Dario LORENZONI

Raffele NOCERINO

Marco PETTORALI

Riccardo XEFRAJ

NEWSCRAPER

Task 2

Index

[Description of the system 3](#_Toc37849949)

[System requirements 4](#_Toc37849950)

[Functional requirements 4](#_Toc37849951)

[Non-functional requirements 4](#_Toc37849952)

[Main actors 5](#_Toc37849953)

[Use Cases Diagram 5](#_Toc37849954)

[Analysis Class Diagram 7](#_Toc37849955)

[Choosing the database 8](#_Toc37849956)

[Data modelling 9](#_Toc37849957)

[Application’s architecture diagram 12](#_Toc37849958)

[Newscraper protocol 13](#_Toc37849959)

[User side 13](#_Toc37849960)

[Administrator side 14](#_Toc37849961)

[Server components 15](#_Toc37849962)

[Scraping 15](#_Toc37849963)

[Scraper class 15](#_Toc37849964)

[Scraping activity 15](#_Toc37849965)

[Twitter’s article links retrieval 16](#_Toc37849966)

[Retrieving information from websites 16](#_Toc37849967)

[Workflow of scraping process 17](#_Toc37849968)

[Text Analyzer 18](#_Toc37849969)

[keywordAnalysis() method 18](#_Toc37849970)

[loadTintPipeline() method 19](#_Toc37849971)

[loadAuxiliaryWords() method 19](#_Toc37849972)

[getKeywords() method 19](#_Toc37849973)

[normalizeAndFilter() method 19](#_Toc37849974)

[countWordsOccurrences() method 19](#_Toc37849975)

[sortMapByValue() method 20](#_Toc37849976)

[MongoDBManager 21](#_Toc37849977)

[Indexes performances 21](#_Toc37849978)

[Aggregate queries 25](#_Toc37849979)

[Server classes 27](#_Toc37849980)

[ServerMain Class 27](#_Toc37849981)

[ServerRequestListener Class 27](#_Toc37849982)

[ServerWorker Class 28](#_Toc37849983)

[ServerAsynchronousWorker Class 28](#_Toc37849984)

[Classes 29](#_Toc37849985)

[Client’s classes 29](#_Toc37849986)

[Server’s classes 30](#_Toc37849987)

[Messages classes 32](#_Toc37849988)

[UML Diagram 34](#_Toc37849989)

[Server UML Diagram 34](#_Toc37849990)

[Client UML Diagram 35](#_Toc37849991)

[User manual 36](#_Toc37849992)

[Login 36](#_Toc37849993)

[Sign-in 37](#_Toc37849994)

[Working with the application 38](#_Toc37849995)

## Description of the system

The main purpose of this application is to provide to the users an overview of the trending topics using statistic reports based on the articles published by the most famous Italian online newspapers.

After the login, users can watch which are the most used words among the articles written in the last few days; words are collected in a ranking ordered by their importance (that depends on how many times a certain word appears in each article and on how many articles talk about that specific word).

Users can also watch the statistics of the trending keywords in a pie chart representing the most important trending keywords.

The system also provides a search engine for retrieving articles, in which the user can perform queries based on a specific keyword and on some filters (category of the article, author, newspaper from which it comes, geographical region related to the article).

The system keeps track of the searches made by the user and, using that history of searches, shows him/her a customized list of suggested articles.

An administrator can manage the users signed into the service, setting the rate at which articles are collected and analyzed and can force an asynchronous scraping of the articles stored into the database.

## System requirements

### Functional requirements

* The user can sign in to the service filling a form;
* The user can login to the service after he/she signed in;
* The system shows the list of the trending keywords, ordered by their importance;
* The system shows a pie chart which represents the most important keywords;
* The system provides a search engine to query the database of the articles by keyword;
* Queries can involve the following filters: category of the articles, author, online newspaper, geographical region of the articles;
* When the user moves the cursor over an article, the system shows the main keywords of the article and their occurrences;
* If the user clicks on an article, the system opens the link of the article into the browser;
* The system shows to the user a customized list of suggested articles, based on its last searches;
* The administrator can login to the service;
* The administrator can view the list of all users;
* The administrator can delete a user from the system;
* The administrator can set the period at which articles are download from the web and analyzed;
* The administrator can force in any moment a collection of new articles from the web and the subsequent calculation of the statistics;

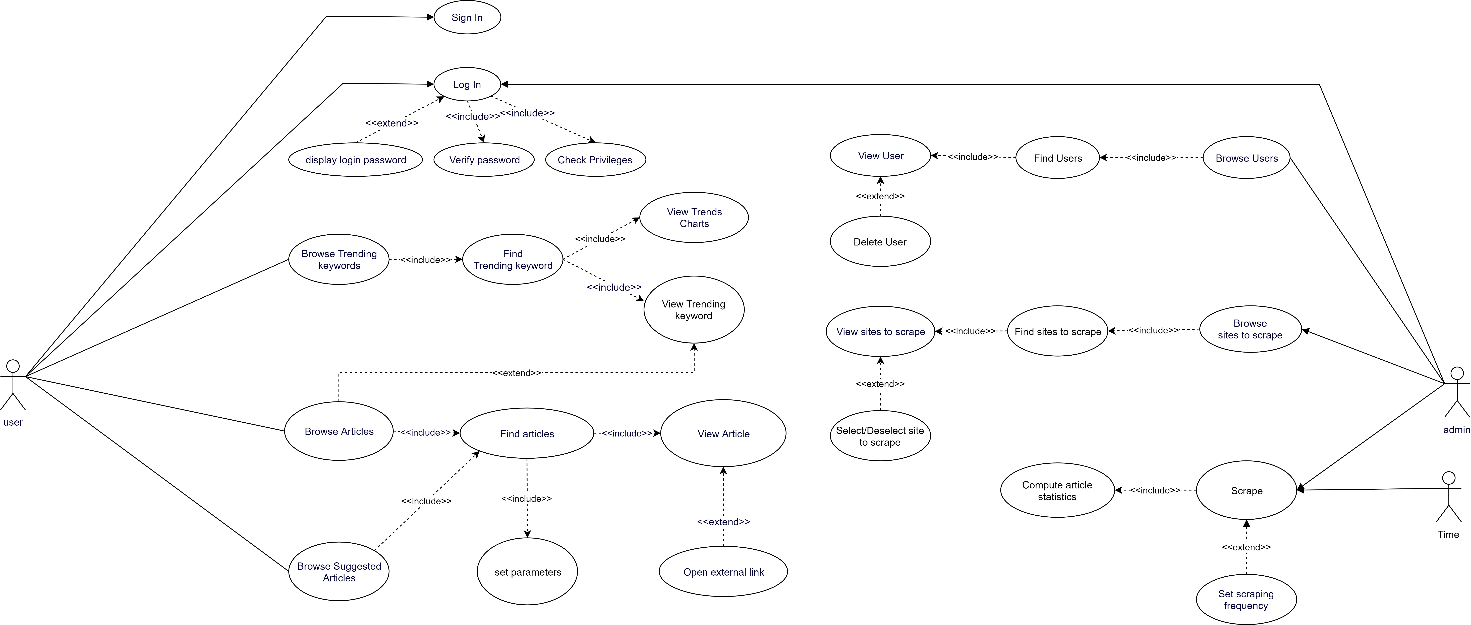
### Non-functional requirements

* The service must be reachable by the users at every time, independently of the state of the data to which he/she has access;
* Collection of articles from the online newspapers must not be invasive and cause them no harm;
* Users must experience a fast response of the system when they make queries in the search engine;
* Every time an error occurs, the system shows an error message with a short description of the error;

## Main actors

* User
* Administrator
* Time

## Use Cases Diagram



Only Users can sign in as users, system admins are already present in the system.

Both users and admins can login to the system. The login procedure includes the verification of the password and the privilegies check, in order to determine which panel (normal user or administrator one) to show to the one that is logging in.

The list of trending keywords and a chart representing them are shown to the user. If he/she wants, the user can click on a keyword in the Trending Keywords table: the system will perform a search operation using that keyword. The resulting articles are stored in the Articles table.

The user can perform a search operation using the search engine provided by the system; some parameters, such as the keyword (mandatory) and other already mentioned filters, are provided to the search engine. The result of this operation is the Articles table filled with the articles found after any search operation.

Some suggested articles are shown to the user. To collect them, the system calculates the suggested keywords for the logged in user and performs a search operation for each of them. As always, for any operation involving the search engine, resulting articles are stored in the Articles table.

The administrator can view every user signed in in the system and their information. If he wants, he can also select one of them from the Users table in order to delete it from the system.

The administrator can also view and select/deselect the sites that he wants the system to scrape.

The administrator can force a scraping round, after the which some statistics on the collected articles are calculated, such as the keywords analysis. If he/she wants, the admin can also set a different scraping frequency, rather than using the default one.

When the scraping period has elapsed from the last scraping activity, another scraping round is performed autonomously by the system.

## Analysis Class Diagram

Immagine che contiene screenshot

Descrizione generata automaticamente

The information that are needed to be collected are:

* The users in the system;
* The articles collected by the system;
* The keyword analysis of each article in the database;
* Which articles users have decided to view;
* Which filters users have applied to their searches.

In particular, each user can perform an unlimited number of article views, but each View belongs to a single user only.

Views refer to a single article, but an article can be seen by many users, and so it can have many views that refer to it, or even no one if the article was not seen by anyone.

A View is certainly preceded by a search operation: each View has a linked Filter combination. Each Filter combination can be used by many users but at least one, since that only filter combinations that are used at least once are stored.

View and Filter objects are stored in the system in order to compute the suggested articles for each user in the system.

## Choosing the database

Our requirements and data model show that data handled by the system has a loose structure. For instance, Filters object can contain one or more elements, some of them are mandatory (keyword), others are optional (e.g. newspaper, author). Another example is the class Article, in which some fields may miss (e.g. the geographical area or the category of the article), since not every online newspaper from the which the system collects articles provides them.

A traditional SQL-like scheme would not reflect the nature of our data, so we used a NoSQL database manager: we decided to use a document database (MongoDB).

In fact, document databases provide a high flexibility since they are scheme-less, but also provide the possibility to perform complex queries supported by indexes, necessary for getting a considerable boost up in the operations of article retrieve (for instance, when we calculate the suggested articles).

Another feature provided by document databases is the possibility to embed documents (and therefore, objects) inside others: in fact, as we will see in the following chapter, it makes no sense to implement Filter objects outside View documents by reserving an entire collection just to store all the permutations of the filters. Filters will be nested inside View documents. The same applies for Article and their Keyword Analysis: since the cardinality that lies in between these two objects is “One-to-One”, we will put the Keyword Analysis document inside the related Article one.

## Data modelling

In our database, called Article, we have created three collections of documents:

* **Articles**: in this collection, we have stored the list of all the articles scraped from the four most important Italian newspapers’ website.

The structure of each article’s document is:

* + Title: representing the title of the article;
  + Link: the link of the article on the website from the which the article was taken
  + Topic: category of the article (only for the sites that provides it)
  + Newspaper: newspaper from which the article was taken
  + Text: representing the content of the article
  + Date: represents the date of the article provided by the sites
  + Keywords: array containing the keyword analysis of the article. It contains for each word of the article its related number of occurrences.

Here is an example of a document in the Articles collection:

Immagine che contiene screenshot

Descrizione generata automaticamente

* **Search**: it is used to implement the functionality of recommended articles for each user. Every time a user clicks on an article, the application stores into the database some information that will be used in future to compute the recommended articles for the user.

A document in the Search collection includes the following fields:

* + userId: the username of the user that has viewed the article;
  + linkArticle: the link of the article viewed by the user;
  + dateRead: the date in which the article was read by the user;
  + filters: a nested document which contains all the filters used by the user in order to perform the search related to the viewed article.

Here is an example of a document contained in the Search collection:

Immagine che contiene screenshot

Descrizione generata automaticamente

* **User:** this collection is used to store all the information about the users registered in the system.

A document in the User collection includes the following fields:

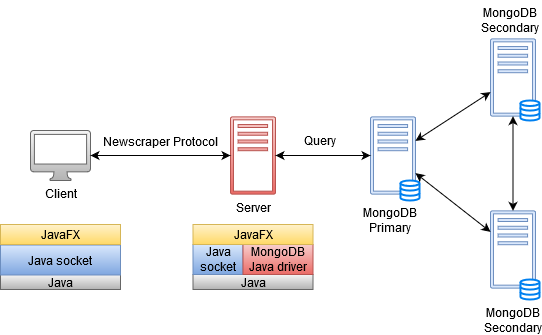
* + userId: the username of the user that has viewed the article;
  + firstName: it represents the name of the user;
  + lastName: it represents the surname of the user;
  + dateOfBirth: it represents the date of birth of the user;
  + email: it represents the email of the user;
  + password: it contains the password of the user;
  + adminStatus: it’s a Boolean value containing ‘true’ if the user is an admin.

Here is an example of a document in the User collection:

Immagine che contiene screenshot

Descrizione generata automaticamente

## Application’s architecture diagram



Our application is based on the Client-Server paradigm. On the client-side we have the GUI used by the users to perform a search, view recommended or result articles, view trending keywords or the article’s text analysis.

All the information requested by the user are computed within a server. In particular the server does periodically a scrape of the four most important newspaper’s website, storing into the db all the collected articles.

After that the server computes for each saved article its related text analysis, which stand for calculating for each word of the article its number of occurrences.

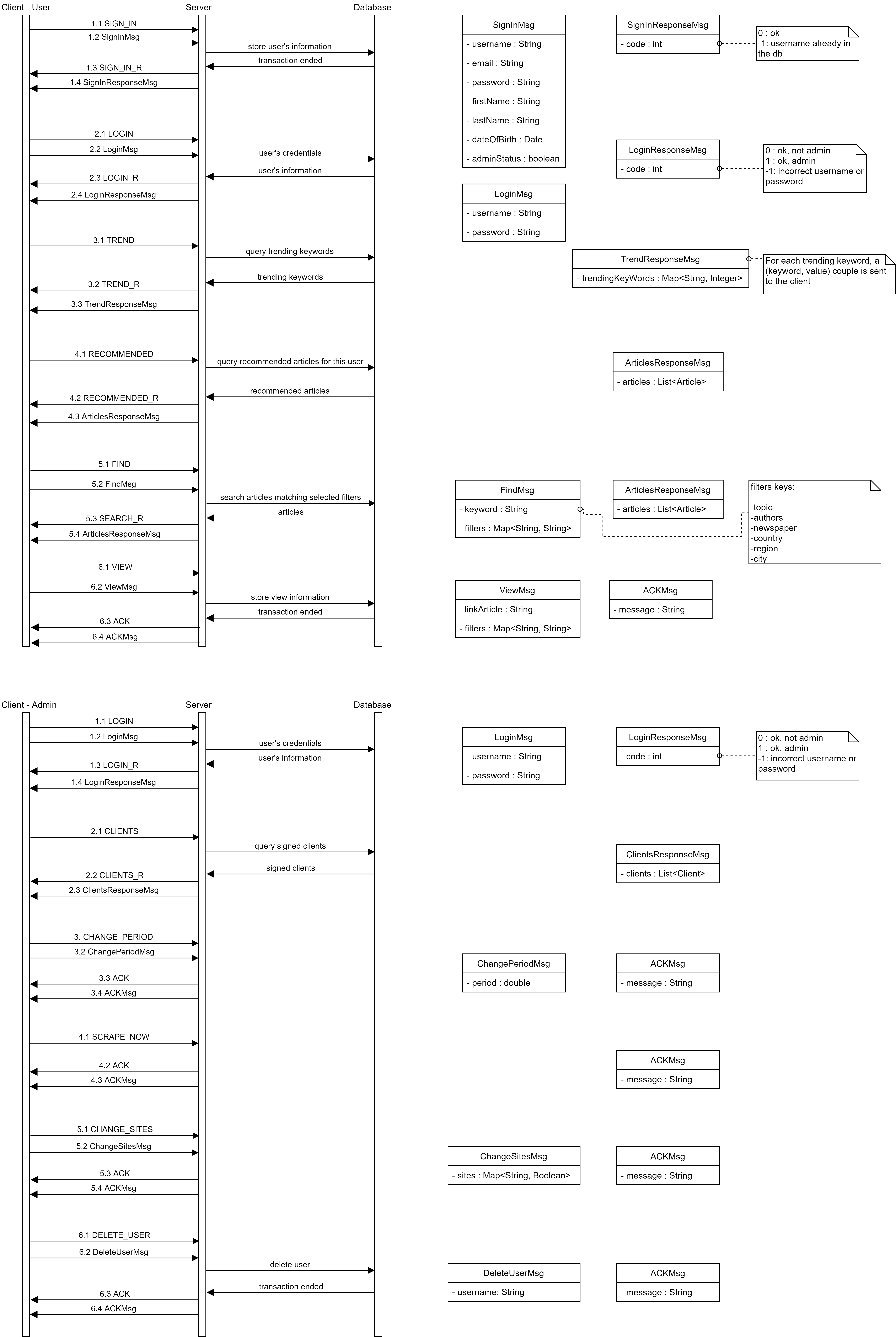
Moreover the server communicates with the clients receiving some requests. The server receives the request of recommended articles, the request of trending keywords but also a search request. Once the server has received a request, elaborates a response and using a socket send the answer to the client.

To guaranty high availability of our database a replica set has been created.

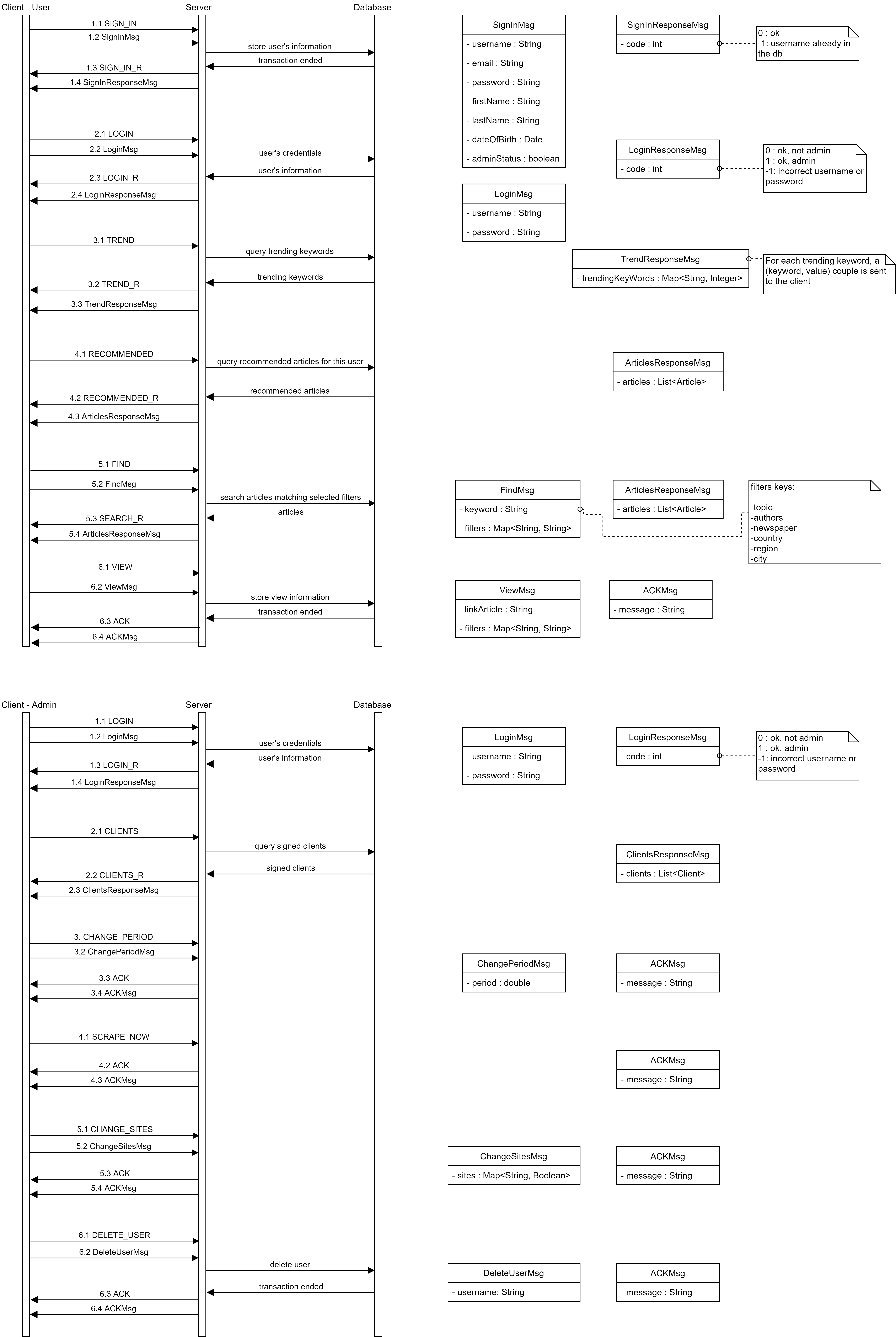
## Newscraper protocol

In order to make client and server communicate, we designed a specific protocol. Here is a UML diagram of it:

### User side



### Administrator side



## Server components

We can summarize the four parts of which the server application consists:

* **Scraper**: the component of the server used to collect articles from the 4 most important Italian newspaper websites.
* **TextAnalyzer**: the component of the server which computes for each article its text analysis.
* **MongoDBManager**: the component of the server used to store the information into the document database.
* **ServerWorker**: the component of the server used to handle the communication with the clients.

In the following pages, we will analyse in detail each of these components.

## Scraping

### Scraper class

The Scraper class contains all the method useful in order to perform the scraping activity. In particular the server can call the main method *scrape()* for the purpose of start a new scraping session.

Our aim is to create first and then update a Database that store the information about the article published from the major Italian newspaper web-site (la Repubblica, il Corriere, Sky TG 24 and ANSA, even if the code was arranged in order to make adding new newspaper website easy). This is made in a structured way, distinguishing the authors, the title, the text, and etcetera.

### Scraping activity

The scraping activity can be divided in two separated parts: retrieving the link of the new article and analyze the content of the article found.

### Twitter’s article links retrieval

In order to perform this first part, we choose to retrieve the link through Twitter for several reason the main ones are:

* The tweet’s structure is fixed, while the homepage of the newspaper websites are a bit messy and of course to analyze a fixed structure in order to fine a string that match your requirement is easier.
* Every tweet has an associated id, so it’s also easier search the find the link of the new article pushing aside the other ones.
* Twitter accounts of the major newspapers are constantly updated
* Finding the article link through Twitter, and more in particular through its specific API, reduces the request to the newspaper websites, and this is a good news because we have always the risk of being banned form the website because of an high number of request!

The already mentioned used API is *Twitter4j*;in particular the main class used is *Status* that represent one single status of a user.

Every time that a new scraping session start a new instance of *Twitter* class is created by the method *getTwitterFactory()* and for every newspapers’ Twitter account we try to retrieve the id associated to the last Tweet scraped (that is stored in a file called lastTweetId). If the file doesn’t exist, or we don’t have information about the account to scrape, we chose to take into consideration the last 1000 tweets posted by the user. Otherwise, if we have information about the last tweet scraped, we retrieve all the tweets up to this one.

After retrieving the new tweet, we store the id of the more recent tweet of each Newspaper in the file lastTweetId in order to guarantee that an article isn’t analyzed twice. At the end we have some tweets from which using the method *getArticleLink()* we extract the link of the article that eventually are related to that tweets.

### Retrieving information from websites

As regard the article analysis the code, organized in methods that are called using the convention *scrape+NewspaperName(),* could become messy because of the various possible structure of the web page. In order to retrieve much information as possible, during the design phase we used a log file that stores the part of the Article that our program wasn’t able to find with the associated link.

In order to perform a fair scrape, but trying to reduce as much as possible the time required at the same time, we decided to send a request to each website every 30 seconds.

### Workflow of scraping process

The behavior of the program is the following: assuming that we are analyzing 4 newspaper, and we have at least one article that is available and is waiting to be analyzed for each newspaper, we begin finding a link of an article associated to the newspaper number 1. Then, we make a request to that web page and we analyze it. We repeat these operations for each newspaper, and finally the program stops and wait for 30 seconds.

This operation is repeated while there are no more article left.

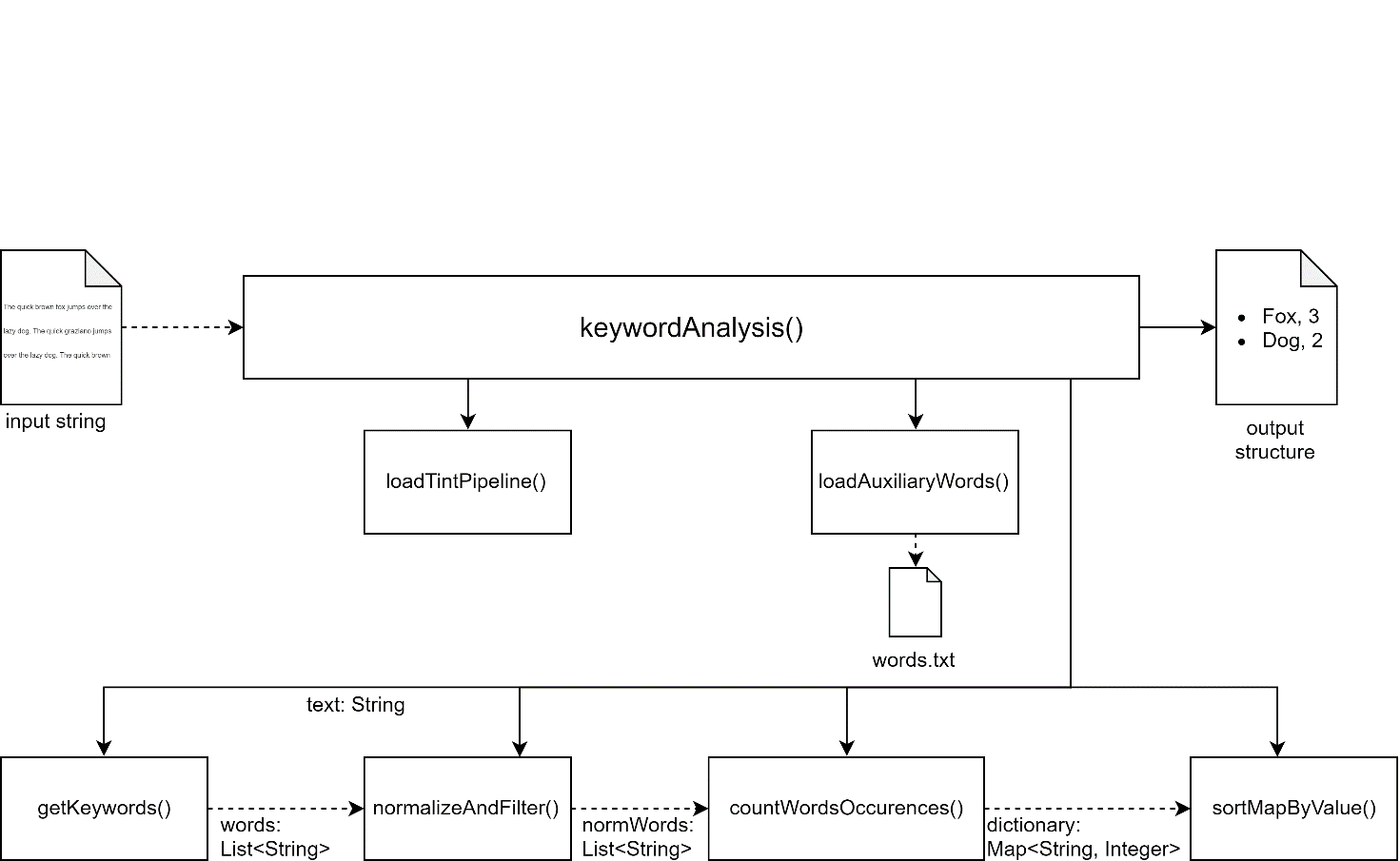
## Text Analyzer

TextAnalyzer class contains the methods for analyzing a text and extracting its keywords and their number of occurrences within the text. Keywords are proper or common nouns, and adjectives: each verb, adverb and other parts of the speech are not considered, since they don’t define the content of an article.

This class contains only one public static method, that is keywordAnalysis(). It takes a string as a parameter, and it will call some other functions to return the list of all the keywords, together with their occurrences, sorted by number of occurrences.

TextAnalyzer exploits a NLP (Natural Language Processing) framework, called Tint (The Italian NLP Tool), based on Stanford CoreNLP. This framework provides useful methods for the recognition of the role of a word within a sentence, and it’s designed for the Italian language.

A scheme representing the function calls for keywordAnalysis() is the following:



### keywordAnalysis() method

This is the main, public static method of the class: it has the task to call the other methods in order to deliver the keyword analysis of the text that it’s passed to it as a string parameter. The output of this method is a map between string and integer: for each keyword found in the article, its number of occurrences is given.

### loadTintPipeline() method

This method creates a new TintPipeline and prepare it in order to prepare the Tint tools for the text analysis.

### loadAuxiliaryWords() method

loadAuxirialyWords() loads the content of “words.txt” file, that is the file containing all the words that must not be considered as keywords. This exclusion regards all the nouns with a non-specific meaning, or very trivial one. In fact, these words would pass the filter due to the Tint analysis, since they’re mere nouns.

### getKeywords() method

This method deletes all the special characters in the string passed as parameter (the text to analyze), because they sometimes create problems with the Tint pipeline. Then the text is fed to the Tint pipeline to get the so called ‘token annotation’: for each word, its grammar role is specified. The words that are not recognized as nouns or adjectives are deleted. Proper names of people or cities that are distributed in two or more words are glued together to get a unique word.

This method returns a list of string, each one representing a word (or a cluster of words in the case of proper names).

### normalizeAndFilter() method

This method takes the list of strings found after the getKeywords() call, and converts all the keywords to lowercase. It also removes the words that appear in the file “words.txt”. The remaining words are the keywords of the text to analyze.

### countWordsOccurrences() method

countWordsOccurrences() takes a list of normalized keywords, and returns a map between string and integer; this is the method that counts the number of occurrences for each word in the list.

### sortMapByValue() method

This method only sorts the map between string and integer returned by countWordsOccurrences() by the number of occurrences, in order to get a more usable version of the map.

## MongoDBManager

As said before, the database used in our application is MongoDB. We have used a document database in order to exploit some advantages, such as high flexibility and high-speed access to data, that are the most important characteristic that the database must have.

In fact, we have used the MongoDB indexes reducing the number of reads before obtaining the correct answer. This is the code about the creation of those indexes, that in this chapter will be explained:

 Indexes

public static void createIndexes() {

MongoCollection<Document> collection =database.getCollection("Article");

BasicDBObject obj = new BasicDBObject();

obj.put("Topic", 1); //

obj.put("Date", -1);

collection.createIndex(obj);

obj = new BasicDBObject();

obj.put("Keywords.keyword", 1);

obj.put("Date", -1);

collection.createIndex(obj);

collection = database.getCollection("Users");

obj = new BasicDBObject();

obj.put("userID", 1);

collection.createIndex(obj);

}

### Indexes performance

We analysed the read-heavy operations on the database, and see which one of them could get some advantage by using certain indexes.

#### User authentication

The user authentication is an operation done with an elevate frequency. Once a user is stored in our database, no more write operations are required for that user. So write operations in the User collections are very limited in number.

For what concerns read operations we assume that, on average, a user accesses to our service 2 times a day. Assuming 200 users are using our service every day, we will have 200 writes on the user collection and 200 users \* 2 times a day \* 30 days in a month = 12800 read operations per month.

Using MongoDB’s explain clause, we got some statistics of the execution time of the related queries with and without the indexes on the userID field of User documents:

This is the query used for our experiments. It implements the most expensive part of the login procedure, that is the search of the username in the database:

db.Users.find({"userID":"RiccardoXe"}).explain("executionStats")

These are the results obtained without indexes:

"executionStats" : {

"executionSuccess" : true,

"nReturned" : 1,

"executionTimeMillis" : 1,

"totalKeysExamined" : 0,

"totalDocsExamined" : 1602,

…

}

These are the results obtained after the implementation of the indexes:

"executionStats" : {

"executionSuccess" : true,

"nReturned" : 1,

"executionTimeMillis" : 0,

"totalKeysExamined" : 1,

"totalDocsExamined" : 1,

…

}

We can see that the introduction of indexes has really improved the performance of the query.

*Articles retrieve*

Usually users are interested in specific topics. If 200 users access to our service twice a day, with a high probability they will make at least 1 search using filters they are interested in.

If we assume four major trending topics, and 10/20 articles scraped per day for each of them, we have 20\*4 = 80 write per day in the Article collection referring to articles that will be accessed by many users. In fact we will have 200 users \* 2 accesses per day \* 1 search about one trending topic = 400 read operations per day. Obviously, this index is more useful in a scenario where an elevated number of users use our application every day.

This is the query used for this experiment:

db.Article.find({"Topic":"Serie A"}).explain("executionStats")

These are the results of the query without using indexes

"executionStats" : {

"executionSuccess" : true,

"nReturned" : 161,

"executionTimeMillis" : 21,

"totalKeysExamined" : 0,

"totalDocsExamined" : 22720,

These are the results of the query using indexes on Topic and Date fields (search is made by topic and search operations affect only the articles written in the last 7 days)

"executionStats" : {

"executionSuccess" : true,

"nReturned" : 161,

"executionTimeMillis" : 0,

"totalKeysExamined" : 161,

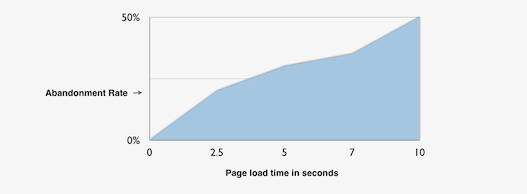
"totalDocsExamined" : 161,

In this case there is a considerable boost in performances, passing from 21ms of execution time per query without using indexes, to the less than 1ms when using indexes.

#### Keywords

The system provides a specific table where users can find the trending keywords in the period of the last seven days. When a user select one of the keywords, a search operation finds all the documents containing it. We assume that users are very likely to read articles that contain trending keywords every day.

Our main objective is to ensure fast response time to our users. If a user has to wait more than 2/3 second for a response is less prone to keep using our application.



Source: https://www.relentlesstechnology.com/wp-content/uploads/2012/02/site-speed-visitor-patience1.png

Assuming that 50 articles are written each day, and each one contain 10 keywords each, and 200 users are using our service every day, each one clicking on the top 5 trending keywords, we have 50 new articles \* 10 keywords per articles = 500 new keywords per day. We also have 200 users \*5 keywords = 1000 keywords search operation.

We used this query to measure some statistics about the queries involving keywords:

db.Article.find({"Keywords.keyword":"prestito"}).explain("executionStats")

These are the performances without using indexes:

"executionStats" : {

"executionSuccess" : true,

"nReturned" : 66,

"executionTimeMillis" : 279,

"totalKeysExamined" : 0,

"totalDocsExamined" : 22720

These are the performances using indexes

"executionStats" : {

"executionSuccess" : true,

"nReturned" : 66,

"executionTimeMillis" : 0,

"totalKeysExamined" : 66,

"totalDocsExamined" : 66,

Even in this case we have a huge impact of indexes in these family of queries.

### Aggregate queries

Using MongoDB, we still had the possibility of performing aggregate queries. In this way, the server is able to perform complex queries and providing interesting results to the users.

Let’s analyse now the aggregate queries performed by our application.

#### retrieveUsersInformation()

This simple aggregation query is used to retrieve the userID and the number of searches done for each user to allow the administrators to have an overview of the most active users

results = collection.aggregate(Arrays.asList(

new Document("$group", new Document("\_id","$userID")

.append("value", new Document("$sum", 1))),

new Document("$sort", new Document("value", -1))));

#### calculateTrendingKeyWords()

This aggregation query returns all the articles written from the queryDate to the current date. Keywords are then taken and grouped from each document.

For each keyword, the occurrences on every article are summed up and stored in "Occur" field. Moreover, it’s stored also the number of articles in which the keyword is present and saved in "NumberOfArticles".

This information are used to compute the formula for the trending keywords of the specified period. The trending keywords are the first 500 keywords ordered by value.

Value of the keyword = (Number of total occurrences) \* (Number of Article Containing Keyword)2

This formula gives more importance to the quantity of Article in which a specific keyword is present, than the total number of occurrences. This is needed to avoid to give importance to keywords present in fewer articles but with an elevated number of occurrences.

//ValueOfKeyword=NumberOfArticles^2\*Occ

results = collection.aggregate(Arrays.asList(

new Document("$match", new Document("date", new Document("$gt",

queryDate))),

new Document("$unwind", "$Keywords"),

new Document("$group", new Document("\_id", "$Keywords.keyword")

.append("Occur", new Document("$sum", "$Keywords.Occ"))

.append("NumberOfArticles", new Document("$sum", 1))),

new Document("$project",new Document("\_id",1)

.append("Value", new Document("$multiply",indexes) )),

new Document("$sort", new Document("Value", -1)),

new Document("$limit",500)));

#### suggestedArticles(User u)

This aggregation query retrieves the most used filters by a user in a specific period of time (from queryDate to the current date). The top three filters retrieved are used to find the suggested articles for a certain user.

results = collection.aggregate(Arrays.asList(

Aggregates.match(and(eq("userID",u.userID),gte("dateRead",queryDate))),

new Document("$group", new Document("\_id", "$filters")

.append("value", new Document("$sum", 1))),

new Document("$sort", new Document("value", -1)),

new Document("$limit", 3)));

# Replica set Details

MongoDB is more prone to consistency of the reads operation over the availability (when an application insert/query our database the operation is executed by default on the “primary” server). We managed the default settings of MongoDB to permit read operations on the secondary. It’s important to notice that mongoDB implements an asynchronous replication to secondaries and this implies that reads from secondaries may return data that does not reflect the state of the data on the primary.

1. private static final MongoClient mongoClient = MongoClients.create("mongodb://"
2. + "myUserAdmin:abc123@"
3. + "172.16.1.5:27017,"
4. + "172.16.1.7:27018,"
5. + "172.16.1.8:27019"
6. + "/?readPreference=primaryPreferred");

 By default, the read preference is set on “primary”. With “primary” all the operations reads from the replica set primary. Since we don’t want our users to wait to have a response in case of our primary is down we set the read preference on “primaryPreferred” : In most situations, operations read from the primary but if it is unavailable, operations read from secondary members. This is a good compromise between consistency and availability.

In windows to have access to our replica set add this string in the host file (/Windows/System32/drivers/etc)

1. 172.16.1.5 A
2. 172.16.1.7 B
3. 172.16.1.8 C

## Server classes

### ServerMain Class

ServerMain is the main class of the server side of the application. It has the task to initialize the main structures of the server, to launch the worker threads, and to call the back-end methods for the articles’ text analysis and for forcing a scrape round (collecting articles from all sites backwards in time until the last scraped article).

When the server is launched, mongoDB indexes are created first thing. Then the listener thread ServerRequestListener and the scraping thread ServerAsynchronousWorker are created (more details in the following paragraphs).

It contains the scrapingPeriod variable, which determines the cool down period for the scraping process. It’s accessed in a thread-safe way, so that its modifications due to admin requests are concurrent with its readings.

trendingKeywords is a structure representing the top 10 trending keywords in the reference period and their frequency. It’s updated after each scraping round, and sent to each user that logs in to build the trending keyword chart.

When an admin wants to force a scraping round, the scrapeNow() method is called. It at first checks that no scraping threads are already running; if it is, this method returns with an error code. If no scraping threads are active, this method creates a ServerAsynchronousWorker object, and calls its round() method: this will start a round of scraping.

Moreover, each time a scraping round is completed, the articleTextAnalysis() method is called, which invokes the back-end methods for the retrieval of stored articles and the methods for text analysis collected in TextAnalyzer class.

### ServerRequestListener Class

ServerRequestListener is the core of the multi-threaded server architecture. It implements the main thread for the communications with the clients, creating a socket on which requests to the server are listened. Each time a new connection is accepted, a new thread (ServerWorker) is created.

### ServerWorker Class

ServerWorker is a thread communicating with a specific user. It collects all the methods for the intercommunication with the client’s socket.

Each time a connected user sends a request to the server, his correlated thread receives it and, according to the command, calls a specific method of ServerWorker class to handle it.

This class contains a method for each action provided for by the communication protocol; these methods are not represented in the scheme above, because we opted for a compact version of this diagram to make it more readable.

### ServerAsynchronousWorker Class

ServerAsynchronousWorker is a thread that periodically works for updating the database of the system. In particular, each ServerMain.scrapingPeriod seconds, it does a scraping round (collecting all the articles until the last one seen during last round), it calls the methods for updating the articles present in the database by applying them their text analysis, and it finally invokes the method for calculating the new trending keywords.

Since a scraping round can be forced by the admin, only one scraping process must be done at a time: this implies the presence of a semaphore (‘working’ boolean variable) that is set to true each time the round() function is called, and it is set to false when the round is completed. Before invoking this function, this variable is checked: if ‘working’ is true, then the round function returns immediately.

Accessing to working variable is thread-safe, due to the presence of a synchronization on the same variable; this is important to avoid situations in which two processes attempt to change the value of this variable.

## Classes

### Client’s classes

|  |  |  |
| --- | --- | --- |
|  | |  |
| Front-end | **AdminPaneGUI** | This class implements the admin control panel in which the administrator can manage the settings of the application. |
| **SignupPaneGUI** | This class implements the panel used by the users to register themselves to the application. |
| **LoginPaneGUI** | This class implements the panel used to implement the login procedure. |
| **MainPaneGUI** | This class implements the main panel for a normal user. Here the user can perform a search, view trending keywords and recommended articles. |
| **ArticlesOverviewTable** | This class implements the table which contains the Text Analysis of the related article. |
| **ArticlesTable** | This class implements the table which contains a list of articles. (recommended or result articles). |
| **SingleWordAnalysis** | This class represents a single result of an article’s Text Analysis. Contains a word and its related number of occurrences. |
| **Trend** | This class represents the basic element of TrendingKeywords table. Contains a trending keyword and its related Percentage of usage. |
|  | **TrendingKeywordsTable** | This class implements the table used to show the list of the most important trending keywords computed by the application. |
| **UserOverviewTable** | This class implements the table showed into the admin control panel containing all the user’s information. |
| Middleware | **ConnectionToServer** | This class is used to implements the communication, using sockets, with the server. |
| **MainClass** | This class represent the main class of the application. Contains the main method and the method for the initialization of the application. |
| **MessageReceiver** | This class represents a thread that is used to receive messages from the server. In this way the main thread (GUI) is not blocked. |

### Server’s classes

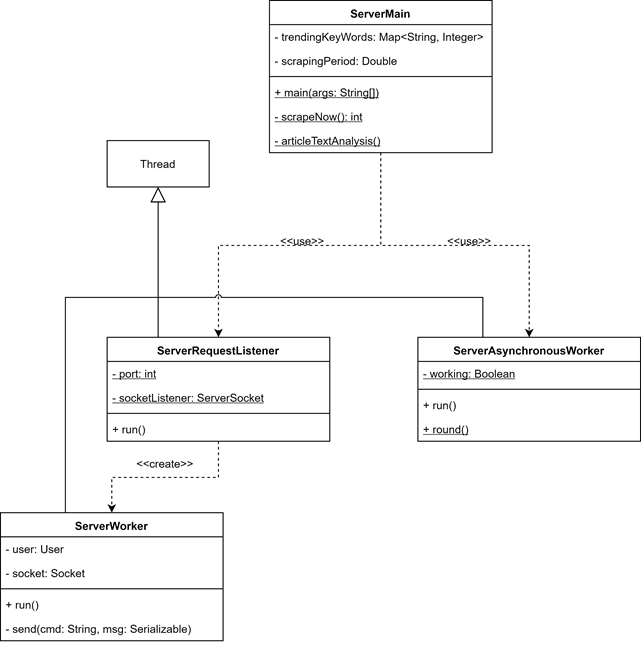
|  |  |  |
| --- | --- | --- |
|  | |  |
| MiddleWare | **Scraper** | It contains all the method useful in order to perform the scraping activity. In particular the server can call the main method *scrape()* for the purpose of start a new scraping session. |
| **ServerAsynchronousWorker** | Implements the main thread for the communications with the clients, creating a socket on which requests to the server are listened. |
| **ServerMain** | ServerMain is the main class of the server side of the application. It initializes the main structures of the server, to launch the worker threads and does the scrape periodically. |
|  | **ServerWorker** | ServerWorker is a thread communicating with a specific user. It collects all the methods for the intercommunication with the client’s socket. |
| **TextAnalyzer** | That class contains the methods for analyzing a text and extracting its keywords and their number of occurrences within the text. |
| Back-end | **Article** | That class represents an article with its related information. Is used to store the information about the articles on the database. |
| **Filters** | This class represents a possible filter object that can be created by a user when a search is performed. The filter object contains some properties corresponding to some fields that the user can specify ( like keyword,City,Topic..) |
| **MongoDBManager** | That class contains all the method used to communicate with the database and used to perform the queries. |
| **User** | That class represents an user with its relative attributes like username, date of birth , email and other private information. |
| **View** | That class represent a possible view that a user can do on a specific article. That object is used to keep trace of the article seen but also to keep trace of the filter used to find that article and the view date. |

### Messages classes

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | **ACKMsg** | That class implements the message sent by the server to the client in order to confirm the reception of the previous object. |
| **ArticleresponseMsg** | That class represents the message sent by the server to the clients containing a list of articles. That messages is sent after a search or at the beginning for the recommended articles. |
| **ChangePeriodMsg** | That class represents the message sent by the client to the server in order to communicate the changing of scraping period. |
| **ChangeSiteMsg** | That class represents the message sent by the client to the server in order to communicate the changing of scraping sites. |
| **ClientResponseMsg** | That class represents the message sent by the server to the client in order to communicate the list of users registered to the application. |
| **FindMsg** | That class implements the message sent by the client to the server when a user clicks on search button. That message contains the parameters of the search (keyword, filters). |
| **LoginMsg** | That class implements the message sent by the client to the server in order perform the login. In that message the client sent the username and password of the user. |
| **LoginResponseMsg** | That class represent the response to the loginMsg sent by the server to the client. That class contains a code which stands for the result of login operation. |
| **SignInMsg** | That class implements the message sent by the client to the server in order to perform the signIn phase. That message contains all the information specified by the user. |
|  | **SignInResponseMsg** | That class represent the response to the SignInMsg sent by the server to the client. That class contains a code which stands for the result of signIn operation. |
|  | **TrendResponseMsg** | That class implements the message sent by the server to the client at the opening of the application. That message is used to sent the list of all the trending keywords with their related percentage of importance. |
|  | **ViewMsg** | That class implements the message sent by the client to the server when a user clicks on an searched article. That message is used to sent to the server some information that will be used to compute recommended articles for the user. |

## UML Diagram

### Server UML Diagram



### Client UML Diagram

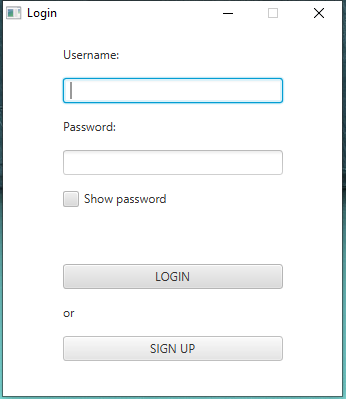
Immagine che contiene testo

Descrizione generata automaticamente

## User manual

### Login

When the application is launched, the system will display a login form. The user has to put in its username and password and then to click on “LOGIN” button. If desired, the user can also see its password as plain text by clicking on the “Show/Hide Password” radio button.



### Sign-in

If the user doesn’t have credentials for the application, he can register himself to the service pressing the button SIGN IN. He has to insert his First Name, Last Name, Date of Birth, email and the desired username and password. After pressing the button REGISTER, the system will show a message which will explain the result of the registration process.

Immagine che contiene screenshot

Descrizione generata automaticamente

### 

### Working with the application

#### Administrator panel

If the user is the administrator of the application, the main page showed will be the administrator control panel in which the admin can manage the setting of the application.

Immagine che contiene screenshot

Descrizione generata automaticamente

Here the admin can see the list of all the users registered and their related information.

Immagine che contiene screenshot

Descrizione generata automaticamente

The admin can also decide from which newspaper sites the application has to do the scraping of the articles. In particular he has to click on the checkbox of the desired websites and after that he has to click on the “APPLY” button.

Immagine che contiene screenshot

Descrizione generata automaticamente

The admin can manage also the scraping period of the application. He can specify in the “Scraping period” textfield the desired interval of time (in minutes ). After the decision of the period, he has to click on the “UPDATE” button.

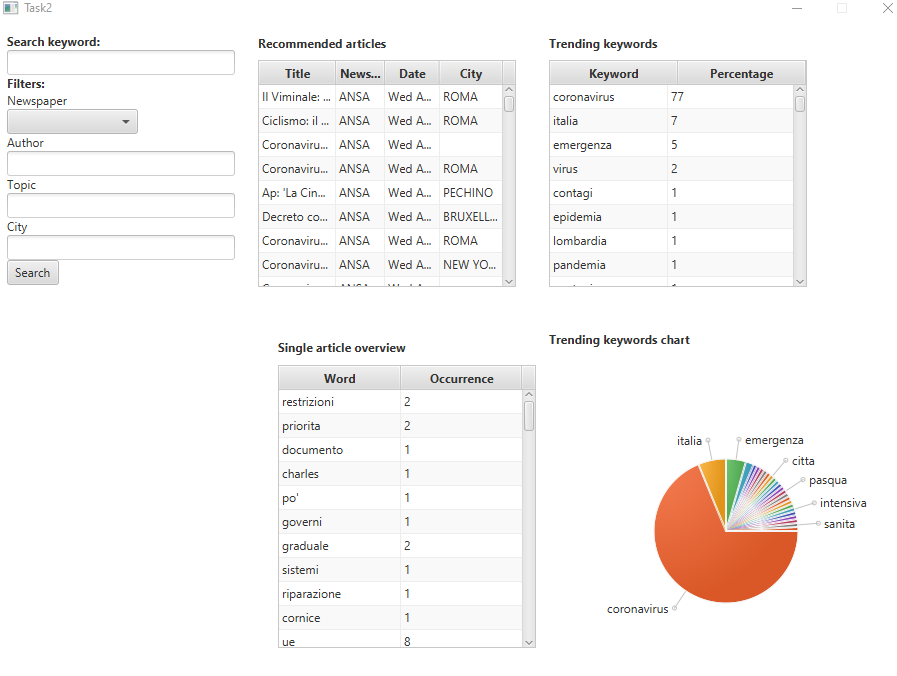
Instead of specifying a new scraping period, the admin can also force the scraping operation clicking on the “SCRAPE NOW” button. In this way the application receive the scraping command and starts to retrieve the articles from the specified sites.

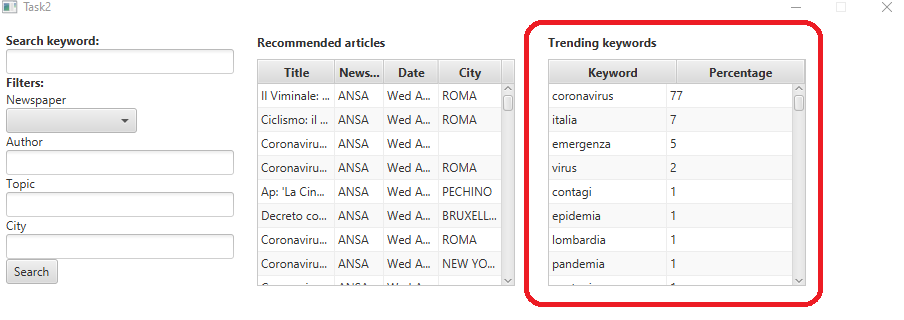
Immagine che contiene screenshot

Descrizione generata automaticamente

#### User panel

Once the application has loaded the main page, the user can visualize two main tables.



The first table contains the trending keywords of the last week. In particular the user can click on one of those keywords. After that, the application will insert the selected word into the “Search Keyword” textfield. The user can start the search of the articles related to the keyword, specifying or not some filters, clicking on the “SEARCH” button.

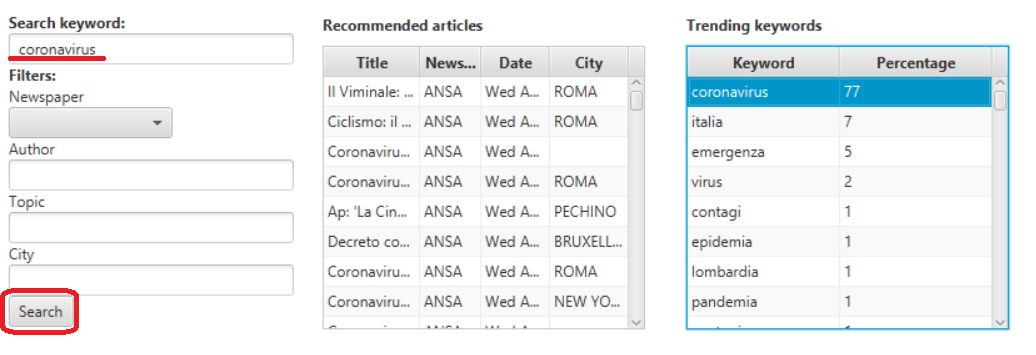


Immagine che contiene screenshot

Descrizione generata automaticamenteThe second table contains a list of recommended articles for the user. That list is based on the history of the user’s view. The user can scroll that list seeing the information about the articles. Moving the cursor on one of those articles, the user will see in the below table (Single Article overview) the article’s text analysis which contains the list of all the article’s word and their number of occurrence.

If the user clicks on a recommended article in the table, the application will open in the browser the newspaper web page containing the entire article.

The user has also the possibility to create his own search. In particular he can specify the desired keyword in the “Search keyword” field and specifying also some filters if he wants (like authors, City, Newspaper). After clicking the “Search” button the application will show the Result table which will contain the results of the user’s search.

The user can scroll that list seeing the information about the articles. Moving the cursor on one of those articles, the user will see in the below table (Single Article overview) the article’s text analysis which contains the list of all the article’s word and their number of occurrences. If the user clicks on an article in the Result table, the application will open in the browser the newspaper web page containing the entire article.

