



Decentralized Cluster-Based NoSQL DB System (capstone)

Name: Akram Jaghoub

Table of Contents

1.0 Introduction	3
2.0 System Architecture and Design	4
2.1 System Design	4
2.2 Database Design	5
3.0 Database Implementation	6
3.1 Indexing	6
3.2 Schema	7
	7
4.0 Data structures	8
5.0 Multithreading and locks	9
5.1 Multithreading in Spring	9
5.2 Collection and Document Level Locking:	9
5.0 Data Consistency	11
6.0 Node hashing and load balancing	12
6.1 users load balancing	12
6.2 Affinity Distribution	13
7.0 Security Issues	14
7.1 Admin	14
7.2 Customers	15
8.0 Code testing	16
9.0 Clean Code	17
9.0 Effective Java Items	20
10.0 SOLID principles	25
10.1 Single Responsibility Principle	25
10.2 Open Closed Principle	25
10.3 Liskov Substitution Principle	25
10.4 Interface Segregation Principle	25
10.5 Dependency Inversion Principle	26
11.0 Design Patterns	26
11.1 Creational	26
11.1.1 Singleton	26
11.1.2 Factory	26
11.1.3 Builder	27
11.2 Behavioral	27
11.2.1 Command	27
12.0 DevOps practice	28
12.1 Docker	28
12.2 Maven	28
12.3 Git and Github	28
12.4 Spring	28
13.0 User Manual	29

1.0 Introduction

In today's world of data management, there's a clear difference between SQL and NoSQL databases. SQL databases, known for their structured tables, are great for handling detailed queries. However, with the growing amount of varied data and the evolving needs of applications, NoSQL databases have emerged as the go-to choice for their flexibility and scalability.

This report sheds light on the creation of a unique, decentralized NoSQL database system. At its core, this system uses a document-based format, wrapping data inside JSON objects that are then saved directly to the file system. Even though it's a disk-based database, fast read and write operations were utilized to compromise this matter.

A pivotal component of this system is the bootstrapper. It initializes the database nodes (also known as worker nodes) and equips them with the essential details to work in harmony. There are two main user types for this system: admins and customers. They interact with the bootstrapper and database nodes through specialized interfaces. Additionally, a sample application demonstrates how one might use these interfaces, from database and collection creation on the admin side to conducting transactions on the customer side.

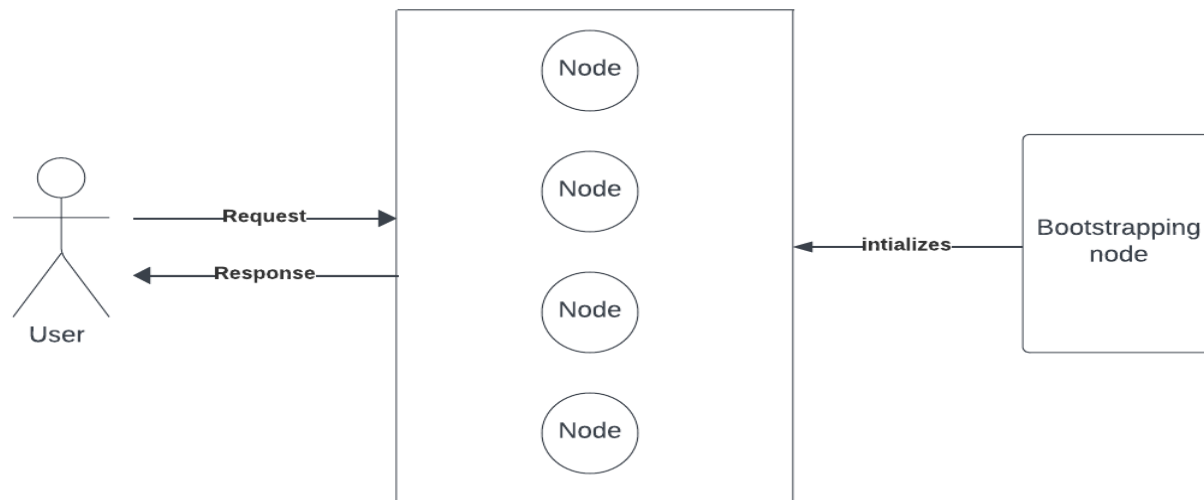
One standout feature is the system's approach to load balancing. This ensures users are efficiently distributed by the bootstrapper.

Furthermore, principles of clean coding and established design patterns have been employed. This ensures that the system remains flexible, ready for modifications in the future.

Finally, the system uses partitioning to spread data across multiple nodes. It also employs replication to ensure data is always accessible and consistent across all worker nodes. Docker and docker network were utilized to help me achieve decentralization.

2.0 System Architecture and Design

2.1 System Design

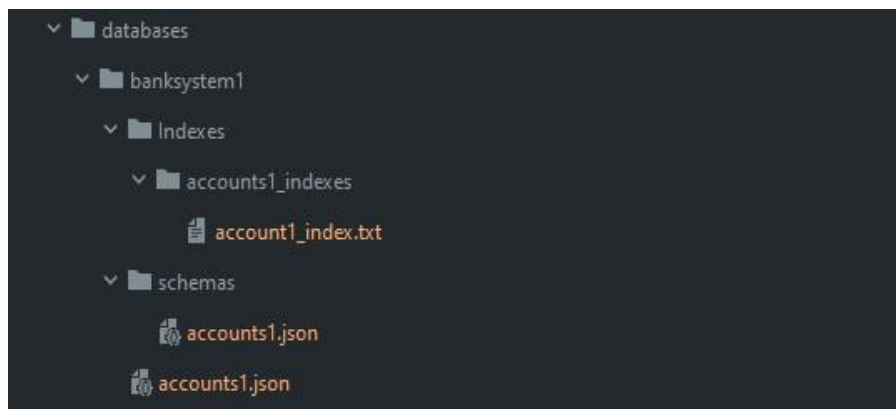


The system comprises four worker nodes and a bootstrapping node, with each operating within its own Docker container. Regarding the database nodes, they are networked together to facilitate node communication and to replicate data amongst themselves. This replication ensures system reliability, as every node maintains a complete data copy. While users can connect to only one node at a given time, the system seamlessly manages node communication through RESTful API calls.

Each node operates through multiple layers before processing a request. Initially, there's the authentication layer, ensuring users connect using verified credentials. Following this is the load balancing layer, which determines the distribution of users across nodes. Finally, the business logic layer handles all database operations.

2.2 Database Design

The database structure or the file system structure is at follows:



As observed, the root directory, "databases", houses the stored databases such as "banksystem1". Within each database, there's a 'schemas' directory containing the schema for every collection. Additionally, the database holds all the collections and an 'indexes' directory that stores the indexes for each collection.

When a new collection is made, two things happen. First, an "accounts1.json" is created in the schema, showing the structure future documents will follow. Second, another collection appears in the "banksystem1" directory. This is where individual documents will be stored. As soon as a document is added, an 'indexes' folder is made, holding the indexes for this collection, named "account1_indexes".

each schema will follow the structure shown below in the picture

```
1  {
2    "type": "object",
3    "properties": {
4      "password": "STRING",
5      "balance": "DOUBLE",
6      "clientName": "STRING",
7      "accountType": "STRING",
8      "accountNumber": "LONG",
9      "hasInsurance": "BOOLEAN"
10   },
11   "required": [
12     "accountNumber",
13     "clientName",
14     "balance",
15     "accountType",
16     "hasInsurance",
17     "password"
18   ]
19 }
```

And each collection will have a JSON Array that contains multiple documents as shown below in the picture.

```
[
  {
    "password": "5994471abb01112afcc18159f6cc74b4f511b99806da59b3caf5a9c173cacfc5",
    "balance": 20.0,
    "clientName": "Akram Jaghoub",
    "accountType": "DEPOSIT",
    "_id": "b308b9ff-8f51-46d0-b81d-0cc86725fbfc",
    "accountNumber": 12345,
    "hasInsurance": "false"
  }
]
```

3.0 Database Implementation

In this project, a NoSQL database system was built using the computer's file system for saving data and unique indexes for each document. This approach offers advantages like quick and flexible data handling. NoSQL databases can manage large amounts of diverse data. With the unique indexes, the system can quickly find and work with data, even when there are millions of documents. As discussed in section 2.2, the project uses a predefined format, or schema, for the documents. Each document is checked against this format to ensure it's correct. This schema keeps the data consistent and helps users understand the database layout more easily. Now let's dive deep into this:

3.1 Indexing

Every database leans heavily on indexing to optimize data retrieval. The choice of an indexing data structure can greatly influence performance, especially when dealing with voluminous datasets. During this project, a deep dive was taken into the world of indexing to determine the optimal data structure to employ. The list of candidates was extensive, ranging from HashMaps to B+ Trees, Tries, and more.

In this project, B+ Tree data structure was adopted. Here's a deeper exploration:

B+ Tree Indexing:

B+ Trees are prevalent indexing structures known for their swift data access capabilities. They offer several benefits, especially when managing large datasets typical of NoSQL databases. B+ Trees are inherently balanced. This ensures that data retrieval, insertion, or deletion operations maintain consistent and peak performance. In this system, B+ Trees were leveraged for two primary purposes:

- Document-level indexing, using each document's ID.

```
1 usage  ▲ Akram Jaghoub
public synchronized void insertIntoIndex(String collectionName, String documentId, int index) {
    String existingValue = getIndex(collectionName).search(documentId);
    if (existingValue == null) {
        getIndex(collectionName).insert(documentId, String.valueOf(index));
        FileService.appendToIndexFile(FileService.getIndexFilePath(collectionName), documentId, String.valueOf(index));
    }
}
```

- Property-level indexing, utilizing a unique identifier crafted from a combination of the document's ID and property name.

```
2 usages Akram Jaghoub *
public synchronized void insertIntoPropertyIndex(String collectionName, String propertyName, String propertyValue, String documentId) {
    String propertyIndexKey = collectionName + "_" + propertyName;
    PropertyIndex propertyIndex = propertyIndexMap.get(propertyIndexKey);
    if (propertyIndex == null) {
        throw new IllegalArgumentException("Property Index does not exist");
    }
    String combinedKey = documentId + "_" + propertyName;
    String existingPropertyValue = propertyIndex.search(combinedKey);
    if (!propertyValue.equals(existingPropertyValue)) {
        System.out.println("Inserted new entry in property index for collection: " + collectionName + " property: " + propertyName);
        propertyIndex.insert(combinedKey, propertyValue);
        FileService.appendToIndexFile(FileService.getPropertyIndexFilePath(collectionName, propertyName), combinedKey, propertyValue);
    } else {
        System.out.println("Entry already exists in property index for collection: " + collectionName + " property: " + propertyName);
    }
}
```

3.2 Schema

As previously mentioned, the creation of a collection goes hand-in-hand with the establishment of a schema. Every time a new document is introduced to this collection, it undergoes a rigorous validation against its associated schema. Should the document not align with the schema, the server promptly communicates the discrepancy to the user. However, if the document successfully matches the schema, it proceeds to be stored in the database. Essentially, this validation process ensures data integrity by confirming the document's conformity to the predetermined schema prior to storage. This process is shown in the pictures below.

```
1 usage Akram Jaghoub *
public boolean schemaValidator(String collectionName, String json) {
    try {
        JSONObject schema = FileService.readSchema(collectionName);
        JsonSchemaFactory jsonSchemaFactory = JsonSchemaFactory.getInstance(SpecVersion.VersionFlag.V201909);
        JsonSchema jsonSchema = jsonSchemaFactory.getSchema(schema.toString());
        ObjectMapper objectMapper = new ObjectMapper();
        JsonNode jsonNode = objectMapper.readTree(json);
        Set<ValidationMessage> validationMessageSet = jsonSchema.validate(jsonNode);
        return validationMessageSet.isEmpty();
    } catch (IOException e) {
        throw new RuntimeException("Validation failed: " + e.getMessage());
    }
}
```

```
if (document.isValidDocument(schemaValidator, collectionName)) {
    String documentId = document.getId();
    int workerPort = affinityManager.getWorkerPort(documentId); //adding affinity based on the document's id
```

```
} else {
    return new ApiResponse( message: "Document does not match the schema for " + collectionName, HttpStatus.CONFLICT);
}
```

4.0 Data structures

B+ Tree: At the heart of the indexing mechanism lies the B+ Tree. Its ability to provide consistent access times, irrespective of the dataset's size, makes it an ideal choice. The primary utilization of the B+ Tree in this system is centered around its indexing capabilities. Through a system of unique identifiers, each document's ID becomes an access point, allowing for rapid data retrieval and modification.

HashMap: Another integral component of the system's data structure is the HashMap. The project employs a special construct, known as the 'account directory', built upon HashMap. It maps a customer's account number to their specific database, collection, and document ID. This direct mapping alleviates the need for exhaustive searches across the filesystem, ensuring efficient data access. Also hashmaps were used in many other parts of the program.

```
public class AccountDirectoryService {  
    4 usages  
    private final Map<String, AccountReference> accountDirectory = new ConcurrentHashMap<>();  
  
    no usages  Akram Jaghoub *  
    @PostConstruct  
    public void init() { this.accountDirectory.putAll(FileService.loadAccountDirectory()); }  
  
    1 usage  Akram Jaghoub  
    public void registerAccount(String accountNumber, String dbName, String collectionName, String documentId) {  
        accountDirectory.put(accountNumber, new AccountReference(dbName, collectionName, documentId));  
        FileService.saveAccountDirectory(this.accountDirectory);  
    }  
  
    3 usages  Akram Jaghoub  
    public AccountReference getAccountLocation(String accountNumber) { return accountDirectory.get(accountNumber); }  
}
```

List: Throughout various segments of the program, lists play a significant role, especially when retrieving collections or databases. The list data structure offers sequential access and aids in organizing and displaying data sets like collections or databases in a structured manner.

```
1 usage  Akram Jaghoub  
public List<String> readCollections(Database database) {  
    Set<String> uniqueCollections = new HashSet<>();  
    database.getCollectionLock().lock();  
    try {  
        List<String> inMemoryCollections = database.readCollections();  
        uniqueCollections.addAll(inMemoryCollections);  
        List<String> inFileCollections = DatabaseFileOperations.readCollections();  
        uniqueCollections.addAll(inFileCollections);  
        return new ArrayList<>(uniqueCollections);  
    } finally {  
        database.getCollectionLock().unlock();  
    }  
}
```


5.0 Multithreading and locks

In the realm of database systems, concurrency is pivotal. It ensures that multiple users or processes can interact with the database simultaneously without causing data inconsistency or system crashes. This project incorporates advanced multithreading techniques and locking mechanisms to safeguard data integrity and provide efficient access.

5.1 Multithreading in Spring

Spring, being the foundational framework for the project, inherently supports multithreading. By default, Spring handles incoming HTTP requests with a servlet container that spawns a new thread for each request. This means that with every user or process interaction with the database, a new thread is created, ensuring smooth and concurrent operations.

5.2 Collection and Document Level Locking:

To avoid race conditions when creating or deleting collections in the database, I utilized the `ReentrantLock` in my implementation. This allowed me to ensure that only one thread at a time could access the critical section of the code for (creating, reading, updating, deleting) operations. By doing so, I was able to prevent data inconsistencies and conflicts that could arise from multiple threads attempting to access the database simultaneously. With the `ReentrantLock` in place, I could rest assure that my application was robust and could handle concurrent access without any issues. Which was done on two levels:

- **Collection-Level Locking:**

```
1 usage  ~ Akram Jaghoub *  
public ApiResponse createCollection(Database database, String collectionName, JSONObject jsonSchema) {  
    database.getCollectionLock().lock();  
    try {  
        database.createCollection(collectionName);  
        return DatabaseFileOperations.createCollection(collectionName, jsonSchema);  
    } finally {  
        database.getCollectionLock().unlock();  
    }  
}
```

- Document-Level Locking:

```
2 usages  Akram Jaghoub
public ApiResponse updateDocumentProperty(Collection collection, Document document) {
    String collectionName = collection.getCollectionName().toLowerCase();
    String propertyName = document.getPropertyName();
    String newPropertyValue = document.getPropertyValue().toString();
    Object castedValue = DataTypeUtil.castToDataType(newPropertyValue, collectionName, propertyName);
    String castedValueString = castedValue.toString();
    collection.getDocumentLock().lock();
    try {
        if (!indexManager.propertyIndexExists(collectionName, propertyName)) {
            indexManager.createPropertyIndex(collectionName, propertyName);
        }
        return DatabaseFileOperations.updateDocumentProperty(collectionName, document, propertyName, castedValueString, indexManager);
    } finally {
        collection.getDocumentLock().unlock();
    }
}
```

```
public class Collection {
    2 usages
    private final ReentrantLock documentLock;
    2 usages
    private final String collectionName;

    Akram Jaghoub *
    public Collection(String collectionName) {
        this.collectionName = collectionName;
        documentLock = new ReentrantLock();
    }

    Akram Jaghoub
    public String getCollectionName() { return collectionName; }

    10 usages  Akram Jaghoub *
    public ReentrantLock getDocumentLock() { ←
        return documentLock;
    }
}
```

5.0 Data Consistency

To ensure data consistency across all workers in the system, a mechanism was implemented using the database API. When a write operation (creating, deleting, updating) is performed on one worker, the same request is broadcasted to all other workers. To prevent endless broadcasting of requests, a variable called "X-Broadcasted" is included in the header of each write query which is set to be false by default. If the "X-Broadcasted" variable is set to false, it indicates that the request should be broadcasted to all of the other workers after the write operation is completed. Once the request is broadcasted, the variable is set to true to prevent it from being broadcasted again. If the variable is already set to true, it indicates that the request has already been broadcasted and should not be broadcasted again. This mechanism ensures that all workers have consistent data and the same copy of data "data replication" at all times.

```
private void broadcast(String url, HttpMethod method, boolean isBroadcasted, Optional<JSONObject> dataOpt, Map<String, String> additionalHeaders) {
    try {
        JsonNode adminCredentials = FileService.readAdminCredentialsFromJson();
        RestTemplate restTemplate = new RestTemplate();
        HttpHeaders headers = new HttpHeaders();
        headers.set("username", adminCredentials.get("username").asText());
        headers.set("password", adminCredentials.get("password").asText());
        headers.set("X-Broadcast", String.valueOf(isBroadcasted));
        additionalHeaders.forEach(headers::set);
        if (dataOpt.isPresent()) {
            headers.setContentType(MediaType.APPLICATION_JSON);
        }
        HttpEntity<String> requestEntity = dataOpt.map(jsonObject →
            new HttpEntity<>(jsonObject.toJSONString(), headers)).orElseGet(() → new HttpEntity<>(headers));
        restTemplate.exchange(url, method, requestEntity, String.class);
    } catch (Exception e) {
        System.out.println("[ERROR] Broadcasting failed. URL: " + url);
        e.printStackTrace();
    }
}
```

And for example, we wanted to broadcast a database creation request is as follows:

```
@Override
public void broadcastOperation(JSONObject details) {
    System.out.println("[INFO] Starting broadcasting database creation to others..");
    Database database = (Database) details.get("database");
    int originatingWorkerPort = (int) details.get("originatingWorkerPort");
    for (int i = 1; i ≤ affinityManager.getNumberOfNodes(); i++) {
        if (i == originatingWorkerPort) {
            System.out.println("[SKIP] Skipping broadcast to worker " + i + " (origin node)...");
            continue;
        }
        System.out.println("[BROADCAST] Broadcasting to worker " + i + "...");
        String url = "http://worker" + i + ":9000/api/createdb" + "/" + database.getDatabaseName();
        System.out.println("[BROADCAST] Broadcasting to URL: " + url);
        broadcastService.broadcast(url, HttpMethod.POST, isBroadcasted: true);
    }
}
```

6.0 Node hashing and load balancing

6.1 users load balancing

The project leverages the round-robin technique for load balancing, a proven method that ensures an equitable distribution of users across all worker nodes. By adopting this approach, the system significantly enhances its performance, safeguarding against potential bottlenecks or overloads on any individual node. Furthermore, when the system initializes containers with pre-existing users or experiences a user deletion, meticulous care is taken to rebalance and preserve a harmonious user distribution. This strategic balance optimizes both consistency and overall operational efficiency.

```
3 usage  # Akram Jaghoub
public Node assignUserToNextNode(String identity) {
    Node node = getNextNode();
    if (!nodeUsers.containsKey(node)) {
        nodeUsers.put(node, new ArrayList<>());
    }
    nodeUsers.get(node).add(identity);
    System.out.println("user is assigned to node " + node.getNodeNumber() + " and the ip address is " + node.getNodeIP());
    return node;
}

1 usage  # Akram Jaghoub
public synchronized Node getNextNode() {
    if (nodesService.getNodes().isEmpty()) {
        throw new IllegalStateException("No nodes available for balancing");
    }
    Node nextNode = nodesService.getNodes().get(nextNodeIndex);
    updateNextNodeIndex();
    return nextNode;
}

1 usage  # Akram Jaghoub
private void updateNextNodeIndex() { nextNodeIndex = (nextNodeIndex + 1) % nodesService.getNodes().size(); }
```

```
public void balanceExistingUsers() {
    nextNodeIndex = 0;
    System.out.println("balancing existing users.....");
    File customersFile = new File(FileServices.getUserJsonPath( fileName: "customers"));
    if (FileServices.isFileExists(customersFile.getPath())) {
        JSONArray usersArray = FileServices.readJsonArrayFile(customersFile);
        if (usersArray != null) {
            for (Object obj : usersArray) {
                JSONObject userJson = (JSONObject) obj;
                String accountNumber = (String) userJson.get("accountNumber");
                this.assignUserToNextNode(accountNumber);
            }
        }
    } else {
        System.out.println("No existing users found...");
    }
}
```

6.2 Affinity Distribution

In the project, instead of implementing a typical round-robin load balancing method, I chose an affinity distribution approach. This involves calculating the document ID through a hashing mechanism and then taking the modulus by the number of worker nodes, yielding one of the four possible worker nodes. Admittedly, this method might lead to an uneven load on some servers due to an accumulation of affinity documents.

However, the rationale behind this design choice is speed and efficiency. Consider a scenario where we have four workers, and worker1 initiates a request to create or update a document, but worker4 holds the affinity for that document. Given our system, by computing the document ID's hash and modulating it by the number of nodes, we can instantly pinpoint the affinity node and direct the request appropriately. On the contrary, if we had employed a round-robin load balancing system, the system would need to cycle through each worker until reaching worker4, which can be time-consuming. While there are advantages to traditional load balancing, such as more even server loads, the chosen approach prioritizes immediate data access and efficiency.

```
public synchronized void setCurrentWorkerPort(String currentWorkerName) {
    System.out.println(currentWorkerName);
    try {
        this.currentWorkerNumber = Integer.parseInt(currentWorkerName.trim().replace( target: "worker", replacement: ""));
        System.out.println(currentWorkerNumber);
    } catch (NumberFormatException e) {
        throw new RuntimeException("Invalid worker name format.");
    }
}

4 usages  Akram Jaghoub
public int getWorkerPort(String documentId) {
    int index = calculateWorkerIndex(documentId);
    return index + 1;
}

1 usage  Akram Jaghoub
@ public int calculateWorkerIndex(String documentId) {
    int hashCode = documentId.hashCode();
    return Math.abs(hashCode) % NUMBER_OF_NODES;
}
```

```
int workerWithAffinity = affinityManager.getWorkerPort(document.getId()); ←
int currentWorkerPort = affinityManager.getCurrentWorkerPort();
if (!isBroadcasted && currentWorkerPort != workerWithAffinity) {
    System.out.println("The current worker is not the one with affinity. Redirecting...");
    document.setReplicated(true); //set replication to true to not change on the data and avoid duplications
    return redirectionService.redirectToWorkerForCreation(database, collection, document, workerWithAffinity);
}
```

7.0 Security Issues

7.1 Admin

To enhance the system's security, I adopted a single-admin approach. This admin is in charge of all database tasks, from managing databases and collections to handling documents. At the start of the application, every worker node is informed about this admin through the bootstrapping process. Before accessing the banking interface, the admin needs to log in.

Afterwards the admin goes under an authentication process, the admin's credentials are verified against a worker node. I chose to check against the first worker node, though any node could have been used. Crucially, passwords aren't stored in plain text. Instead, they are hashed using SHA-256. Thus, during verification, the system compares the hashed versions of passwords. The algorithm and the validation process are shown in the pictures below.

Once authenticated and inside the banking system interface, any subsequent action undertaken by the admin undergoes a further security check. Each operation requires the validation of the admin's active session, which is passed along with every request. This added layer of security ensures that only authorized personnel, specifically the admin, can execute specific actions.

```
private ApiResponse verifyCredentials(String userType, String identity, String password) {
    ObjectMapper objectMapper = new ObjectMapper();
    String path = FileService.getUserJsonPath(userType);
    String identityField = userType.equals("admin") ? "username" : "accountNumber";
    String hashedPassword = PasswordHashing.hashPassword(password); //To compare against stored hash password

    try {
        // admin
        if (userType.equals("admin")) {
            JsonNode rootNode = objectMapper.readTree(new File(path));
            String storedIdentity = rootNode.path(identityField).asText();
            String storedPassword = rootNode.path("password").asText();
            if (identity.equals(storedIdentity) && hashedPassword.equals(storedPassword)) {
                return new ApiResponse( message: "Authenticated", HttpStatus.OK);
            } else {
                return new ApiResponse( message: "Wrong Username or Password", HttpStatus.BAD_REQUEST);
            }
        }
    }
}
```

```
4 usages
public final class PasswordHashing {
    no usages
    private PasswordHashing(){}

    4 usages
    @
    public static String hashPassword(String password) {
        try {
            MessageDigest md = MessageDigest.getInstance( algorithm: "SHA-256");
            byte[] hashBytes = md.digest(password.getBytes(StandardCharsets.UTF_8));
            StringBuilder sb = new StringBuilder();
            for (byte b : hashBytes) {
                sb.append(String.format("%02x", b));
            }
            return sb.toString();
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException("SHA-256 algorithm not available.", e);
        }
    }
}
```

7.2 Customers

Let's discuss how customer accounts are set up. When the admin creates a document, they are essentially opening an account for a customer similar to how a bank representative would in real life. Once the account is successfully registered in the database, the next step involves the bootstrapper. The customer's account number and password, set by the admin, are sent to the bootstrapper, which then assigns the customer to a specific worker node. This worker node receives the customer's credentials and adds them to its local database. Note that, for security, passwords are hashed in the same manner as the admin's. With their credentials set, customers can log in using their account numbers and passwords to access their accounts, make transactions, view balances, or make deposits and withdrawals.

In the demo app after successful creation of the document (account) as mentioned above we are adding the customer by sending it the bootstrapper and then database worker node.

```
ResponseEntity<String> responseEntity = documentService.createAccount(dbName, collectionName, bankAccount, session);
HttpStatus status = (HttpStatus) responseEntity.getStatusCode();
String message = responseEntity.getBody();
if (status == HttpStatus.CREATED) {
    List<BankAccount> accounts = documentService.readAccounts(dbName, collectionName, session);
    //add the customer to the bootstrapper after creating their account
    userService.addCustomer(bankAccount.getAccountNumber(), bankAccount.getPassword(), session);
    return ResponseEntity.status(status).body(accounts);
} else if (status == HttpStatus.CONFLICT) {
```

While the system relies on identity (username or account number) and a password for user authentication, there's still room to reinforce node communication security. One way to strengthen this is by incorporating JWT (JSON Web Tokens). Here's how it works: Whenever a node sends a message to another, it will attach a JWT token. The receiving node then verifies this token before processing the message. The secret key used to generate these tokens is supplied initially by the bootstrapper during configuration. This added layer ensures more secure and trusted exchanges between nodes.

8.0 Code testing

When developing the system, I prioritized thorough testing for each new feature. I began by focusing on the database operations, testing them extensively using the Postman API. Debug statements were generously integrated to trace issues and to get a deeper understanding of the code flow. After ensuring the database operations were robust, I shifted my attention to the bootstrapper, applying similar rigorous testing methods.

However, when I integrated the demo app and transitioned the testing from Postman to the frontend, I encountered numerous challenges. Notably, there were persistent issues related to database indexing. These problems only surfaced during frontend testing, emphasizing the importance of testing across different interfaces. By simultaneously using both the frontend and Postman, I was able to identify and rectify these issues.

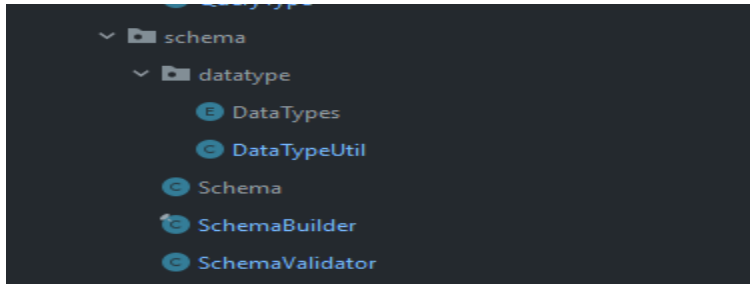
To further validate synchronization and locking mechanisms, I simulated concurrent scenarios by sending multiple simultaneous requests through Postman. This ensured the system's reliability under high-demand situations.

9.0 Clean Code

1. Meaningful and consistent names:

Used meaningful and descriptive names in all parts of the code that represents the functionality, and it follows consistent naming convention and formatting style of it such as:

- Classes



- Variables

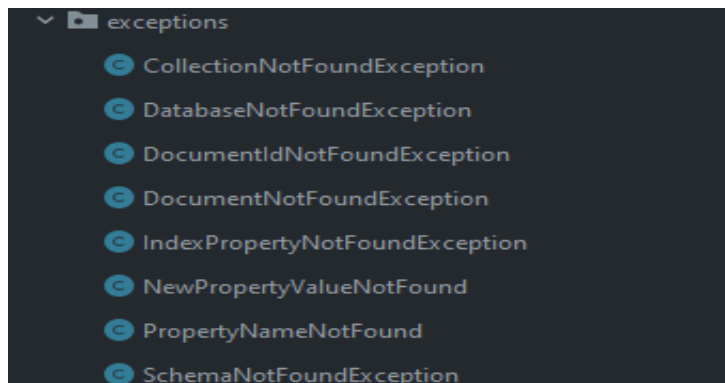
```
no usages
private String propertyName;
no usages
private Object propertyValue;
no usages
private boolean hasAffinity;
```

- Methods

```
1 usage  Akram Jaghoub
public void registerAccount(String accountNumber, String dbName, String collectionName, String documentId) {
    accountDirectory.put(accountNumber, new AccountReference(dbName, collectionName, documentId));
    FileService.saveAccountDirectory(this.accountDirectory);
}

3 usages  Akram Jaghoub
public AccountReference getAccountLocation(String accountNumber) { return accountDirectory.get(accountNumber); }
}
```

- Exceptions



2. Comments:

Used comments only when needed for the parts that need to be understood.

```
Usage: Akram Jagnoub

public void addCustomer(Customer customer) {
    Node node = loadBalancer.assignUserToNextNode(customer.getAccountNumber()); //assign the user to a worker node
    String hashedPassword = PasswordHashing.hashPassword(customer.getPassword()); //hashing the password for security
    customer.setPassword(hashedPassword);
    FileServices.saveUserToJson( fileName: "customers", customer.toJson()); //adding the customer to json file
    String url = "http://" + node.getNodeIP() + ":9000/api/add/customer";
    HttpHeaders headers = new HttpHeaders();
    headers.set("accountNumber", customer.getAccountNumber());
    headers.set("password", customer.getPassword());

    if (status == HttpStatus.CREATED) {
        List<BankAccount> accounts = documentService.readAccounts(dbName, collectionName, session);
        //add the customer to the bootstrapper after creating their account
        userService.addCustomer(bankAccount.getAccountNumber(), bankAccount.getPassword(), session);
        return ResponseEntity.status(status).body(accounts);
    } else if (status == HttpStatus.CONFLICT) {
        return ResponseEntity.status(status).body(accounts);
    }
}
```

3. Avoiding Returning Null:

I made efforts to circumvent returning null values. Instead, I opted to return empty lists or any message that indicates the failed situation. Moreover, I utilized Java's Optional class as a more robust and expressive alternative to handle cases where a value might be absent.

```
3 usages: new *

public static Optional<Admin> getAdminCredentials() {
    ObjectMapper mapper = new ObjectMapper();
    String path = FileServices.adminJsonFilePath();
    File file = new File(path);
    if (!file.exists()) {
        return Optional.empty();
    }
    try {
        Admin credentials = mapper.readValue(file, Admin.class);
        return Optional.of(credentials);
    } catch (Exception e) {
        e.printStackTrace();
        return Optional.empty();
    }
}
```

4. DRY Principle:

In an effort to uphold the DRY (Don't Repeat Yourself) principle, I recognized a significant amount of redundancy in our broadcasting logic. To tackle this issue, I employed the builder design pattern, especially during broadcasting. This not only eliminated repetitive code but also streamlined the process, making it more efficient and maintainable. By encapsulating the broadcasting logic in a builder, I ensured that any future modifications would be centralized, reducing the risk of inconsistencies. This design choice has greatly benefitted various parts of the system, allowing for a cleaner and more coherent codebase.

```
@Override
public void broadcastOperation(JSONObject details) {
    Database database = (Database) details.get("database");
    Collection collection = (Collection) details.get("collection");
    Document document = (Document) details.get("document");
    int originatingWorkerPort = (int) details.get("originatingWorkerPort");
    for (int i = 1; i <= affinityManager.getNumberOfNodes(); i++) {
        if (i == originatingWorkerPort)
            continue; // Skip the worker which already deleted the document
        System.out.println("Node " + originatingWorkerPort + " broadcasting to worker" + i + " for ID: " + document.getId() + ". Current version: " + document.getVersion());
        String url = "http://worker" + i + ":9000/api/" + database.getDatabaseName() + "/"
            + collection.getCollectionName() + "/updateDoc/" + document.getPropertyName() + "?doc_id=" + document.getId();
        System.out.println(url + " in broadcast url");
        Map<String, String> additionalHeaders = new HashMap<>();
        additionalHeaders.put("newPropertyValue", document.getPropertyValue().toString());

        new BroadcastService.BroadcastRequestBuilder() ←
            .withUrl(url)
            .withMethod(HttpMethod.PUT)
            .isBroadcasted(true)
            .withAdditionalHeaders(additionalHeaders)
            .broadcast();
    }
}
```

5. Try-with-resources:

In the project, try-with-resources was constantly applied, emphasizing efficient and safe resource management. By leveraging this construct, they ensured that resources, such as streams and files, were properly and automatically closed post-usage. This practice not only minimized potential resource leaks but also contributed to the code's clarity and robustness.

```
try (RandomAccessFile stream = new RandomAccessFile(file, mode: "rw");
     FileChannel channel = stream.getChannel()) {
    channel.truncate(size: 0);
    byte[] bytes = jsonArray.toJSONString().getBytes();
    ByteBuffer buffer = ByteBuffer.allocate(bytes.length);
    buffer.clear();
    buffer.put(bytes);
    buffer.flip();
    while (buffer.hasRemaining()) {
        channel.write(buffer);
    }
}
return true;
```

9.0 Effective Java Items

- Item 1: Consider static factory methods instead of constructors

```
2 usages  Akram Jaghoub *
public class QueryObjectFactory {

    no usages  new *
    private QueryObjectFactory() {
    }

    1 usage  Akram Jaghoub *
    @ public static Map<QueryType, QueryCommand> createQueryMap(List<QueryCommand> commandList) {
        Map<QueryType, QueryCommand> queryMap = new HashMap<>();
        for (QueryCommand command : commandList) {
            queryMap.put(command.getQueryType(), command);
        }
        return Collections.unmodifiableMap(queryMap);
    }
}
```

- Item 2: Consider a builder when faced with many constructor parameters: the builder pattern was used with the JsonBuilder that builds JSON objects and used it in the BroadcastRequestBuilder .

```
public ApiResponse createDocument(String databaseName, String collectionName, JSONObject document, String isBroadcasted) {
    JSONObject jsonObject = JsonBuilder.getBuilder()
        .add("queryType", QueryType.CREATE_DOCUMENT.toString())
        .add("databaseName", databaseName)
        .add("collectionName", collectionName)
        .add("document", document)
        .add("X-Broadcast", isBroadcasted)
        .build();
    System.out.println(isBroadcasted);
    return execute(jsonObject);
}

@Override
public void broadcastOperation(JSONObject details) {
    System.out.println("[INFO] Starting broadcasting database creation to others..");
    Database database = (Database) details.get("database");
    int originatingWorkerPort = (int) details.get("originatingWorkerPort");
    for (int i = 1; i <= affinityManager.getNumberOfNodes(); i++) {
        if (i == originatingWorkerPort) {
            System.out.println("[SKIP] Skipping broadcast to worker " + i + " (origin node)...");
            continue;
        }
        System.out.println("[BROADCAST] Broadcasting to worker " + i + "...");
        String url = "http://worker" + i + ":9000/api/createdb" + "/" + database.getDatabaseName();
        System.out.println("[BROADCAST] Broadcasting to URL: " + url);
        new BroadcastService.BroadcastRequestBuilder()
            .withUrl(url)
            .withMethod(HttpMethod.POST)
            .isBroadcasted(true)
            .broadcast();
    }
}
```

- Item 3: Enforce the singleton property with a private constructor or an Enum type, also spring by default have singleton on the bean level if you use annotations `@Service` `@Component` `@Controller`.

```

9 usages  Akram Jaghoub
public class InMemoryDatabase {

    6 usages
    private final Map<String, Database> databases;

    3 usages
    private static InMemoryDatabase instance;

    1 usage  Akram Jaghoub
    private InMemoryDatabase() { this.databases = new HashMap<>(); }

    4 usages  Akram Jaghoub
    public static InMemoryDatabase getInstance() {
        if (instance == null)
            instance = new InMemoryDatabase();
        return instance;
    }
}

```

- Item 4: Enforce noninstantiability with a private constructor Used in utility classes like the PassworHashing, DataTypeUtil, FileService classes.

```

2 usages  Akram Jaghoub *
public final class DataTypeUtil {

    no usages  new *
    private DataTypeUtil(){

    }

    1 usage  Akram Jaghoub
    public static Object castToDataType(String value, String collectionName, String propertyName) {
        String dataType = getDataType(collectionName, propertyName);
        return switch (Objects.requireNonNull(dataType).toUpperCase()) {
            case "STRING" → value;
            case "LONG" → Long.parseLong(value);
            case "DOUBLE" → Double.parseDouble(value);
            case "BOOLEAN" → Boolean.parseBoolean(value);
            default → null;
        };
    }
}

```

- Item 6: Avoid creating unnecessary objects: No new objects are created when we can reuse an existing one.
- Item 8: Avoid finalizers and cleaners: when dealing with file resources I used try with resources over finalizers.

```
1 usage  Akram Jaghoub
public static void writeJsonObjectFile(File file, JSONObject jsonObject) {
    try (FileWriter fileWriter = new FileWriter(file)) {
        fileWriter.write(jsonObject.toJSONString());
        fileWriter.flush();
    } catch (IOException e) {
        e.printStackTrace();
        System.err.println("Error writing to the file: " + e.getMessage());
    }
}
```

- Item 9: Prefer try-with-resources to try-finally: as mentioned in item 8.
- Item 15: Minimize the accessibility of classes and members: I wrote all class members as private, and they can only be accessed outside the class through getters and setters (encapsulation).

```
Akram Jaghoub
@Getter
@Setter
@AllArgsConstructor
public class ApiResponse {
    no usages
    private String message;
    no usages
    private HttpStatus status;
}
```

- Item 17: Minimize Mutability: which was done through utility classes I mentioned in item4 as I have made classes not extendable and most variables are final and private.
- item 27: Eliminate unchecked warnings

```
@SuppressWarnings("unchecked")
public JsonBuilder add(String key, Object value){
    jsonObject.put(key, value);
    return this;
}
```

- item 28: Prefer lists to arrays

```
1 usage  Akram Jaghoub
public List<String> readDBs() {
    List<String> inMemoryDBs = InMemoryDatabase.getInstance().readDatabases();
    Set<String> uniqueDatabases = new HashSet<>(inMemoryDBs);
    List<String> inFileDBS = DatabaseFileOperations.readDatabases();
    uniqueDatabases.addAll(inFileDBS);
    return new ArrayList<>(uniqueDatabases);
}
```

- Item 39: Prefer annotations to naming patterns

```
22 usages  Akram Jaghoub
@Getter
@Setter
public class Node {
    no usages
    private int nodeNumber;
    no usages
    private JSONObject nodeJsonObject;
    no usages
    private String nodeIP;
    no usages
    private boolean isActive;
}
```

- Item 40: Consistently use the Override annotation.

```
Akram Jaghoub
@Override
public QueryType getQueryType() { return QueryType.CREATE_DATABASE; }
```

- Item 49: Check parameters for validity

```
usage  Akram Jaghoub
public boolean isCustomerExists(Customer customer) {
    if (customer == null || customer.getAccountNumber() == null) { ←
        return false;
    }
    String path = FileServices.getUserJsonPath( fileName: "customers");
    File jsonFile = new File(path);
    if (jsonFile.exists()) {
        JSONArray jsonArray = FileServices.readJsonArrayFile(jsonFile);
    }
}
```

- Item 55. Return optionals judiciously: As mentioned in the clean code section.

- Item 58. Prefer for-each loop to traditional for loops Used the for each in many parts of my code.

```
JSONArray updatedArray = new JSONArray();
for (Object obj : jsonArray) {
    JSONObject jsonObject = (JSONObject) obj;
    String userAccountNumber = (String) jsonObject.get("accountNumber");
    if (!userAccountNumber.equals(accountNumber)) {
        updatedArray.add(jsonObject);
    }
}
writeJsonArrayFile(new File(documentPath), updatedArray);
```

- Item 78. Synchronize access to shared mutable data.
- Item 79. Avoid excessive synchronization: the program had synchronizations only when needed and where data sharing occurs (database, collection, document).

10.0 SOLID principles

10.1 Single Responsibility Principle

The Single Responsibility Principle states that a class should have only one reason to change, meaning that it should have a single job to do. This is achieved by designing classes with a single responsibility which was utilized in many classes of the system, such as the Command classes, Broadcast class, SchemaValidator, and many others.

10.2 Open Closed Principle

In the project, extensibility and flexibility are key. The Open/Closed Principle is applied through the QueryCommand interface. New database commands are easily added by implementing this interface, without altering existing code.

Additionally, the system leverages the Builder pattern, seen in the BroadcastRequestBuilder. It allows step-by-step creation of complex objects and supports different configurations. New options can be introduced by extending this builder without modifying existing methods.

10.3 Liskov Substitution Principle

Objects are replaceable with their subtypes without affecting the correctness. By defining the common interface QueryCommand, different query command implementations can be used interchangeably in the QueryManager without changing the behavior of the system. The execute method in each query command subtype is appropriately implemented to perform its specific task, and QueryManager uses polymorphism to execute these commands without knowing their concrete types.

10.4 Interface Segregation Principle

In the project, the QueryCommand interface adheres to the Interface Segregation Principle (ISP) by offering essential methods like getQueryType and execute, ensuring that all query commands have the necessary functionality. It also incorporates an optional broadcastOperation method (default), aligning with ISP's principle of not imposing unnecessary methods on classes.

10.5 Dependency Inversion Principle

The QueryCommand interface aligns with the Dependency Inversion Principle (DIP), ensuring that high-level modules in the database system, such as the QueryManager, rely on abstractions (interfaces) rather than concrete implementations.

11.0 Design Patterns

11.1 Creational

11.1.1 Singleton

The Singleton pattern is effectively implemented in the InMemoryDatabase class, ensuring that only one instance of this class exists in the program. This is achieved by making the constructor of the class private and providing a static method called "getInstance()" to access the single instance.

Moreover, in the context of Spring, the framework itself manages singleton beans by default when you annotate a class with annotations like @Service, @Component, or @Controller. This means that Spring will create and maintain a single instance of such classes within the Spring application context, ensuring that they are effectively singletons.

11.1.2 Factory

Utilized through the QueryCommandFactory class, to simplify the creation of different types of commands. By encapsulating the command creation logic in one place, it provides a convenient and flexible way to instantiate commands without exposing the details to the client code. This promotes code reuse, maintainability, and scalability by allowing the addition of new command types in the future.

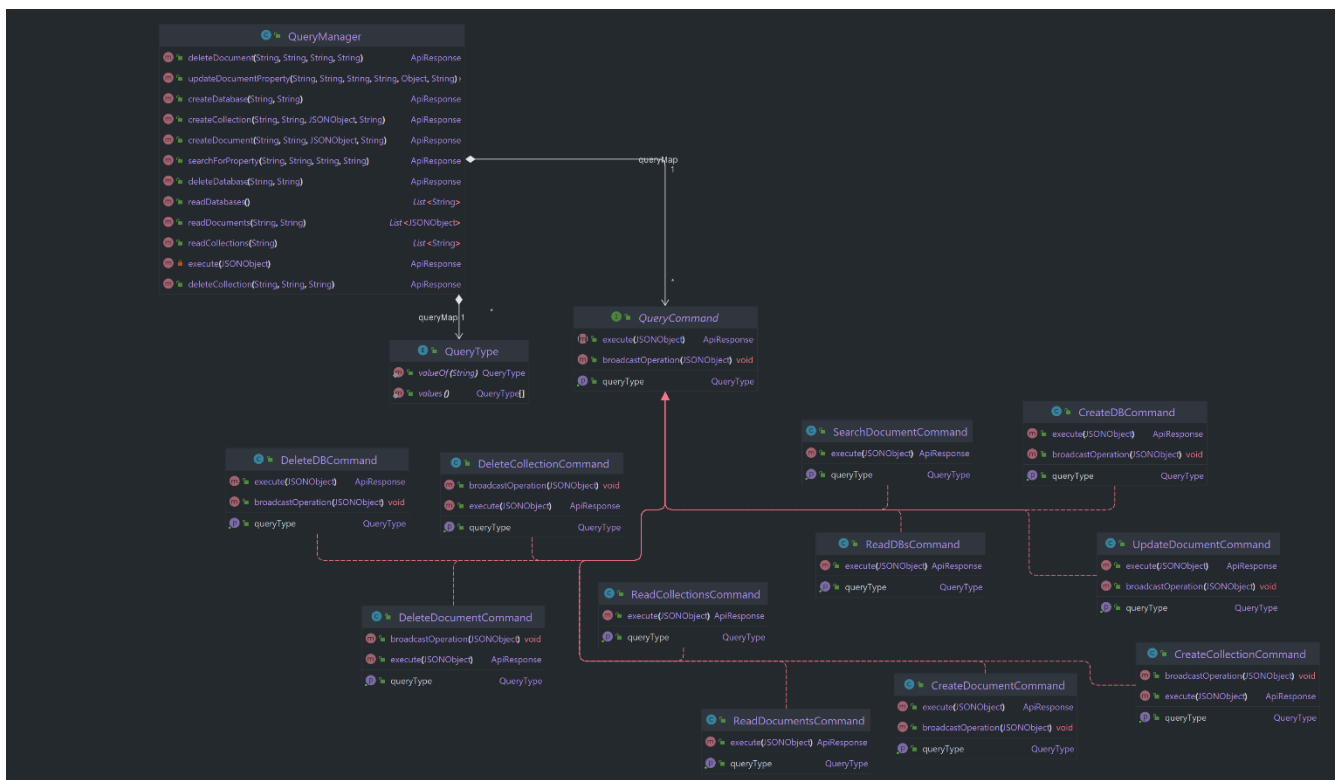
11.1.3 Builder

The Builder design pattern was effectively utilized in the **BroadcastRequestBuilder** and **JSONBuilder** classes. This pattern provides a structured and clean approach for constructing complex objects, allowing you to set various attributes or properties step by step. In the case of the **BroadcastRequestBuilder**, it facilitates the creation of HTTP request objects with different configurations, making it more versatile and readable. Similarly, the **JSONBuilder** simplifies the creation of JSON objects by offering a fluent and method-chaining interface, enhancing code clarity and maintainability.

11.2 Behavioral

11.2.1 Command

The Command design pattern is applied effectively in the project, where database operations are encapsulated as objects. This approach promotes flexibility and extensibility. By implementing concrete command classes like **CreateDatabaseCommand**, new operations can be added seamlessly without altering existing code. This pattern is also extended to support broadcasting operations, improving scalability and maintainability.



12.0 DevOps practice

12.1 Docker

To fulfill the requirement of representing each node as a virtual machine, I utilized Docker and its network features to establish communication among the nodes. By leveraging Docker's containerization technology, I was able to create separate virtual environments for each node, and Docker's networking capabilities allowed these nodes to communicate with each other seamlessly.

12.2 Maven

In the project, Maven plays a pivotal role in managing dependencies, building executable JAR files, and creating Docker images. It streamlines the development process, ensuring that all required libraries and resources are seamlessly integrated

12.3 Git and Github

which were used in every process of building the project.

12.4 Spring

To construct a language-independent database that anyone can use, I utilized Spring-Boot as the transaction layer. To achieve this, I developed my REST API project and incorporated the Maven project as a library, then used Spring-Boot's Restful controller to handle user requests.

13.0 User Manual

To start the system we first run the docker:

docker-compose up

By doing this now you can access the interface through:

<http://localhost:8080/login>

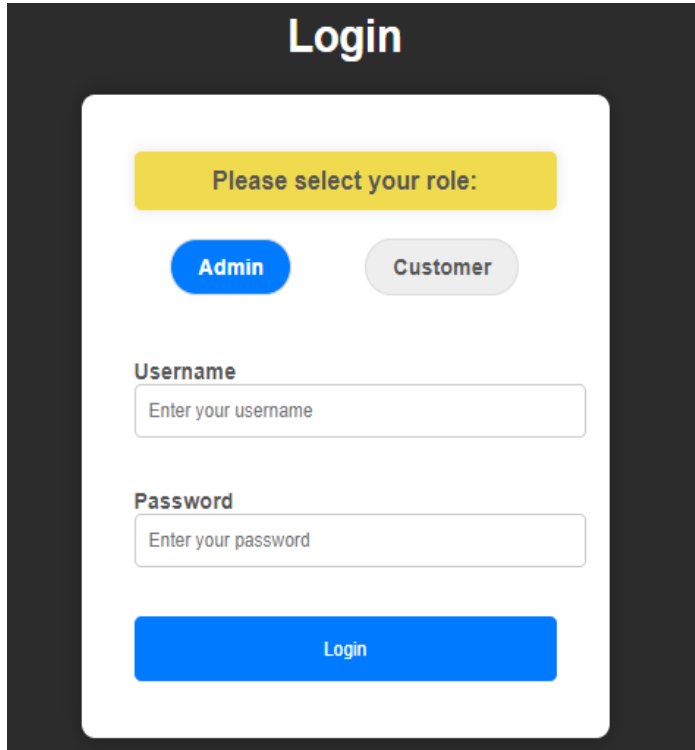
- **Login page:**

As mentioned above here we have the login and the authentication process to access the admin's interface use:

Username: admin

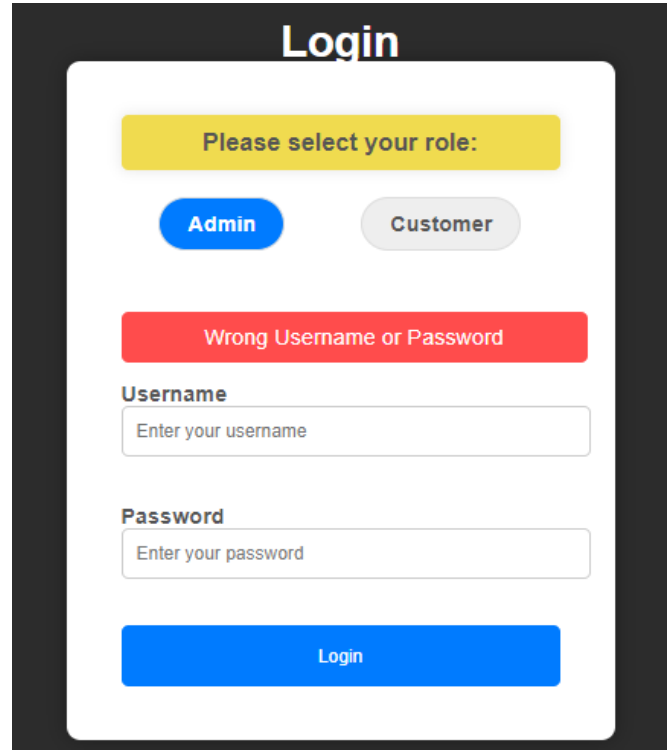
Password: admin@12345

For the customer interface it is only accessible if an account existed with that customer (document) and same validations occur but if both username and password are wrong a message shown as **“customer does not exist please contact your bank or try again”**



The image shows a login form titled "Login" on a dark background. The form is white and contains a yellow box with the text "Please select your role:". Below this are two buttons: "Admin" (blue) and "Customer" (grey). Underneath are two input fields: "Username" with the placeholder "Enter your username" and "Password" with the placeholder "Enter your password". At the bottom is a blue "Login" button.

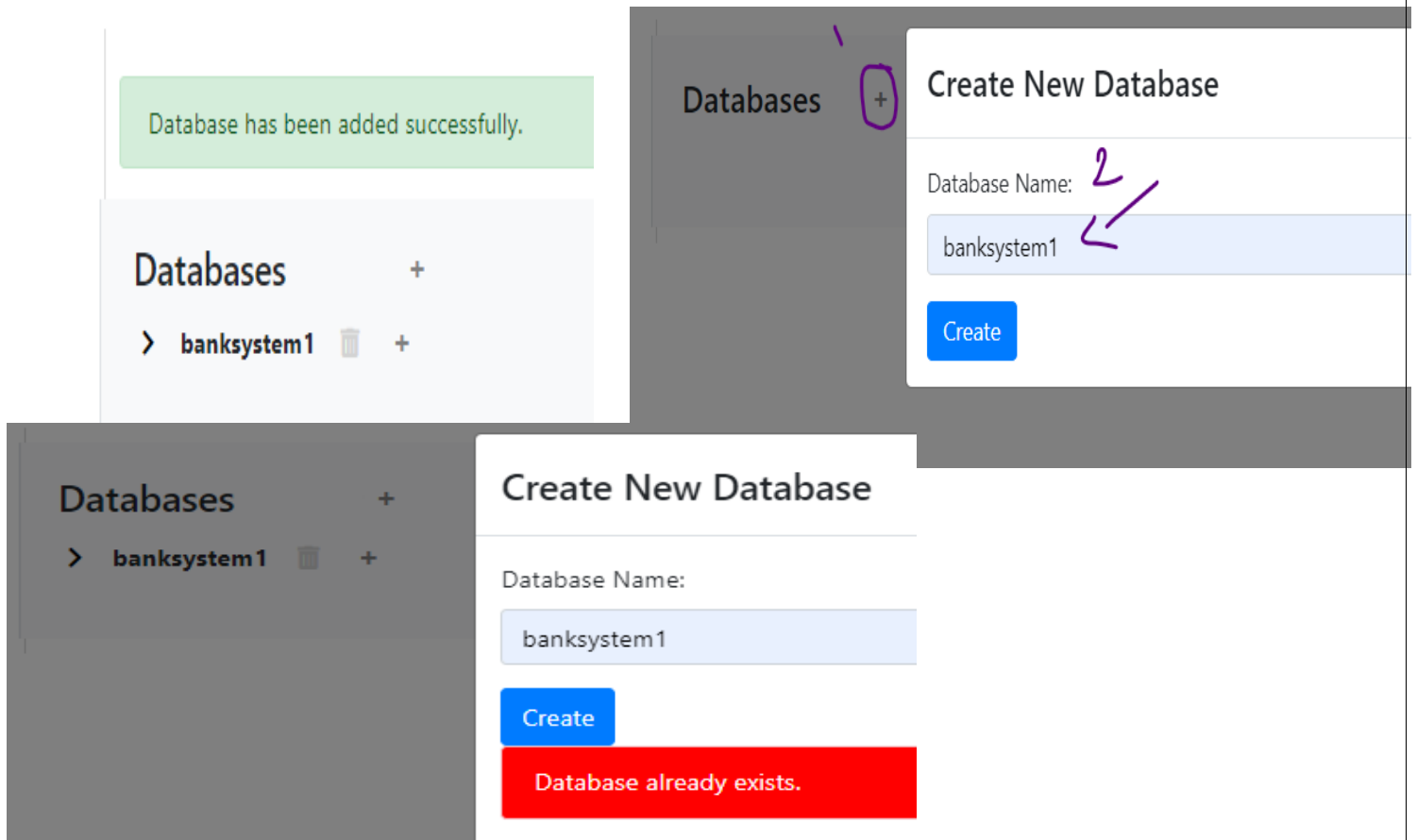
Success



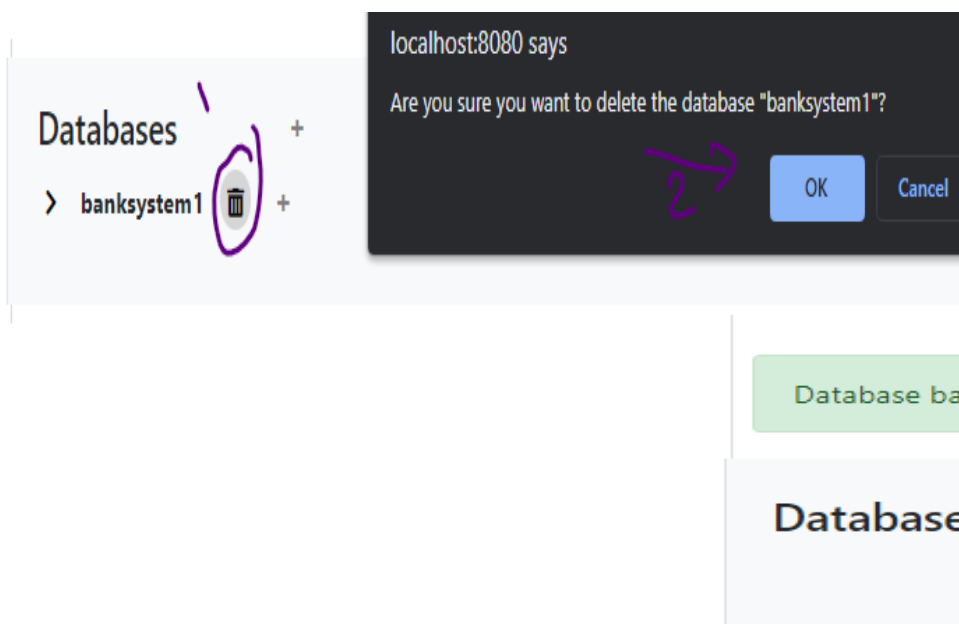
The image shows a login form titled "Login" on a dark background. The form is white and contains a yellow box with the text "Please select your role:". Below this are two buttons: "Admin" (blue) and "Customer" (grey). A red box with the text "Wrong Username or Password" is displayed above the input fields. The input fields are "Username" with the placeholder "Enter your username" and "Password" with the placeholder "Enter your password". At the bottom is a blue "Login" button.

Failure

- Admin dashboard
 - Creating a database



- deleting a database



- **Creating a collection**

The top row shows a successful collection creation. On the left, a green notification bar states "Collection created successfully." In the center, the "Databases" sidebar shows "banksystem1" expanded with "accounts1" listed below it. On the right, the "Create New Collection in banksystem1" dialog is shown with "Collection Name: accounts1" and a blue "Create" button. A purple arrow points to the "Collection Name" field.

The bottom row shows an attempt to create a collection that already exists. The "Databases" sidebar on the left shows "banksystem1" expanded with "accounts1" listed below it. The "Create New Collection in banksystem1" dialog is shown with "Collection Name: accounts1" and a blue "Create" button. A red error bar at the bottom states "Collection already exists." A purple arrow points to the "Collection Name" field.

- **Deleting a collection**

The top row shows the deletion process. On the left, the "Databases" sidebar shows "banksystem1" expanded with "accounts1" listed below it. A purple circle highlights the trash icon next to "accounts1". On the right, a dark gray dialog box titled "localhost:8080 says" asks "Are you sure you want to delete the collection 'accounts1'?" with "OK" and "Cancel" buttons. A purple arrow points to the "OK" button.

The bottom row shows the successful deletion. A green notification bar states "Collection accounts1 has been deleted in banksystem1 database". Below it, the "Databases" sidebar shows "banksystem1" expanded with "accounts1" listed below it.

- Creating a document (account)

Databases

▼ banksystem1

accounts1

+

+

Open New Account

2 ✓

Account Number:

12345

Client Name:

Akram Jaghoub

Balance:

500

Account Type:

Deposit

Has Insurance:

☒ Yes ☐ No

Password:

.....

Open Account

Account created successfully.

Databases

> banksystem1

accounts1

+

+

Displaying documents 1 - 1

accountNumber: 12345

clientName: Akram Jaghoub

balance: 20

accountType: DEPOSIT

hasInsurance: Yes

password:

5994471abb01112afcc18159f6cc74b4f511b99806da59b3caf5a9c173cacfc5

_id: ObjectId("f7fc9617-db9b-4080-a521-e0553a1d0b31")

Delete Account

You are on the first document

multiple docs →

prev doc next doc

Displaying documents 1 - 2

accountNumber: 20002421
clientName: Fahed Jubair
balance: 100000
accountType: DEPOSIT
hasInsurance: Yes
password:
5994471abb01112afcc18159f6cc74b4f511b99806da59b3caf5a9c173cacfc5
_id: ObjectId("b23d1b64-2a2b-4c1a-9c4f-63cad9448a21")

Delete Account

You are on the first document

Databases

> banksystem1

accounts1

Open New Account

Account Number:

20002421

Client Name:

Motasem Aldiab

Balance:

50000

Account Type:

Loan

Has Insurance:

☒ Yes ☐ No

Password:

Open Account

An account with the same account number already exists.

- deleting a document (account)

localhost:8080 says

Are you sure you want to delete the document with account number 12345?

OK Cancel

Displaying documents 1 - 1

accountNumber: 12345

clientName: Akram Jaghoub

balance: 300

accountType: DEPOSIT

hasInsurance: Yes

password:
5994471abb01112afcc18159f6cc74b4f511b99806da59b3caf5a9c173cacfc5

_id: ObjectId("8f0c5271-3ecc-4004-ba69-daa862f4ee35")

Delete Account

Account has been deleted.

Databases

▼ banksystem1 +

account1 +

No documents exist yet. Add a new one to see what happens.

- **Customer Dashboard**

As mentioned before accessed only in case the admin created a document (account)

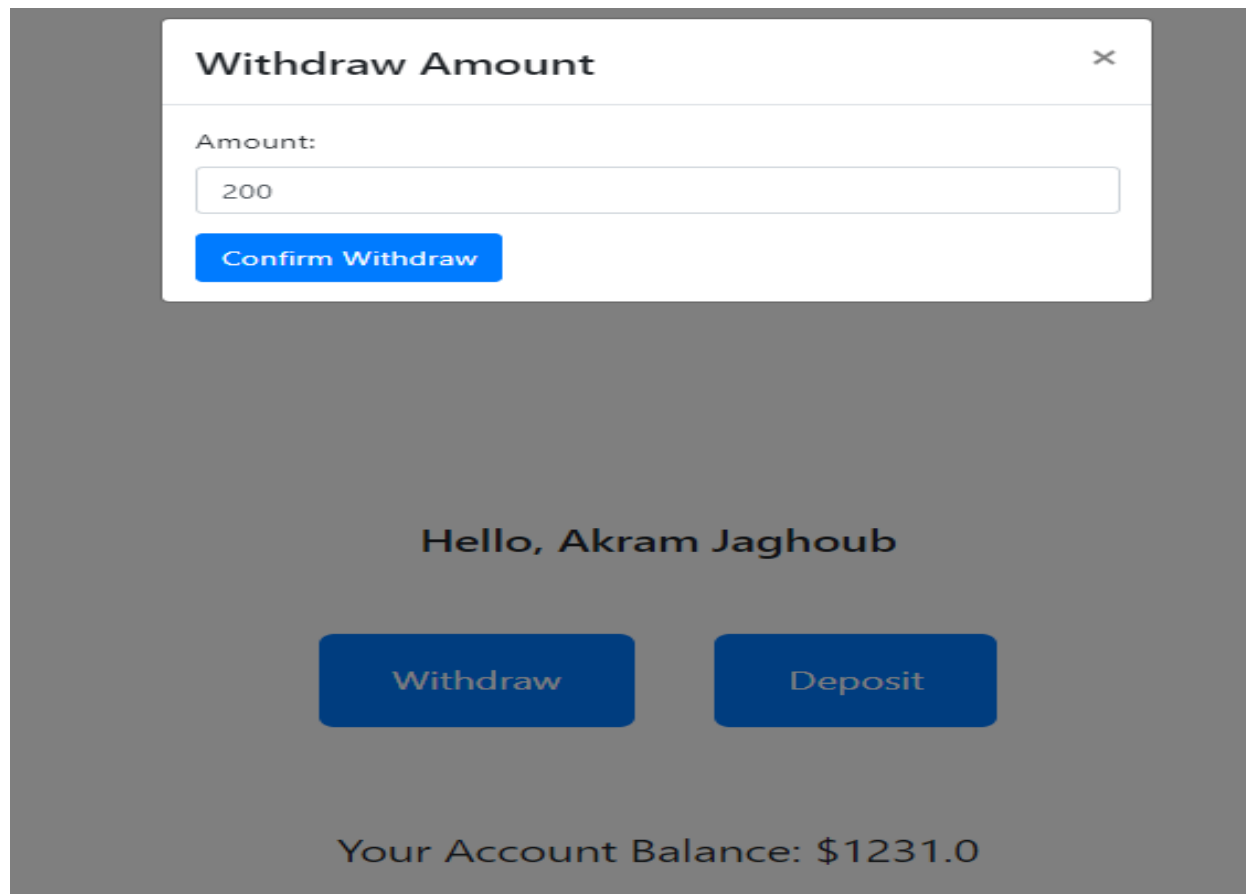
Hello, Akram Jaghoub

Withdraw

Deposit

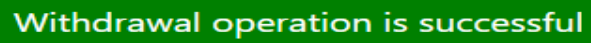
Your Account Balance: \$1231.0

- **Withdraw an amount (update document)**



The screenshot shows the Customer Dashboard with a modal titled "Withdraw Amount" open. The modal contains a label "Amount:", a text input field with the value "200", and a blue button labeled "Confirm Withdraw". The background dashboard is dimmed and shows the greeting "Hello, Akram Jaghoub", two buttons "Withdraw" and "Deposit", and the account balance "Your Account Balance: \$1231.0".

Withdrawal operation is successful

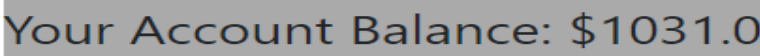


Hello, Akram Jaghoub

Withdraw

Deposit

Your Account Balance: \$1031.0



Withdraw Amount

x

Amount:

2000

Please withdraw an amount less than your balance.

Confirm Withdraw

Your Account Balance: \$1031.0

- Deposit an amount (update document)

Deposit Amount

Amount:

500

Confirm Deposit

Hello, Akram Jaghoub

Withdraw

Deposit

Your Account Balance: \$1031.0

Deposit operation is successful

Hello, Akram Jaghoub

Withdraw

Deposit

Your Account Balance: \$1531.0