Atypon

Decentralized Cluster-Based NoSQL DB System

By Rayyan Al-hourani

A-Database implementation

Bootstrapping step

In the Bootstrapping Node, users can either log in or sign up to access one of the workker nodes. When a user logs in, the Bootstrapping Node verifies their credentials and returns the URL of the node along with a token that the user needs to send to the server for authorization. Here, we have an "AuthController" with three endpoints: "/login," "/signup," and "/signout."

1. **"/login" Endpoint:**
    - Receives user information.
    - Forwards user information to the **login()** function in the **AuthService class.**
    - If **login()** returns null, sends an error message to the user indicating incorrect username or password.
    - If successful, send the user information to the **assignUserToNode()** function in **NodeService class**, which returns a string containing the token and the node URL.
2. **"/signup" Endpoint:**
    - Sends user information to the **signup()** function in the **AuthService class.**
    - The function checks if the user signup was successful and responds accordingly.
3. **"/signout" Endpoint:**
    - Sends user information to the **clearNode()** function in the **NodeService** class to allow the user to sign out, enabling load balancing for other users.

Additionally, we have three models:

1. Node Class:

- The Node class represents individual nodes within your system. Each node has the following attributes:
    - ➢ **nodeName**: A name or identifier for the node.
    - ➢ **nodeNumber**: A unique number or ID associated with the node.
    - ➢ **nodePort**: The port number on which the node operates in docker network.

2. User Class:

- The User class represents user accounts in your system. Each user has the following attributes:
    - ➢ **username:** The username chosen by the user for identification.
    - ➢ **password:** The password associated with the user's account for authentication.
    - ➢ **role:** The role or access level associated with the user, which might determine their permissions within the system.

3. UserToNode Class:

- The **UserToNode class** is responsible for managing the mapping of users to nodes, which helps distribute users evenly among available nodes to achieve load balancing in your system. Here are the key methods of the **UserToNode class:**

A. addUserToNode(User user)

  - This method is called when a user needs to be assigned to a node. It performs the following steps:
    - ➢ Obtains the user object as input.
    - ➢ Selects the next available node (possibly from a queue or a load balancing algorithm).
    - ➢ Associates the user's username with the selected nodeURL.
    - ➢ Returns the assigned node.
- The primary purpose of this method is to maintain a balanced distribution of users across nodes while keeping track of the user-node assignments.

B. removeUserFromNode(User user)

  - This method is called when a user logs out or disconnects from the system. It performs the following steps:
    - ➢ Obtains the user object as input.
    - ➢ Removes the user's association with the node (possibly by unassigning the username from the nodeURL).
    - ➢ Performs any necessary load balancing actions (e.g., moving the node to the front of a queue to prioritize it for the next user assignment).
  - The purpose of this method is to ensure that when a user leaves the system, their resources are released, and the node they were using can be allocated to another user to maintain system efficiency.

Synchronization:

Both **addUserToNode(User user) and removeUserFromNode(User user)** methods are typically synchronized to ensure that concurrent requests for user assignment and removal do not result in conflicts or data inconsistencies. Synchronization ensures that only one thread can access and modify the user-node mapping at a time, preventing race conditions.

In summary, the **UserToNode class** plays a critical role in your system by managing the assignment of users to nodes for load balancing purposes. It helps maintain a balanced distribution of users, ensuring efficient resource utilization across nodes.

And, we have four service classes

1. **AuthService Class:**
    - The AuthService class is responsible for user authentication and registration. It contains two primary methods:
        - ***login(User user):*** This method is responsible for user authentication. It checks if the user's information is already cached. If not, it reads the user data from a file and stores it in a map where usernames are keys, and user objects are values. Then, it verifies the username and password. If the user is valid, it returns the user object; otherwise, it returns null.

        - ***signup(User user):*** When a user wants to sign up, this method first checks if the username already exists. If the username is available, it adds the user to the map containing user data and stores this updated map as a JSON file for later use.

2. **NodeService Class**:
    - The NodeService class manages the assignment of users to nodes and the handling of user sign-outs. It consists of two main methods:
        - ***assignUserToNode(User user):*** When a user requests access, this method verifies if the user is already logged in and checks if their username is associated with a node URL and token in a mapping. If not, it calls the **addUserToNode()** method to assign the user to a node, generates a token using the **createToken()** method from the **TokenService class**, and sends this token to the user for validation.

        - ***clearNode(User user):*** When a user logs out or disconnects, this method calls **removeUserFromNode()** to remove the user from their assigned node, allowing other users to take their place. It also removes the user's token from the map that contains usernames and tokens. Both methods are synchronized to ensure proper handling of concurrent requests.

3. **TokenService Class:**
    - The TokenService class is responsible for generating tokens for user authentication. It contains a single method:
        - ***createToken(User user):*** This method generates a token containing user information and a timestamp to ensure its validity and uniqueness.
4. **FileService Class:**
    - The FileService class handles reading and writing data to files, allowing for persistent storage of user data and system information. It contains two methods:
        - ***writeFile(String path, String data):*** This method is used to write data to a specified file path. It allows the system to store user data and other information persistently.
        - ***readFile(String path):*** This method reads the content of a file specified by the provided file path, enabling the system to retrieve user data and other stored information as needed.

Node implementation

here's the information about the controllers, endpoints, and models in the node

1. AuthController:

a. **"/signUser" Endpoint:**

- ➢ This endpoint is responsible for allowing users to access the node after login. It receives a user token in the request header and sends it to the **setToken**() method in the **AssignedUser class**. This method essentially marks the user with the provided token as authorized to use the node.

b. **"/logout" Endpoint:**

- ➢ The purpose of this endpoint is to handle user logouts. It takes the token from the request header and sends it to the **clearNode**() method in the **AssignedUser class.** This method informs the Bootstrapping Node that the user has signed out, making it possible for other users to use the node for load balancing.

2. DBController:

a. **"/database/create/{databaseName}" Endpoint:**

- ➢ This endpoint is used to create a new database. It receives the desired database name as a URL parameter. It first checks the validity of the user's token and whether the query is broadcasted. If the token is valid, it creates the specified database using the **createDatabase()** method in the **DBService class.** After successful creation, it broadcasts a message to all nodes to inform them about the new database.

b. **"/database/delete/{databaseName}" Endpoint:**

- ➢ This endpoint is responsible for deleting a database. It takes the database name as a URL parameter and checks if the user is an admin. If the user has admin privileges, it proceeds to delete the specified database using the **deleteDatabase()** method in the **DBService class.**

c. **"/databases" Endpoint:**

- ➢ This endpoint returns an array containing the names of all databases. However, it first checks if the user is authorized to access this endpoint, ensuring that only authorized users can see the database names.

3. DocumentController:

a. **"/database/create/{databaseName}/{collectionName}/{docName}" Endpoint:**
   - This endpoint allows users to create a new document within a specified database and collection. It receives the database name, collection name, and document name as URL parameters. It checks the validity of the user's token. If all conditions are met, it creates the document, broadcasts the URL for the new document, and returns a response to the user.

b. **"/database/delete/document/{id}" Endpoint:**
   - For deleting a document, this endpoint takes the document ID as a URL parameter. It checks if the user is an admin before proceeding with the document deletion process. And it's the same process in controller in deleting database.

c. **"/database/get/{id}" Endpoint:**
   - This endpoint retrieves a document by its ID. Before performing this action, it verifies the validity of the user's token.

d. **"/database/edit/{id}" Endpoint:**
   - For editing a document, this endpoint receives a JSON file in the request body. It checks user authorization and verifies if the current node is the affinity node for the document. If the current node is not the affinity node, it redirects the request to the affinity node for editing , else that it edits the document and broadcasts the modified file.

e. **"/database/affinity" Endpoint:**
   - This endpoint stores a file containing document IDs and their corresponding affinity node URLs that are sent in request body.

f. **"/database/docs" Endpoint:**
   - Like the previous endpoint, this one stores a file containing information about all documents in that sent in request body.

g. **"/affinity/database/edit/{id}" Endpoint:**
   - Used by the affinity node, this endpoint receives an edited document to from affinity node and stores the document.

h. **"/redirect/{id}" Endpoint:**
   - This endpoint facilitates editing of documents after a non-affinity node redirects the edited document to the affinity node. The affinity node performs the edit and broadcasts the updated document.

i. **"/documents" Endpoint:**
   - This endpoint returns a file that contains information about all documents in the system.

Models:

1. User Class:

- Contains user information, including the username, password, and role. This class represents individual users in the system.

2. Token Class:

- Represents user tokens used for authentication. It includes attributes such as user information, the token string, creation time, and expiration time.

3. Database Class:

- Represents databases in the system. It stores database names and unique IDs.

4. Document Class:

- Represents individual documents within databases. It includes attributes like document ID, name, database name, collection name, and version.

5. AssignedUser Class:

- **The AssignedUser class** plays a crucial role in managing user tokens and their access to the node. It is responsible for verifying user tokens, setting user tokens, and performing actions when a user logs out.Below are the key methods and functionalities of the **AssignedUser class:**

  A. setToken(String token)

  1. Input: User token.
  2. Functionality: This synchronized method is used to set a user token, making the user associated with that token authorized to use the node.
     - When a user logs in, the Bootstrapping Node typically sends the user's token to this method.
     - The method extracts user information from the token, sets the token's expiration time, and associates the token with the user.
     - The synchronization ensures that only one thread at a time can set a token, preventing conflicts when multiple users log in simultaneously.

  B. isTokenValid(String token)

  1. Input: User token.
  2. Output: Boolean (true if the token is valid, false otherwise).
  3. Functionality: Checks if a given token is still valid based on its expiration time.
     - This method verifies whether the token's expiration time is in the future or has already passed.
     - If the token is still valid, it returns true; otherwise, it returns false.

C. clearNode(String token)

1. Input: User token.
2. Functionality: Notifies the Bootstrapping Node that a user has logged out or disconnected.
   - When a user logs out, the Bootstrapping Node sends the user's token to this method.
   - The method uses the token to identify the user and then informs the Bootstrapping Node that the user has signed out.
   - This action allows the Bootstrapping Node to release the resources associated with that user and potentially assign the node to another user for load balancing.

D. isAdmin(String token)

1. Input: User token.
2. Output: Boolean (true if the user is an admin, false otherwise).
3. Functionality: Checks if the user associated with the token has admin privileges.
   - This method typically involves inspecting the user's role or permissions stored in the token.
   - If the user has admin privileges, it returns true; otherwise, it returns false.

- The **AssignedUser class** is critical for managing user sessions, ensuring token validity, and handling user logouts. It helps maintain the security and proper functioning of your system by controlling user access and privileges. The synchronization of methods ensures thread-safe token management.

Service Classes

DBService Class:

- The DBService class manages the creation and deletion of databases in the node. It includes synchronized methods for these operations, ensuring data integrity and preventing conflicts during concurrent access. Here are the key methods and their functionalities:

A. **createDatabase(String databaseName):**
   1. Input: Database name.
   2. Functionality: Creates a new database with the given name if it doesn't already exist.
      - Checks if the database already exists and returns an error if it does.
      - If the database doesn't exist, it creates a folder with the specified database name using the **createFolder()** method in the **FileService class.**
      - Returns an error if any issues occur during the creation.

B. **deleteDatabase(databaseName):**
   1. Input: Database name.
   2. Functionality: Deletes the specified database, including its associated documents and collections.
      - Checks if the database exists and returns an error message if it doesn't.
      - Deletes all documents that belong to this database, removing their information from the documents File that contains all documents details.
      - Deletes the database and its collections using the **deleteFolder()** method in the **FileService class.**
      - Returns a message indicating successful deletion.

C. **getDatabases():**
   1. Functionality: Retrieves the names of all databases in the system.
      - Gather the database names by examining the folder that contains all databases.
      - Returns the list of database names as a string.

DocumentService Class:

- The DocumentService class handles various operations related to documents within your system. It includes methods for adding, deleting, editing, and retrieving documents. Here are the key methods and their functionalities:

A. addDocument(*String* databaseName, *String* collectionName, *String* docName):

1. Input: Database name, collection name, document name.
2. Functionality: Adds a new document to the specified database and collection.
   - Checks if the Documents file, which contains information about all documents, is cached. If not, it reads the file and put the json file in map.
   - Verifies the existence of the specified database. Returns an error message if the database doesn't exist.
   - Checks if the collection exists. If not, it creates a folder with the specified collection name using the **createFolder() method** in **the FileService class.**
   - Checks if the document already exists and returns an error message if it does.
   - Generates a unique ID for the document from DocCounter file, sets its version to 0, and stores it.
   - Adds the document's affinity using the **addDocumentAffinity()** method in the **Affinity class.**
   - Updates the document counter, stores it, and returns a success message.

B. deleteDocument(int id):

1. Input: Document ID.
2. Functionality: Deletes a document by its ID.
   - Checks if the Documents file is cached and if the document ID exists.
   - Obtains the JSON file path for the document using the **getDocumentPath method**.
   - Deletes the JSON file, removes its affinity, and updates the Documents file.
   - Returns a message confirming the document deletion.

C. getDocument(int id):

1. Input: Document ID.
2. Output: Document information as a string.
3. Functionality: Retrieves a document by its ID.
   - Checks if the Documents file is cached and if the document ID exists.
   - Determines whether the JSON file is cached. If not, it reads the JSON file and caches it.
   - Returns the document's information and data as a string.

D. editDocument(int id, *String* editedDocument):

1. Input: Document ID, edited document.
2. Functionality: Edits a document by its ID.
   - Checks if the Documents file is cached and if the document ID exists.
   - Compares the version of the document with the version of the edited document to implement optimistic locking.
   - If no errors are detected, increases the document's version, updates the JSON file, and returns a message confirming the modification.

E. getDocumentsFile():

1. Functionality: Reads the file containing information about all documents and stores it in a map.

F. storeDocumentsFile():

1. Functionality: Converts the documents map to a string and stores it.

G. getDocumentPath(int id):

1. Input: Document ID.
2. Output: Document JSON file path.
3. Functionality: Retrieves the path of a document by its ID.

H. checkVersion(*String* editedDocument,int version):

1. Input: Edited document, expected version.
2. Output: Edited data or an error message.
3. Functionality: Checks the version of an edited document to ensure optimistic locking. Returns the edited data if the versions match, or an error message if they differ.

I. deleteByDatabaseName(*String* databaseName):

1. Input: Database name.
2. Functionality: Deletes all documents associated with a specified database.

J. storeCounterFile():

1. Functionality: Stores the counter of documents.

K. readCounterFile():

1. Output: Number of documents in the counter file.

L. getDocuments():

1. Functionality: Returns the file containing information about all documents.

FileService Class:

- The FileService class provides methods for handling file operations, such as reading and writing files, creating, and deleting folders, and managing JSON files. Here are the key methods and their functionalities:

1. **writeFile(*String* pathOfFile, *String* data):**
   a. Input: File path, data to write.
   b. Functionality: Writes data to a specified file.
2. **readFile(*String* path):**
   a. Input: File path.
   b. Output: File content as a string.
   c. Functionality: Reads the content of a specified file and returns it as a string.
3. **deleteFolder(*String* path):**
   a. Input: Folder path.
   b. Functionality: Deletes a specified folder.
4. **createFolder(*String* path):**
   a. Input: Folder path.
   b. Functionality: Creates a folder at the specified path.
5. **createDocument(*String* path):**
   a. Input: Document path.
   b. Functionality: Creates a JSON document file at the specified path.
6. **deleteDocument(*String* path):**
   a. Input: Document path.
   b. Functionality: Deletes a JSON document file at the specified path.

Affinity Class:

- The Affinity class manages the mapping between node IDs and document IDs, facilitating document affinity control. It includes methods for checking document affinity and updating the affinity information.

1. **checkDocumentAffinity(int docID):**
    a. Input: Document ID.
    b. Output: Boolean (true if the current node is the affinity node for the document, false otherwise).
    c. Functionality: Checks if the current node is the affinity node for a specified document.
2. **addDocumentAffinity(int docID):**
    a. Input: Document ID.
    b. Functionality: Adds the document's affinity information to the system.
3. **deleteDocumentAffinity(int docID):**
    a. Input: Document ID.
    b. Functionality: Deletes the document's affinity information from the system.
4. **storeAffinityFile():**
    a. Functionality: Converts the affinity map to a string and stores it in a file, broadcasting the affinity information to other nodes.
5. **getDocumentAffinity(int docID):**
    a. Input: Document ID.
    b. Output: Affinity node information for the document.
    c. Functionality: Retrieves the affinity information for a specified document.

Broadcast Class:

- The Broadcast class is responsible for broadcasting changes made on the current node to other nodes in the system. It includes methods for broadcasting URLs, affinity files, document files, edited documents, and redirecting edited documents to their affinity nodes.

1. **broadcastingURL(String url):**
   a. Input: URL to broadcast.
   b. Functionality: Broadcasts a URL to all nodes in the network.
2. **broadcastEdit(String editedDocument):**
   a. Input: Edited document data as string.
   b. Functionality: Broadcasts an edited document to all nodes, typically used by the affinity node.
3. **RedirectToAffinity(int docID, String file):**
   a. Input: Document ID,
   b. Functionality: Redirects an edited document to its affinity node.

The Broadcast class utilizes multithreading to efficiently send broadcasts to all nodes simultaneously, ensuring synchronization and data consistency across the system.

The data structures used

In our project, we employ several key data structures for efficient operations. These data structures are primarily hashmaps, deques, and arrays. Let's break down their usage across various classes:

Bootstrapping Node:

- Within our bootstrapping node, we utilize several key data structures for efficient operations across various classes:

1. **UserNode Class:**
   a. **userToNode hashmap**: This hashmap, named userToNode, maps a username to the corresponding user node. It allows for quick retrieval of user data when needed.
   b. **availableNodes deque :** In the UserNode class, we use a deque named availableNodes to manage the assignment of nodes to users. When a user logs out, their node is placed at the front of the deque for load balancing.
2. **AuthService Class:**
   a. **cachedUser hashmap** : Within the AuthService class, the cachedUser hashmap is employed. It associates usernames with user objects, eliminating the need to repeatedly read user data from a file.
3. **TokenService Class:**
   a. **Hashmap for token management** : In the TokenService class, we use a hashmap to store and convert data to strings for transmission to clients and nodes.
4. **NodeService Class:**
   a. **usersToken hashmap**: The NodeService class employs a usersToken hashmap, linking usernames to tokens. This hashmap facilitates direct access to a user's token when required.

Worker Node:

1. **DocumentService Class:**
   a. **docs hashmap :** This hashmap, named docs, links document IDs to document objects in the DocumentService class. It enables rapid document retrieval by ID.
   b. **cache hashmap** : The cache hashmap, associated with document IDs and document content, is used to minimize redundant document reads in the DocumentService class.
2. **AssignedUser Class:**
   a. tokenToUser hashmap : In the AssignedUser class, the tokenToUser hashmap associates usernames with tokens to validate user requests.

3. **Affinity Class:**
    a. documentsAffinity hashmap: In the Affinity class, the documentsAffinity hashmap maps document IDs to their affinity nodes, streamlining access to these relationships.

➢ **Additionally, an ArrayList is employed in the deleteByDatabaseName method within the DocumentService class, which accumulates document IDs for subsequent deletion using the deleteDocument() method. This approach ensures efficient management of document deletion within the "Node" project, which operates as an integral component of the bootstrapping node.**

Multithreading and locks

1. **Multithreading:** The Broadcasting class employs multithreading to broadcast messages simultaneously to all nodes, ensuring efficient communication.
2. **Synchronized Methods in Bootstrapping:**
    - Methods like **login(), signup(), assignUserToNode(), and clearNode()** in the Bootstrapping node are synchronized to prevent race conditions and multithreading issues during user authentication, signup, and node assignment.
3. **Synchronized Methods in Node:**
    - In the Node, various synchronized methods are implemented to ensure thread safety and data integrity. These include methods like **setToken(), createDatabase(), deleteDatabase(), addDocument(), deleteDocument(), storeDocumentsFile(), deleteByDatabaseName(), storeCounterFile(), and editDocument().**
    - The **editDocument()** method utilizes optimistic locking to prevent conflicts during document editing, enhancing overall system stability.

➢ **These synchronization measures are crucial for maintaining data consistency, preventing concurrency issues, and ensuring the reliable operation of your system.**

Node hashing and load balancing

1. **Node Hashing:**
   a. Each node is assigned a unique name, such as node1, node2, to access them within the Docker network.
   b. Nodes are also assigned unique external ports for access from outside the network.
2. **Load Balancing:**
   a. Load balancing is achieved using a deque (double-ended queue) containing objects representing all nodes in the system.
   b. When a user logs in, they are assigned the first node in the deque, and that node is moved to the end of the deque.
   c. When a user logs out, their assigned node moves to the front of the deque, making it available for the next user.
3. **Affinity Load Balancing:**
   a. For affinity-based load balancing, the allocation of documents is done in a round-robin or circular manner.
   b. Each document created is assigned to nodes in a sequential and balanced way.

Communication protocols between nodes.

1. **Docker Network Communication:**
   - Nodes operate within a Docker network, ensuring a secure and isolated environment for communication.
2. **Node Discovery:**
   - Each node is aware of the URLs of all other nodes in the network. This knowledge enables direct communication among nodes.
3. **Data Sharing:**
   - Nodes exchange data by sending strings and JSON files to each other.
4. Communication Endpoints:
   - Specific endpoints are defined for nodes to send and receive data from one another. These endpoints facilitate data sharing and collaboration among nodes.

Security issues

1. **Lack of Data and Token Encryption:**
    - Data and tokens are transmitted without encryption, potentially exposing sensitive information during communication between nodes.
2. **Weak Password Restrictions during Signup:**
    - The system lacks robust password restrictions during user signup, which could lead to weak or easily guessable passwords.
3. **Endpoint Exposure:**
    - Endpoints used for data sharing between nodes are not adequately protected or restricted, making them potentially accessible to unauthorized users.

Code testing

Testing Methodology:

1. **Postman Testing:** Use Postman to simulate user actions by sending requests and accessing various endpoints.
2. **Docker Desktop:** Verify the behavior of your code by deploying it within Docker containers using Docker Desktop.
3. **Observation with Printing:** Implement print statements in your code to track its execution and ensure that functions are accessed correctly.

- **By combining Postman for API testing, Docker Desktop for deployment and system testing, and print statements for code-level debugging and validation, you create a comprehensive testing strategy to ensure the correct functionality and behavior of your decentralized NoSQL DB system.**

Clean Code

here's a more concise summary of the practices that I have employed to keep my code clean and well-organized:

1. **Meaningful Variable Names:** Use descriptive variable names instead of single letters to enhance code readability.
2. **Descriptive Function Names:** Each function has a name that clearly explains its purpose, reducing ambiguity.
3. **In-Function Comments:** Add comments inside functions to provide additional context or explanation, especially when function names might not be self-explanatory.
4. **Code Organization:** Organize classes into packages based on their types or functionalities to maintain a structured codebase.
5. Code Formatting: Format your code consistently to make it more readable and maintainable.
6. **SOLID Principles:** Implement SOLID principles (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) to design more modular and maintainable code.
7. **OOP Principles:** Adhere to Object-Oriented Programming (OOP) principles, including encapsulation, inheritance, polymorphism, and abstraction, to create well-structured and reusable code.

By following these practices, you ensure that your code is clean, organized, and adheres to best coding practices, making it easier to understand, maintain, and extend in the future.

Effective java

here's a more streamlined presentation of the items that I have mentioned from the "Effective Java" book:

1. **Singleton Pattern:** Enforce the singleton property by using a private constructor.
2. **Access Control:** Minimize the accessibility of classes and members to encapsulate implementation details.
3. **Local Variables:** Keep the scope of local variables as narrow as possible to improve code clarity.
4. **Avoid Null:** Avoid returning null from methods; instead, return empty collections or use Optional where applicable.
5. **Lists over Arrays:** Prefer using lists (e.g., ArrayList) over arrays for better flexibility and type safety.
6. **Interfaces over Abstract Classes:** Prefer interfaces when defining contracts to allow multiple interface inheritance.
7. **Interface References:** Refer to objects by their interfaces to promote flexibility and decouple code dependencies.

These principles from "Effective Java" focus on writing clean, maintainable, and robust Java code by emphasizing good practices in design and coding.

Design patterns

I used only singleton design pattern in two classes , first one is UserNode class in bootstrapping node , the other one in AssingedUser in worker node.

SOLID Principles

1. **Single Responsibility Principle (SPR):**
    - Each class in the application is designed with a single, clear responsibility, making it easier to manage and maintain.
2. **Open/Closed Principle (OCP):**
    - Classes are closed for modification, meaning existing code doesn't require changes, and they are open for extension, allowing developers to build new functionality through inheritance and extension.
3. **Liskov Substitution Principle (LSP):**
    - Inheritance is used from interfaces, aligning with the LSP principle by ensuring that derived classes can be used interchangeably with their base classes.
4. **Interface Segregation Principle (ISP):**
    - Interfaces in the application are designed to be small, serving one specific purpose, which eliminates the need for splitting them.
5. **Dependency Inversion Principle (DIP):**
    - The application implements dependency inversion by providing interfaces with no implementation details, leaving the implementation to child classes. This promotes flexibility and reduces coupling between components.

DevOps practices

1. **Planning**:
    - The project follows a structured planning phase to define requirements, objectives, and timelines.
2. **Development**:
    - Development is carried out in adherence to established requirements and coding standards.
3. **Testing:**
    - A robust testing process is in place to ensure the quality and reliability of the code.
4. **Build Automation:**
    - Maven is used for build automation, streamlining the process of compiling and packaging the application.
5. **Version Control:**
    - GitHub is employed for version control, allowing for collaborative development, tracking changes, and managing code history.
6. **Containerization:**
    - Docker is used to containerize nodes and create an efficient network for inter-node communication, enhancing scalability and deployment flexibility.