

Document Database Report

ESSAM ABDO

1. Table of Contents

2. Introduction.....	6
Learning.....	7
Requirement Analysis	7
Choosing The Tool.....	7
3. Project Structure	8
4. File System	9
ID Uniqueness.....	11
5. Shared-packages package.....	13
rw operations.....	13
authentication	14
exporting	14
Logging	15
indexing.....	16
6. Master Node.....	17
Definition	17
Roles :	17
Dockerfile.....	17
Controllers	18
– Main-controller	19
– Read-controller.....	21
– Scale controller.....	22
7. Slave Node.....	23
Definition	23
Roles	23
Dockerfile.....	24

Controller.....	24
8. Eureka-server	26
Definition	26
Roles	26
Dockerfile.....	27
9. Scale server	27
Definition	28
Role.....	28
Controller.....	28
10. Consistency.....	32
11. Thread Safety	33
12. Failure handling	33
Exceptions handling	33
Spring retry	34
13. Logging	35
14. Effective Java Principles.....	36
Creating And Destroying Objects	36
– Item 4 Enforce non-instantiability with a private constructor.....	36
– Item 5 Avoid creating objects	36
– Item 6 Eliminate obsolete object references	37
Methods Common To All Objects	37
– Item 8 Obey the general contract when overriding equals	37
– Item 9 Always override hashCode when you override equals.....	37
– Item 10 Always override toString.....	37
– Item 11 Override clone judiciously	37
Classes And Interfaces	37

– Item 13 Minimize the accessibility of classes and members	37
– Item 14 In public classes, use accessor methods, not public fields	37
– Item 15 Minimize Mutability	37
– Item 16 Favour composition over inheritance	37
– Item 17 Design and document for inheritance or else prohibit it.	37
– Item 18 Prefer interfaces to abstract classes	38
– Item 20 Prefer class hierarchies to tagged classes	38
– Item 22 Favour static member classes over non-static	38
Generics.....	38
– Item 23 Don't use raw types in new code.....	38
– Item 24 Eliminate unchecked warnings	38
– Item 25 Prefer lists to arrays.....	38
– Item 29 Consider type safe heterogeneous containers.....	38
Enums And Annotations.....	38
– Item 35 Prefer annotations to naming patterns	38
– Item 36 Consistently use the Override annotation	38
Methods	38
– Item 38 Check parameters for validity.....	38
– Item 39 Make defensive copies when needed.....	39
– Item 40 Design method signatures carefully.....	39
– Item 41 Use overloading judiciously	39
– Item 43 Return empty arrays or collections, not nulls	39
General Programming.....	39
– Item 45 Minimize the scope of local variables.	39
– Item 46 Prefer for-each loops to traditional for loops.	39

– Item 47 Know and use libraries	39
– Item 48 Avoid float and double if exact answer are required	39
– Item 49 Prefer primitive types to boxed primitives.....	39
– Item 50 Avoid Strings where other types are more appropriate.....	39
– Item 51 Beware the performance of string concatenation	40
– Item 52 Refer to objects by their interface	40
– Item 55 Optimize judiciously	40
– Item 56 Adhere to generally accepted naming conventions	40
Exceptions.....	40
– Item 57 Use exceptions only for exceptional conditions.....	40
– Item 58 Use checked exceptions for recoverable conditions and runtime exceptions for programming errors	40
– Item 59 Avoid unnecessary use of checked exceptions	40
– Item 60 Favour the use of standard exceptions.....	40
– Item 61 Throw exceptions appropriate to the abstraction	40
– item 65 Don't ignore exceptions.....	40
Concurrency.....	40
– item 66 Synchronize access to shared mutable data	40
– item 67 Avoid excessive synchronization.....	41
– Item 71 Use lazy initialization judiciously.....	41
– Item 73 Avoid thread groups	41
15. Clean Code Principles	41
General Rules.....	41
Names Rules	41
Functions rules.....	42
Comments rules.....	42

Source Code Structure	42
Classes	42
16. SOLID Principles	43
S : Single Responsibility Principle	43
O : Open / Closed Principle (OCP).....	43
L : Liskov substitution principle (LSP).....	44
I : Interface Segregation Principle (ISP)	44
D : Dependency Inversion Principle (DIP)	44
17. Technologies used	44
Java	44
Maven.....	45
Spring boot	45
Eureka server	45
JSON In Java	45
ZT Zip	46
Spring retry	46
Swagger	46
Docker	46
18. User Manual	47
19. References.....	47

2. Introduction

In this report , I will explain the project structure , used technologies and my methodology for my **Document Database** project .

Learning

First of all , in order to build a **Document Database Management System (DDMS)** myself , I had to study the internal design of NOSQL databases , learning about its functionality , its features and the differences between it and relational databases .

The book I chose for studying was :

NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence .

Along side a YouTube playlist that contain some videos which somehow related to the main topic of this project .

Requirement Analysis

Yes , you have guessed it ! , the next step was to study the requirement of the system I need to build , I watched the two uploaded videos and read the assignment description very carefully , until I reached a good and strong understanding of the system .

Choosing The Tool

After days and days of learning , I found myself ready for the next step ;

Choosing the tool I need for building the application , and for that I chose **Spring Boot Java Framework** , why Spring boot ? a question you might ask .

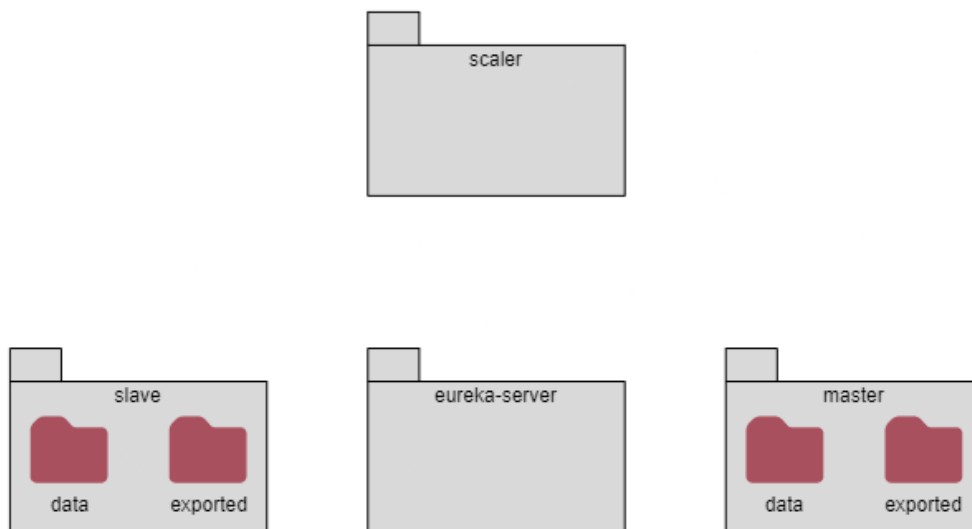
Basically , Spring Boot helps developers create applications that just run . Specifically , it lets you create standalone applications that run on their own ,

without relying on an external web server , by embedding a web server such as Tomcat or Netty into your app during the initialization process . With its simple syntax , Spring boot was really easy to learn and get started with .

3. Project Structure

In this section I am going to talk about the structure and the main components of the project .

The picture below shows the project structure from **IntelliJ** perspective :



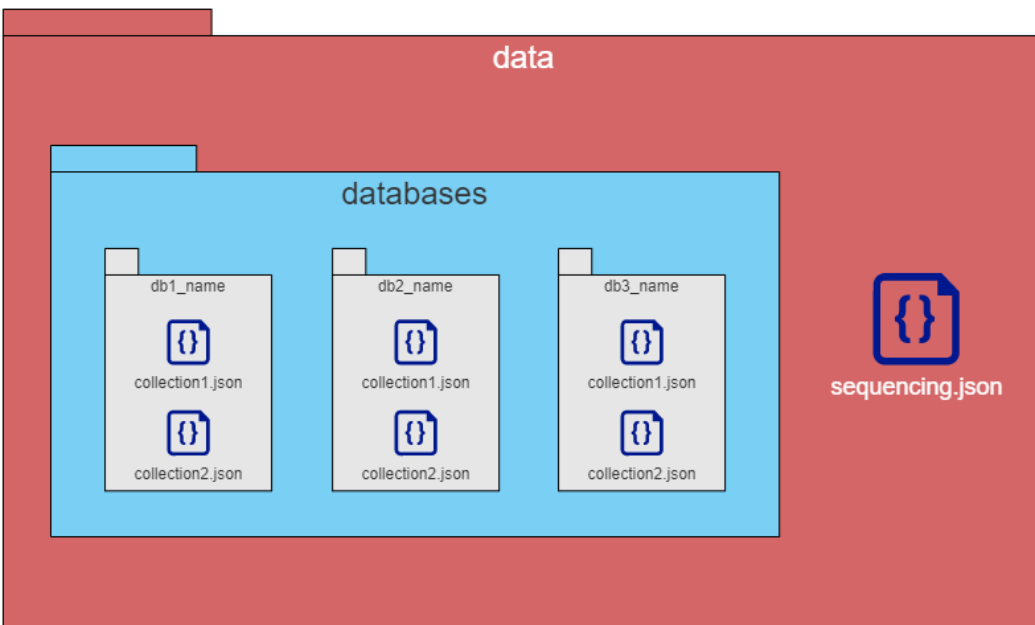
The project consists of the following components :

- 1 - Eureka-server package : this package consists of a spring boot application called eureka that plays the role of the router in my application and has the ability to balance the load between the slave nodes .
- 2 - Master package : this package has multiple packages itself , it plays the role of the master node .

- 3 - Slave package : this package has multiple packages itself , it is considered a blueprint for slave node instances .
- 4 - Shared-packages package : this package has multiple package that both the master and the slave depend on , this package will be discussed in other section in this report .
- 5 - Scaler package : this package is used when the user wants to add a new slave to the group.

4. File System

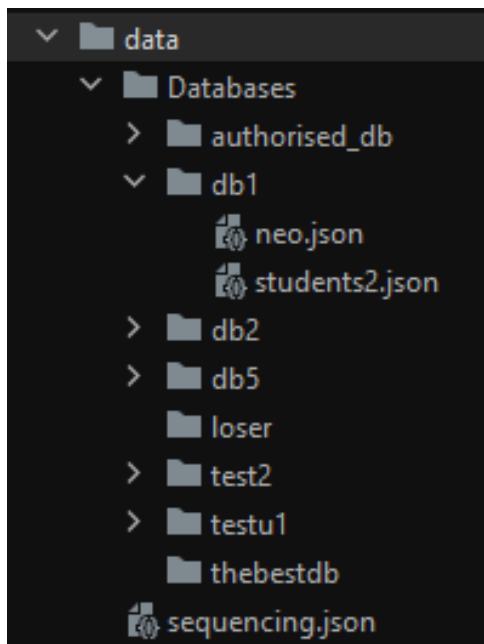
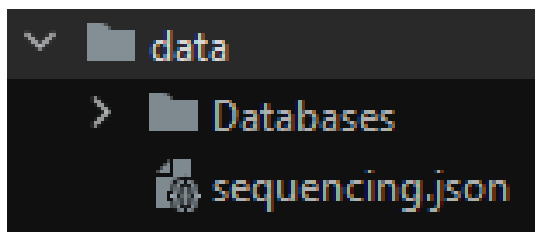
When it comes for file system , the following figure demonstrates the file system structure :

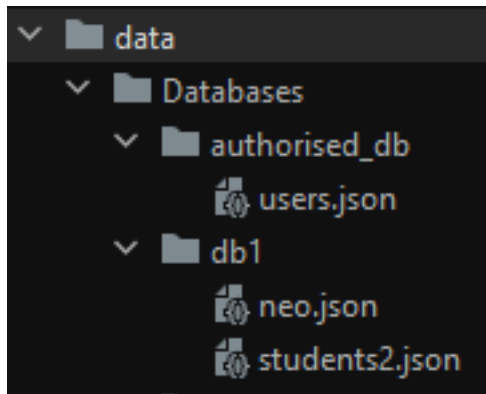


- data directory

- databases directory
 - db1 (directory that represents a database with the name db1)
 - collection1.json (json file / represents a collection with name collection1 / contains multiple documents)
 - collection2.json
 - db2
- sequencing.json

the following are screenshots that display the file system structure from IntelliJ perspective :





Here is a sample of what a typical collection would look like internally :

A collection is basically a json file that contains mainly one json array , that has multiple json objects (documents) .



ID Uniqueness

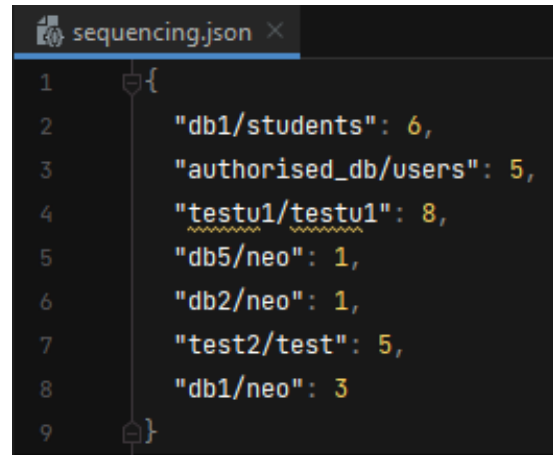
In order to insure ID uniqueness inside the collection , an action is needed to be taken .

Here it comes the role of **sequencing.json** file .

sequencing.json : is a json file that contains one json object , this object has multiple properties , each one consists of the following structure :

**“database_name/collection_name” :
the_next_id**

When new a document is inserted in a collection :



```
1  {
2    "db1/students": 6,
3    "authorised_db/users": 5,
4    "testu1/testu1": 8,
5    "db5/neo": 1,
6    "db2/neo": 1,
7    "test2/test": 5,
8    "db1/neo": 3
9  }
```

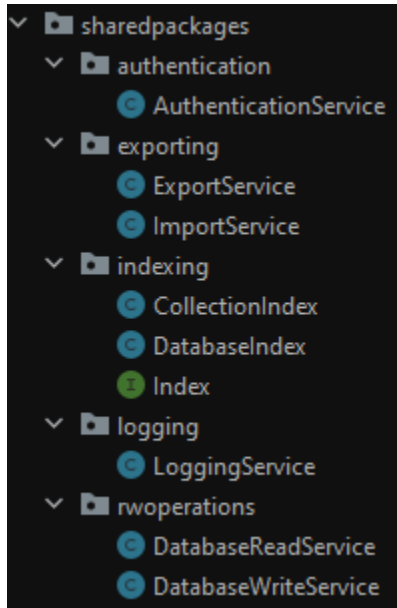
- If this document has an ID , a certain procedure will insure that this id is unique among the documents in the same collection .
if it is not unique i.e. (there exists a document with the same id) , the insertion process is rejected .
if it is unique , the insertion process will be done successfully .
- If the document doesn't have an ID , it will automatically get a unique ID using sequencing.json file .
for example :
“db1/students” : 6
this line means that in database db1 , inside collection students , the next document that does not have an ID , will get the ID = 6 .
After it gets the ID , this value will be incremented by one , to be :
“db1/students” : 7

Note :

- This process will insure that documents are unique inside the collection .
- If the user wanted to change database name or collection name , names in this file will be changed as well , to insure correctness of ID uniqueness .

5. Shared-packages package

This package exists in both master and slave , and it consists of multiple packages :



The following is a description for each sub-package in **sharedpackages** package :

rw operations

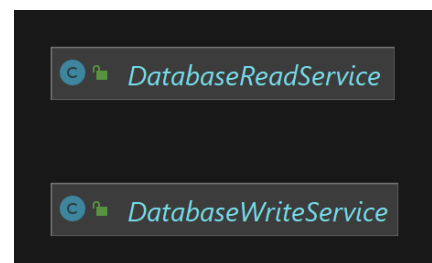
This package is the layer that communicates directly with the file system , it has two classes only :

1 - DatabaseReadService :

This class is responsible for all read operations from the database .

2 - DatabaseWriteService :

This class is responsible for all write operations from the database .



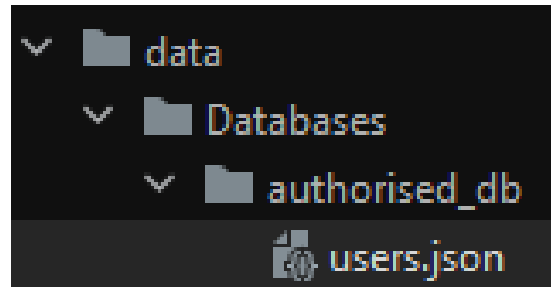
authentication

This package has one class only :

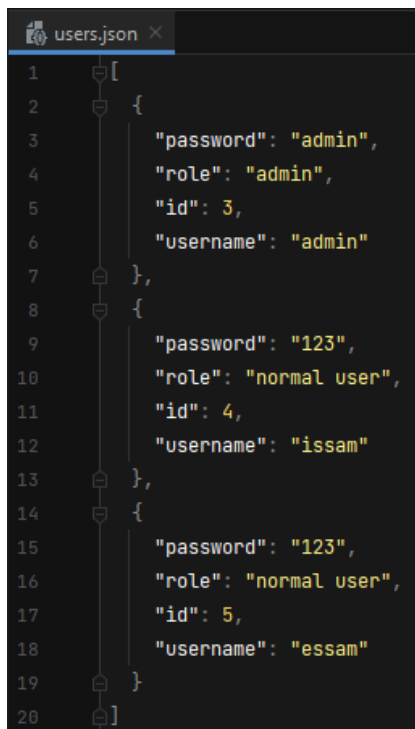
AuthenticationService :

This service is responsible for User/Admin authentication .

Admin and user can log in using their username and their password , their credentials are stored in a database called **authorized_db** , in a **users** collection .



The following figure shows how documents are stored inside **users** collection :



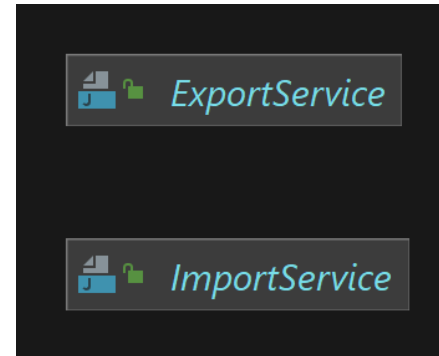
exporting

This package is responsible for database schema importing and exporting , and it has two classes :

1 - **ExportService** :

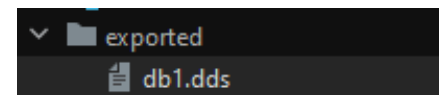
This class is responsible for exporting schemas , when a database is exported , it is saved in **exported** directory .

Exported databases are saved in this directory as **.dds** (**D**ocument **D**atabase **S**chema) file , this is a custom extension the I use for my **DDMS** .



2 - **ImportService** :

This class is responsible for importing database schemas , it only import files that of type **dds** , and adds the imported database schema as a normal database in **data directory** .



Logging

When it comes to monitoring your applications, **system.out.println()** is not the best choice , due to its bad performance, so I had to search for a replacement for it .

Spring boot was the saviour too , it has the ability to export the logs into an external file in the configuration that you chose, using what's **LogBack** .

I created a class to serve this purpose , it is inside the **logging package** and it is called **LoggingService** , this class takes the desired messages and logs them based on their priority.

indexing

You cannot deny that in a way or another that indexing is one of the essences of this project, and this is why it should took you too long to think about the right data structure to use , and for that I chose **HashMap** .

HashMap has a read complexity of $O(1)$, which is very fast , but it has a limitation which is the maximum number of elements that can be stored in it .

A **HashMap** in Java can have a maximum of 2^{30} buckets for storing entries - this is because the bucket-assignment technique used by **java.util.HashMap** requires the number of buckets to be a power of 2, and since **ints** are signed in Java, the maximum positive value is $2^{31} - 1$, so the maximum power of 2 is 2^{30} .

And to handle this , I used the concept of polymorphism whenever I used the HashMap index , so I can change the index easily whenever it is needed .

One possible replacement for **HashMap** index , is the **O2-tree** , as it has a better performance , more than **T-tree** , **B+tree** , **AVL-tree** and **Red-Black tree** . (Daniel Ohene-Kwofie, 2012)

Let's demonstrate the scenario of indexing a collection in an example :

- 1 - The user wants to read from collection users , so he/she sends the request to the master .
- 2 - The master redirects the read request to the one of the slaves .
- 3 - The slave check if it has index for the chosen collection , and if not it asks the master to create one and sends it to the slave .
- 4 - Then it serves the request and sends the result back to the master so it can be displayed to the user .

If a write has done on any collection , the master will send a request for all the registered slaves in order to flush their indices that are no longer valid .

6. Master Node

Definition

The master is the main component and the controller of whole project .

It is run in a docker container , then it contacts the Eureka-server so it can be discovered by it later . It is responsible mainly for writing .

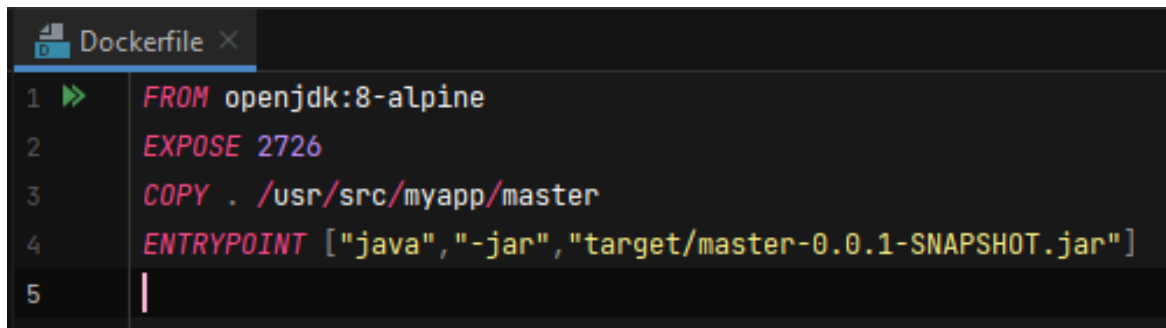
Roles :

The master node is responsible for :

- 1 - Contacting Eureka-Server so it can be registered in it .
- 2 - Receiving and resolving write requests by itself .
- 3 - Receiving and redirecting read requests to the slave nodes .
- 4 - Propagating the data to the slaves whenever it is changed (replication) .
- 5 - Exporting and importing database schemas .
- 6 - Propagating the exported schemas to the slaves whenever a new one is exported (replication) .
- 7 - Receiving and redirecting scaling requests to the scale server .
- 8 - Index initiation and creation and propagation .

Dockerfile

This is the dockerfile of the master node :



```
Dockerfile x
1  FROM openjdk:8-alpine
2  EXPOSE 2726
3  COPY . /usr/src/myapp/master
4  ENTRYPOINT ["java","-jar","target/master-0.0.1-SNAPSHOT.jar"]
5  |
```


As you can see the master is exposed on **port 2726** .


Controllers

The master has three controllers :

- Main-controller
- Read-controller
- Scale-controller

The following is a screenshot of the controllers using **swagger** .

 **Swagger**
Supported by SMARTBEAR

Select a definition default 

Api Documentation ^{1.0}

[Base URL: localhost:2726/]
<http://localhost:2726/v2/api-docs>

Api Documentation

[Terms of service](#)

[Apache 2.0](#)

main-controller Main Controller

>

read-controller Read Controller

>

scale-controller Scale Controller

>

– Main-controller

Here is a screenshot for the main controller http requests :

main-controller Main Controller

POST /master/{database} createDatabase

PUT /master/{database} updateDatabaseName

DELETE /master/{database} deleteDatabase

POST /master/{database}/{collection} createCollection

PUT /master/{database}/{collection} updateCollectionName

DELETE /master/{database}/{collection} deleteCollection

PUT /master/{database}/{collection}/{id} updateDocument

DELETE /master/{database}/{collection}/{id} deleteDocument

DELETE /master/{database}/{collection}/{id}/{property} deleteField

GET /master/{database}/{collection}/{property}/index getIndex

POST /master/{database}/{collection}/insertDocument insertDocument

POST /master/export/{database} exportDatabase

POST /master/exportLocally/{database} exportDatabaseLocally

POST /master/import importSchema

POST /master/importLocally importSchemaLocally

POST /master/propagateData/{slave_id} propagateDataToSlave

POST /master/propagateDataToAllSlaves propagateDataToAllSlaves

POST /master/propagateSchemasToSlaves propagateSchemasToAll

This controller is used for write operations on the databases and many other features :

- 1 - Create a database
- 2 - Update a database name
- 3 - Delete a database
- 4 - Create a collection
- 5 - Update a collection name
- 6 - Delete a collection
- 7 - Update a document
- 8 - Delete a document
- 9 - Delete a field/property in a document
- 10 - Get index (this http request is used by the slaves when they need a new index)
- 11 - Insert a document
- 12 - Export a database
- 13 - Export a database to a certain path/location
- 14 - Import a database schema
- 15 - Import a database schema from a certain path/location
- 16 - Propagate data to a slave
- 17 - Propagate data to all slaves
- 18 - Propagate schemas to a slave

— Read-controller

This is a screenshot for the read controller http requests , this user interface is generated using **Swagger** :

read-controller Read Controller

GET	/master/read/{database}	getDatabase
GET	/master/read/{database}/{collection}	getCollection
GET	/master/read/{database}/{collection}/document	getDocument
GET	/master/read/{database}/{collection}/documents	getDocuments
GET	/master/read/{database}/{collection}/documents/{id}/{field}	getField
GET	/master/read/{database}/collectionsNames	getAllCollectionNames

Whenever the user need to grab any information from a database , the read request has to pass from the master node , using the read-controller , then this controller passes the request to a slave to serve the read operation .

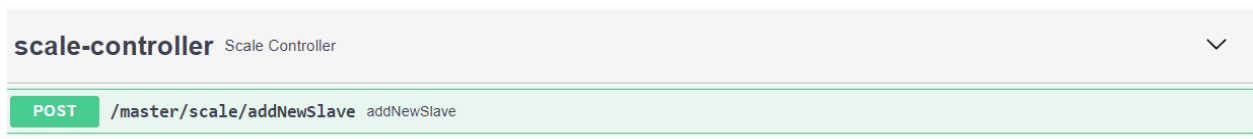
These are the http requests that the controller will pass to the slave :

- 1 - Get a whole database as a JSON object
- 2 - Get a whole collection as a JSON object
- 3 - Get a whole document as a JSON object
- 4 - Get documents that has a certain value to a specific property as an array of JSON objects
- 5 - Get some field inside a certain document
- 6 - Get the names of all collections inside a specific database

– Scale controller

This controller in the master node is responsible for taking the user requests to expand a new node and then passes it to another server (**scale spring boot application port 1234**)

The following is a screenshot of the scale controller in the master node :



This controller only has one http request which is for adding a new slave node , this request actually talks to another http request on another spring boot application .

7. Slave Node

Definition

It is our secondary node , which is used for serving read requests . It is run on a docker container and can be replicated , starting from **port 2613** .

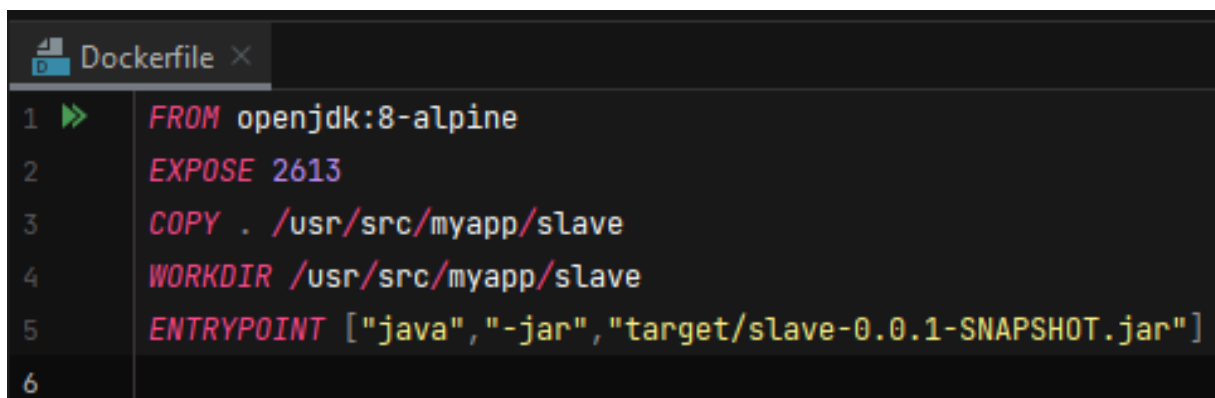
Roles

Slave nodes have multiple tasks :

- 1 - Contacting Eureka-Server so it can be registered in it .
- 2 - Receiving the latest updated data from the master .
- 3 - Receiving the latest updated exported schemas from the master .
- 4 - Receiving read requests from the master , serving them then returning the responses back to the master .
- 5 - Asking the master for indices then storing them .

Dockerfile

The following is a snapshot of the dockerfile for a slave node :



```
1  >> FROM openjdk:8-alpine
2  EXPOSE 2613
3  COPY . /usr/src/myapp/slave
4  WORKDIR /usr/src/myapp/slave
5  ENTRYPOINT ["java", "-jar", "target/slave-0.0.1-SNAPSHOT.jar"]
6
```

As you can see , it is very similar to the master's .

Controller

Slave node has only one controller which is used for read operations :

read-controller Read Controller	
GET	/slave/{database} getDatabase
GET	/slave/{database}/{collection} getCollection
GET	/slave/{database}/{collection}/document getDocument
GET	/slave/{database}/{collection}/documents getDocuments
GET	/slave/{database}/{collection}/documents/{id}/{field} getField
GET	/slave/{database}/collectionsNames getAllCollectionNames
POST	/slave/flushIndex flushIndex
POST	/slave/updateData updateData
POST	/slave/updateExported updateExported

As you can see in the screenshot , these are the http requests that the slave node can serve :

- 1 - Get a whole database as a JSON object
- 2 - Get a whole collection as a JSON object
- 3 - Get a whole document as a JSON object
- 4 - Get documents that has a certain value to a specific property as an array of JSON objects
- 5 - Get some field inside a certain document
- 6 - Get the names of all collections inside a specific database
- 7 - Flush a collection index
- 8 - Update data directory
- 9 - Update exported schemas directory

As mentioned above , when the master receives a read request , it directs it to the one of the slave to serve it .

8. Eureka-server


Definition

It is a service in spring cloud , that can help with the process of service discovery and load balancing , this spring boot application is run in a docker container that is mapped to **port 8761** .

Roles

- 1 - Registering master node .
- 2 - Registering slave nodes .
- 3 - It balances the load send from the master to the slaves , using **Round Robin** algorithm .

This is a screenshot of the Eureka-server user interface .


Toggle navigation

System Status

Environment	test	Current time	2022-06-30T19:40:30 +0000
Data center	default	Uptime	00:57
		Lease expiration enabled	true
		Renews threshold	8
		Renews (last min)	16

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
MASTER	n/a (1)	(1)	UP (1) - 51c5a5ab5e55:master:2726
SLAVE	n/a (3)	(3)	UP (3) - d73975e2d378:slave:2615 , e2ab8be98645:slave:2613 , 126c07f8fa80:slave:2614

As you can see below , Eureka-server has discovered **one master 2726** container and **three slaves containers 2613 , 2614 and 2615** .

Dockerfile

```

1  FROM openjdk:8-alpine
2  EXPOSE 8761
3  ADD target/eureka-server-0.0.1-SNAPSHOT.jar eureka-server-0.0.1-SNAPSHOT.jar
4  ENTRYPOINT ["java", "-jar", "/eureka-server-0.0.1-SNAPSHOT.jar"]
5

```

9. Scale server

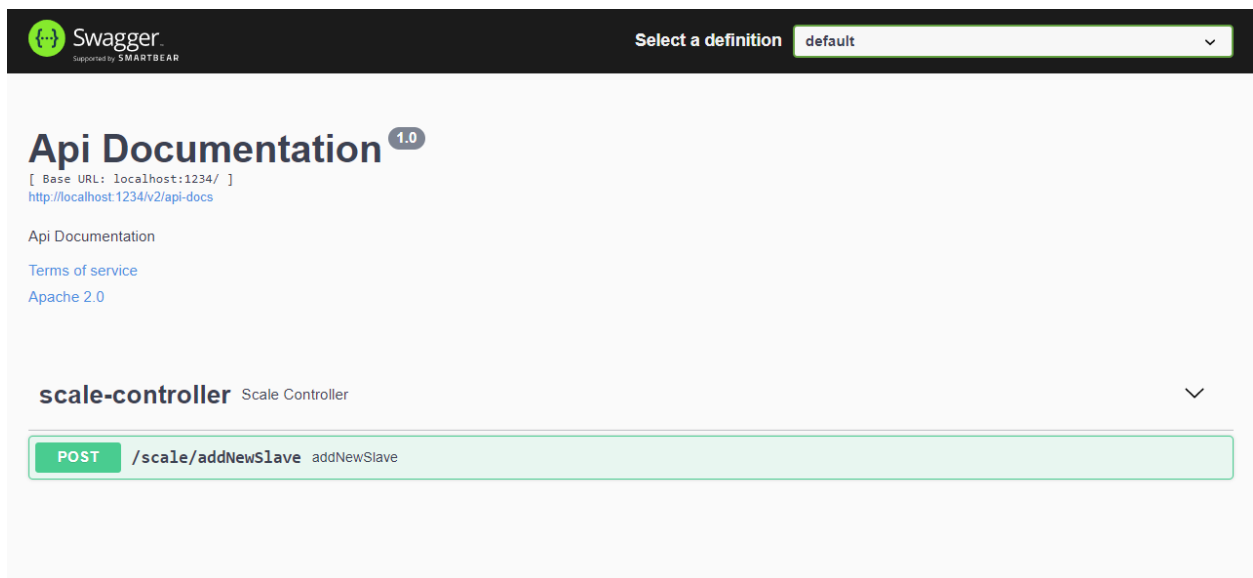
Definition

It is a spring boot application that is used to serve the purpose of adding new slave nodes , it is hosted on **port 1234** , locally (not as a docker container) .

Role

The only role for this server is to receive the order to add a new slave from the master , and doing so .

Controller



The scale server has only one controller that has only one http request , which is for adding a new slave , this operation is done by running a new docker container for a slave image .

The following is a screenshot of the method that adds a new slave :

```

@PostMapping (value = @RequestMapping("/addNewSlave"))
@Retryable (value = { Exception.class }, maxAttempts = 4, backoff = @Backoff (delay = 1000))
public void addNewSlave()
{
    try
    {
        //run docker image demo_slave
        String command =
            "docker run demo_slave1 " + " -p " + id + ":" + id + " --network=group1 " + " --rm " + " --name=slave" + id + " -v " + "./slave" +
            ":/usr/src/myapp/slave ";
        //specify the entrypoint of the container
        command += " --entrypoint java -jar target/slave-0.0.1-SNAPSHOT.jar --server.port=" + (id++);
        //command += " demo_slave1";
        Process p = Runtime.getRuntime().exec( command );
        System.out.println( "Running command: " + command );
    }
    catch(IOException e)
    {
        throw new RuntimeException( e );
    }
}

```

This is an example of the command that is going to be run by this method :

```

docker run demo_slave1

-p 2616:2616

--network=group1

--rm

--name=slave2616

-v ./slave:/usr/src/myapp/slave

--entrypoint java -jar target/slave-0.0.1-SNAPSHOT.jar --server.port=2616

```

Demo_slave1 is a docker image of the slave image .

As mentioned above , once a new slave spins up , it contacts **Eureka-server** so it get registered as a slave and to receive new read requests from the master .

The following is a screenshot of **Eureka-server** after spinning up two new slaves :

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
MASTER	n/a (1)	(1)	UP (1) - 51c5a5ab5e55:master:2726
SLAVE	n/a (5)	(5)	UP (5) - d73975e2d378:slave:2615 , 3e35c2cf8670:slave:2616 , e2ab8be98645:slave:2613 , 126c07f8fa80:slave:2614 , ab3b3617faf0:slave:2617

As you can see , there are five slave nodes registered with **Eureka-server** , three of them are spined by default using docker compose :

```
version: '3'
services:
  eureka-server:
    container_name: eureka-server
    build: ./eureka-server
    ports:
      - "8761:8761"
    restart: unless-stopped
    networks:
      - group1

  master:
    container_name: master
    build: ./master
    ports:
      - "2726:2726"
    volumes:
      - ./master:/usr/src/myapp/master
    working_dir: /usr/src/myapp/master
    restart: unless-stopped
    depends_on:
      - eureka-server
    networks:
      - group1

  slavel:
    container_name: slavel
    build: ./slave
    ports:
      - "2613:2613"
    volumes:
      - ./slave:/usr/src/myapp/slave
    working_dir: /usr/src/myapp/slave
    restart: unless-stopped
    depends_on:
      - eureka-server
      - master
    networks:
      - group1
```

```

slave2:
  container_name: slave2
  build: ./slave
  ports:
    - "2614:2614"
  volumes:
    - ./slave:/usr/src/myapp/slave
  working_dir: /usr/src/myapp/slave
  restart: unless-stopped
  depends_on:
    - eureka-server
    - master
  entrypoint: [ "java", "-jar", "target/slave-0.0.1-SNAPSHOT.jar", "--server.port=2614"
]

networks:
  - group1

slave3:
  container_name: slave3
  build: ./slave
  ports:
    - "2615:2615"
  volumes:
    - ./slave:/usr/src/myapp/slave
  working_dir: /usr/src/myapp/slave
  restart: unless-stopped
  depends_on:
    - eureka-server
    - master
  entrypoint: [ "java", "-jar", "target/slave-0.0.1-SNAPSHOT.jar", "--server.port=2615"
]

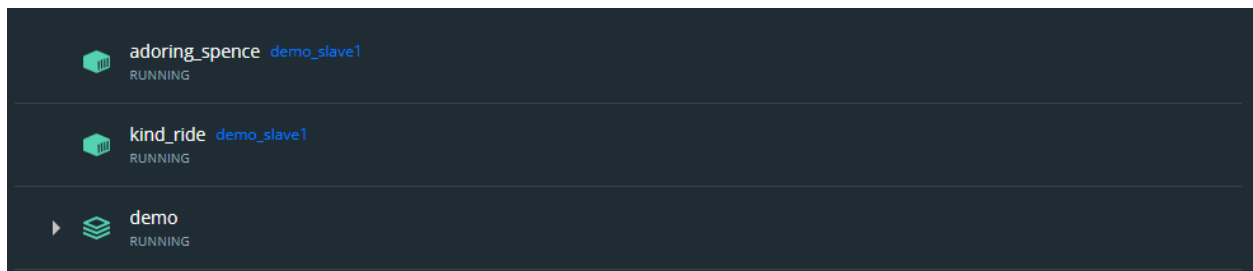
networks:
  - group1

#add all of them to the same network
networks:
  group1:
  group2:

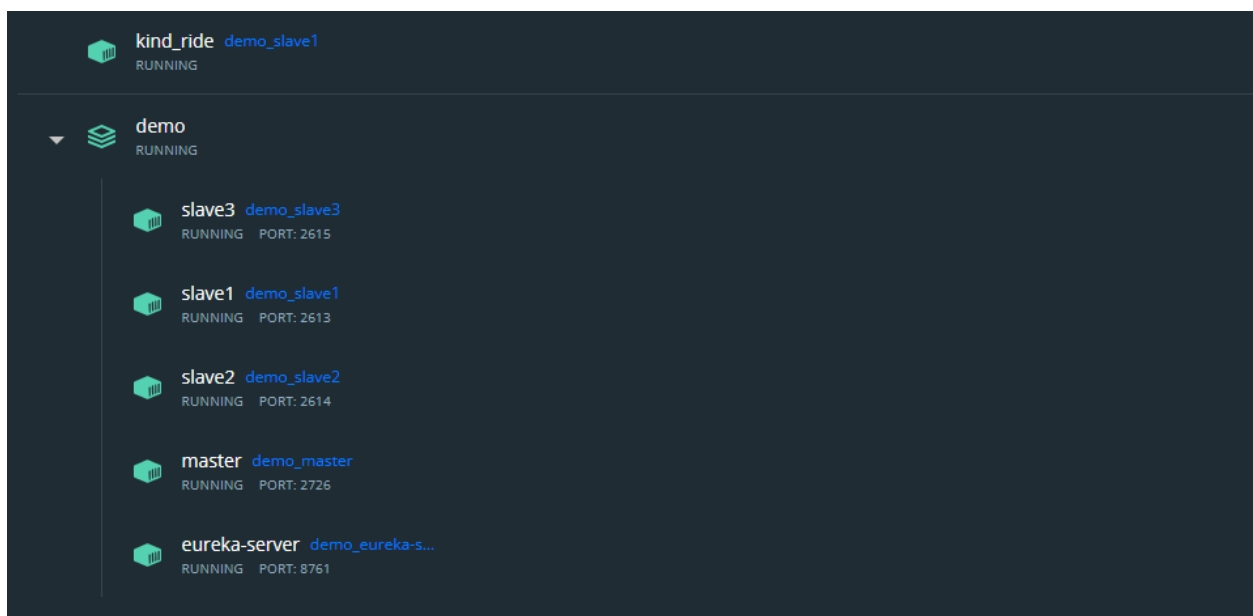
```

slave1 2613 , slave2 2614 and slave3 2615 are slaves that are spined up by default using docker-compose .

but for the other two slaves 2616 and 2617 , they are spined up using scale server , you can see below a screenshot of docker user interface , that demonstrates the current state :



In the screenshot above **adoring_spence** and **kind_ride** are the new docker containers that were spined up using **scale server** . Demo is a docker-compose that I mentioned earlier :



10. Consistency

Consistency is a must when we talk about the distribution model of a NoSQL database , I took care of consistency as well.

The slaves will serve the read requests of the users using the most updated databases , indices and schemas .

And whenever a new change occurs , the master will directly send the new version of the changed component (databases and exported schemas) .

But the thing is different for the indices , the master will inform the slaves when an index becomes invalid or old , then the slave will delete the index , after that when the slave needs an index to perform a read operation , it will ask the master to create one .

11. Thread Safety

Thread safety means that different threads can access the same resources without exposing erroneous behaviour or producing unpredictable results . This programming methodology is known as “ thread-safety . ” .

I took care of this issue too , using **Reentrant Locks** .

I used these locks to ensure that no write-write conflict will occur , these locks were used in the lowest layer (**DatabaseWriteService.java**) .

12. Failure handling

Failure is always a possibility in any system , and it should be handled in a proper way to avoid system crashes . To handle this issue I applied two solutions :

Exceptions handling

This actually a must , exceptions occur everywhere , fortunately we have IntelliJ and inform us and keep us aware of them !

I handled every possible exception in a proper way , to avoid the crashing of the system .

Spring retry

If an error happens , here comes **Spring retry service** . Spring Retry provides an ability to automatically re-invoke a failed operation. This is helpful where the errors may be transient (like a momentary network glitch).

I used this service for every method in the read-controller for the slave , and in the main-controller for the master .

```
91     @PostMapping (value = "/{database}/{collection}/insertDocument")
92     @Retryable (value = { Exception.class }, maxAttempts = 4, backoff = @Backoff (delay = 1000))
93     public void insertDocument(@RequestBody String request_body , @PathVariable ("database") String database ,
94                               @PathVariable ("collection") String collection)
95     {
96         JSONObject request_body_json = new JSONObject( request_body );
97         master_node.insertDocument( database , collection , request_body_json.getJSONObject( key: "json_object" ) );
98         propagateData();
99         flushSlaveIndex( database , collection );
100        flushMasterIndex( database , collection );
101    }
```

As you can see in **line 92** , **Retryable annotation** is used to re-invoke this method again if it has been invoked and faced an exception . It will be invoked again for four times , with a one second delay between each try .

If all the retries have failed as well , a method, named **recover method** , is called to log the error in an organised way .

```

@Recover
public void recover(Exception e)
{
    LoggingService.logError(
        "*****" );
    LoggingService.logError( "All retries completed, so Fallback method called!!!" );
    LoggingService.logError( "Recovering from exception" );
    LoggingService.logError( e.getMessage() );
    LoggingService.logError(
        "*****" );
}

```

13. Logging

Logging is very important feature to any system , so we can monitor our system heart beat properly , **System.out.println()** is considered a poor way for monitoring due to its bad performance on the system , this is why I had to find another way , and I found that **Logback** logger is the best choice .

Logback is one of the most widely used logging frameworks in the Java Community. It's a replacement for its predecessor, Log4j. Logback offers a faster implementation, provides more options for configuration, and more flexibility in archiving old log files.

I used **Logback** in each server (**master** , **slave** , **Eureka-server** and **scaler**) and each one has its own log file that is generated according to configurations I set .

I create a logging service so I can use Logback in the way I want :

```

1  package master.sharedpackages.logging;
2
3  import org.slf4j.Logger;
4  import org.slf4j.LoggerFactory;
5
6  public class LoggingService
7  {
8      private LoggingService()
9      {
10     }
11
12     5 usages
13     private static final Logger logger = LoggerFactory.getLogger( LoggingService.class );
14
15     25 usages
16     public static void logInfo(String message)
17     {
18         logger.info( "{} {} {}", System.currentTimeMillis() , Thread.currentThread().getName() , message );
19     }
20
21     29 usages
22     public static void logError(String message)
23     {
24         logger.error( "{} {} {}", System.currentTimeMillis() , Thread.currentThread().getName() , message );
25     }
26
27     public static void logDebug(String message)
28     {
29     }
30 }

```

14. Effective Java Principles

I tried to apply effective java principles as much as I can , here the rules that were applied properly in my code :

Creating And Destroying Objects

- Item 4 Enforce non-instantiability with a private constructor
For classes that group static methods and static fields.
- Item 5 Avoid creating objects
Reusing immutable objects

- Item 6 Eliminate obsolete object references

Methods Common To All Objects

- Item 8 Obey the general contract when overriding equals
- Item 9 Always override hashCode when you override equals
- Item 10 Always override toString
- Item 11 Override clone judiciously

Classes And Interfaces

- Item 13 Minimize the accessibility of classes and members
- Item 14 In public classes, use accessor methods, not public fields
- Item 15 Minimize Mutability
- Item 16 Favour composition over inheritance
- Item 17 Design and document for inheritance or else prohibit it.
The class must document its self-use of overridable methods.

- Item 18 Prefer interfaces to abstract classes
- Item 20 Prefer class hierarchies to tagged classes
- Item 22 Favour static member classes over non-static

Generics

- Item 23 Don't use raw types in new code
- Item 24 Eliminate unchecked warnings
- Item 25 Prefer lists to arrays
- Item 29 Consider type safe heterogeneous containers

Enums And Annotations

- Item 35 Prefer annotations to naming patterns
- Item 36 Consistently use the Override annotation

Methods

- Item 38 Check parameters for validity
Check parameters before execution as soon as possible.

Do it also in constructors.

- Item 39 Make defensive copies when needed.
You must program defensively, with the assumption that clients of your class will do their best to destroy its invariants.
- Item 40 Design method signatures carefully
- Item 41 Use overloading judiciously
- Item 43 Return empty arrays or collections, not nulls

General Programming

- Item 45 Minimize the scope of local variables.
- Item 46 Prefer for-each loops to traditional for loops.
- Item 47 Know and use libraries
- Item 48 Avoid float and double if exact answer are required
- Item 49 Prefer primitive types to boxed primitives
- Item 50 Avoid Strings where other types are more appropriate

- Item 51 Beware the performance of string concatenation
- Item 52 Refer to objects by their interface
- Item 55 Optimize judiciously
- Item 56 Adhere to generally accepted naming conventions

Exceptions

- Item 57 Use exceptions only for exceptional conditions
- Item 58 Use checked exceptions for recoverable conditions and runtime exceptions for programming errors
- Item 59 Avoid unnecessary use of checked exceptions
- Item 60 Favour the use of standard exceptions
- Item 61 Throw exceptions appropriate to the abstraction
- item 65 Don't ignore exceptions

Concurrency

- item 66 Synchronize access to shared mutable data

- item 67 Avoid excessive synchronization
- Item 71 Use lazy initialization judiciously
- Item 73 Avoid thread groups

15. Clean Code Principles

I tried to apply clean code principles as much as I can , here are some of the considerable rules that were applied properly in my code :

General Rules

- I avoided duplication in my code , (DRY principle or Don't Repeat Yourself) .
- I followed a standard convention .
- I was consistent . If I did something a certain way , all similar things are done in the same way .

Names Rules

- I chose descriptive and unambiguous names .
- I chose meaningful names .
- I chose pronounceable names .
- I chose searchable names .
- I made names easy to remember .
- I used names according to the context .

- I used names that are consistent with each other , for example , it's confusing to have “retrieved” and “get” as equivalent methods in distinct classes .

Functions rules

- My functions do one thing .
- I used descriptive names .
- I avoided large functions .

Comments rules

- They were as less as possible .
- I did not add obvious noise .
- I tried to explain myself more in the code .
- I avoided unnecessary comments .

Source Code Structure

- I declared variables close to their usage .
- My similar functions are close to each other .
- I placed the functions in the downward direction .
- I used white spaces to associate related things and disassociate weakly related .

Classes

- My Classes have only one responsibility .
- The base class know nothing about their derivatives .
- I followed Law of Demeter . A class should know only its direct dependencies .

16. SOLID Principles

I have applied and taken consideration of the SOLID principle , while programming :

S : Single Responsibility Principle

“ A class should have one and only responsibility over a single part of the functionality

provided by the software . ”

All classes and methods I wrote , all of them apply this principle , so no method nor class break this principle .

O : Open / Closed Principle (OCP)

“ Entities should be open for extension , but closed for modification

(We can apply this using abstract class , and other classes should inherit from this abstract class) . ”

L : Liskov substitution principle (LSP)

“ Subtypes must be substitutable for their base types

Derived types must be completely substitutable for their base types ”

I : Interface Segregation Principle (ISP)

“ One fat interface need to be split to many smaller and relevant interfaces ”

D : Dependency Inversion Principle (DIP)

“ Depend upon abstraction (interfaces) not upon concrete classes

Abstractions should not depend on the detailed whereas the details should depend on abstractions

High-level modules should not depend on low level modules ”

17. Technologies used

These are the technologies I used for developing the system :

Java

Java is the main language that I used for developing all the system .

Maven

Maven is a building tool , and it really helps you build faster application , it has a very high level of integration with Java , and it helped me a lot adding various plugins , dependencies and libraries while building the system .

Spring boot

Spring Boot is an open source, microservice-based Java web framework. The Spring Boot framework creates a fully production-ready environment that is completely configurable using its prebuilt code within its codebase .

I used this framework to build the master , the salve , the scaler and Eureka-server .

Eureka server

Eureka Server is an application that holds the information about all client-service applications. Every Micro service will register into the Eureka server and Eureka server knows all the client applications running on each port and IP address. Eureka Server is also known as Discovery Server.

This server helped me with load balancing and service discovery issues .

JSON In Java

JSON is a light-weight, language independent, data interchange format. See <http://www.JSON.org/> The files in this package implement JSON encoders/decoders in Java. It also includes the capability to convert between JSON and XML, HTTP headers, Cookies, and CDL. This is a reference implementation. There is a large number of JSON packages in Java. Perhaps someday the Java community will standardize on one. Until then, choose carefully.

I added a library from maven repository to easily integrate JSON objects and arrays with Java . This was necessary to use in order to store and handle the documents in the collections .

ZT Zip

The project is intended to have a small, easy and fast library to process ZIP archives. Either create, modify or explode them. On disk or in memory.

I used this library to easily zip the data and exported schemas directories to send them from the master to the slave as a file not as a directory , because sending a directory in a http request causes some problems in spring boot .

This library also helped me unzipping the zipped file in the other side (slave) .

Spring retry

Spring Retry provides an ability to automatically re-invoke a failed operation. This is helpful where the errors may be transient (like a momentary network glitch).

I mentioned above how this service was implemented in the system .

Swagger

Swagger is a suite of API developer tools from SmartBear Software and a former specification upon which the OpenAPI Specification is based.

I used swagger in order to test my controllers in an efficient and fast way .

Docker

Docker is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers. The service has both free and

premium tiers. The software that hosts the containers is called Docker Engine. It was first started in 2013 and is developed by Docker, Inc.

I used docker to run each application in a separate container , so I can test real life scenarios for the system .

18. User Manual

I uploaded a video on YouTube that has a full tutorial in Arabic of how to use the system :

<https://youtu.be/6H38bTb52rM>

19. References

(n.d.). Retrieved from

<https://youtube.com/playlist?list=PLfNixlsfp6zQ0rom1QKVZ6Esin405ZMQ>

Daniel Ohene-Kwofie, E. J. (2012). *O2-Tree: A Fast Memory Resident Index for NoSQL Data-Store*.

Sadalage, P. J., & Fowler, M. (n.d.). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*.