UNIVERSITÀ DI PISA

Large Scale and Multi-Structured Databases

Year 2021/22

WeFit - Application Report

Application developed by

Luana Bussu, Luca Caprioli, Veronica Gavagna

Summary

Introduction

WeFit is an application designed for connecting different people from different geographic areas with the same hobby: training. It is based on a community of athletes that can interact among them and share their workout routines.

The aim of the application is to give to subscribers of the community a mean to analyse current and past workout-routines given by their trainers, follow other users, browse their routines, and optionally comment and vote them.

From an administrator point of view, it gives the trainers the possibility to add new exercises, create new routines and run statistics on the users or on their satisfaction regarding exercises and trainers.

The application was developed in Java and data was stored in two different databases: MongoDB and Neo4j.

GitHub project repository

Design

Functional and Non-Functional requirements

The following section summarize the requirements and core functionalities that the application must provide.

Main Actors

The application interacts only with two types of entities: Users and Trainers (administrators):

- The User can use the basic features of the application
- The Trainer has the same privileges of the User in the baseline scope of the application and can also make changes and run statistics on the datasets.

Functional Requirements

An **unregistered User** can only sign up to use the application

A **User** can:

- Login the application
- Logout the application
- Change profile information
- Find a user by parameters
- Find followed users
- Find followers
- Follow/Unfollow a user
- Find workout routines by parameters
- Browse followed users' routines
- View details of workout routines exercises
- Comment a routine
- Retrieve routines' comments
- Retrieve routines you commented
- Vote a routine
- Find most rated trainers
- Find most followed users
- Find most commented routines
- Show owns recommended users list

A **Trainer** can also:

- Add a new exercise
- Add a new routine
- Promote a user (make him a trainer)
- Find most used equipment (in general and for each muscle group)
- Find average age per level
- Find most common exercises in the most rated routines
- Find most fidelity users (with the high number of past routines and oldest subscribed)
- Show the number of users that levelled up within a set period

Non-Functional Requirements

1. The password should be encrypted
2. The application should be user-friendly
3. The application should guarantee data availability**,** for example to see routines and changes every time
4. The application should guarantee a low latency
5. The application should guarantee partition tolerance, avoiding a single point of failure

Solutions to non-Functional Requirements

1. The password are saved on the database as an encrypted string with the SHA-256 hash algorithm (java library “org.apache.commons.codec.digest.DigestUtils”).
2. Despite not having a GUI the application has been developed to be user-friendly.
   The menus are organized as a list of commands to input easy to navigate, with the possibility to both return to the previous step or to the main one and every time a new command has to be inserted, the option list is reprinted.
   Both summary and detailed results are gradually printed as tables to improve readability.

3. 4. 5. Regarding the data availability and the partition tolerance we decide to implement a distributed database using three replicas (implemented in Mongo Atlas) following the AP solution of the CAP Theorem (details about replicas can be found in the Replicas chapter).

Use case

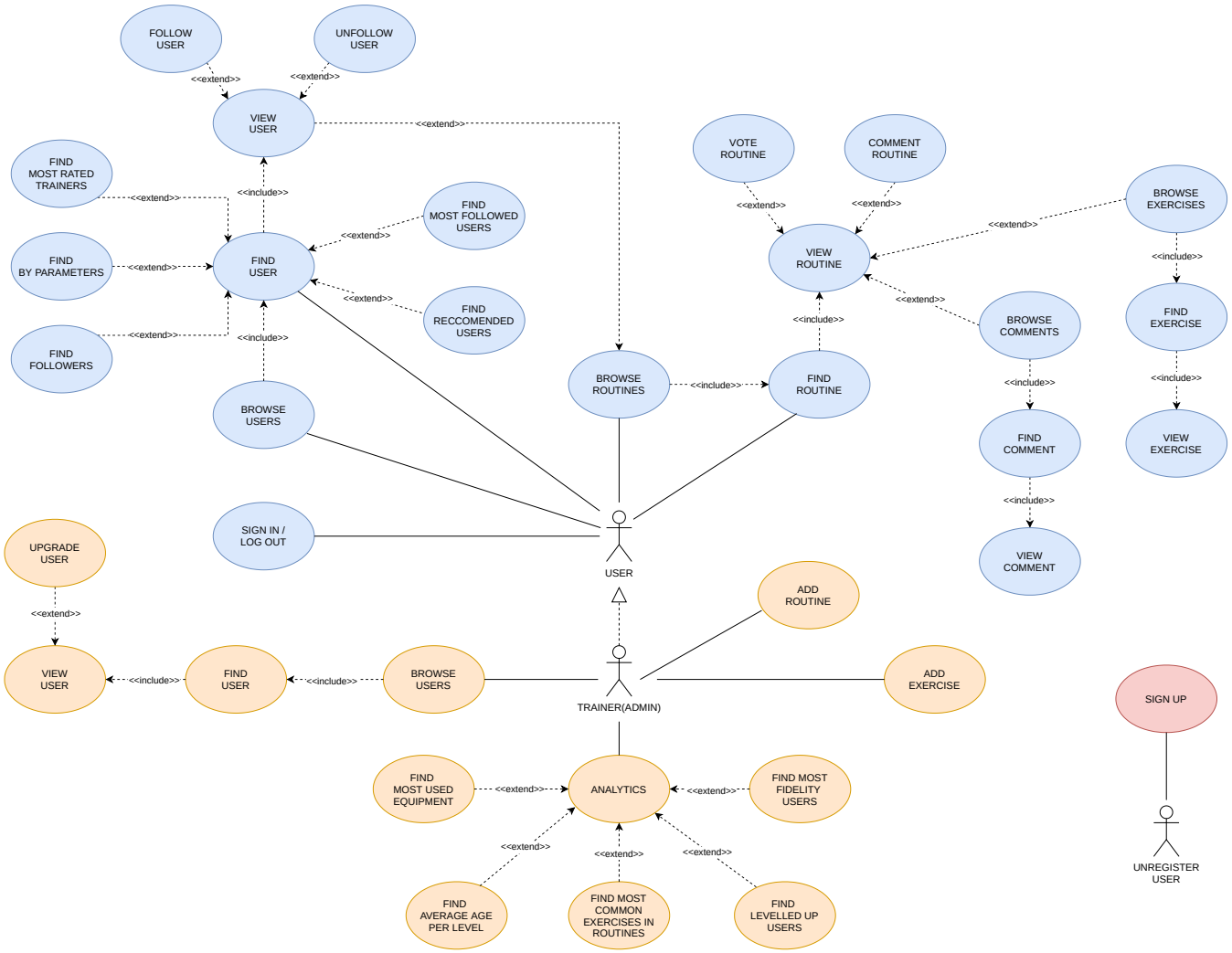

Figure 1: Use case diagram

Class Analysis

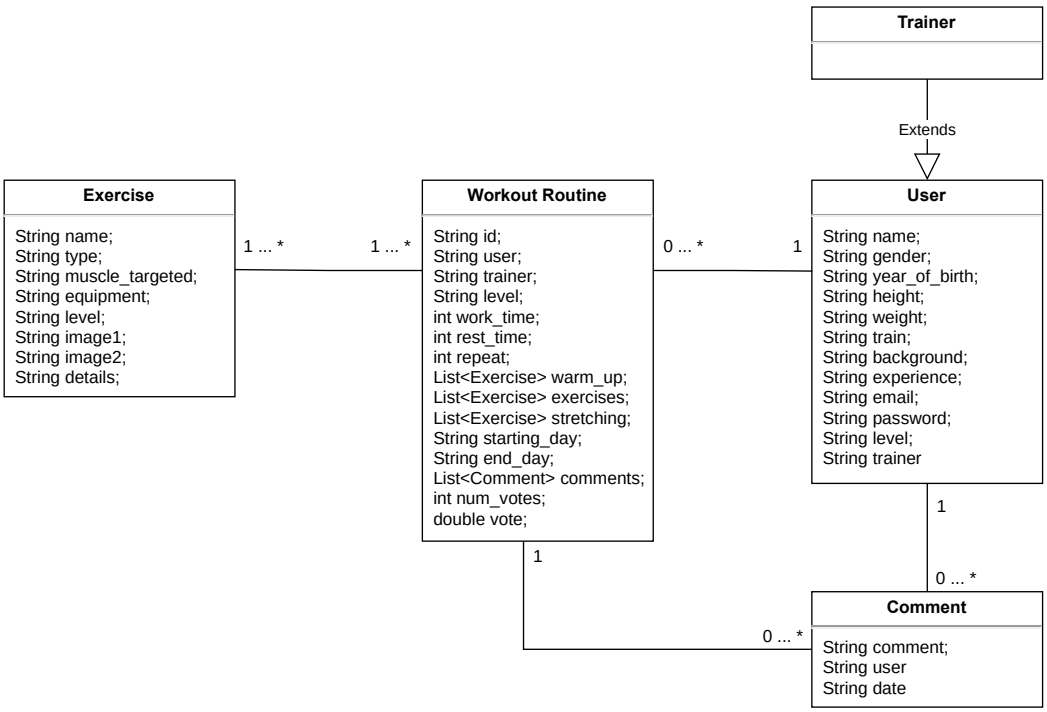

Figure 4: Class diagram

As shown in the picture:

- A user can have 0 or more workout routines, but a workout routine belongs only to one user
- A user can comment 0 or more workout routines and a comment belong to one user and it is related to one workout routine
- A workout routine can have many exercises and an exercise can be in 0 or more routines

Dataset

The dataset of the application comes from different sources like scraping operations or open-source resources.
We scraped the exercises on bodybuilding.com and the comments from YouTube.
We download a dataset of athletes on data.world containing personal and physical information of users and a short description of the trainer background and past experiences; we integrate this dataset with login credentials randomly generated.
All these data were stored in two different databases: Neo4j maintains the social part including main information of users, routines, and the relationship among them, while MongoDB maintains more details information about them but also the comments and the exercises.
Starting from these data we made a script Java to create past routines making use of the exercise's dataset.

```
private void createRoutine(String user, String level, long start) {
    String s="";
    Document routine = new Document("user", user);

    Bson match = match(eq( fieldName: "trainer", value: "yes"));
    Bson sample = sample( size: 1);
    Document t = users.aggregate(Arrays.asList(match, sample)).first();
    if(t!=null) s = t.getString("user_id");
    routine.append("trainer", s).append("level", level)
            .append("work_time(sec)",30).append("rest_time(sec)", 10).append("repeat", 3);

    ArrayList<Document> warm_up = new ArrayList<>();
    switch (level){...} //choose exercise of the same level or lower
    sample = sample( size: 3);
    AggregateIterable<Document> output = workout.aggregate(Arrays.asList(match,sample));
    for(Document d: output)
        createEx(d, warm_up, x: false);
    routine.append("warm_up", warm_up);

    ArrayList<Document> exs = new ArrayList<>();
    for(String m: muscles){
        switch (level){...}  //choose exercise of the same level or lower
        sample = sample( size: 1);
        output = workout.aggregate(Arrays.asList(match,sample));
        for(Document d:output)
            createEx(d, exs, x: true);
    }
    routine.append("exercises",exs);

    ArrayList<Document> stretch = new ArrayList<>();
    switch (level){...} //choose exercise of the same level or lower
    sample = sample( size: 3);
    output = workout.aggregate(Arrays.asList(match,sample));
    for(Document d:output)
        createEx(d, stretch, x: true);
    routine.append("stretching", stretch);
    routine.append("starting_day", LocalDate.ofEpochDay(start).toString());
    routine.append("end_day", LocalDate.ofEpochDay(start).plusDays(30).toString());

    workout.insertOne(routine);
}
```

Figure 7: function for routine creation

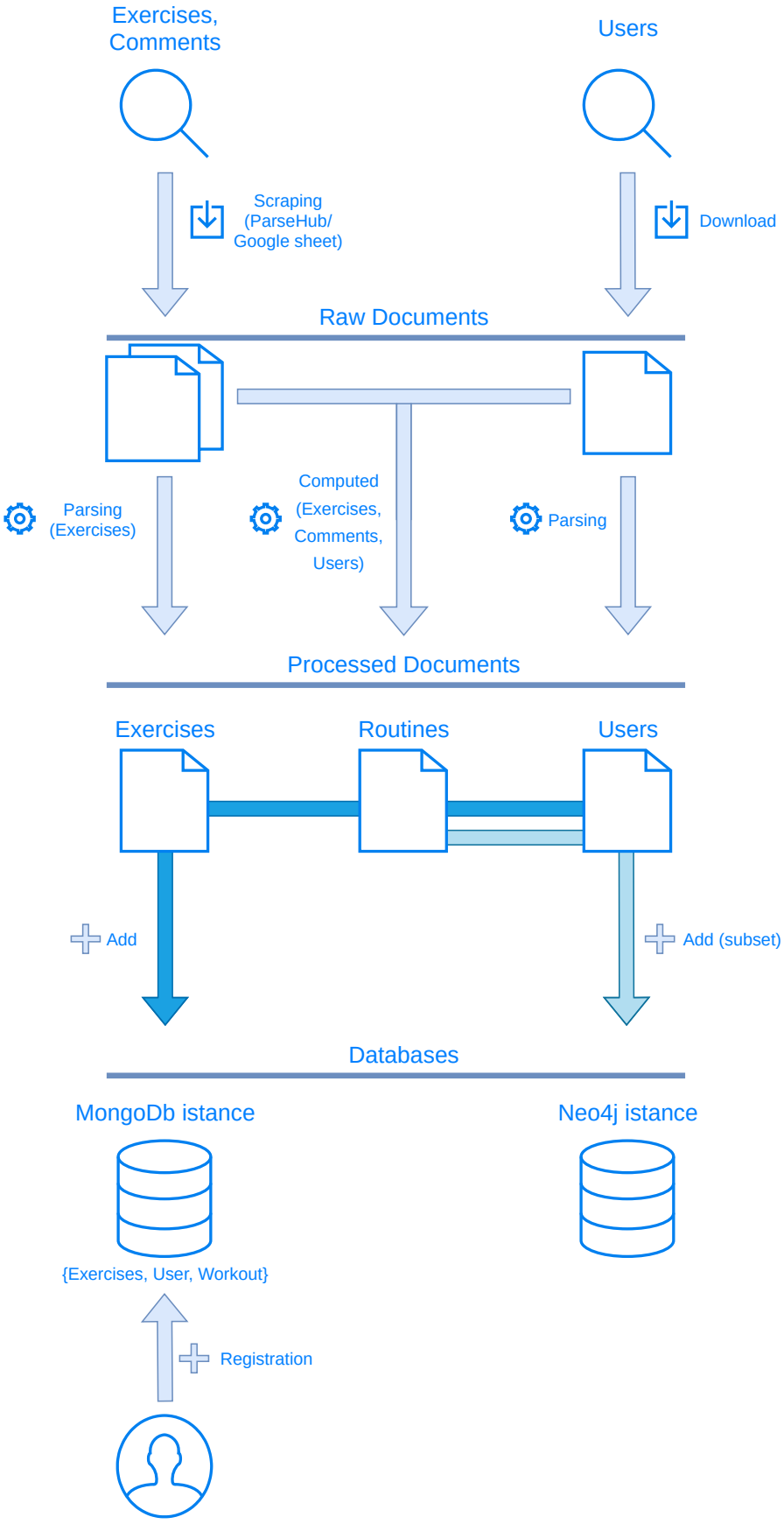

Figure 10: Database creation procedure

MongoDB – Design and Implementation

In MongoDB the data is stored in three different collections: users, exercises and workout.

Users: 50.000 users (9MB)

The Users collection maintains all the information of the users to separate them from the large number of routines and avoid scrolling them every time we need a user information (like the credentials for the login).


```
{
  "_id": {
    "$oid": "61c2fac28ed19ceda7cef426"
  },
  "name": "Judith Eyres",
  "gender": "Female",
  "year_of_birth": "1991",
  "height": "142",
  "weight": "70",
  "train": "I workout mostly at a CrossFit Affiliate",
  "background": "I played youth or high school level sports",
  "experience": "I began CrossFit with a coach",
  "email": "user@gmail.com",
  "password": "user",
  "level": "Expert",
  "trainer": "no",
  "user_id": "241102"
}
```


Figure 13: Example of user's data stored in MongoDB

Exercises: 1.300 exercises (~ 500 kB)

The exercises collection maintains all the detailed information about exercises so they can be easily retrieved if a user wants to see them (in general this is not a common operation as described later).


```
{
  "_id": {
    "$oid": "61b3706a7e0cf012e258e0e0"
  },
  "name": "Jumping rope",
  "type": "Cardio",
  "muscle_targeted": "Quadriceps",
  "equipment": "Body Only",
  "level": "Intermediate",
  "images": [
    {
      "image": "https://www.bodybuilding.com/images/2020/xdb/originals/
xdb-7s-jumping-rope-m1-16x9.jpg"
    },
    {
      "image": "https://www.bodybuilding.com/images/2020/xdb/originals/
xdb-7s-jumping-rope-m2-16x9.jpg"
    }
  ],
  "details": "Hold an end of the rope in each hand.Position the rope
behind you on the ground. Raise your arms up and turn the rope over your
head bringing it down in front of you. When it reaches the ground, jump over it.
Find a good turning pace that can be maintained. Different speeds and techniques
can be used to introduce variation.Rope jumping is exciting, challenges your
coordination, and requires a lot of energy.
A 150 lb person will burn about 350 calories jumping rope for 30 minutes,
compared to over 450 calories running."
}
```


Figure 6: Example of exercise's data stored in

Workout: ~ 120.000 routines (140MB)

The Workout collection stores workout routines and comments that are store as embedded documents on the routines. We decide to store these data together because the comments are strictly related to one routine, and they are never used alone.

Regarding the exercises we store the whole information in the exercises collection, but we also stored summary fields as embedded documents on the routines: with this redundancy we give to the user the opportunity to directly retrieve from the routine the main information of its exercises making the research of the detail's information of an exercise a rare operation.


```
{
  "_id": {
    "$oid": "61d6eaeccfad1d76f01507ea"
  },
  "user": "200779",
  "trainer": "513270",
  "level": "Expert",
  "work_time(sec)": 30,
  "rest_time(sec)": 10,
  "repeat": 3,
  "warm_up": [
    {
      "name": "Trail Running/Walking",
      "muscle_targeted": "Quadriceps",
      "equipment": "Body Only",
      "type": "Cardio"
    },
        ....
  ],
  "exercises": [
    {
      "name": "Seated cable shoulder press",
      "muscle_targeted": "Shoulders",
      "equipment": "Cable",
      "type": "Strength",
      "weight": 17
    },
        ...
  ],
  "stretching": [
    {
      "name": "Calf SMR",
      "muscle_targeted": "Calves",
      "equipment": "Foam Roll",
      "type": "Stretching"
    },
        ...
  ],
  "starting_day": "2018-10-11",
  "end_day": "2018-11-10",
  "comments": [
    {
      "_id": {
        "$oid": "61d1df64d7898e7440962929"
      },
      "Comment": "Real people Real workout ~~~thanks!! Love it!!",
      "Time": "2019-03-25T01:18:41Z",
      "user": "142955",
      "routine": "61d6eaeccfad1d76f01507ea"
    },
        ...
  ],
  "num_votes": 25,
  "vote": 3.56
}
```


Figure 16: Example of workout routine data stored in MongoDB

Neo4j – Design and Implementation

In Neo4j data is stored in two types of nodes (*~ 170 K*) and five type of relationship (*~ 1.6 M)*

Nodes:

- The **User** node, represent a registered user and store the main information of the user: (user_id, name, level, gender, birth)
- The **Routine** nodes, represent a routine and store the useful information for the graph queries (_id, starting day, end day, user, trainer, level, and average vote)
  We introduce the redundancy vote filed in the node to be able to easily retrieve the vote of the routine every time it is shown (and avoid making a query on the votes relationships every time)

Relationship:

- **User** –[**: HAS_ROUTINE**]→ **Routine** which represent a past or current routine of the user. This relationship has no attributes.
- **User** –[: **CREATE_ROUTINE**]→ **Routine** which represent a routine created by the user (the user is a trainer). This relationship has no attributes.
- **User** –[: **FOLLOW**]→ **User**, which represents a user following another in the application. This relationship has no attributes.
- **User** –[: **COMMENT**]→ **Routine**, which represents a comment by a user on the routine. This relationship has no attributes.
- **User** –[: **VOTE**]→ **Routine**, which represents a vote by a user on the routine. This relationship has the vote attribute (number between 1 and 5).

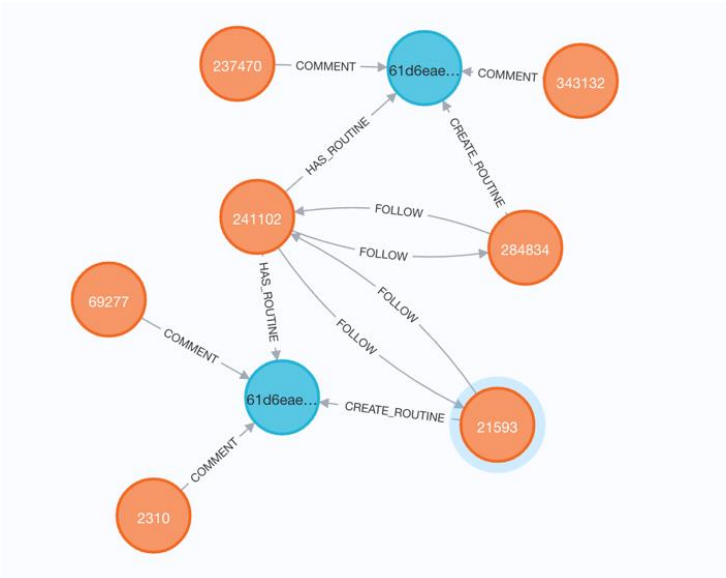

Figure 19: Example of nodes and relationship in

Cross-Database consistency management

Based on how the datasets and the application queries were structured, it is necessary to manage possible state of inconsistency between duplicated information on MongoDB and Neo4j. In addition to attempting automatic consistency recovery mechanisms, errors are presented at runtime on the application interface, giving the user, be it normal or trainer, to retry the operation.
The operations requiring cross-database consistency are the following:

signUp

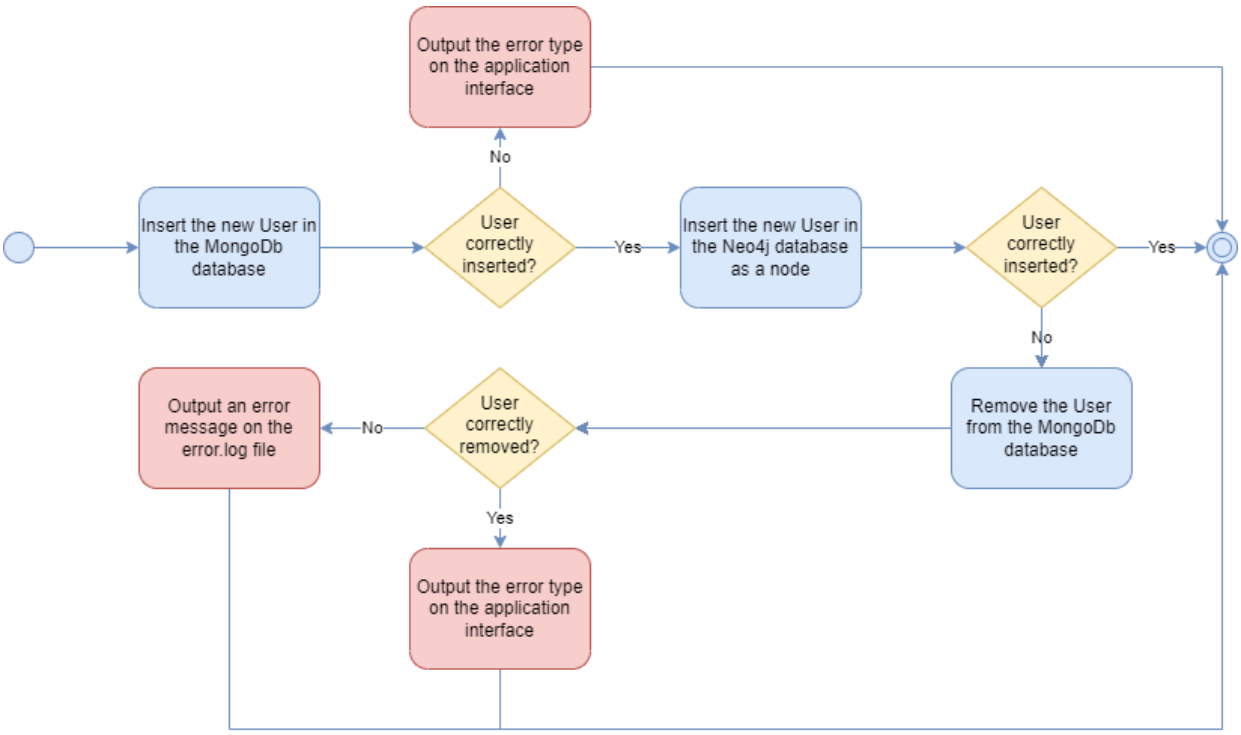

Figure 22: signUp DB consistency flowchart

changeProfile

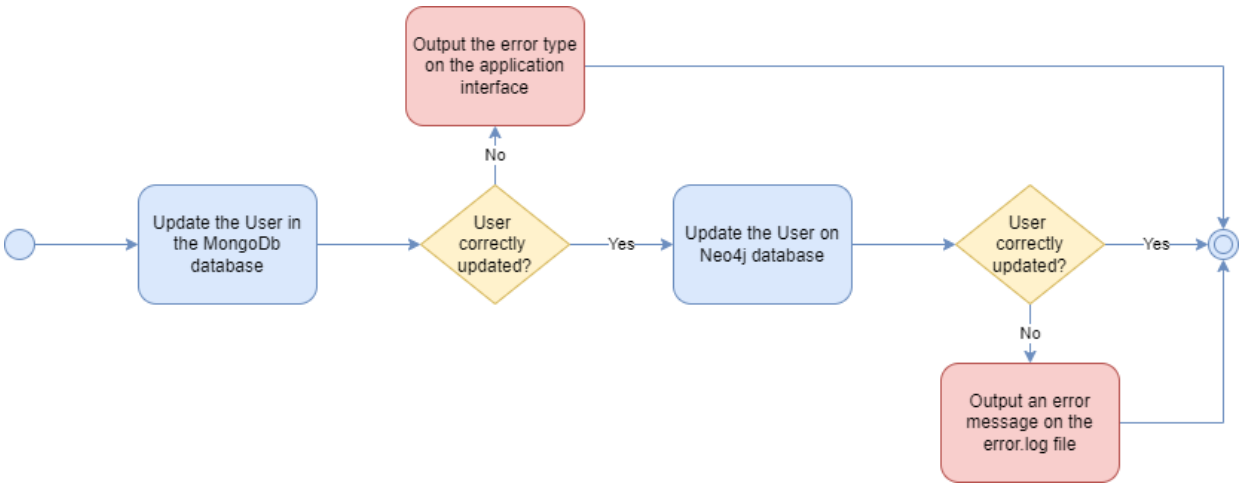

Figure 25: changeProfile consistency flowchart

insertRoutine

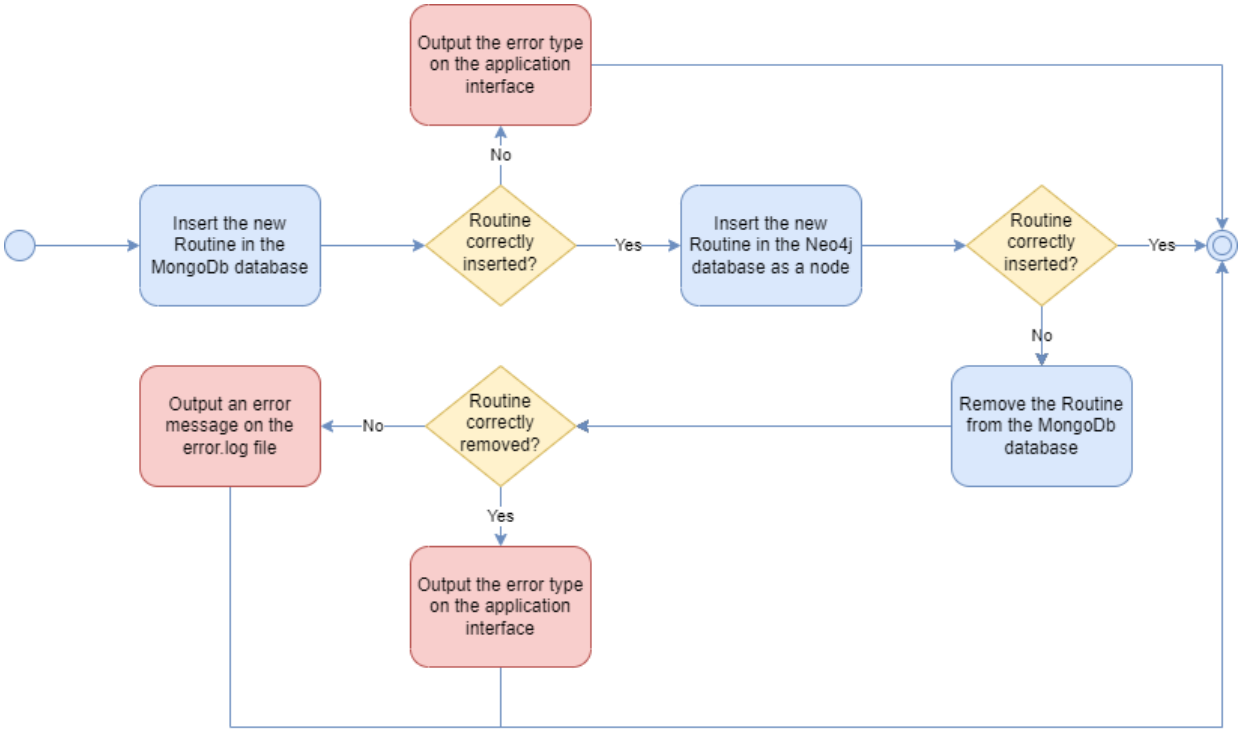

Figure 28: insertRoutine DB consistency flowchart

insertComment

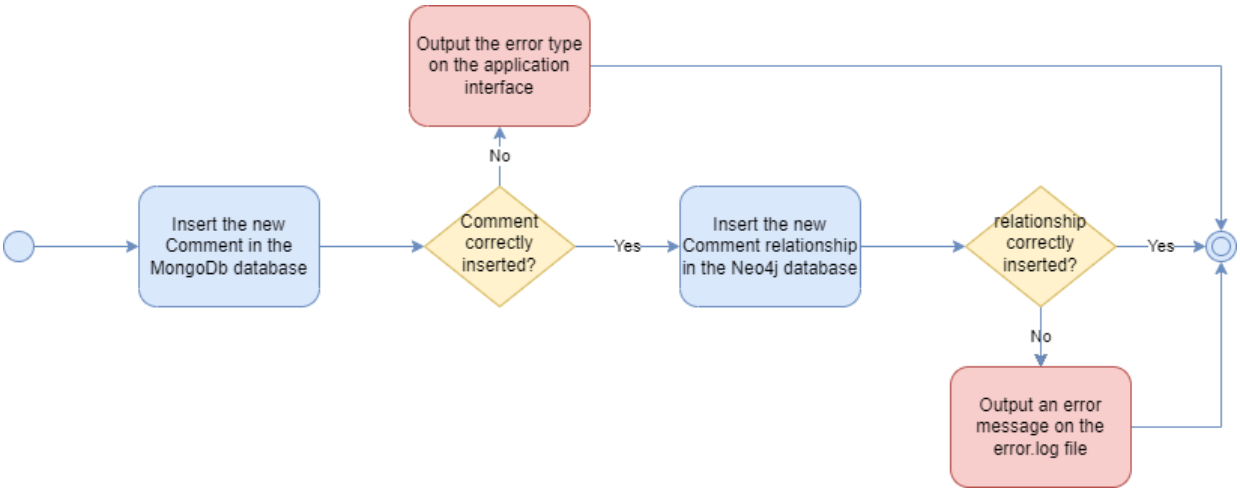

Figure 31: insertComment DB consistency flowchart

deleteComment

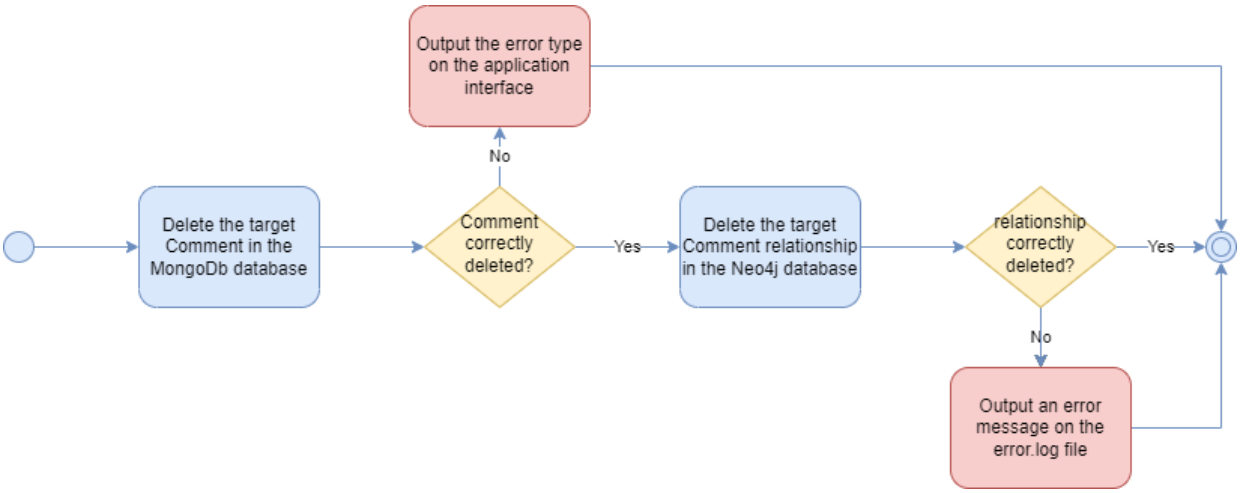

Figure 34: deleteComment DB consistency flowchart

insertVote

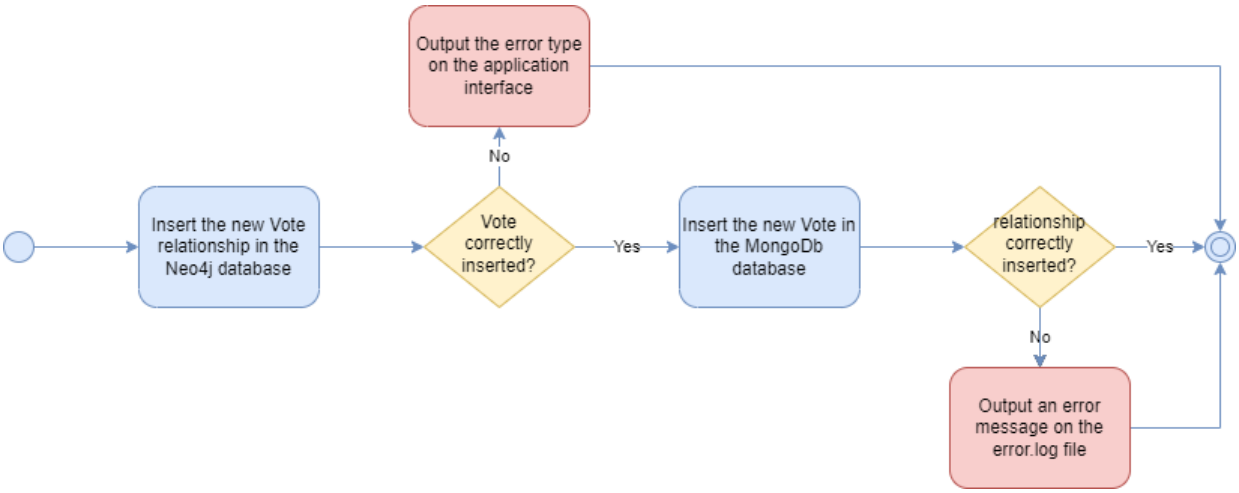

Figure 37: insertVote DB consistency flowchart

Implementation

Package structure

The Package structure decided for this project is the follow:

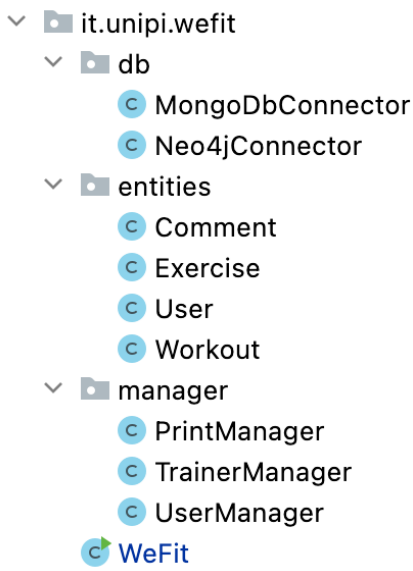

Figure 40: Package structure

We have not implemented a GUI.
We decided to create three different packages to ensure the readability and maintainability of the code. In this chapter will be explained what is inside of each package.

Entities (Java bean)

The package *"entities"* contains all the classes related to the actors, used in the code and in both databases:

- ***Comment:*** used for instantiating object that refers to a specific comment
- ***Exercise:*** used for instantiating object that refers to a specific exercise
- ***Workout:*** used for instantiating object that refers to a specific workout routine
- ***User:*** used for instantiating object that refers to a specific user

Db

The package "*db*" contains two classes:

- **MongoDbConnector:** contains all the functions to make queries and analytics on mongodb, for example functions to sign in the app, to retrieve a user or a routine, to search exercises/routines/users using the given filters.

- **Neo4jConnector:** contains all the functions to make queries or analytics in neo4j, for example show all the routines of a given trainer, to retrieve the most followed users or the most rated trainers.

These classes contain all the methods related to the communication with database, respectively MongoDB and Neo4j.

Manager

The package *"manager"* contains three classes created to maintain utility functions. The scope of this package is to keep the main more readable and to divide classes responsibility.

- **UserManager**: contains all the function which represent the functionality that a normal user has, and it also has the task to interfacing with the user.
- **TrainerManager**: is an extension of the UserManager class and contains all the additional functionality of a trainer (for example to add a new routine)
- **PrintManager**: contains all the print functions.

```
What do you need?
1)  See your current routine
2)  See your past routines
3)  See your followed list
4)  See your followers list
5)  See routines you commented
6)  Find a routine by parameter
7)  Find a user by parameter
----------ANALYTICS----------
8)  Find n-most rated personal trainer
9)  Find n-most followed users
10) Find n-most commented routines
----------------------------
11) Modify your profile
12) Log out
0)  Exit
```

Figure 46: User's menu

```
What do you need?
1)  See your routines
2)  Add a new routine
3)  Add a new exercise
4)  Add a new trainer
5)  See normal user menu
----------ANALYTICS----------
6)  See average age per level
7)  Find most fidelity users
8)  Find most used equipments per muscle
9)  Find most common exercises in most rated routines
10) Show level up
----------------------------
11) Log out
0)  Exit the app
```

Figure 43: Trainer's menu

Project Object Model (POM) File

The entire development cycle of the Java application relied on the support of the Maven project management tool, where the Project Object Model (POM) file that was used is shown below:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>WeFit</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.mongodb</groupId>
            <artifactId>mongodb-driver-sync</artifactId>
            <version>4.1.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-slf4j-impl</artifactId>
            <version>2.14.1</version>
        </dependency>
        <dependency>
            <groupId>org.neo4j.driver</groupId>
            <artifactId>neo4j-java-driver</artifactId>
            <version>4.1.1</version>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.22</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <build>
        <sourceDirectory>src</sourceDirectory>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>16</source>
                    <target>16</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Figure 49: POM file

MongoDB Analysis

Queries Analysis

Read Operations		
Operation	**Expected Frequency**	**Cost**
Get a user by the name	High	Low (1 read)
Retrieve the most Used Equipment	Low	High (Complex Aggregations)
Retrieve the most common exercises in most rated routines	Average	High (Complex Aggregations)
Search exercise(s) by name	Average	Average (multiple reads)
Search routine(s) using the given filters	High	Average (multiple reads)
Search user(s) using the given filters	Average	Average (multiple reads)
Sign in in the app (search credentials in the database)	High	Low (1 read)
Retrieve the average age level	Low	High (Complex Aggregations)
Show comments	Average	Average (multiple reads)

Write Operations		
Operation	**Expected Frequency**	**Cost**
Change profile's properties in the database	Low	Low (modify a document)
Insert a comment into a routine	High	Low (modify a document)
Insert a routine in the database	High	Low (insert a document)
Insert a vote in the database	High	Low (modify a document)
Sign up in the app (insert new user in the database)	Average	Low (insert a document)

MongoDB Analytics

Find the average age per level and the most frequent level

The analytics is based on the users, it need as input the threshold age and shows as output the average age per level and the most frequent level of users younger and older the given age.

```
                    YOUNGERS                          OLDERS
-------------------------------------------------------------------------
            LEVEL      COUNT              LEVEL      COUNT
           Expert      26143            Beginner      2862


            LEVEL          AVERAGE AGE
-------------------------------------------------------------------------
           Expert                   27
         Beginner                   36
     Intermediate                   32
```

Figure 52: Printed result of the analytic

In the following are shown the two queries in the mongo shell and with Java, for the second query is shown the level with most users younger than the given age:

```
wefit> db.users.aggregate(
            {$match: {"trainer":"no"}},
            {$group: {"_id":"$level", "AVG":{$avg: {$toInt: "$year_of_birth"}}}},
            )
```

Figure 55: MongoDB-shell implementation (average per level)

```
Bson match = match(eq( fieldName: "trainer", value: "no"));
Bson group = group( id: "$level", Accumulators.avg( fieldName: "Avg",
        eq( fieldName: "$toInt", value: "$year_of_birth")));
users.aggregate(Arrays.asList(match, group)).forEach(document -> {  /* .. */});
```

Figure 21: Java implementation (average per level)

```
wefit> db.users.aggregate(
            {$match: {"trainer":"no", "year_of:birth":{$gt:"1982"}}},
            {$group: {"_id":"$level", "count":{$sum: 1}}},
            {$sort: {"count": -1}}, {$limit: 1}
            )
```

Figure 22: MongoDB-shell implementation (most frequent level)

```
match = match(and(eq( fieldName: "trainer", value: "no"),gte( fieldName: "year_of_birth", threshold)));
group = group( id: "$level", sum( fieldName: "count", expression: 1));
Bson sort = sort(descending( ...fieldNames: "count"));
Bson limit = limit(1);
Document youngers = users.aggregate(Arrays.asList(match, group, sort,limit)).first();
```

Figure 23: Java implementation (most present level)

Find the most used equipment in the workout routines

The analytics is based on the workout routines and count how many times an equipment is present in all the routines, and how many times per each muscle group and return the most commons. The analytics shows as output 18 rows and for each row shows three fields: the muscle considered (“ALL” when all of them are considered), the most common equipment used and how many times it occurs

MUSCLE	EQUIPMENT	COUNT
ALL	Dumbbell	304908
Shoulders	Dumbbell	35818
Traps	Machine	39706
	...	

Figure 24: Printed result of the analytic

In the following implementation with the mongo shell is shown the query for the shoulder muscle, in the general case that consider all the muscles the second $match is omitted.

```
wefit> db.workout.aggregate(
            {$unwind: "$exercises"},
            {$match: {"exercises.muscle_targeted":"Shoulders"}},
            {$group: {"_id":"$exercises.equipment", "count":{$sum: 1}}},
            {$match: {"_id": {$ne:"Body Only"}}},
            {$sort: {"count": -1}}, {$limit: 1}
            )
```

Figure 25: MongoDB-shell implementation

```
ArrayList<Bson> filters = new ArrayList<>();
Bson unwind = unwind( fieldName: "$exercises");
filters.add(unwind);
if(muscle != null){
    Bson match_muscle = match(eq( fieldName: "exercises.muscle_targeted",muscle));
    filters.add(match_muscle);
} else { muscle = "ALL"; }

Bson group = group( id: "$exercises.equipment", sum( fieldName: "count",  expression: 1));
Bson match = match(ne( fieldName: "_id", value: "Body Only"));
Bson sort = sort(descending( ...fieldNames: "count"));
Bson limit = limit(1);
filters.add(group); filters.add(match);
filters.add(sort);  filters.add(limit);
```

Figure 26: Java implementation

Find the most common exercises in the most rated routines

The analytics is based on the workout routines and count how many times an exercise is present in the most rated routines and return the most commons. The analytics need as input the number of most rated routines to consider (*n)* and the number of most common exercises to show (*m*); it shows as output *m* rows and for each row shows two fields: the name of the exercises and how many times it is present.

In the following is shown the output with n=20, m=3

EXERCISE	COUNT
Band Hip Adductions	8
Isometric Neck Exercise - Sides	7
Fire Hydrant	6
Upright Row - With Bands	6

Figure 27: Printed result of the analytic

In the following images are shown implementation with the mongo shell and with Java:

```
wefit> db.workout.aggregate(
            {$sort: {"vote":-1}},
            {$limit: 20},
            {$unwind: "$exercises"},
            {$group: {"_id":"$exercises.name", "count":{$sum: 1}}},
            {$sort: {"count": -1}}, {$limit: 3}
            )
```

Figura 28: MongoDB-shell implementation

```
Bson order_vote = sort(descending( ...fieldNames: "vote"));
Bson limit_vote = limit(max_vote);
Bson unwind = unwind( fieldName: "$exercises");
Bson group = group( id: "$exercises.name", sum( fieldName: "count", expression: 1));
Bson order_ex = sort(descending( ...fieldNames: "count"));
Bson limit_ex = limit(max_ex);

List<Bson> most_pipeline = Arrays.asList(order_vote, limit_vote, unwind, group, order_ex, limit_ex);
ArrayList<Document> most_result = new ArrayList<>();
workout.aggregate(most_pipeline).into(most_result);
```

Figure 29: Java implementation

MongoDB Index Analysis

We decided to use indexes to speed up the application. We performed some test to measure the speed improvement obtained by using indexes. This test is carried out using the *explain()* function offered by MongoDB. In the following table are shown indexes we decide to create.

Index Name	Index Type	Collection	Attribute(s)	Kept
name_index	Unique	Exercises	name	Yes
last_user_index	Single	User	last_user	Yes
name_index	Single	User	name	Yes
email_index	Unique	User	email	Yes
trainer_index	Single	Workout	trainer	Yes
vote_index	Single	Workout	vote	**No**

Exercises Collection Tests

1. Find an exercise by name

```
wefit> db.exercises.find( {"name": "Lateral Speed Step"} )
```

name_index	Document Returned	Index Keys Examined	Documents Examined	Execution Time (ms)
False	1	0	1331	7
True	1	1	1	1

The exercise_index has been added to faster retrieve an exercise and to assure that no exercises with the same name will be added.

User Collection Tests

1. Find the last user

```
wefit> db.users.find( {"last_user": {$ne:null}} )
```

last_user_index	Document Returned	Index Keys Examined	Documents Examined	Execution Time (ms)
False	1	0	50006	67
True	1	2	1	1

2. Find a user by name

```
wefit> db.users.find( {“name”: “Trisha Briggs”} )
```

name_index	Document Returned	Index Keys Examined	Documents Examined	Execution Time (ms)
False	1	0	50006	52
True	1	2	1	1

Even if in our application is possible to find a user by multiple filters, we have supposed that the search by name is the most common and, for this reason, the name_index has been added.

3. Find a user by email

```
wefit> db.users.find( {“email”: “Rachael_Fox1459325778@irrepsy.com”} )
```

email_index	Document Returned	Index Keys Examined	Documents Examined	Execution Time (ms)
False	1	0	50001	26
True	1	1	1	2

The email_index has been added to faster retrieve a user at the login and to ensure that no user with the same email will be added.

Workout Collection Tests

1. Find routine by trainer

```
wefit> db.workout.aggregate(
            {$sort: {"vote":-1}},
            {$limit: 3}
            )
```

trainer_index	Document Returned	Index Keys Examined	Documents Examined	Execution Time (ms)
False	63	0	121154	510
True	63	63	63	5

Even if in our application is possible to find a routine by multiple filters, we have supposed that the search by trainer is one of the most commons and, for this reason, the trainer_index has been added.

2. Find most rated routines

```
wefit> db.workout.aggregate(
            {$sort: {"vote":-1}},
            {$limit: 3}
            )
```

vote_index	Document Returned	Index Keys Examined	Documents Examined	Execution Time (ms)
False	3	0	121154	134
True	3	3	3	3

3. Find most common exercises in most rated routines

```
wefit> db.workout.aggregate(
            {$sort: {"vote":-1}},
            {$limit: 20},
            {$unwind: "$exercises"},
            {$group: {"_id":"$exercises.name", "count":{$sum: 1}}},
            {$sort: {"count": -1}}, {$limit: 3}
            )
```

vote_index	Document Returned	Index Keys Examined	Documents Examined	Execution Time (ms)
False	20	0	121154	136
True	20	20	20	4

Regarding the tests 2 and 3, we consider that even if the improvement in terms of execution times is very high, in our application the analytics are carried out only by the trainer and not with a high frequency, while multiples users can vote the same routine, and this would involve frequent writing to the vote. For this reason, we supposed that for the vote field write operations are more commons that read operations and we decide to remove this index.

Neo4j Analysis

Queries Analysis

Read Operations		
Operation	**Expected Frequency**	**Cost**
Show all the routines commented by a given user	Average	Average (multiple reads)
Show all the routines of a given trainer	High	Average (multiple reads)
Retrieve the most followed users	Low	High (Complex Aggregations)
Retrieve the most rated trainers	Average	High (Complex Aggregations)
Retrieve the list of followers/followed users	Average	Average (multiple reads)
Show recommended users to follow	Low	Average (multiple reads)
Show how many users gained a level in the interval specified	Low	High (Complex Aggregations)
Show the most fidelity users	Low	High (Complex Aggregations)
Show all the past routines of the given user	Low	Average (multiple reads)
Show the current routine of a given user	High	Low (1 read)

Write Operations		
Operation	**Expected Frequency**	**Cost**
Change profile's properties in the db	Low	Low (modify a node)
Follow a user	Average	Low (add a relationship)
Vote a routine	Average	Low (add a relationship)
Add a new user	Average	Low (insert a node)
Add a new routine	High	Low (insert a node)
Unfollow a user	Low	Low (delete a relationship)

Neo4j Analytics

Find the most n rated personal trainers

The analytics is based on the workout routines created by a trainer and shows the most rated. The analytics need as input the number of trainers to show (*n)* and shows for each row of the output four fields: the user_id of the trainer, the name, the gender, and the vote of the trainer find as average of the routines vote.

	User_Id	Name	Gender	Vote
1)	524052	Valerie Wright	Female	3.722
2)	477868	Doris Miller	Female	3.701
3)	159191	Denis Richards	Male	3.667
4)	422295	Chester Jenkins	Male	3.664

Figure 30: Printed result of the analytic

```
ArrayList<Record> trainers;
try ( Session session = graph_driver.session() ) {
    trainers = (ArrayList<Record>) session.readTransaction(tx -> {
        List<Record> records;
        records = tx.run( s: "MATCH (u:User)-[:CREATE_ROUTINE]->(r:Routine)"+
                        "WITH u, AVG(toInteger(r.vote)) AS avg_vote "+
                        "RETURN u AS user, avg_vote ORDER BY avg_vote DESC LIMIT $num",
                parameters( ...keysAndValues: "num", num)).list();
        return records;
    });
```

Figure 31: Java implementation

Find the most n followed users

The analytics is based on the users and shows the most followed. The analytics need as input the number of users to show (*n)* and shows for each row of the output four fields: the user_id, the name, the gender of the number of follower's vote.

	User_Id	Name	Gender	Vote
1)	524052	Valerie Wright	Female	3.722
2)	477868	Doris Miller	Female	3.701
3)	159191	Denis Richards	Male	3.667
4)	422295	Chester Jenkins	Male	3.664

Figure 32: Printed result of the analytic

```
ArrayList<Record> trainers;
try ( Session session = graph_driver.session() ) {
    trainers = (ArrayList<Record>) session.readTransaction(tx -> {
        List<Record> records;
        records = tx.run( s: "MATCH ()-[f:FOLLOW]->(u:User) "+
                        "WITH u, COUNT(f) AS num_followers "+
                        "RETURN u AS user, num_followers ORDER BY num_followers DESC LIMIT $num",
                parameters( ...keysAndValues: "num", num)).list();
        return records;
    });
```

Figure 33: Java implementation

Find the most n commented routines

The analytics is based on the routines and shows the most commented. The analytics need as input the number of routines to show (*n)* and shows for each row of the output six fields: the user, the trainer, the level, the starting and the end day and the number of comments.

	User	Trainer	Level	Starting day	End day	Comments
1)	303279	540364	Expert	2020-01-29	"2020-02-29"	12
2)	262914	5347	Expert	2019-07-02	"2019-08-02"	11
3)	265502	119191	Beginner	2020-07-12	"2019-05-19"	11
4)	289780	61131	Expert	2020-02-20	"2020-03-22"	10

Figure 34: Printed result of the analytic

```
ArrayList<Record> routines;
try ( Session session = graph_driver.session() ) {
    routines = (ArrayList<Record>) session.readTransaction(tx -> {
        List<Record> records;
        records = tx.run( s: "MATCH ()-[c:COMMENT]->(r:Routine) "+
                "WITH r, SUM(c.num) AS num_comments "+
                "RETURN r AS routine, num_comments ORDER BY num_comments DESC LIMIT $num",
                parameters( ...keysAndValues: "num", num)).list();
        return records;
    });
}
```

Figure 35: Java implementation

Find the most fidelity user

The analytics is based on the workout routines and count for each user how many routines he/she has, and the top *n* are shown; if two users have the same number of past routines the one with the oldest first routine is selected. The analytics need as input the value of *n* and show as output *n* rows and for each row shows 3 fields: the user_id of the user, the number of past routines and the starting day of the first routines.

In the following are shown the output (*n* = 4) and the java implementation.

	User	Past routines	First routine
1)	250328	8	2019-05-06
2)	238675	5	2018-07-23
3)	230235	5	2020-01-29
4)	448076	4	2018-01-01

Figure 36: Printed result of the analytic

```
String date = LocalDate.now().toString();
try (Session session = graph_driver.session()) {
    ArrayList<Record> users = (ArrayList<Record>) session.readTransaction(tx -> {
        List<Record> persons;
        persons = tx.run( s: "MATCH (a:User)-[:HAS_ROUTINE]->(r:Routine) " +
                        "WHERE r.end_day < $date " +
                        "RETURN a AS user, COUNT(r) AS past_routines, min(r.starting_day) AS first_routine " +
                        "ORDER BY past_routines DESC, first_routine LIMIT $num",
                parameters( ...keysAndValues: "date", date, "num", num)).list();
        return persons;
    });
```

Figure 37: Java implementation

Find users that levelled up

The analytics is based on the workout routines and count how many users levelled up in each period. The analytics need as input the two dates and shown as output the number of users that pass from “Beginner” to “Intermediate”, from “Intermediate” to “Expert” and from “Beginner” to “Expert”.

```
The number of users that leveled up from 2021-01-01 to 2021-12-31:

Beginner->Intermediate      Intermediate->Expert          Beginner->Expert
                  1496                      1664                       280
```

Figure 38: Printed result of the analytic

```
try (Session session = graph_driver.session()) {
    ArrayList<Record> recommended = (ArrayList<Record>) session.readTransaction(tx -> {
        List<Record> persons;
        persons = tx.run( s: "MATCH (s:Routine)<-[:HAS_ROUTINE]-(a:User)-[:HAS_ROUTINE]->(r:Routine) " +
                          "WHERE r.starting_day >= $start AND s.end_day <= $end AND " +
                          "r.level = \"Beginner\" AND s.level = \"Intermediate\" AND " +
                          "s.starting_day > r.starting_day " +
                          "RETURN count(DISTINCT a) AS user",
                parameters( ...keysAndValues: "start", start, "end", end)).list();
        return persons;
    });
```

Figure 39: Java implementation

Neo4j Index Analysis

We decided to use indexes to speed up the application. We performed some test to measure the speed improvement obtained by using indexes.

Index Name	Index Type	Label	Attribute(s)
user_index	Single	User	user_id
routine_index	Single	Routine	_id

1. Find a user by user_id

```
1  MATCH (u:User {user_id:"17033"})
2  RETURN u
```

user_index	ExecutionTime (ms)
False	110
True	8

2. Find a routine by _id

```
1  MATCH (r:Routine {_id:" 61d707d8cfad1d76f016195a"})
2  RETURN r
```

routine_index	ExecutionTime (ms)
False	120
True	6

Replicas

MongoDB Replica Set – Atlas

To maintain the service always available and avoid problems from “single point of failure”, a cluster on the cloud service of MongoDB Atlas has been set up.
The cluster is composed by a replica set, one primary (wefit2022-shard-00-01.kxfyu.mongodb.net) that takes the request coming from the client and two secondaries (wefit2022-shard-00-00.kxfyu.mongodb.net, wefit2022-shard-00-02.kxfyu.mongodb.net) that keep the copies of the primary’s data.
All the replicas of the cluster can communicate one another and have the same listening port.

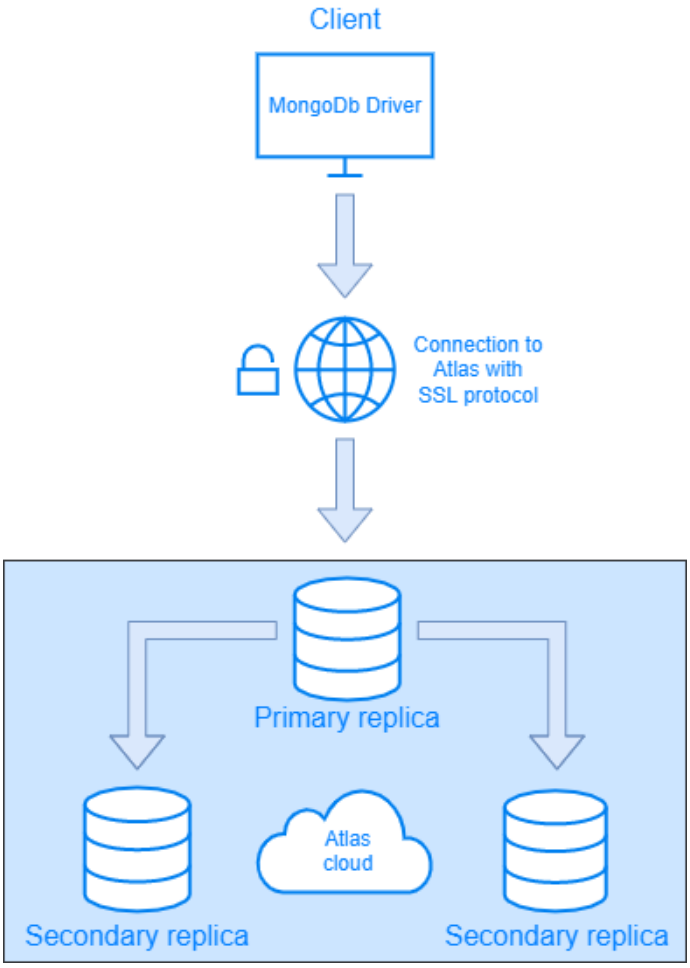

Figure 40: Replica set structure

The application is mainly based on read operation, often involving different documents and less frequently collections, while the write operations are surely present but less often than so. The application is thus a read-oriented one.

Replicas ensure Availability but given the use and the aim of the application, changes are often not to be seen immediately. By these means the cluster is initialised to update immediately the primary replica and give back the control to the client.
The secondary replicas will be eventually updated too, but not instantaneously so to diminish the latency of the operations themselves and thus implementing the Eventual Consistency.

Following the URL to connect to the cluster on Atlas is reported, full of the setting's details:

mongodb+srv://wefit:<password>@wefit2022.kxfyu.mongodb.net/wefit?authSource=admin&replicaSet=atlas-vt45gi-shard-0&w=1&readPreference=primaryPreferred&ssl=true

- **wefit:<password>**: Username and password for connecting to the replica set on Atlas.
- **Wefit2022.kxfyu.mongodb.net**: Address at which the replica set is located.
- **wefit**: database's name on the Atlas cloud.
- **authSource=admin**: authentication method to log-in on the db.
- **replicaSet=atlas-vt45gi-shard-0**: name of the replica set.
- **w=1**: Write Concern, wait only for the primary replica to be updated.
- **readPreference=primaryPreferred**: Read Preference, read from the primary replica and in case of failure read from the secondary replicas.
- **ssl=true**: Security protocol used for connecting to the Atlas cluster.

Sharding Proposal

In order to achieve a better load balancing and parallelization of the requests as well as a faster response on queries/analytics, it is possible to apply sharding in two different ways:

1. The first one utilizes the following sharding keys
   a. **Workout:** for this collection the **user** field was chosen as key, being the standard user operations the bulk of the application; dividing the load based on this attribute could bring faster accessibility to the database.
   b. **Users:** for this collection the **user_id** field was chosen because, as said for the previous point, user operations are the most present in the application.
   c. **Exercises:** for this collection, being it a limited set of documents, the sharding can be omitted, but an interesting way could be to divide it based on the **type/level** key, so that different part of the routine could be accessed individually.

   This kind of sharding is optimized to work under load given by operations coming from the user side, which represents the main load on the DBs, given their ever-increasing numbers.

   Finally, a **Hashed Strategy** is proposed (mainly for the first two collections) so to evenly balance as much as possible the load on the shards, given that the users ids are saved in the database as an auto-incrementing number.

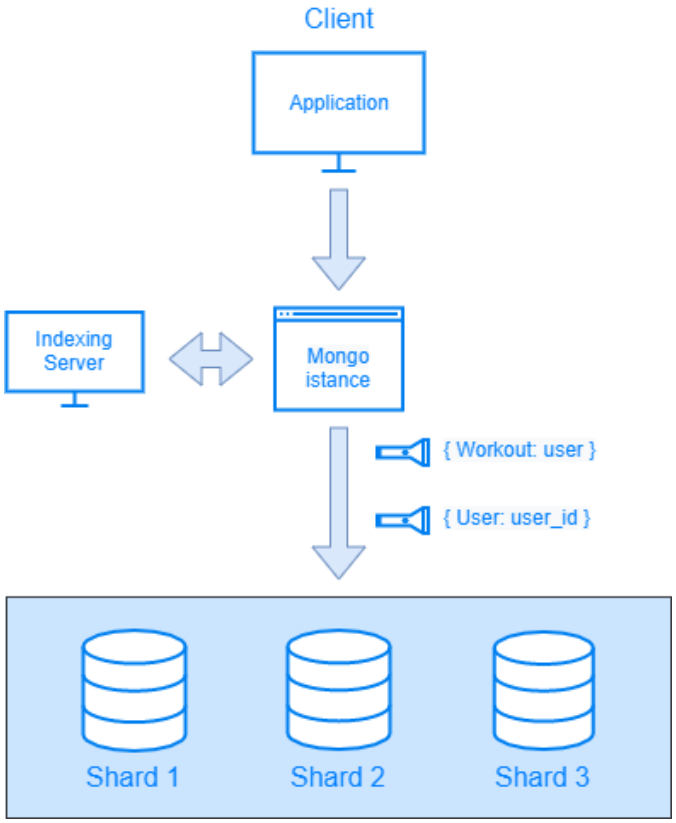

Figure 41: Sharding cluster structure

2. The second sharding proposal utilizes the following keys:
    a. **Workout:** for this collection the **vote** field was chosen as the sharding key, based on the analytics proposed which use this attribute as the basis for calculate the aggregations.
    b. **Users:** for this collection the **level** field was chosen as the sharding key because the analytics performed on this collection use mainly this attribute.
    c. **Exercises:** for the sake of analytics is useless to apply sharding to this collection, but as before one could shard it based on the **type/level** attribute to balance the load access on the exercise's details.

As already said this kind of sharding is more oriented to the analytic part of the application that, even being the less loaded one, is the most computational heavy.
The only warnings in this kind of sharding is that the keys are not fixed fields so a new server, namely **Shard update Server**, must be added to maintain the consistency of the cluster itself by single write updates, as the key attributes do not change so often.
In this case we adopt, for the Workout collection the **Range Sharding** based on the level key ranges.
For the Users collection we simply apply sharding based on the three key values, utilizing a **List Sharding**.

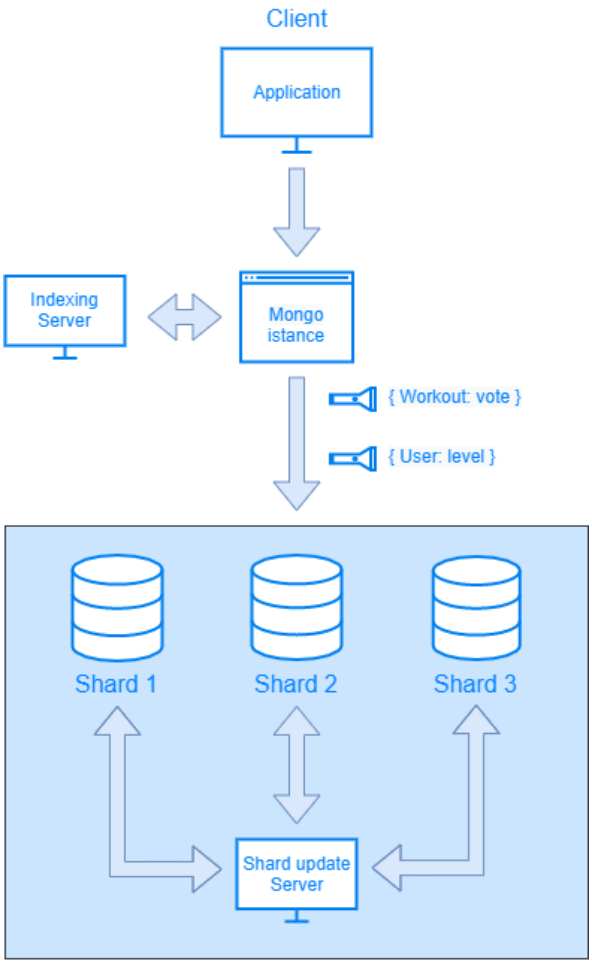

Figure 42: Sharding cluster structure (analytics)

The actors of the whole system are the following:

- **Mongo instance:** instance in charge of receiving read/write requests coming from clients and routing them to the shards.
- **Indexing server:** this server contains all the information regarding the state and organization for all the data within the cluster. It contains also information regarding the position of the chunks in the cluster and the ranges of them.
- **Shards:** starting from the replica set we divided the dataset in 3 different shards.
- **Shard update Server:** this server is linked to the 3 shards and oversees updating them in case of key-changing events if a sharding based on optimizing analytics is adopted.