((Report))

Capstone Project

Name: Zain Alabdeen Adelelah Aburayya

Contents

Introduction:

Different between NoSQL and SQL :-

SQL databases use tables with a fixed schema to store structured data and are best for applications with complex querying needs. NoSQL databases use flexible data models to store unstructured or rapidly changing data and are better for applications that need to scale and handle large amounts of data.

NoSQL types :-

The main types of NoSQL databases are document-based, key-value, graph-based, column-family, and object-oriented databases. Each type has its own unique strengths and is suited for different use cases , I built a document-based database using JSON objects, so all data will be stored in file system.

DB Implementation

Overall:-

In my view, implementing a document-based NoSQL database using the file system for data storage and unique indexes for databases, collections, and documents is a compelling approach. This approach offers several benefits, such as efficient and scalable data storage. Since NoSQL databases can handle unstructured and semi-structured data, using a document-based approach can be advantageous. Storing data as documents in the file system allows for fast and easy access to the data, which is especially useful when dealing with large data sets.

Using unique indexes for databases, collections, and documents can further improve performance and scalability. These indexes allow for fast lookup of data and can help optimize query performance. By using indexes, the database can quickly locate specific documents, even when the database contains millions of documents.

Organizing collections with a separate schema folder can also enhance consistency and organization. This can help prevent data inconsistencies, ensure data quality, and make it easier to maintain and update the database. Having a well-organized schema can also help reduce development time and effort by providing a clear understanding of the database structure.

It's worth noting that while the approach outlined above can be effective, there are many different ways to implement a document-based NoSQL database. The specific approach chosen will depend on the use case, data model, and requirements. Therefore, it's important to carefully evaluate different options and select the one that best meets your needs.

Indexing: -

As a developer, schema validation is a process I use to ensure that data adheres to a defined schema or structure. This involves checking that the data meets the rules and requirements specified in the schema, such as the expected fields and data types. By validating the schema, I can ensure that the data is properly formatted and that it conforms to the expected structure. This helps to prevent invalid data from being inserted into the database, improving data consistency and accuracy. Ultimately, schema validation is a critical step in ensuring that the data is reliable and meets the required standards, which is essential for developing robust and dependable software systems.

```java
package com.example.db.index;

import lombok.AllArgsConstructor;
import lombok.EqualsAndHashCode;
import lombok.Getter;
import lombok.Setter;

@Setter
@Getter
@AllArgsConstructor
@EqualsAndHashCode
public class Indexing {
    private String database;
    private String collection;
    private String property;
    private String value;
}
```

HashIndexing class:

This is a Java class named HashIndexing that represents a hash index for a database. It is responsible for indexing the data based on a specific property in the data. The class contains several methods to add, retrieve, and delete data from the index.

The HashMap hashMap variable is used to store the indexed data. The Workers class is used to perform database operations. The Database class is used to get the path of the database.

The getInstance() method returns an instance of the HashIndexing class.

The getByProperty() method takes four arguments: db_name, collection_name, prop, and value. It returns a string representation of the list of integers that match the given property and value.

The isExistId() method takes three arguments: db_name, collection_name, and json. It returns a boolean value indicating whether the given id exists in the database.

The addToIndexing() method takes three arguments: db_name, collection_name, and json. It adds the given data to the index based on the specified property and value.

The size() method takes two arguments: db_name and collection_name. It returns the number of documents in the specified collection.

The deleteFromIndexing() method takes four arguments: db_name, collection_name, prop, value, and index. It deletes the specified document from the index.

The deleteById() method takes four arguments: db_name, collection_name, value, and update. It deletes all documents that match the given id from the index.

The getIndex() method takes three arguments: db_name, collection_name, and id. It returns the index of the document in the index that matches the specified id.

The deleteAllProperty() method takes four arguments: db_name, collection_name, id, and update. It deletes the document from the index and the data file.

The deleteFromFile() method takes three arguments: db_name, collection_name, and index. It deletes the specified document from the data file.

All methods use the Jackson library to parse and manipulate JSON data. The @SneakyThrows annotation is used to indicate that the method throws checked exceptions, but they are automatically caught and rethrown as unchecked exceptions.

Schema: -

In our NoSQL database implementation, we have incorporated a schema generation process to ensure data consistency and validity. When a user wants to add a new collection, they input the desired fields for that collection.

Based on this input, our system generates a schema specific to that collection. This schema is then used to validate incoming documents and ensure that they conform to the expected structure.

Each collection has its own schema, which enables us to create appropriate indexes for each field, allowing for efficient querying of data. Having a separate schema for each collection also helps prevent the introduction of unexpected or invalid data into the database.

Overall, our schema generation approach is a common one in document-based NoSQL databases and has proven to be an effective way to ensure data consistency and validity.

In this implementation, a general schema for all databases or collections was not used. Instead, a unique schema is generated for each collection based on the user's input of desired fields. This technique was chosen to simplify the user experience and make the system more user-friendly.

The ability for users to input desired fields for each collection offers greater control over the structure and types of data stored in the database. This approach also offers greater flexibility and customization as opposed to using a single schema for all collections.

Overall, the schema generation technique implemented is a user-friendly approach that prioritizes ease of use and flexibility.

Data-Structure

As mentioned previously, the indexing technique utilized in this implementation involved using a Hash Map. The Hash Map was used to store and quickly retrieve the property and value of data from the JSON file. In addition, Object Mapper was used to facilitating the retrieval of data from the JSON file.

To enable the user to easily view the databases and collections, a List was implemented for storing this information. This feature is accessible through the demo application, which provides a simple and user-friendly interface for interacting with the database.

By incorporating these features, our NoSQL database implementation aims to provide a convenient and user-friendly experience for users. The use of a Hash Map and Object Mapper enables efficient data retrieval and indexing, while the List implementation provides a simple and accessible way for users to view their data.

I am using the HashMap data structure to manage the locks, which allows me to map lock names to their corresponding lock objects efficiently and in a thread-safe manner. The HashMap is a convenient and effective way to ensure data consistency and thread safety in multi-threaded applications.

```
private static HashMap<String,Object> locks;
```

```
HashMap<Indexing, List<Integer>> hashMap = new HashMap<>();
private final Workers workers = new Workers();
private static HashIndexing instance = null;
public static HashIndexing getInstance(){
    if( instance == null)
        instance = new HashIndexing();
    return instance;
}
```

```
public List<String> getList(){return database.DatabaseList();}
```

Multithreading and locks

I have created a class that provides a lock for every collection in each database on demand. The purpose of this class is to ensure that multiple threads cannot access the same collection simultaneously and cause data inconsistencies.

Whenever a thread needs to access a collection, it can request a lock for that collection from the lock class. The lock class checks if a lock already exists for that collection, and if not, it creates a new lock for it. The lock is then acquired by the requesting thread, and any other threads that try to acquire the same lock will be blocked until the lock is released.

This approach ensures that the collections in each database are accessed in a synchronized and thread-safe manner. By creating locks only when they are needed, the class minimizes the overhead associated with locking and allows for more efficient use of system resources.

Overall, this class provides a convenient and efficient way to synchronize access to collections in multiple databases, ensuring that data is consistent and thread-safe.

This is a Java class called "Lock" that provides a thread-safe mechanism for obtaining locks on named objects.

The class has a private static field called "locks", which is a HashMap that maps lock names (strings) to lock objects (Objects).

```
public class Lock {
    private static HashMap<String,Object> locks;
    private static Lock instance = null;

    private Lock(){
        locks = new HashMap<>();
    }

    public static Lock getInstance(){
        if(instance == null)
            instance = new Lock();
        return instance;
    }

    public Object getLock(String lockName){
        if(!locks.containsKey(lockName))
            initLock(lockName);
        return locks.get(lockName);
    }

    private void initLock(String lockName){
        Object lock = new Object();
        locks.put(lockName,lock);
    }

}
```

For example :

- If there are multiple users attempting to add a document to the same collection at the same time, this can create a race condition. This occurs when multiple threads try to access the same resource simultaneously, leading to inconsistent and unpredictable behaviour.

```java
@SneakyThrows
public String addDocument(String db_name , String collection_name , String json, String update){
    String path = Database.getInstance().getDB_PATH() + db_name + "/" + collection_name + ".json";
    File file = new File(path);
    boolean isUpdate = update.equalsIgnoreCase("update");
    Object lock = Lock.getInstance().getLock(db_name + "-" + collection_name);
    synchronized (lock){
        if (!file.exists()) {
            return "file doesn't exist ...";
        }
        if (!jsonValidation.schemaValidator(db_name, collection_name, json)) {
            return "failed to add document...";
        }
        if (HashIndexing.getInstance().isExistId(db_name, collection_name, json)) {
            return "Id is already exist ...";
        }

        ObjectMapper objectMapper = new ObjectMapper();
        ObjectNode root = (ObjectNode) objectMapper.readTree(new File(path));
        ArrayNode data = (ArrayNode) root.get("data");
        if (data == null) {
            data = objectMapper.createArrayNode();
            root.set("data", data);
        }
        JsonNode jsonNode = objectMapper.readTree(json);
        addToIndexing(db_name, collection_name, json);
        data.add(jsonNode);
        FileWriter writer = new FileWriter(path);
        objectMapper.writeValue(writer, root);
        writer.close();
        if (isUpdate)
            workers.buildDocument(db_name, collection_name, json);
        Affinity.getInstance().updateAffinity();
    }
    return "success add a document ... ";
}
```

Data Consistency issues in the DB

Schema validation is an essential process that ensures that data adheres to a defined structure or schema. Essentially, schema validation checks that data meets the rules and requirements outlined in the schema. By doing so, it confirms that the data is properly formatted, and it contains the expected fields and data types. As a result, validating the schema helps to prevent the insertion of invalid data into the database. This improves data consistency and accuracy, ensuring that the data is reliable and can be used with confidence. Therefore, schema validation is crucial for maintaining the integrity of the data and ensuring that it meets the required standards.

```
@SneakyThrows
public boolean schemaValidator(String db_name , String collection_name , String json){
    String path = Database.getInstance().getDB_PATH() + db_name + "/schema/schema_"+collection_name+".json";
    JsonSchemaFactory jsonSchemaFactory = JsonSchemaFactory.getInstance();
    JsonSchema jsonSchema = jsonSchemaFactory.getSchema(readSchemaFile(path));
    ObjectMapper objectMapper = new ObjectMapper();
    JsonNode jsonNode = objectMapper.readTree(json);
    Set<ValidationMessage> validationMessageSet =jsonSchema.validate(jsonNode);

    return validationMessageSet.isEmpty();
}

public String readSchemaFile(String path){
    try {
        return new String(Files.readAllBytes(Paths.get(path)));
    } catch (IOException e) {
        throw new RuntimeException(e.getMessage());
    }
}
```

Node hashing and load balancing

To implement load balancing, I created a JSON file with a single property called "affinity", which is assigned a number value based on the number of workers. For instance, if there are 4 workers, then the affinity value will be between 0 and 3. This is how load balancing is achieved in the system.

To ensure smooth interaction and synchronization between the different components of the system, I utilized the HTTP protocol as the primary communication protocol. This was due to its numerous advantages, including widespread adoption, making it easy to integrate into the system, and having clear, standardized methods and status codes for easy monitoring and debugging of network traffic.

In addition, the HTTP protocol supports a variety of data exchange formats, including JSON and XML, allowing for flexible data transmission. Moreover, it is designed to work seamlessly across different networks and devices, making it ideal for communication between nodes in a distributed system.

Overall, HTTP protocol served as a reliable and efficient communication mechanism in the system, enabling easy monitoring and updating of the various components as needed.

```java
@SneakyThrows
public boolean checkWorkersDatabase(String db_name){
    for(String url : urls) {
        if(("http://w" +Affinity.getInstance().getValue() + ":8080").equalsIgnoreCase(url))
            continue;
        URL dist = new URL(url + "/worker/db/find/" + db_name);
        HttpURLConnection conn = (HttpURLConnection) dist.openConnection();
        conn.setRequestMethod("GET");
```

Worker class :

This is a Java code for a service called "Workers" that performs various operations related to a database.

The code contains methods for checking whether a database or a collection exists in any of the worker nodes, creating a new database, creating a new collection within a database, and adding a new document to a collection. The worker nodes are specified as an array of URLs.

The code uses HTTP requests to communicate with the worker nodes. It sets the appropriate request headers, such as "Content-Type" and "TOKEN", and sends the request using the "HttpURLConnection" class. The response from the worker nodes is then read and processed.

The code also includes the "@Service" annotation, which is a Spring Framework annotation that marks the class as a service and allows it to be autowired into other Spring components.

Security issues

To ensure secure access to the system, I implemented a login and signup feature. For logging in, the user is required to provide their username and token, which is validated after every request. To sign up, the user needs to enter an admin name and token for authentication. This approach helps to prevent unauthorized access to the system and ensures that only authorized users can perform actions within it.

To enhance the security of the system, I utilized two types of tokens, Base64 and UUID, to reduce the likelihood of two users having the same token at the same time. For login, users are required to provide their username and token, with validation taking place after every request. As for signing up, users must enter an admin name and token for authentication purposes.

```java
package com.example.bootstrapping.token;

public interface Token {
    public String generateNewToken();
}
```

```java
public class Base64Token implements Token{
    private static final SecureRandom secureRandom = new SecureRandom();
    private static final Base64.Encoder base64Encoder = Base64.getUrlEncoder();
    @Override
    public String generateNewToken() {
        byte[] randomBytes = new byte[24];
        secureRandom.nextBytes(randomBytes);
        return base64Encoder.encodeToString(randomBytes);
    }
}
```

Clean Code principles

Clean Code principles are essential guidelines and best practices that can help software developers create code that is easy to read, understand, and maintain. The concept of Clean Code was introduced by Robert C. Martin in his book "Clean Code: A Handbook of Agile Software Craftsmanship." The principles of Clean Code emphasize the importance of code readability, maintainability, and efficiency.

One of the most fundamental principles of Clean Code is the use of clear and meaningful names for variables, functions, and classes. By using descriptive names that accurately convey their purpose, developers can significantly improve the readability of the code. It's also important to avoid cryptic abbreviations or single-letter variable names, as they can make the code harder to understand and maintain. Another crucial principle of Clean Code is avoiding the duplication of code. Duplicated code can lead to maintenance issues and bugs, and it can make the codebase harder to maintain. Therefore, developers should always look for opportunities to refactor duplicate code into reusable functions or classes. This not only improves the efficiency of the code but also makes it more manageable.

```java
@SneakyThrows
private void addAdminToJson(AuthenticationModel model){
    String path = "./src/main/resources/admin.json";
    ObjectMapper mapper = new ObjectMapper();
    File file = new File(path);
    ObjectNode root = mapper.readValue(file, ObjectNode.class);
    ArrayNode users = (ArrayNode) root.get("admin");
    String json = mapper.writeValueAsString(model);
    JsonNode jsonNode = mapper.readTree(json);
    users.add(jsonNode);
    FileWriter writer = new FileWriter(path);
    mapper.writeValue(writer, root);
    writer.close();
    Affinity.getInstance().updateAffinity();
}
```

The Single Responsibility Principle is another critical principle of Clean Code. This principle suggests that a function, class, or module should have only one responsibility. By following this principle, developers can make the code easier to comprehend and maintain, reducing the risk of introducing bugs and making it easier to update or modify the code.

Simplicity is also a key principle of Clean Code. Developers should strive to keep the code as simple as possible, avoiding unnecessary layers of abstraction and complexity that can make the code harder to understand and maintain. This means avoiding overcomplicating the

code, as simplicity helps to reduce the likelihood of introducing bugs and makes it easier to extend or modify the codebase.

In summary, Clean Code principles provide developers with a set of guidelines and best practices that help create code that is easy to read, understand, and maintain. By following these principles, developers can produce software that is efficient, reliable, and robust.

In regards to the use of multiple languages in one source file, the ideal is for a source file to contain one language. However, in some cases, it may be necessary to use more than one language. In such cases, I should aim to minimize the number and extent of extra languages used, to keep the code as readable and understandable as possible. For instance, in my project, I can keep every language separate from each other to maintain consistency, readability, and understandability of the codebase.

When naming conventions are inconsistent, it can lead to confusion and make it harder to understand the code. Inconsistent naming may indicate a lack of clarity and can lead to errors when code is modified or extended.

Another Clean Code principle is to encapsulate conditionals. Boolean logic can be difficult to understand when presented in the context of an if or while statement. To improve code readability and maintainability, it is recommended to extract functions that explain the intent of the conditional. This can make it easier to understand the code and make modifications or extensions without introducing errors. By encapsulating conditionals, code becomes more readable, understandable, and maintainable, which is crucial for writing robust and reliable software.

```java
@PostMapping("/delete")
public String postDeleteCollection(@RequestParam("db_name") String db_name,
                                   @RequestParam("collection_name") String collection_name,
                                   HttpSession session , Model model){
    User login = (User) session.getAttribute("login");
    if(login == null)
        return "login";
    if(!collectionService.isExist(db_name,collection_name,session) ||
            !databaseService.isExist(db_name,session)){
        model.addAttribute("result","no database or collection");
        return "response";
    }
    String response = collectionService.deleteCollection(db_name,collection_name,session);
    if(!response.equals("delete-success")){
        model.addAttribute("result",response);
        return "response";
    }
    return "delete-collection";
}
```

Effective Java

In my perspective, it's essential to avoid creating unnecessary objects in my code. I believe that every object created should serve a clear purpose and contribute to making the program faster and more efficient. Additionally, unnecessary objects can consume valuable memory and lead to performance issues.

I also think that it's crucial to avoid using finalizers and cleaners in my code. While they may seem useful at first, finalizers and cleaners can be unpredictable, dangerous, and often unnecessary. One major drawback of these methods is that there is no guarantee that they will be executed

promptly or in the correct order, which can lead to unexpected behavior and bugs in the program.

Instead of relying on finalizers and cleaners, I prefer to use explicit resource management techniques, such as try-with-resources statements, to ensure that resources are properly cleaned up and released when they are no longer needed. This can help prevent memory leaks and improve the overall stability and reliability of the program.

SOLID-principles

The Single Responsibility Principle: -

The Single Responsibility Principle is a fundamental principle in software engineering that emphasizes the importance of keeping each module, class, or function responsible for a single aspect of a program's functionality. Essentially, this means that each component should have one clear and specific task or responsibility, and it should be encapsulated within that component.

For instance, if we have a function for deleting a collection, it should have only one responsibility, which is to delete the collection. It should not have any additional responsibilities or tasks that could potentially complicate the function or reduce its clarity. This approach can make the code easier to read, understand, and maintain, and it can help to prevent bugs or errors caused by complex and confusing code.

By applying the Single Responsibility Principle to our code, we can create more modular, maintainable, and scalable software that is easier to test, debug, and extend over time.

```
@Async
public Future<Boolean> deleteCollection(String db_name, String collection_name, String update){
    String path = Database.getInstance().getDB_PATH() + db_name + "/" + collection_name + ".json";
    File file = new File(path);
    if(file.exists()){
        if(update.equalsIgnoreCase("update"))
            workers.deleteCollection(db_name,collection_name);
        Affinity.getInstance().updateAffinity();
        file.delete();
        path = Database.getInstance().getDB_PATH() + db_name + "/schema/schema_" + collection_name +".json";
        file = new File(path);
        return CompletableFuture.completedFuture(file.delete());
    }
    return CompletableFuture.completedFuture(false);
}
```

Interface Segregation principles: -

The Interface Segregation Principle states that I should not force my clients to depend on interfaces they do not use. I should design my interfaces in a way that ensures that implementing classes only need to implement the methods that they actually need, and not any extra or unnecessary methods.

In my example, the TokenGenerator interface specifies the methods that are necessary for generating tokens, and the UUIDToken and Base64Token classes implement those methods in their own unique ways. By separating the interface from the implementation, I am following the Interface Segregation Principle and creating more flexible and maintainable code.

This approach can make it easier for me to add new token generation methods in the future without affecting existing code, and it can also make testing and debugging the code simpler and more straightforward.

Overall, following the SOLID principles can help me improve the quality, readability, and maintainability of my code.

```java
package com.example.bootstrapping.token;

import java.security.SecureRandom;
import java.util.Base64;

public class Base64Token implements Token{
    private static final SecureRandom secureRandom = new SecureRandom();
    private static final Base64.Encoder base64Encoder = Base64.getUrlEncoder();
    @Override
    public String generateNewToken() {
        byte[] randomBytes = new byte[24];
        secureRandom.nextBytes(randomBytes);
        return base64Encoder.encodeToString(randomBytes);
    }
}
```

Design-Patterns

The Singleton design pattern has been implemented for both the HashIndexing class and the Affinity class in my codebase. This means that there is only one instance of each class throughout the program, ensuring that any changes made to their respective functionalities are consistent across all instances that access them. This approach can help to simplify the management of these classes and prevent conflicts that may arise from multiple instances, The Lock class also uses the Singleton pattern, ensuring that only one instance of the Lock object exists at any given time. This helps prevent conflicts and ensures thread safety when multiple threads need to access the same collection.

```
public class HashIndexing {
    HashMap<Indexing, List<Integer>> hashMap = new HashMap<>();
    private final Workers workers = new Workers();
    private static HashIndexing instance = null;
    public static HashIndexing getInstance(){
        if( instance == null)
            instance = new HashIndexing();
        return instance;
    }
```

```
public class Affinity {
    private final AffinityBroadcast broadcast = new AffinityBroadcast();

    private Affinity(){}
    private static Affinity instance = null;
    public static Affinity getInstance(){
        if(instance == null)
            instance = new Affinity();
        return instance;
    }
```

DevOps Practices

Docker: -

In this project, I have utilized Docker and containerization to build images for three distinct components - Bootstrap, WorkerDB, and WebApplication. By containerizing each component, I have been able to create a portable, isolated environment that can be easily tested and replicated across different platforms and environments. This approach has allowed me to

streamline the testing and deployment process and has been one of the most valuable skills that I have learned during my internship.

Using Docker and containerization also brings a number of other benefits to the project. For example, it allows us to package all of the dependencies required for each component within the image, making it easier to manage and update the software stack. Additionally, it enables us to maintain a consistent environment for development, testing, and production, reducing the risk of compatibility issues and errors.

Furthermore, the use of Docker and containerization can make it simpler to replicate services and deploy them in different environments. By encapsulating each component in its own container, we can easily move the application across different systems without worrying about configuration issues or dependencies. This makes it easier to scale the application and add new functionality as needed. Overall, the use of Docker and containerization has been a powerful tool for improving the efficiency and reliability of the project.

Dockerfile:-

All the Dockerfiles in this project use the same base image, which is the openjdk:19 image. This image contains the OpenJDK runtime environment, which is required to run Java applications.

The Dockerfile starts by using the FROM keyword to specify the base image. Then, the WORKDIR command sets the working directory for the Docker container to /app. This directory will be used to store the application code and other files required for running the application.

The COPY command copies the contents of the current directory to the /app directory in the Docker container. This includes the application code, dependencies, and any other required files.

The EXPOSE command specifies that the container will be listening on port 9090. This is required to allow incoming network connections to the container.

Finally, the CMD command specifies the command that should be executed when the container starts up. In this case, the command is to run the java executable with the -jar option and pass the path to the web application JAR file as an argument. This will start the web application and make it accessible via the exposed port on the Docker container.

```
FROM openjdk:19
WORKDIR /app
COPY . /app
EXPOSE 8080
CMD ["java", "-jar", "/app/target/db-0.0.1-SNAPSHOT.jar"]
```

```
FROM openjdk:19
WORKDIR /app
COPY . /app
EXPOSE 7070
CMD ["java", "-jar", "/app/target/bootstrapping-0.0.1-SNAPSHOT.jar"]
```

```
FROM openjdk:19
WORKDIR /app
COPY . /app
EXPOSE 9090
CMD ["java", "-jar", "/app/target/web-0.0.1-SNAPSHOT.jar"]
```

Docker-compose: -

docker-compose.yml

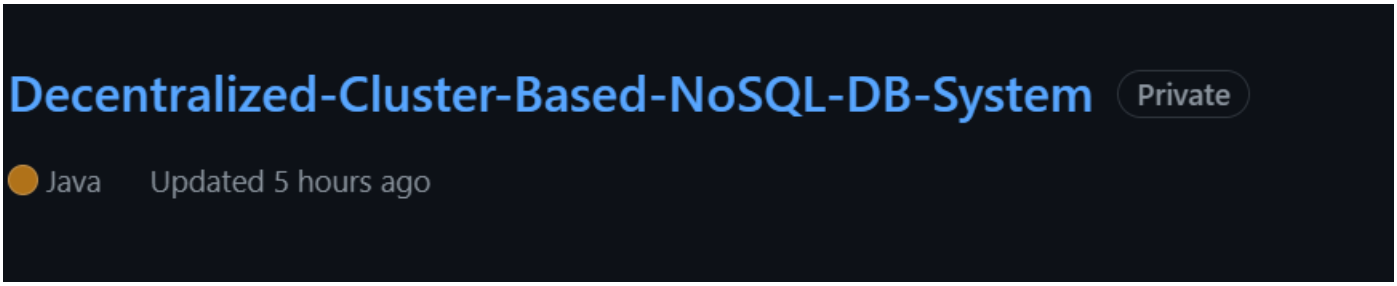
```yaml
version: '3'
services:

  worker0:
    build: ./db
    container_name: w0
    restart: always
    ports:
      - "8080:8080"

  ## here must be more worker ,
  ## just for the screenshot

  bootstrap:
    build: ./bootstrapping
    container_name: boot
    restart: always
    ports:
      - "7070:7070"
    depends_on:
      - worker0
      - worker1
      - worker2
      - worker3

  web:
    build: ./web
    container_name: web
    restart: always
    ports:
      - 9090:9090
    depends_on:
      - bootstrap
```

Command-Line to run the docker-compose :

" docker-compose up"

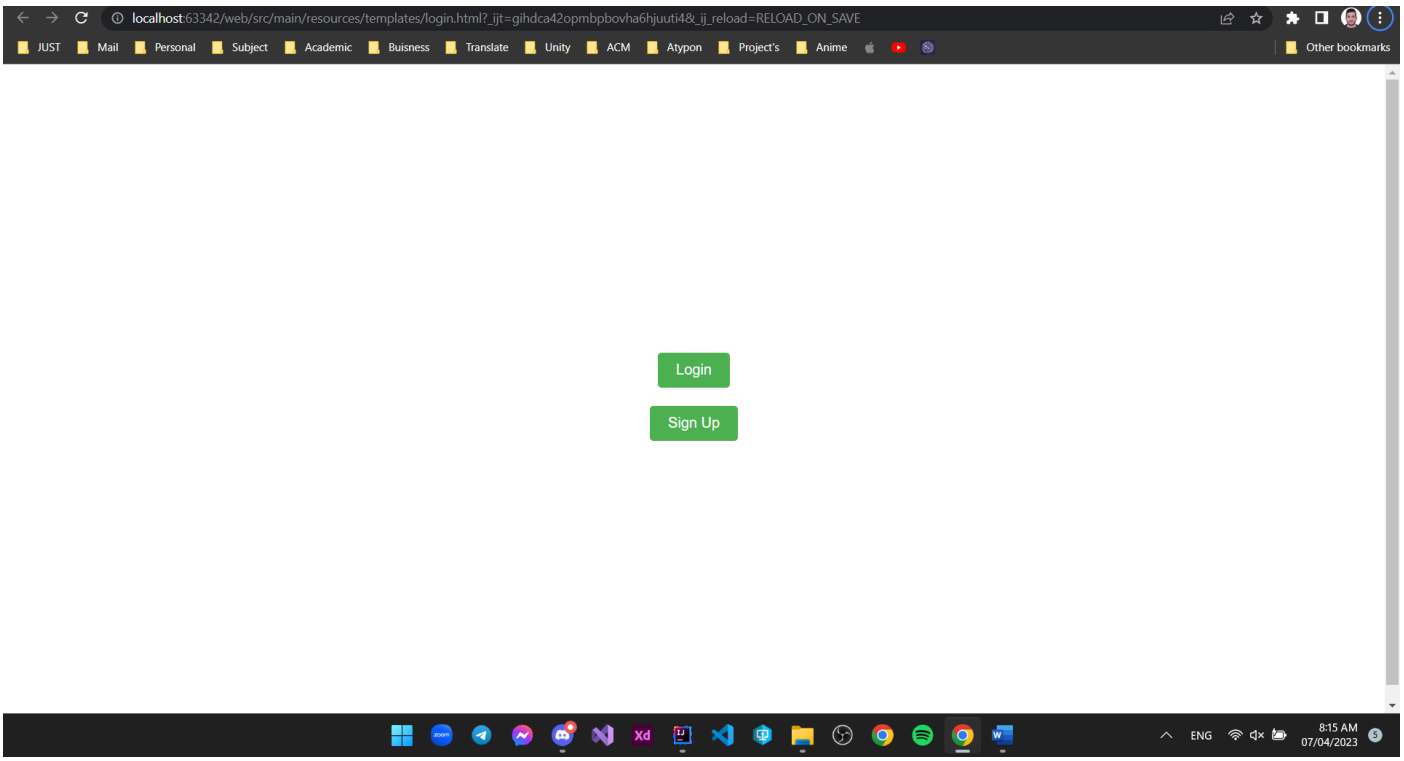
GitHub: -

I have been using a private repository on GitHub to store my project. This allows me to securely backup my code and track changes without the risk of it being publicly accessible. I can also easily collaborate with other team members if necessary by adding them as collaborators with appropriate permissions. By keeping my project in a private repository, I can control who has access to the code and ensure that it remains confidential.

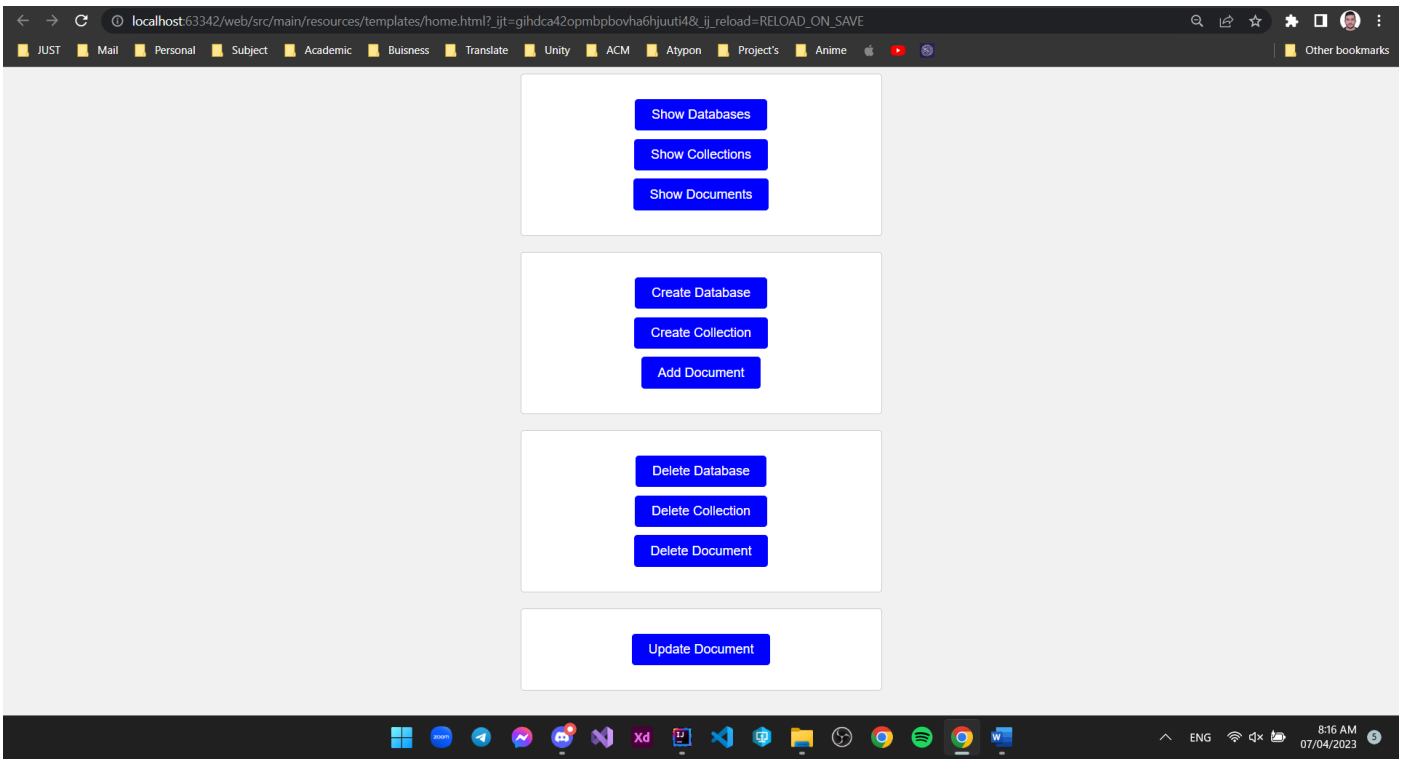


Web Application

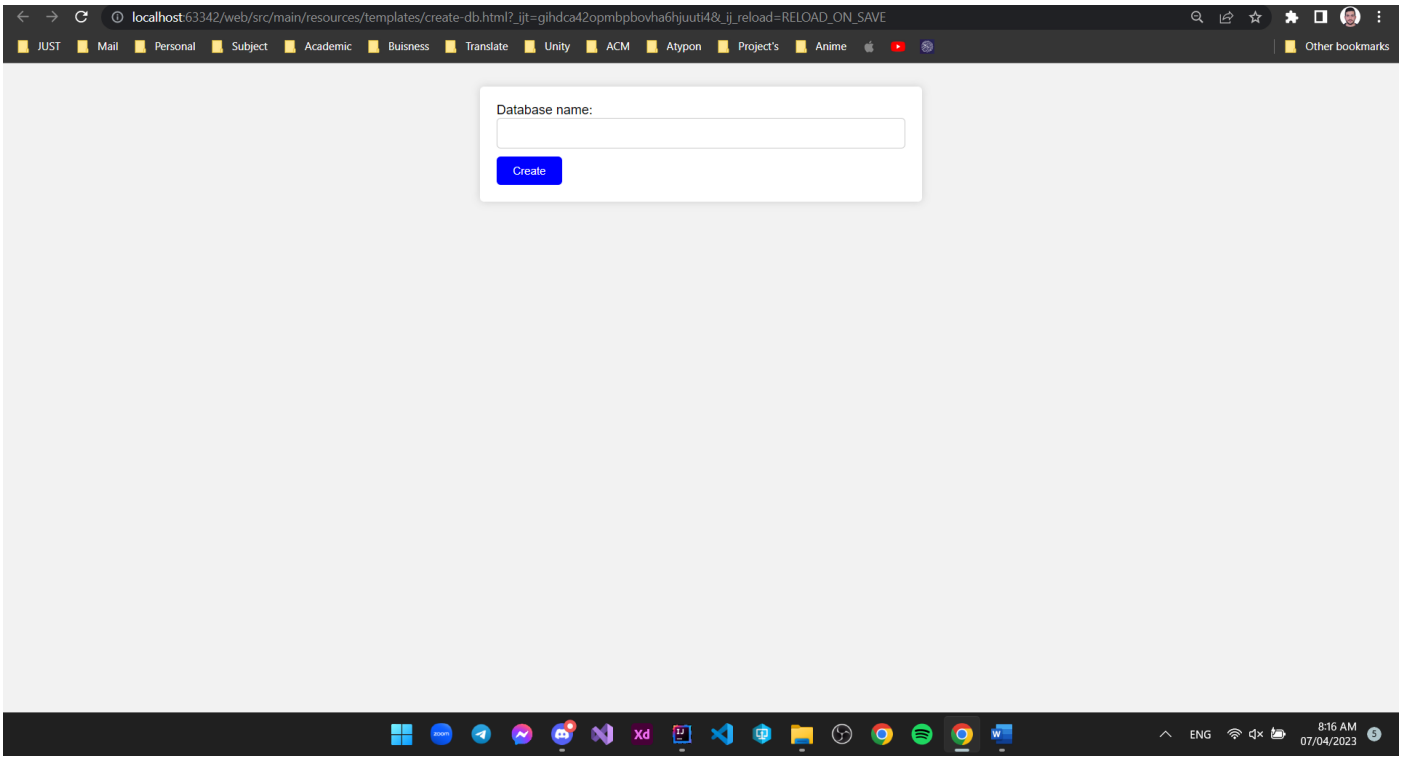
Login page :



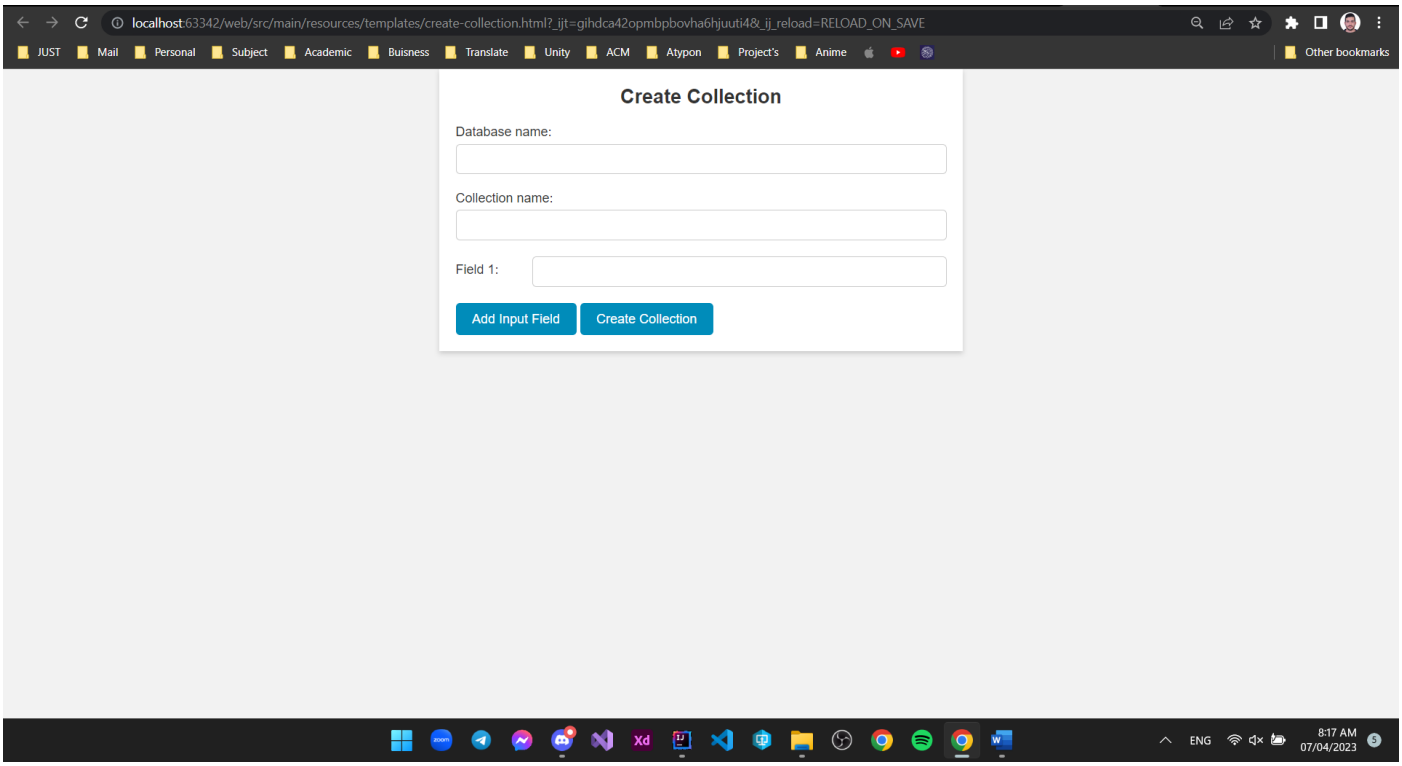
Home Page :



Create database page :



Create Collection page :



THE END