CAPSTONE REPORT

ZAIN OTOOM

ATYPON TRAINING—SEPTEMBER 2023

TABLE OF CONTENTS

INTRODUCTION

This report documents the **unfinished** capstone project and the steps that has been taken so far in the attempt of completing it. I will highlight what plans I had in mind and how I wanted to go about them.

For this report I will go over my overall design of the database and its implementation, what data structures I used, security issues, protocols used for querying, and design patterns. I also will defend my code against clean code principles, effective Java, and the SOLID principles.

As for the rest of the requirements, I will go over what I had planned for each unmet requirement.

PROBLEM DESCRIPTION AND OVERVIEW

In this assignment, we are asked to create a decentralized cluster-based noSQL database management system. The cluster is many nodes or virtual machines that communicate with each other via a network, and in this case a docker network. Each node accommodates multiple users at once for read and write operations.

The focal point of this task was that the database and its operations should be replicated across all nodes. This points to a redundancy issue in this sort of design. However, it seems, with the growth of technical improvement on hardware and even software, memory isn't a big issue when compared to big data and the best ways to deal with it. Redundancy can be even a pro as it is more fault tolerant; if one node fails, the data is still safe. This leaves us with the issue of ensuring consistency across all replicas, the capability of broadcasting updates as efficiently as possible, and the underlying query clashing.

While there are a few differences, especially in purpose, the idea behind this project strikes a big similarity with blockchain in principle. That helped in grasping the idea a bit more.

The biggest part of the project, and most time consuming was the planning and reading. It had taken a lot of time and effort trying to flesh everything out and make design decisions that wouldn't hurdle the project at some point of the implementation—I have concluded the fact that a project like this, in a professional environment, must be divided between a multitude of teams with different specializations; this project had many unexpected layers.

DATABASE STRUCTUE

One decision to make was where to store the data that will represent the database. An easy method for this seemed to be inside the application itself. I used an absolute path for testing the flows of the database, but, as this would cause the nodes to have a shared storage, it means that the paths should be changed to something relative. Relative to the node running itself while still in the same container. This simply done by fixing the paths I've used. However, to make sure that the data inside each container is persistent, I learned a bit about Docker volumes. A volume is mounted to each container created to save the data of every node. I should note that this is one part that is not be fully complete and has not been tested.

Now, as for the structure, I have created a "storage" folder inside the project. Inside "storage" will be all the databases created by the system's administrator. However, one "database" is prebuilt into storage. It is the "system" database. It contains two "collections": nodes and users. In each there are JSON files representing the nodes and users of the system. A node file looks like this:

```
{
  "nodeID":"345c43ef-ef7b-4ccc-80bc-c4f7896f1a2e",
  "IPAddress":"10.0.0.1",
  "databaseList":[],
  "assignedUsers":[
    {
      "ID":"dcc312db-4be7-4e4b-9099-e8facb9751c9",
      "name":"diaa_otoom",

"password":"3700adf1f25fab8202c1343c4b0b4e3fec706d57cad574086467b8b3ddf273e
c",
      "email":"diaa@email.com",
      "role":"SYSTEM_ADMIN",
      "nodeIP":"10.0.0.1"
    }
  ]
}
```

And as you can see, a user file is grouped into it so no need to show an example of a user.

It is important to note, that this directory is excluded from every CRUD operation done on every other database, as this data should not be messed with.

Inside storage, I have created a METADATA file that contains data about the database: what databases are present, what collections are inside them, and what files are inside those collections. It also includes data about the system as well, users and nodes. This is done for efficiency of finding the users and nodes of the system by them IDs.

This file is also excluded from CRUD operations.

BOOSTRAPPER AND NODES

Since everything is one application, creating multiple images of the same application wasn't going to work in this way. I didn't want to separate my bootstrapper as another application because that would cause redundancy in used services. That as well as the need to keep the bootstrapper connected to the system somehow for whenever a new user registers.

So, I found a way, using environment variables and if statements, to isolate the workings of the bootstrapper and every other node working as an image of the same application.

```
public static void main(String[] args) {
   if (System.getenv("BOOTSTRAPPER") != null) {
      BootstrapperNodeService.start();
   }
   SpringApplication.run(NoSqlDecentralizedDatabaseApplication.class,
args);
}
```

The Bootstrapper is the only image with the environment variable “BOOTSTRAPPER” assigned. When running the image, defining it is essential, like so:

```
docker run -e BOOTSTRAPPER=true DB
```

This ensures that the start() method is only called once during the entire initial run of the system. As for the method itself, this is what it does:

```
public static void start() {
    clearAssignedUsersInNodes();
    initializeUsers();
    initializeNodes();
    balanceNodes();
    NodeService.setIsBootstrap(true);
    try {
        for(int i = 0; i <= nodes.size(); i++) {
            exec("docker",
                    "run",
                    "-d",
                    "-p", 8000 + i + ":8081",
                    "--name", "node_" + i,
                    "--network", "cluster",
                    "--ip", "10.1.4." + i,
                    "-e BOOTSTRAPPER=false",
                    "DB");
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

First, it clears whatever users were assigned to any node in previous run, then it calls on the current nodes and current users and balances them in the round robin method.
And for every other service that the bootstrapper only has access to them, I’ve created a service to flag the node as a bootstrapper.
The latter part of this method is to create the containers of the other nodes based on the number of nodes given. Notice how the environment variable BOOTSTRAPPER is set to false to indicate a node that will only facilitate the CRUD database services.

The bootstrapper uses many services to facilitate its need:

DataLayerService

This service facilitates the communication from the bootstrapper with the system data, the users the nodes. The DataLayerService in turn, communicates with a FileService that does the work or creation or retrieval. The two methods inside DataLayerService are very general

which was intentional.

```
public static List<?> getList(String path, String collectionName, Class<?>
dataType) {
    List<Object> list = new ArrayList<>();
    List<String> IDs = FileService.getIDsFromMetadata(collectionName);
    for (String ID : IDs) {
        Object object = FileService.readJSONFile(path, collectionName,
dataType, ID);
        list.add(object);
    }
    return list;
}
public static <T> void create(T object, String ID, String collectionName) {
    boolean created = FileService.createJSONFile(FileService.getPath() +
"/system/" + collectionName + "/" + ID + ".json", object);
    if (created) {
        FileService.updateSystemMetaData("system" , collectionName, ID);
    }
}
```

Bootstrapper uses this service to initialize both the users and nodes.

```
private static void initializeUsers() {
    users = (List<User>) DatalayerService.getList(systemPath,
userCollection, User.class);
}
private static void initializeNodes() {
    nodes = (List<Node>) DatalayerService.getList(systemPath,
nodeCollection, Node.class);
}
```

LoadBalancerService

This is simple service that has one method that takes a list of users to balance against a list of nodes. It returns an important value which is the currentNode in the bootstrapper. This number represents the index inside the nodes list of the node that should take the next user that registers into the system. This way, we don't have to rebalance every time. Just keep track of the last node in the cycle.

Since the bootstrapper takes care of newly registered users, you can see the use of currentNode in createUser method:

```
public static String createUser(String name, String email, String password)
{
    if (AuthenticationService.findUser(email, password)) {
        return null;
    } else {
        Node node = nodes.get(currentNode);
        User user = new User(IDGeneratorService.createId(),name,
HashingService.hash(password), email, Role.SYSTEM_USER,
node.getIPAddress());
        DatalayerService.create(user, user.getID(), "users");
        users.add(user);
        node.getAssignedUsers().add(user);
        FileService.updateJSONFile(FileService.getPath() + "/system/nodes/"
+ node.getNodeID() + ".json", node);
        currentNode = (currentNode + 1) % nodes.size();
        return node.getIPAddress();
    }
}
```

AUTHENTICATIONSERVICE

As you can see above, this service simply checks if the user is already in the system of not.

```
public static boolean findUser(String email, String password) {
    if (email == null || password == null || email.isEmpty() ||
password.isEmpty()) {
        throw new IllegalArgumentException("Email and password cannot be
null or empty");
    }
    List<User> userList = (List<User>) DatalayerService.getList("/system",
"users", User.class);
    for (User user : userList) {
        if (user.getEmail().equals(email)) {
            if (HashingService.hash(password).equals(user.getPassword())) {
                    return true;
            }
        }
    }
    return false;
}
```

As you can see, it also communicates with the DataLayerService to get a list of Users to compare against.

IDGENERATORSERVICE

This service is used to create indexing for the files created. It will be used everywhere a JSON file is created. This way, calling on a file using this ID will ensure the quickest retrieval of the file.

```
public static String createId() {
    return UUID.randomUUID().toString();
}
```

HASHINGSERVICE

This service contains a simple SHA-256 hashing method that will be used to hash passwords everywhere.

```
public static String hash(String input) {
    try {
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        byte[] hashBytes =
digest.digest(input.getBytes(StandardCharsets.UTF_8));
        StringBuilder hexString = new StringBuilder();
        for (byte b : hashBytes) {
            String hex = Integer.toHexString(0xff & b);
            if (hex.length() == 1) hexString.append('0');
            hexString.append(hex);
        }
        return hexString.toString();
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
        return null;
    }
}
```

NODESERVICE

This is the service that create the flag related to the bootstrapper. As you can see, the initial value is false, which is for every container off this image expect when the environment

variable BOOTSTRAPPER is set.

```
private static boolean isBootstrap = false;

public static boolean isBootstrap() {
    return isBootstrap;
}

public static void setIsBootstrap(boolean isBootstrap) {
    NodeService.isBootstrap = isBootstrap;
}
```

FILESERVICE

This is the most crucial service of the entire system. It is the service that takes care of creating, deleting, and updating every folder and every file. You will see its uses as we go on.

DATABASE SYSTEM IMPLEMENTATION

To make use of what I have learned about Spring and the MVC design pattern, I decided to use them for my database overall design. My project has the following main packages: Controllers, Services, Models, and Security.
To elaborate more on this design choice, it removed the need for a querying API. I used endpoints as my queries. The HTTP methods, POST, GET, and DELETE, correspond perfectly to the CRUD operations in databases.

Let's start with the models created for this system. Please note, I used Lombok.

MODELS

Database

```
@Getter
@Setter
@AllArgsConstructor
public class Database {
    String name;

}
```

Collection

```
@Getter
@Setter
@AllArgsConstructor
public class Collection {
    String name;
    String databaseName;
}
```

Document

```
@Getter
@AllArgsConstructor
@EqualsAndHashCode
public class Document {
    String ID;
    String collectionName;
    String databaseName;
}
```

Node

```
@AllArgsConstructor
@NoArgsConstructor
@Getter
@Setter
@ToString
public class Node {
    String nodeID;
    String IPAddress;
    List<Database> databaseList;
    List<User> assignedUsers;
}
```

User

```
@AllArgsConstructor
@NoArgsConstructor
@Getter
@Setter
@ToString
public class User {
    private String ID;
    private String name;
    private String password;
    private String email;
    private Role role;
    private String nodeIP;

}
```

PropertySchema

```
@AllArgsConstructor
@NoArgsConstructor
@Getter
@Setter
public class PropertySchema {
    private String name;
    private String type;

}
```

CollectionSchema

```
@AllArgsConstructor
@NoArgsConstructor
@Getter
@Setter
public class CollectionSchema {
    String collectionName;
    private List<PropertySchema> properties;

}
```

Note that the collection schema is simply a list of the properties of the files inside the collections, their name and type.

CONTROLLERS

AuthenticationController

This controller takes care of the login/register part of the application. It includes a Get method for retrieving the login page.

```
@GetMapping("/login")
public String getLoginPage(@RequestParam(value = "nodeIP", required =
false) String nodeIP, Model model) {
    if (!NodeService.isBootstrap()) {
        System.out.println("BAD REQUEST");
        return null;
    }
    if (nodeIP != null) {
        model.addAttribute("nodeIP", nodeIP);
    }
    return "login";
}
```

And a Post method for retrieving the information used to login. This method redirects the admin to the database page, while redirecting normal users to the documents page.

```
@PostMapping("/login")
public String login(@RequestParam("username") String username,
                    @RequestParam("password") String password,
                    @RequestParam("nodeIP") String nodeIP,
                    HttpSession session) {
    if (!NodeService.isBootstrap()) {
        System.out.println("BAD REQUEST");
        return null;
    }
    User user = AuthenticationService.findUser(username, password, nodeIP);
    if (user != null) {
        System.out.println("Login successful.");
        String role = String.valueOf(user.getRole());
        session.setAttribute("role", role);
        if (role.equals("SYSTEM_ADMIN")) {
            return "redirect:/database";
        } else {
            return "redirect:/documents";
        }
    } else {
        System.out.println("Login unsuccessful.");
        System.out.println("BAD REQUEST");
    }
    return null;
}
```

A Get method for registering.

```
@GetMapping("/register")
public String showRegisterPage() {
    return "register";
}
```

A Post method for creating the user after submitting the information for signing up.

```
@PostMapping("/register")
public String register(@RequestParam("username") String username,
                       @RequestParam("password") String password,
                       @RequestParam("confirmPassword") String
confirmPassword,
                       @RequestParam("email") String email,
                       RedirectAttributes redirectAttributes) {
    if (!NodeService.isBootstrap()) {
        System.out.println("BAD REQUEST");
    }
    if (!password.equals(confirmPassword)) {
        System.out.println("Passwords mismatch.");
        return "redirect:/register";
    }
    String nodeIP = BootstrapperNodeService.createUser(username, email,
password);
    if (nodeIP != null) {
        System.out.println("Registration successful.");
        redirectAttributes.addAttribute("nodeIP", nodeIP);
    } else {
        System.out.println("Registration unsuccessful.");
    }
    return "redirect:/login";
}
```

Finally, a method for logging out and ending the session.

```
@GetMapping("/logout")
public String logout(HttpSession session) {
    session.invalidate();
    return "redirect:/login";
}
```

DATABASECONTROLLER

A Get method to retrieve the adminDashboard.

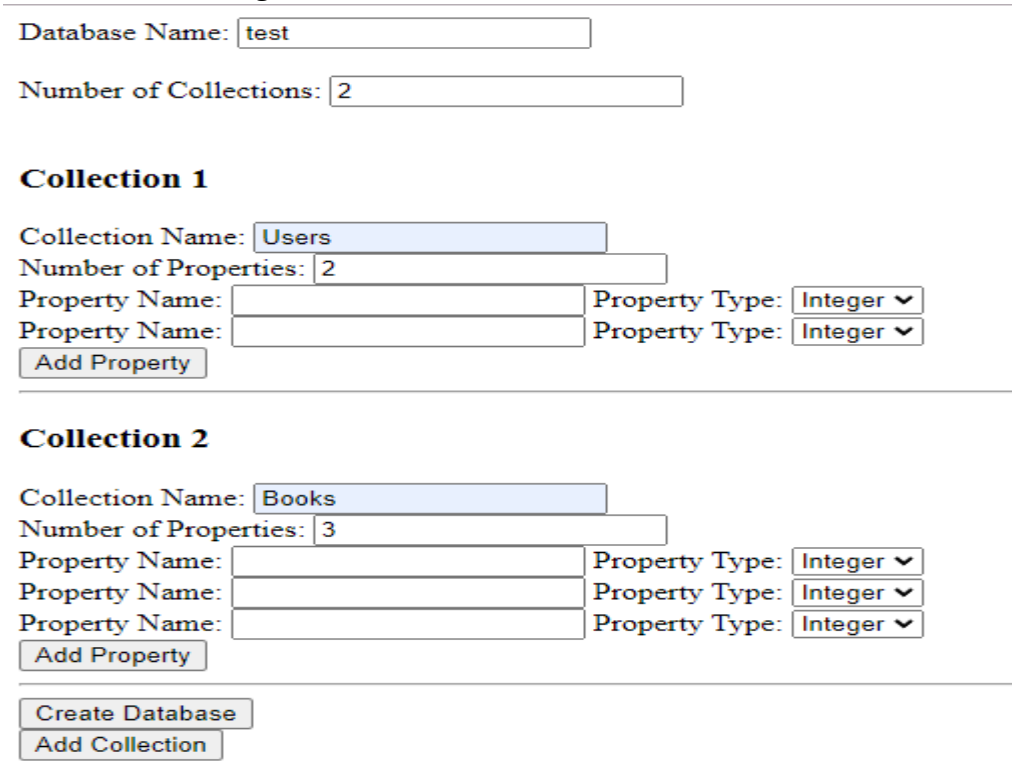
```
@GetMapping("/database")
public String getAdminPage() {
    return "adminDashboard";
}
```

This is what the admin dashboard looks like, and I'll go over each button here.

Create New Database is mapped to this method which returns another page for filling out a form to create a database.

```
@GetMapping("/createDatabase")
public String showCreateDatabasePage() {
    if (NodeService.isBootstrap()) {
        System.out.println("BAD REQUEST");
    }
    return "createDatabase";
}
```

This is what creating a database would look like.



When the admin submits the form by clicking Create Database, this method runs.

```
@PostMapping("/createDatabase")
public ResponseEntity<Void> createDatabase(@RequestParam ("databaseName")
String databaseName, @RequestParam("numCollections") int numCollections,
HttpServletRequest request) {
    if (NodeService.isBootstrap()) {
        System.out.println("BAD REQUEST");
    }
    List<CollectionSchema> collectionSchemas =
DatabaseService.getCollectionSchemas(numCollections, request);
    boolean created = DatabaseService.createDatabase(databaseName,
collectionSchemas);
    if (created) {
        System.out.println("Database with the name " + databaseName + "
successfully created.");
        return new ResponseEntity<>(HttpStatus.CREATED);
    } else {
        System.out.println("Unable to create database.");
        return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    }
}
```

Based on the number of collections the admin decides to create, CollectionSchemas are put into a list from the method getCollectionSchema from the databaseService.

```
public static List<CollectionSchema> getCollectionSchemas(int
numCollections, HttpServletRequest request) {
    List<CollectionSchema> collectionSchemas = new ArrayList<>();
    for (int i = 0; i < numCollections; i++) {
        int numProperties =
Integer.parseInt(request.getParameter("numProperties" + i));
        List<PropertySchema> propertySchemas = new ArrayList<>();
        for (int j = 0; j < numProperties; j++) {
            String propertyName = request.getParameter("propertyName" + i +
"-" + j);
            String propertyType = request.getParameter("propertyType" + i +
"-" + j);
            PropertySchema propertySchema = new
PropertySchema(propertyName, propertyType);
            propertySchemas.add(propertySchema);
        }
        CollectionSchema collectionSchema = new
CollectionSchema(request.getParameter("collectionName" + i),
propertySchemas);
        collectionSchemas.add(collectionSchema);
    }
    return collectionSchemas;
}
```

The above method creates a schema for each property based on name and type, then fills the list of collectionSchemas with these property schemas one by one.

Keep in mind creating the collection schema is very important for when we need to add documents in this collection.

Now after creating the schema, it’s time to create the database inside the storage package mentioned earlier.

This the createDatabase method found in the databaseService.

```
public static boolean createDatabase(String
databaseName,List<CollectionSchema> collectionSchemas) {
    if (FileService.createFolder("", databaseName)) {
        Database database = new Database(databaseName);
        FileService.addToMetadata(database);
        for (CollectionSchema collectionSchema : collectionSchemas) {
            String collectionName = collectionSchema.getCollectionName();
            CollectionsService.createCollection(databaseName,
collectionName);
            FileService.createJSONFileGivenPath(databaseName +
File.separator + collectionName + File.separator + collectionName +
"_schema.json", collectionSchema);
            FileService.addToMetadata(new Document(collectionName +
"_schema", collectionName, databaseName));
        }
        return true;
    }
    return false;
}
```

It creates a database object and makes sure to update the metadata file. Then through the collectionsService collections are created in a similar manner and update the metadata. As for

the files being created here, they are the schemas of each collection, how a document inside this collection should look like. These JSON files are stored inside every collection folder, but they are excluded from the CRUD operations.

Going back to the admin dashboard, let's see the Delete Database button.

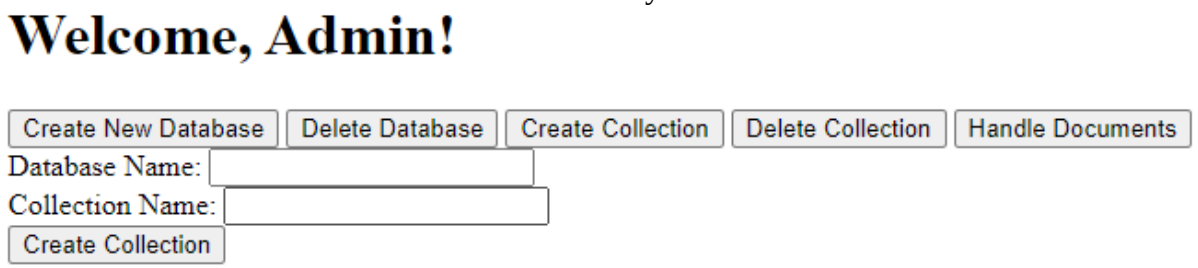
It simply asks for the name of the database and calls on this delete method.

```
@DeleteMapping("/deleteDatabase")
public ResponseEntity<Void> deleteDatabase(@RequestParam String
databaseName) {
    if (NodeService.isBootstrap()) {
        System.out.println("BAD REQUEST");
    }
    boolean deleted = DatabaseService.deleteDatabase(databaseName);
    if (deleted) {
        System.out.println("Database with the name " + databaseName + "
successfully deleted.");
        return new ResponseEntity<>(HttpStatus.OK);
    } else {
        System.out.println("Unable to delete database.");
        return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    }
}
```

Which simply calls on the databaseService to delete the entire folder and its contents.

```
public static boolean deleteDatabase(String databaseName) {
    boolean deleted = FileService.deleteFolder("", databaseName);
    if (deleted) {
        FileService.removeFromMetadata(new Database(databaseName));
        return true;
    }
    return false;
}
```

As for Create Collection and Delete Collection they both work in similar manners.



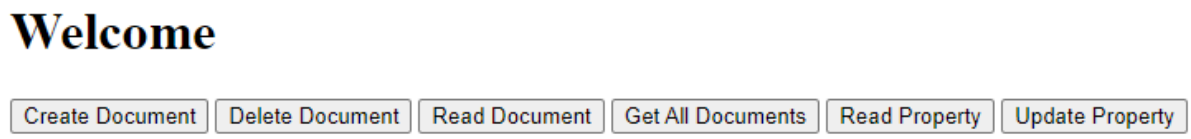
The create collection method in the create database method is used here, and the delete is the same way to delete the database.

Now, for the Handle Documents button, it simply takes the user to the page where we can access the documents from. This is the landing page for system users unlike the admin who has their own landing page. This is all in the documentsController.

DOCUMENTSCONTROLLER

```
@GetMapping("/documents")
public String getPage() {
    return "userDashboard";
}
```

This is the page for handling documents.



Upon clicking on Create Document, the names of the database and collection are asked for. When the user submits, they are given a form where they can fill the properties of the document they want to submit.

```
@GetMapping("/createDocument")
public String getForm(@RequestParam String databaseName, @RequestParam
String collectionName) {
    if (NodeService.isBootstrap()) {
        System.out.println("BAD REQUEST");
    }
    String schema = FileService.readJSONSchema(FileService.getPath() +
File.separator + databaseName + File.separator + collectionName +
File.separator + collectionName + "_schema.json");
    return FormGeneratorService.generateFormFromSchema(schema,
databaseName);
}
```

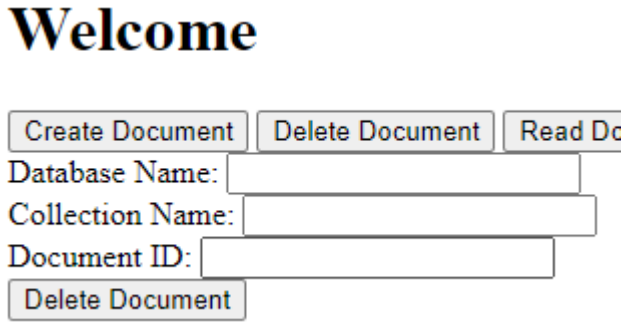
The FormGeneratrorService allows for the creation of the form based on the given schema. The generateFormFromSchema method builds an HTML file along with a JavaScript code to build this form. The submit button to this form is mapped to this end post point.

```
@PostMapping("/documents/{databaseName}/{collectionName}")
public ResponseEntity<Void> createDocument(@PathVariable String
databaseName, @PathVariable String collectionName, @RequestBody String
data) {
    if (NodeService.isBootstrap()) {
        System.out.println("BAD REQUEST");
    }
    boolean created = DocumentsService.createDocument(databaseName,
collectionName, data);
    // confirming msgs
}
```

This the create document method in the DocumentsService.

```
public static boolean createDocument(String databaseName, String
collectionName, String data) {
    Gson gson = new Gson();
    String ID = IDGeneratorService.createId();
    JsonObject jsonObject = gson.fromJson(data, JsonObject.class);
    boolean created = FileService.createJSONFile(FileService.getPath() +
File.separator + databaseName + File.separator + collectionName +
File.separator + ID + ".json", jsonObject);
    if (created) {
        FileService.addToMetadata(new Document(ID, collectionName,
databaseName));
        return true;
    }
    return false;
}
```

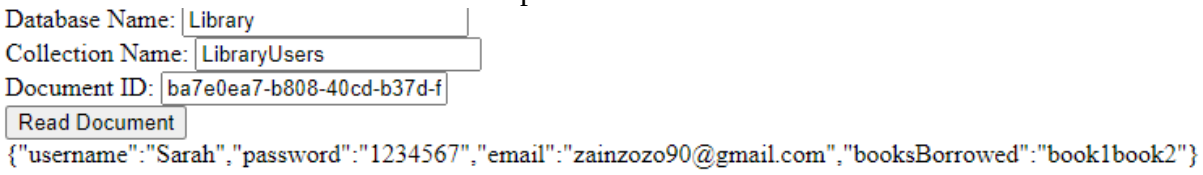
Deleting a document simply takes the ID of the file and removes it from the path it is in.



This is the delete document method in the DocumentsService. Note how in both, deletion and creation, the metadata is updated.

```
public static boolean deleteDocument(String databaseName, String
collectionName, String documentId) {
    boolean deleted = FileService.deleteJSONFile(databaseName +
File.separator + collectionName, documentId + ".json");
    if (deleted) {
        FileService.removeFromMetadata(new Document(documentId,
collectionName, databaseName));
        return true;
    }
    return false;
}
```

Read document takes the same data and prints out the data.



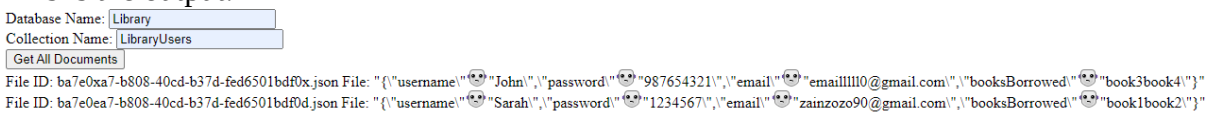
This is the method that reads the document.

```
public static String readDocument(String databaseName, String
collectionName, String documentId) {
    return String.valueOf(FileService.readJSONFile(databaseName,
collectionName, JsonObject.class, documentId + ".json"));
}
```

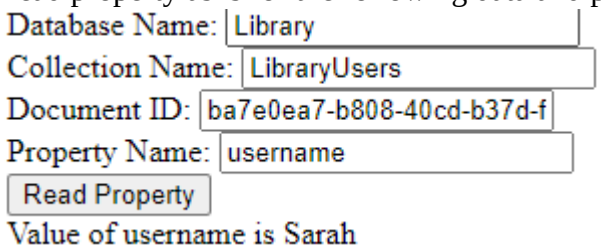
Now, the Get All Documents is similar, but I used a map to store the ID of the file as the key since they are simply the names of the JSON files, and the JSON file as a JsonObject in the value. This is the method.

```
public static Map<String, JsonObject> getAllDocumentsInCollection(String
databaseName, String collectionName) {
    Map<String, JsonObject> JsonFiles = new HashMap<>();
    Gson gson = new Gson();
    List<String> JsonFilesIDs =
FileService.getJSONFileNames(FileService.getPath() +  File.separator +
databaseName + File.separator + collectionName);
    for (String ID: JsonFilesIDs) {
        JsonFiles.put(ID,
gson.fromJson(String.valueOf(FileService.readJSONFile(databaseName,
collectionName, JsonObject.class, ID)), JsonObject.class));
    }
    return JsonFiles;
}
```

This is the output.



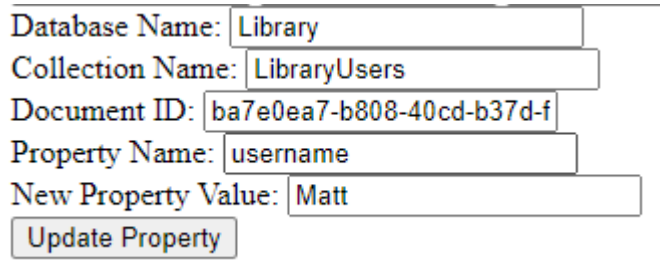
Read property asks for the following data and prints the result out.



This is the method for reading properties.

```
public static String readProperty(String databaseName, String
collectionName, String documentId, String propertyName) {
    Optional<String> propertyValue =
FileService.readProperty(databaseName, collectionName, documentId +
".json", propertyName);
    return propertyValue.orElse("not found.");
}
```

Updating the property works similarly.



After clicking on the update button, the file should be updated.

This is the method for updating the properties.

```
public static boolean updateProperty(String databaseName, String
collectionName, String documentId, String propertyName, String
updatedValue) {
    JsonObject jsonObject =  FileService.readJSONFile(databaseName,
collectionName, JsonObject.class, documentId + ".json");
    assert jsonObject != null;
    jsonObject.addProperty(propertyName, updatedValue);
    if (deleteDocument(databaseName, collectionName, documentId)) {
        FileService.createJSONFile(FileService.getPath() + File.separator +
databaseName + File.separator + collectionName + File.separator +
documentId + ".json", jsonObject);
        return true;
    }
    return false;
}
```

As for the templates I used simple HTML and JavaScript.

SECURITY

Two simple classes that have not been fully utilized: the permissions and roles classes. The plan was to use them more effectively in determining what a user can do and cannot do.

Role

```
public enum Role {
    SYSTEM_ADMIN,
    SYSTEM_USER
}
```

Permissions, which has two methods: setSystemAdminPermissions and setSystemUserPermissions.

```
@Getter
@Setter
public class Permission {
    private boolean createDatabase;
    private boolean deleteDatabase;
    private boolean createCollection;
    private boolean deleteCollection;
    private boolean createDocument;
    private boolean deleteDocument;
    private boolean updateDocument;
    private boolean readDocument;
}
```

DEFENSE AGAINST CLEAN CODE AND SOLID PRINCIPLES AND EFFECTIVE JAVA

1) Comments

I did not use many comments.

2) Duplicate code

I omitted any duplicate code as soon as I saw it by creating a method that would eliminate it. Since duplicate code means I need this functionality many times, it made sense to make it a function with a clear call.

3) Long Methods

Most of the methods are short except for the methods inside FileService due to its nature.

4) Long Parameter Lists

Most methods have many parameters, but they represent data sent over requests.

5) Data Classes

I do not have any data classes.

6) Shotgun Surgery

The main shotgun surgery that might exist is the consistency between parameter lists among the different service classes, which can be fixed easily with time.

7) Speculative Generality

There is no speculative generality as everything written is clear in its objective.

8) Primitive Obsession

I did not overuse any primitives.

9) Naming

All names—methods and variables—in my project are related to their functionality.

10) Returning Null

Due to the lack of experience in working with Sprin much, I had to make use of returning null a couple of time.

11) Conditions

My conditions are very clear, especially due to naming my methods that have Boolean return well.

12) Lists Over Arrays

I only used list and not arrays.

SOLID

1) Single Responsibility principle

All my methods and classes have a single responsibility to take care of.

2) Open/Closed principle

This project is open to extension. As you can see since I have services doing the work, this makes the process simpler.

3) Liskov's Substitution principle

I did not user inheritance in this project.

4) Interface Segregation principle

There is no unnecessary coupling, as everything is a service class that doesn't depend much on anything, save for the dependency on the FileService.

5) Dependency Inversion principle

I tried to make everything generic and not depend on higher modules. However, you may see some inconsistencies in the methods of FileServices that were created to aid something in a higher-level service, this can be fixed by making sure certain parameters are the same.

UNMET REQUIREMENTS

It is important for the reader to know that I have not met every specified requirement. The VMs of the nodes, which I talked about how I planned to do, are not tested, therefore optimistic locking was not implemented either.

DOCKER NODES AND NETWORK

I talked about that in the bootstrapper and node part.

OPTIMISTIC LOCKING

My plan for it was to use the ReadWriteLock, which would be an edit on the end points, which readLock for read operations and writeLock on write operations. However, this also was not tested.

DOCUMENT TO NODE AFFINITY

This was also unimplemented. My plan was to simply add a property in each document dictating which node can write to it. Much like how the node contains its users. The document would be checked for its node, retrieving the intended IP address, and communicating this request to the correct node.

BROADCASTING

The idea was a simple communication will all the nodes, since the nodes' data is part of the storage inside the system, so all nodes know each other's IP address. However, this was also untested.

REFLECTIONS

Unfortunately, I did not have enough time to finish this due to some personal matter. I apologize for the unfinished work, but I'd worked this far—both on this project and others—so I thought I should submit whatever I have already finished.

This project required the most research and planning out of everything we have done in this training so far. There were unexpected layers to it that one had to deal with in whatever manner works with what we already have, as working from scratch was not an option.

However, I feel like I understood the MVC concept more after working on this project and how services should behave inside one project.