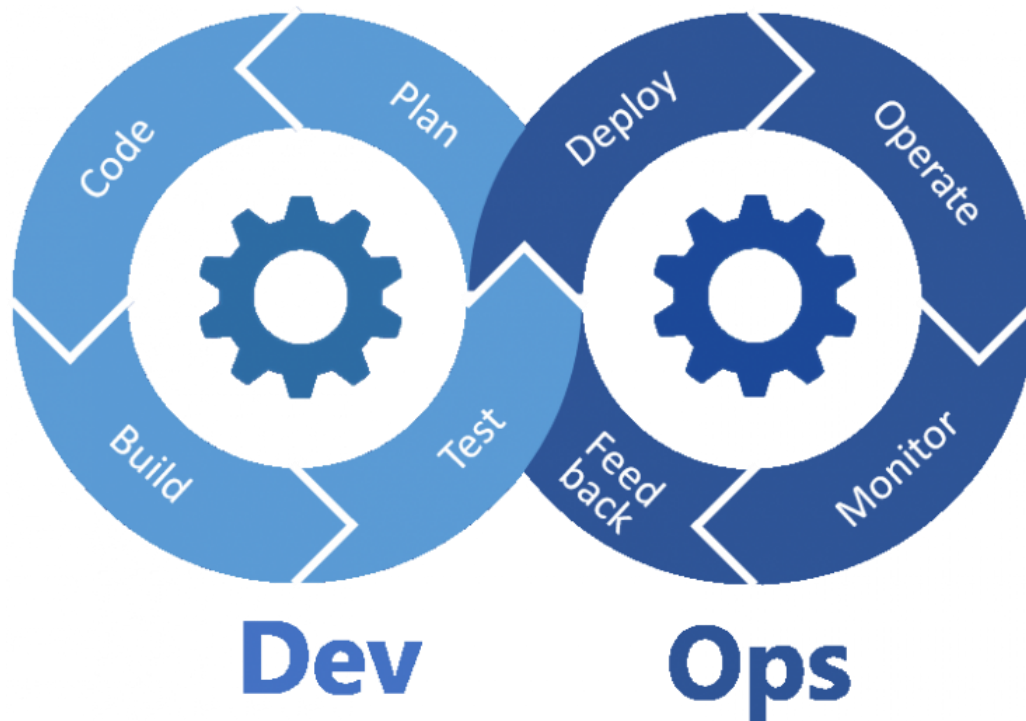
I have used this design several times in places where there should only be one instance of the class, such as AffinityService and IndexingRepo.

Data Access Object (DAO)

Data Access Object pattern is a design pattern that separates the data access logic from the rest of the application. It provides a centralized interface for accessing data stored in various data sources such as databases, files, or APIs, allowing for easier maintenance and testing of data-related operations.

I used this design a lot in my work, so I needed to create 4 packages to serve this design (repo(DAO) , Service , model, controller (APIs))

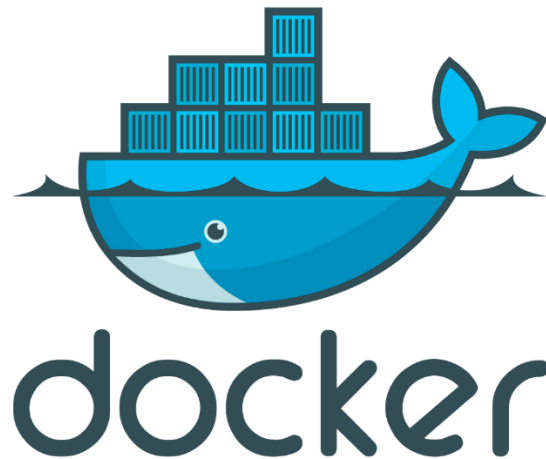
DevOps Practices



DevOps is a set of practices that aim to improve collaboration between software development (Dev) and IT operations (Ops) teams. Key DevOps practices include continuous integration, continuous delivery.

These practices enable faster and more reliable software development and deployment.

DOCKER



Docker is a handy tool that makes it easy to create, share, and run applications. It helps you keep your software separate from the technical stuff underneath, making it faster to deliver software.

By using Docker, you can quickly package, test, and launch your code, saving a lot of time from writing it to running it live.

In my case, I utilized Docker to execute my code in a real-world environment. To achieve this, I created a Docker file for each component of my system, along with a Docker Compose file that outlines the services.

All nodes share the same Docker file and Java code . it should listen on port 4001 when it runs, also Copy the entire project into the container to ensure that file paths are accessed accurately.

```
1 FROM openjdk:17
2
3 WORKDIR /app
4
5 COPY . .
6
7 EXPOSE 4001
8
9 CMD ["java", "-jar", "target/NodeVM-0.0.1-SNAPSHOT.jar"]
```

bootstrap node will listen on port 4000 when it run

```
1 FROM openjdk:17
2
3 WORKDIR /app
4
5 COPY . .
6
7 EXPOSE 4000
8
9 CMD ["java", "-jar", "target/Bootstrap-0.0.1-SNAPSHOT.jar"]
```

to run the whole system, we will use
docker-compose

Docker Compose configuration defines four services:

services :

- node1, node2, and node3:

These services are built from the same Docker image located in the ./NodeVM directory.

Each one is given a unique name and maps port 4001 from the container to a different port on the host machine (4001, 4002, and 4003)

- bootstrap:

This service is built from a Docker image located in the ./Bootstrap directory. It's named "bootstrapNode" and maps port 4000 from the container to port 4000 on the host machine.

It also depends on node1, node2, and node3, it will only start after those services are up and running.

```
docker-compose.yml
H: > atypn > capson_project > finalCode > Capson_project
1  version: '3.9'
2
3
4  services:
5
6    node1:
7      build: ./NodeVM
8      container_name: node1
9      ports:
10       - "4001:4001"
11      networks:
12       - cluster_network
13
14    node2:
15      build: ./NodeVM
16      container_name: node2
17      ports:
18       - "4002:4001"
19      networks:
20       - cluster_network
21
22    node3:
23      build: ./NodeVM
24      container_name: node3
25      ports:
26       - "4003:4001"
27      networks:
28       - cluster_network
29
30    bootstrap:
31      build: ./Bootstrap
32      container_name: bootstrapNode
33      ports:
34       - "4000:4000"
35      depends_on:
36       - node1
37       - node2
38       - node3
39      networks:
40       - cluster_network
41
42  networks:
43    cluster_network:
```


Network:

Docker Compose automatically generates a virtual network to connect the services you define. However, I chose to create a network named `cluster_network` manually. All the services will be linked to this network, allowing containers within the same network to communicate with each other using their respective service names.

Maven



Maven is a useful tool for building Java projects. It works for all Java projects, making the building process simpler. It does this by fetching the necessary tools and libraries from the internet automatically, so you don't have to worry about finding and installing them yourself

Building my Spring Boot project was a breeze thanks to Maven. It effortlessly handled dependencies, streamlined the build process, and kept my project organized. Maven was like my project's secret weapon, making everything smoother and more efficient.

Demo

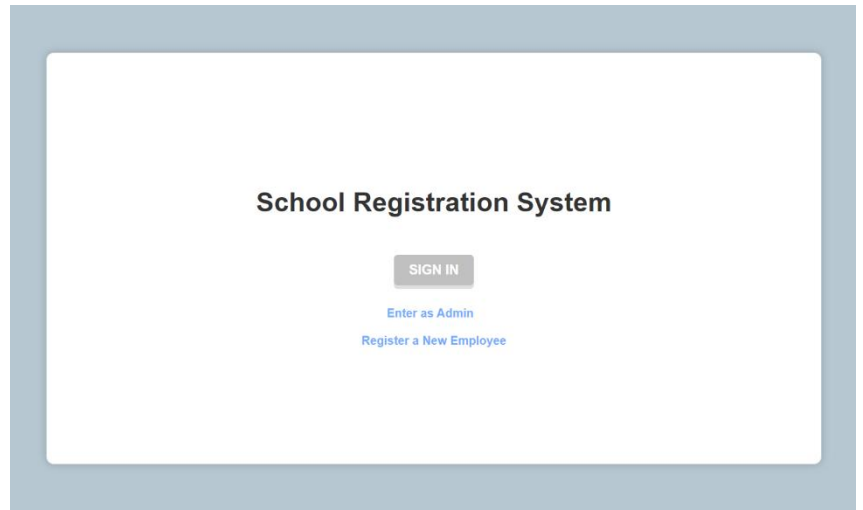
Now, I'll discuss the application I created to demonstrate and prove the efficiency of my system

The application is intended to function as a school registration system. Its main purpose is to make the job of registration employees easier. With this app, they can easily enter, edit, or delete student and instructor information. Plus, they can access this data whenever they need it.

Before we can check out the application, we need to learn how to use it.

1. you must have Docker installed on your machine and running then use this command : “ **docker-compose up** ” to run docker-compose to setup the database system
2. must run the school registration system using any java IDE
3. go to any browser you use and go to this URL “ localhost:8080 ”

4. will get this page
you must click on
“Register a New
Employee” the add
username



5. This process will register new employee to the system and give him valid data can used to sign in to the system
also you can note to any node will assigned this employee, this
will happen in load balance manner

Registration Information

Username: hamza

Token: 89a62e5b-ec74-45f5-94fe-aa5e8b51476f

- copy the token

Home Page

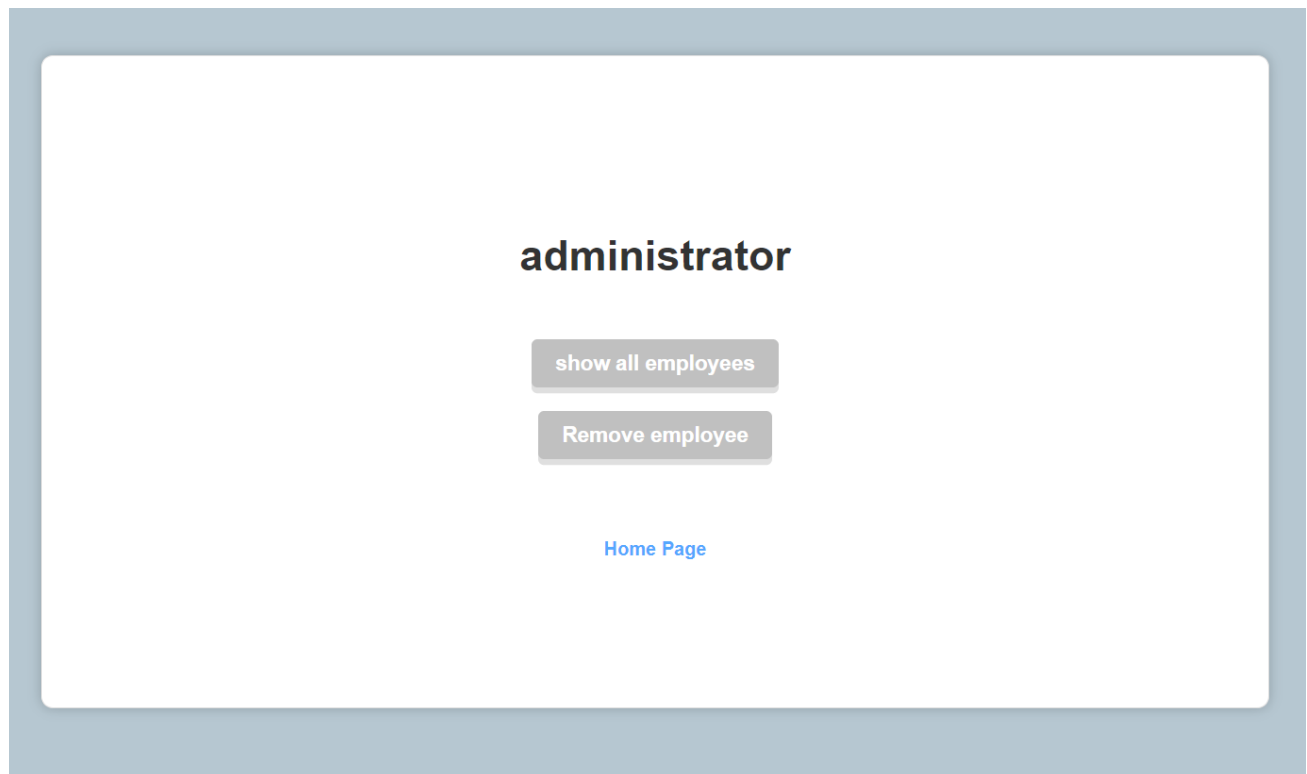
this employee assigned to node1

6. You can enter as admin also using this data :

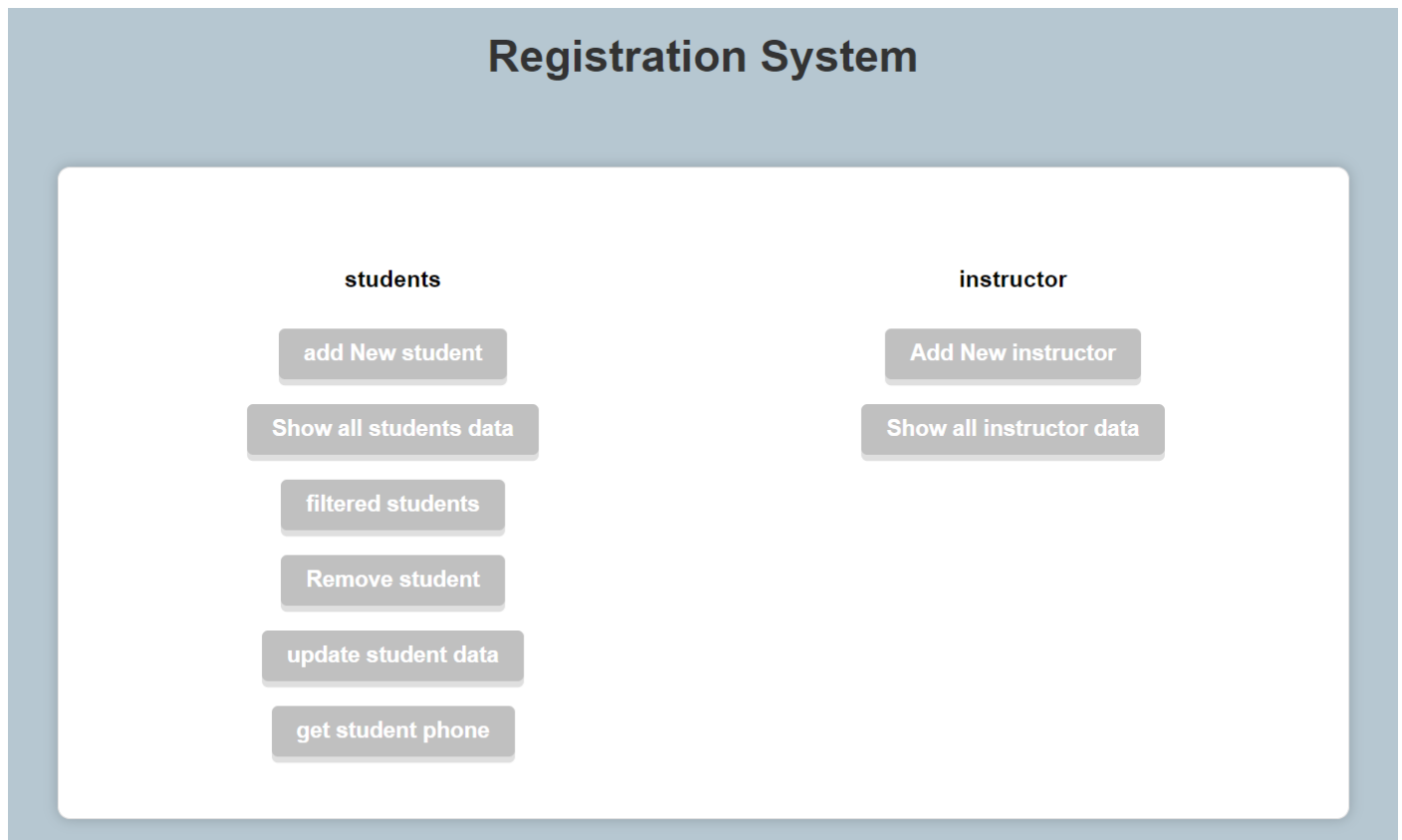
username : root

token : root123

There you can show all employees registered for the system
and can remove employees from the system



7. After sing in successfully you can finally get the system



the database comprises two collections:
one for students and another for instructors, each of which
adheres to a predefined schema. These schemas will be utilized
in the creation of demonstration HTML forms.

Add new student (add document)

Register student

Student Name

student Phone Number

student Address

section number

add student

[Back](#)

Show all student (get all document)

All Students Data

ID	Name	Phone	Address	sectionNumber
c61d1624-4a13-476e-aacc-fae5af48c668	hamza	0780903483	amman	2
d77c3bfe-a6e5-45be-aafa-bc963e1eef58	hamza	0489649	amman	3
23a38935-6f95-4c28-b70e-19fe165e559e	omar	0498412	zarqa	1
c36869e8-94e3-4e30-923a-9e45919f5ac4	hamza121	01329655	05460	3
0f20cc0b-d7fa-41e4-9dda-98a38f3552e4	ha	48	05460	3
674ff823-a40f-468a-b027-e2df73f29ebd	ha	48	05460	3
d9e858b2-beae-441b-b4b3-93d8ca14fa3d	hamza	48	05460	3
6240d574-121f-4b49-b043-2db4385a5016	hamza	07800656	amman	1
317e3241-d8ad-46bd-81ae-137ba4f424f5	hamza	48964944	amman	1

[Back](#)

Update student data (update document)

Update Students Data

ID	Name	Phone	Edit Phone	Address	Edit Address	section Number	Edit section Number
c61d1624-4a13-476e-aacc-fae5af48c668	hamza	0780903483	<input type="button" value="Edit"/>	amman	<input type="button" value="Edit"/>	2	<input type="button" value="Edit"/>
d77c3bfe-a6e5-45be-aafa-bc963e1eef58	hamza	0489649	<input type="button" value="Edit"/>	amman	<input type="button" value="Edit"/>	3	<input type="button" value="Edit"/>
23a38935-6f95-4c28-b70e-19fe165e559e	omar	0498412	<input type="button" value="Edit"/>	zarqa	<input type="button" value="Edit"/>	1	<input type="button" value="Edit"/>
c36869e8-94e3-4e30-923a-9e45919f5ac4	hamza121	01329655	<input type="button" value="Edit"/>	05460	<input type="button" value="Edit"/>	3	<input type="button" value="Edit"/>
0f20cc0b-d7fa-41e4-9dda-98a38f3552e4	ha	48	<input type="button" value="Edit"/>	05460	<input type="button" value="Edit"/>	3	<input type="button" value="Edit"/>

Filtered document :

filtered Students

select method to search or filtered students

[Back](#)

Get student phone (reading specific property)

Student ID:

search

phone : 0780903483

[Home Page](#)

also I've created a special page to display evidence of correctness and load balance

This interface shows the document ID along with the name of Node affinity which is responsible for this document.

load balance data

Document ID	Node name
c61d1624-4a13-476e-aacc-fae5af48c668	node1
d77c3bfe-a6e5-45be-aafa-bc963e1eef58	node2
23a38935-6f95-4c28-b70e-19fe165e559e	node3
c36869e8-94e3-4e30-923a-9e45919f5ac4	node1
0f20cc0b-d7fa-41e4-9dda-98a38f3552e4	node2
674ff823-a40f-468a-b027-e2df73f29ebd	node3
d9e858b2-beae-441b-b4b3-93d8ca14fa3d	node1
6240d574-121f-4b49-b043-2db4385a5016	node2
317e3241-d8ad-46bd-81ae-137ba4f424f5	node1

[Back](#)

As we note and as we explained previously, the load balance works in a circular approach

The end