ATYPON

Java and DevOps Bootcamp (Nov 2023)
Capstone Project
Decentralized Cluster-Based NoSQL DB System

Name: Farouq Tahsin Gfishi

Contents

1. Introduction

This report provides an overview of the design of this database, detailing the technologies and techniques utilized throughout its development. It aims to offer insights into the inner workings of the database. By the end of this report, you will gain a comprehensive understanding of the database's architecture and functionality.

Beginning with an exploration of the database's design, each component is analyzed to uncover the tools and methods utilized. From designing the database system to implementing efficient data structures, every aspect of the design process is explained. Additionally, the report illuminates the diverse range of technologies employed to guarantee the database's robustness and scalability.

Following the discussion on design, the report delves into testing techniques essential for evaluating the database's efficiency and performance. Emphasis is placed on the importance of thorough testing to identify any potential issues.

Concluding the report is a hands-on demo illustrating how to use the database in real-world scenarios. Using a series of examples, users will be walked through common tasks, highlighting the user-friendly features of the database. Whether you're new to the system or a seasoned user, the demo is designed to equip you with the skills needed to make the most of the database for your specific needs. So, dive in and explore the database's capabilities firsthand!

2. Software Architecture

In this system, the software architecture is organized into three main components:

- **Bootstrapping node**
  This component consists of a Spring Boot application responsible for initiating the cluster and assigning new users to the system. It serves as the entry point for the system, facilitating the startup process and ensuring that new users are properly integrated into the cluster.

- **Affinity node**
  The Affinity Node is another Spring Boot application designed to handle both writing and reading operations to and from the database. This component is responsible for managing the data stored in the system, ensuring its integrity, and facilitating efficient access for users.

- **Node**
  The Node component is also a Spring Boot application, but it is limited to only reading documents from the database. Unlike the Affinity Node, this component does not have the ability to write data to the database. Instead, it focuses solely on retrieving and presenting information to users.

Communication between these nodes is facilitated using Docker, a platform that allows for the creation, deployment, and management of containerized applications. Docker provides a standardized environment for running applications, making it easier to ensure consistency and reliability across different nodes in the system.

By leveraging Docker for communication between nodes, the system benefits from enhanced scalability, flexibility, and portability. Docker containers encapsulate each component's dependencies and configurations, making it simple to deploy and manage them across various environments.

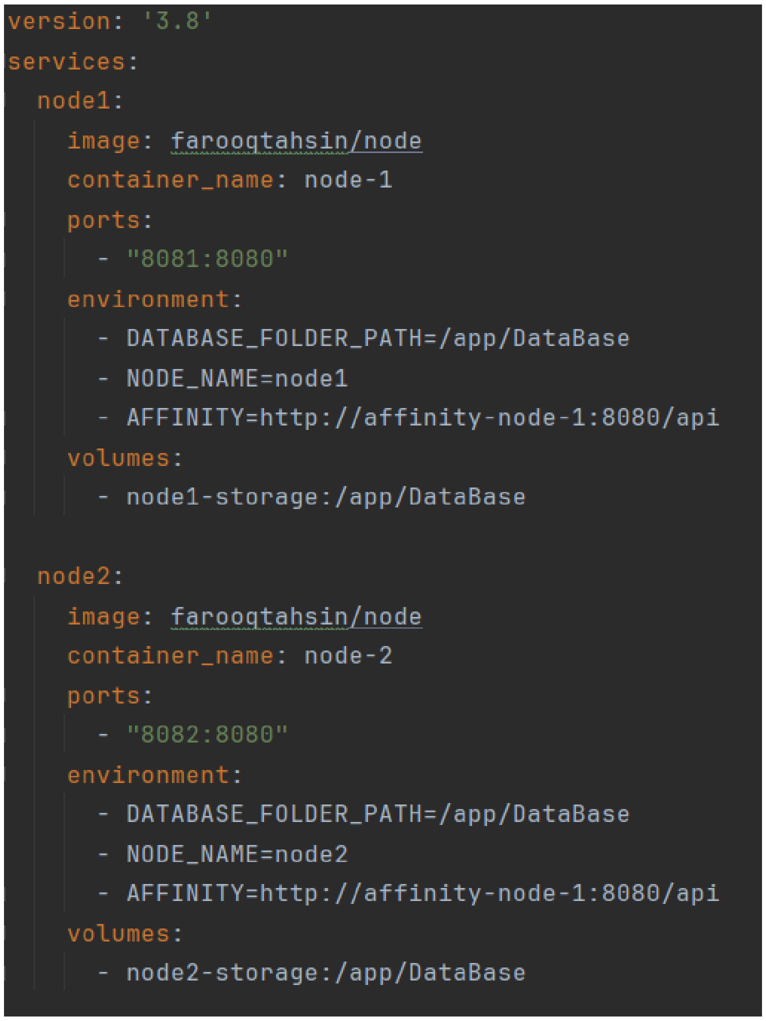
Overall, the software architecture of this system is designed to promote modularity, scalability, and efficiency, allowing for seamless interaction between different components while ensuring robust performance and reliability.

3. Database Implementation

In this chapter, the implementation of the database system will be elaborated, beginning from a broad perspective, and gradually delving into finer details, spanning from high level to low level.

3.1 Docker Container

Using a Docker Compose file, we can set up the initialization of node information such as the node name and port for communication with other nodes.



```yaml
version: '3.8'
services:
  node1:
    image: farooqtahsin/node
    container_name: node-1
    ports:
      - "8081:8080"
    environment:
      - DATABASE_FOLDER_PATH=/app/DataBase
      - NODE_NAME=node1
      - AFFINITY=http://affinity-node-1:8080/api
    volumes:
      - node1-storage:/app/DataBase

  node2:
    image: farooqtahsin/node
    container_name: node-2
    ports:
      - "8082:8080"
    environment:
      - DATABASE_FOLDER_PATH=/app/DataBase
      - NODE_NAME=node2
      - AFFINITY=http://affinity-node-1:8080/api
    volumes:
      - node2-storage:/app/DataBase
```

Running the docker-compose file sets up a network with different types of nodes: a bootstrapping node, regular nodes, and affinity nodes. Each node has its job in making sure everything runs smoothly. The bootstrapping node gets things started and helps new nodes join the network. Regular nodes and affinity nodes handle specific tasks to keep the system working well.

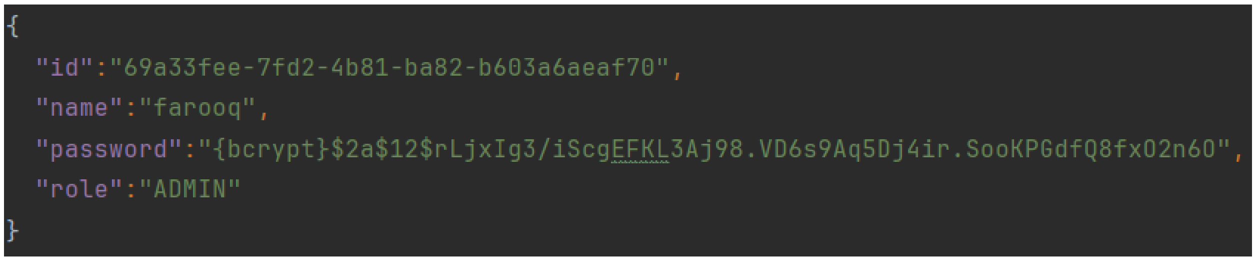
This setup isn't just about connecting nodes; it's also about making sure the system can handle more work when needed. With Docker Compose, we create a setup that's flexible and can grow as the system grows. So, even as things get busier, our system stays strong and can handle whatever comes its way.

3.2 Bootstrapping Node

This node has two main jobs:

- Getting the cluster started.
- Adding new users to the system.

It will have a folder called Database to store users as JSON files. Whenever a new user joins, their details will be saved in the Database/user folder. Each user's information will include:



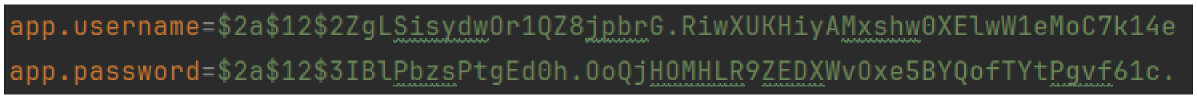
```json
{
  "id":"69a33fee-7fd2-4b81-ba82-b603a6aeaf70",
  "name":"farooq",
  "password":"{bcrypt}$2a$12$rLjxIg3/iScgEFKL3Aj98.VD6s9Aq5Dj4ir.SooKPGdfQ8fxO2n6O",
  "role":"ADMIN"
}
```

- ID: Unique to that user.
- Name: Their specific name.
- Password: Encrypted for security.
- Role: They'll be classified as either an ADMIN or USER.

The admin role is crucial as it allows for initiating the cluster and managing the addition of new users to the nodes. Alongside this, there will be a default admin present in the system to aid in communication between the nodes. This default admin will be automatically established within each node except for the bootstrapping node.

In practice, the default admin serves as a communication facilitator among the nodes. For instance, when the bootstrapping node needs to initialize the cluster, a process which involves distributing users across the nodes, it relies on this default admin. It's important to ensure security within the nodes, ensuring that only the admin can perform certain tasks. However, given the absence of any users initially, a default admin is created during the setup of each node to address this issue proactively.

Default admin information:



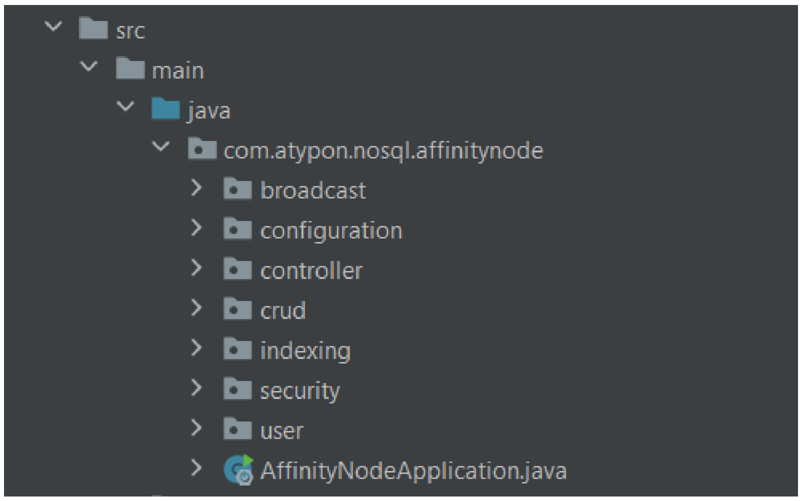
```
app.username=$2a$12$2ZgLSisydw0r1QZ8jpbrG.RiwXUKHiyAMxshw0XElwW1eMoC7k14e
app.password=$2a$12$3IBlPbzsPtgEd0h.0oQjHOMHLR9ZEDXWv0xe5BYQofTYtPgvf61c.
```

3.3 Affinity Node

This node will handle all operations on the database. In this section, it will describe how this node works, while other details will be mentioned in the remaining sections.

The key components of this node include:

- **Broadcast**
  - Facilitates broadcasting changes made to the database to other nodes.
- **Controller**
  - Furnishes endpoints for utilizing the node's functionalities.
- **CRUD**
  - Handles direct communication with the file system, encompassing reading and writing operations.
- **Indexing**
  - Implements indexing techniques to enhance reading efficiency.
- **Security**
  - Incorporates Spring Security to safeguard endpoints.
- **User**
  - Assists in managing user access to the node**.**



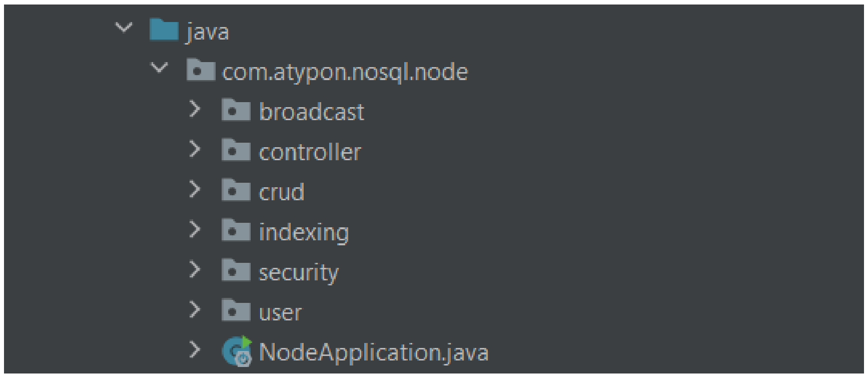
```
src
  main
    java
      com.atypon.nosql.affinitynode
        broadcast
        configuration
        controller
        crud
        indexing
        security
        user
        AffinityNodeApplication.java
```

3.4 Node

This node will handle only read operations on the database. In this section, it will describe how this node works, while other details will be mentioned in the remaining sections.

The key components of this node include:

- **Broadcast**
  - This part will be responsible for calling the affinity node when a write operation is called from this node.
- **Controller**
  - Furnishes endpoints for utilizing the node's functionalities.
- **CRUD**
  - Handles direct communication with the file system, encompassing reading operation.
- **Indexing**
  - Implements indexing techniques to enhance reading efficiency.
- **Security**
  - Incorporates Spring Security to safeguard endpoints.
- **User**
  - Assists in managing user access to the node**.**



```
java
  com.atypon.nosql.node
    broadcast
    controller
    crud
    indexing
    security
    user
    NodeApplication.java
```

3.5 Technologies

In this section all technologies used will be mentioned:

- **Docker**
  - Used to utilize each node as virtual machine and for communication between nodes.
- **Spring REST**
  - Use Spring Boot as it handles multithreaded server and provided many useful features that will help in this project.
- **Spring Security**
  - Used to generate the required security features to the API's.
- **Postman**
  - Used to test the API's.
- **JSON Library**
  - Used to parse, read, write, and modify the JSON files and texts.
- **Lombok Library**
  - Used to create setters, getters and contractors using java notations.

4. The Data Structure Used

Data structures are fundamental components of computer science, utilized to store and organize data in a manner facilitating easy access, modification, and manipulation. Each data structure boasts unique characteristics suitable for different types of applications. This chapter centers on a specific data structure commonly employed in programming the project.

I employ the map data structure within the HashIndexing class for indexing purposes in the NoSQL database system. This choice is driven by several factors that contribute to the effectiveness and efficiency of managing indexes:

- **Efficient Data Retrieval:**
  - The use of a hash map allows for constant-time average case complexity for key-based operations like adding, retrieving, and removing entries.
  - This efficiency is crucial for large-scale databases where quick data access is paramount for performance.

- **Fast Index Creation and Lookup:**
  - Hash maps provide rapid index creation, making them ideal for dynamic environments where indexes need to be created and updated frequently.
  - Retrieving data based on keys from a hash map is typically faster compared to linear search-based structures, especially when dealing with large datasets.

- **Space Efficiency:**
  - Hash maps offer space efficiency by storing data in key-value pairs, consuming memory proportional to the number of entries rather than the size of the entire dataset.
  - This makes hash maps suitable for systems with limited memory resources or when memory optimization is a concern.

- **Flexibility in Key-Value Pairing:**
  - Hash maps accommodate flexible key-value pairings, allowing complex data structures to be stored as values associated with unique keys.
  - This flexibility enables the storage of various metadata alongside data entries, enhancing the richness of information stored in the index.

- **Scalability:**
  - Hash maps exhibit good scalability characteristics, maintaining constant-time performance even as the size of the dataset grows.
  - This scalability makes hash maps suitable for applications requiring horizontal scaling, where additional nodes or resources can be added seamlessly to handle increased data volume.

In conclusion, the HashIndexing class leverages the hash map data structure to provide efficient, scalable, and flexible indexing capabilities for NoSQL database systems. Its utilization offers benefits such as fast data retrieval, space efficiency, flexibility, and ease of integration, making it a suitable choice for managing indexes in diverse application scenarios.

5. Multithreading and Locks

This chapter will be divided into two-part multithreading and locks. In each part all usage of these parts will be mentioned and explained in detail.

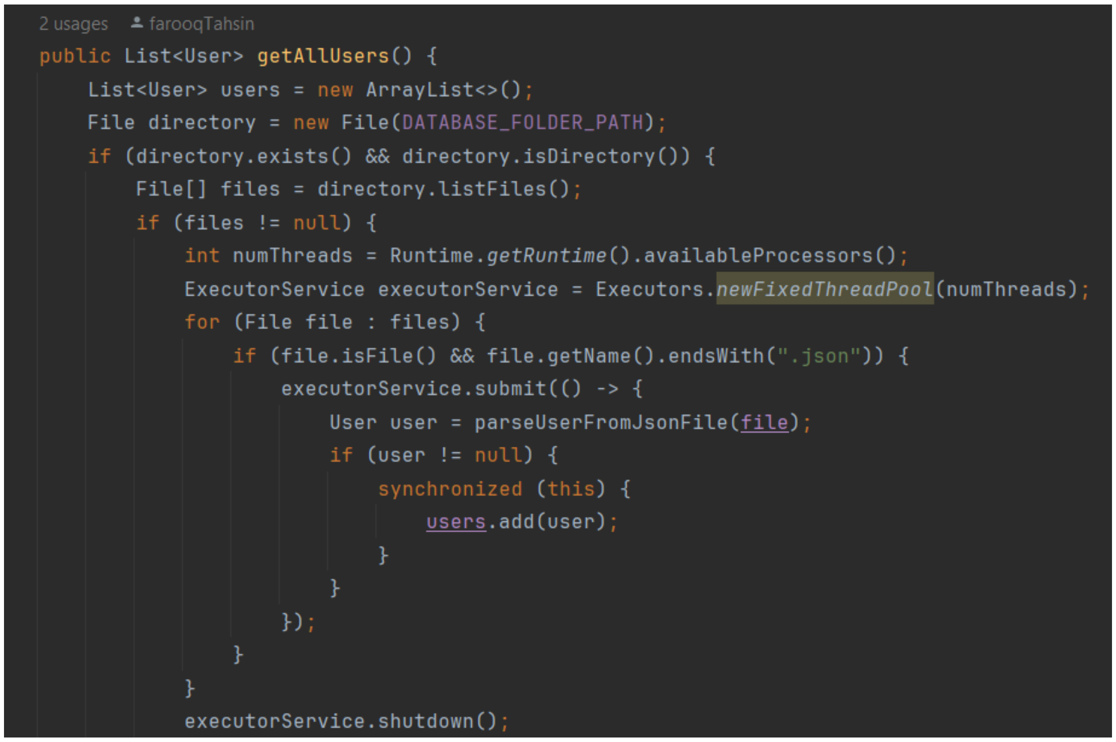
5.1 Multithreading

As mentioned earlier, Spring Boot will give a multithreaded server but still lots of work needs to be considered according to multithreading.

1. Multithreading in bootstrapping node:

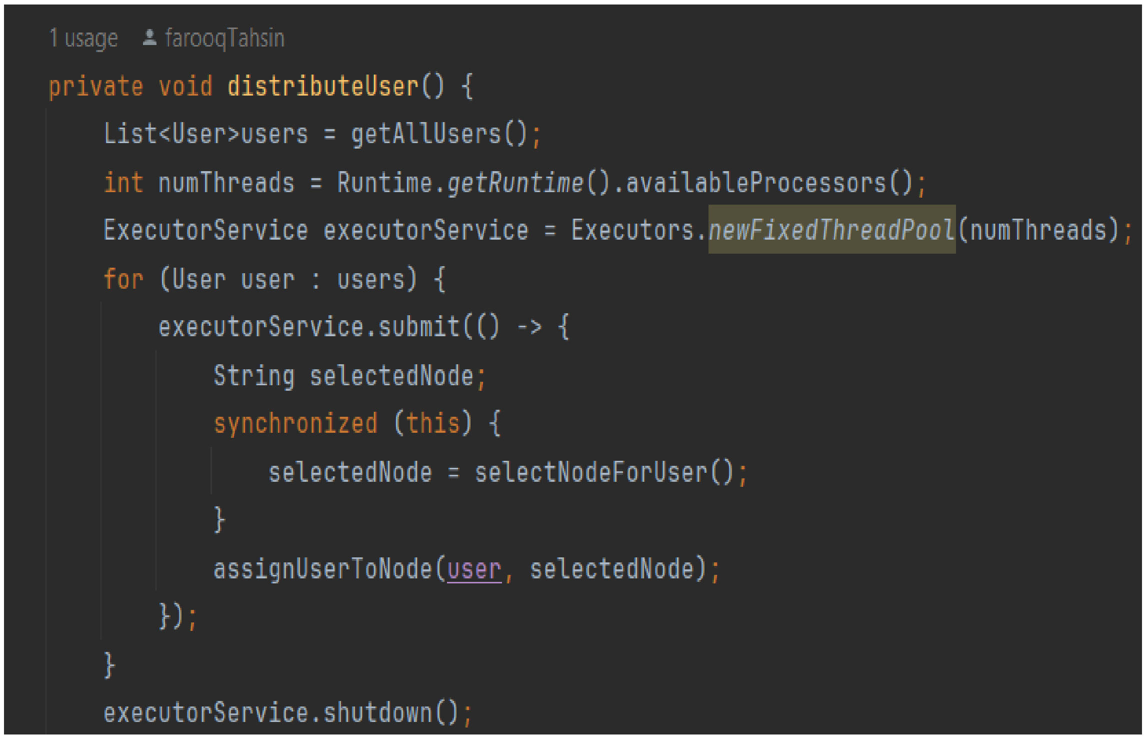
- **Used to optimize the retrieval of user data from the database during system initialization.**
  - This approach was chosen primarily to enhance system performance by leveraging concurrent processing capabilities. By utilizing multiple threads, the bootstrapping node can efficiently retrieve user information stored in JSON files within the database folder.
  - The implementation relies on an ExecutorService to manage the execution of multiple threads effectively. This service provides a convenient abstraction for handling thread pools and executing tasks asynchronously. A fixed-size thread pool is created where the number of threads is dynamically determined based on the available processing resources, typically equivalent to the number of processor cores.

- One of the main advantages of employing multithreading in this context is the ability to parallelize the retrieval of user data. Instead of sequentially processing each JSON file, multiple files can be processed simultaneously by different threads. This parallel processing significantly reduces the overall time required to retrieve all user data, thereby expediting the system startup process.



```
2 usages   farooqTahsin
public List<User> getAllUsers() {
    List<User> users = new ArrayList<>();
    File directory = new File(DATABASE_FOLDER_PATH);
    if (directory.exists() && directory.isDirectory()) {
        File[] files = directory.listFiles();
        if (files != null) {
            int numThreads = Runtime.getRuntime().availableProcessors();
            ExecutorService executorService = Executors.newFixedThreadPool(numThreads);
            for (File file : files) {
                if (file.isFile() && file.getName().endsWith(".json")) {
                    executorService.submit(() -> {
                        User user = parseUserFromJsonFile(file);
                        if (user != null) {
                            synchronized (this) {
                                users.add(user);
                            }
                        }
                    });
                }
            }
            executorService.shutdown();
```

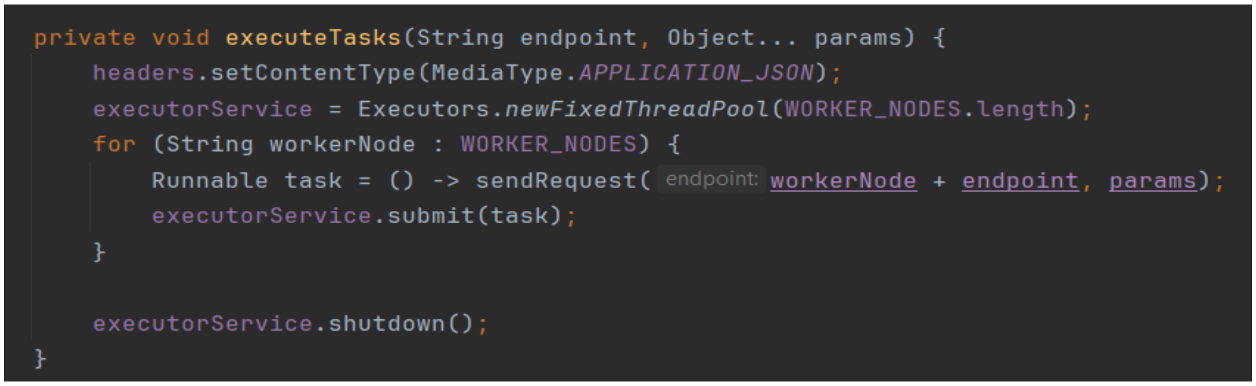
- **Used to distribute users to nodes within the system.**
  - This strategy is adopted to optimize the distribution process and ensure that user data is evenly spread across the available nodes.
  - A fixed-size thread pool is created based on the available processing resources, ensuring efficient utilization of system resources. Each user is processed asynchronously by submitting a task to the ExecutorService for execution.
  - Using it will improve performance through parallelization, optimized resource utilization by efficiently leveraging available processing resources, enhanced scalability to accommodate growing workloads seamlessly.



```
1 usage  farooqTahsin
private void distributeUser() {
    List<User>users = getAllUsers();
    int numThreads = Runtime.getRuntime().availableProcessors();
    ExecutorService executorService = Executors.newFixedThreadPool(numThreads);
    for (User user : users) {
        executorService.submit(() -> {
            String selectedNode;
            synchronized (this) {
                selectedNode = selectNodeForUser();
            }
            assignUserToNode(user, selectedNode);
        });
    }
    executorService.shutdown();
```

2. Multithreading in broadcasting:

- Used to enhance the efficiency of broadcasting.
    - Multithreading is utilized to enhance the efficiency of broadcasting requests to multiple worker nodes concurrently. Instead of sending requests to each worker node sequentially, which could lead to significant delays, the application leverages multithreading to distribute the workload across multiple threads.
    - By employing a fixed thread pool using ExecutorService, the application manages the execution of tasks in parallel. Each task represents a request to a specific worker node, and the thread pool ensures that a controlled number of threads are running simultaneously, preventing resource exhaustion and maximizing the utilization of available CPU cores.
    - This approach offers several advantages. First, it significantly reduces the time taken to broadcast requests to all worker nodes, thereby improving overall system responsiveness. Second, it allows the application to handle a larger volume of requests efficiently, making it more scalable. Finally, by utilizing multithreading, the application can make optimal use of available resources, enhancing its performance.



```java
private void executeTasks(String endpoint, Object... params) {
    headers.setContentType(MediaType.APPLICATION_JSON);
    executorService = Executors.newFixedThreadPool(WORKER_NODES.length);
    for (String workerNode : WORKER_NODES) {
        Runnable task = () -> sendRequest( endpoint: workerNode + endpoint, params);
        executorService.submit(task);
    }

    executorService.shutdown();
}
```

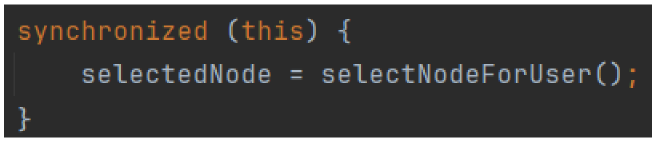
5.2 Locks

When designing a database, it is essential to ensure that both read and write operations are thread safe. This is because multiple threads may attempt to access the database simultaneously, which can result in conflicts and data inconsistencies.

1. Lock in bootstrapping node:

To ensure load balancing when assigning nodes to users during the distribution process, it's essential to use locks. This guarantees that the node selection process remains synchronized, preventing potential concurrency issues.

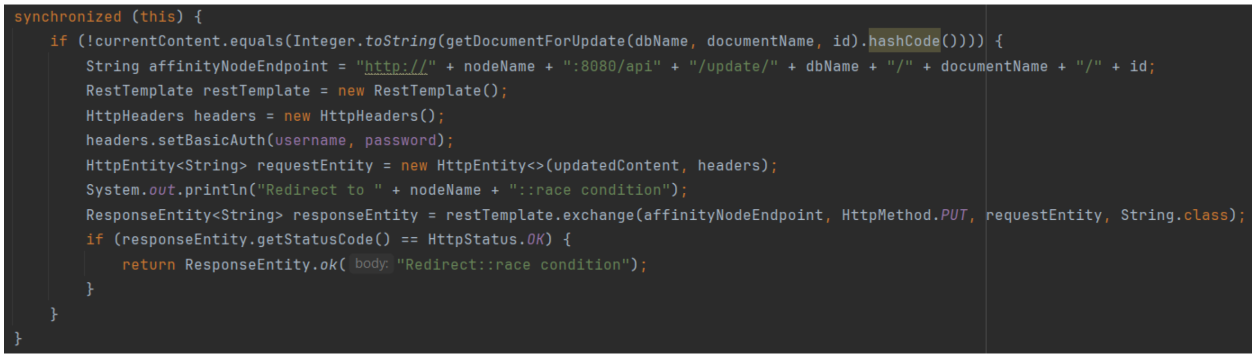
This critical section of code is enclosed within a synchronized block to ensure thread safety during node selection. Once a node is selected, the user is assigned to it.



```java
synchronized (this) {
    selectedNode = selectNodeForUser();
}
```

2. Lock in affinity node:

This ensures that only one thread can execute these sections at a time, preventing potential race conditions and data inconsistencies. The use of locks is particularly important when updating documents across multiple nodes in a distributed system, as concurrent access to shared resources could lead to conflicts. By enclosing the relevant code within a synchronized block, the method ensures that updates are carried out in a controlled and orderly manner, maintaining the integrity of the documents being modified. This approach enhances the reliability and consistency of document updates within the system, contributing to overall system stability and performance.



```java
synchronized (this) {
    if (!currentContent.equals(Integer.toString(getDocumentForUpdate(dbName, documentName, id).hashCode()))) {
        String affinityNodeEndpoint = "http://" + nodeName + ":8080/api" + "/update/" + dbName + "/" + documentName + "/" + id;
        RestTemplate restTemplate = new RestTemplate();
        HttpHeaders headers = new HttpHeaders();
        headers.setBasicAuth(username, password);
        HttpEntity<String> requestEntity = new HttpEntity<>(updatedContent, headers);
        System.out.println("Redirect to " + nodeName + "::race condition");
        ResponseEntity<String> responseEntity = restTemplate.exchange(affinityNodeEndpoint, HttpMethod.PUT, requestEntity, String.class);
        if (responseEntity.getStatusCode() == HttpStatus.OK) {
            return ResponseEntity.ok( body: "Redirect::race condition");
        }
    }
}
```

6. Indexing

In NoSQL databases, quickly getting the data you need is super important. That's where hash indexing comes in handy. This section is all about how I use hash indexing in this project.

How It Works:

The HashIndexing class is like the brain behind how I manage indexes in this database. When it starts up, it sets up some basic stuff like where the database files are stored, and a special kind of memory called a Map to keep track of our indexes.

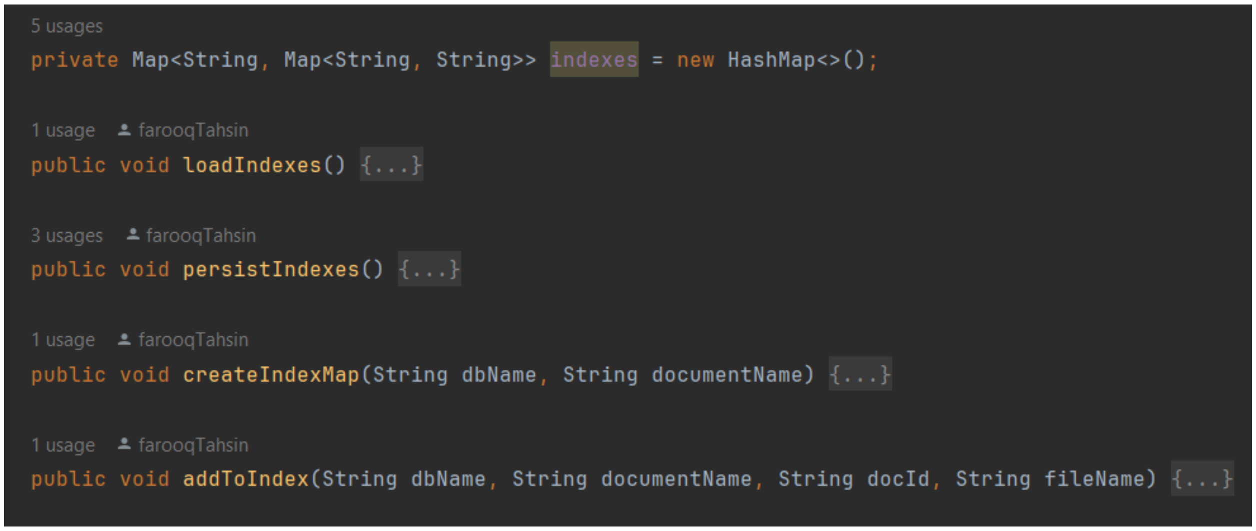
Loading indexes is the first big job of the HashIndexing class. It reads from a file called index.json to fill up its Map with the index data. It's like getting all the instructions on how to find things in our database. If there's no index file or there's a problem reading it, the class knows how to handle that without crashing the whole system.

Saving indexes back to the index.json file is another important task. After making changes to the index, like adding new data, the class turns everything back into a special code called JSON and writes it back to the file. Even if something goes wrong during this process, the class knows how to deal with it safely.

Managing Indexes:

The cool thing about the HashIndexing class is that it can create, update, and find indexes super quickly. When we create a new index, it's like adding a new way to find stuff in our database. Then, when we add data, like a new document, the class keeps track of where it is. And when we need to find something, we can ask the class to show us all the indexes, so we know exactly where to look.

Using hash indexing makes this database work faster and smoother. It helps to find data in a snap, which is really important, especially when we have lots of data to deal with.



```
5 usages
private Map<String, Map<String, String>> indexes = new HashMap<>();

1 usage  farooqTahsin
public void loadIndexes() {...}

3 usages  farooqTahsin
public void persistIndexes() {...}

1 usage  farooqTahsin
public void createIndexMap(String dbName, String documentName) {...}

1 usage  farooqTahsin
public void addToIndex(String dbName, String documentName, String docId, String fileName) {...}
```

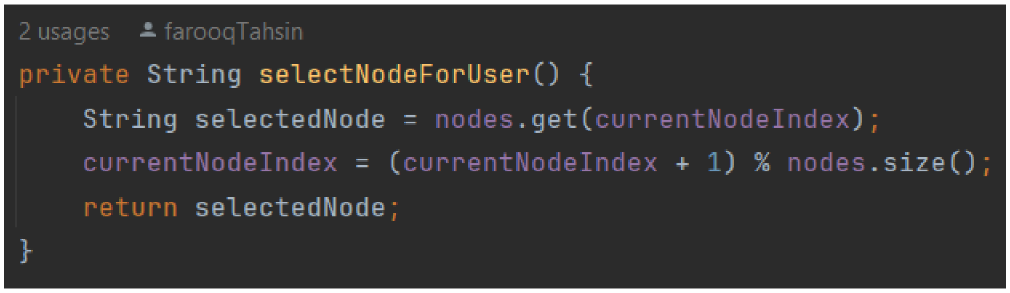
7. Node Hashing and Load Balancing

To make sure that users are evenly distributed across nodes when setting up the initial nodes, a straightforward technique is employed and thoroughly tested for effectiveness. Details about testing this technique will be provided in the testing section.

Given that Docker is utilized in this project, at the start of the application, we already know the number of nodes available to serve users. Additionally, if there are existing users in the system, their count is also known. Therefore, in the bootstrapping node, I utilize a variable to determine which node should be assigned to a user to maintain load balance.

Each time a new user signs up for the system and after the registration process, they are assigned to a node in a way that ensures load balancing behind the scenes.

This approach outlines the process where a user is initially assigned their node using the current node index variable. Subsequently, this variable is updated to point to the next node, ensuring load balancing.



```
private String selectNodeForUser() {
    String selectedNode = nodes.get(currentNodeIndex);
    currentNodeIndex = (currentNodeIndex + 1) % nodes.size();
    return selectedNode;
}
```

8. Communication Protocols Between Nodes

The communication protocol relies on the HTTP(S) protocol, a widely used standard for data communication on the World Wide Web. Specifically, the application employs HTTP requests to convey instructions from the central service to each worker node. These requests are typically of the POST method, allowing the transmission of data in the request body, which contains essential parameters like the database name, document name, content, and unique identifiers.

Furthermore, the communication protocol leverages HTTP basic authentication to ensure secure access to the worker nodes. By embedding the username and password within the HTTP headers of each request, the application verifies its identity with the worker nodes, thereby preventing unauthorized access and ensuring data integrity.

The worker nodes, identified by their respective URLs, receive the HTTP requests, and execute the specified commands accordingly. Upon receiving a request, each worker node processes the instructions contained within the request body, such as creating or deleting databases, adding, or updating documents, and responding with an appropriate HTTP status code to indicate the success or failure of the operation.

Overall, the communication protocol employed in this data base is built upon industry-standard HTTP-based mechanisms, ensuring compatibility, reliability, and security in transmitting commands and data between the central service and distributed worker nodes. Through the effective utilization of HTTP requests and basic authentication, the application facilitates seamless communication and coordination in managing NoSQL databases across multiple nodes in a distributed environment.

9. Security Issues

In this system, Spring Security is used to keep all nodes safe. We have two roles: one for regular users who can do stuff like adding or deleting data, and another for admins who set up the cluster.

But there's a tricky part with our security setup. Each node is locked down, only letting in the user it's assigned to. So, when we're using HTTP to talk between nodes, things get a bit complicated. Say Node 1 gets a request to add a new document to the database. It then needs to tell all the other nodes about it. But how does it know the right login details for each node?

Imagine if Node 1 had to talk to the other 999 nodes. That would mean sending 999 requests, each with different login info. To make things easier, I came up with a solution: I made a default admin account on each node. This way, when nodes need to talk to each other, they can just use the login details for this default admin.

This idea has some great perks. It makes communication simpler because we only need to keep track of one set of login details for each node. Plus, it makes the system more flexible. New nodes can join without needing lots of complicated setups.

And it makes things more reliable too. By using the default admin account for all communications between nodes, we keep things consistent and less likely to go wrong. This means fewer errors and a more stable system overall.

So, by using default admin login details for node-to-node talks, we solve the tricky problem of handling lots of different logins. And it makes our system simpler, more flexible, and more dependable.

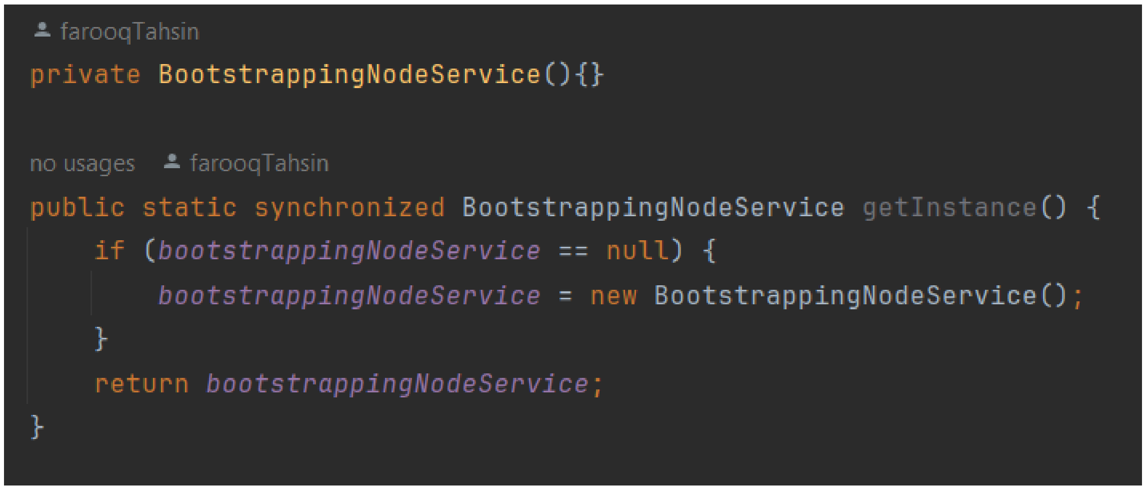
10. Design Patterns

The Singleton design pattern is a blueprint frequently used in software engineering to ensure that a class has only one instance and provides a global point of access to that instance. It's like having a single key to a room—no matter how many people need to enter, they all use the same key to access the room.

In this system, the BootstrappingNodeService class follows the Singleton pattern. This means that only one instance of the BootstrappingNodeService can exist throughout the application's lifecycle. The constructor of the class is made private, preventing other classes from creating instances of it directly.

Instead of allowing direct instantiation, the BootstrappingNodeService provides a static method called getInstance(). This method is responsible for creating and returning the single instance of the class. If an instance doesn't already exist, it creates one; otherwise, it returns the existing instance. This ensures that the same instance is reused whenever the class is accessed.

Overall, the usage of singleton ensures that only one instance of the service exists throughout the application's lifecycle, offering benefits such as efficient resource utilization, centralized access, consistent behavior, and simplified configuration management. This ensures that the bootstrapping process, responsible for distributing users among nodes in the cluster, is executed reliably and with optimal performance, while also facilitating scalability and flexibility as the application evolves.



```
farooqTahsin
private BootstrappingNodeService(){}

no usages   farooqTahsin
public static synchronized BootstrappingNodeService getInstance() {
    if (bootstrappingNodeService == null) {
        bootstrappingNodeService = new BootstrappingNodeService();
    }
    return bootstrappingNodeService;
}
```

11. Clean Code

- **Separation of Concerns:**
  - While the AffinityNodeController manages HTTP requests for database operations, the UserController handles user-related functionalities. Ensuring each controller focuses solely on its respective domain would enhance clarity and maintainability.

- **Code Reusability:**
  - The SecurityConfiguration class is responsible for managing user authentication and authorization. It's essential to abstract common security-related functionalities into reusable components, promoting code reuse and reducing duplication across the application.

- **Error Handling:**
  - Like other parts of the code, the UserController could benefit from improved error handling. Providing meaningful error messages and handling exceptional cases gracefully would enhance the user experience and facilitate troubleshooting.

- **Consistency and Naming Conventions:**
  - Maintaining consistent naming conventions across classes, methods, and variables improves code readability and comprehension. Ensuring uniformity in naming conventions, such as camelCase for method names and PascalCase for class names, would contribute to code consistency.

- **Logging:**
  - Including appropriate logging statements, especially for error scenarios and critical operations, aids in debugging and monitoring the application's behavior.

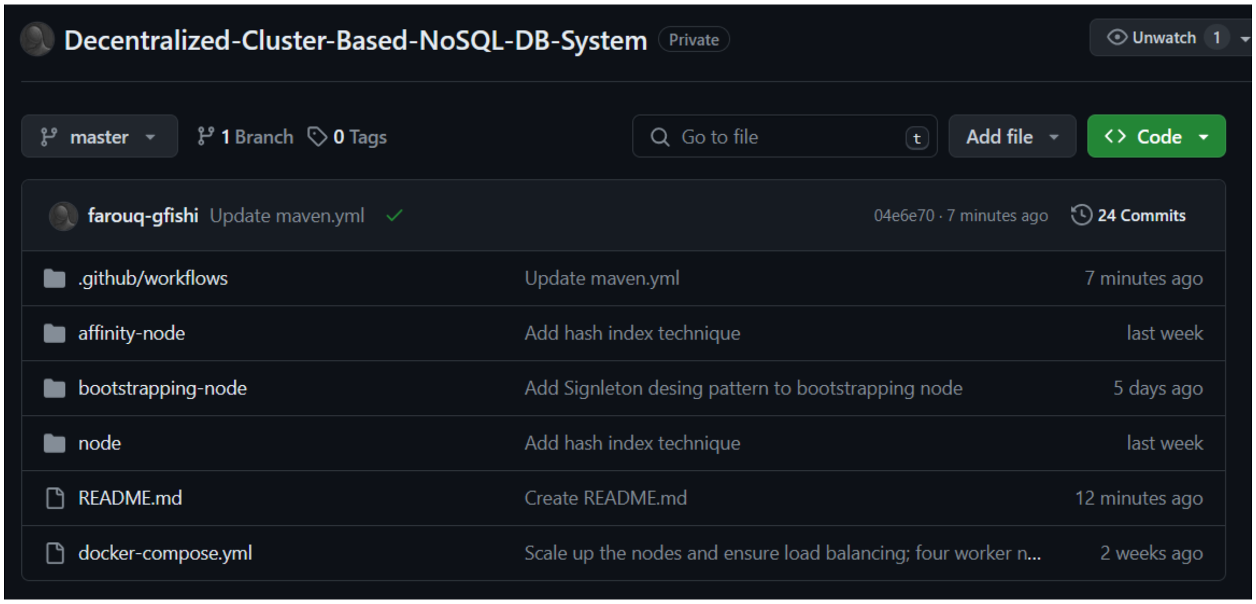
- **Consistent Formatting:**
  - Consistency in code formatting helps maintain readability. Ensure consistent indentation, spacing, and brace placement throughout the codebase.

- **Dependency Injection:**
  - Dependency injection is utilized effectively, particularly in the controller and security configuration classes, improving testability and decoupling.

12. DevOps practices

In my month of work, I used GitHub a lot. It was like my online desk where I saved all my project updates. With GitHub, I could try out different things and easily go back if something didn't work. It made my work easier, letting me focus on coding instead of worrying about saving stuff.

GitHub helped me stick to a good routine for building my project. I could test different parts separately to make sure everything worked smoothly. Even though I was working alone, GitHub's tools made me feel accountable and made my work more organized. Plus, it helped me set up tests and deploy my project faster, making everything run more smoothly.

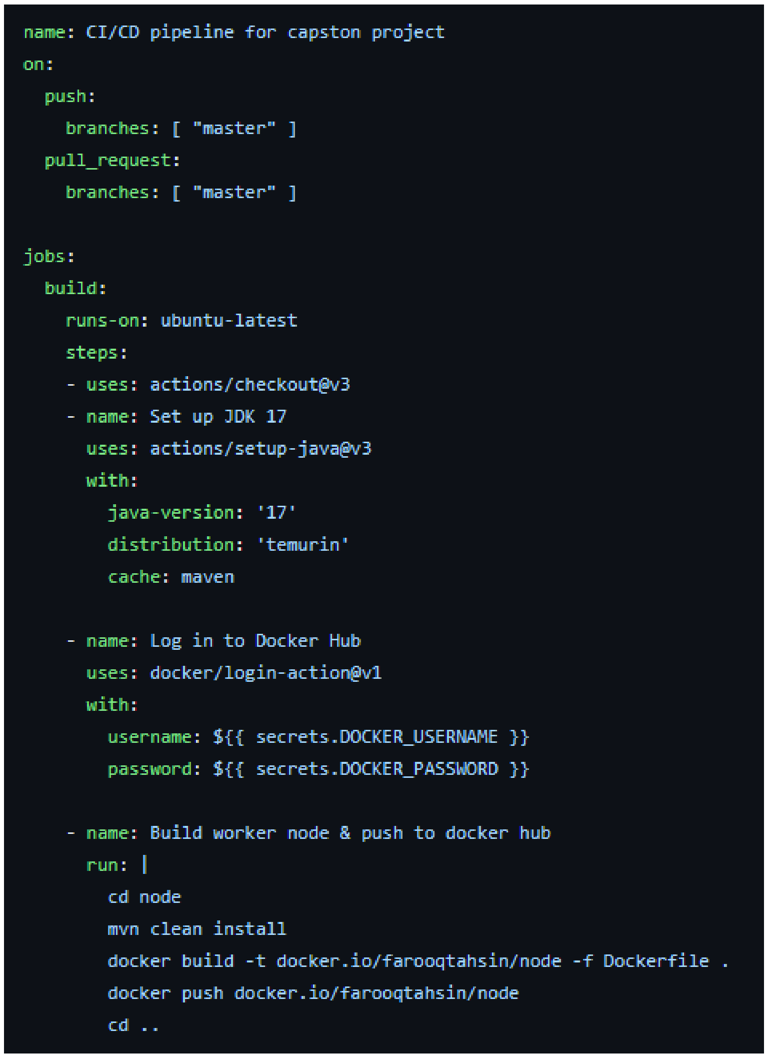


farouq-gfishi Update maven.yml ✓ 04e6e70 · 7 minutes ago 24 Commits

.github/workflows	Update maven.yml	7 minutes ago
affinity-node	Add hash index technique	last week
bootstrapping-node	Add Signleton desing pattern to bootstrapping node	5 days ago
node	Add hash index technique	last week
README.md	Create README.md	12 minutes ago
docker-compose.yml	Scale up the nodes and ensure load balancing; four worker n...	2 weeks ago

Build CI/CD pipeline:

I set up a CI/CD pipeline to make my work easier and faster. Without it, I'd have to manually run Maven each time I made a change, then push the code to Docker, and so on. But with this pipeline, everything happens automatically. Whenever I make a change, the pipeline kicks in, runs tests, builds the project, and deploys it using Docker containers. This saves me a ton of effort and time. Plus, using Docker makes sure that my project behaves the same way in any environment. With CI/CD, it's easier for me to work on new features and improvements, and it encourages collaboration because everything gets delivered smoothly.

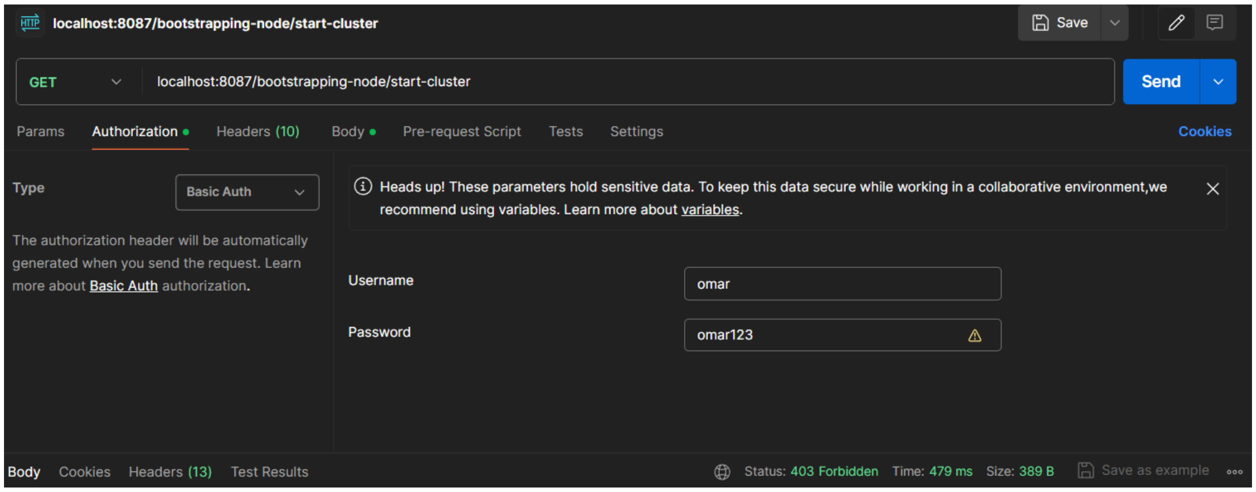


```
name: CI/CD pipeline for capston project
on:
  push:
    branches: [ "master" ]
  pull_request:
    branches: [ "master" ]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    - name: Set up JDK 17
      uses: actions/setup-java@v3
      with:
        java-version: '17'
        distribution: 'temurin'
        cache: maven

    - name: Log in to Docker Hub
      uses: docker/login-action@v1
      with:
        username: ${{ secrets.DOCKER_USERNAME }}
        password: ${{ secrets.DOCKER_PASSWORD }}

    - name: Build worker node & push to docker hub
      run: |
        cd node
        mvn clean install
        docker build -t docker.io/farooqtahsin/node -f Dockerfile .
        docker push docker.io/farooqtahsin/node
        cd ..
```

13. Code Testing

13.1 Bootstrapping Node Testing

Verify that only the admin responsible for starting the cluster can do so. For example, we have two users: Omar (USER) and Farooq (ADMIN).



GET localhost:8087/bootstrapping-node/start-cluster

Type: Basic Auth

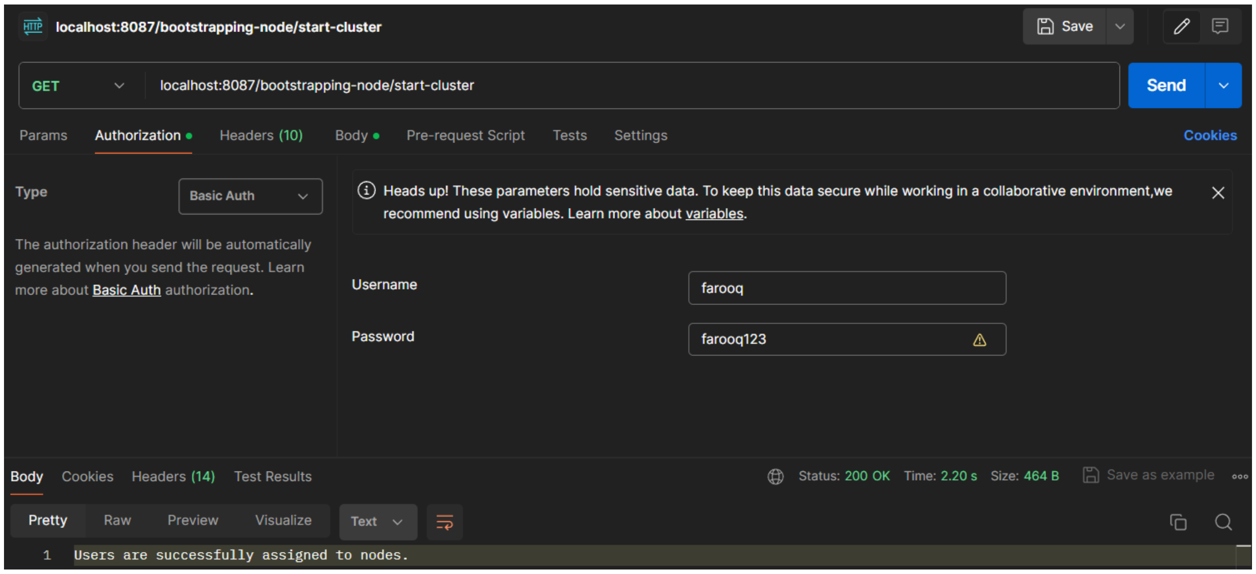
The authorization header will be automatically generated when you send the request. Learn more about Basic Auth authorization.

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment,we recommend using variables. Learn more about variables.

Username: omar

Password: omar123

Status: 403 Forbidden Time: 479 ms Size: 389 B



GET localhost:8087/bootstrapping-node/start-cluster

Type: Basic Auth

The authorization header will be automatically generated when you send the request. Learn more about Basic Auth authorization.

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment,we recommend using variables. Learn more about variables.

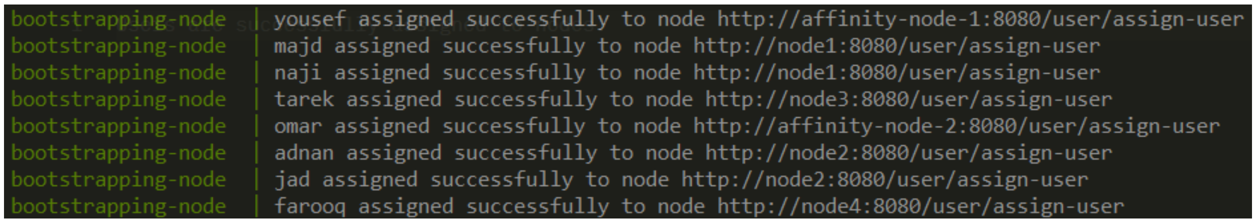
Username: farooq

Password: farooq123

Status: 200 OK Time: 2.20 s Size: 464 B

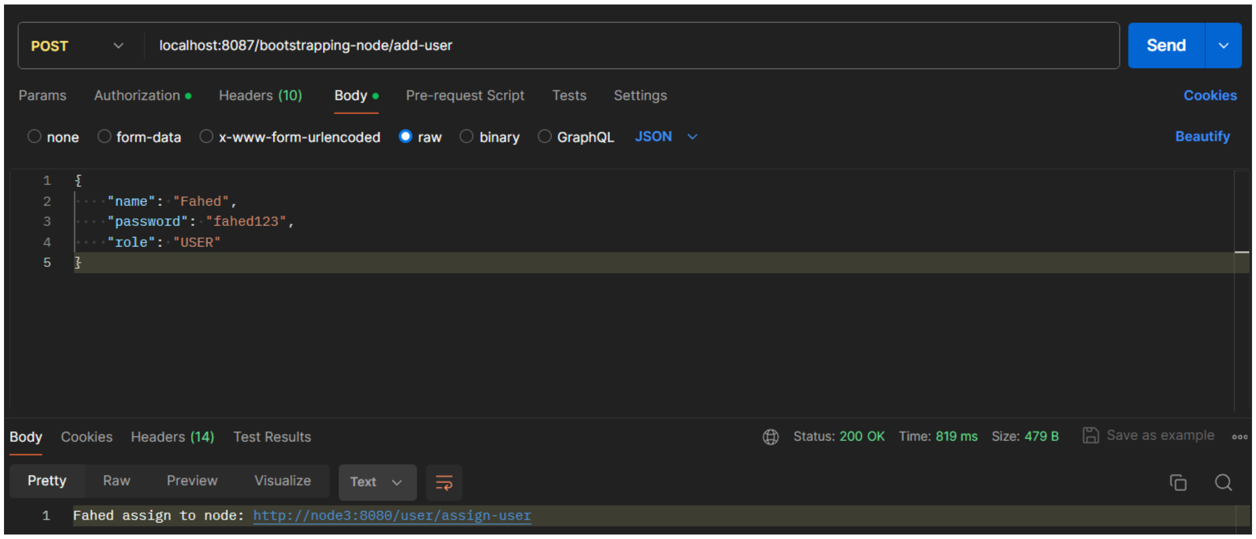
```
Users are successfully assigned to nodes.
```

Check if users are distributed evenly across nodes for load balancing. For instance, if there are 8 users and 6 nodes, Nodes 1 and 2 should each get two users, while the rest of the nodes should get one user each.



```
bootstrapping-node  | yousef assigned successfully to node http://affinity-node-1:8080/user/assign-user
bootstrapping-node  | majd assigned successfully to node http://node1:8080/user/assign-user
bootstrapping-node  | naji assigned successfully to node http://node1:8080/user/assign-user
bootstrapping-node  | tarek assigned successfully to node http://node3:8080/user/assign-user
bootstrapping-node  | omar assigned successfully to node http://affinity-node-2:8080/user/assign-user
bootstrapping-node  | adnan assigned successfully to node http://node2:8080/user/assign-user
bootstrapping-node  | jad assigned successfully to node http://node2:8080/user/assign-user
bootstrapping-node  | farooq assigned successfully to node http://node4:8080/user/assign-user
```

Test adding a new user while the system is running to ensure load balancing. For example, if a new user is added, they should be assigned to a node with the least load, like Node 3 if it only has one user.

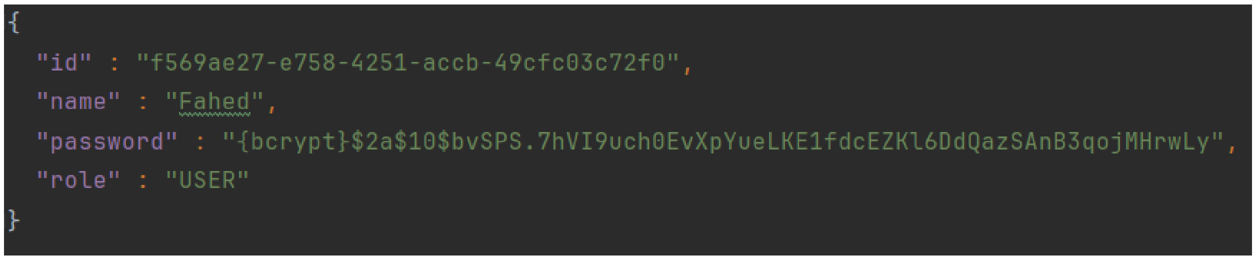


POST localhost:8087/bootstrapping-node/add-user

```
{
    "name": "Fahed",
    "password": "fahed123",
    "role": "USER"
}
```

Status: 200 OK Time: 819 ms Size: 479 B

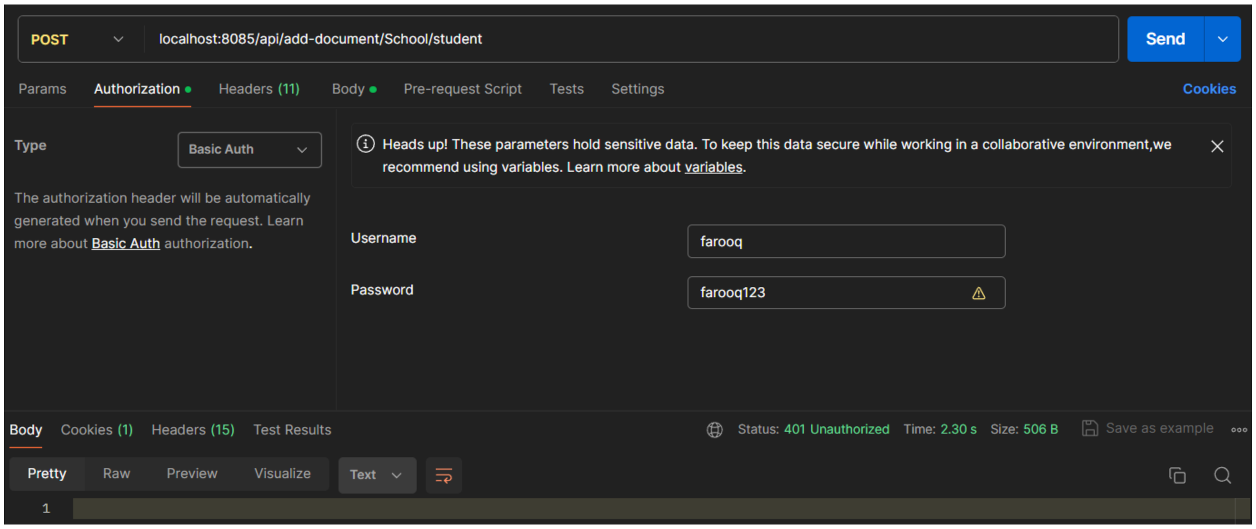
```
Fahed assign to node: http://node3:8080/user/assign-user
```



```
{
  "id" : "f569ae27-e758-4251-accb-49cfc03c72f0",
  "name" : "Fahed",
  "password" : "{bcrypt}$2a$10$bvSPS.7hVI9uch0EvXpYueLKE1fdcEZKl6DdQazSAnB3qojMHrwLy",
  "role" : "USER"
}
```

13.2 Affinity Node Testing

Validate that only the designated user (e.g., Yousef) can access the affinity node.



POST localhost:8085/api/add-document/School/student

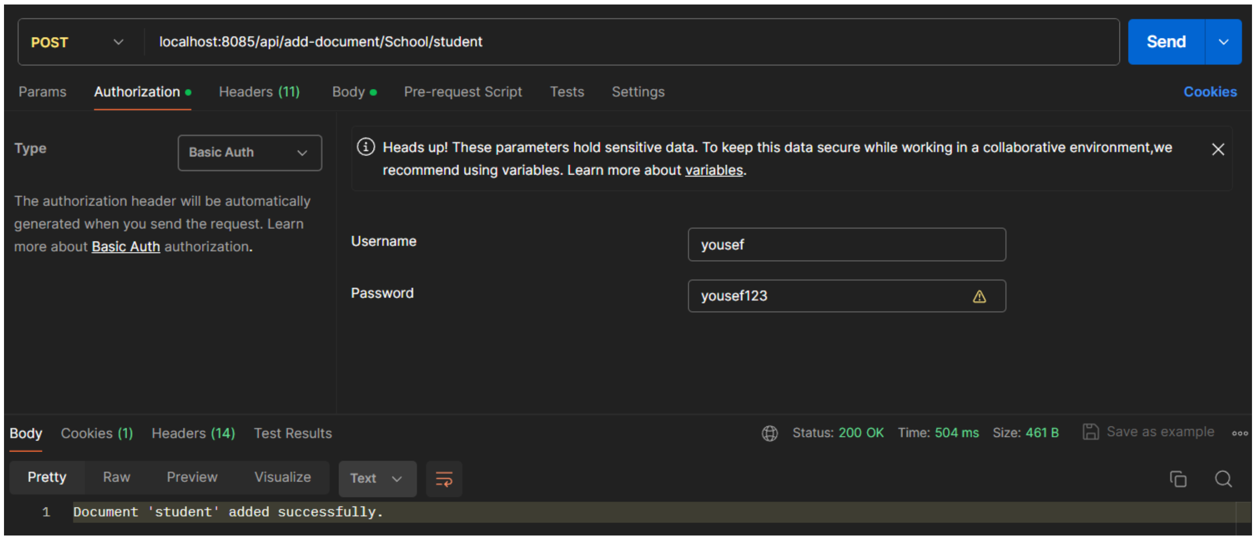
Type: Basic Auth

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment,we recommend using variables. Learn more about variables.

Username: farooq

Password: farooq123

Status: 401 Unauthorized Time: 2.30 s Size: 506 B



POST localhost:8085/api/add-document/School/student

Type: Basic Auth

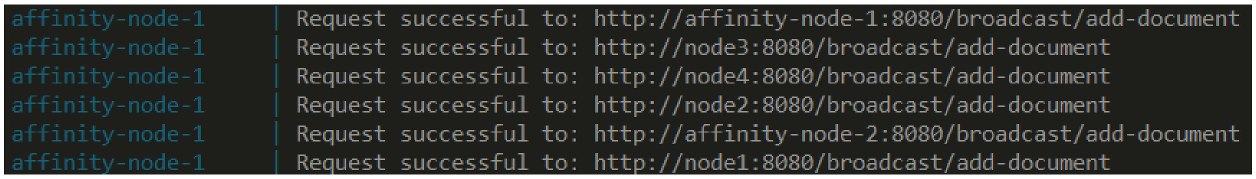
Username: yousef

Password: yousef123

Status: 200 OK Time: 504 ms Size: 461 B

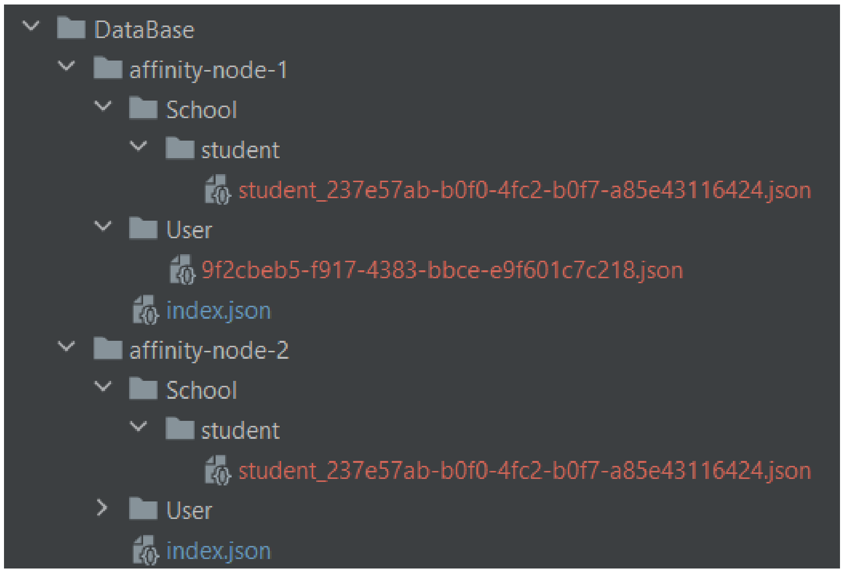
```
Document 'student' added successfully.
```

Ensure that broadcasting occurs after writing to the database.

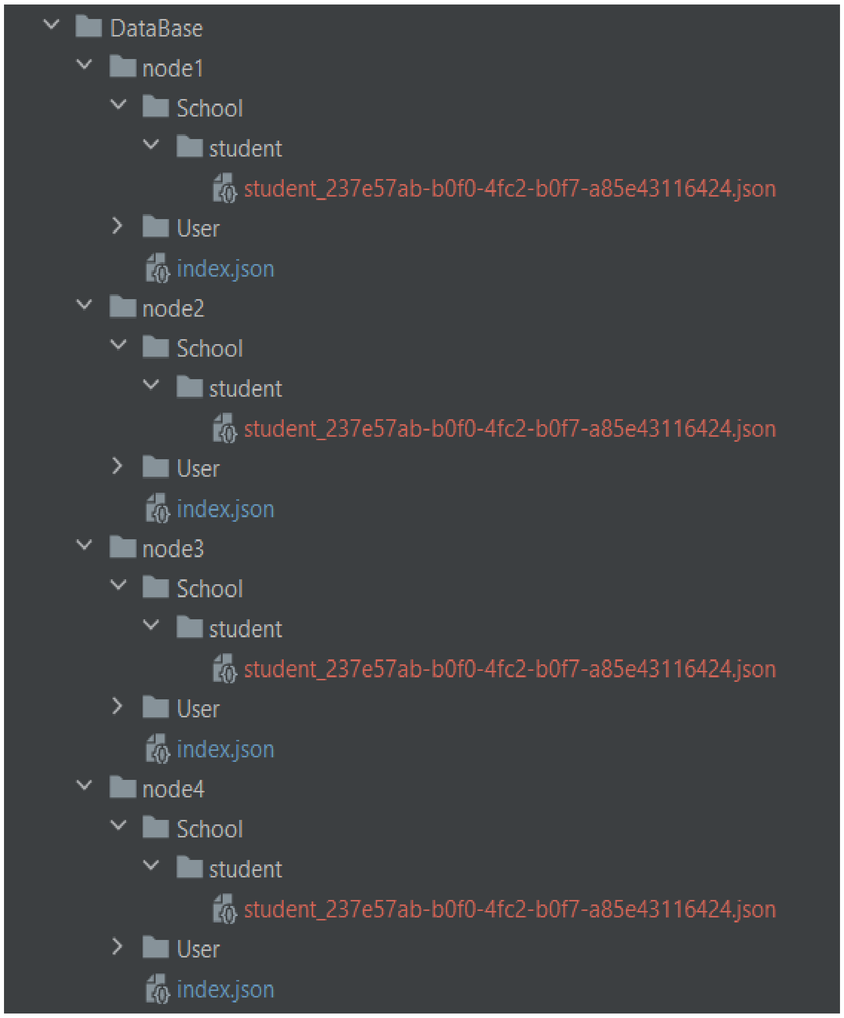


```
affinity-node-1     | Request successful to: http://affinity-node-1:8080/broadcast/add-document
affinity-node-1     | Request successful to: http://node3:8080/broadcast/add-document
affinity-node-1     | Request successful to: http://node4:8080/broadcast/add-document
affinity-node-1     | Request successful to: http://node2:8080/broadcast/add-document
affinity-node-1     | Request successful to: http://affinity-node-2:8080/broadcast/add-document
affinity-node-1     | Request successful to: http://node1:8080/broadcast/add-document
```

Verify consistency.



```
DataBase
  affinity-node-1
    School
      student
        student_237e57ab-b0f0-4fc2-b0f7-a85e43116424.json
    User
      9f2cbeb5-f917-4383-bbce-e9f601c7c218.json
    index.json
  affinity-node-2
    School
      student
        student_237e57ab-b0f0-4fc2-b0f7-a85e43116424.json
    User
    index.json
```

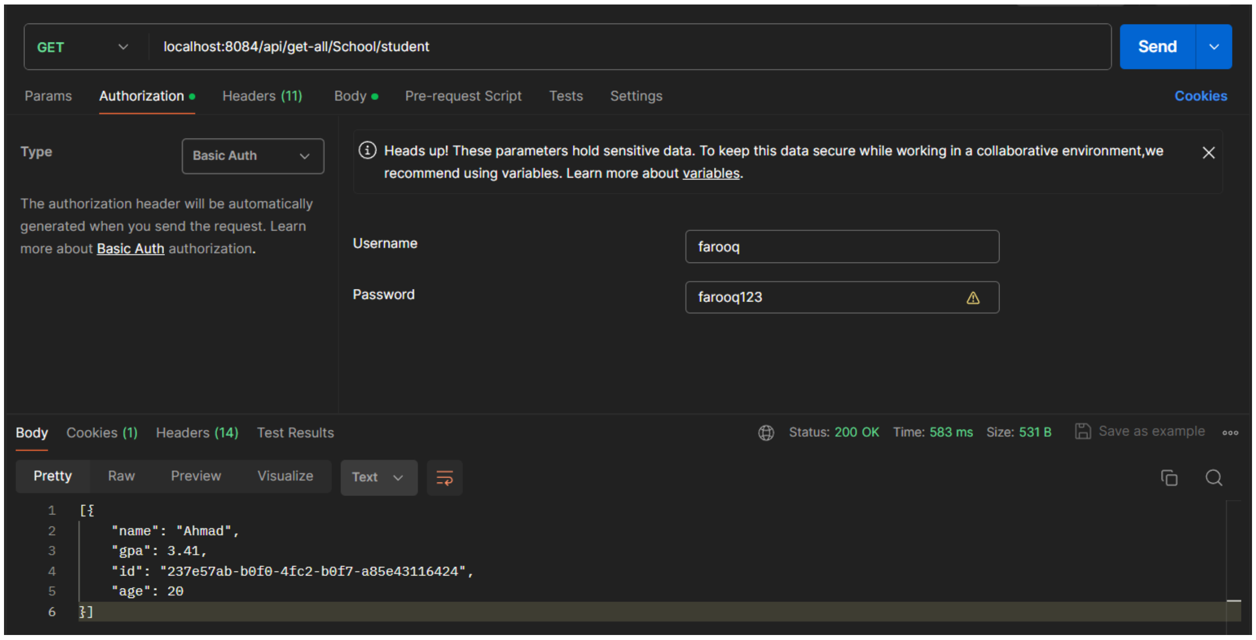


```
DataBase
  node1
    School
      student
        student_237e57ab-b0f0-4fc2-b0f7-a85e43116424.json
    User
    index.json
  node2
    School
      student
        student_237e57ab-b0f0-4fc2-b0f7-a85e43116424.json
    User
    index.json
  node3
    School
      student
        student_237e57ab-b0f0-4fc2-b0f7-a85e43116424.json
    User
    index.json
  node4
    School
      student
        student_237e57ab-b0f0-4fc2-b0f7-a85e43116424.json
    User
    index.json
```

13.3 Worker Node Testing

In node 4 there is only one user which is Farooq.

Test retrieving data added from the affinity node.



GET localhost:8084/api/get-all/School/student

Type: Basic Auth

The authorization header will be automatically generated when you send the request. Learn more about Basic Auth authorization.

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment,we recommend using variables. Learn more about variables.

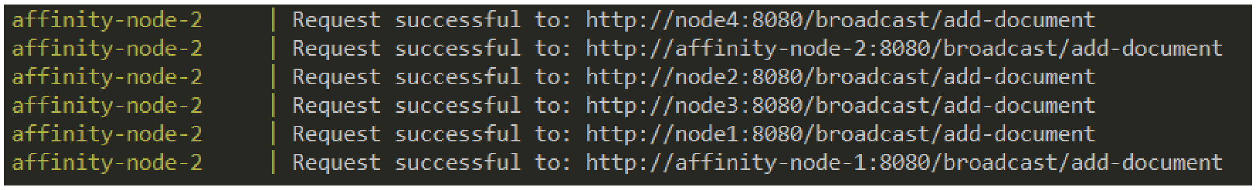
Username: farooq

Password: farooq123

Status: 200 OK Time: 583 ms Size: 531 B

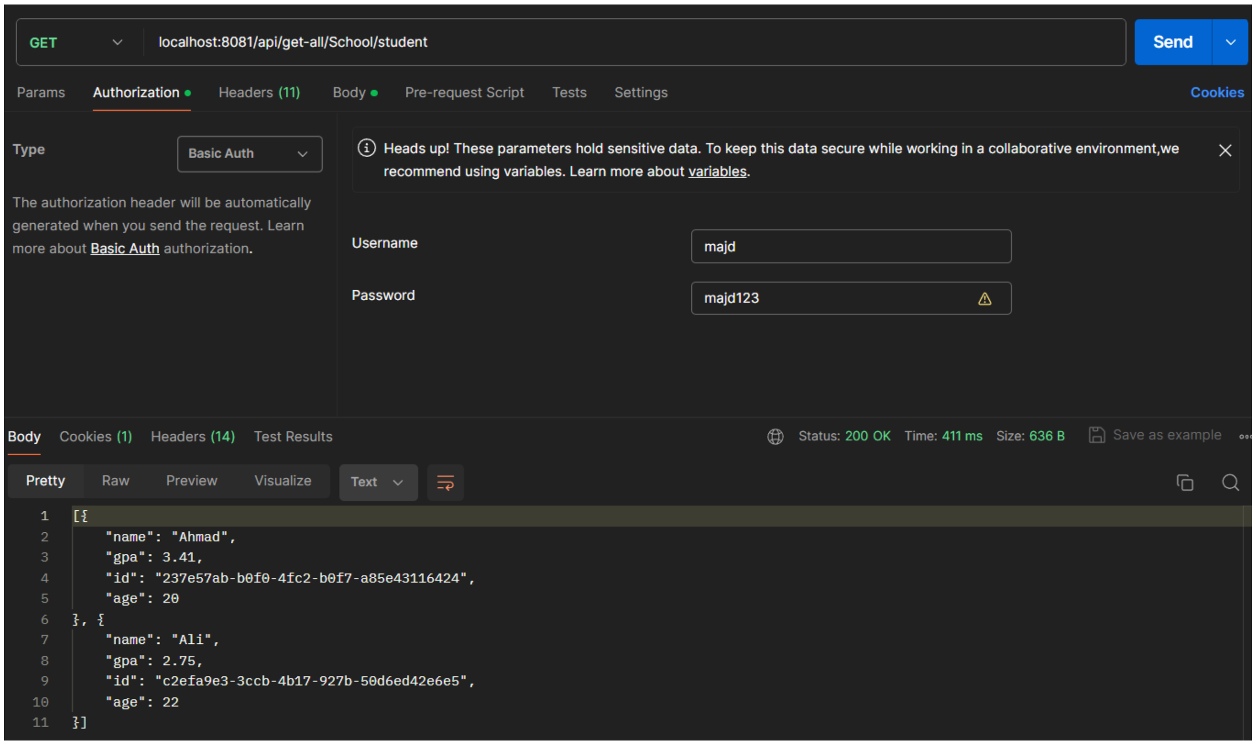
```
[{
    "name": "Ahmad",
    "gpa": 3.41,
    "id": "237e57ab-b0f0-4fc2-b0f7-a85e43116424",
    "age": 20
}]
```

Add a new document from Node 4 to a student and observe which affinity node handles the addition. For example, Node 4 might send a request to Affinity Node 2 to write the document, and then Affinity Node 4 broadcasts the changes to other nodes.



```
affinity-node-2      | Request successful to: http://node4:8080/broadcast/add-document
affinity-node-2      | Request successful to: http://affinity-node-2:8080/broadcast/add-document
affinity-node-2      | Request successful to: http://node2:8080/broadcast/add-document
affinity-node-2      | Request successful to: http://node3:8080/broadcast/add-document
affinity-node-2      | Request successful to: http://node1:8080/broadcast/add-document
affinity-node-2      | Request successful to: http://affinity-node-1:8080/broadcast/add-document
```

After adding two documents to a student, try retrieving them from Node 1.



localhost:8081/api/get-all/School/student

Type: Basic Auth

The authorization header will be automatically generated when you send the request. Learn more about Basic Auth authorization.

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment,we recommend using variables. Learn more about variables.

Username: majd

Password: majd123

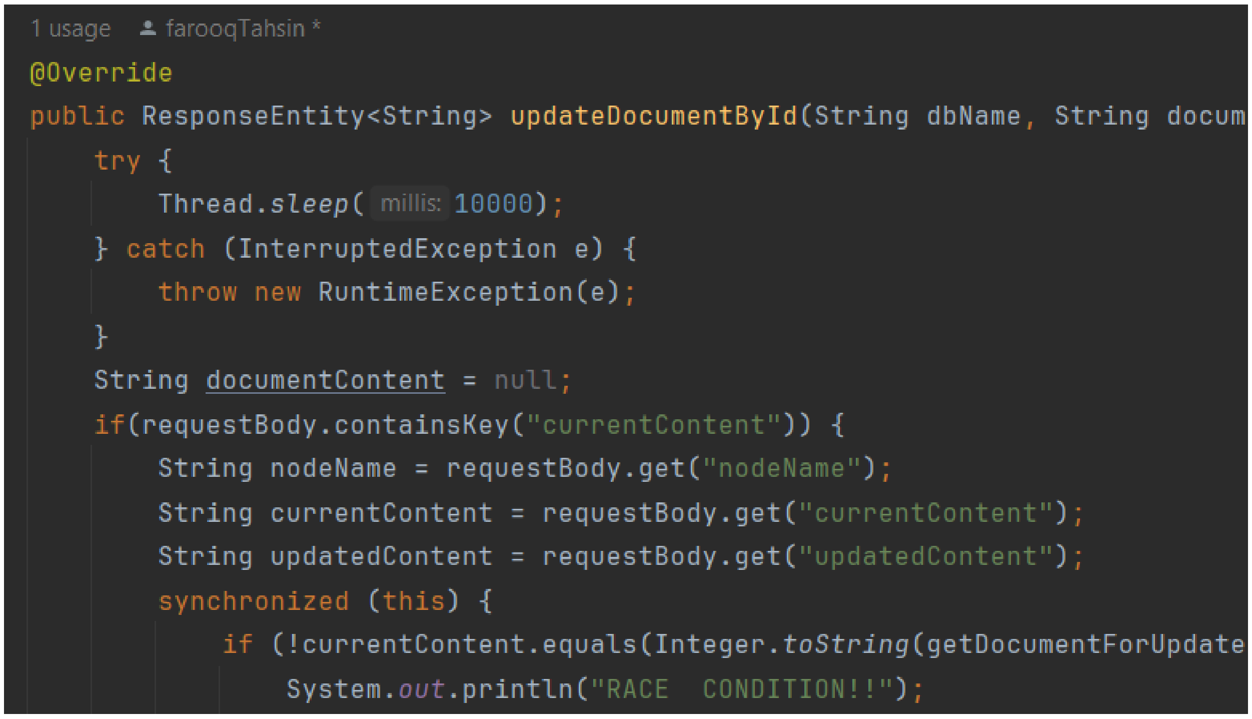
Status: 200 OK Time: 411 ms Size: 636 B

```
[{
    "name": "Ahmad",
    "gpa": 3.41,
    "id": "237e57ab-b0f0-4fc2-b0f7-a85e43116424",
    "age": 20
}, {
    "name": "Ali",
    "gpa": 2.75,
    "id": "c2efa9e3-3ccb-4b17-927b-50d6ed42e6e5",
    "age": 22
}]
```

13.4 Race Condition Testing

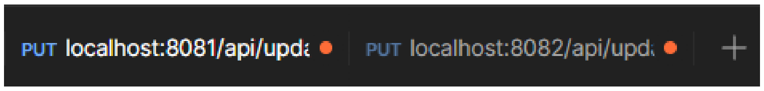
Test what happens when two requests attempt to update a document simultaneously.

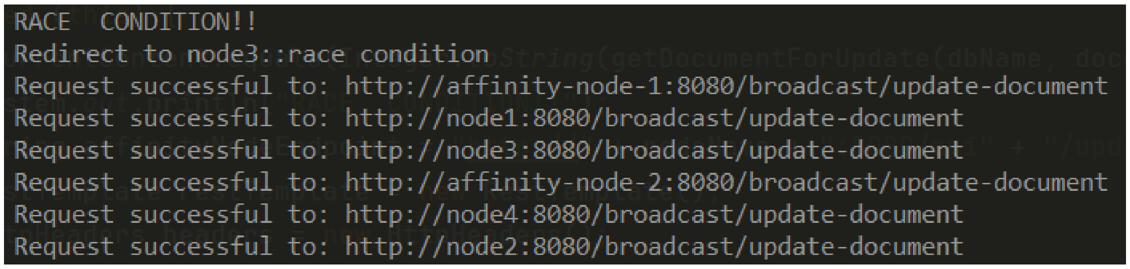
Introduce delays to simulate real-world conditions.



```
1 usage   farooqTahsin *
@Override
public ResponseEntity<String> updateDocumentById(String dbName, String docum
    try {
        Thread.sleep( millis: 10000);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    String documentContent = null;
    if(requestBody.containsKey("currentContent")) {
        String nodeName = requestBody.get("nodeName");
        String currentContent = requestBody.get("currentContent");
        String updatedContent = requestBody.get("updatedContent");
        synchronized (this) {
            if (!currentContent.equals(Integer.toString(getDocumentForUpdate
                System.out.println("RACE  CONDITION!!");
```

Send two requests simultaneously and review the logs for any race condition issues.





```
RACE  CONDITION!!
Redirect to node3::race condition
Request successful to: http://affinity-node-1:8080/broadcast/update-document
Request successful to: http://node1:8080/broadcast/update-document
Request successful to: http://node3:8080/broadcast/update-document
Request successful to: http://affinity-node-2:8080/broadcast/update-document
Request successful to: http://node4:8080/broadcast/update-document
Request successful to: http://node2:8080/broadcast/update-document
```