



Document Database

Final Project

Mammon Ahmad Alshamali

11/6/2022

Contents

Introduction	5
Requirements	5
Solution.....	6
Database implementation	7
Working scenario.....	7
Implementation	8
Authentication.....	8
Creating Schema	10
Adding objects	12
Collection.....	13
Index.....	14
Database	16
Node Manager.....	17
Docker Util	18
Node Util	19
Node	20
Caching	21
Database Operations	22
Database Import and Export	25
Data Structure	26
- General maps in the system:	26
- LRU cache.....	26
- Inverted Index	27
Multithreading the locks.....	28
Reentrant Read Write Lock	28

Synchronized	30
Scalability and Consistency	31
Security	36
The protocol.....	37
Clean Code	42
The naming	42
Functions.....	43
Comments	43
Return Null and Pass Null	44
Classes	45
Inherit constant	45
Encapsulation.....	45
Effective Java	46
Static factory	46
Avoid creating unnecessary objects	46
Override (to string, equal, hash code)	46
Minimize the accessibility of classes and members.....	46
In public classes, use accessor methods, not public fields.....	47
Other Items	47
SOLID	48
single-responsibility principle	48
The open-closed principle	49
The Liskov substitution principle.....	49
The interface segregation principle	49
Dependency Inversion Principle	49
Design pattern.....	50

Singleton	50
Facade	50
Mediator	51
DAO	51
MVC	51
Template	53
DevOps	54
GitHub	54
Maven	54
GitHub action.....	54
Docker.....	56
Docker in Docker.....	59
Kubernetes	60

Introduction

Requirements

Build a document DB that fulfills the following requirements:

- Reads are the majority of transactions (from any node in a cluster)
- Writes are done only rarely (only through a connection to the database controller/connection manager).
- Has to have fast reads (e.g., caching in memory).
- Document-based database (JSON objects).
- Each document type has a JSON object schema, and each object schema should belong to a database schema.
- A document has an ID that should be unique and indexed efficiently.
- Create indexes on a single JSON property.
- (Optionally) support multi-property indexes (static indexes).
- Data and Schema changes including Index initiation and creation will be done on the controller but Data and schema and indexes will be replicated to nodes so that searches for reads are done locally in a node.
- Can scale the cluster horizontally (data should replicate to new nodes up to 3 nodes or more), the client will ask the database controller/connection manager for a connection (to a specific database schema) and this should connect it to one of the load balancing nodes for the duration of the client connection (e.g. docker network or different ports for the nodes or virtual machines).
- Allow for database schema to be exported and imported (from the controller).
- Default administrator to use Controller to manage roles of users and administrators (login name, password, and role as user or administrator),

default administrator has to change his password the first time. - Don't use indexing libraries such as Apache Solr.

- Controller is a bottleneck and single point of failure, typically it should be redundant, but this is not required at this stage.
- Create a demo application that is either a command-line application or a web application that utilizes the database using its schema (e.g., email application, catalog application...etc).

Solution

The suggested solution for the requirements is as follows:

- A master controller for CRUD operations, creating nodes, deleting nodes, and self-healing. By self-healing means, in case of a fall for the system, the master controller frees all resources, creates nodes, and replicates the data to them. Also, administrative task to solve the database problems like increase nodes, delete nodes.
- The data will be stored on storage and in memory for the master controller and memory for the nodes.
- Every collection of data will be alone in one node. Like if there are 4 collections there will be at minimum 4 nodes (Similar data together).

Database implementation

Working scenario

The working scenario is as follows, there will be a default database for the first time run. This default database contains the default administrator who can create users and admins. This admin needs to change his password for the first time using the database.

The admin needs to create a schema for the objects which will be stored in the database, this will be done by sending a sample of the object then a schema will be generated for this object also the admin should provide the namespace and the schema name with the sample. After that, a node for this schema will be created. What is more, users and admins can add objects to the database and these objects will be validated depending on the schema. After each valid object, the node will be updated.

The index will be created after the first object is added to the collection, and the index will be updated after each adding, updating, and deleting operation. Each collection will have its index and all the object properties will be indexed as well.

Deleting a schema will result in deleting the related nodes as well as the stored data in the storage.

Reading from nodes will be done by redirecting the users after validating their username and password.

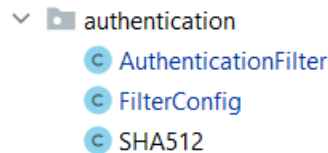
The database can be exported and imported as a zip file. In case of importing, the zip file will be extracted and stored in the storage and the memory, as well as a node, will be created or updated for the new collection.

Implementation

The database is built as a rest API using spring boot.

The implementation will be discussed as the working scenario:

Authentication



The authentication is done by providing a username, password, and role in the header or the request. Each request needs to pass the filter. The reply messages are valid, not valid, and set a new password.

Passwords are encrypted by SHA-512 except for the default password which the database will start with because it will change after the first operation on the database.

The authentication filter is built using a spring boot filter with configuration for the filter.

AuthenticationFilter		
m	AuthenticationFilter()	
m	startValidation(ServletRequest, boolean, HttpServletRequest)	void
m	redirectToError(ServletRequest, HttpServletRequest)	ServletRequest
m	checkIfValidUser(ServletRequest, Map<String, Object>)	void
m	doFilter(ServletRequest, ServletResponse, FilterChain)	void
m	checkIfValidHeader(String, String, String)	void
m	updateAdminFirstTime(HttpServletRequest, Map<String, Object>, String, String)	boolean
m	checkIfNewPasswordSet()	void
m	destroy()	void

Users can use “/login” to check if the username, password, and role are valid, in case not valid they will be redirected to “/error” with an error message like not valid or set a new password.

```
@RestController
public class LoginController {

    @RequestMapping(value="/error/{error}")
    public String errorMessage(@PathVariable("error") String error){
        return error;
    }

    @PostMapping("/login")
    public String validUser() throws JsonProcessingException {
        return "valid";
    }
}
```

Creating Schema

The admin should create a collection by providing a sample of the object with default values. Also, the admin needs to provide the namespace and schema name.

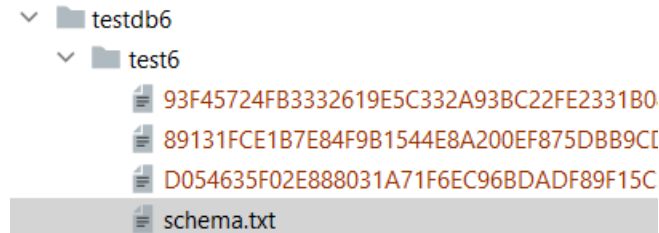
Like if these data are provided:

```
1  {  
2    .... "name": "mamoon",  
3    .... "family": "alshamali",  
4    .... "age": 23,  
5    .... "job": "software engineer",  
6    .... "hate": "evil"  
7  }
```

A generated schema will be this:

```
{  
  "additionalProperties": false,  
  "title": "test6",  
  "type": "object",  
  "properties": {  
    "name": {  
      "type": "string"  
    },  
    "hate": {  
      "type": "string"  
    },  
    "family": {  
      "type": "string"  
    },  
    "job": {  
      "type": "string"  
    },  
    "age": {  
      "type": "number"  
    }  
  }  
}
```

After the schema is generated successfully, it will be stored in the memory and the storage. Also, a node will be created using the docker container.



The admins can add their schema using the post method to this path “/v2/add/schema/{database name}/{schema name}”.

```
@PostMapping("/v2/add/schema/{databaseName}/{schemaName}")
public ResponseEntity<Object> addSchema(
    @RequestBody Map<String, Object> body,
    @PathVariable("schemaName") String schemaName,
    @PathVariable("databaseName") String databaseName,
    ServletRequest req) throws IOException {
    String role=(String)req.getAttribute("role");
    if(schemaName.equals("user")&&databaseName.equals("users")&&!role.equals("admin"))
        return new ResponseEntity<>(HttpStatus.FORBIDDEN);
    databaseDAO.addSchema(databaseName, schemaName, body);
    return ResponseEntity.ok().build();
}
```

There is also a class that represents the schema

SchemaObject	
schemaJson	Map<String, Object>
name	String
SchemaObject(String, Map<String, Object>)	
SchemaObject()	
toString()	String
createSchema(String, Map<String, Object>)	SchemaObject
setName(String)	void
getSchemaJson()	Map<String, Object>
getName()	String
setSchemaJson(Map<String, Object>)	void
hashCode()	int
equals(Object)	boolean

Adding objects

After the schema is created the objects will be added to the collection, and each object needs to pass the validation with the schema. After validating the object an id will be created and added to the object. The Object will be stored in the memory and in the storage with its id name as shown in the previous figure.

The id for the object is generated using the UUIDv4 with SHA-256. By this, it is almost impossible for two objects to have the same id.

```
public static String getUniqueID() throws NoSuchAlgorithmException {  
    MessageDigest salt = MessageDigest.getInstance("SHA-256");  
    salt.update(UUID.randomUUID().toString().getBytes(StandardCharsets.UTF_8));  
    return bytesToHex(salt.digest());  
}
```

Also, there is a class representing the objects called document.

Document	
Document()	
Document(String, Map<String, Object>)	
setId(String)	void
getFields()	Map<String, Object>
createDataObject(String, Map<String, Object>)	Document
getId()	String
toString()	String
hashCode()	int
equals(Object)	boolean
setData(Map<String, Object>)	void
getData()	Map<String, Object>

Collection

After having the document and schema, we can have a document collection class, which will present a list of documents related to the same schema. After each document is added to the collection the index will be updated.

DocumentCollection		
f	data	HashMap<String, Document>
f	indexByJsonProperty	IndexByJsonProperty
f	schema	SchemaObject
f	indexedProperties	List<String>
m	isDataEmpty()	boolean
m	getIndexedProperties()	List<String>
m	equals(Object)	boolean
m	getIndexByJsonProperty()	IndexByJsonProperty
m	setIndexedProperties(List<String>)	void
m	setData(HashMap<String, Document>)	void
m	buildIndexMulti(List<String>)	void
m	getDataAsJson()	List<Map<String, Object>>
m	remove(String)	void
m	addDataObject(Document)	void
m	setIndexByJsonProperty(IndexByJsonProperty)	void
m	toString()	String
m	getData()	HashMap<String, Document>
m	hashCode()	int
m	findInCollection(String)	List<Map<String, Object>>
m	createCollection(SchemaObject)	DocumentCollection
m	getSchema()	SchemaObject
m	setSchema(SchemaObject)	void

Index

The index method used is the inverse index, which is used a lot in many different databases. It has a slow build and fast read but after optimizing it has a fast building and fast reading.

The index building will start from the first object added to the collection and with each object added or deleted or updated the index will be updated not rebuilt, there is no case the index needs to rebuild by this slow-building problem is solved.

IndexByJsonProperty		
f	documentCount	HashMap <String, Integer >
f	sourcesCount	int
f	index	HashMap <String, HashSet <Integer > >
f	sources	Map <Integer, String >
f	built	boolean
f	indexedProperties	List <String >
m	IndexByJsonProperty (HashMap <String, Document >, List <String >)	
m	IndexByJsonProperty ()	
m	createEmptyIndex ()	IndexByJsonProperty
m	build (HashMap <String, Document >, List <String >)	void
m	delete (List <String >, Document)	void
m	doIndexing (List <String >, int, Document)	void
m	find (String)	List <String >
m	createMultiIndex (HashMap <String, Document >, List <String >) IndexByJsonProperty	
m	update (Document)	void
d	index	HashMap <String, HashSet <Integer > >
d	sources	Map <Integer, String >
d	sourcesCount	int
d	documentCount	HashMap <String, Integer >
d	indexedProperties	List <String >
d	built	boolean

Value	File Count
Year=1999	1, 2, 5
Name=Ali	1, 3, 4, 5
Job=engineer	2, 7, 8, 9

File Count	File ID/ Name
1	56084BEA7101AAB657D8983BBA62C1D8F9D4E1B76DAC367057DF4FD98628E89C
2	F23F3B8C6427DBB61E7F90238A4078D707718AF11AD5A6AE73CC4AD537BDF134
3	44D82DEC172A4DC233EE4D73E0DA46EFD9045C93868B94D38117F404A3F3E635
4	4109CEA3C87EA83A89A8AE547D23D0962EDEE9A620C0CFD207D4D347D10117A3

Database

Having multiple collections under the same namespace, a database class can be created to represent the related collections.

Database		
f	name	String
f	collections	Map<String, DocumentCollection>
m	Database(String)	
m	Database()	
m	getCollections()	Map<String, DocumentCollection>
m	equals(Object)	boolean
m	setCollections(Map<String, DocumentCollection>)	void
m	toString()	String
m	getName()	String
m	createDatabase(String)	Database
m	addCollection(String, DocumentCollection)	void
m	setName(String)	void
m	getCollection(String)	Optional<DocumentCollection>
m	hashCode()	int

And since there will multiple databases in the system, a map can be used to access different databases. Using maps will make fast reads.

Node Manager

Creating and deleting the nodes will make the necessary to use a class to do this job. Creating a node manager class to create, update, and delete nodes. This class can be classified as mediator because it is acting with many classes to manage the creating, deleting, and updating nodes.

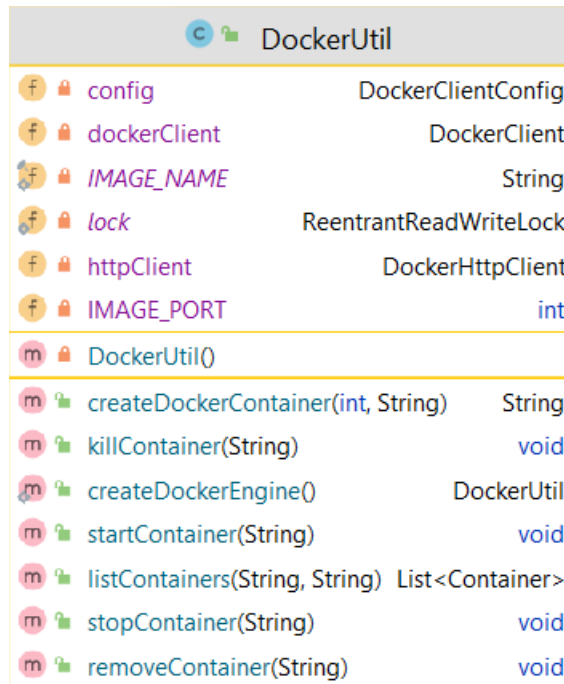
Creating nodes needs to map them to an available port on the local host to able the user to use the node. Then save this port to use when trying to redirect the user-specific node.

NodeManagerV2		
f	docker	Docker
f	nodesIP	Map<String, String>
f	restTemplate	RestTemplate
f	lock	ReentrantReadWriteLock
f	nodeUtil	NodeUtil
m	NodeManagerV2 ()	
m	createNode (String, String)	void
m	destroyAllNodes ()	void
m	updateNode (String, String, DocumentCollection)	void
m	checkIfNodeCreated (String, String)	boolean
m	createReadNodeManager ()	NodeManagerV2
m	removeNode (String, String)	void
m	updateNodeRestTemplate (String, int, String, DocumentCollection , Optional <Integer >)	void
m	setDatabaseNodesData (Map<String, Database >)	void
m	startAllNodes (Map<String, Database >)	void
m	createNodes (String, String, int)	void
m	pingNode (String)	boolean
m	getPort (String, String)	Optional <Integer >
m	removeNodes (String, String)	void
m	deleteOldDataNodes (Map<String, Database >)	void
m	checkNodeResponse (String, int, Optional <Integer >)	boolean
m	startDatabaseNodes (Map<String, Database >)	void

Also, this class is responsible for saving the IP address of the node like 172.17.0.9/16 to able the master from pinging and updating the nodes.

Docker Util

This class is responsible for docker client communication, like create container, stop container, remove container, list containers, and getting IP address of the containers.



DockerUtil		
f	config	DockerClientConfig
f	dockerClient	DockerClient
f	IMAGE_NAME	String
f	lock	ReentrantReadWriteLock
f	httpClient	DockerHttpClient
f	IMAGE_PORT	int
m DockerUtil()		
m	createDockerContainer(int, String)	String
m	killContainer(String)	void
m	createDockerEngine()	DockerUtil
m	startContainer(String)	void
m	listContainers(String, String)	List<Container>
m	stopContainer(String)	void
m	removeContainer(String)	void

This class was responsible for creating and detecting the initial network but to able the master to work in Kubernetes environments, it was deleted.




















Also, this class need multithreading to able functioning correctly.

The library which used in the class:

<https://github.com/docker-java/docker-java>

Node Util

This class is responsible for naming for the counter, store localhost port value, and load balancing of the counter (request balancing). The load balancing for the nodes is round robin, which is synchronized.

NodeUtil		
 	<i>nodeConnectionIndex</i>	Map<String, Integer>
 	<i>lock</i>	ReentrantReadWriteLock
 	<i>nodes</i>	Map<String, Integer>
 	<i>nodeInstanceNumber</i>	Map<String, Integer>
	NodeUtil()	
	storeNodeData(String, Integer)	void
	getPortRobinBalancing(String, String)	Optional<Integer>
	checkIfNodeCreated(String, String)	boolean
	getPort(String)	Optional<Integer>
	removeNodes(String, String)	void
	getPortsForCollection(String, String)	Optional<List<String>>
	createName(String, String)	String
	removeNode(String, String)	Optional<String>
	getAllNodes()	Set<String>
	getNumberOfInstance(String, String)	int

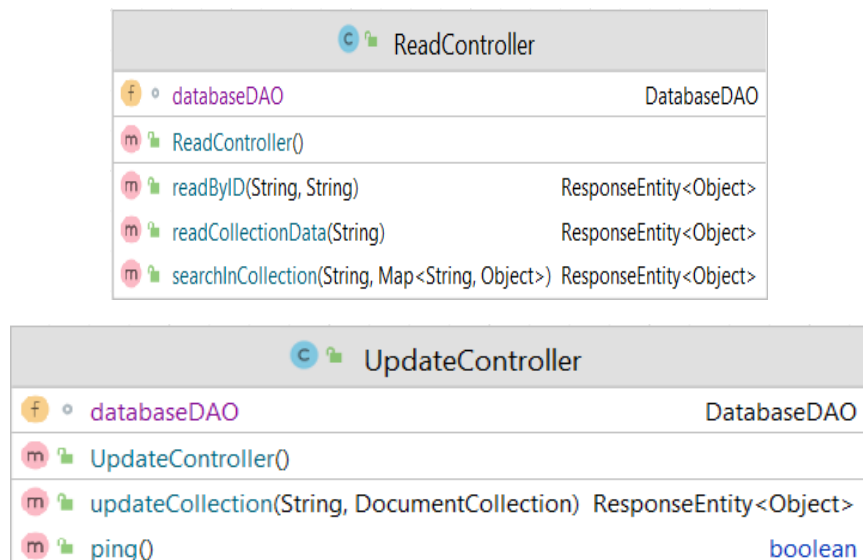
Node

The node is a rest spring boot application with 4 operations which are “read all data”, “read by id”, “search”, and “update collection”.

The node will contain only one collection to make read fast and this is similar to the k mean concept where similar data is together. The node will be updated every time a document is added or deleted or updated. This also means the master controller is acting as a data center.

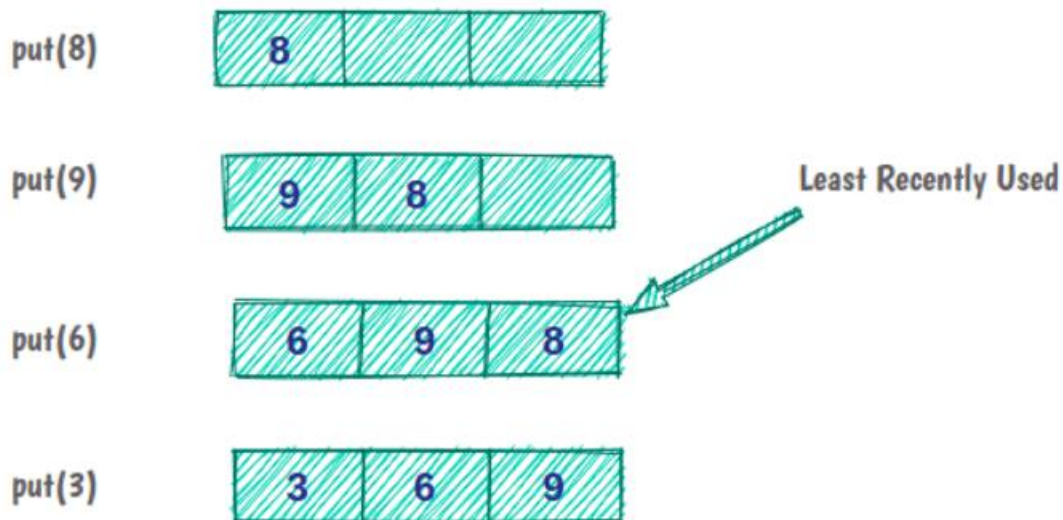
To make reading fast, the communication between the user and the node will be between the user and the node. The master will act as a gateway to redirect requests to the specific node to save the master from being a bottleneck. What is more, the authentication will be done on the master filter so this will result in less work for the nodes.

To make reads faster data is stored in the memory and multithreading LRU caching has been used. What is more, an advanced locking is used which allows multiple readers and one writer. And in case of writing is locked, the users will read from an old version of data until the data in the node is updated.



Caching

The Least Recently Used (LRU) cache is a cache eviction algorithm that organizes elements in order of use. In LRU, as the name suggests, the element that hasn't been used for the longest time will be evicted from the cache.



LRU cache features:

- All operations should run in order of $O(1)$
- The cache has a limited size
- It's mandatory that all cache operations support concurrency
- If the cache is full, adding a new item must invoke the LRU strategy

The LRU was used in the node to read, for the current version of the master, it doesn't use the LRU cache. Using the cache in the nodes reduces the needed time to get data from nodes.

Note that the source code of the LRU cache was taken from

<https://www.baeldung.com/java-lru-cache>

Database Operations

Anything related to adding, deleting, updating, index creation, update, will be done on the Master controller. While reading will be on nodes, the reading also can be done on the master and that will be optional or may be needed like reading users' data.

The database operations are multithreaded using read-write locks, the node threaded in advanced way like having to type of locks the temporary lock which used in case of updating the node.

The node threading example:

```
@Override
public Optional<String> searchDocumentByID(String schemaName, String id) {
    if (lock.isWriteLocked() && !collectionName.equals("empty")) {
        try {
            lockTmp.readLock().lock();
            return searchDocumentByID(schemaName, id, collectionTmp);
        } finally {
            lockTmp.readLock().unlock();
        }
    } else if (collectionName.equals("empty")) {
        return Optional.empty();
    } else {
        try {
            lock.readLock().lock();
            return searchDocumentByID(schemaName, id, collection);
        } finally {
            lock.readLock().unlock();
        }
    }
}
```

These locks support multiple readers and one writer. What is more, these locks support fairness.

The node DAO

DAO		
(m)	search(String, Map<String, Object>)	Optional<String>
(m)	searchDocumentByID(String, String)	Optional<String>
(m)	readData(String)	Optional<String>
(m)	setCollection(String, DocumentCollection)	void



















The Master DAO

DAO		
(m)	deleteCollection(String, String)	boolean
(m)	getCollection(String, String)	Optional<DocumentCollection>
(m)	importDatabase(MultipartFile)	void
(m)	exportDatabase(String)	Optional<File>
(m)	addObject(String, String, Map<String, Object>)	boolean
(m)	updateV2(String, String, Map<String, Object>[])	boolean
(m)	searchRead(String, String, Map<String, Object>)	Optional<JSONArray>
(m)	readStoredData()	void
(m)	deleteByProperties(String, String, Map<String, Object>)	boolean
(m)	deleteByID(String, String, String)	boolean
(m)	addSchema(String, String, Map<String, Object>)	void
(m)	searchDocumentByID(String, String, String)	Optional<String>
(m)	readData(String, String)	Optional<String>

File Input and Output

- ▼ database
 - ▼ animal-shop
 - > cat
 - ▼ dog
 - 8A27D6A8B0D4C05285789A20187CA10BF1D8925D2334B8DA6C8182773F36C9E9.txt
 - 85115DB823FB8AB52A5F0773731731B2CE6693BC59E9CF41A971EB20A9510BF3.txt
 - B9D32AD53169587018EEA235B593D29715B4AE4721CE3064131825EFBD7F6FCD.txt
 - F34FA9002FB99EDFA176675C07B8EA7BA30FC7E9330D652A120C93590BCC6234.txt
 - schema.txt
 - ▼ testdb6
 - ▼ test6
 - 93F45724FB3332619E5C332A93BC22FE2331B0804D683CFA1257DA7131A5591E.txt
 - 89131FCE1B7E84F9B1544E8A200EF875DBB9CD3EE69329AAE6E3CABBC7632B2E.txt
 - D054635F02E888031A71F6EC96BDADF89F15C3A0819DC48BF875E81E48346748.txt
 - schema.txt
 - ▼ users
 - ▼ user
 - 0FC5B71FEE05F12D75750033EF4DF8673F7892B028778FF5CB9CB451A39F1666.txt
 - 38F723D76B7BD3612CFE0398EFBC2B0FB87DA60F5709562D4746100F1D4760CE.txt
 - D7832363F02023C03B32DDEB6D51E88E3548FDF8C908678F13D6206519AAD7AB.txt
 - schema.txt

FileOperation

-   readAllDatabases () Optional <HashMap<String, Database>>
-   exportDatabase (String) File
-   importDatabase (MultipartFile) void
-   readDatabase (String) Optional <Database>
-   writeCollection (String, DocumentCollection) void
-   delete (String, String, String) void
-   deleteCollection (String, String) void
-   updateFile (String, String, JSONObject) void
-   readCollection (String, String) Optional <DocumentCollection>

Database Import and Export

The database can be exported and imported as a zip file, in case of import the nodes will be created or updated.

ImportExportController		
f	databaseDAO	DatabaseDAO
m	ImportExportController()	
m	exportDatabase(String)	ResponseEntity<Resource>
m	importDatabase(MultipartFile)	Object

```
@Override
public Optional<File> exportDatabase(String databaseName) throws IOException {
    lock.readLock().lock();
    try {
        if (!isDatabaseCreated(databaseName))
            return Optional.empty();
        return Optional.of(fileIO.exportDatabase(databaseName));
    } finally {
        lock.readLock().unlock();
    }
}

@Override
public void importDatabase(MultipartFile zipFile) throws IOException, InterruptedException, NoSuchAlgorithmException {
    String fileName = zipFile.getOriginalFilename();
    lock.writeLock().lock();
    try {
        fileIO.importDatabase(zipFile);
        Optional<Database> database = fileIO.readDatabase(fileName);
        if (!database.isPresent())
            return;
        databases.put(zipFile.getOriginalFilename(), database.get());
    } finally {
        lock.writeLock().unlock();
    }
    if (fileName.equals("users"))
        return;
    UpdateNodesAfterImport(fileName);
}
```

Data Structure

Many data structures have been used such as:

- General maps in the system:
 - `private static Map<String, Database> databases;`
 - `private Map<String, DocumentCollection> collections;`
 - `Map<String, Object> data;`
 - `private static Map<String, Integer> collectionsPorts;`

The map is important to receive different kinds of objects, like when receiving schema:

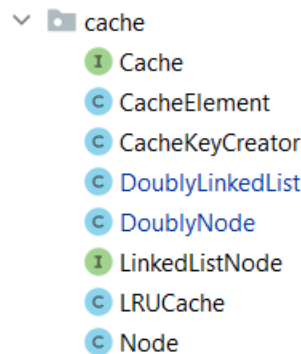
```
@RequestBody Map<String, Object> body,
```

- LRU cache

The source code was taken from a GitHub repository

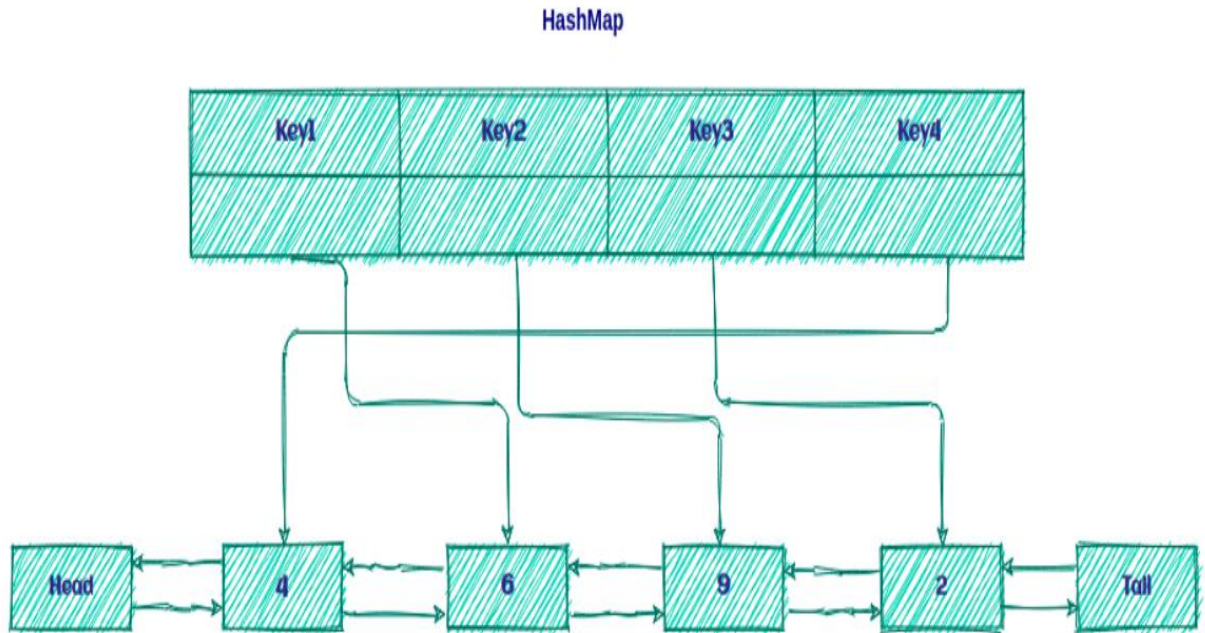
<https://github.com/eugenp/tutorials/tree/master/data-structures/src/main/java/com/baeldung/lrucache>

many data structures are used to build LRU cache algorithm



The LRU is a combination of the Doubly LinkedList and the HashMap

```
private final Map<K, LinkedListNode<CacheElement<K, V>>> linkedListNodeMap= new ConcurrentHashMap<>(size);  
private final DoublyLinkedList<CacheElement<K, V>> doublyLinkedList= new DoublyLinkedList<>();
```



- Inverted Index

- An inverted index is an index data structure storing a mapping from content, such as words or numbers, to its locations in a document or a set of documents. In simple words, it is a HashMap-like data structure that directs you from a word to a document.
- Advantages of Inverted Index are:
 - It is easy to develop.
 - It is the most popular data structure used in document retrieval systems, used on a large scale for example in search engines.

○

```
private Map<Integer, String> sources;
private HashMap<String, HashSet<Integer>> index;
private List<String> indexedProperties;
private HashMap<String, Integer> documentCount;
```

Multithreading the locks

Reentrant Read Write Lock

Read Write Lock is used to make a multithreading system, the advantages of this type of lock allows multiple readers at the same time and one writer at the same time. What is more, allow the fairness which means the priority for the longest waited thread.

- Read lock: If no thread has requested the write lock and the lock for writing, then multiple threads can lock the lock for reading. It means multiple threads can read the data at the very moment, as long as there's no thread to write the data or to update the data.
- Write Lock: If no threads are writing or reading, only one thread at a moment can lock the lock for writing. Other threads have to wait until the lock gets released. It means, that only one thread can write the data at that very moment, and other threads have to wait.

Using try and finally open and close the locks, and this is important to avoid getting locked in case of an interruption thread stopped for any reason.

The threading in the node is handled in a better way than master for the current version of the system. The node uses two locks to be able fast to read in case of an update.

These locks were used in many classes: LRU cache, database DAO, node util, and docker util.

An example of threading in the node:

```
private static DocumentCollection collection=new DocumentCollection();
private static DocumentCollection collectionTmp=new DocumentCollection();
ReentrantReadWriteLock lock= new ReentrantReadWriteLock( fair: true);
ReentrantReadWriteLock lockTmp= new ReentrantReadWriteLock( fair: true);

@Override
public void setCollection(String schemaName, DocumentCollection collection) {
    try {
        lockTmp.writeLock().lock();
        if (!collectionName.equals("empty")) {
            collectionTmp = collection;
        }
    } finally {
        lockTmp.writeLock().unlock();
    }
    lock.writeLock().lock();
    try {
        collectionName = schemaName;
        DatabaseDAO.collection = collection;
        if(!readCache.isEmpty())readCache.clear();
        System.out.println("collection name is" + collectionName);
    } finally {
        lock.writeLock().unlock();
    }
}

@Override
public Optional<String> readData(String schemaName) throws IOException {
    if (lock.isWriteLocked() && !collectionName.equals("empty")) {
        try {
            lockTmp.readLock().lock();
            return readData(schemaName, collectionTmp);
        } finally {
            lockTmp.readLock().unlock();
        }
    } else if (collectionName.equals("empty")) {
        return Optional.empty();
    } else {
        try {
            lock.readLock().lock();
            return readData(schemaName, collection);
        } finally {
            lock.readLock().unlock();
        }
    }
}
```

Synchronized

Sometimes the powerful read write locks is not needed, a simple multithreading way can do the job like synchronized. This was used in around three places in the code like:

```
synchronized (this) {  
    if (index + 1 > instanceNumber)  
        | nodeConnectionIndex.put(name, 1);  
    else  
        | nodeConnectionIndex.put(name, index + 1);  
    return getPort( nodeName: name + "-" + index);  
}
```

```
synchronized (this) {  
    | nodesIP.put(nodeName, ipAddress);  
}
```

```
synchronized (this) {  
    return nodeUtil.getPortRobinBalancing(databaseName, collectionName);  
}
```

Scalability and Consistency

The database design was inspired by the Cassandra Apache documentation:

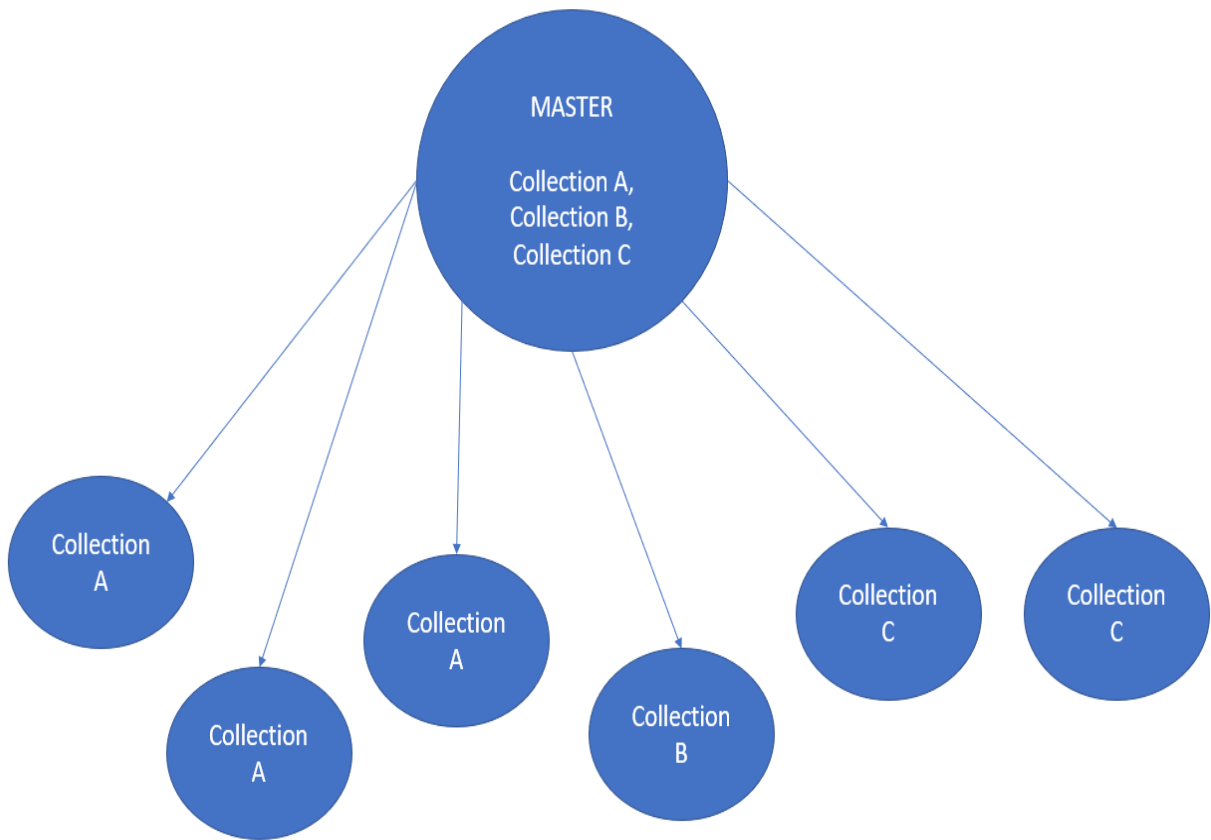
<https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/architecture/archDataDistributeDistribute.html>

<https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/architecture/archDataDistributeReplication.html>

The system uses a simple strategy which is one data center and multiple nodes containing a partition of the data. This strategy is chosen because there is no need for multiple data centers since this cluster is not for multi regions.

The master will create two nodes for each collection. Like if there are 5 collections there will be 10 nodes. This idea was inspired by the k means algorithm which clusters the similar data together and because the collection has similar data, the data clustering stage is already done.

The admin can add nodes or delete nodes by just submitting a number to the server to create this number or delete this number of the nodes. As an example, the nodes for collection A are 5 and for collection A are 1. In case admin put 0 nodes that will forbid the reading for the collection.





















To make reading fast and avoid making the master controller a bottleneck for the requests, the responsibility of the master for the normal request is authentication then redirecting the user to the correct node to read data.

The node which will be chosen for request is depending on the Round Robin algorithm. The round robin algorithm work as follows for two nodes:

Request NO	Two Nodes
1	Node 1
2	Node 2
3	Node 1
4	Node 2

For the current version of the system, the master controller is responsible for:

- Create all nodes for any added schema or the saved schema in the storage when the system is restarted after falls. The master will choose at random an available port and map the node to it.
- Free up the resources by destroying the nodes when the system is shut down normally or restarting after a fall.
- Update each node when data is added or updated or deleted. In case of deleting a collection, the related node will be deleted as well.
- The self-healing abilities are (for the current version):
 - If the system falls, the nodes will stay working. when it is restarted the nodes which are still running or stopped or exited from the previous run will be deleted and reinitialized again. After initializing the nodes, the nodes will be updated with stored data in the storage.
 - The data is stored in the memory as well as in the storage.
 - In the memory to able for fast modification and reading.
 - In the storage to make a backup in case of the fall of the system
 - Every time the system is started, it will create the nodes and update them as a part of the starting the master controller.

ReadNode		
	 removeNodes(String, String)	void
	 updateNode(String, String, DocumentCollection)	void
	 startAllNodes(Map<String, Database>)	void
	 createNodes(String, String, int)	void
	 destroyAllNodes()	void
	 createNode(String, String)	void
	 checkIfNodeCreated(String, String)	boolean
	 removeNode(String, String)	void
	 getPort(String, String)	Optional<Integer>

How does the Master redirect the requests? The Request will be redirected as following

Get request:

```
@GetMapping("/node/read/{databaseName}/{schemaName}")
public void readCollection(
    @PathVariable("schemaName") String schemaName,
    @PathVariable("databaseName") String databaseName,
    HttpServletResponse response) throws IOException {
    Optional<Integer> port = nodeManager.getPort(databaseName, schemaName);
    if (!port.isPresent())
        return;
    String nodeUri = HOST + port.get() + "/read/collection/" + schemaName;
    response.sendRedirect(nodeUri);
}
```

Post request:

According to RFC2616 with HTTP/1.1 you can send a 307-response code, which will make the user-agent repeat its POST request to the provided host.

```
@PostMapping("/node/search/{databaseName}/{schemaName}")
public void search(
    @PathVariable("databaseName") String databaseName,
    @PathVariable("schemaName") String schemaName,
    @RequestBody Map<String, Object> fields,
    HttpServletResponse response) {
    Optional<Integer> port = nodeManager.getPort(databaseName, schemaName);
    if (!port.isPresent())
        return;
    String nodeUri = HOST + port.get() + "/search/collection/" + schemaName;
    response.setStatus(HttpServletResponse.SC_TEMPORARY_REDIRECT);
    response.setHeader("Location", nodeUri);
}
```

The database will not accept any object not valid to the generated schema, the schema validation operations are done as it is explained on the website:

<https://www.baeldung.com/introduction-to-json-schema-in-java>

And this validation work as it is mentioned on this site:

<https://json-schema.org/understanding-json-schema/reference/object.html>

For the current version of the system, the validation will not valid an object for these reasons:

- Not valid type like the value is a string and provided is an integer.
- Not valid Key like if the does not exist in the schema.

The system will validate the missing keys since it is valid by default.

Security

A Basic authentication system has been used, the user needs to provide his username, password, and role into the header of the request. The default username and password for the first run for the system are:

- Username: super
- Password: 1234
- Role: admin

The first time using the system, the user must provide a header containing a new password otherwise no login.

The passwords are encrypted in the database using SHA-512. Almost all requests must pass the authentication every time needed to access the database. This is done by using spring filters and filter configuration.

The admin can add users and update their data using the master. Also, the admin can delete other users, but the admin can't delete the root admin otherwise the system will not function correctly in some cases.

Other security problems are in the nodes, the nodes don't have an authentication method. The only thing needs to access the node is knowing the port number which generated at random in some cases the port was 30000, 60000, 5000, ...etc. but since every time rerun the master a new port will be chosen which give some of security. To add more security for the node name must be provided which is encrypted by SHA-512 to access the data.

The protocol

There are two types of protocols, a protocol between the client and the server, the other protocol is between the master and the nodes. Some of the operations can't be done for normal user, it needs an admin role to perform.

This is how to check if valid user or not or need to set a new password. The needed values are username, password, and role which will be placed in the header of the request.

```
@RequestMapping(value="/error/{error}")
public String errorMessage(@PathVariable("error") String error) { return error; }

@PostMapping("/login")
public String validUser() throws JsonProcessingException {
    return "valid";
}
```

This is the how to add schema and objects to the database, to create the schema the user must provide a sample of the object and a schema will be generated

```
@PostMapping("/v2/add/schema/{databaseName}/{schemaName}")
public ResponseEntity<Object> addSchema(
    @RequestBody Map<String, Object> body,
    @PathVariable("schemaName") String schemaName,
    @PathVariable("databaseName") String databaseName,
    ServletRequest req) throws IOException {...}

@PostMapping("/v2/add/document/{databaseName}/{schemaName}")
public ResponseEntity<Object> addDocument(
    @RequestBody Map<String, Object> body,
    @PathVariable("schemaName") String schemaName,
    @PathVariable("databaseName") String databaseName,
    ServletRequest req) throws NoSuchAlgorithmException, IOException, InterruptedException {...}
```

This is how to update an object; the user needs to provide two JSON objects. The first object is the search values like every object contains these keys and values will be updated. The second object is the new data which saved in the objects.

```
@PostMapping("/v2/update/{databaseName}/{schemaName}")
public ResponseEntity<Object> update(
    @RequestBody Map<String, Object>[] fields,
    @PathVariable("schemaName") String schemaName,
    @PathVariable("databaseName") String databaseName,
    ServletRequest req) throws NoSuchAlgorithmException, IOException, InterruptedException {...}
```

This is how to delete an object, the user has two provide the id or provide keys and value

```
@PostMapping("/v2/delete/{databaseName}/{schemaName}")
public ResponseEntity<Object> delete(
    @RequestBody Map<String, Object> fields,
    @PathVariable("schemaName") String schemaName,
    @PathVariable("databaseName") String databaseName,
    ServletRequest req) throws IOException, InterruptedException {...}

@GetMapping("/v2/delete/{databaseName}/{schemaName}/{id}")
public ResponseEntity<Object> deleteByID(
    @PathVariable("databaseName") String databaseName,
    @PathVariable("schemaName") String schemaName,
    @PathVariable("id") String id,
    ServletRequest req) throws IOException, InterruptedException {...}
```

This is how to delete a collection on the database

```
@GetMapping("/v2/delete/collection/{databaseName}/{collectionName}")
public ResponseEntity<Object> deleteCollection(
    @PathVariable("databaseName") String databaseName,
    @PathVariable("collectionName") String collectionName,
    ServletRequest req) throws IOException {...}
```

This is how the user import and export a database, in the import case the user needs to provide the same exported file.

```
@GetMapping(path = "/v2/export/{databaseName}")
public ResponseEntity<Resource> exportDatabase(
    @PathVariable("databaseName") String databaseName) throws IOException {...}

@PostMapping(path = "/v2/import")
public Object importDatabase(
    @RequestParam("database") MultipartFile zipFile) throws IOException, InterruptedException {...}
```

This is how the user read from the nodes

```
@GetMapping("/node/read/{databaseName}/{schemaName}")
public void readCollection(
    @PathVariable("schemaName") String schemaName,
    @PathVariable("databaseName") String databaseName,
    HttpServletResponse response) throws IOException {...}

@GetMapping("/node/read/{databaseName}/{schemaName}/{id}")
public void readByID(
    @PathVariable("databaseName") String databaseName,
    @PathVariable("schemaName") String schemaName,
    @PathVariable("id") String id,
    HttpServletResponse response) throws IOException {...}

@PostMapping("/node/search/{databaseName}/{schemaName}")
public void search(
    @PathVariable("databaseName") String databaseName,
    @PathVariable("schemaName") String schemaName,
    @RequestBody Map<String, Object> fields,
    HttpServletResponse response) {...}
```

This is how the user read from the master, it is needed because users' data will not sent to the nodes.

```
@GetMapping(path="/v2/read/document/{databaseName}/{schemaName}", produces= MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<Object> readDocumentOfSchema(
    @PathVariable("schemaName") String schemaName,
    @PathVariable("databaseName") String databaseName) throws IOException {...}

@GetMapping(path="/v2/search/document/{databaseName}/{schemaName}/{id}")
public ResponseEntity<Object> searchForDocumentById(
    @PathVariable("schemaName") String schemaName,
    @PathVariable("databaseName") String databaseName,
    @PathVariable("id") String id){...}

@PostMapping(path="/v2/search/document/{databaseName}/{schemaName}")
public ResponseEntity<Object> searchByProperties(
    @PathVariable("schemaName") String schemaName,
    @PathVariable("databaseName") String databaseName,
    @RequestBody Map<String, Object> fields) throws JsonProcessingException {...}
```

This is how the master update the nodes

```
String uri = "http://" + nodesIP.get(nodeName + "-" + instanceNumber) + ":" + 8090;
String nodeUri = uri + "/collection/"
    + SHA512.toHexString(SHA512.getSHA(collectionName));
System.out.println(nodeUri);
HttpEntity<DocumentCollection> request = new HttpEntity<>(collection);
restTemplate.postForObject(nodeUri, request, DocumentCollection.class);
```


This is how the node communicate with master and the user

```
@GetMapping("/read/collection/{collectionName}")
public ResponseEntity<Object> readCollectionData(
    @PathVariable("collectionName") String collectionName) {...}

@GetMapping("/read/collection/{collectionName}/{id}")
public ResponseEntity<Object> readByID(
    @PathVariable("collectionName") String collectionName,
    @PathVariable("id") String id) {...}

@PostMapping("/search/collection/{collectionName}")
public ResponseEntity<Object> searchInCollection(
    @PathVariable("collectionName") String collectionName,
    @RequestBody Map<String, Object> fields) {...}

@PostMapping("/collection/{schemaName}")
public ResponseEntity<Object> updateCollection(
    @PathVariable("schemaName") String schemaName,
    @RequestBody DocumentCollection collection) {...}

@GetMapping("/ping")
public boolean ping() { return true; }
```

Clean Code

The naming

All the names are descriptive and unambiguous. It has a meaningful distinction, searchable. And there are no magic numbers. Maybe some functions names in the code will disagree here because they are too long like “is valid piece to move” and “is rook and king at start place”. Finally, there is no abbreviation and typos.

As example for variables names

```
private static Map<String, Database> databases;  
ReentrantReadWriteLock lock = new ReentrantReadWriteLock(fair: true);  
ReadNode nodeManager = ReadNodeManager.createReadNodeManager();  
private final FileOperation fileIO = DatabaseIO.createIOOperation();
```

As example for function names

ReadNode	
createNode(String, String)	int
destroyAllNodes()	void
checkIfNodeCreated(String, String)	boolean
getPort(String, String)	Optional<Integer>
startAllNodes(Map<String, Database>)	void
updateNode(String, String, DocumentCollection)	void
removeNode(String, String)	int
createNetwork()	void

As example for class names

fileio
DatabaseIO
FileOperation
Index
Index
IndexByJsonProperty
node
ReadNode
ReadNodeManager

Functions

Functions need to be small and do one thing

But it is a hard to keep this principle valid for all functions but at least most functions will follow this principle correctly.

For function arguments, no function needs 4 arguments or more. If the function needs 4 arguments, it will be private function and there is around one private function need 4 arguments in the code. Also, most of the functions are simple and not complex.

As example for doing one thing, small, less than 4 arguments, and simple:

```
public static String getUniqueID() throws NoSuchAlgorithmException {  
    MessageDigest salt = MessageDigest.getInstance("SHA-256");  
    salt.update(UUID.randomUUID().toString().getBytes(StandardCharsets.UTF_8));  
    return bytesToHex(salt.digest());  
}
```

Comments

From the book view, most of the comments are bad and there is a little useful comment. But because of having multiple versions of the system some comments is needed. Like comment a function from old version until making sure it is not needed then the commented code is deleted.

The TODO comments used a lot while writing the code and most of them is deleted after completing the TODO comment.

Some functions not understandable because it is need a technical information.

```
/**
 * According to RFC2616 with HTTP/1.1 you can send 307 response code,
 * which will make user-agent to repeat it's POST request to provided host
 **/
```

Return Null and Pass Null

From the book view, the code should not return or pass null. This is solved by using Optional<V>, Using the Optional allow to return empty.

```
@Override
public Optional<File> exportDatabase(String databaseName) throws IOException {
    lock.readLock().lock();
    try {
        if (!isDatabaseCreated(databaseName))
            return Optional.empty();
        return Optional.of(fileIO.exportDatabase(databaseName));
    } finally {
        lock.readLock().unlock();
    }
}
```

Optional is a container object used to contain not-null objects. Optional object is used to represent null with absent value. This class has various utility methods to facilitate code to handle values as 'available' or 'not available' instead of checking null values.

Classes

Most of classes is small except some classes is very big like database DAO and database IO because these two classes have many functions.

DAO	
<code>searchRead(String, String, Map<String, Object>)</code>	<code>Optional<JSONArray></code>
<code>importDatabase(MultipartFile)</code>	<code>void</code>
<code>readData(String, String)</code>	<code>Optional<String></code>
<code>exportDatabase(String)</code>	<code>Optional<File></code>
<code>addSchema(String, String, Map<String, Object>)</code>	<code>void</code>
<code>searchDocumentById(String, String, String)</code>	<code>Optional<String></code>
<code>readStoredData()</code>	<code>void</code>
<code>deleteCollection(String, String)</code>	<code>boolean</code>
<code>deleteById(String, String, String)</code>	<code>boolean</code>
<code>deleteByProperties(String, String, Map<String, Object>)</code>	<code>boolean</code>
<code>addObject(String, String, Map<String, Object>)</code>	<code>boolean</code>
<code>updateV2(String, String, Map<String, Object>[])</code>	<code>boolean</code>

FileOperation	
<code>deleteCollection(String, String)</code>	<code>void</code>
<code>importDatabase(MultipartFile)</code>	<code>void</code>
<code>readCollection(String, String)</code>	<code>Optional<DocumentCollection></code>
<code>delete(String, String, String)</code>	<code>void</code>
<code>writeCollection(String, DocumentCollection)</code>	<code>void</code>
<code>readAllDatabases()</code>	<code>Optional<HashMap<String, Database>></code>
<code>exportDatabase(String)</code>	<code>File</code>
<code>updateFile(String, String, JSONObject)</code>	<code>void</code>
<code>readDatabase(String)</code>	<code>Optional<Database></code>

Inherit constant

There is no constant inherited in the code.

Encapsulation

All variables are private and have getter and setters to access them.

Effective Java

Static factory

Using static factories over constructors is used in the design but some classes do not have them because the need for empty constructor when auto wire a class. Also, the constructor should be public when using rest template to update nodes.

```
private ReadNodeManager() {  
}
```

```
public static ReadNodeManager createReadNodeManager() { return new ReadNodeManager(); }
```

Avoid creating unnecessary objects

Reusing expensive objects for improved performance, and that's appeared when using a static object to store data instead of reading all the data from the storage every time the database is called. Also, prefer primitives to boxed primitives which occur in many parts of the code.

```
private static Map<String, Database> databases;
```

Override (to string, equal, hash code)

These functions were overridden in most classes like document, schema object, collection, database, index.

For how the hash was coded it was using the One-line hash code method - mediocre performance.

Minimize the accessibility of classes and members

This is done by making all fields private.

```
private static String collectionName="empty";  
private static DocumentCollection collection=new DocumentCollection();  
private static DocumentCollection collectionTmp=new DocumentCollection();
```

In public classes, use accessor methods, not public fields

Each class in the code has accessor methods to its private fields and there are no public fields. Also, the book mentions that to reach the lowest access level, but getter and setters is important in spring boot so it can't be achieved.

Other Items

Favor composition over inheritance

Composition is used widely in the code

Prefer interface to abstract class

No abstract classes has been used in the code only interface has been used.

Prefer list to array

Almost there is no arrays in the code, most of data structure is are list and maps.

Design method signatures carefully

The code agrees with using the naming convention, there is no long parameters list, in the code.

Prefer foreach on traditional for

Foreach is used almost for all loops.

Refer to objects by their interfaces

Most of the code agree with it, some errors resulted from referring using interface and these errors appeared when rest template is used to update the node.

SOLID

single-responsibility principle

almost all the classes in the design follow this principle but having big functions inside the class will make the need to make private functions. And that will result in miss leading the class not following this principle. An example where this appears:

DatabaseDAO		
nodeManager		ReadNode
fileIO		FileOperation
lock		ReentrantReadWriteLock
databases		Map<String, Database>
DatabaseDAO ()		
readData (String, String)		Optional<String>
searchDocumentByID (String, String, String)		Optional<String>
importDatabase (MultipartFile)		void
updateV2 (String, String, Map<String, Object>[])		boolean
deleteByProperties (String, String, Map<String, Object>)		boolean
readStoredData ()		void
searchRead (String, String, Map<String, Object>)		Optional<JSONArray>
isValidJson (JSONObject, JSONObject)		boolean
addSchema (String, String, Map<String, Object>)		void
exportDatabase (String)		Optional<File>
createDatabase (String)		void
updateV2 (Map<String, Object>, JSONArray, DocumentCollection)		List<String>
checkDatabaseAndSchemaExist (String, String)		boolean
getID (JSONArray)		List<String>
deleteByID (String, String, String)		boolean
UpdateNodesAfterImport (String)		void
getSearchValues (Map<String, Object>)		String
addObject (String, String, Map<String, Object>)		boolean
search (String, String, Map<String, Object>, boolean)		Optional<JSONArray>
isDatabaseCreated (String)		boolean
deleteCollection (String, String)		boolean

This class is responsible for the database operations and activating other related functions from different class after storing the data like file IO.

The open-closed principle

The design follows this principle in many places like having index, DAO, file operation, read node, and cache interfaces. These interfaces can be extended and make future version of the system.

The Liskov substitution principle

The design agrees with this principle since there is no change in the behavior in subclasses when inheriting the superclass.

The interface segregation principle

The design agrees and disagrees with this principle, it agrees with the read node interface since it is small, but it disagrees with the game DAO interface because it is too long.

Dependency Inversion Principle

The design agrees with this principle since most the classes are not dependent on low-level concrete classes, instead they depend on abstractions or interfaces.

As mentioned before some problems appeared when applying this principle. In order to solve errors, it needs to break it.

Design pattern

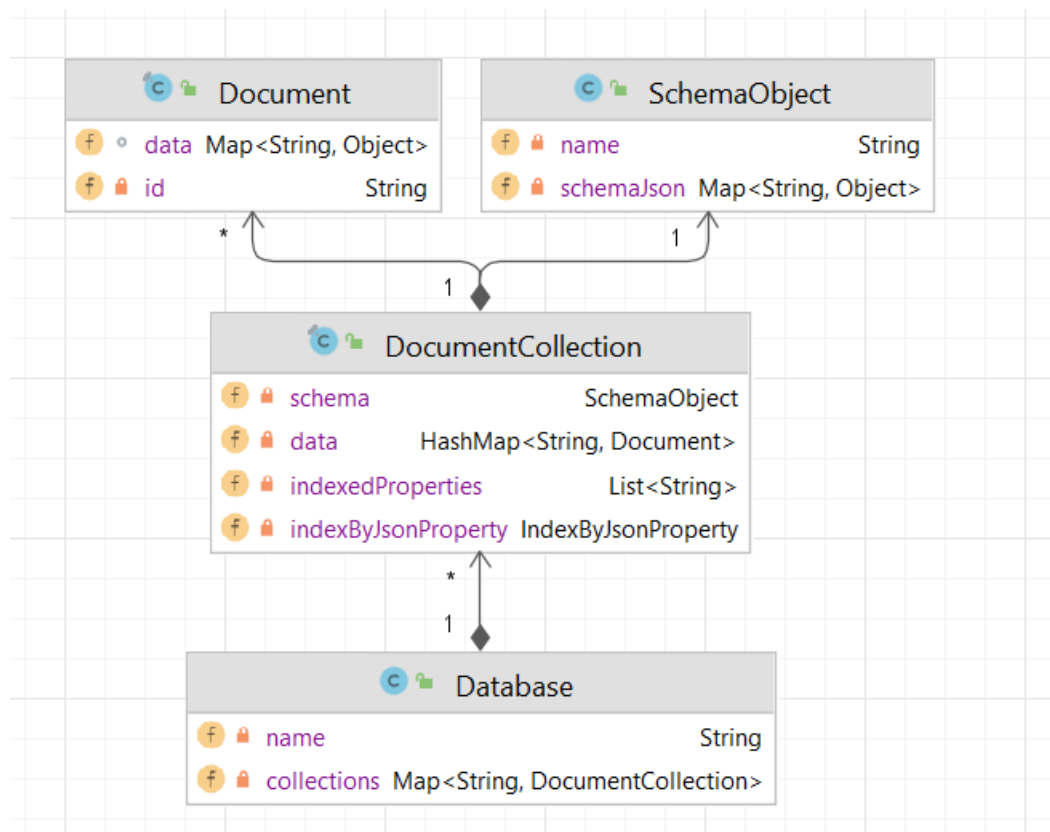
Singleton

Because using spring some design patterns was avoided such as singleton, instead using scope (singleton) like in the cache

```
@Scope("singleton")  
public class LRUCache<K, V> implements Cache<K, V> {
```

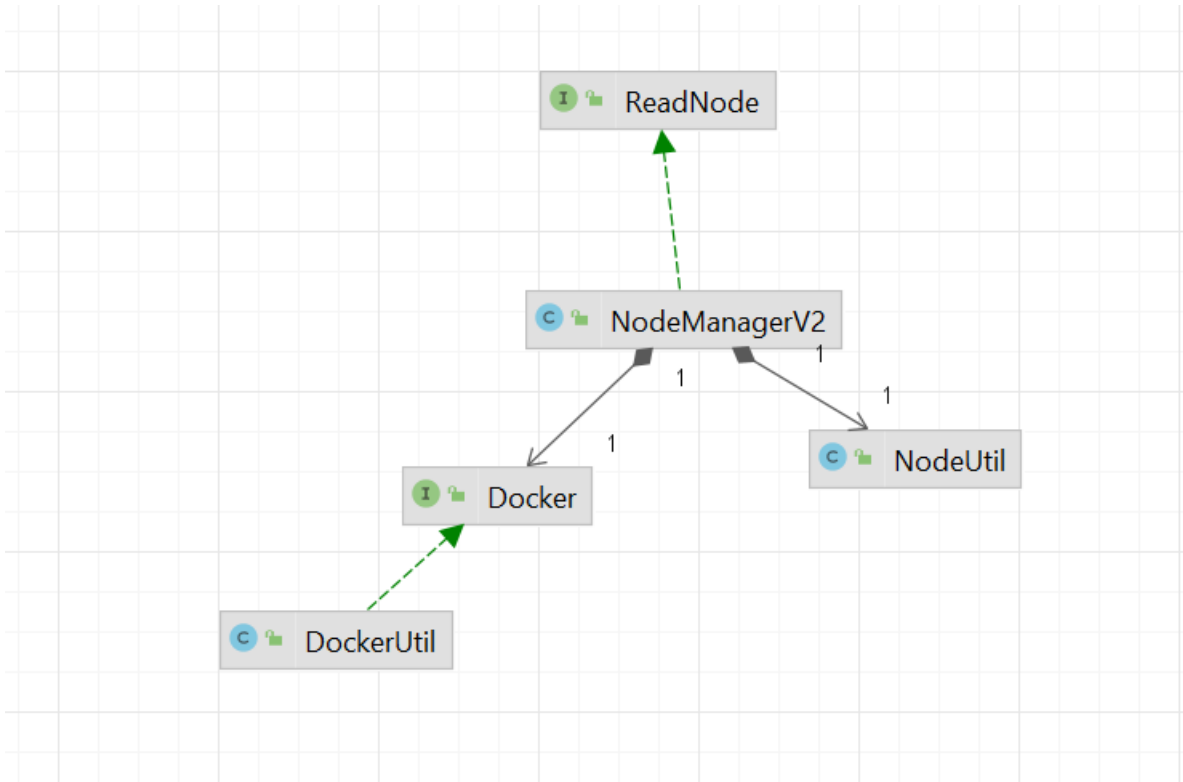
Facade

The facade design pattern can be noticed in Database class



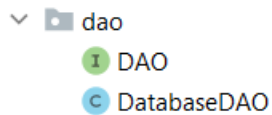
Mediator

Mediator design pattern can be noticed in node manager class and in database DAO class.



DAO

The DAO design pattern can be noticed in the node and in the master



MVC

The MVC design pattern is used in the client application

- ▼ src
 - ▼ main
 - ▼ java
 - ▼ com.mamoon.demodocumentdb
 - ▼ controller
 - CatController
 - CollectionController
 - DogController
 - ErrorController
 - HomeController
 - LoginController
 - LogoutController
 - NodeController
 - UserController
 - ▼ model
 - Cat
 - Dog
 - User
 - > util
 - DemodocumentdbApplication
 - ▼ resources
 - static
 - ▼ templates
 - addCat.html
 - addCollection.html
 - addDog.html
 - addUser.html
 - adminHome.html
 - deleteCat.html
 - deleteDog.html
 - error.html
 - login.html

Template

The template design pattern concept can be noticed in the node manager, since the operation of starting the system will be like a template. First delete free resources in case of starting after the master fall without shut down, then create node. After node is created update it. And this is done every time the system is started.

```
public void startAllNodes(Map<String, Database> databases)
{
    int initializeTime = 5000;
    int removingTime = 3000;
    deleteOldDataNodes(databases);
    Thread.sleep(removingTime);
    startDatabaseNodes(databases);
    Thread.sleep(initializeTime);
    setDatabaseNodesData(databases);
}
```

DevOps

GitHub

GitHub is used to store the project allow for working remotely on different devices and manage the versions of the project.

documntdb Private

● Java Updated 12 hours ago

documentdbnodev2 Private

● Java Updated 2 days ago

Maven

Maven is a build automation tool used primarily for Java projects. It was used to manage the dependency and plugins used in the project and build the application as maven has its own life cycle.

GitHub action

Used to build CICD for the project, the GitHub repository will not accept any push didn't pass the CICD. Also, it provides a way to build a multistage docker image for the project.

The CICD in node project

```

name: Java CI with Maven

on:
  push:
    branches: [ "master" ]
  pull_request:
    branches: [ "master" ]
env:
  cache: enabled
  CACHE_IMAGE: mjhea0/docker-ci-cache
  DOCKER_BUILDKIT: 1.3

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3
    - name: Set up JDK 11
      uses: actions/setup-java@v3
      with:
        java-version: '11'
        distribution: 'temurin'
        cache: maven
    - name: change permission
      run: chmod +x mvnw
    - name: Build with Maven
      run: mvn -B package --file pom.xml
    - name: Set up Maven
      uses: stCarolas/setup-maven@v4.3
      with:
        maven-version: 3.8.2
    - name: Login to Docker Hub
      uses: docker/login-action@v1
      with:
        username: ${ secrets.DOCKER_HUB_USERNAME }
        password: ${ secrets.DOCKER_HUB_PASSWORD }
    - name: Docker Setup Buildx
      uses: docker/setup-buildx-action@v2.0.0
      with:
        buildkitd-flags: 1.3
        install: true
    - name: Build and push
      uses: docker/build-push-action@v2
      with:
        context: .
        file: ./Dockerfile
        push: true
        tags: ${ secrets.DOCKER_HUB_USERNAME }/node-reader-document:2.0

```

Docker

Docker multistage build is used for the development of the node for the current version of the system. Using the advance way to build docker image help in fast pulling the image in the future, low size like 100 mb.

```
# syntax=docker/dockerfile:1.3
FROM openjdk:8-jdk-alpine as build
RUN addgroup -S user && adduser -S user -G user
USER user

WORKDIR /workspace/app

COPY mvnw .
COPY .mvn .mvn
COPY pom.xml .
COPY src src

RUN --mount=type=cache,target=/root/.m2 ./mvnw install -DskipTests
RUN mkdir -p target/dependency && (cd target/dependency; jar -xf ../*.jar)

FROM openjdk:8-jre-alpine
VOLUME /tmp
ARG DEPENDENCY=/workspace/app/target/dependency
COPY --from=build ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY --from=build ${DEPENDENCY}/META-INF /app/META-INF
COPY --from=build ${DEPENDENCY}/BOOT-INF/classes /app
EXPOSE 8081
ENTRYPOINT ["java", "-cp", "app:app/lib/*", "com.mammon.documentdbnodev2.Documentdbnodev2Application"]
```


The base image is `openjdk:8-jdk-alpine`. The alpine images are smaller than the standard OpenJDK library images from the Docker hub. What is more, “20 MB can be saved in the base image by using the JRE label instead of JDK and this can be done in the next base image. Not all applications work with a JRE (as opposed to a JDK), but most do. The advantage of working with a JRE is avoiding the risk of misuse of some of the JDK features.”

“A Spring Boot fat JAR naturally has “layers” because of the way that the JAR itself is packaged. If we unpack it first, it is already divided into external and internal dependencies. To do this in one step in the docker build, we need to unpack the JAR first.”

The following commands will do the job:


```
RUN mkdir -p target/dependency && (cd target/dependency; jar -xf ../*.jar)
ARG DEPENDENCY=/workspace/app/target/dependency
COPY --from=build ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY --from=build ${DEPENDENCY}/META-INF /app/META-INF
COPY --from=build ${DEPENDENCY}/BOOT-INF/classes /app
```

“After advancing, the multistage was added to the docker file it allows to build JAR file and copying the result from one image to another. The first image is labeled `build`, and it is used to run Maven, build the fat JAR, and unpack it. the source code has been split into four layers. The later layers contain the build configuration and the source code for the application, and the earlier layers contain the build system itself (the Maven wrapper). This is a small optimization, and it also means that we do not have to copy the target directory to a docker image, even a temporary one used for the build.”

“Every build where the source code changes are slow because the Maven cache must be re-created in the first `RUN` section. But we have a completely standalone build that anyone can run to get our application running as long as they have docker. That can be quite

useful in some environments — for example, where you need to share your code with people who do not know Java.”

The following commands do the job:

```
FROM openjdk:8-jdk-alpine as  build
WORKDIR /workspace/app
```

```
COPY mvnw .
COPY .mvn .mvn
COPY pom.xml .
COPY src src
```

The first line makes a “way to cache build dependencies. The RUN directive accepts a new flag: --mount for more details”
<https://github.com/moby/buildkit/blob/master/frontend/dockerfile/docs/syntax.md>

The following commands show do the job:

```
# syntax=docker/dockerfile:1.3
RUN --mount=type=cache,target=/root/.m2 ./mvnw install -DskipTests
```

“For security aspects, the processes should not be run with root permissions. Instead, the image should contain a non-root user that runs the application.”

“In a Docker file, this can be achieved by adding another layer that adds a (system) user and group and set it as the current user.”

The following commands do the job:

```
RUN addgroup -S user && adduser -S user -G user
USER user
```

“In case someone manages to break out of your application and run system commands inside the container, this precaution limits their capabilities”.

Docker in Docker

The master work docker client to create nodes, the first version was using process builder in java to build docker containers. But after moving to the next stage where it is needed to run docker in docker, the process builder fail at this stage.

The solution is to use docker client API for java

<https://github.com/docker-java/docker-java>

The problem from using this library to build containers not enough information on how to use the library. The most useful resources is only the GitHub repository for the library.

The power of docker in docker can be noticed in the following:

<input type="checkbox"/>	gcr.io/k8s-minikube/kicbase:v0.0.30@sha256:02c921df998f95e849058af685e5c5a58de (minikube)	exited		
<input type="checkbox"/>	k8s.gcr.io/pause:3.7 (k8s_POD_test-database-k8s-64d758c485-rcceb_default_bcb1877e-63a5-4e57-8f521f34c6feeb)	about 12 hour running		
<input type="checkbox"/>	mamoosh/document-db-docker@sha256:9c62da9c779a0554bb682f93d8a55b838e26 (k8s_document-db-docker_test-database-k8s-64d758c485-rcceb_default_bcb1877e-63a5-4e57-8f521f34c6feeb)	exited		

<input type="checkbox"/>	gcr.io/k8s-minikube/kicbase:v0.0.30@sha256:02c921df998f95e849058af685e5c5a58de (minikube)	exited		
<input type="checkbox"/>	k8s.gcr.io/pause:3.7 (k8s_POD_test-database-k8s-64d758c485-rccc8_default_bcb1877e-63a5-4e57-8f521f34c6feeb)	about 12 hour	running	
<input type="checkbox"/>	mamoosh/document-db-docker@sha256:9c62da9c779a0554bb682f938ec3eb34f2ed (k8s_document-db-docker_test-database-k8s-64d758c485-rccc8_default_bcb1877e-63a5-4e57-8f521f34c6feeb)	1 minute ago	running	
<input type="checkbox"/>	mamoosh/node-reader-document:2.0 (animal-shop-cat-1)	1 minute ago	running	
<input type="checkbox"/>	mamoosh/node-reader-document:2.0 (animal-shop-cat-2)	1 minute ago	running	
<input type="checkbox"/>	mamoosh/node-reader-document:2.0 (animal-shop-dog-1)	1 minute ago	running	
<input type="checkbox"/>	mamoosh/node-reader-document:2.0 (animal-shop-dog-2)	1 minute ago	running	

Kubernetes

Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications.

It is used to deploy the master, the result for this was cheerful because if the master stopped or fall, it will start again with master self-healing abilities.

The solution at the beginning was using Kind to create k8s cluster but as a result the docker in docker stopped form working because docker daemon isn't available. Then docker desktop for k8s is used to solve this problem.

The k8s guide with kind:

<https://spring.io/guides/topicals/spring-on-kubernetes/>

how to communicate between the master in the pod and the containers outside the pod? The answer for this started by docker exec ping to see if it is possible to communicate, and the surprise that ping worked!!

Testing how to communicate, the docker by default add the new containers to bridge network (default network). Taking their IP address to intercommunication between the master and the nodes was a successful idea. As result the communication start to be like this:

<http://172.17.0.5:8090/collection/3bbed9c106ceaea9e1d1f851b493a52582ae6f6deab8170da43da346a8710622b21d6c2ca14c6336bdc770f161673bef5edad6e65d86b05be62817bc9088d924>

This link used in the rest template to update a node with address 172.17.0.5:8090.

The k8s deployment

```

apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: test-database-k8s
  name: test-database-k8s
spec:
  replicas: 1
  selector:
    matchLabels:
      app: test-database-k8s
  strategy: {}
  template:
    metadata:
      labels:
        app: test-database-k8s
    spec:
      containers:
        - image: mamoonsh/document-db-docker:latest
          name: document-db-docker
          resources: {}
          volumeMounts:
            - name: dockersock
              mountPath: "/var/run/docker.sock"
            - name: database
              mountPath: "./database"
      volumes:
        - name: dockersock
          hostPath:
            path: /var/run/docker.sock
        - name: database
          hostPath:
            path: documentdb1

```

The k8s service

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: test-database-k8s
  name: test-database-k8s
spec:
  ports:
  - name: 80-8080
    port: 80
    protocol: TCP
    targetPort: 8080
  selector:
    app: test-database-k8s
  type: ClusterIP
status:
  loadBalancer: {}
```

The command which needed to k8s deployment:

```
C:\Users\Dell>kubect1 apply -f ./k8s
deployment.apps/test-database-k8s created
service/test-database-k8s configured
```

```
C:\Users\Dell>kubect1 port-forward svc/test-database-k8s 8080:80
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
```