Decentralized Cluster-Based NoSQL DB System

Done By: Jafar Farwaneh

Content

Figures

1.0 Introduction

The exponential growth of data in the digital era, coupled with the need for highly available and responsive applications, has fueled the adoption of NoSQL (Not Only SQL) databases. Unlike traditional relational databases with their rigid schemas, NoSQL databases offer flexibility in data modeling, making them well-suited for handling semi-structured or rapidly evolving data. Their ability to scale horizontally across distributed systems provides advantages in performance and resilience.

Decentralized NoSQL clusters take this scalability a step further. By eliminating a central point of control or a management node, they offer the potential for enhanced fault tolerance and autonomy. However, decentralization introduces complexities that must be carefully addressed to ensure the database system's integrity and efficiency. Key challenges include:

- **Data Consistency:** In the absence of a central authority, maintaining consistent replicas of data across multiple nodes becomes paramount. Concurrent writes to the same data on different nodes can lead to conflicts that need to be resolved without compromising data integrity. Strategies such as optimistic locking or consensus protocols are often employed to address this challenge.
- **Load Balancing:** Distributing incoming requests and workloads evenly across the cluster's nodes is essential to avoid performance bottlenecks. As no single node has complete knowledge of the cluster's state, sophisticated load balancing algorithms are required.

- **Efficient Query Processing:** Finding the correct node to service a write request (node affinity) and efficiently locating data for read requests is complex in a decentralized system. Indexing strategies need to be adapted to this distributed context.

This project presents the design and implementation of a decentralized NoSQL database system in Java. The system simulates the interactions between users and nodes within a cluster environment. Crucial components include:

- **Document-Oriented Data Model:** Documents, represented in JSON format, offer flexibility in data representation and schema enforcement.
- **Custom Indexing:** Indexing mechanisms are developed to facilitate efficient retrieval of documents based on single JSON properties.
- **Decentralized Load Balancing:** Algorithms are implemented to distribute user connections across nodes without relying on a central coordinator.
- **Data Consistency:** Techniques such as optimistic locking are employed to address conflicts arising from concurrent writes and ensure data consistency across replicas.

This report delves into the system's architecture, implementation details, the results of performance testing, and an analysis of how design choices address the complexities of decentralized data management.

Additionally, a demo App that uses the database (Task Assignment App) is discussed in this report.

2.0 System Architecture and Design

2.1 System Design

- The system architecture centers around a cluster of four worker nodes responsible for data storage and processing, along with a dedicated bootstrapping node. Each node operates within its own Docker container, providing process isolation and simplified deployment.
- The worker nodes form a network to communicate and replicate data across the cluster, this is done using a Docker network after some configuration through a docker-compose file.

- This replication strategy is crucial for system reliability, guaranteeing that a complete copy of the data is available even if individual nodes fail.

- User interaction is streamlined; a user connects to a single node, and the system transparently handles node-to-node communications using RESTful APIs. Incoming requests pass through a layered architecture within each node. The authentication layer rigorously verifies user credentials for secure access. Next, the load balancing layer distributes the workload across the cluster, preventing bottlenecks. Finally, the business logic layer executes all core database operations, ensuring data integrity and consistency.

2.2 File System Structure

- As the picture shows, there are 2 main directories in the database node file system: 1- Databases 2- Users.
- The Databases directory includes a separate directory for each database.
- Every database directory consists of 3 main directories:

  1- Collections (their content)

  2- Indexes

  3- Schemas

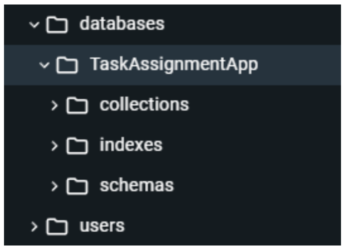


Figure 1: File System Structure

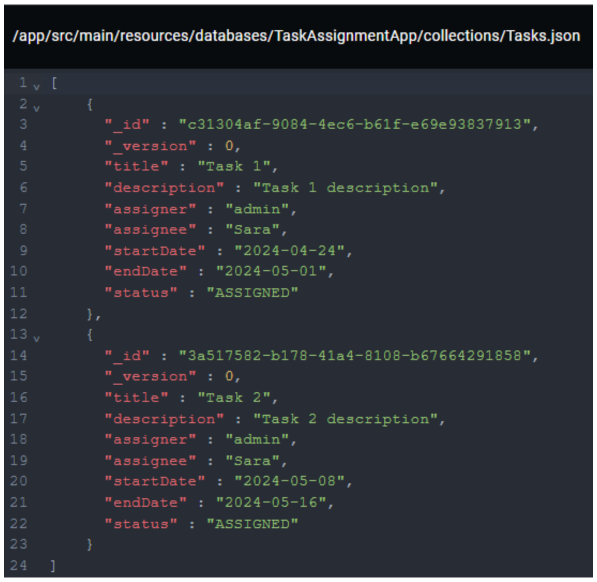
- The collections and their documents are stored as JSON arrays and JSON documents respectively.

3.0 Database Implementation

- When implementing the database, I had to choose between 2 implementations: 1- In-Memory Database 2- On-Disk Database, and due to the possibility of the database growing to contain a lot more than just tens of documents, and for the sake of persistence and non-volatility, I've chosen the **On-Disk Implementation**.
- That doesn't mean that In-Memory data structures were not used, they are used through the database, but only for lightweight data, like indexes and resources name.

3.1 Data Model & Storage

- As mentioned earlier, the documents are stored as JSON documents, and grouped inside collections, documents content varies from a collection to another as it must adhere to the collection schema, but there are 2 essential properties that all documents have:
  1- _id : resembles the unique Id of the document (UUID).
  2- _version : resembles the current version of the document, used to apply optimistic locking strategy when requesting an update (will talk about it later).
- Collections are JSON arrays that host the JSON documents, and each collection is a separate JSON file.



/app/src/main/resources/databases/TaskAssignmentApp/collections/Tasks.json

```
[
    {
      "_id" : "c31304af-9084-4ec6-b61f-e69e93837913",
      "_version" : 0,
      "title" : "Task 1",
      "description" : "Task 1 description",
      "assigner" : "admin",
      "assignee" : "Sara",
      "startDate" : "2024-04-24",
      "endDate" : "2024-05-01",
      "status" : "ASSIGNED"
    },
    {
      "_id" : "3a517582-b178-41a4-8108-b67664291858",
      "_version" : 0,
      "title" : "Task 2",
      "description" : "Task 2 description",
      "assigner" : "admin",
      "assignee" : "Sara",
      "startDate" : "2024-05-08",
      "endDate" : "2024-05-16",
      "status" : "ASSIGNED"
    }
]
```

Figure 2: Documents in a Collection

- All documents are replicated across all nodes in the same format, to ensure maximum reliability.
- Jackson library was utilized to manipulate JSON documents and collections.

3.2 Schema Handling

- A schema is essential for a collection to be created because it defines how data is organized within itself and describes the allowed structure of the documents that will be stored.

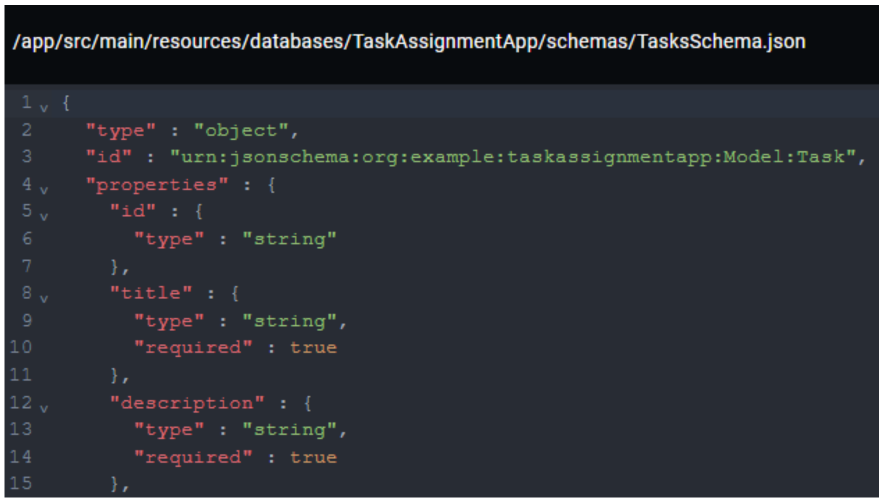
- Before a document creation operation is carried out, a validation of the document structure and properties values is performed against the collection schema.

- For that I've created the model Schema that resembles the components of a JSON schema as fields (type, properties map and required properties array), and contains a set of useful methods like:

  1- toJson (): Converts the Schema model object to a JSON schema object.

  2- fromClass (Class<?> clazz): Converts a Model class to a JSON schema object (needs related annotations in the class).

  3- of (String jsonSchema): Converts a string of a schema to a JSON schema object.



/app/src/main/resources/databases/TaskAssignmentApp/schemas/TasksSchema.json

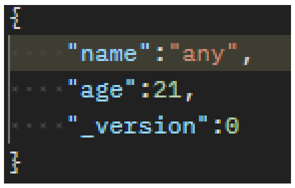
```
{
  "type" : "object",
  "id" : "urn:jsonschema:org:example:taskassignmentapp:Model:Task",
  "properties" : {
    "id" : {
      "type" : "string"
    },
    "title" : {
      "type" : "string",
      "required" : true
    },
    "description" : {
      "type" : "string",
      "required" : true
    },
```

Figure 3: Schema File

3.3 Basic CRUD Operations

1. **Create & Insert**: Databases and collections can be easily created after specifying their names, the same applies for documents after specifying their content and belonging collection and database, their indexes are also created.

2. **Read**: All types of reading operations are supported, including:
    a. Fetching all Documents/Collection/Databases.
    b. Fetching all documents with certain property value.
    c. Fetching a single property value of a document.
3. **Update**: Documents can be updated by sending a request with a body that contains a JSON document that contains the changes and the expected _version of the document which is needed for optimistic locking.



```
{
    "name":"any",
    "age":21,
    "_version":0
}
```

Figure 4: Update(PUT) Request Body

4. **Delete**: Deleting is straightforward, just specify the unique identifier of the resource and it'll be deleted with anything related to it whether it be data or indexes.

3.4 Endpoints

3.4.1 Database Controller

Base URL: /api/databases

- **createDatabase**
  - **Endpoint**: Base URL /{db_name}
  - **Method** : POST
  - **Description**: This method is used to create a new database.
  - **Params**:
    - **db_name** (Path Variable): The name of the database to create.
    - **isBroadcast** (Header, Optional, Default = false): Whether the request is broadcasted.
    - **username** (Header): The username for authentication.
    - **password** (Header): The password for authentication.

- **deleteDatabase**
  - **Endpoint**: Base URL /{db_name}
  - **Method:** DELETE
  - **Description**: This method is used to delete an existing database.
  - **Params**:
    - **db_name** (Path Variable): The name of the database to delete.
    - **isBroadcast** (Header, Optional, Default = false): Whether the request is broadcasted.
    - **username** (Header): The username for authentication.
    - **password** (Header): The password for authentication.

- **fetchExistingDatabases**
  - **Endpoint**: Base URL
  - **Method:** GET
  - **Description**: This method is used to fetch the list of existing databases.
  - **Params**:
    - **username** (Header): The username for authentication.
    - **password** (Header): The password for authentication.

3.4.2 Collection Controller

Base URL: /api/databases/{db_name}/collections

- **createCollection**
  - **Endpoint URL**: Base URL /{collection_name}
  - **Method:** POST
  - **Description**: This method is used to create a new collection in a database.
  - **Params**:
    - **db_name** (Path Variable): The name of the database.
    - **collection_name** (Path Variable): The name of the collection to create.
    - **schema** (Request Body): The schema for the collection.
    - **isBroadcast** (Header, Optional, Default = false): Whether the request is broadcasted.
    - **username** (Header): The username for authentication.
    - **password** (Header): The password for authentication.

- **deleteCollection**
  - **Endpoint URL**: Base URL /{collection_name}
  - **Method:** DELETE
  - **Description**: This method is used to delete an existing collection from a database.
  - **Params**:

    - **db_name** (Path Variable): The name of the database.
    - **collection_name** (Path Variable): The name of the collection to delete.
    - **isBroadcast** (Header, Optional, Default = false): Whether the request is broadcasted.
    - **username** (Header): The username for authentication.
    - **password** (Header): The password for authentication.

- **fetchExistingCollections**
  - **Endpoint URL**: Base URL
  - **Method:** GET
  - **Description**: This method is used to fetch the list of existing collections in a database.
  - **Params**:
    - **db_name** (Path Variable): The name of the database.
    - **username** (Header): The username for authentication.
    - **password** (Header): The password for authentication.

3.4.3 Document Controller

Base URL: /api/databases/{db_name}/collections/{collection_name}/documents

- **createDocument**
  - **Endpoint URL**: Base URL
  - **Method:** POST
  - **Description**: This method is used to create a new document in a collection.
  - **Params**:
    - **db_name** (Path Variable): The name of the database.
    - **collection_name** (Path Variable): The name of the collection.

    - **documentNode** (Request Body): The document to be created.
    - **username** (Header): The username for authentication.
    - **password** (Header): The password for authentication.
    - **isBroadcast** (Header, Optional, Default = false): Whether the request is broadcasted.

- **updateDocument**
  - **Endpoint URL**: Base URL /{doc_id}
  - **Method:** PUT
  - **Description**: This method is used to update an existing document in a collection.
  - **Params**:
    - **db_name** (Path Variable): The name of the database.
    - **collection_name** (Path Variable): The name of the collection.
    - **doc_id** (Path Variable): The ID of the document to update.
    - **updatedProperties** (Request Body): The updated properties of the document.
    - **username** (Header): The username for authentication.
    - **password** (Header): The password for authentication.
    - **isBroadcast** (Header, Optional, Default = false): Whether the request is broadcasted.

- **deleteDocument**
  - **Endpoint URL**: Base URL /{doc_id}
  - **Method:** DELETE
  - **Description**: This method is used to delete an existing document from a collection.
  - **Params**:
    - **db_name** (Path Variable): The name of the database.
    - **collection_name** (Path Variable): The name of the collection.
    - **doc_id** (Path Variable): The ID of the document to delete.

        - **username** (Header): The username for authentication.
        - **password** (Header): The password for authentication.
        - **isBroadcast** (Header, Optional, Default = false): Whether the request is broadcasted.

- **deleteCollectionDocuments**
    - **Endpoint URL**: Base URL
    - **Method:** DELETE
    - **Description**: This method is used to delete all documents from a collection.
    - **Params**:
        - **db_name** (Path Variable): The name of the database.
        - **collection_name** (Path Variable): The name of the collection.
        - **property_name** (Query Parameter, Optional): The name of the property.
        - **property_value** (Query Parameter, Optional): The value of the property.
        - **username** (Header): The username for authentication.
        - **password** (Header): The password for authentication.

- **fetchDocumentById**
    - **Endpoint URL**: Base URL /{doc_id}
    - **Method:** GET
    - **Description**: This method is used to fetch a document by its ID from a collection.
    - **Params**:
        - **db_name** (Path Variable): The name of the database.
        - **collection_name** (Path Variable): The name of the collection.
        - **doc_id** (Path Variable): The ID of the document to fetch.
        - **username** (Header): The username for authentication.
        - **password** (Header): The password for authentication.

- **fetchCollectionDocuments**
    - **Endpoint URL**: Base URL
    - **Method:** GET
    - **Description**: This method is used to fetch all documents from a collection.
    - **Params**:
        - **db_name** (Path Variable): The name of the database.
        - **collection_name** (Path Variable): The name of the collection.
        - **property_name** (Query Parameter, Optional): The name of the property.
        - **property_value** (Query Parameter, Optional): The value of the property.
        - **username** (Header): The username for authentication.
        - **password** (Header): The password for authentication.

- **readDocumentProperty**
    - **Endpoint URL**: Base URL /{doc_id}/{propertyName}
    - **Method:** GET
    - **Description**: This method is used to read a property of a document.
    - **Params**:
        - **db_name** (Path Variable): The name of the database.
        - **collection_name** (Path Variable): The name of the collection.
        - **doc_id** (Path Variable): The ID of the document.
        - **propertyName** (Path Variable): The name of the property to read.
        - **username** (Header): The username for authentication.
        - **password** (Header): The password for authentication.

3.5 Indexing

- Indexing is a technique used to locate and quickly access data in databases, it increases the efficiency of performing various CRUD operations.
- I used simple text files to store some key data in pairs.
- All indexes are loaded from disk on startup.
- There are 3 types of indexes in my database:
    - **Collection Index**: An index that stores a key value pair of a document's ID and its position in a collection, to optimize fetching documents by Id.
    - **Property Index:** An index that stores a key value pair of a document's ID and its property value, to optimize reading a single property of a document.
    - **Inverted Property Index:** An index that groups together the documents that share the same property value, to optimize fetching a list of documents that share a property value.
- All these indexes used a data structure called B+ Tree, which I'll talk about in the Data Structures section, and are managed by a manager class called IndexingManager.

4.0 Data Structures

4.1 B+ Tree

- I chose to implement B+ tree structures for indexing documents and specific properties within my NoSQL database.
- B+ trees are a valuable choice for disk-based indexes due to their efficient search, insertion, and deletion operations.
- Their structure is optimized for block-based storage, minimizing disk I/O.
- Additionally, B+ trees guarantee a balanced structure, providing predictable query performance, which is crucial in a decentralized cluster environment where query distribution is less controllable.

4.2 HashMap

- HashMaps were used in many classes in my implementation to hold some lightweight important data like users and their nodes, and the index objects.
- HashMaps help in retrieving these important values in an efficient way.

4.3 List

- Lists were used in read methods to return a group of documents or users.

5.0 Multithreading and locks

In a database environment, it’s essential that multiple users or processes can interact with the system simultaneously without compromising data or causing errors. This project utilizes robust multithreading and locking techniques to provide a reliable and responsive user experience, even under heavy usage.

5.1 Spring Built-in Multithreading

- Spring MVC, the web framework built into Spring, naturally supports multithreading within controllers.
- Each incoming HTTP request to my controller is handled by a separate thread.
- This means multiple user requests can be processed concurrently without my code needing to explicitly create and manage threads.
- Spring handles the underlying thread pool and ensures that my controller methods are thread-safe, allowing me to focus on implementing the request-handling logic itself. This built-in multithreading capability is what enables my Spring-based application to serve multiple users simultaneously and efficiently.

5.1 Resources Locking

- For any operation other than reading, the targeted resource whether it’s database, collection or document is locked using a shared ReentrantLock object.
- All locks are created, given, and deleted by the utility class LocksManager.

6.0 Data Consistency and Replication

- In my decentralized NoSQL database project, data consistency across the cluster is a core concern. To achieve this, I've implemented a replication strategy where each database node broadcasts any modification to its data to all other nodes in the cluster.
- When a node receives a broadcast update, it applies the corresponding changes to its local replica.
- While this broadcast approach introduces a degree of network overhead, it aims to maintain strong consistency by ensuring that all nodes eventually converge to the same data state.
- This is particularly important in a decentralized system where users might connect to different nodes for subsequent reads.
- The Broadcaster class is responsible for this.

7.0 Node Hashing and Load Balancing

Load balancing is very important in a distributed environment, in my database, load balancing was performed for things:

1- Users:

- The bootstrapper node in my system plays a critical role in initial load balancing.
- When a new user connects, the bootstrapper employs a simple round-robin algorithm to assign the user to a database node within the cluster.
- This algorithm iterates through the available nodes in a circular fashion, distributing new connections evenly.
- While round-robin doesn't consider individual node workloads, it provides a lightweight and effective

mechanism for initial user distribution in a decentralized environment.

2- Documents Affinity:

- To achieve load balancing and manage document affinity (ownership and write rights), I've employed consistent hashing.
- This technique maps documents to specific nodes based on a hash function, ensuring that write requests for a given document are always routed to the correct node.
- By carefully specifying the number of replicas for each node within the consistent hash ring, I've achieved a remarkably even distribution of documents across the cluster.
- While my cluster currently has a fixed size, incorporating consistent hashing creates a robust foundation for future scalability, as this approach minimizes the disruption caused by adding or removing nodes, so it inherently adapts to changes in the cluster's topology without requiring centralized rebalancing.

8.0 Security Issues

Even though I used Spring Boot, I didn't use any of Spring Security utilities as I want to rely a lot on a framework, so I implemented a simple username-password authentication that is performed as soon as a request is received.

Hashing passwords was essential for privacy and security, so no password is stored in its original form.

I implemented a class JwtUtil to establish token-based authentication, but didn't have time to integrate it, so it will be a future work.

9.0 Code Testing

Testing during development is crucial and saves a lot of time and ensures the accuracy of the application, through the development cycle I've tested my code in 3 different ways:

1- **JUnit tests**: when developing the lower-level classes that deal with the file system, indexes, and schemas I've used unit testing to ensure maximum accuracy and reliability.

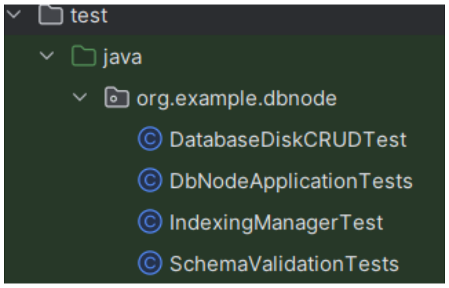


Figure 5: JUnit Tests

2- **Postman**: when my controllers were ready, I created an API collection that contains various CRUD requests and tested my controllers using it.

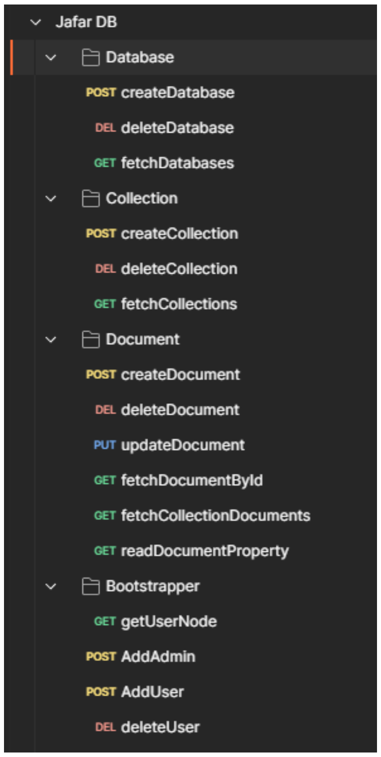
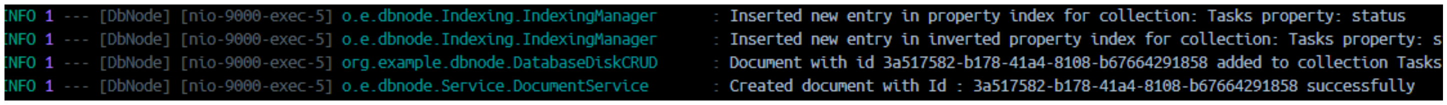


Figure 6: Postman Api Collection

3- **Demo App:** to demonstrate the usability of my database, I've created a Spring Boot app called **Task Assignment App** that utilized my database by sending requests to its controllers and nodes.

Note: Even though It's not a test itself, but using logs across my services and controllers allowed for very fast and easy debugging, I used Log4j2 for that .



```
INFO 1 --- [DbNode] [nio-9000-exec-5] o.e.dbnode.Indexing.IndexingManager      : Inserted new entry in property index for collection: Tasks property: status
INFO 1 --- [DbNode] [nio-9000-exec-5] o.e.dbnode.Indexing.IndexingManager      : Inserted new entry in inverted property index for collection: Tasks property: s
INFO 1 --- [DbNode] [nio-9000-exec-5] org.example.dbnode.DatabaseDiskCRUD      : Document with id 3a517582-b178-41a4-8108-b67664291858 added to collection Tasks
INFO 1 --- [DbNode] [nio-9000-exec-5] o.e.dbnode.Service.DocumentService       : Created document with Id : 3a517582-b178-41a4-8108-b67664291858 successfully
```

Figure 7: Log4j2 Logs

10.0 Clean Code

- **DRY Principle:**
    - To promote the DRY (Don't Repeat Yourself) principle, I observed a pattern of repetitive data-casting code throughout my project. Addressing this, I created a centralized DataTypeCaster class. This class encapsulates the logic for casting data between different types, eliminating redundancy and reducing the possibility of errors. Whenever I need to perform data casting, I now leverage the DataTypeCaster, resulting in a cleaner codebase and improved maintainability. Any future changes in casting logic can be easily managed within this single class.

- **Try-with-resources:**
    - The project extensively employs the try-with-resources construct to ensure the safe and efficient management of resources like files and streams. This guarantees that resources are automatically closed after use, preventing resource leaks, and promoting a cleaner, more reliable codebase.

- **Meaningful Names:**
    - I try my best to choose variables, methods, and classes names that clearly convey their purpose. It might take a bit more thought upfront, but it makes the code much easier to understand in the long run.

- **Comments:**
    - I use comments sparingly but effectively. Whenever there's a section of code with complex logic or a design decision that might not be obvious, I add a comment to explain the reasoning behind it.

- **Avoiding Returning Null:**
    - I've learned that returning null can be a recipe for unexpected errors. Where possible, I use things like Optional or throw exceptions to indicate missing values or error conditions. This makes my code more robust and easier to debug.

11.0 Effective Java Items

- **item3**: Enforce the singleton property with a private constructor or an Enum type.
- **Item 13**: Minimize the accessibility of classes and members, my code completely adhered to this.
- **Item 24**: Eliminate unchecked warnings: I've used '@SuppressWarnings' annotation on the smallest scope possible, to not mask other, critical warnings.
- **Item 27**: Favor generic methods: I've used generic methods in B+ Tree and InvertedPropertyIndex.
- **Item 15**: Minimize the accessibility of classes and members I wrote class members as private, and they can only be accessed outside the class through getters.
- **Item28**: Prefer lists to arrays, this is very clear in my code.
- **Item 40**: Consistently use the Override annotation, I used this notation in all my overridden methods.

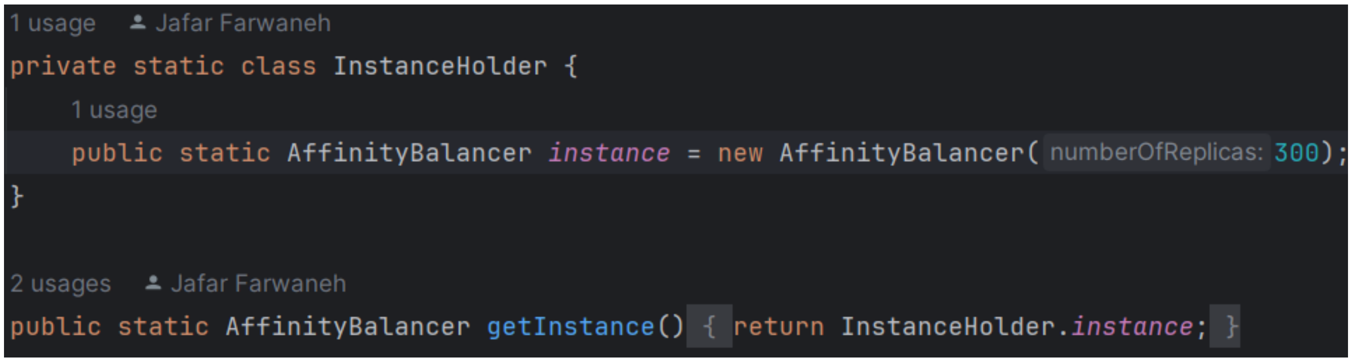
- **Item 58**: Prefer for-each loop to traditional for loops, I’ve Used the for each in many parts of my code.

12.0 SOLID principles

- **Single Responsibility Principle (SRP):** I try to keep my classes and modules focused on specific tasks. For example, I have separate classes for handling document operations, managing indexes, and coordinating node communication.
- **Open-Closed Principle (OCP):** I use interfaces and abstract classes for key components of my system, such as data storage or indexing. This means I can add new implementations in the future without needing to overhaul my entire codebase.
- **Liskov Substitution Principle (LSP):** When designing interfaces and inheritance hierarchies, I aim to make sure subclasses can be used interchangeably with their base classes.
- **Interface Segregation Principle (ISP):** I avoid creating massive interfaces that do too many things. Instead, I break them down into smaller, more focused interfaces, which makes my code more adaptable to changes.
- **Dependency Inversion Principle (DIP):** I try to make my modules depend on abstractions rather than concrete implementations. By using interfaces for things like database services, I can change underlying logic more easily down the road.

13.0 Design Patterns

- **Singleton:**
    - A lot of utility classes were singletons in my project, some of them were handled by spring because they're considered Components , and others were normal classes ,so I used a special method to make these classes singleton while avoiding concurrency issues (without relying on spring), which is the **Instance Holder Class techniqu**e, a private static class holds the instance of the targeted class, and whenever the inner class is loaded, the instance is created, this makes use of the thread-safe nature of class loading in java.



```
1 usage   Jafar Farwaneh
private static class InstanceHolder {
    1 usage
    public static AffinityBalancer instance = new AffinityBalancer( numberOfReplicas: 300);
}

2 usages   Jafar Farwaneh
public static AffinityBalancer getInstance() { return InstanceHolder.instance; }
```

Figure 8: Singelton with Instance Holder

- **Builder:**
    - The Builder design pattern was used in the **Request** class, as it has a lot of parameters and fields, which allowed for smoother creation of requests.

14.0 DevOps Practices

- **Version Control:**
  - Used Git and GitHub to version control my code base.
- **Containerization:**
  - Every node was in a separate container, the same goes for the bootstrapper and the demo app, which allows for consistent deployment across different environments.
- **Continuous Logging:**
  - Every layer in my implementation has a good number of logs which allows for easy and early error handling.

15.0 Demo App

- I developed a Spring Boot Application that is centered about admins assigning tasks to users, and users accessing their respective tasks and modifying their status.
- Below are some screenshots:

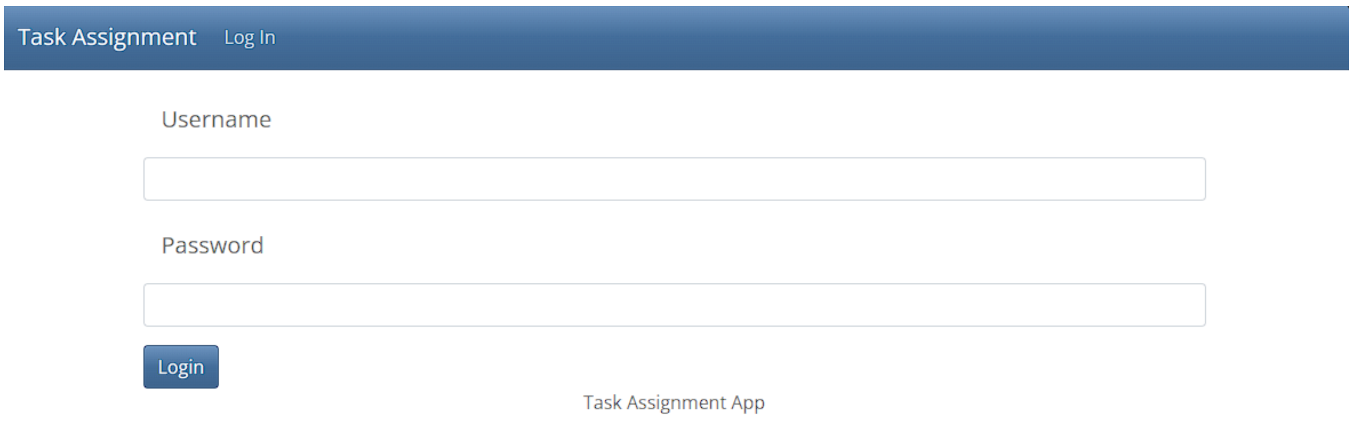
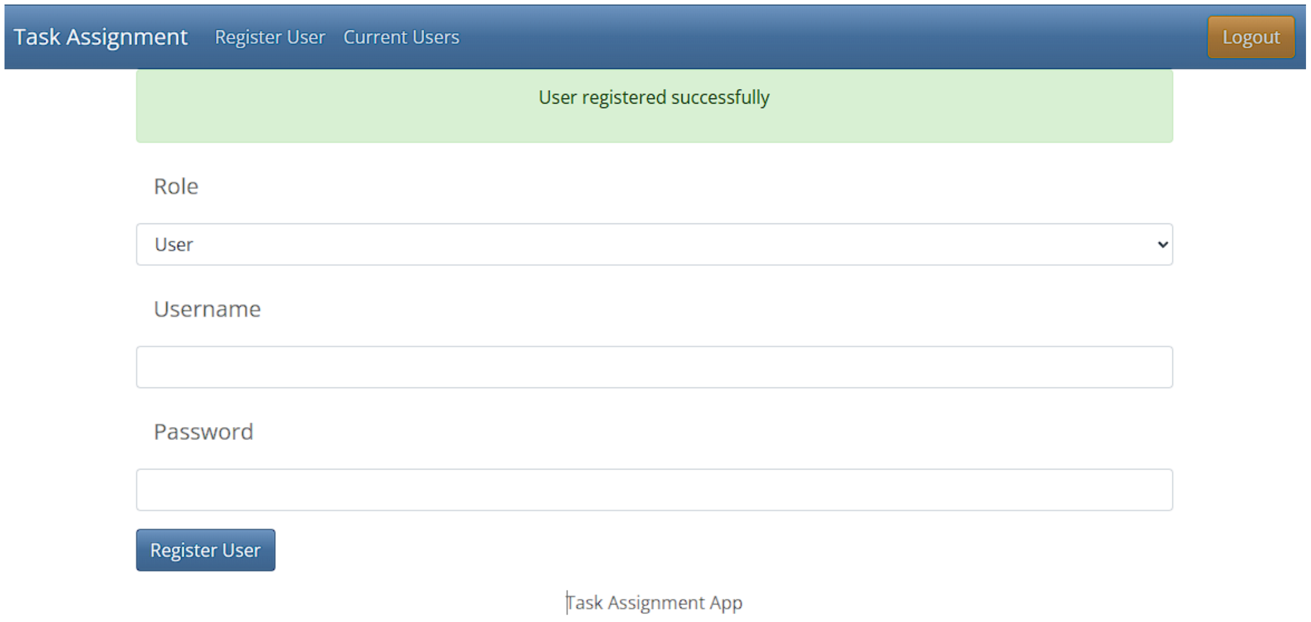


Figure 9: Task Assignment App: Login Page



User registered successfully

Role

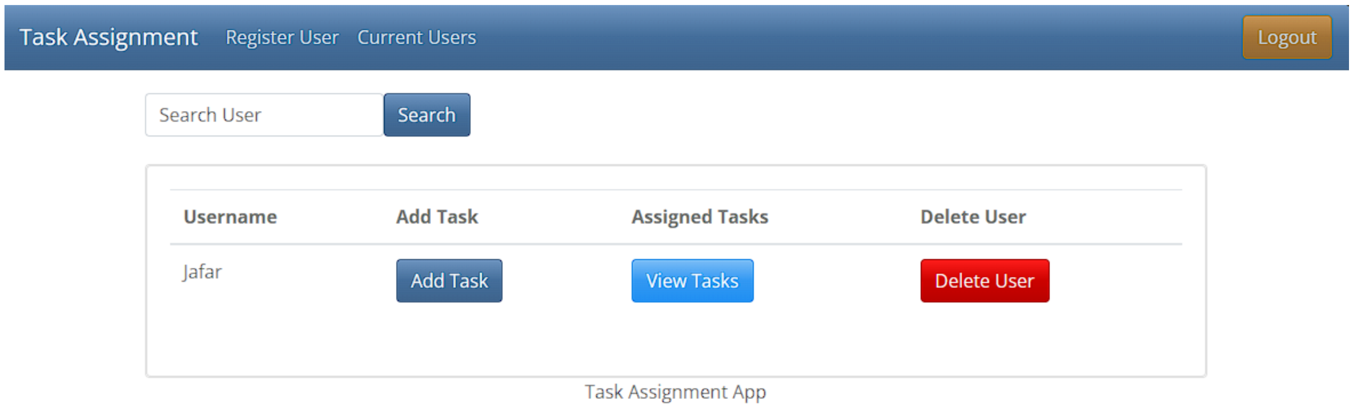
User

Username

Password

Task Assignment App

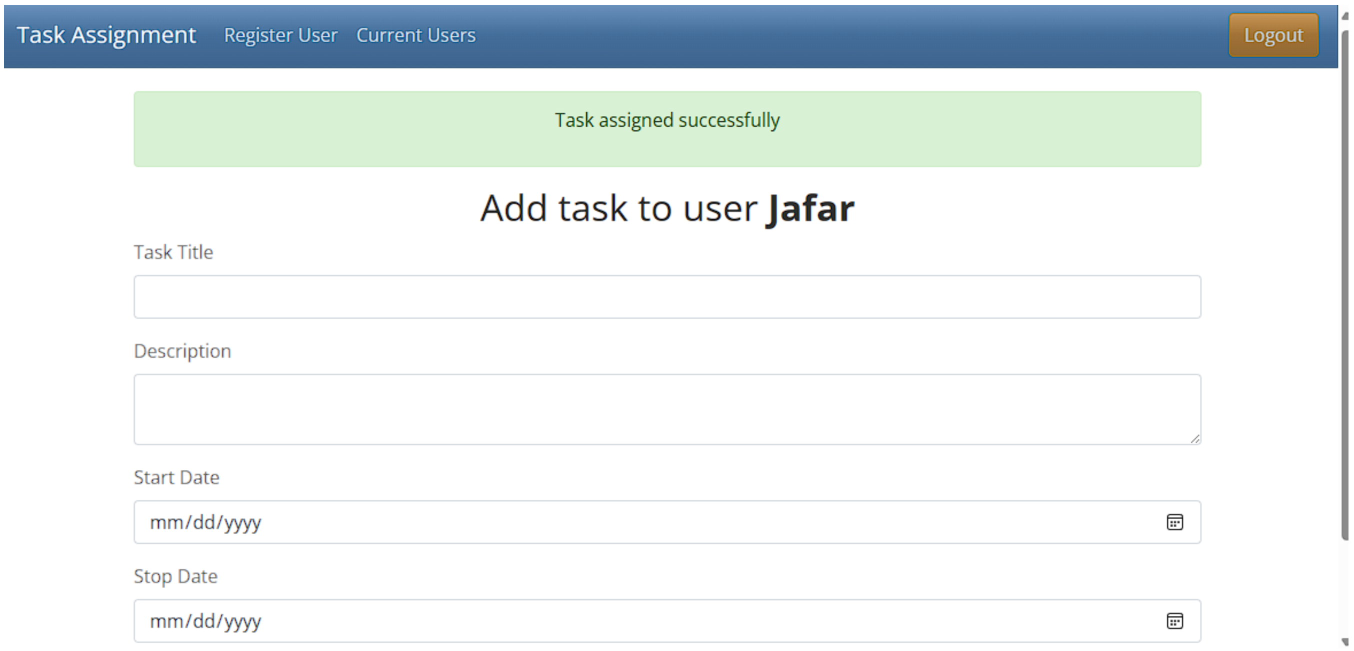
Figure 10: Task Assignment App: User Registration Page



Username	Add Task	Assigned Tasks	Delete User
Jafar	Add Task	View Tasks	Delete User

Task Assignment App

Figure 11: Task Assignment App: Users List

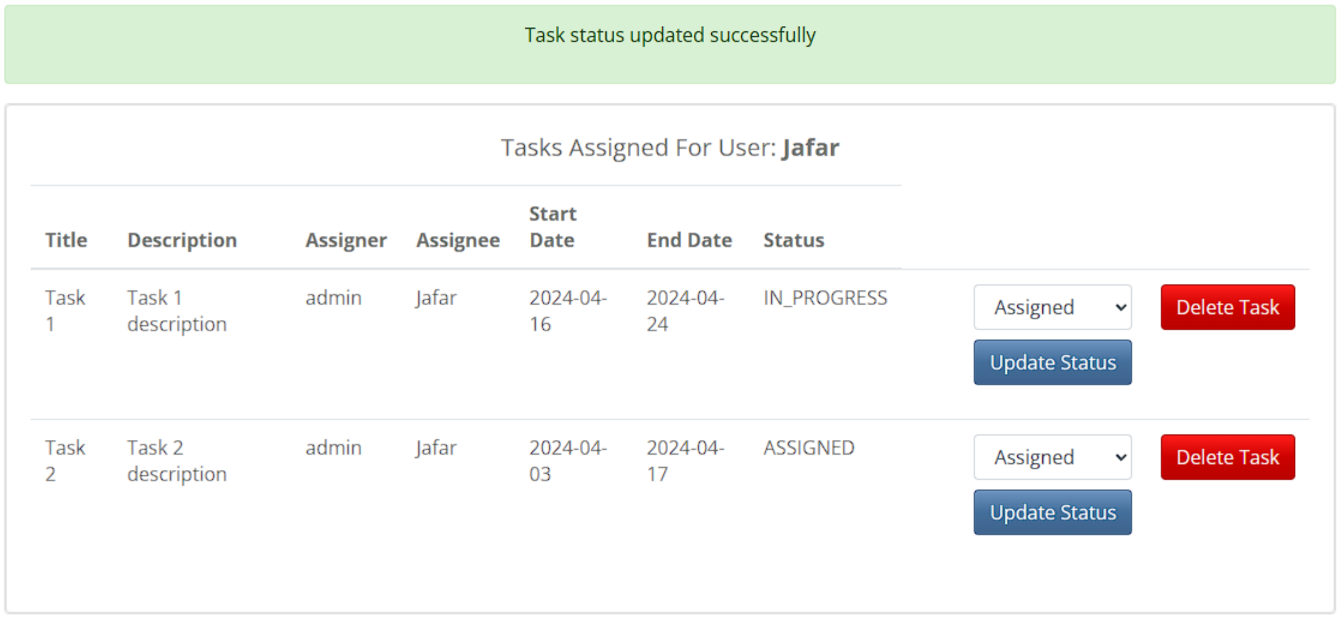


Task assigned successfully

Add task to user **Jafar**

Task Title	
Description	
Start Date	mm/dd/yyyy
Stop Date	mm/dd/yyyy

Figure 12: Task Assignment App: Task Assignment Page

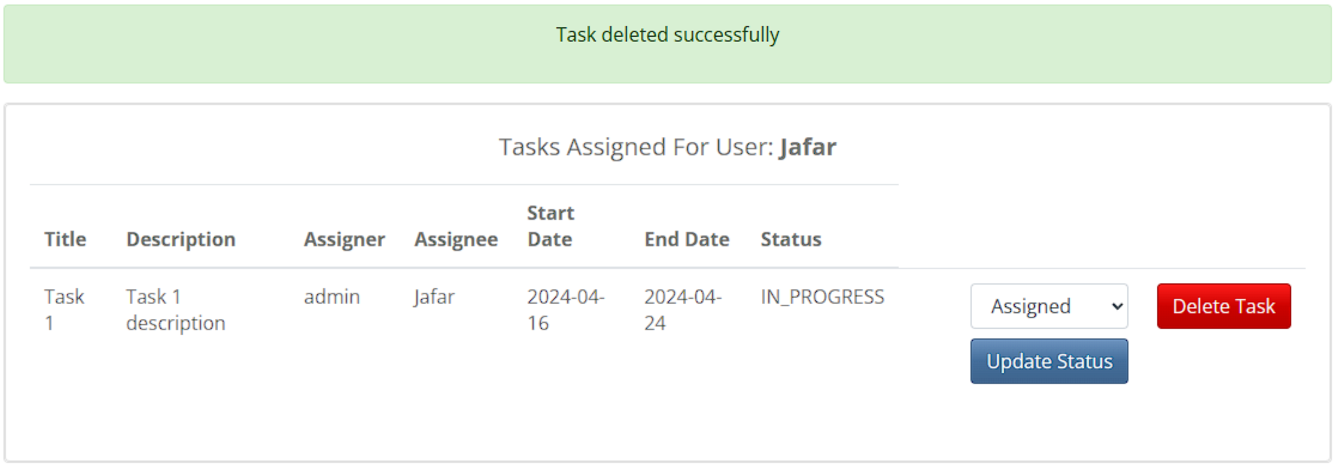


Task status updated successfully

Tasks Assigned For User: **Jafar**

Title	Description	Assigner	Assignee	Start Date	End Date	Status		
Task 1	Task 1 description	admin	Jafar	2024-04-16	2024-04-24	IN_PROGRESS	Assigned / Update Status	Delete Task
Task 2	Task 2 description	admin	Jafar	2024-04-03	2024-04-17	ASSIGNED	Assigned / Update Status	Delete Task

Figure 13: Task Assignment App: Task Status Update


Task deleted successfully

Tasks Assigned For User: **Jafar**

Title	Description	Assigner	Assignee	Start Date	End Date	Status		
Task 1	Task 1 description	admin	Jafar	2024-04-16	2024-04-24	IN_PROGRESS	Assigned	Delete Task

Figure 14: Task Assignment App: Task Deletion

The End