LearnIt!

Large-scale and Multi-structured Databases

Federico Minniti

Francesca Pezzuti

Matteo Del Seppia

Academic Year 2021-2022

Contents

Introduction to the application

LearnIt! is a social-network where people can find courses to follow, read or write reviews about courses and follow other users.

The *LearnIt!* application is designed to handle a large amount of data that is typical of the social-network era, ensuring good performances, for these reasons we decided to use two different *NoSQL database* (GraphDB and DocumentDB).

The project is available on GitHub:
https://github.com/federicominniti/LearnIt

Dataset

To develop the *LearnIt!* application we had to find a large database of courses and reviews. We used two datasets[1] found on the internet for the courses and one dataset[2] for the reviews. The users, the social relationships between users and the relationships between users and courses have been generated ensuring the likelihood with the reality. Since the review's dataset was too small for our application, we had to generate more reviews from the existing ones.

2.1 Raw datasets' characteristics

- Tot courses: 49.13MB

- Tot reviews: 12.12MB

2.2 Data cleaning

During the data cleaning phase, fields had to be normalized to allow the application to handle them in a efficient way. For example, price and duration of courses were converted from strings into double values where possible. What's more, one of the two datasets had a stringified ObjectId in every document, so we had to translate the field into a true ObjectId.

2.3 Data integration

Since the datasets we have chosen came from two different sources (*Udemy* and *Coursera*), we had to integrate the data to ensure uniformity and to avoid duplicates or inconsistencies. In particular we had to perform attribute selection on both the databases in order to maintain the same attributes and to remove irrelevant features. We added some attributes that were present in just one database to the other database and we considered those attribute as optional attributes. We had to change some attributes name in order to have the same name for

---

[1] https://www.kaggle.com/trajput508/coursera-data, www.kaggle.com/rexxxxxxx/udemy-courses

[2] www.kaggle.com/koutetsu/udemy-courses-and-reviews

the same attribute in both the databases. We checked if some duplicates were present to ensure that no duplicate data was present.

Languages have been normalized between the Coursera and Udemy databases because there was a slight difference between the language identifiers (e.g. "French" vs "Francais"). Also the levels of the courses have been restricted to only four types (both the databases had a certain number of courses were the level was a non-standardized String).

2.4 Final datasets' characteristics

- Courses: 107.100

- Reviews: 541.000

- Users: 6.300

Design

3.1 Main actors

The main actors of the application are:

unregistered user
> User that accesses to the application for the first time. They have to sign up to become a registered user.

registered user
> User that is already registered to the application. They can use the application after the login.

administrator
> They are the most powerful actor of the application since they can delete users, courses and reviews in order to avoid offensive content and toxic behavior.

3.2 Requirements

3.2.1 Functional requirements

Features offered to the unregistered user:

sign in
> To have access to the community, a user must be registered.

Features offered to the registered user:

login/logout
> After the sign up, a user can login *LearnIt!* using their credentials to start using the application. After logging in, the logout from the application is always possible.

add a course
> The user can create and make available a new course.

search a course
>	The user can search courses by title, level or language.

find courses under a certain duration or price
>	The user can filter courses setting a threshold parameter on the duration and/or price.

browse suggested courses
>	The user receives personalized suggestions about courses

browse most liked courses
>	The user can discover which are the most liked courses.

browse the best courses based on rating
>	The user can discover which are the courses having the higher ratings.

browse the trending courses based on reviews
>	The user can browse the trending courses according to the reviews written in the last month.

delete their own courses
>	The user can remove a course they are teaching.

edit their own course
>	The user can edit the information of a course they are teaching.

like or unlike courses
>	The user can like/unlike a course.

review a course
>	When the user completes a course, they can write a review with a numeric rating and optionally leave a comment and a title.

edit their own review
>	The user can always edit a review they've written.

delete their own review
>	The user can always remove a review they've written.

browse courses followed, liked or completed by a specific user
>	The user can browse the profile page of a certain user and check the list of the liked, followed or completed courses by that specific user.

view stats of user
>	The user can browse the profile page of a certain user to check the statistics (number, average price and duration of completed courses) of the specific user.

follow/unfollow another user
>	The user can follow or unfollow other users.

edit their own profile
> The user can edit their personal information in their profile section.

browse suggested users
> The user receives suggestions about other users, according to the courses the user has completed.

browse most active users
> The user can browse the users that have written most reviews in the last month

browse best users in choosing courses
> The user can browse the users that proved to be the best when choosing the courses to follow, based on the courses price and duration

view follower/following users
> The user can see which users does he/her follow or the users that follow her/him and the total number of followers and following users.

Features offered to the administrator user:

create administrator
> An admin can create a new user as an administrator.

delete users
> An administrator can delete a user possibly disrespecting other users.

delete courses
> An administrator can remove a possibly disrespectful course.

delete reviews
> An administrator can remove possibly disrespectful reviews.

3.2.2 Non-functional requirements

The non-functional requirements of *LearnIt!* are:

usability
> The application must be **user-friendly** and must ensure a **low response-time**.

availability
> The service must be always available for users in order to guarantee the best possible user experience, providing a response to any query.

low latency
> The response to the requests should be fast.

partition protection
> The service must be tolerant to partitioning caused by failures, at the cost of being only eventually consistent.

reliability

The system must be stable, must return reproducible results and handle exceptions if needed.

flexibility

The user that makes a course available may choose only the fields they find appropriate to describe their course. The data should be handled in a flexible way.

portability

The system can be ported to different operating systems with no visible difference in its behavior.

privacy

The user credentials must be handled securely

maintainability

The code should be maintainable and readable

3.3 Use case diagram

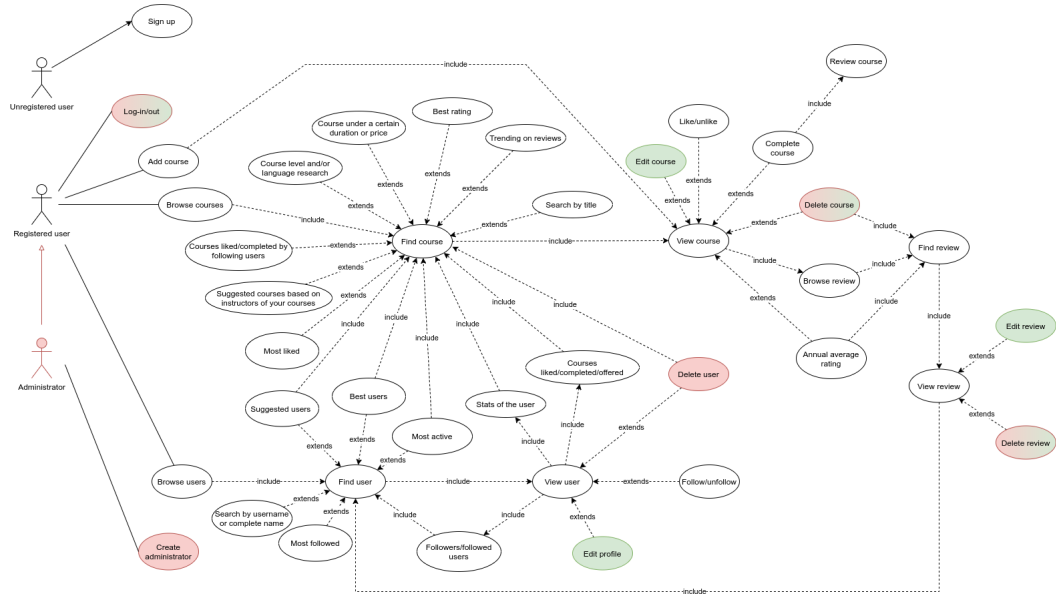The use case diagram of the application is described in Figure 3.1.

Figure 3.1: Use case diagram

3.3.1 Observations on use-case diagram

- *Red actions* can be performed only by an <u>Administrator</u>.

- *Green actions* can be performed only by the registered user that created the object (the <u>owner</u>).

- We assume that the user that leaves a review, has completed the course. This information is used to provide personalized suggestions about courses or users.

3.4 UML class diagram

The class diagram of the application is described in Figure 3.2.
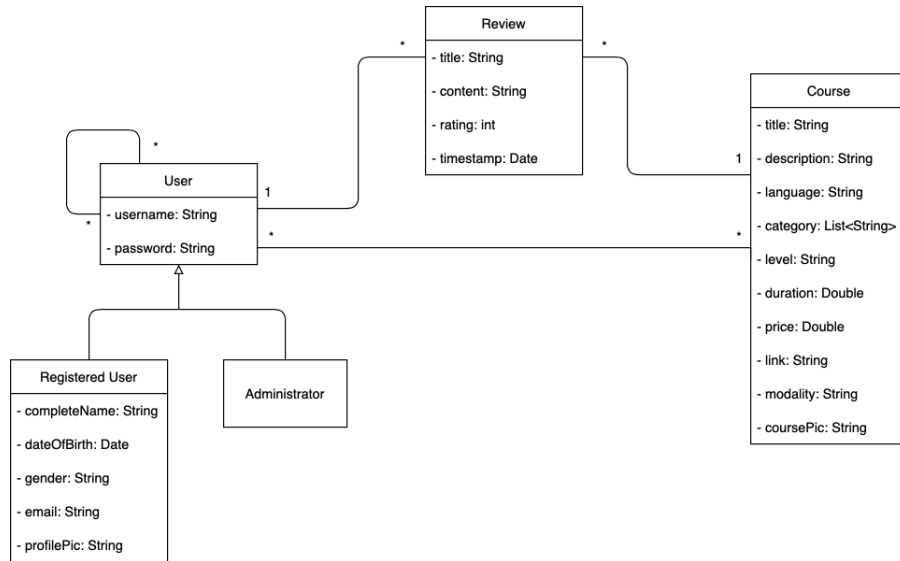

Figure 3.2: UML class diagram of *LearnIt!* application

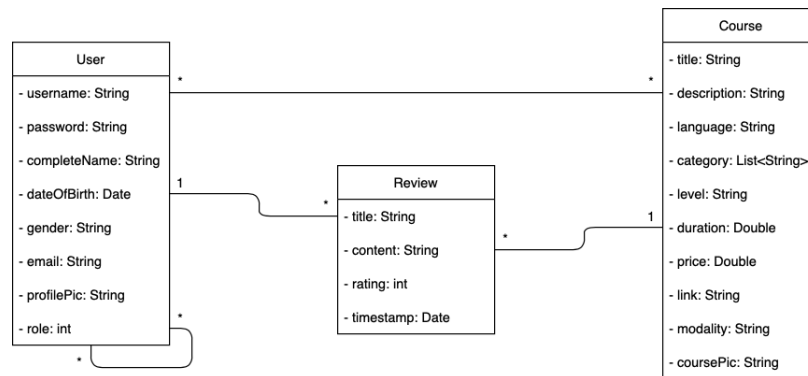The restructured version of the class diagram of the application is shown in Figure 3.3.

Figure 3.3: Restructured UML class diagram of *LearnIt!* application

3.4.1   Classes definitions

There are 3 main entities: *User*, *Course* and *Review*. User is a generalization of the two types of users that are: *Registered User* and *Administrator*.

3.4.2   Relationships between classes

Depending on *role* attribute of the *User*, the user can be considered a *Registered User* or a *Admin user*. The actions they can perform are different and are reported here below.

Registered User's relationships:

- a *User* can write zero or more *Reviews* related to different courses. An additional constraint is that a *User* can write only one *Review* about a *Course*.

- a *Review* is written by one and only one *User*

- a *Review* is related to one *Course*

- a *Course* can have zero or more *Reviews*

- a *User* can like zero or more *Courses*

- a *Course* can be liked by zero or more *Users*

- a *User* can offer zero or more *Courses*

- a *Course* is offered by one and only one *User*

- a *User* can follow zero or more different *Users*

- a *User* can be followed by zero or more different *Users*

Admin User's relationships:

- a *User* can delete zero or more *Reviews* related to different courses.

- a *Review* can be deleted by only one *User*

- a *User* can delete zero or more *Courses*

- a *Course* can be deleted by only one *User*

- a *User* can delete zero or more *Users*

- a *User* can be deleted by only one *User*

3.5 Design of the classes

In this section we have a short description of the entities' attributes. Some of them are optional.

3.5.1 User's attributes

username
     uniquely represents users in *LearnIt!*, it's picked by the user at the sign up and it's then used to perform the login. It must be unique.

complete name
     contains the first and last name of the user

date of birth
     is the date of birth of the user (optional)

gender
     describes the gender of the user (optional)

email
     contains the email of the user

password
     is the password that the user must enter at the login

role
     the user can be a normal user or an administrator in *LearnIt!*.

profile picture
     contains the URL of the user's profile picture (optional)

3.5.2 Course's attributes

title
     is the title of the course, it must be unique

description
     contains a short description of the contents of the course

instructor
     is the User that teaches the course

language
     is the language spoken in the lectures of the course

category
     is the set of categories the course belongs to (optional)

level

is the difficulty level of the course

duration

estimates the amount of time needed to complete the course (optional)

price

is the cost of the course (optional)

link

is the URL of the web-page of the course (optional)

picture

is the URL of the picture of the course (optional)

modality

states how users can attend the course (optional)

3.5.3 Review's attributes

title

is a short title of the review (optional)

content

is the comment about the course referred by the review (optional)

rating

is the score assigned to the course referred by the review

edited timestamp

is the timestamp of the last modification of the review

3.6 Data model

3.6.1 DocumentDB

The document database has been chosen because *LearnIt!* needs to handle large amounts of data in a flexible and efficient way. The fact that some entities' attributes are optional in our design of the classes is an indication that the document database is the mean to satisfy the *flexibility* requirement.

Example of json document representing a *Course*

```
1  {
2    "_id": {"$oid":"61bdf95cb37a46697e47da1a"},
3    "description": "Major Keys to Success",
4    "language": "English",
5    "duration": 82,
6    "level": "All Levels",
7    "title": "IELTS Band 7+ Complete Prep Course",
8    "course_pic": "https://i.imgur.com/R7KOk45.png",
9    "price": 134.99,
10   "category": ["Test Prep","IELTS"],
11   "link": "www.udemy.com/course/ielts-band-7-preparation-course",
12   "instructor": "adamanderson",
13   "num_reviews": 2,
14   "sum_ratings": 6,
15   "reviews": [
16       {"title": "Excellent",
17        "content": "Great Course!!",
18        "username": "norrisrogers",
19        "rating": 5,
20        "edited_timestamp": {"$date":"2016-08-21T00:00:00.000Z"}
21       },
22       {"title": "Really bad",
23        "content": "Quite a few of the questions were
24                    missing words or had typos that made
25                it difficult to determine what was
26                being asked.",
27        "username": "ikebauer",
28        "rating": 1,
29        "edited_timestamp": {"$date":"2017-10-17T00:00:00.000Z"}
30       }
31   ]
32  }
```

Since the reviews are always used in combination with their course's information and to solve the relationship *many-to-one* between a *Review* and the *User* who wrote it, we decided to embed the information about *reviews* inside the document of the object they refer to (a *course*). Furthermore, this organization of the documents allows to find all the information about a *course* and all its *review*s in a single operation.

With the embedding of the reviews in the courses' documents, we solved the *one-to-many* relationship between a course and its reviews.

We have chosen to store in the *course*'s document the username of the *user* teaching it because these detail is always needed when a *course* is to be shown.

In the *review*s' documents are stored the username of the *user* who wrote it, since it is needed to be shown when the *review* has to be visualized in the application.

Example of json document representing a *User*

```
1  {
2    "_id": {"$oid": "61f2bfc0d16c2b599a15a000"},
3    "username": "adamanderson",
4    "complete_name": "Adam Anderson",
5    "password": "8ee6a817d5511d9a3",
6    "birthdate": {"$date":"2000-10-26T22:00:00.000Z"},
7    "gender": "man",
8    "pic": "https://tinyurl.com/yckufye8",
9    "email": "adam.anderson@email.com",
10   "role": 0,
11   "reviewed":[
12      {
13        "title": "Mandarin Chinese Part 1",
14        "duration": 17,
15        "price": 18,
16        "course_pic": "https://i.imgur.com/R7KOk45.png",
17        "review_timestamp": {"$date":"2017-12-22T00:00:00.000Z"}
18      },
19      {
20        "title": "22 Practical Lessons in Leadership",
21        "duration": 1.5,
22        "price": 108.99,
23        "course_pic": "https://i.imgur.com/R7KOk45.png",
24        "review_timestamp": {"$date":"2018-07-02T00:00:00.000Z"}
25      },
26      {
27        "title": "Ace your Android Developer Test or Interview",
```

```
28        "duration": 1.5,
29        "course_pic": "https://i.imgur.com/R7KOk45.png",
30        "review_timestamp": {"$date":"2021-07-01T00:00:00.000Z"}
31      },
32   ]
33 }
```

We applied the embedding of the reviewed courses' snapshots in an array in the document of the *User* who completed them. The embedding of the courses makes faster the information retrieval about the completed courses (reviewed courses), at the cost of having to ensure the consistency, so at the cost of having to keep updated the courses' documents both in the user's document and in the course's document.

When a course is edited by its instructor, if the edited fields are in the set of attributes maintained by the snapshot of the course in the users' documents we have to ensure that the data is updated. To maintain also information about the real history of completed courses for a user, including the real duration and price of the course at the time of the review by the user, we added to the snapshot the fields "new_year" and "new_duration", so users can browse the history of completed courses of other users and see the correct actual information for the courses that have been updated, while the statistics of the completed courses of a user remain consistent.

Too many embedded documents problem

Since we decided to embed *Reviews* inside *Course*'s document, we had to handle the problem that can arise when too many embedded reviews should be stored inside a course's document that will grow very big. The solution we adopted is to use a threshold to limit the number of reviews to store in a document and if the threshold is reached then a specific module of our application should be executed by the Database Administrator to remove some reviews to make free space for new reviews according to some standards that we have defined. In particular, we decided to keep a number of embedded reviews that should be equal to a prefixed threshold (that he can specify in the config.xml file of the module DatabaseMaintenance). The reviews should be kept only if they are representative of the course (meaning that they have a rating similar to the average rating of the course), if instead their rating is very distant from the average rating, then they will be removed from the document; if a minimum threshold of representative reviews about a course is not reached, we keep the most recent reviews instead. The reviews removed from DocumentDB will still be present as a relationship in GraphDB since the application should consider the course as completed in order to make precise personalized suggestions about courses and avoid duplicated reviews from the same user.

We rejected the hypothesis that a the User's document can grow too much caused by a too large number of embedded courses' documents since it is very

17

unlikely (a User should write approximately 32.000 reviews).

3.6.2 GraphDB

The graph database is used to perform fast networking, suggestion queries that need to scan several relationships between users or relationships between users and courses and avoid multiple joins that may be too computationally intensive and would not guarantee the *low latency* requirement.
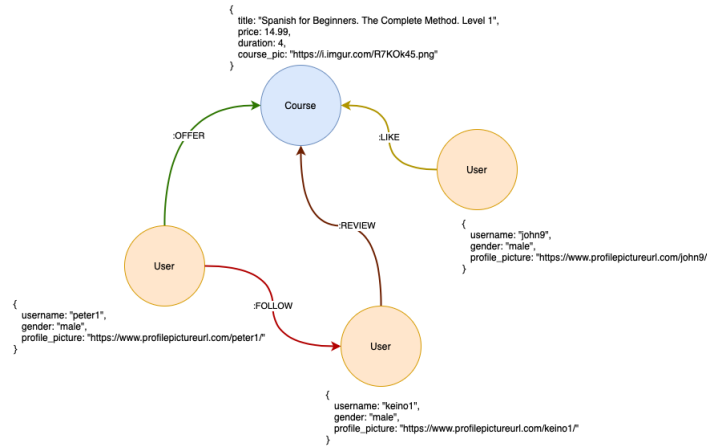
Example of graph

Figure 3.4: An example of graph

Nodes of the graph

Within the database, there are two type of nodes:

User
> represents a user inside the graph. Its attributes are: *username, gender, profile picture.*

Course
> represents a course inside the graph. Its attributes are: *title, price, duration, course picture*

We decided to use GraphDB to handle the social part of *LearnIt!*, while all the other features are provided by DocumentDB because if it's velocity, so the only information stored in graphDB are those that are needed to show a snapshot of a user or a course and the information needed to compute personalized courses' suggestions.

We decided that was better to store some redundant attributes (already present in DocumentDB) with the aim of having all the information needed to perform a query in the database where the query will be performed (so avoiding queries that involve both the databases).

Relationships between nodes of the graph

The GraphDB relationships are:

Like

If the user U leaves a like on a course C, we have a relationship *:LIKE* from U to C ($U \xrightarrow{LIKE} C$)

Review

If the user U completes a course C, they have to leave a review for the course, creating a *:REVIEW* relationship from U to C ($U \xrightarrow{REVIEW} C$)

Follow

If the user U follows another user S, there is a *:FOLLOW* relationship from U to S ($U \xrightarrow{FOLLOW} S$)

Offer

If the user U wants to offer a course C as a teacher, there is a *:OFFER* relationship from U to C ($U \xrightarrow{OFFER} C$)

3.7 Distributed Database Design

According to the non-functional requirements of the *LearnIt!* application, we should guarantee **high availability**, **low-latency** and **partition protection**. We oriented our application to the **AP edge** of the CAP triangle ensuring the **eventual consistency**.

In order to ensure that the **partition protection** and **availability** requirements were satisfied, we've designed a distributed system (exploiting 3 virtual machines provided by University of Pisa). Thanks to the three different replicas we decided to use, the system guarantees avoiding the single point of failure and guarantee also the **low latency** requirement thanks to the balancing of the load (different requests made by clients are handled by different servers in order to guarantee a fast response) and thanks to the fact that after a write operation is received only one replica's data will be immediately updated while the data on the other replicas will be updated in a second moment in order to don't keep the server busy for too much time during a write operation.

3.7.1 Replicas

Three replicas are present in our system for the document database, one for each machine of the cluster provided by the University of Pisa. For graph database we have only one replica, but theoretically there should have been three of them (it's a premium feature). Summing up everything, the system has:

- 3 replicas for the document database

- 1 instance for the graph database

- only one replica out of three must be updated in order to commit a write operation

- we choose nearest read preference

Write concern

For what concerns the write operations, we choose that they can be considered completed as soon as the primary replica set member has successfully completed the write. Our decision was led by the need to ensure the **low-latency** requirement.

Read preference

We decided to set the read preference as **nearest** because we want to guarantee as fast as possible responses to the clients' requests. Since we can consider *LearnIt!* as a **read-heavy application**, it is fundamental to quickly respond to the client, this is why we choose to set the read preference to nearest replica.

3.7.2   Sharding

We have to choose two shard keys: one for the collection of courses and one for the collection of users. The partitioning method we find the most appropriate for our case is consistent hashing. It allows us to change nodes in the cluster and remap only keys mapped to the neighbors of the node that has been removed/added. For the course collection, many fields are always present in all documents. For the purpose of consistent hashing we have chosen the Object ID of the documents. For the users collection, we thought of using the username as shard key.

3.8   Software Architecture
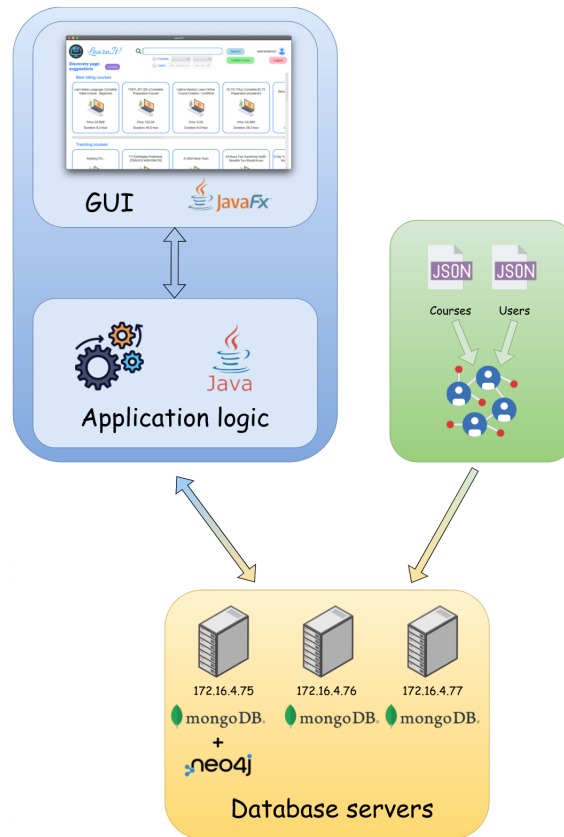
Figure 3.5: Software architecture of *LearnIt!*

3.8.1   Client-Server architecture

Client
The client is composed of two modules:

- front-end, consisting of the GUI, allowing the users to have an intuitive interaction with the application

- middle-ware, interfacing the client with the server

Server

The server side consists of three virtual machines which serve the requests of the clients.

3.8.2 Inter-Databases Consistency

Considering the two types of databases that we use to store the data (DocumentDB and GraphDB), we have to take in consideration all the problems derived by the need of consistency between data since there are some redundancies in the courses' attributes stored in Graph and Document and at every change, the information on both the databases should be updated.

We should also consider this case: the first update may be successful but in the second something goes wrong; in that case we have to undo the successful update to maintain consistency. But we could also have the opposite situation: the first update operation is not successful and in that case we can't even start the second one.

Operations that can cause inconsistencies are:

- Add a course
- Update course (name, price, duration)
- Delete a course
- Delete a review
- Add a review
- Update a review
- Delete a user
- Update user (username, gender, profile-picture)
- Add a user

Implementation & Test

4.1   Package structure

4.1.1   Main Packages and classes

In this section we describe the packages and the classes of which is composed the *LearnIt!* application.

it.unipi.dii.inginf.lsdb.learnitapp.model

This package contains the classes needed for the java bean model of the application.

- User: class that contains user's information

- Review: class that contains review's information

- Course: class that contains course's information

- Session: class that contains information about the session

it.unipi.dii.inginf.lsdb.learnitapp.persistence

This package contains the classes that are used to interface with the database.

- Neo4jDriver: implement the methods to access Neo4j and implements the queries to interact with Neo4j

- MongoDBDriver: implement the methods needed to access MongoDB and implements the queries to interact with MongoDB

- DBDriver: this class is an interface for a generic DBDriver

it.unipi.dii.inginf.lsdb.learnitapp.service

This package contains the class that is used to handle the inter-database consistencies and the roll-back operation.

- LogicService: handles the queries that involve both the databases

it.unipi.dii.inginf.lsdb.learnitapp.controller

This package contains the controllers of all the different pages that can be shown to the user.

- CoursePageController: controller of the page of a course

- CourseSnapshotController: controller of the snap-shot of a course

- DiscoveryPageController: controller of the home page

- ElementsLineController: controller of a line of snap-shots (of users or courses)

- LoginPageController: controller of the login page

- NewCoursePageController: controller of the new-course page

- PersonalPageController: controller of the page from which the user can edit their personal information

- ProfilePageController: controller of the page of a user

- RegitrationPageController: controller of the sign-in page

- ReviewSnapshotPageController: controller of the snap-shot of a review

- CourseSnapshotController: controller of the snap-shot of a user

it.unipi.dii.inginf.lsdb.learnitapp.utils

This package contains the class used to implement some utility function.

- Utils: implements some utility function used by more classes

it.unipi.dii.inginf.lsdb.learnitapp.log

This package contains the class used to handle the log of the errors.

- LearnitLogger: handles the log operations

it.unipi.dii.inginf.lsdb.learnitapp.config

This package contains the class used to handle the configuration parameters of the *LearnIt!* application.

- ConfigParams: class which handles the configuration parameters

it.unipi.dii.inginf.lsdb.learnitapp.app

This package contains the main of the *LearnIt!* application.

- Learnit: main of the application

4.2 Redundancies introduction

For the purpose of making some queries faster, two redundant field was added to the database: *sum_ratings* and *num_reviews* which are used to compute the mean rating of a course without needing to scan all the reviews about a course every time it is required to retrieve the mean rating.

4.3 Handling inter-database consistency

To prevent possible inter-database inconsistencies we had to perform some queries on both the database and to ensure that those queries are successfully completed only if the operation on both the databases ends with a success and if one of the two operations fails, we should guarantee a consistent state. To solve this problem we decided to use an error log described here below.
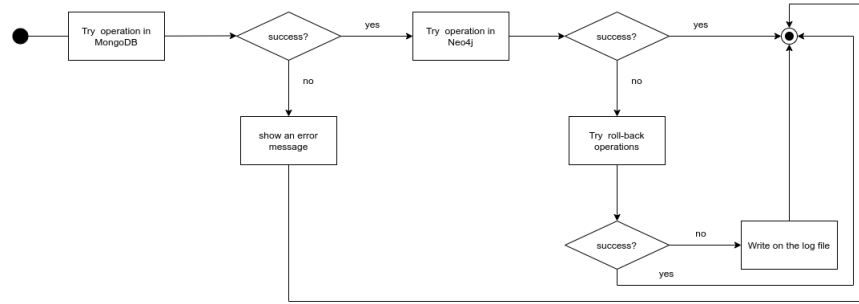
Figure 4.1: Handling inter-database consistency
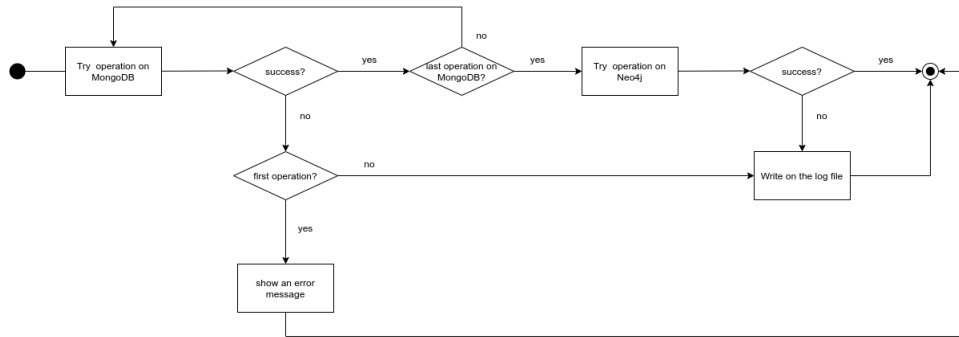
Figure 4.2: Handling inter-database consistency of the delete user operation

27

Error log

The log file contains information about the type of operation that failed and the object that should be restored.

```
 1  ERROR [LogicService.java:244] - Timed out after 30000 ms while waiting to
            connect. Client view of cluster state is {type=UNKNOWN, servers=[{
            address=127.0.0.1:27017, type=UNKNOWN, state=CONNECTING, exception={
            com.mongodb.MongoSocketOpenException: Exception opening socket},
            caused by {java.net.ConnectException: Connection refused}}]
 2  ERROR [LogicService.java:245] - ADD USER: ROLLBACK FAILED
 3  ERROR [LogicService.java:246] - it.unipi.dii.inginf.lsdb.learnitapp.model
            .User@5b7bd437[
 4    username=provaerrore
 5    completeName=prova
 6    dateOfBirth=1999-02-19
 7    gender=Female
 8    email=email@email.com
 9    profilePic=picture
10  ]
11
12  ERROR [LogicService.java:57] - Timed out after 30000 ms while waiting to
            connect. Client view of cluster state is {type=UNKNOWN, servers=[{
            address=127.0.0.1:27017, type=UNKNOWN, state=CONNECTING, exception={
            com.mongodb.MongoSocketOpenException: Exception opening socket},
            caused by {java.net.ConnectException: Connection refused}}]
13  ERROR [LogicService.java:58] - DELETE COURSE: ROLLBACK FAILED
14  ERROR [LogicService.java:59] - it.unipi.dii.inginf.lsdb.learnitapp.model.
            Course@628c89f2[
15    title=Spanish for Beginners. The Complete Method. Level 1
16    description=The complete, non-stop SPEAKING Spanish course: Mastery of
              the basics for BEGINNERS - in a matter of hours, not years.
17    instructor=adamanderson
18    language=English
19    level=Beginner
20    category=[Teaching & Academics, Language Learning, Spanish Language]
21    price=131.0
22    duration=4.0
23    modality=<null>
24    numReviews=3
25    sumRatings=12
26    link=https://www.udemy.com/course/el-metodo-spanish-1/
27    coursePic=https://i.imgur.com/R7KOk45.png
28  ]
29
30  ERROR [LogicService.java:117] - Timed out after 30000 ms while waiting to
            connect. Client view of cluster state is {type=UNKNOWN, servers=[{
            address=127.0.0.1:27017, type=UNKNOWN, state=CONNECTING, exception={
            com.mongodb.MongoSocketOpenException: Exception opening socket},
            caused by {java.net.ConnectException: Connection refused}}]
```

```
31  ERROR [LogicService.java:118] - DELETE REVIEW: ROLLBACK FAILED
32  ERROR [LogicService.java:119] - it.unipi.dii.inginf.lsdb.learnitapp.model
        .Course@553220eb[
33    title=4 Soft Skills for Videographers & Photographers To Succeed
34    description=Skills needed to grow your business, packaged in a single
          course.
35    instructor=montehaworth
36    language=English
37    level=Intermediate
38    category=<null>
39    price=110.0
40    duration=1.0
41    modality=<null>
42    numReviews=16
43    sumRatings=71
44    link=https://www.udemy.com/course/4-soft-skills-for-videographers-
          photographers-to-succeed/
45    coursePic=https://i.imgur.com/R7KOk45.png
46  ]
47  ERROR [LogicService.java:120] - it.unipi.dii.inginf.lsdb.learnitapp.model
        .Review@ea10413[
48    username=benanderson
49    title=Really good
50    content=
51    rating=4
52    timestamp=Sat Oct 09 02:00:00 CEST 2021
53  ]
```

4.4 Java Model of the Entities

Here we report the Java Model used for handling the entities of the databases. For readability setters, getters and constructors were omitted.

4.4.1 User

```java
public class User {
    @BsonProperty(value = "username")
    private String username;
    @BsonProperty(value = "password")
    private String password;
    @BsonProperty(value = "complete_name")
    private String completeName;
    @BsonProperty(value = "date_of_birth")
    private Date dateOfBirth;
    @BsonProperty(value = "gender")
    private String gender;
    @BsonProperty(value = "email")
    private String email;
    @BsonProperty(value = "role")
    private Role role;
    @BsonProperty(value = "profile_picture")
    private String profilePic;
}
```

4.4.2 Review

```java
public class Review {
    @BsonProperty(value = "title")
    private String title;
    @BsonProperty(value = "content")
    private String content;
    @BsonProperty(value = "rating")
    private int rating;
    @BsonProperty(value = "edited_timestamp")
    private Date timestamp;
    @BsonProperty(value = "author")
    private User author;
}
```

4.4.3 Course

```java
public class Course {
    @BsonId
    private ObjectId id;
    @BsonProperty(value = "title")
    private String title;
    @BsonProperty(value = "description")
    private String description;
    @BsonProperty(value = "instructor")
    private User instructor;
    @BsonProperty(value = "language")
    private String language;
    @BsonProperty(value = "category")
    private List<String> category;
    @BsonProperty(value = "level")
    private String level;
    @BsonProperty(value = "duration")
    private double duration;
    @BsonProperty(value = "price")
    private double price;
    @BsonProperty(value = "link")
    private String link;
    @BsonProperty(value = "modality")
    private String modality;
    @BsonProperty(value = "reviews")
    private List<Review> reviews;
    @BsonProperty(value="num_reviews")
    private int num_reviews;
    @BsonProperty(value="sum_ratings")
    private int sum_ratings;
    @BsonProperty(value="course_pic")
    private String coursePic;
}
```

4.5 Most relevant queries

In this section we report some of the most relevant queries giving a short description of what the queries do, their inputs and outputs and their implementation. Where possible, we tried to not use the unwind operation, but in some cases it was the

4.5.1 MongoDB

In some of the MongoDB queries we used the *$unwind* operator because we needed to compute some aggregate values (average, sum, group by, ...) using separately each of the fields of an array of embedded documents (array of reviews or array of courses).

Find course given some filters or given the title
The query searches for courses that match some specific characteristics. In particular you can decide to specify: a threshold for the maximum price, a threshold for the maximum duration, the difficulty level of the course, the language of the course and a sub-string that should be contained in the title of the course.

- Input: price threshold, duration threshold, title, level, language, skip, limit

- Output: a reduced set of founded courses' snap-shots

```
db.learnit.find({
 $or: [{price: {$lte: <price>}}, {price: undefined}],
 $or: [{duration: {lte: <duration>}}, {duration: undefined}],
 level: "<level>",
 language: "<language>",
 title: {$regex: /^.*<title>.*$/i}
}).skip(<skip>).limit(<limit>)
```

Find courses with best rating

This query finds the X courses with the higher ratings where X is the number of courses to get.

- Input: limit (X)

- Output: set of at most X courses sorted for higher rating

```
db.learnit.aggregate([
 {$match: {num_reviews: {$gt: 0}}},
 {$addFields: {
             avg: {$divide: ['$sum_ratings',
                             '$num_reviews']
                 }
            }
 },
 {$sort: {avg: -1}},
 {$limit: <limit>}
])
```

View annual average ratings of a specific course

This query computes the average annual rating of a specified course associated to the relative year.

- Input: id of the course

- Output: list of years and mean annual rating relative to the years

```
db.learnit.aggregate([
 {$match: {_id: ObjectId('<id>')}},
 {$unwind: {path: '$reviews'}},
 {$group:
       {_id: {year:
                {$year: '$reviews.edited_timestamp'}
             },
        sum_ratings: {$sum: '$reviews.rating'},
        num_reviews: {$sum: 1}
       }
 },
 {$set: {year: '$_id.year'}},
 {$sort: {year: -1}}
])
```

Best users

This query finds the top X users based on their ability on choosing the best courses based on the ratio between the duration and the price.

- Input: limit (X), skip

- Output: a list of X users' snap-shots

```
db.users.aggregate([
 {$unwind: {path: '$reviewed'}},
 {$match: {'reviewed.duration': {$gt: 0},
           'reviewed.price': {$gt: 0}
          }
 },
 {$addFields:
           {quality_ratio: {$divide:
                                   ['$reviewed.duration',
                                    '$reviewed.price'
                                   ]
                           }
           }
 },
 {$group: {_id: {username: '$username'},
           value: {$avg: 'quality_ratio'}
          }
 },
 {$project: {_id: 0,
             username: '$_id.username',
             value: '$value'
            }
 },
 {$sort: {value: -1}},
 {$skip: <skip>},
 {$limit: <limit>}
])
```

Trending courses

This query discovers which are the X courses having the largest number of reviews written in the last week (the most popular courses of the month).

- Input: limit (X), date (30 days before today's date)

- Output: a list of X courses' snap-shots

```
db.courses.aggregate([
 {$match: {'reviews.edited_timestamp': {$gt: <date>}}},
 {$unwind: {path: '$reviews'}},
 {$group: {_id: {_id: '$_id'},
          size: {$sum: 1}
          }
 },
 {$sort: {size: -1}},
 {$limit: <limit>}
])
```

Most active users

This query searches for the X users that have written the highest number of reviews in the last 30 days.

- Input: limit (X), date (30 days before today's date)

- Output: a list of X users' snap-shots

```
db.users.aggregate([
 {$match: {reviewed: {$exists: true}}},
 {$unwind: {path: '$reviewed'}},
 {$match: {'reviewed.review_timestamp': {$gt: <date>}}},
 {$group: {_id: {username: '$username'},
          count: {$sum: 1}
          }
 },
 {$sort: {count: -1}},
 {$project: {username: '$_id.username', _id: 0}},
 {$limit: <limit>}
])
```

Compute average statistics of a user

The aim of this query is to compute the average statistics of a user, in particular it calculates the average money spent on courses, the average duration of reviewed courses and total number of reviewed courses.

- Input: username of the user

- Output: average price, average duration, number of completed courses

```
db.users.aggregation([
 {$match: {username: '<username>',
          reviewed: {$exists: true}
          }
 },
 {$unwind: {path: '$reviewed'}},
 {$group: {_id: {username: '$username'},
          count: {$sum: 1},
          avgprice: {$avg: '$reviewed.price'},
          avgduration: {$avg: '$reviewed.duration'}
          }
 },
 {$project: {_id: 0,
             username: '$_id.username',
             count: 1,
             avgprice: 1,
             avgduration: 1
             }
 }
])
```

4.5.2 GraphDB

Find suggested courses considering instructors of the others courses you have completed
This query finds a set of suggested courses for the specified user based on the courses offered by the same instructor of other courses that the specified user completed (reviewed).

- Input: username, skip, limit

- Output: a reduced set of snap-shots of suggested courses

```
MATCH (c:Course)<-[:OFFER]-(u2: User)-[:OFFER]->(c2:Course)<-[:
    REVIEW]-(u:User {username :<username>})
WHERE NOT ((u)-[:REVIEW]->(c) )
        AND (c.id <> c2.id)
        AND (u2.username <> u.username)
RETURN c
SKIP <skip> LIMIT <limit>
```

Get stats about follower and following users
This query, given a specified user, computes the total number of users followed by the specified users and the total number of users who follow the user.

- Input: username

- Output: follower and following numbers

```
MATCH (u:User {username: <username>})
OPTIONAL MATCH (u)<-[f1:FOLLOW]-()
OPTIONAL MATCH (u)-[f2:FOLLOW]->()
RETURN COUNT(DISTINCT f1) AS follower
        COUNT(DISTINCT f2) AS following
```

Find suggested courses

This query recommends courses to a specified user based on two levels of suggestion: the first levels suggests courses liked by most users followed by this user, while the second level finds the courses liked by the users (having at least "numRelationships" liked courses in common with this user) followed by users already followed by this user. Suggested courses are returned only if they are not already reviewed or liked by this user and are ordered by decreasing duration and ascending price.

- Input: username, skip1 (1^{st}level), skip2 (2^{nd}level), limit1 (1^{st}level), limit2 (2^{nd}level), relationships (lower threshold for the number of relationships for the second level)

- Output: a set of snap-shots of suggested courses

```
MATCH (c:Course)<-[l:LIKE]-(u:User)<-[f:FOLLOW]-(me:User{username
    :<username>})
WHERE NOT EXISTS((me)-[:LIKE]->(c))
WITH DISTINCT(c) as c, COUNT(*) as numUser
RETURN c
ORDER BY numUser DESC
SKIP <skip1> LIMIT <limit1>

UNION

MATCH (me:User{username:<username>})-[l:LIKE]->(c:Course)<-[l1:
    LIKE]-(u:User)<-[:FOLLOW*2..2]-(me)
WHERE NOT EXISTS((me)-[:FOLLOW]->(u)) AND me.username <> u.
    username
WITH DISTINCT(u) as user, COUNT(DISTINCT(c)) as numRelationships
WHERE numRelationships > <relationships>
MATCH (user)-[:LIKE|:REVIEW]->(c:Course)
WHERE NOT EXISTS((:User{username:<username>})-[:LIKE|:REVIEW]->(c
    ))
RETURN DISTINCT c
ORDER BY CASE c.duration WHEN null THEN 0 ELSE c.duration END
    DESC,
        CASE c.price WHEN null THEN 0 ELSE c.price END ASC
SKIP <skip2> LIMIT <limit2>
```

Find suggested users

This query recommends users to a specified user based on two levels of suggestion: the first level suggests not followed users who are followed by at least X (followed threshold) users followed by the specified user, ordered by total number of followers in common between the users, while the second level recommends those users (not followed by the user) who like at least K (common courses threshold) courses also liked by the specified user, ordered by the total number of liked courses in common between the users.

- Input: followed threshold (X), common courses threshold (K), username, skip1 (1^{st}level), skip2 (2^{nd}level), limit1 (1^{st}level), limit2 (2^{nd}level)

- Output: a set of snap-shots of suggested users

```
MATCH (me:User {username: <username>})-[:FOLLOW*2..2]->(u:User)
WHERE NOT EXISTS((me)-[:FOLLOW]->(u)) AND u.username <> me.
    username
WITH DISTINCT(u) as user, COUNT(u) as followed
WHERE followed > <followed threshold>
RETURN user
ORDER BY followed DESC
SKIP <skip1> LIMIT <limit1>

UNION

MATCH (me:User {username: <username>})-[:LIKE]->(commonCourse:
    Course)<-[:LIKE]-(u:User)
WHERE NOT EXISTS((me)-[:FOLLOW]->(u))
WITH DISTINCT(u) as user, COUNT(commonCourse) as numCommonCourses
WHERE numCommonCourses > <common courses threshold>
RETURN user
ORDER BY numCommonCourses DESC
SKIP <skip2> LIMIT <limit2>
```

Most liked courses

The aim of this query is to find the X most liked courses ordered by total number of likes and higher ratio between duration and price.

- Input: limit

- Output: a set of at most limit (X) courses' snap-shots

```
MATCH (:User)-[l:LIKE]->(c:Course)
RETURN c,
       COUNT (l) AS like_count,
       CASE c.duration
            WHEN null THEN 0
                ELSE CASE c.price
                          WHEN null THEN 1
                          ELSE c.duration/c.price
                     END
       END AS quality_ratio
ORDER BY like_count DESC, quality_ratio DESC
LIMIT <limit>
```

Most followed users

This query searches for the X most popular user based on the total number of followers.

- Input: limit

- Output: a set of at most limit (X) users' snap-shots

```
MATCH (u:User)<-[f:FOLLOW]-(u2:User)
OPTIONAL MATCH (u)-[l:LIKE]->(:Course)
WHERE u.username <> u2.username
WITH DISTINCT (u), COUNT(DISTINCT(f)) AS followers, COUNT(
    DISTINCT(l)) AS likes
RETURN u
ORDER BY followers DESC, likes DESC
LIMIT <limit>
```

4.6 Tests and Statistical Analysis

4.6.1 Index Analysis

To improve the performances of the application we performed index analysis, in this subsection we will discuss the introduction of indexes for both Neo4J and MongoDB.

Indexes introduction on MongoDB

Query	Index	ms	keys	docs
Find courses offered by a user	-	96	0	107.100
	instructor	3	17	17
Get course by title	-	81	0	107.100
	title	0	1	1
Find course by parameters	-	156	0	107.100
	title	250	107100	7003
	language	212	101384	101384
	level	144	53798	53798
Find average statistics of a user	-	25	0	6300
	username	8	1	1
Login	-	54	0	6300
	username	5	1	1
searchUserByUsername	-	65	0	6300
	username	65	0	6300
	complete_name	25	6300	150
GetUserByUsername	-	62	0	6300
	username	8	1	1

Indexes

Given the results of the experiments we performed to see which indexes improve more the performances of the queries, we decided to create three indexes on MongoDB: one on the course's *title*, one on the course's attribute *instructor* and one on the user's attribute *username*.

```
db.courses.createIndex({"title": 1},{unique: true, name: "
    title_index"})
db.courses.createIndex({"instructor": 1},{name: "instructor_index
    "})
db.users.createIndex({username: 1},{unique: true, name:
    username_index})
```

Indexes introduction on GraphDB

Query	Index	Hits	ms
findSuggestedCoursesByCompletedCourses	-	507306	680
	username	27473	509
FindSuggestedCourses	-	45239	399
	username	7435	996
FindSuggestedUsers	-	49903	1012
	username	12379	675
FindCoursesLikedOrCompletedByUser	-	19197	330
	username	295	205
GetFollowStats	-	19489	292
	username	587	398
GetCourseLikes	-	126258	480
	title	10	74
IsCourseLikedByUser	-	19002	303
	username	100	294
	title	12	428
isCourseReviewedByUser	-	19164	32
	username	262	184
	title	20	284

Indexes

As we can notice from the tables reported above, it is extremely convenient to add two indexes to neo4j's database in order to improve the performances of the queries. The two indexes that we created are on the attribute *username* on the entity *User* and on the attribute *title* of the entity *Course*.

```
CREATE INDEX username_index FOR (u:User) ON u.username
CREATE INDEX title_index FOR (c:Course) ON c.title
```

User Manual

In this section we explain how a user can use *LearnIt!*.

5.1   Configuration

The user can configure some parameters by updating the configuration file. The configuration file is validated using the *config.xsd* file.

5.1.1   Configuration file - config.xml

```xml
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <it.unipi.dii.inginf.lsdb.learnitapp.config.ConfigParams>
 3      <mongoDBPrimaryIP>172.16.4.77</mongoDBPrimaryIP>
 4      <mongoDBPrimaryPort>27020</mongoDBPrimaryPort>
 5      <mongoDBSecondIP>172.16.4.76</mongoDBSecondIP>
 6      <mongoDBSecondPort>27020</mongoDBSecondPort>
 7      <mongoDBThirdIP>172.16.4.75</mongoDBThirdIP>
 8      <mongoDBThirdPort>27020</mongoDBThirdPort>
 9      <mongoDBUsername>root</mongoDBUsername>
10      <mongoDBPassword></mongoDBPassword>
11      <mongoDBName>db</mongoDBName>
12      <mongoDBCollectionCourses>learnit</mongoDBCollectionCourses>
13      <mongoDBCollectionUsers>users</mongoDBCollectionUsers>
14      <neo4jIP>127.0.0.1</neo4jIP>
15      <neo4jPort>7687</neo4jPort>
16      <neo4jUsername>neo4j</neo4jUsername>
17      <neo4jPassword>password</neo4jPassword>
18      <limitNumber>10</limitNumber>
19      <limitFirstLvl>5</limitFirstLvl>
20      <limitSecondLvl>5</limitSecondLvl>
21      <numRelationships>1</numRelationships>
22      <numCommonCourses>1</numCommonCourses>
23      <followedThreshold>1</followedThreshold>
24  </it.unipi.dii.inginf.lsdb.learnitapp.config.ConfigParams>
```

5.1.2 Validating file - config.xsd

```xml
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3    <xs:element name="it.unipi.dii.inginf.lsdb.learnitapp.config.
         ConfigParams">
4      <xs:complexType>
5        <xs:sequence>
6          <xs:element name="mongoDBPrimaryIP" type="xs:string"/>
7          <xs:element name="mongoDBPrimaryPort" type="xs:int"/>
8          <xs:element name="mongoDBSecondIP" type="xs:string"/>
9          <xs:element name="mongoDBSecondPort" type="xs:int"/>
10         <xs:element name="mongoDBThirdIP" type="xs:string"/>
11         <xs:element name="mongoDBThirdPort" type="xs:int"/>
12         <xs:element name="mongoDBUsername" type="xs:string"/>
13         <xs:element name="mongoDBPassword" type="xs:string"/>
14         <xs:element name="mongoDBName" type="xs:string"/>
15         <xs:element name="mongoDBCollectionCourses" type="xs:string"/>
16         <xs:element name="mongoDBCollectionUsers" type="xs:string"/>
17         <xs:element name="neo4jIP" type="xs:string"/>
18         <xs:element name="neo4jPort" type="xs:int"/>
19         <xs:element name="neo4jUsername" type="xs:string"/>
20         <xs:element name="neo4jPassword" type="xs:string"/>
21         <xs:element name="limitNumber" type="xs:int"/>
22         <xs:element name="limitFirstLvl" type="xs:int"/>
23         <xs:element name="limitSecondLvl" type="xs:int"/>
24         <xs:element name="numRelationships" type="xs:int"/>
25         <xs:element name="numCommonCourses" type="xs:int"/>
26         <xs:element name="followedThreshold" type="xs:int"/>
27       </xs:sequence>
28     </xs:complexType>
29   </xs:element>
30 </xs:schema>
```

5.2   Login and registration

To have access to all the functionalities of the application, the user must be logged; after the login, the *Discovery Page* is shown.
Registered users can login to *LearnIt!* after they inserted their username and password on the login form and they clicked the login button.

Figure 5.1: Login form

If a user is not registered to the application, they can join the community after clicking on the sign up button, inserting their personal information on the registration form and clicking on the registration button. To ensure security , all the information is checked before the registration of a user and if there is a field that does not respect the rules of the *LearnIt!* application, the registration is rejected with an error message.

Figure 5.2: Registration form

5.3   Homepage - Discovery page
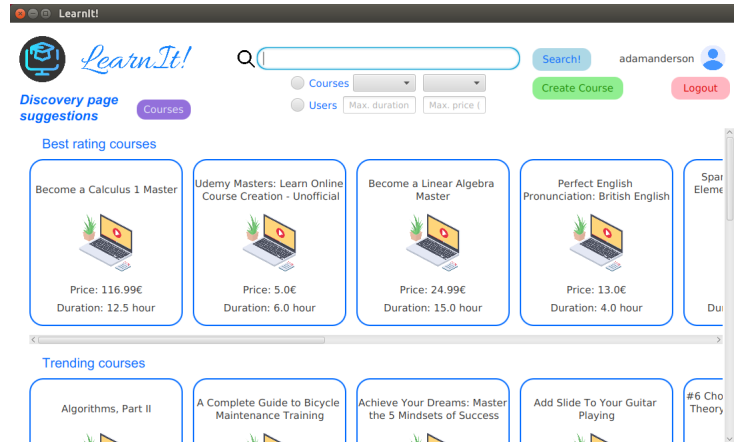
Figure 5.3: Discovery page

From this page, users can search for courses or users by selecting the type of research they are interested to and by typing the course's title or user's username on the search bar and pressing the search button. During a courses' search, users can specify some filters of search by selecting language, level or price/duration thresholds.
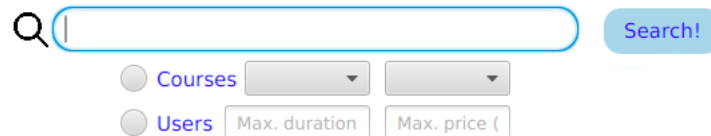
Figure 5.4: Search bar and research filters

The results of the research are shown in a grid like the one reported in figure 5.5. In every moment, the user can go back to the suggestions' section with a click on the *LearnIt!* logo or they can search another user or course using again the search bar and the search filters.

In this page, users can also receive suggestion about courses or about users to follow. By default in the Discovery Page are shown courses' suggestion, but clicking on the suggestions' button the user can switch the type of suggestions they want to receive.

With a click on a course's or user's snap-shot, the user will be redirected to the *Course Page* or to the *Profile Page* relative to the course or user clicked.

By clicking on the profile image in the high-right corner, users can have access to their *Personal Page*.
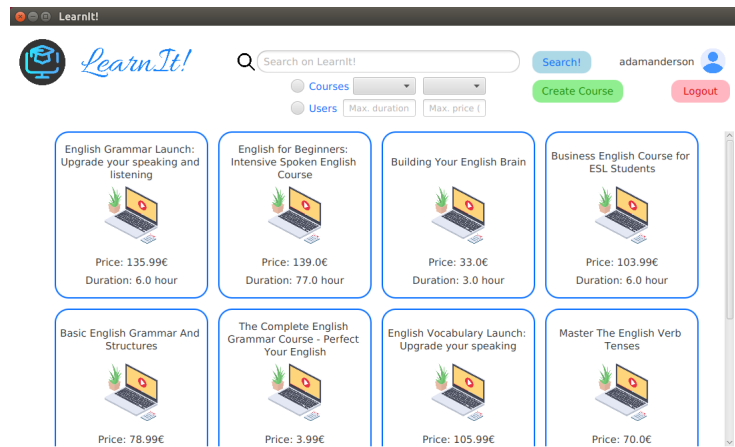
Figure 5.5: Search results
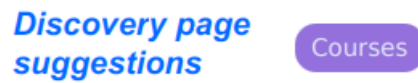
Figure 5.6: Suggestions' button on Discovery Page

Figure 5.7: User's snap-shot
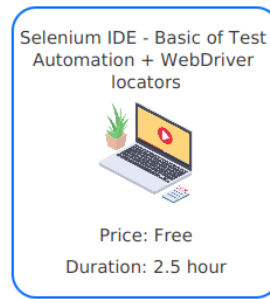
Figure 5.8: Course's snap-shot
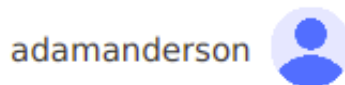
Figure 5.9: How to go to your personal page by the Discovery Page

If a user wants to create a new course, they can click on the create new course button and they will be redirected to the *New Course Page* where a form will be shown and the user will have to insert some information about the course and then click on the save button (the new course will be registered only if it respects the *LearnIt!*'s rules, else it will be shown an error message).

Figure 5.10: New course button on Discovery Page

Figure 5.11: New Course Page

If a user wants to delete a course taught by him or her, they can press on the trashcan button in the page of the course.
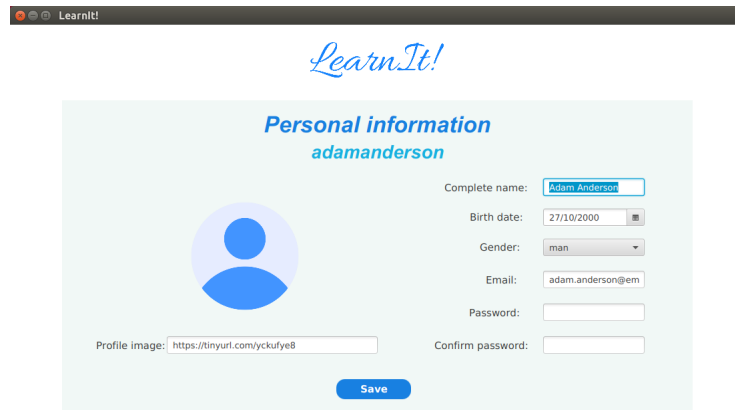
Figure 5.12: Delete course on course's page

5.4 Personal page

In the personal page, users can see their personal information and edit them, by clicking on the save button, if the new informations respect the rules of *LearnIt!*, they will be updated.

Figure 5.13: Personal page

5.5 Handle a course

5.5.1 Handling courses as an instructor

A user can add a new course from the *Discovery Page* or they can have access to all the information related to his course by searching it on the *Discovery Page* and going to its *Course Page.*
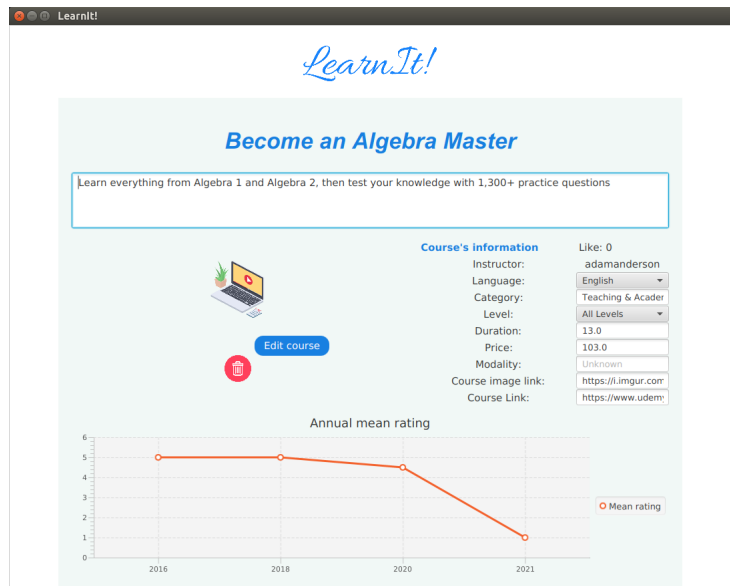
Figure 5.14: Course Page of a taught course

From the *Course Page* of a course for which the logged user is an instructor, the user can see all the informations about the course and they can edit them or they can delete the course by clicking on the trash can button.

From the same page, the user can see the reviews about the course and visit the reviews' authors' profiles by clicking on the profile images shown in the reviews' snap-shots.

Figure 5.15: A review snap-shot

5.5.2 Handling courses as a participant

From a *Course Page*, the logged user can see all the informations, statistics and reviews about a course and can like or unlike the course by clicking on the like/unlike button.
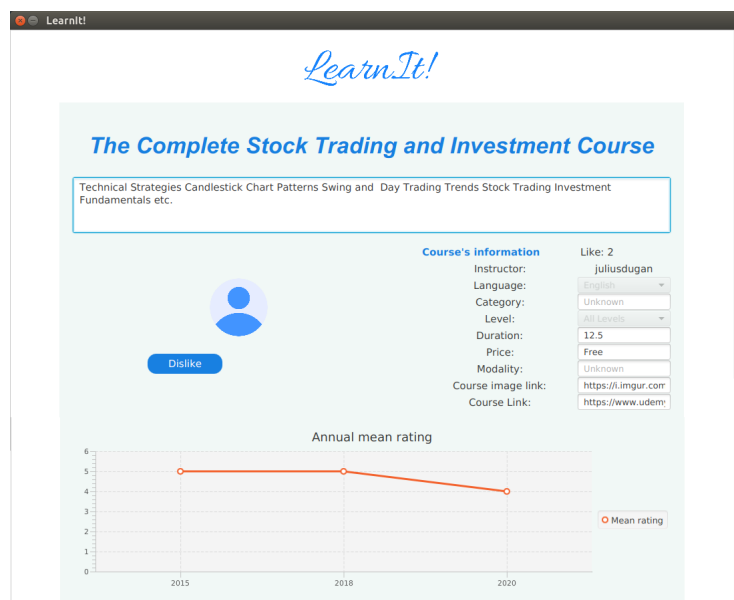
Figure 5.16: Course information of a non-taught by the user course

If the user has already written a review about the course, it is shown. An example of personal review about a course is shown in Figure 5.17 and it can be edited or deleted.
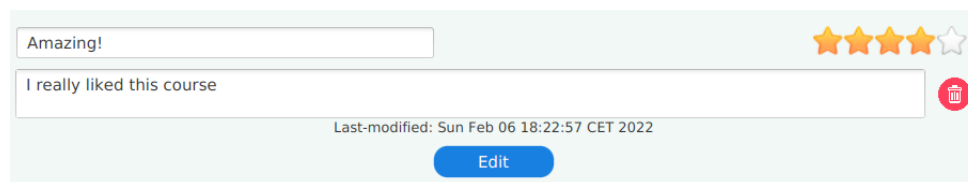
Figure 5.17: Personal review about a course

If the user has not written a review, they can do it easily by clicking on the rating stars and by optionally inserting a title and a comment and saving the information.

Each user can see all the reviews about the course, right after the section about their review. By clicking on the profile image of the author of a review, the user can visit their *Profile Page*.

Figure 5.18: New review about a course

Figure 5.19: A review snap-shot

5.6 Visit a Profile Page

5.6.1 Personal profile page

From the *Personal profile page*, a user can see all the information that are also shown in the *Random user's profile page*, and you can have access to the *Personal Page* from which you can see the complete information about your user and edit them.

Figure 5.20: Profile page of the logged user

5.6.2   Random user's profile page

From a random user's profile page, a user can follow or unfollow the user by clicking on the follow/unfollow button or (s)he can view the information about the user, the courses (s)he completed, the followers and the following users of the user.
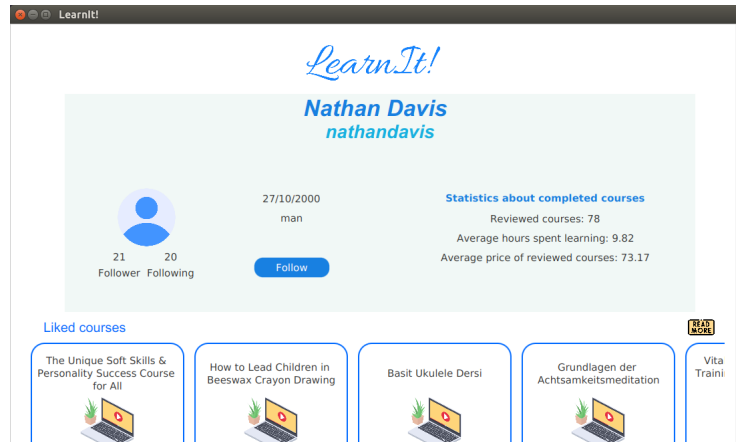
Figure 5.21: Profile page of a user

5.7 Administrator functionalities

An administrator can have access to all the features reserved to administrators only, if (s)he logs in to *LearnIt!* using their administrator's credentials (which are different from their credentials as standard users).

5.7.1 Delete a user

An administrator can delete a user by visiting the user's page and then pressing the trashcan button.
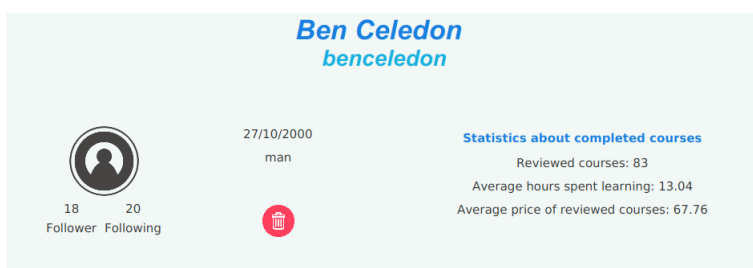
Figure 5.22: Delete user on user's profile page

5.7.2 Delete a review

An administrator can delete a review by visiting the course's page and then pressing the trashcan button relative to the review.
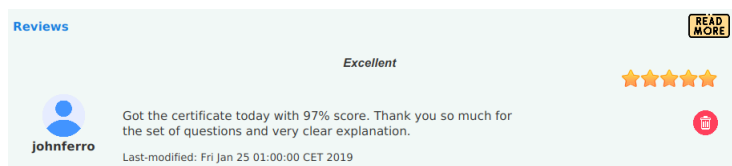
Figure 5.23: Delete review on course's page

5.7.3 Delete a course

An administrator can delete a course by visiting the course's page and then pressing the trashcan button relative to the course.

5.7.4 Create a new admin

To create a new admin, an administrator has to go to the *Discovery Page* and then to press the create new admin button. The admin will be redirected to a
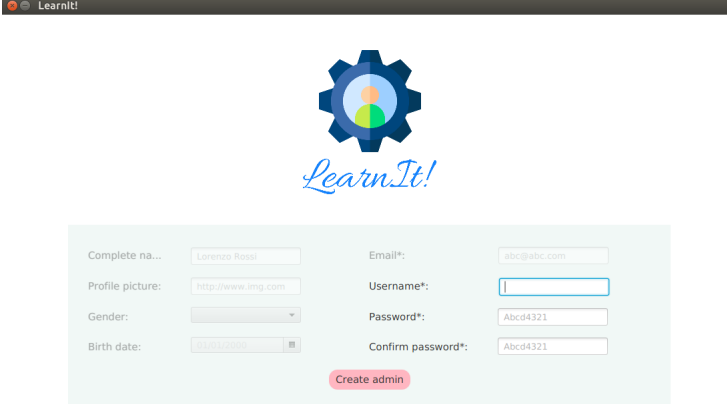
Figure 5.24: Delete course on course's page

new page where (s)he should type the new admin's credentials and press save.

Figure 5.25: Create new admin button on *Discovery Page*

Figure 5.26: Create new admin page

5.8 Logout

From the *Discovery Page*, a logged user can click on the logout button: they will be redirected to the *Login Page* and they will be asked to login again to *LearnIt!*.
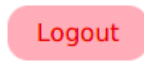

Figure 5.27: Logout button on *Discovery Page*