UNIVERSITÀ DI PISA

MSc in Computer Engineering
Information Systems
A.A. 2019/2020

Workgroup Task 2

Designing and Implementing an Application
which interacts with a Document Database

Task Report

Chiara Bonsignori
Marco Del Gamba
Alessio Ercolani
Federico Garzelli

Contents

1 Design

1.1 Informal Requirements

The system implements a restaurant reviewing platform. Users can browse restaurants as well as pubs or steakhouses and filter them according to several parameters, such as type of cooking or position. Customers can log in to write a review of a restaurant they visited. A review consists of a score, a title and a short text. Users can read these reviews to know what other customers think about a restaurant to decide if it is worth visiting it.

Restaurant owners insert all the information needed to help customers to find their restaurants and let them know the type of food they cook. Owners can also reply to customers' reviews. Moreover, the application provides them with a dashboard showing statistics about their restaurants to help them examine customers' feedback and analyze the comparison of this evaluation with those of similar restaurants.

A restaurant owner sends a request when he wants to add a restaurant to the platform. This request is analyzed and then approved or denied by an administrator. If the restaurant is accepted it becomes accessible to users who can start reviewing it.

1.2 Actors and Functionalities

The application has three actors: customers, restaurant owners and administrators.

1.2.1 Functional Requirements

Each type of user, even if not logged in, shall be able to:

- search for restaurants by name in a given city
- search for best restaurants in a city
- search for best restaurants of a given category in a city
- search for restaurants of a given category in a city and with a given name
- visualize the information of a restaurant
- visualize the reviews of a restaurant
- visualize the distribution of the reviews of a restaurant

A **Customer** shall be able to:

- register to the platform
- log in and log out
- write and upload a review

- visualize her past reviews
- delete one of her reviews
- delete her account

A **Restaurant Owner** shall be able to:

- register to the platform
- log in and log out
- send a request to add a restaurant to the platform
- update the information of one of her restaurants
- delete one of her restaurant
- reply to a review of one of her restaurants
- visualize the trend of the reviews of one of her restaurants
- visualize a comparison of the reviews of a restaurant with respect to competitor's restaurants of the same category in the same city
- delete her account

An **Administrator** shall be able to:

- log in and log out
- delete a review
- delete a customer's account
- delete a restaurant owner's account
- visualize the list of requests to add a restaurant to the platform
- accept or refuse a pending request
- create a new administrator's account
- insert new restaurants to the application

1.2.2 Non-Functional Requirements

Availability The application shall be highly available to let users always be able to search and review restaurants.

Scalability and Performance Scalability is an important requirement of our application, which shall be able to handle a growing number of users. Scalability in our case is strictly related to performance, which shall be approximately constant regardless of the number of users connected to the platform at the same time. In particular, each user shall be affected by the same latency, which shall not exceed a certain threshold to guarantee a responsive experience of the application.

Security Customers, Restaurant Owners and Administrators shall be able to log in to the system. Each one of them shall have access to its own functionalities. Access to all the functionalities must be protected by the user log in screen that requires username and password.

Consistency Consistency is not one of the fundamental requirement of the system. Our application is indeed read-heavy and it is not necessary that all users have immediately the same consistent view of the recently written reviews. Hence, we can adopt an eventual consistent model to improve performance.

Recoverability The information hosted in the system are crucial so different backup-ups shall be present to avoid losing data in case of fault of part of the infrastructure.

Usability Users shall interact with the application using a graphical interface.

Portability The system shall be run on every Windows, Linux or MacOS platform having Java Runtime Environment 1.8.

1.3 UML Use Case Diagram

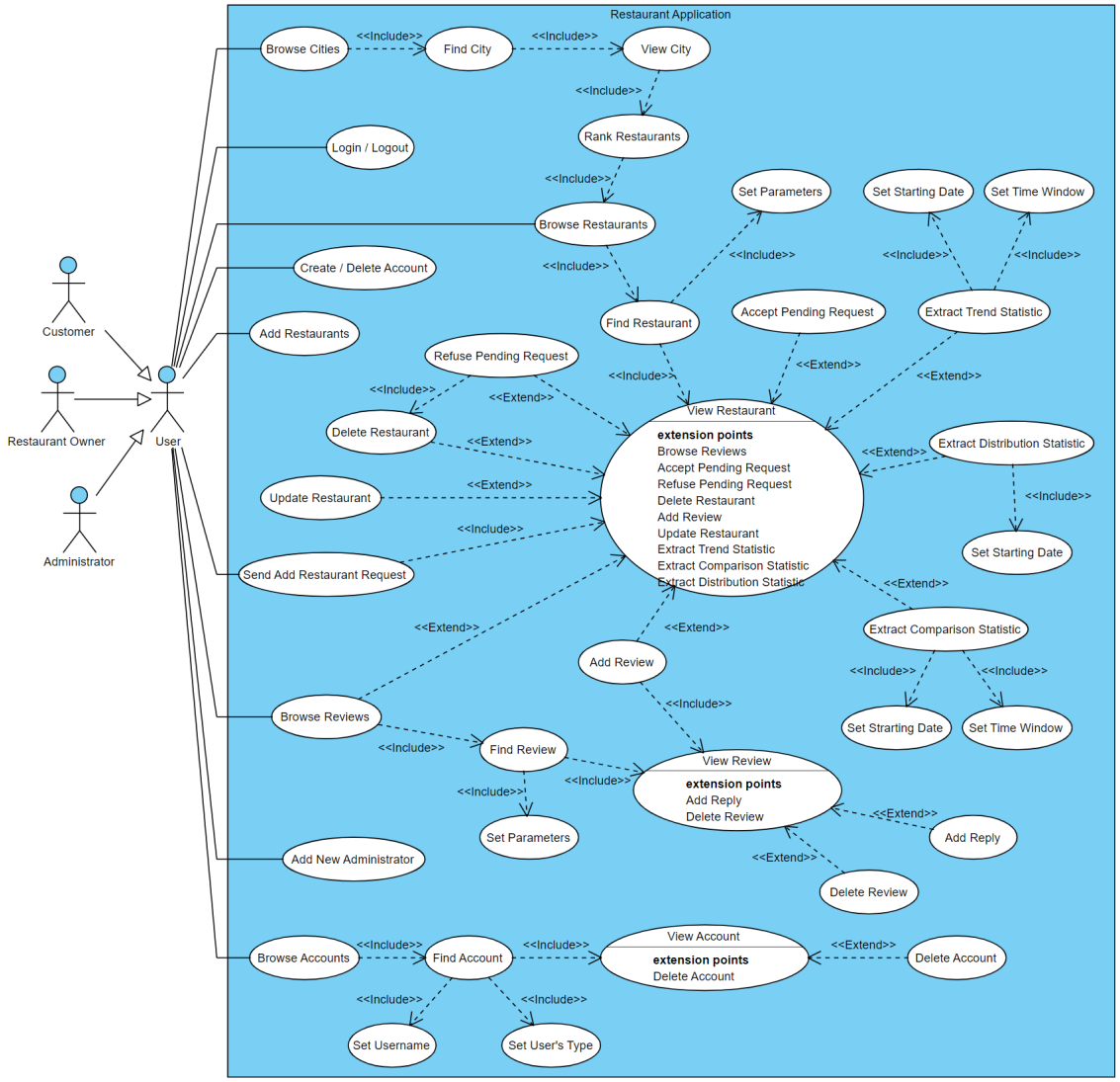


Figure 1.1: Use Case Diagram

Figure 1.1 shows the main use cases of the application. Three main actors are represented as specializations of a User actor. It is however necessary to underline that not all use cases can be performed by all actors. In particular:

- a Customer and a Restaurant Owner can delete their own account;
- an Administrator can delete a Customer or a Restaurant Owner's account searching it by username and user's type;
- only an Administrator can perform Accept Request, Refuse Request, Add New Administrator and Add Restaurants use cases;
- Add Review use case is performed by a Customer;
- Delete Review use case can be performed by a Customer to delete one of his previous inserted reviews or by an Administrator to delete an inappropriate review;

- Add Reply use case is executed by a Restaurant Owner;
- Send Add Restaurant Request, Update Restaurant, Delete Restaurant use cases can be performed by a Restaurant Owner.

Moreover, we need to highlight the following aspects for some use cases:

- Browse Restaurants use case
  - can be performed by a generic User to search restaurants in a given city, indeed is an extension of View City use case;
  - can be performed by a Restaurant Owner to visualize his own restaurants or by an Administrator to see restaurants that need to be approved, indeed it is also directly connected to the User actor.
- Browse Reviews use case
  - can be performed by a generic User who visualize reviews of a given restaurant, indeed is an extension of View Restaurant use case;
  - can be performed by a Customer who can visualize his own reviews and indeed it is also directly connected to the User actor.

1.4 UML Analysis Classes Diagram

Figure 1.2 shows the UML diagram of analysis classes of the application, which highlights the main components of the system and their relationships.

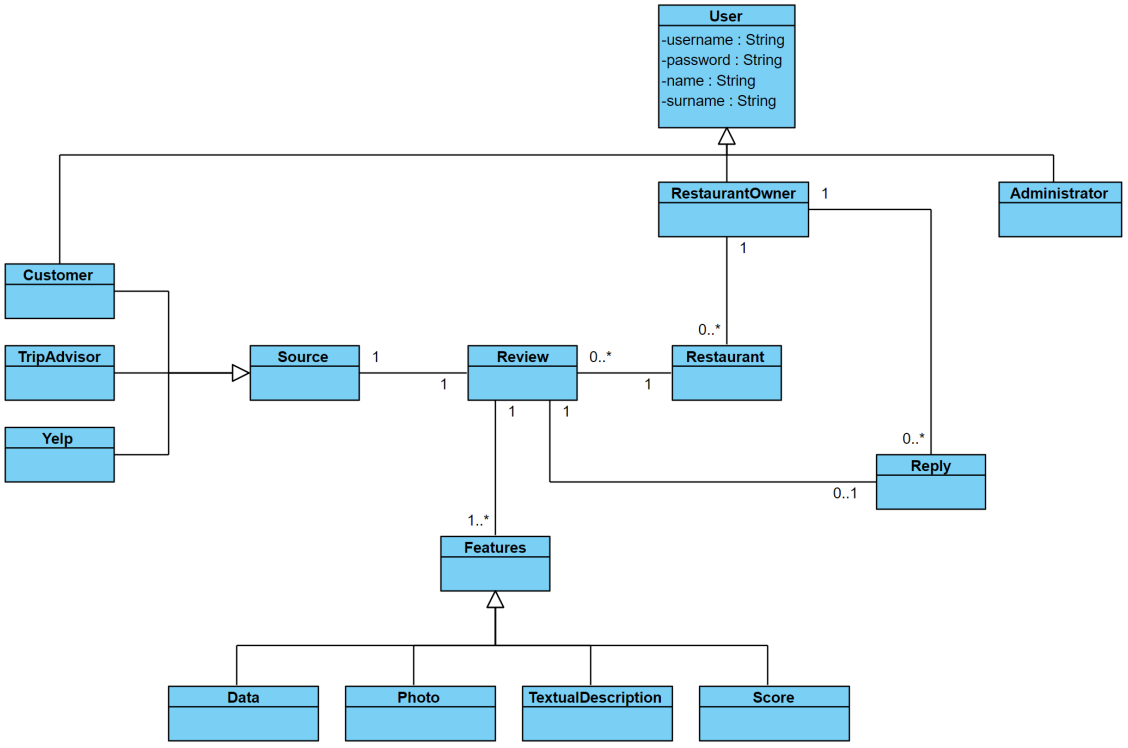


Figure 1.2: UML Analysis Classes Diagram

1.5 Data Model

Our data model is made up of three collections:

- `restaurants`: documents in this collection contain information about restaurants and requests to add restaurants to the reviewing platform; we decided to insert a field in each restaurant document that tells whether it has been already approved or not
- `reviews`: documents in this collection contain information about reviews of restaurants
- `users`: documents in this collection contain information about Customers, Restaurant Owners and Administrators

In the following we will describe how the relationships between the entities of our system are modelled in a document database. First of all, it is important to point out that the main operations of our application, namely the ones that are executed more frequently, are read operations. Therefore, having the same piece of information in more than one collection is beneficial to the performance, because we do not have to query different collections, but we can gather the information we need with just one read. Moreover, since write operations are relatively few, maintaining the information in many places is the price to pay to have such performance.

A restaurant can have many reviews, and they can be relatively large from the storage point of view. If we used embedded documents to model this relationship, restaurants with tens of thousands of reviews could easily reach the 16 MB document limit of MongoDB. However, showing the reviews of a restaurant is one of the main use cases of the application and exploiting embedded documents can be crucial. It is important to note that the flexibility of a document database let us store reviews in different forms. Therefore, we decided to embed in each restaurant document just the 10 most recent reviews in full and to keep only the score and the date of all the others. This does not affect the queries that deeply, since a user does not need to see the text of all the reviews all at once and usually he considers only the most recent ones. This solution also improves the performance of analytic queries, which do not need text, since they involve just scores and dates of the reviews. By storing these pieces of information in the `restaurants` collection, we do not need to access also the `reviews` collection when performing an analytic function.

Documents in the `reviews` collection must contain an identifier of the restaurant they refer to, as the name of the restaurant is not sufficient to identify it. In this way it is possible to retrieve the reviews of a given restaurant which are not embedded in it.

Reviews are embedded (in full) in the Customer document. Customers generally write just few reviews and embedding them allows to speed up the search for reviews of a given Customer, because there is no need to browse them all. Like restaurants, reviews are rarely updated or deleted.

A Restaurant Owner document embeds all Restaurants belonging to her. In this way, the search for an Owner's restaurants is fast. This implies that information about a restaurant is replicated in two documents: restaurant and Restaurant Owner. However, these restaurants contain the 10 most recent reviews in full, but not the others, not even

in a shortened form. Indeed, in this case, there is no need to keep old reviews embedded; the recent ones are instead kept to be shown to the Restaurant Owner as she loads the page of one of her restaurants.

A review may or may not contain a reply from a Restaurant Owner. This relationship is modeled by embedding the reply, if present, inside the review document.

Restaurant Owners can write many replies to different reviews of her restaurants. However, in our application, Owners will rarely want to access directly their replies, but rather the reviews of their restaurants. Therefore, in our data model there is no direct connection between a Restaurant Owner and her replies.

Table 1.1 summarizes the modelling of the relationship we discussed.

Relationship	**Model**
Restaurant - Review	Embed the 10 most recent ones in full, the others in shortened form + Parent reference
Customer - Review	Embedded document (all reviews in full)
Restaurant Owner - Restaurant	Embedded document (also embed the 10 most recent reviews in full inside the Restaurant)
Review - Reply	Embedded document
Restaurant Owner - Reply	-

Table 1.1: Model of Relationships

1.5.1 Structure of the Collections

In the following we present the structure of the collections, as implemented in MongoDB.

Restaurants An example of the typical document of the `restaurants` collection is shown below. The 10 most recent reviews are stored in full, whereas the others are kept in a shortened form.


```
{
  "_id": "5e0489dbaf73587cf7a1e12c",
  "name": "AD BRACERIA",
  "address": {
    "city": "Arezzo",
    "country": "Italia",
    "postcode": "52044",
    "street": "Via Nazionale 10"
  },
  "approved": true,
  "rating": 4.784845132743363,
  "numberOfReviews": 800,
  "phoneNumber": "+39 0575 638185",
  "openingHours": [
    {
```

```
      "days": "mar - dom",
      "times": [
        "18:00 - 23:00"
      ]
    },
    {
      "days": "lun",
      "times": [
        "09:00 - 23:00"
      ]
    }
  ],
  "priceRange": {
    "maxPrice": 35,
    "minPrice": 20
  },
  "categories": [
    "Italiana",
    "Pesce",
    "Mediterranea",
    "Barbecue",
    "Toscana",
    "Italiana (centro)"
  ],
  "options": [
    "Per vegetariani",
    "Opzioni senza glutine"
  ],
  "reviews": [
    {
      "_id": "5e0489dbaf73587cf7a1e12e",
      "date": "2011-01-16T23:00:00.000Z",
      "rating": 5
    },
    /* ... */
    {
      "_id": "5e0489e5af73587cf7a1e731",
      "date": "2019-11-02T23:00:00.000Z",
      "rating": 5
    },
    {
      "_id": "5e0489e5af73587cf7a1e733",
      "date": "2019-11-04T23:00:00.000Z",
      "rating": 5,
      "restaurant": "AD BRACERIA",
      "restaurantId": "5e0489dbaf73587cf7a1e12c",
      "text": "Cena in piena sintonia con il territorio e
         ottima materia prima, non solo di carne. Buona anche
         la pizza, ben lievitata e cotta. Ambiente
```

```
        accogliente con tavoli ben distribuiti. Gestori
        molto cordiali.",
      "title": "Carne e non solo...",
      "username": "RoccoMaT"
    },
    /* ... */
    {
      "_id": "5e0489e6af73587cf7a1e743",
      "date": "2019-12-12T23:00:00.000Z",
      "rating": 4,
      "restaurant": "AD BRACERIA",
      "restaurantId": "5e0489dbaf73587cf7a1e12c",
      "text": "Abbiamo cenato in questo ristorante domenica
        sera...molto carino, sembra di fare un salto nel
        passato, accogliente e con personale qualificato...
        non proprio economico, ma...location e cibo
        veramente al top!!! Bravi",
      "title": "Location incantevole",
      "username": "Luciana B"
    }
  ]
}
```


Reviews An example of the typical document of the `reviews` collection is shown below.


```
{
  "_id": "5e0489d9af73587cf7a1e00a",
  "date": "2016-12-04T23:00:00.000Z",
  "rating": 5,
  "restaurant": "1250 Ristorante Pizzeria",
  "restaurantId": "5e0489d8af73587cf7a1dffe",
  "text": "Rinnovato con molto gusto, accogliente, luminoso e
    pulito. Menu' contenuto di pochi piatti ma davvero tutti
    da provare. Ottima la pizza ottimi i piatti di pesce
    ottimi i dolci. Intrattenimento e carinerie al tavolo
    per riempire i momenti di attesa. In centro a Reggello
    mancava un localino del genere.",
  "title": "Ottimo",
  "username": "francoirenetrip"
}
```


Users As explained before, the three types of users are all stored in the `users` collection. An example of a user of type "Customer" is shown below. It embeds all the reviews she wrote (in full).


```
{
  "_id": "5e0489d9af73587cf7a1e003",
  "name": "Ilaria",
  "surname": "Previato",
```

```
  "username": "IlariaPreviato",
  "password": "pwd",
  "type": "Customer",
  "reviews": [
    {
      "_id": "5e0489d9af73587cf7a1e004",
      "date": "2016-11-22T23:00:00.000Z",
      "rating": 5,
      "restaurant": "1250 Ristorante Pizzeria",
      "restaurantId": "5e0489d8af73587cf7a1dffe",
      "text": "Sono stata nel ristorante di Virginia e
         Antonio a cena e ve lo consiglio, un ristorante
         davvero accogliente adatto a tutte le occasioni. I
         piatti che mi hanno colpito sono stati i ravioli ed
         il polpo. Ottimo il servizio e anche il conto.
         Consigliatissimo!!!",
      "title": "Che spettacolo!!!!",
      "username": "IlariaPreviato"
    }
  ]
}
```


A Restaurant Owner embeds all her restaurants, which contain just the 10 most recent reviews (in full). An example is shown below.


```
{
  "_id": "5e0cc6fe87c76d357dec42f0",
  "name": "Franco",
  "surname": "Castaldi",
  "username": "Franco986",
  "password": "pwd",
  "type": "RestaurantOwner",
  "restaurants": [
    {
      "_id": "5e0cc6fe87c76d357dec42f1",
      "address": {
        "city": "Firenze",
        "country": "Italia",
        "postcode": "50124",
        "street": "Borgo San Frediano 169/r"
      },
      "approved": true,
      "categories": [
        "Contemporanea",
        "Fusion",
        "Italiana",
        "Pizza",
        "Mediterranea",
        "Internazionale",
```

```
      "Salutistica"
    ],
    "name": "La Pepiniere Ristorante Biologico",
    "numberOfReviews": 143,
    "openingHours": [
      {
        "days": "mar - dom",
        "times": [
          "19:30 - 22:30"
        ]
      }
    ],
    "options": [
      "Per vegetariani",
      "Opzioni vegane",
      "Opzioni senza glutine"
    ],
    "phoneNumber": "+39 055 223914",
    "priceRange": {
      "maxPrice": 40,
      "minPrice": 15
    },
    "rating": 4.685314685314685,
    "reviews": [
      {
        "_id": "5e0cc6ff87c76d357dec43f2",
        "date": "2019-10-10T22:00:00.000Z",
        "rating": 5,
        "restaurant": "La Pepiniere Ristorante Biologico",
        "restaurantId": "5e0cc6fe87c76d357dec42f1",
        "text": "Lo consiglio a chi ama ambienti intimi e a
           chi piace sperimentare cibi nuovi: una cucina
           vegana ricca di ricette internazionali
           rivisitate molto gustose. A chi, come me,
           vorrebbe assaggiare tutto, ma non riesce a
           mangiare troppo, consiglio il menu'
           degustazione.",
        "title": "Gusto e salute",
        "username": "Giulia A"
      },
      /* ... */
      {
        "_id": "5e0cc70087c76d357dec43fe",
        "date": "2019-11-12T23:00:00.000Z",
        "rating": 4,
        "reply": {
          "owner": "Franco986",
          "text": "Grazie Tiziana per questa condivisione.
             Vi aspettiamo alla prossima allora"
```

```
            },
            "restaurant": "La Pepiniere Ristorante Biologico",
            "restaurantId": "5e0cc6fe87c76d357dec42f1",
            "text": "Siamo capitati in questo locale per caso.
               Che bella scoperta! Nonostante fosse tardi il
               personale ci ha permesso comunque di cenare. I
               piatti sono molto particolari e diversi dai
               soliti sapori tradizionali. Il personale molto
               gentile e attento alla clientela. Torneremo
               sicuramente!",
            "title": "Molto buono!",
            "username": "780tizianap"
         }
      ]
   }
  ]
}
```


An example document of a user of type "Administrator" is presented below.


```
{
  "_id": "5e1843dd7dcca7374146cf60",
  "name": "Elisa",
  "surname": "Azzurri",
  "username": "admin5",
  "password": "pwd",
  "type": "Administrator"
}
```


1.6 Software Architecture

The client-side part of the system is composed only by the front-end. It is used to interact with the user through a graphical interface.

The server-side layer is instead divided into two sub-layers: the middleware, which contains all the modules for the business logic and for the interaction with the database, and back-end, which contains the MongoDB database.

A schematic representation of such architecture is shown in Figure 1.3.

Figure 1.3: Software Architecture

2 Web Scraping

We wrote a web scraper to retrieve a real dataset from Trip Advisor. All the data but the users' names and surnames had been scraped using the "Beautiful Soup" Python library.

```
from bs4 import BeautifulSoup
import requests
import re
import pandas as pd
import json
from datetime import date, timedelta, datetime
import dateparser
from unidecode import unidecode
import sys

# RETRIEVE THE BeautifulSoup OBJECT OF A URL
def get_soup(url):
    headers = {
        'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)'
                      ' AppleWebKit/537.36 (KHTML, like Gecko) Chrome/78.0.3904.108
 Safari/537.36'}
    r = s.get(url, headers=headers)

    if r.status_code != 200:
        print('status code:', r.status_code)
    else:
        soup = BeautifulSoup(r.text, 'html.parser')
        return soup


# RETRIEVE THE NEXT PAGE
def find_page(current_url, response):
    if not response:
        print('no response:', current_url)
        return None
    next_page = response.find('span', class_='pageNum current')
    next_page = next_page.find_next('a', class_='pageNum taLnk')
    if next_page is None:
        return None
    next_url = next_page.get('href')
    next_url = "https://www.tripadvisor.it" + next_url
    city_urls.append(next_url)
    return next_url


# RETRIEVE THE RESTAURANT
def find_restaurants(current_url, response):
    if not response:
        print('no response:', current_url)
        return
    for a in response.find_all('a', class_='restaurants-list-
    ListCell__restaurantName--2aSdo'):
        # print("url:", "https://www.tripadvisor.it" + a['href'])
        start_urls.append("https://www.tripadvisor.it" + a['href'])


# PARSE THE RETRIEVED RESTAURANT
def parse(url, response):
    if not response:
        print('no response:', url)
        return
    # print(response.prettify())

    # json object
```

```
    restaurant = {}

    # find returns None if can't find anything
    # find_all returns [] if can't find anything

    # get name

    restaurant_name = response.find('h1', class_='ui_header h1')
    if restaurant_name is None:
        return restaurant_name
    else:
        restaurant_name = restaurant_name.get_text()
        restaurant["name"] = restaurant_name

    # ----------------------------------------------------------------

    # get address

    # json object
    total_address = {}
    address = response.find('span', class_='street-address')
    if address is not None:
        address = address.get_text()
        address = address.replace(",", "")
        address = address.strip()
        total_address["street"] = address

    locality = response.find('span', class_='locality')
    if locality is not None:
        locality = locality.get_text()
        locality = locality.replace(",", "")
        locality = locality.strip()
        total_address["locality"] = locality

    country_name = response.find('span', class_='country-name')
    if country_name is not None:
        country_name = country_name.get_text()
        country_name = country_name.replace(",", "")
        country_name = country_name.strip()
        total_address["country"] = country_name
    restaurant["address"] = total_address

    # ----------------------------------------------------------------

    restaurant["approved"] = True

    # ----------------------------------------------------------------

    # get phone number

    phone_number = response.find('span', class_='detail is-hidden-mobile')
    if phone_number is not None:
        restaurant["phoneNumber"] = phone_number.get_text()

    # ----------------------------------------------------------------

    # get opening hours

    r = requests.get(url)

    times = re.findall('."days":\s*"[A-Za-z]+\s*.\s*[A-Za-z]+","times":.\s
    *"[0-9]+.[0-9]+\s*.\s*[0-9]+.[0-9]+'
                       '".(?:"[0-9]+.[0-9]+\s*.\s*[0-9]+.[0-9]+".)?.', r.text)
    # vector of json objects
    opening_hours = []
    if times:
        # to have unique objects
        times = set(times)
        for time in times:
            opening_hour = json.loads(time)
```

```
            opening_hours.append(opening_hour)
        restaurant["openingHours"] = opening_hours

    # ------------------------------------------------------------

    info = response.find_all('div',
                             class_='restaurants-detail-overview-cards-
DetailsSectionOverviewCard__categoryTitle--2RJP_')
    # json object
    prices = {}

    # vectors of strings
    types_list = []
    moments_list = []
    options_list = []
    functions_list = []

    for elem in info:
        elem_text = elem.get_text()
        next_elem = elem.find_next('div', class_='restaurants-detail-overview-cards
-DetailsSectionOverviewCard__tagText--1OH6h')
        if "FASCIA PREZZO" in elem_text:
            price_range = next_elem
            if price_range is not None:
                price_range = price_range.get_text()
                price_range = price_range.split("-")
                prices["minPrice"] = int(price_range[0])
                prices["maxPrice"] = int(price_range[1])
                restaurant["priceRange"] = prices
        if "CUCINE" in elem_text:
            type_of_cooking = next_elem
            if type_of_cooking is not None:
                type_of_cooking = type_of_cooking.get_text()
                for cooking in type_of_cooking.split(","):
                    cooking = cooking.strip()
                    types_list.append(cooking)
                restaurant["typesOfCooking"] = types_list
        if "Pasti" in elem_text:
            moments = next_elem
            if moments is not None:
                moments = moments.get_text()
                for moment in moments.split(","):
                    moment = moment.strip()
                    moments_list.append(moment)
                restaurant["mealtimes"] = moments_list
        if "Diete speciali" in elem_text:
            options = next_elem
            if options is not None:
                options = options.get_text()
                for option in options.split(","):
                    option = option.strip()
                    options_list.append(option)
                restaurant["options"] = options_list
        if "FUNZIONALITA" in elem_text:
            functions = next_elem
            if functions is not None:
                functions = functions.get_text()
                for function in functions.split(","):
                    function = function.strip()
                    functions_list.append(function)
                restaurant["features"] = functions_list

    # ------------------------------------------------------------

    # get number of reviews

    reviews = response.find('div', class_='pagination-details')
    if reviews is not None:
        reviews = reviews.find_all('b')[2]
        num_reviews = int(reviews.get_text().replace(".", ""))
```

```
195             restaurant["numberOfReviews"] = num_reviews
196         else:
197             reviews = response.find('span', class_='reviewCount')
198             if reviews is not None:
199                 reviews = reviews.get_text()
200                 num_reviews = reviews.split(" ")
201                 num_reviews = num_reviews[0]
202                 # conversion to integer value
203                 num_reviews = int(num_reviews.replace(".", ""))
204                 restaurant["numberOfReviews"] = num_reviews
205 
206         # ----------------------------------------------------------------
207 
208         # vector of json objects
209         ratings_list = []
210 
211         aggregated_ratings = re.findall('"aggregateRating":{".type":"AggregateRating"."
            ratingValue":"\d.\d"."reviewCount":"\d\d\d"}', r.text)
212         if aggregated_ratings:
213             aggregated_ratings = aggregated_ratings[0]
214         if "ratingValue" in aggregated_ratings:
215             aggregated_ratings = aggregated_ratings.replace('"aggregateRating":', "")
216             aggregated_ratings = json.loads(aggregated_ratings)
217             rating = {}
218             rating["name"] = "Globale"
219             value = aggregated_ratings.pop("ratingValue")
220             rating["rating"] = float(value)
221             ratings_list.append(rating)
222         # ----------------------------------------------------------------
223 
224         ratings = re.findall('"ratingQuestions":.(?:."name":"Cucina".\s*"rating":\s*\d
            *.\s*"icon":"\w*".)?.?'
225                              '(?:."name":\s*"Servizio".\s*"rating":\s*\d*.\s*"icon":"\w
            *".)?.?'
226                              '(?:."name":\s*"Qualit.......prezzo".\s*"rating":\s*\d*.\s
            *"icon":\s*"\w*.\w*".)?.?'
227                              '(?:."name":\s*"Atmosfera".\s*"rating":\s*\d*.\s*"icon":\s
            *"\w*"..)?', r.text)
228 
229         if ratings:
230             ratings = ratings[0]
231             if "name" in ratings:
232                 ratings = ratings.replace('"ratingQuestions":', "")
233                 ratings = ratings.replace("[", "")
234                 ratings = ratings.replace("]", "")
235                 ratings = ratings.replace("}", "}}")
236                 ratings = ratings.split("},")
237                 ratings[-1] = ratings[-1].replace("}}", "}")
238                 for rating in ratings:
239                     rating = json.loads(rating)
240                     rating["rating"] = rating["rating"] / 10
241                     rating.pop("icon", None)
242                     ratings_list.append(rating)
243                 restaurant["ratings"] = ratings_list
244 
245         # ----------------------------------------------------------------
246 
247         # get reviews
248 
249         url = url.replace('-Reviews-', '-Reviews-or{}-')
250         print('template:', url)
251 
252         # vector of json objects
253         reviews_list = []
254         # load pages with reviews
255         for offset in range(0, num_reviews, 10):
256             # print('url:', url.format(offset))
257             url_ = url.format(offset)
258             find_reviews(url_, get_soup(url_), r.text, reviews_list, num_reviews)
259             # print("offset:", offset)
```

```
260
261         restaurant["reviews"] = reviews_list
262
263         return restaurant
264
265
266     # RETRIEVE REVIEWS OF A RESTAURANT
267     def find_reviews(url, response, source_code, review_list, num_reviews):
268         # print('review:', url)
269         if not response:
270             print('no response:', url)
271             return
272         # print(response.prettify())
273
274         divs = response.find_all('div', class_='ui_column is-9')
275         # "widgetEvCall('handlers.usernameClick', event, this);"><div>133leonardoe</div
            >
276         # regex = re.compile('widgetEvCall..handlers.usernameClick..\s*event.\s*this
            .....div.\w*(?:\s*\w*)?..div.')#va con underscore
277
278         for div in divs:
279             if len(review_list) >= num_reviews:
280                 return
281
282             # for div, match in zip(divs, regex.finditer(source_code)):
283             review = {}
284
285             href = div.find('a', class_='title')
286
287             if href:
288                 href = href.get('href')
289                 #print("https://www.tripadvisor.it" + href)
290                 res = get_soup("https://www.tripadvisor.it" + href)
291
292                 if res is None:
293                     continue
294
295                 # ------------------------------------------------------------------
296
297                 # get date
298
299                 review_date_span = res.find('span', class_='ratingDate relativeDate')
300                 # le domande hanno lo stesso div ma non hanno lo span dentro
301                 if not review_date_span:
302                     return
303                 review_date = review_date_span.get_text()
304                 if review_date is not None:
305                     if "Recensito il " in review_date:
306                         review_date = review_date.split("Recensito il ")[1]
307                         r_date = dateparser.parse(review_date, languages=['it'])
308                     elif "Recensito " in review_date:
309                         review_date = review_date.split("Recensito ")[1]
310                         r_date = dateparser.parse(review_date, languages=['it'])
311                     r_date = r_date.date()
312                     review["date"] = str(r_date)
313
314                 # ------------------------------------------------------------------
315
316                 ratings = []
317
318                 rating = {}
319
320                 # get global rating
321
322                 review_span = res.find('span', class_=re.compile("ui_bubble_rating
            bubble_.0"))
323                 # print(review_span.get('class'))
324                 if review_span is not None:
325                     review_span = review_span.get('class')[1]
326                     score = int(int(review_span.split("_")[1]) / 10)
```

```
                if score is not None:
                    rating["name"] = "Globale"
                    rating["rating"] = float(score)
                    ratings.append(rating)

            # ------------------------------------------------------------

            # get single ratings

            ratings_list = res.find('div', class_='rating-list')
            ratings_list = ratings_list.find('li')
            if ratings_list:
                ratings_list = ratings_list.find_all('ul')
                for elem in ratings_list:
                    li = elem.find_all('li')
                    for l in li:
                        rating = {}
                        name = l.find('div', class_='recommend-description').
    get_text()
                        if "Cibo" in name:
                            name = "Cucina"
                        rating["name"] = name
                        value = str(l.find('div'))
                        value = value.replace("ui_bubble_rating bubble_", "")
                        value = value.split("><")[0]
                        value = value.split("class=")[1]
                        value = value.replace('"', '')
                        value = int(value) / 10
                        rating["rating"] = value
                        ratings.append(rating)

            review["ratings"] = ratings

            # ------------------------------------------------------------

            review_block = res.find('div',
                                    class_='review hsx_review ui_columns is-
    multiline is-mobile inlineReviewUpdate provider0')

            # get title

            review_title = review_block.find('span', class_='noQuotes')
            if review_title:
                review_title = review_title.get_text()
                if review_title is not None:
                    review["title"] = review_title

            # ------------------------------------------------------------

            # get text

            review_text_div = review_block.find('p', class_='partial_entry')
            if review_text_div is not None:
                review_text = review_text_div.get_text()
                review["text"] = review_text

            # ------------------------------------------------------------

            # get username

            username = review_block.find('span', class_='expand_inline scrname')
            if username is not None:
                username = username.get_text()
                review["username"] = username

            # ------------------------------------------------------------

            # get reply of restaurant owner

            owner_reply = review_block.find('div', class_='mgrRspnInline')
```

```
            if owner_reply:
                reply = owner_reply.find_next('p', class_='partial_entry')
                if reply:
                    reply = reply.get_text()
                    review["ownerReply"] = reply
                owner_username = owner_reply.find('div', class_='header')
                if owner_username:
                    owner_username = owner_username.get_text()
                    owner_username = owner_username.split(",")[0]
                    review["ownerUsername"] = owner_username

            review_list.append(review)

        # ------------------------------------------------------------

    return review_list


# MAIN

start_urls = []

"""
city = "pisa"
city_urls = ['https://www.tripadvisor.it/Restaurants-g187899-
    Pisa_Province_of_Pisa_Tuscany.html']
"""

"""
city = "firenze"
city_urls = ['https://www.tripadvisor.it/Restaurants-g2043770-
    Province_of_Florence_Tuscany.html']
"""


city = "siena"
city_urls = ['https://www.tripadvisor.it/Restaurants-g2043779-
    Province_of_Siena_Tuscany.html']

"""
city = "livorno"
city_urls = ['https://www.tripadvisor.it/Restaurants-g2043772-
    Province_of_Livorno_Tuscany.html']
"""
"""
city = "arezzo"
city_urls = ['https://www.tripadvisor.it/Restaurants-g2043769-
    Province_of_Arezzo_Tuscany.html']
"""
s = requests.Session()

city_url = city_urls[0]

# get just the first 10 pages (about 300 restaurants for each city)
while (len(city_urls) < 10) and (city_url is not None):
    city_url = find_page(city_url, get_soup(city_url))

for city_url in city_urls:
    find_restaurants(city_url, get_soup(city_url))

for restaurant_url in start_urls:
    start = datetime.now()

    # parse information for each url
    restaurant = parse(restaurant_url, get_soup(restaurant_url))

    if restaurant is None:
        continue

    punctuations = '''!()-[]{};:'"\,<>./?@#$%^&*_~'''
```

```
460
461     try:
462         json_file_name = restaurant["name"].lower().replace(" ", "")
463
464         formatted_name = ""
465         for char in json_file_name:
466             if char not in punctuations:
467                 formatted_name = formatted_name + char
468
469         json_file_name = "./../data/" + city + "-" + formatted_name + ".json"
470         json_file = open(json_file_name, 'w', encoding='utf-8')
471         json.dump(restaurant, json_file, ensure_ascii=False, indent=4)
472     except:
473         print("Error in writing " + json_file_name + " " + str(sys.exc_info()[0]))
474         raise
475
476     end = datetime.now()
477     print(end - start)
```

The data collected through the script got parsed in order to fit the structure of the collections.

3 Implementation

3.1 UML Class Diagram

Figure 3.1 shows the UML class diagram.

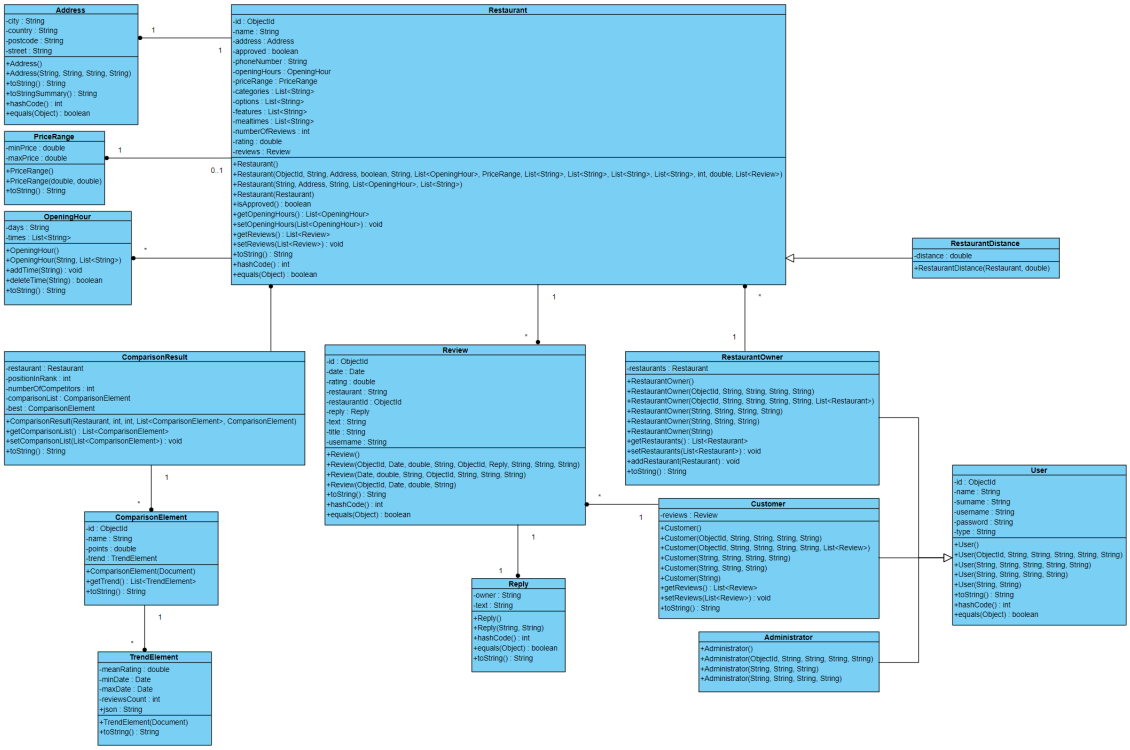


Figure 3.1: UML Class Diagram

3.2 Description of Software Modules

In our project, Java classes are organized in different packages according to their role:

- `application`
- `backend`, which also contains:
  - `modules`
  - `exc`
- `test`

Such packages are explained more in detail in the following sections.

3.2.1 The `application` Package

The `application` package contains all the classes involved in the management of the graphical user interface, which was build using JavaFX. Within this framework, the main structure of the different scenes is contained in FXML files, whereas the logic and the behavior associated to a specific file are implemented through a Controller, which is a Java class. In the following, a description of each class of the package is provided.

`App` is the central class for the graphical user interface. It coordinates the other controllers and classes of the front-end layer of the application.

`Controller` contains some common methods that are used by the other controller classes. In this way, each child class does not have to re-implement every time the same method.

`FistSceneController` is the class that manages the first scene of the application. In this scene the user can decide in which city the application should search restaurants or login as a Restaurant Owner or as an Administrator.

`CustomerController` is the class involved in the management of the Customer's main scene. It offers many functionalities, including for example:

- the search for a particular restaurant after the setting of some parameters;
- the possibility to read and write a review of a restaurant;
- the possibility to visualize a chart of the distribution of the review scores for a particular restaurant;
- the possibility to register a new a client in the system;
- the possibility to create a new account.

`HistoryReviewsController` is the class that allows a Customer, after the login, to visualize her past reviews, if any.

`RestaurantOwnerController` is the class that handles the login and the registration of Restaurant Owners.

`RestaurantOwnerLoggedController` lets an authenticated Restaurant Owner visualize her restaurants, including the ones that are not approved yet, and reply to reviews. This controller also lets the Restaurant Owner visualize the trend of the reviews of her restaurants and a comparison with competitors.

`NewRestaurantController` manages the insertion of a new restaurant in the system.

`AdministratorFirstSceneController` is the class that handles the authentication of an Administrator.

`AdministratorLoggedController` provides an authenticated Administrator with many functionalities, such as:

- the elimination from the system of a Restaurant Owner or a Customer;
- the possibility to accept or refuse the request of adding a new restaurant in the system;
- the removal of a review.

3.2.2 The `backend` Package

The `backend` package contains all the classes that allow the interaction with the MongoDB database, which is provided by the MongoDB Java Driver.

`DatabaseManager` is the main class of this package. It opens and closes the connections to the database and performs queries, providing a simple interface to the other classes. This class hides the complexity and the details of the database from the rest of the application by mapping database entities into Java objects. To implement such associations between documents and objects, we used Jackson, a JSON processor for Java.

The `backend.modules` package contains all the Java classes that are needed to map the JSON documents provided by MongoDB Java Driver to objects. In other words, these classes are models for the different database entities. Such classes are used throughout the whole application, and this let us decouple the different software layers: modules exchange data through Java classes, not JSON documents, making database details transparent.

Each document structure has a corresponding class. The `Restaurant` class is shown below as an example. Inner JSON documents are modelled as class member variables, while arrays are modelled as Java collections. Other classes are implemented analogously.

```
1  public class Restaurant {
2
3      @JsonDeserialize(using = ObjectIdDeserializer.class)
4      @JsonProperty("_id")
5      private ObjectId id;
6      private String name;
7      private Address address;
8      private boolean approved;
9      private String phoneNumber;
10     private List<OpeningHour> openingHours;
11     private PriceRange priceRange;
12     private List<String> categories;
13     private List<String> options;
14     private List<String> features;
15     private List<String> mealtimes;
16     private int numberOfReviews;
17     private double rating;
18     private List<Review> reviews;
19
20     // constructors, getters and setters omitted for brevity
21 }
```

Model classes constitute the parameters and/or the return type of the public methods of the `DatabaseManager` class, which has to serialize and deserialize them to/from JSON documents. For example, the `getRestaurants()` method reads restaurants from the database and returns them as a `List<Restaurant>`. The actual mapping is performed at line 20 of the code snippet shown below.

```
1  public List<Restaurant> getRestaurants(String category, String city, int
       pageNum, boolean approved)
2              throws DatabaseManagerException {
3
4      MongoCollection<Document> collection = mongoDatabase.getCollection("
       restaurants");
5
6      Bson filter = and(
7          eq("address.city", city),
8          eq("categories", category),
9          eq("approved", approved));
10     MongoCursor<Document> cursor = collection
11             .find(filter)
12             .skip(pageSize * (pageNum - 1))
13             .limit(pageSize)
14             .iterator();
15
16     List<Restaurant> list = new ArrayList<>(pageSize);
17     ObjectMapper mapper = new ObjectMapper();
18     try {
19         while (cursor.hasNext()) {
20             Restaurant restaurant = mapper.readValue(cursor.next().toJson(),
        Restaurant.class);
21             list.add(restaurant);
22         }
23     } catch (Exception e) {
24         throw new DatabaseManagerException(e.getMessage());
25     } finally {
26         cursor.close();
27     }
28
29     return list;
30 }
```

The serialization of classes to JSON documents is performed through private methods that return a `org.bson.Document`, which is then passed to the methods provided by the MongoDB Java Driver, such as `insertOne()`. An example is shown below.

```
1  private Document initializeRestaurantDocument(Restaurant restaurant) {
2      Document document = new Document();
3
4      document.append("name", restaurant.getName())
5          .append("address", restaurant.getAddress())
6          .append("approved", restaurant.isApproved())
7          .append("rating", restaurant.getRating())
8          .append("numberOfReviews", restaurant.getNumberOfReviews());
9      if (restaurant.getPhoneNumber() != null)
10         document.append("phoneNumber", restaurant.getPhoneNumber());
11     if (restaurant.getOpeningHours() != null)
12         document.append("openingHours", restaurant.getOpeningHours());
13     if (restaurant.getPriceRange() != null)
```

```
14            document.append("priceRange", restaurant.getPriceRange());
15        if (restaurant.getCategories() != null)
16            document.append("categories", restaurant.getCategories());
17        if (restaurant.getOptions() != null)
18            document.append("options", restaurant.getOptions());
19        if (restaurant.getFeatures() != null)
20            document.append("features", restaurant.getFeatures());
21        if (restaurant.getMealtimes() != null)
22            document.append("mealtimes", restaurant.getMealtimes());
23
24        return document;
25    }
```

The `backend.exc` package contains all the exceptions that `DatabaseManager` can throw. We defined such exceptions to ease the management of errors when invoking the methods of such class.

3.2.3 The `test` Package

The `test` package contains the classes used to perform an automatic test of the functionalities provided by the `DatabaseManager` class. Tests will be discussed more in detail in the section 3.7.

3.3 Transactions

All write and update operations that involve more than one document of the dataset are implemented using MongoDB multi-document transactions. We have reviews and restaurants duplicated across multiple documents and each operation that inserts or updates a restaurant or a review must update all the documents that contain information about it.

Multi-document transactions are atomic, which means that:

- before the transaction commit, all data changes executed in the transaction are not visible outside the transaction;
- when a transaction commits, all the performed changes are saved and visible outside the transaction;
- when a transaction aborts, all data changes made in the transaction are discarded without ever becoming visible.

Multi-document transactions may affect performances of operations, however we recall that they are used only for write operations and our application is mainly read-heavy.

We report, as an example, the implementation of the operation that deletes an account. It is implemented as a multi-document transaction because, if the user to delete is a restaurant owner, also all his restaurants and related reviews must be deleted and three collections in the dataset must be accessed. When a multi-document transaction contains a read operation, we must explicitly specify that the operation must be handled by the primary node. Details about primary node are provided when talking about replica set in section 3.4.1.

```java
public User deleteAccount(User user) throws UserNotFoundException {

    startSession();

    TransactionBody<User> transaction = new TransactionBody<User>() {

    @Override
    public User execute() {

        MongoCollection<Document> usersCollection =
            mongoDatabase.getCollection("users");

        // if the user is a restaurant owner retrieve his restaurants
        RestaurantOwner owner = null;

        Bson filter;

        // if the user is a restaurant owner remove all his restaurants
        if (user instanceof RestaurantOwner) {
            try {
                ObjectMapper mapper = new ObjectMapper();
                filter = eq("username", user.getUsername());
                owner = mapper.readValue(
                            usersCollection.find(clientSession, filter)
                            .cursor().next().toJson(),
                            RestaurantOwner.class);
            } catch (Exception e) {
                return null;
            }

            MongoCollection<Document> restaurantsCollection =
                mongoDatabase.getCollection("restaurants");
            MongoCollection<Document> reviewsCollection =
                mongoDatabase.getCollection("reviews");

            // for each restaurant
            for (Restaurant restaurant : owner.getRestaurants()) {

                // delete the restaurant from the collection of restaurants
                filter = eq("_id", restaurant.getId());
                restaurantsCollection.deleteOne(clientSession, filter);

                // delete reviews of that restaurant from the collection
                // of reviews
                filter = eq("restaurantId", restaurant.getId())
                reviewsCollection.deleteMany(clientSession, filter);
            }
        }

        // delete the user from the collection of users
        filter = eq("username", user.getUsername());
        DeleteResult res = usersCollection.deleteOne(clientSession, filter);
        if (res.getDeletedCount() == 0)
            return null;

        return user;
    }   };

    // read preference in a transaction must be primary
    TransactionOptions options = TransactionOptions.builder()
```

```
61                                      .readPreference(ReadPreference.primary())
62                                      .build();
63
64      User deletedUser = clientSession.withTransaction(transaction, options);
65
66      if (deletedUser == null) {
67          throw new UserNotFoundException();
68      }
69
70      closeSession();
71
72      return deletedUser;
73  }
```

3.4 Implementation of Non-Functional Requirements

Availability High availability and redundancy can be provided by using replication. By having multiple copies of data on different servers, the application can be also fault tolerant against the loss of one or more database servers. In MongoDB, the mechanism exploited to support replication and automated failover is the replica set, a group of servers with a primary and multiple secondaries. Section 3.4.1 describes in detail how we decided to implement a replica set for our application.

Scalability and Performance Our application must be able to handle an increasing number of users without affecting the overall performances. As explained in section 3.4.1, if read operations are distributed across nodes we can achieve better performance. Since the application is read-heavy, read operations should be distributed to secondary members and indexes should be used to support the most time consuming queries, as described in section 3.4.2. The application shall also read from the closest copy of the data to have low latency.

Consistency Writes are issued to the primary server of the replica set, while the application reads from the nearest server in order to improve performance. In case the closest node is not the primary server, the copy of the data may not be up to date, since we admit reading from secondaries. In our scenario it is acceptable for data to be slightly out of date since latency is more important than consistency. Hence the eventual consistency model is the most suitable for our application.

3.4.1 Replica Set

The application shall be high available to let users always be able to search restaurants and visualize reviews. A suitable architecture to ensure high availability consists of a geographically redundant replica set. A good compromise between redundancy and complexity is a replica set composed of three members distributed across two data centers: two members are placed in one data center and the remaining server is placed in another one.

In a MongoDB replica set one and only one node is chosen as **primary** node and is in charge of handling write requests. Other nodes are called **secondaries** and maintain a copy of the dataset of the primary. The primary node records all the changes performed to

its dataset in an operation log, called oplog, and secondary nodes asynchronously replicate, on their own dataset, the operations contained in the oplog of the primary.

Write and Read Requests As already mentioned, write requests are always handled by the primary. Read operations, instead, are routed to the primary by default, but they can be configured to be handled also by secondary nodes. In this way, however, the user is not guaranteed to see the most recent version of data. Our application is read-heavy and main read operations retrieve information about restaurants and reviews. For such operations it is not an issue if read data are stale: indeed, if a recently inserted restaurant or review is missing in the result set of a query, it is not a real problem because there are a lot of other restaurants and reviews to visualize. Hence, since consistency is not an issue, we decided to specify *nearest* as MongoDB read preference option. Read preference describes how MongoDB clients route read operations to the members of a replica set. If we specify *nearest* as read preference, reads are routed to the member with the lowest network latency, without considering whether a member is a primary or a secondary. In this way, we also try to improve performances, indeed fast response is one of the other requirements of our application.

We also decided to set MongoDB write concern w equal to 1. Write concern for a replica set determines the number of members that must acknowledge a write operation before returning success. When w is set to 1, the write operation is acknowledged when it has been propagated only to the primary node. In this way the application does not need to wait for data to be replicated on multiple nodes and this consents to have the fastest response possible.

Protection against Single-Instance Failure By using a three-members replica set we provide protection against single-instance failures. In a replica set, a primary node requires a majority to be elected: in a three-members replica set the majority required to elect a primary is composed of two nodes. In the chosen replica set architecture, if one node goes down, there are still two available members that can constitute a majority and thus are able to elect another primary, so that application can continue to work. If two members go down, instead, there is no more possibility of electing a primary and, therefore, it is not possible to manage handling write requests. However, the dataset is still available on the remaining node and the application can continue to accept read operations (i.e. users can continue to search restaurants and reviews). Indeed we decided to route read operations to the nearest node regardless the status of each member. When one of the failed nodes is substituted or repaired, the application can start again to handle write operations. In section 3.7.2 some examples are reported to show how nodes behave in case of failure and restore of nodes.

Protection against Data Center Failure If data centers are placed far enough, we also provide protection against the failure of one data center. Indeed, in case of issues such as power outages, network interruptions or natural disasters, just one data center would be damaged. If the data center with one node goes down, the application continues to work because nodes in the other data center are enough to elect a primary and both read and write operations can be handled. If the data center with two nodes goes down, the replica set becomes read-only because there is still another member alive in the other data center that can accept read operations.

Implementation

Mongod Instances The servers of the replica set are implemented as three `mongod` instances. Each `mongod` instance is launched using the following command in which the configuration name is specified: `mongod -config ./config0.yalm`.

Three MongoDB configuration files are used, one for each node; they contain the following fields:

- `port` - the port on which the server listens,
- `replSetName` - the name of the replica set,
- `dbPath` - the path of the folder in which the copy of the database hold by the node is stored,
- `destination`, `path`, `logAppend` - details on how the server log is maintained.

One configuration file is reported as example:

```
net:
  port: 20000

replication:
  replSetName: replicaSet

storage:
  dbPath: .\replicaSet\dataCenter0\server0

systemLog:
  destination: file
  path: .\replicaSet\dataCenter0\server0.log
  logAppend: true
```

Read Preference Read preference is instead configured in the Java code when opening the connection to the replica set.

```
public void openMongoDBConnection(usageType type) {
    CodecRegistry pojoCodecRegistry = fromRegistries(
            MongoClientSettings.getDefaultCodecRegistry(),
            fromProviders(PojoCodecProvider.builder().automatic(true)
                    .build()));

    // ...

    if(type == type.APP) {
        mongoClient = MongoClients.create(MongoClientSettings.builder()
                .applyToClusterSettings(
                    builder -> builder.hosts(Arrays.asList(
                        new ServerAddress("localhost", 20000),
                        new ServerAddress("localhost", 20001),
                        new ServerAddress("localhost", 20002))))
                .readPreference(ReadPreference.nearest())
                .codecRegistry(pojoCodecRegistry).build());
    }
}
```

3.4.2 Indexes

Indexes support the efficient execution of queries in MongoDB. In this section we will present the indexes used and their execution statistics, evaluated for the replica set. All the indexes presented below are regular and compound.

Approved, City, Category, Rating Index This index supports both analytic functions and searches of restaurants by city and category. The rating field is part of the index since this usually provides better performance than using the sort method at the application level. The order of the fields is chosen according to the winning plan provided by the query optimizer, automatically called by the explain function operating in the `queryPlanner` mode, which compared it with all the indexes with the same attributes ordered in a different way.

If the index is not used, the number of documents scanned depends on the city, since documents referring to restaurants of a given city are contiguous inside the collection. In most cases, the whole collection is not scanned because the limit function, which is used to paginate results at the application level, returns as soon as it reaches the amount of documents required. The JavaScript equivalent query we used in our project is shown below along with its performance.

```
db.restaurants.find({
    "address.city": "Arezzo",
    "categories": "Italiana",
    "approved": true
}).limit(10).explain("executionStats")
```

```
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 10,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 10,
        "totalDocsExamined" : 10,
```

Figure 3.2: With Index

```
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 10,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 0,
        "totalDocsExamined" : 402,
```

Figure 3.3: Without Index

As it can be seen from the figures, the amount of documents scanned when the index is used is considerably lower.

Approved, City, Rating Index This index supports both searches by city and by city and name. There exists a single index because there is a single query for both cases. The idea is that when the name is provided, we use the Apache Commons Text library to filter, at the application level, the results returned by the query by comparing the name typed by the user with the names of the restaurants belonging to a specific city. Here again, the rating as part of the index avoids the use of the sort function. The index is chosen by the query optimizer as the winning plan, while the `cityApprovedRating` one is rejected.

As it holds for the previous case, even here if the index is not used, the selected city determines the number of documents scanned. Nevertheless, performance are improved.

```
db.find({
    "address.city": "Pisa",
    "approved": true
}).limit(10).explain("executionStats")
```

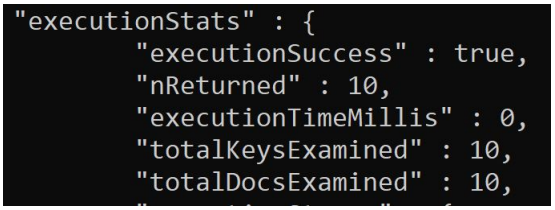


Figure 3.4: With Index

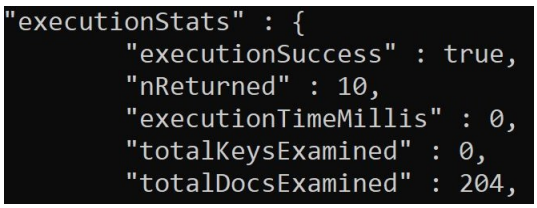


Figure 3.5: Without Index

The figures again show that the number of documents scanned drops when the index is used.

If the approved field is set to false, the scenario is pretty similar. There is only one restaurant that needs to be approved or rejected in Pisa, therefore, even if the limit function is used, the whole collection needs to be scanned.

```
db.find({
    "address.city": "Pisa",
    "approved": false
}).limit(10).explain("executionStats")
```

```
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 1,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 1,
        "totalDocsExamined" : 1,
```


Figure 3.6: With Index


```
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 1,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 0,
        "totalDocsExamined" : 692,
```


Figure 3.7: Without Index

Type, Username Index This index supports the login operation. This is the most time-expensive query, since the `users` collection is the one containing the largest number of documents, therefore this index allows the application to save most of the time with respect to the case without index. The index is selected as the winning plan by the query optimizer, while the inverse one is rejected.

The performance gain obtained by the use of this index is the largest one with respect to all the cases analyzed. The index does not contain the password field since the username is unique.

```
db.users.find({
    "type": "Customer",
    "username": "Bigo1973",
    "password": "pwd"
}).explain("executionStats")
```


```
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 1,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 1,
        "totalDocsExamined" : 1,
```

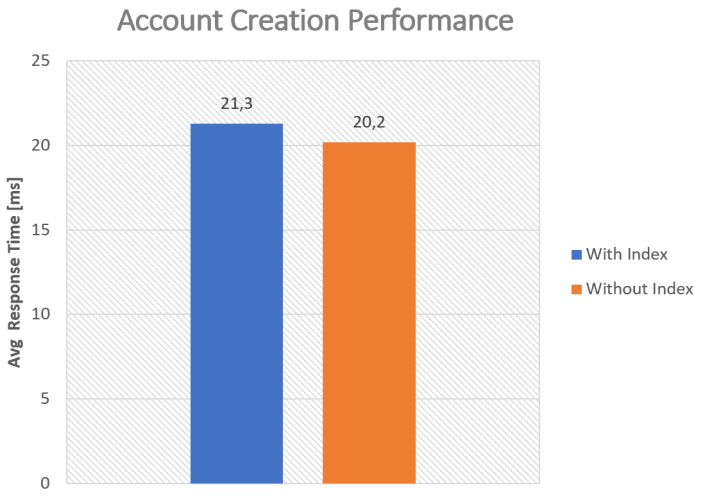

Figure 3.8: With Index

```
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 1,
        "executionTimeMillis" : 386,
        "totalKeysExamined" : 0,
        "totalDocsExamined" : 195424,
```

Figure 3.9: Without Index

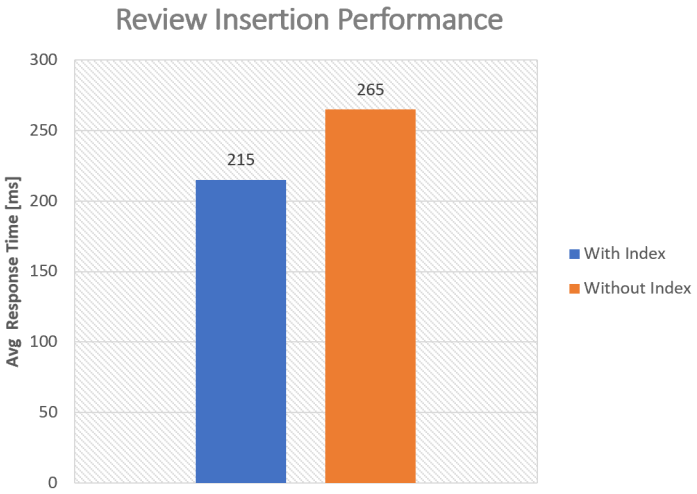
As it holds for the previous indexes, the number of documents scanned is smaller with respect to the case without index. Moreover, since the query is really time consuming, it can be noticed there is a large improvement regarding the time required for the execution of the query that passes from 386 ms to less than 1 ms.

Account Insertion In general, the performance gains that indexes provide for read operations are worth the insertion penalty. In order to see if this is true in our case, we consider the operation of creation of a new account. The chart shown below illustrates the average performance of the insert operation when the index is used and when it is not.



The penalty is not that large: 21.3 ms with the index and 20.2 ms without it. On the top of that, the create account operation is not that frequent with respect to the login one, therefore it is worth using the index.

Review Insertion Something different happens when we need to insert a review. This functionality contains several update operations which are needed to match the data model. Since the updates take advantage of the `typeUsername` index, it can be noticed, by looking at the chart below, that also the overall review insertion operation performs better when indexes are used. The average time required to insert a review is 215 ms when the index is used, otherwise it is 265 ms.



3.5 Sharded Cluster

Sharding solves the demand of data growth by allowing to store data across multiple machines. As already mentioned, scalability is one of the main requirements of our application and sharding can be a good solution to support a high query rate on large datasets. In a sharded cluster data are stored across multiple machines, in this way each shard can process a subset of the overall cluster read and write operations. This approach also provides high availability since each shard is a separate replica set.

The choice of the shard key is crucial. In our case, we can use the city as the shard key for the `restaurants` collection because it divides evenly the dataset across the different shards. In order to shard the collection, an index that contains the shard key as prefix is needed, therefore indexes have to be modified to match this pattern.

The same reasoning holds for the `users` and the `reviews` collections. In order to split them in the same way we need to add a field in both collections. The reviews collection should contain, for each review document, a field containing the city of the restaurant and the users collection should incorporate a field for the user's city.

To implement a geographical partitioning we can use zone sharding, which creates zones of sharded data based on the shard key. With such implementation we can route each query to the nearest shard in order to have low-latency read and write operations for globally distributed clients. For example, if we had three cities we would have the following situation:

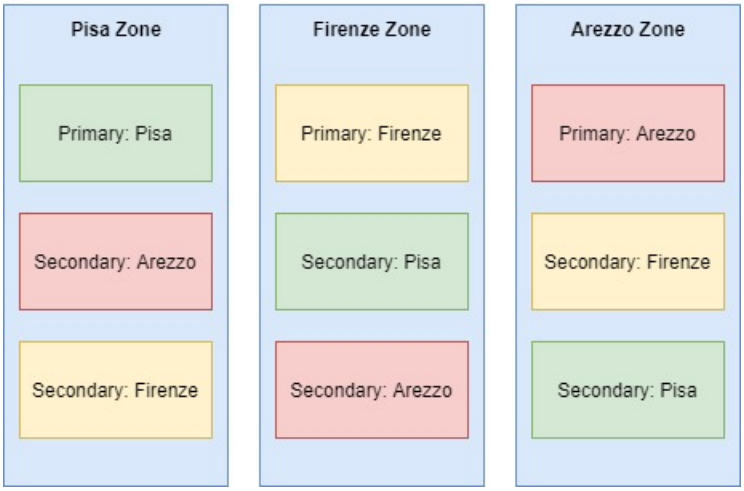


Figure 3.10: Example with Three Cities

A possible implementation of this approach consists of having shards in three data centers - one in Pisa, one in Firenze and one in Arezzo. Each data center is associated to a zone. Each zone contains the primary replica set member for the shard or shards associated with that zone, to ensure responsive reads and writes, and read-only secondaries to facilitate local reads of globally distributed data. Local application instances will presumably interact with the primary member since restaurants, and the corresponding reviews, in a given city are likely to be searched by citizens or people visiting the city, but they can also efficiently retrieve data generated in other zones thanks to the presence of the secondaries.

3.6 Analytics

As previously anticipated in the Functional Requirement section, the application provides some analytics functionalities, which are implemented using MongoDB aggregation pipeline.

Restaurant Owners can see the trend of a restaurant, i.e. the temporal evolution of its reviews. Reviews are aggregated in temporal windows, whose typical length is a week. The trend is constituted by the series of the mean values of reviews computed window by window. Old reviews are filtered out. Figure 3.11 shows an example of trend, with a window of two weeks.

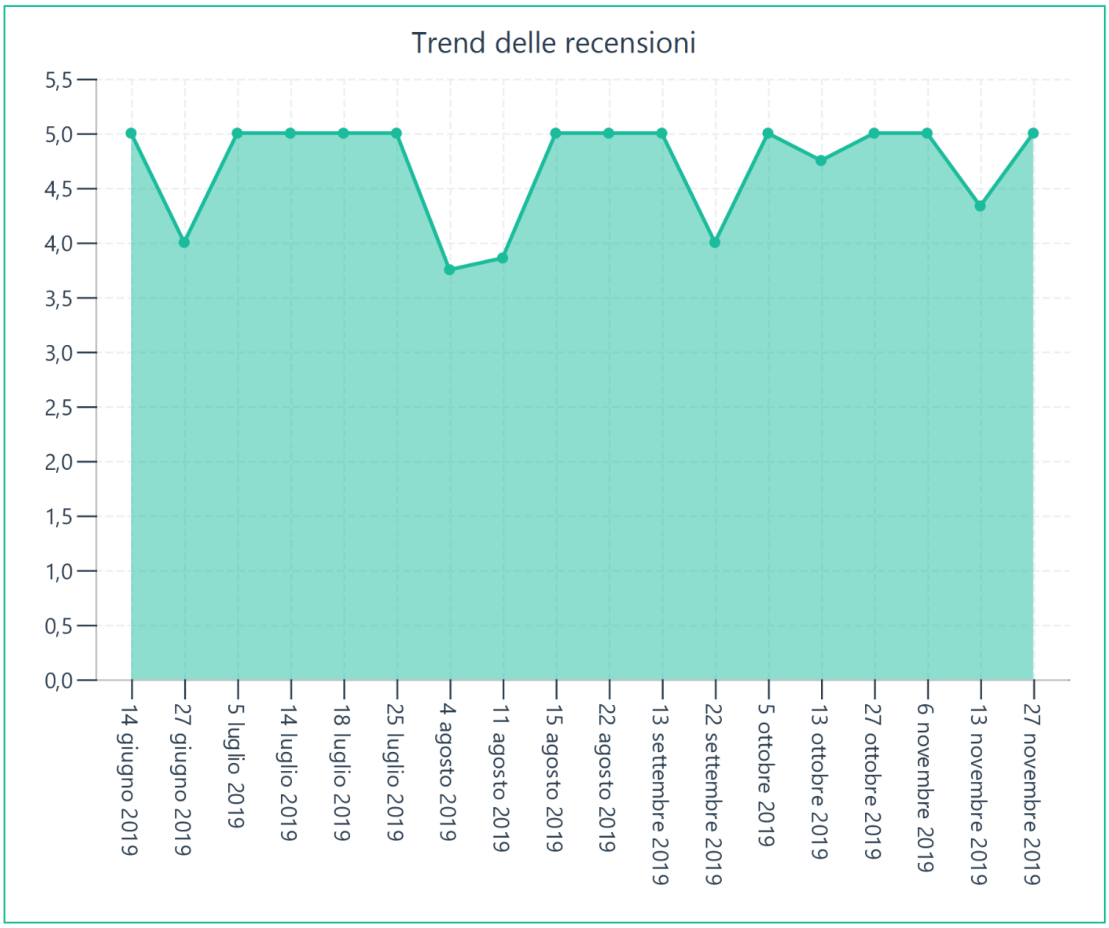


Figure 3.11: Restaurant Trend

Restaurant Owners can see a comparison between a restaurant and its competitors. For the purpose of this functionality, two restaurants are considered as competitors if they are in the same city and have a number of common categories which is above a certain threshold. To establish the rank of restaurants, competitors are sorted according to the mean value of the last reviews (old reviews are again filtered out). Competitors can be many, therefore only close competitors and the very best one are shown in order to have a cleaner chart. For each of them, the trend is computed and shown to the Restaurant Owner. The result also contains the number of competitors and the position of the given restaurant in the rank.

Figure 3.12 illustrates an example of this comparison. On the left-hand side, a bar chart shows the restaurants and the mean values of their reviews. On the right-hand side there are their trends.

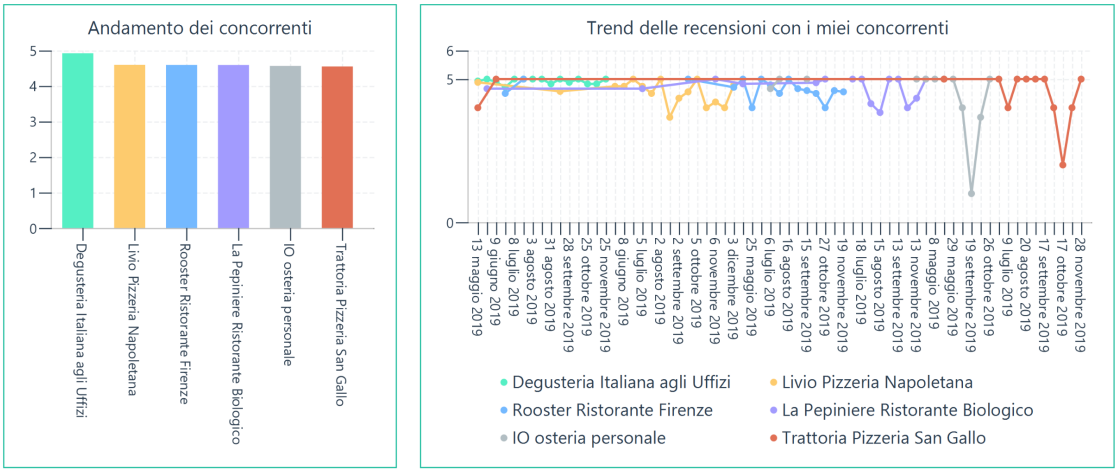


Figure 3.12: Comparison with Competitors

Users can see the distribution of the reviews of a restaurant. Reviews are aggregated by rating in order to compute the percentage of reviews of each possible vote, so that users can immediately understand whether a restaurant is generally appreciated or not. Old reviews are filtered out. As shown in figure 3.13, the application represents the distribution of reviews with a pie chart.

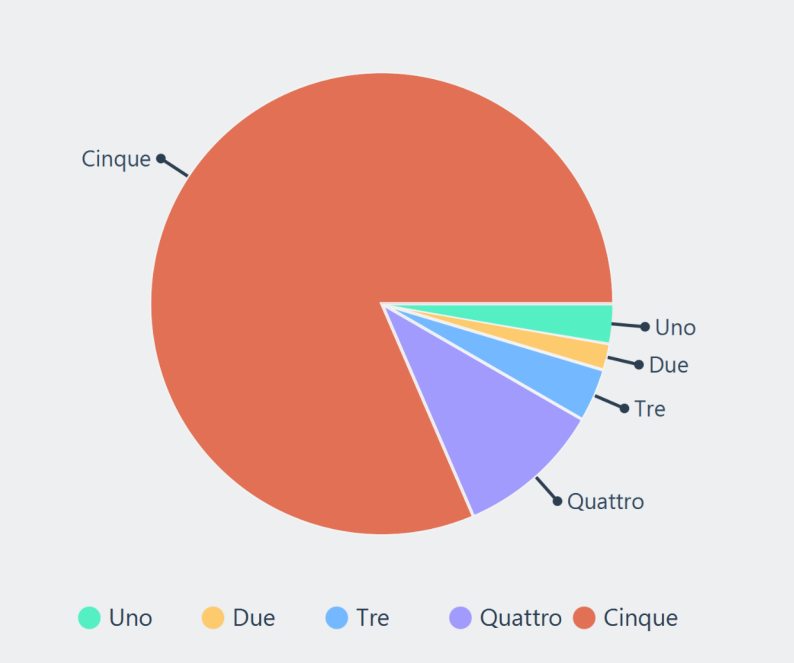


Figure 3.13: Reviews Distribution

3.7 Validation and Test

3.7.1 JUnit Tests

The deployed Java queries against the MongoDB database are tested using JUnit. During the test phase we must use *primary* as read preference mode, otherwise the test might fail. This is due to the fact that each write operation in the test is immediately followed by a read operation that verifies if the first one has been successfully executed. If we used the replica set with *nearest* as read preference, this read operation could be sent to a secondary member which might have not yet performed the write operation on its copy of the dataset yet. Using *primary* as read preference instead, we can correctly evaluate the implemented operations.

The following functionalities are tested:

- the insertion of a new user
- the login
- the deletion of a user
- the insertion and the update of a restaurant
- the read of restaurants and pending requests
- the approval and the rejection of a pending request
- the insertion of a review
- the read of reviews of a given restaurant
- the insertion of a reply
- the deletion of a review

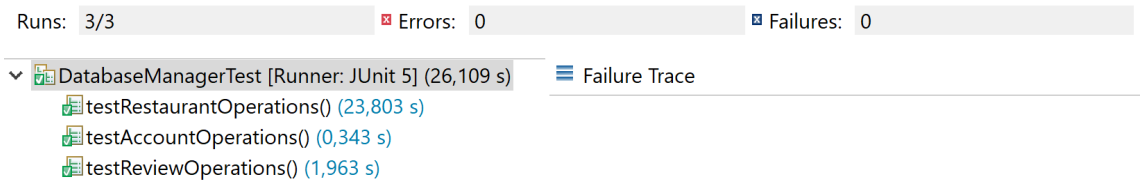


Figure 3.14: JUnit Test

3.7.2 Replica Set Tests

We start from an initial configuration in which the first data center is composed of two servers which listen respectively on port `20000` and `20001`, whereas the server on the other data center listens on port `20002`. If we connect with a Mongo Shell to one of the ports and type `rs.status()` command we can see that one node has been elected as primary, whereas the others are secondaries. In the example shown in the following, the node that listens on port `20000` is the primary. In the configuration of secondaries the `syncingTo` field indicates the node from which they replicate their copy of the database.

```
{
        "_id" : 0,
        "name" : "localhost:20000",
        "ip" : "127.0.0.1",
        "health" : 1,
        "state" : 1,
        "stateStr" : "PRIMARY",
        "uptime" : 1096,
        "optime" : {
                "ts" : Timestamp(1579455009, 1),
                "t" : NumberLong(2)
        },
        "optimeDurable" : {
                "ts" : Timestamp(1579455009, 1),
                "t" : NumberLong(2)
        },
        "optimeDate" : ISODate("2020-01-19T17:30:09Z"),
        "optimeDurableDate" : ISODate("2020-01-19T17:30:09Z"),
        "lastHeartbeat" : ISODate("2020-01-19T17:30:14.204Z"),
        "lastHeartbeatRecv" : ISODate("2020-01-19T17:30:14.681Z"),
        "pingMs" : NumberLong(0),
        "lastHeartbeatMessage" : "",
        "syncingTo" : "",
        "syncSourceHost" : "",
        "syncSourceId" : -1,
        "infoMessage" : "",
        "electionTime" : Timestamp(1579453928, 1),
        "electionDate" : ISODate("2020-01-19T17:12:08Z"),
        "configVersion" : 3
},
```


Figure 3.15: Configuration server on port 20000 (primary)


```
{
        "_id" : 1,
        "name" : "localhost:20001",
        "ip" : "127.0.0.1",
        "health" : 1,
        "state" : 2,
        "stateStr" : "SECONDARY",
        "uptime" : 1100,
        "optime" : {
                "ts" : Timestamp(1579455009, 1),
                "t" : NumberLong(2)
        },
        "optimeDate" : ISODate("2020-01-19T17:30:09Z"),
        "syncingTo" : "localhost:20000",
        "syncSourceHost" : "localhost:20000",
        "syncSourceId" : 0,
        "infoMessage" : "",
        "configVersion" : 3,
        "self" : true,
        "lastHeartbeatMessage" : ""
},
```


Figure 3.16: Configuration server on port 20001 (secondary)

```
{
        "_id" : 2,
        "name" : "localhost:20002",
        "ip" : "127.0.0.1",
        "health" : 1,
        "state" : 2,
        "stateStr" : "SECONDARY",
        "uptime" : 1093,
        "optime" : {
                "ts" : Timestamp(1579455009, 1),
                "t" : NumberLong(2)
        },
        "optimeDurable" : {
                "ts" : Timestamp(1579455009, 1),
                "t" : NumberLong(2)
        },
        "optimeDate" : ISODate("2020-01-19T17:30:09Z"),
        "optimeDurableDate" : ISODate("2020-01-19T17:30:09Z"),
        "lastHeartbeat" : ISODate("2020-01-19T17:30:14.229Z"),
        "lastHeartbeatRecv" : ISODate("2020-01-19T17:30:14.340Z"),
        "pingMs" : NumberLong(0),
        "lastHeartbeatMessage" : "",
        "syncingTo" : "localhost:20000",
        "syncSourceHost" : "localhost:20000",
        "syncSourceId" : 0,
        "infoMessage" : "",
        "configVersion" : 3
}
```


Figure 3.17: Configuration server on port 20002 (secondary)

If we try to shutdown the primary server, we can see that the remaining nodes are still enough to elect a new primary and the application is still available. In the following snapshots of the configuration we can see that the node on port 20001 is the new primary.


```
                {
                        "_id" : 0,
                        "name" : "localhost:20000",
                        "ip" : "127.0.0.1",
                        "health" : 0,
                        "state" : 8,
                        "stateStr" : "(not reachable/healthy)",
                        "uptime" : 0,
                        "optime" : {
                                "ts" : Timestamp(0, 0),
                                "t" : NumberLong(-1)
                        },
                        "optimeDurable" : {
                                "ts" : Timestamp(0, 0),
                                "t" : NumberLong(-1)
                        },
                        "optimeDate" : ISODate("1970-01-01T00:00:00Z"),
                        "optimeDurableDate" : ISODate("1970-01-01T00:00:00Z"),
                        "lastHeartbeat" : ISODate("2020-01-19T17:49:40.070Z"),
                        "lastHeartbeatRecv" : ISODate("2020-01-19T17:49:07.031Z"),
                        "pingMs" : NumberLong(0),
                        "lastHeartbeatMessage" : "Error connecting to localhost:20000 (127.0.0.1:20000) :: caused by ::
Impossibile stabilire la connessione. Rifiuto persistente del computer di destinazione.",
                        "syncingTo" : "",
                        "syncSourceHost" : "",
                        "syncSourceId" : -1,
                        "infoMessage" : "",
                        "configVersion" : -1
                },
```


Figure 3.18: Configuration server on port 20000 after the failure of one node (failed node)

```
{
        "_id" : 1,
        "name" : "localhost:20001",
        "ip" : "127.0.0.1",
        "health" : 1,
        "state" : 1,
        "stateStr" : "PRIMARY",
        "uptime" : 97,
        "optime" : {
                "ts" : Timestamp(1579458352, 1),
                "t" : NumberLong(7)
        },
        "optimeDate" : ISODate("2020-01-19T18:25:52Z"),
        "syncingTo" : "",
        "syncSourceHost" : "",
        "syncSourceId" : -1,
        "infoMessage" : "",
        "electionTime" : Timestamp(1579458301, 1),
        "electionDate" : ISODate("2020-01-19T18:25:01Z"),
        "configVersion" : 3,
        "self" : true,
        "lastHeartbeatMessage" : ""
},
```


Figure 3.19: Configuration server on port 20001 after the failure of one node (new primary)


```
{
        "_id" : 2,
        "name" : "localhost:20002",
        "ip" : "127.0.0.1",
        "health" : 1,
        "state" : 2,
        "stateStr" : "SECONDARY",
        "uptime" : 93,
        "optime" : {
                "ts" : Timestamp(1579458352, 1),
                "t" : NumberLong(7)
        },
        "optimeDurable" : {
                "ts" : Timestamp(1579458352, 1),
                "t" : NumberLong(7)
        },
        "optimeDate" : ISODate("2020-01-19T18:25:52Z"),
        "optimeDurableDate" : ISODate("2020-01-19T18:25:52Z"),
        "lastHeartbeat" : ISODate("2020-01-19T18:25:53.571Z"),
        "lastHeartbeatRecv" : ISODate("2020-01-19T18:25:53.151Z"),
        "pingMs" : NumberLong(0),
        "lastHeartbeatMessage" : "",
        "syncingTo" : "localhost:20001",
        "syncSourceHost" : "localhost:20001",
        "syncSourceId" : 1,
        "infoMessage" : "",
        "configVersion" : 3
}
```


Figure 3.20: Configuration server on port 20002 after the failure of one node (secondary)

If we try to shutdown also the new primary we can see that the replica set is no more able to elect a primary. However, the third node is still available, remains secondary and can accept read requests. Its configuration is shown in the following snapshot. Note that the `syncingTo` field specifies that there is not an available node from which the node can replicate the database.

```
{
        "_id" : 2,
        "name" : "localhost:20002",
        "ip" : "127.0.0.1",
        "health" : 1,
        "state" : 2,
        "stateStr" : "SECONDARY",
        "uptime" : 2920,
        "optime" : {
                "ts" : Timestamp(1579456824, 1),
                "t" : NumberLong(4)
        },
        "optimeDate" : ISODate("2020-01-19T18:00:24Z"),
        "syncingTo" : "",
        "syncSourceHost" : "",
        "syncSourceId" : -1,
        "infoMessage" : "could not find member to sync from",
        "configVersion" : 3,
        "self" : true,
        "lastHeartbeatMessage" : ""
}
```


Figure 3.21: Configuration server on port 20002 after the failure of two nodes (only available node)

If we try to restart again the server on port 20000 we can see that a new primary is elected, in this case the node on port 20002, and the system can start again accepting write requests.


```
{
        "_id" : 0,
        "name" : "localhost:20000",
        "ip" : "127.0.0.1",
        "health" : 1,
        "state" : 2,
        "stateStr" : "SECONDARY",
        "uptime" : 13,
        "optime" : {
                "ts" : Timestamp(1579457258, 1),
                "t" : NumberLong(5)
        },
        "optimeDurable" : {
                "ts" : Timestamp(1579457258, 1),
                "t" : NumberLong(5)
        },
        "optimeDate" : ISODate("2020-01-19T18:07:38Z"),
        "optimeDurableDate" : ISODate("2020-01-19T18:07:38Z"),
        "lastHeartbeat" : ISODate("2020-01-19T18:07:42.451Z"),
        "lastHeartbeatRecv" : ISODate("2020-01-19T18:07:42.179Z"),
        "pingMs" : NumberLong(0),
        "lastHeartbeatMessage" : "",
        "syncingTo" : "localhost:20002",
        "syncSourceHost" : "localhost:20002",
        "syncSourceId" : 2,
        "infoMessage" : "",
        "configVersion" : 3
},
```


Figure 3.22: Configuration server on port 20000 after node restore (secondary)

Figure 3.23: Configuration server on port 20002 after node restore (new primary)

4 Manual of Usage

4.1 Manual of Usage

The users interact with the application through a graphical user interface. In figure 4.1 the first scene of the application is reported.

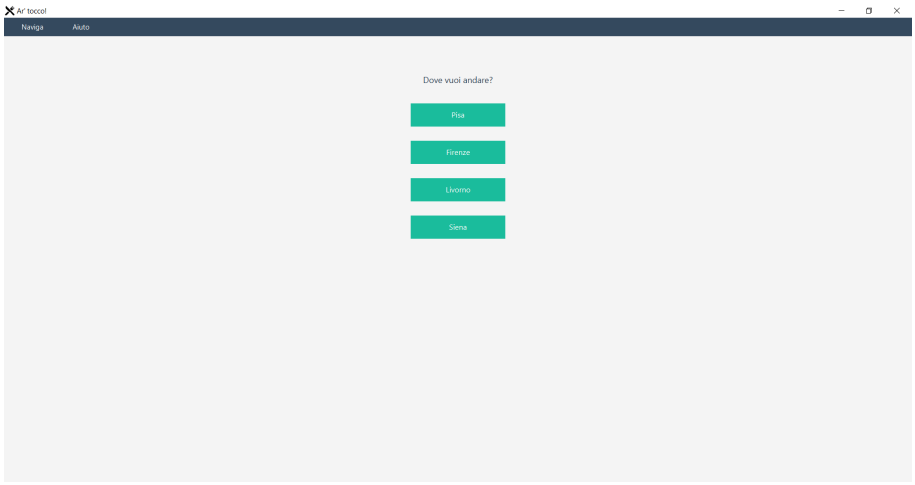


Figure 4.1: First Scene

4.1.1 Customer Manual

A Customer (or a generic user of the application) chooses the city as shown in figure 4.1. A new scene is displayed as reported below in figure 4.2. The restaurants are sorted based on the score of their reviews. The Customer can visualize the information of a restaurant by simply selecting its name in the left panel.

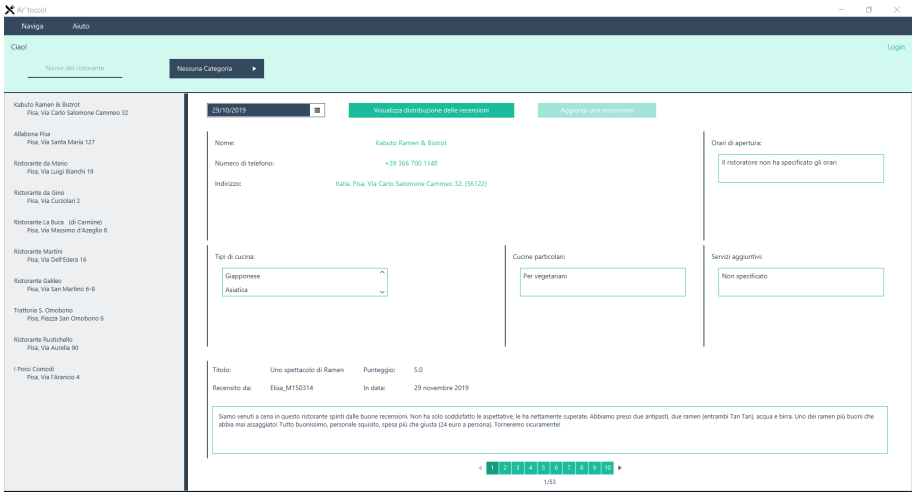


Figure 4.2: Customer Scene

Visualize Reviews Distribution

A Customer can examine a pie chart which describes the distribution of the reviews from a given date to the current date. To do so, she has to select the starting date from the date picker and then click on the `Visualizza distribuzione delle recensioni` button.

Filter Restaurants

A user can search a restaurant by typing the name or selecting the category. It is also possible to filter restaurants by both name and category. The user has to type the name in the text field in the upper part of the scene and to select a category from the drop down menu next to it.

Login

A Customer can login by clicking either on the `Login` label on the top right corner or on the `Login` button in the `Naviga` menu.

Add New Review

After the authentication, the `Aggiungi una recensione` button is enabled and the Customer can insert a new review by clicking on it and filling the form that appears.

Delete Account

The Customer, when logged in, can delete her account selecting the button `Cancella utente` in the `Naviga` menu.

Visualize Past Reviews

The Customer can visualize her past reviews clicking on `Visualizza le mie recensioni`. The application shows the following scene 4.3.

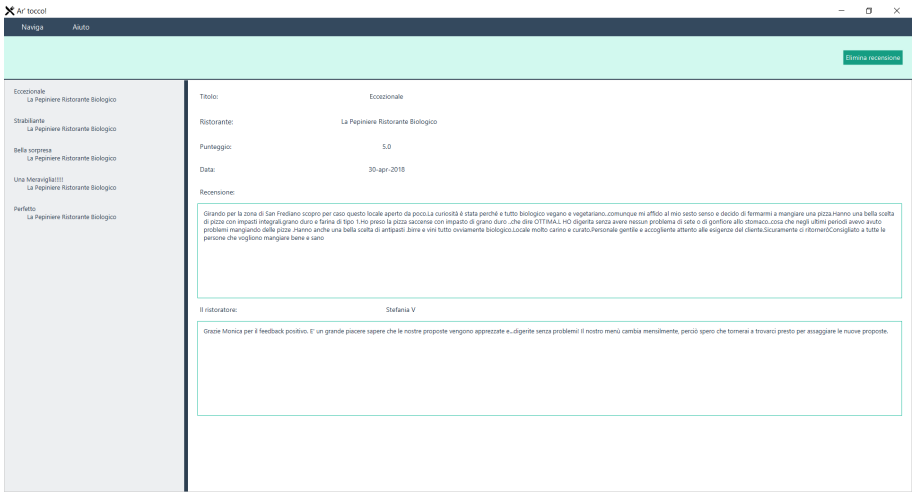


Figure 4.3: Customer Past Reviews Scene

Delete Past Review

The Customer can delete one of her past reviews pressing the `Elimina Recensione` button on the top right corner of the scene, as shown in figure 4.3.

4.1.2 Restaurant Owner Manual

In order to access to her private area, the Restaurant Owner must login with her own credentials, from the scene show in figure 4.1. To login, the Restaurant Owner has to click on the `Naviga` menu, choose the button `Accedi come...` and finally select `Ristoratore`. The application shows the scene in figure 4.4, where the Restaurant Owner can login or register herself.

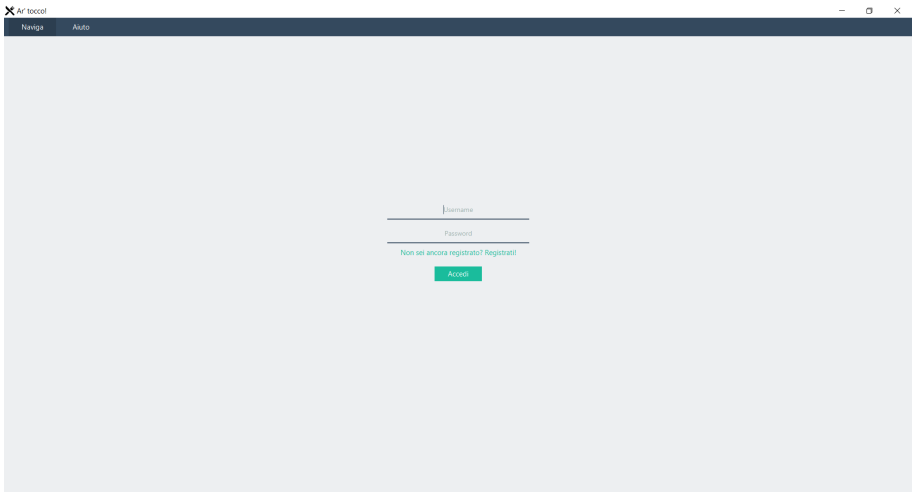


Figure 4.4: Restaurant Owner Login Scene

Visualize Restaurant Owner's Restaurants

After the authentication, in the `I miei ristoranti` section, the Restaurant Owner can visualize the information of her restaurant, as shown in figure 4.5. The Restaurant Owner can select another of her restaurants by clicking on its name in the left panel.

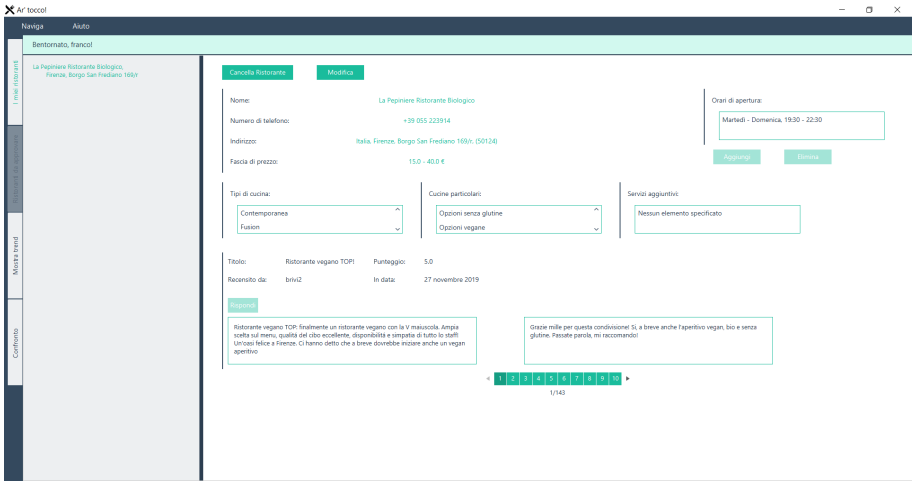


Figure 4.5: Restaurant Information Scene

Delete Restaurant Owner's Account

The Restaurant Owner can delete her account clicking the `Cancella account` button in the `Naviga` menu in the scene shown in figure 4.5.

Delete Restaurant

A Restaurant Owner can delete a restaurant clicking on the button `Cancella Ristorante` as shown in figure 4.5.

Update Restaurant Information

A Restaurant Owner can update the information about one of her restaurants by pressing the `Modifica` button shown in figure 4.5. In this way, the text fields are enabled to let the Restaurant Owner type the updated information, as well as the `Aggiungi` and `Elimina` buttons, which can be used to update the restaurant's opening hours.

Reply to Review

The Restaurant Owner can reply to a review as long as there is not a reply yet. To do so, she has to type the reply in the text area on the right and then press the `Rispondi` button. The latter is disabled if the Restaurant Owner cannot reply. An example is shown in figure 4.5.

Visualize the Information of an Unapproved Restaurant

When a restaurant is not approved yet, the Restaurant Owner can visualize the restaurant information in the section `Ristoranti da approvare`. An example is in figure 4.5.

Show Reviews Trend

The Restaurant Owner can analyze the reviews trend, as shown in figure 4.6. She has to select a starting date through the `Data di partenza` date picker and choose a temporal window from the `Finestra temporale` drop down menu. The trend is computed from the selected date up to the current date and the reviews scores are aggregated according to the chosen window.

Figure 4.6: Reviews Trend Scene

Comparison with Competitors

The Restaurant Owner can analyze the performance of her restaurants with respect to her competitors. Once again, she has to select a starting date through the `Data di partenza` date picker and choose a temporal window from the `Finestra temporale` drop down menu.

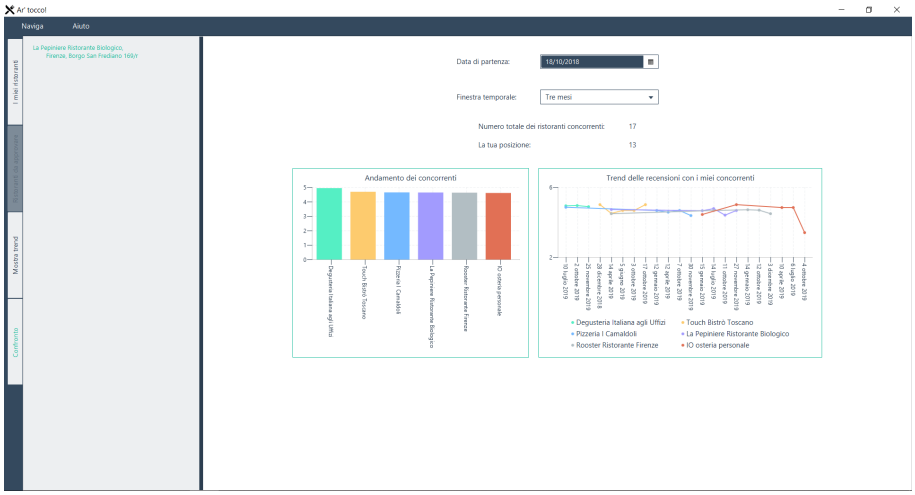


Figure 4.7: Restaurant Comparison Scene

Insert Restaurant

A Restaurant Owner can send a new request to register a restaurant clicking on the button `Aggiungi nuovo ristorante` in the `Naviga` menu. She has to fill the form shown in figure 4.8, select the `Opzionali` section on the left-hand side and then add the information required by the form in figure 4.9.

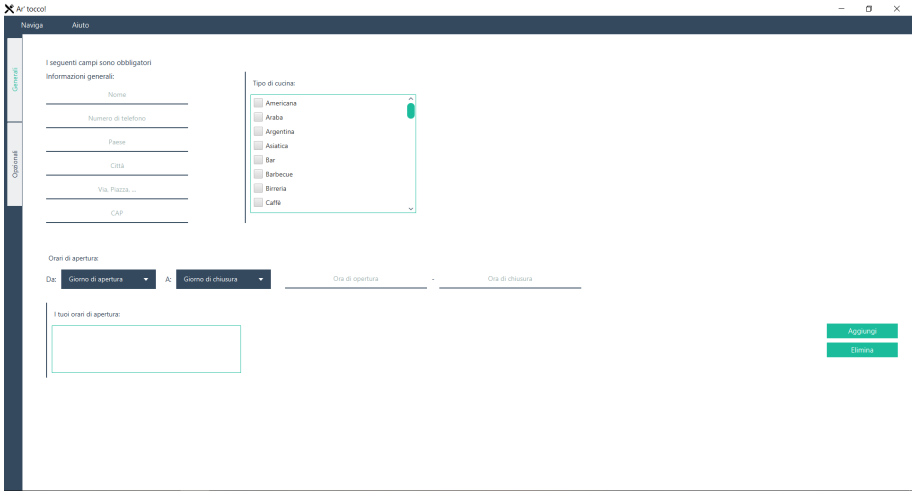


Figure 4.8: Insert New Restaurant Scene

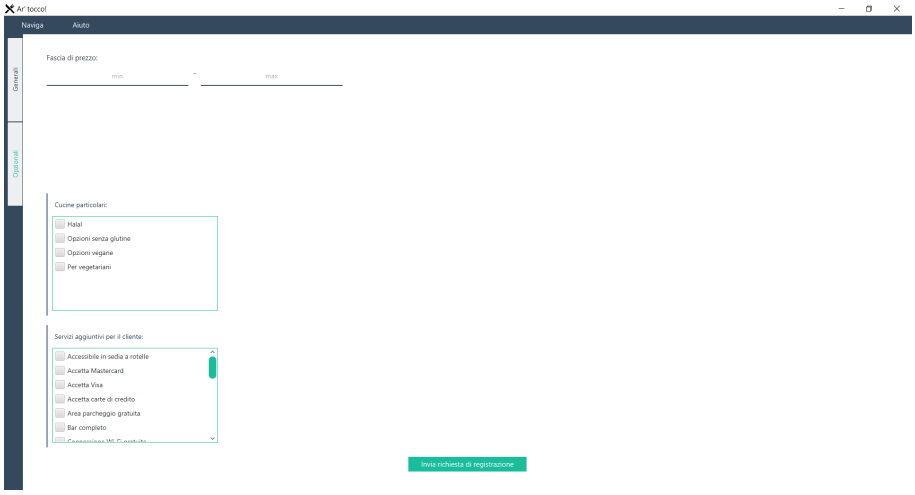


Figure 4.9: Insert New Restaurant Scene

When all the mandatory information are inserted, the Restaurant Owner has to press the `Invia richiesta di registrazione` button.

4.1.3 Administrator Manual

In order to access to his private area, the Administrator must login with his own credentials. He must select `Accedi come...` and then `Amministratore` in the `Naviga` menu, insert username and password and then press the `Accedi` button. The corresponding scene is shown in figure 4.10.

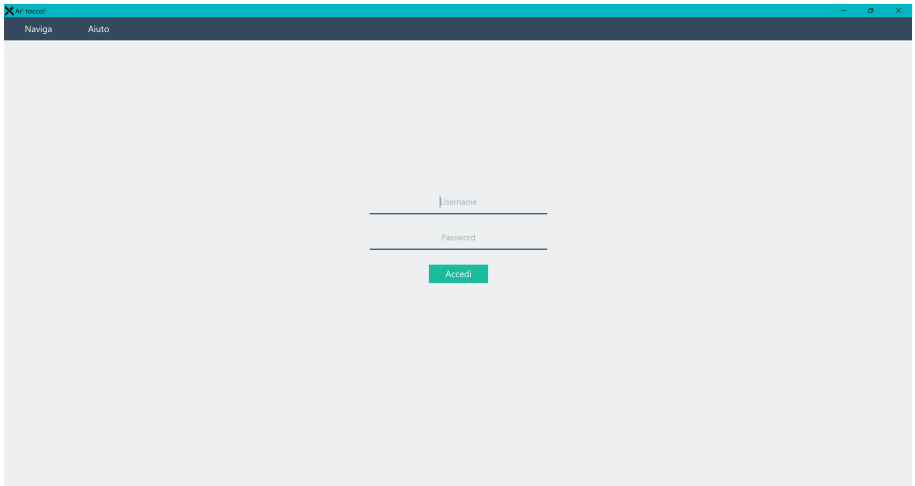


Figure 4.10: Administrator Login Scene

When the administrator is logged, he can perform his operations by selecting the corresponding section on the left side of the visualized scene.

Delete Account

The Administrator can delete a Customer's or a Restaurant Owner's account in the `Cancella Utente` section, which is shown in figure 4.11. He must type the user's username, select the user's type and then press the `Elimina` button.

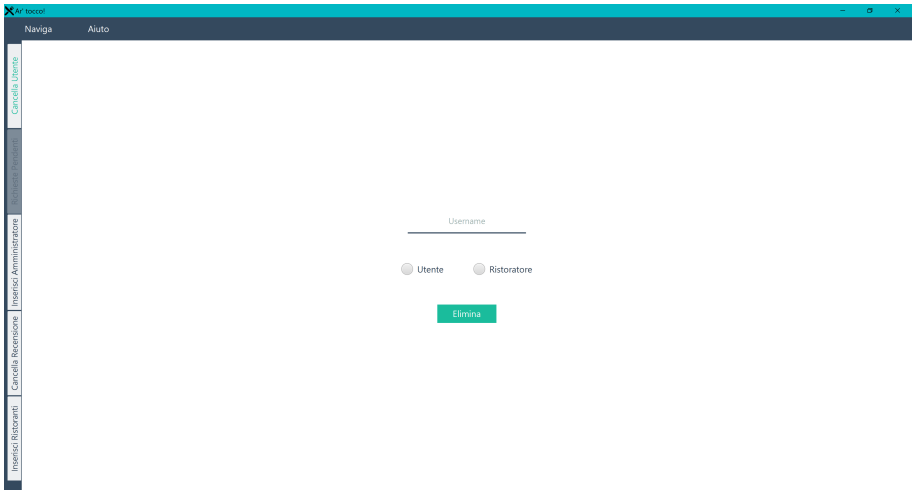


Figure 4.11: Administrator Delete Account Scene

Pending Requests

The `Pending Requests` section is disabled when there are not restaurants to approve or refuse. When there is at least one pending request, instead, the section is enabled and the Administrator can see the list of restaurants that require to be analyzed in the left panel, as shown in figure 4.12. Once a restaurant is selected, the Administrator can check its details, which are displayed on the right panel, and he can refuse or accept it by pressing the `Rifiuta Ristorante` or the `Accetta Ristorante` button.

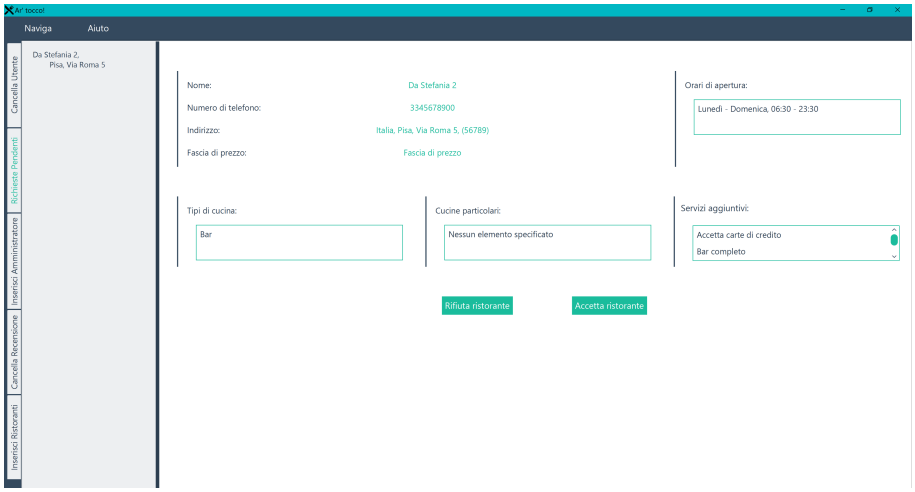


Figure 4.12: Administrator Pending Requests Scene

Add Administrator

The Administrator can also register a new colleague in the `Inserisci Amministratore` section, which is shown in figure 4.13. He must insert the new Administrator's name, surname, username and password and then press the `Inserisci Amministratore` button.

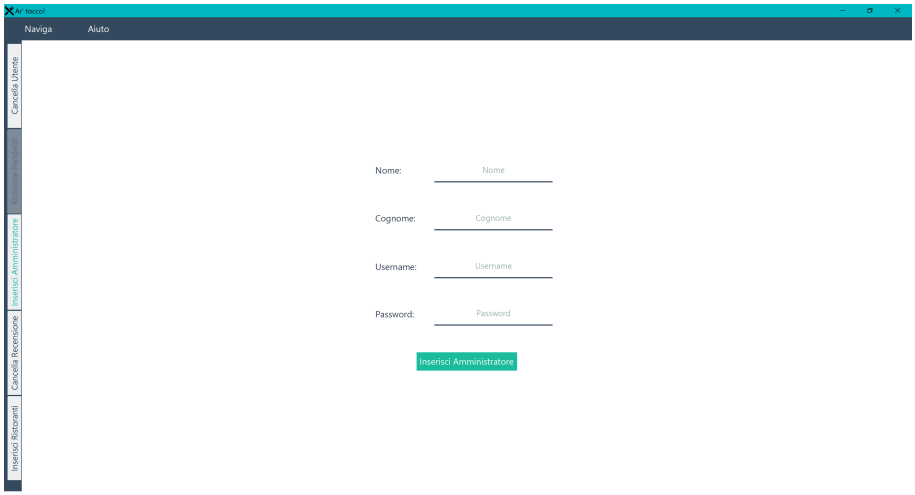


Figure 4.13: Administrator Insert New Colleague

Delete Review

The Administrator can delete the reviews that he considers inappropriate in the `Cancella Recensione` section. As shown in figure 4.14, he must insert the name of the restaurant which received the inappropriate review and the username of the Customer who wrote it. When the `Cerca` button is pressed, the application shows on the left panel the list of reviews that the selected Customer wrote for the selected restaurant. When the Administrator selects one of these reviews, he can visualize its details in the right panel and the `Elimina Recensione` button is enabled. The review can be deleted using this button.

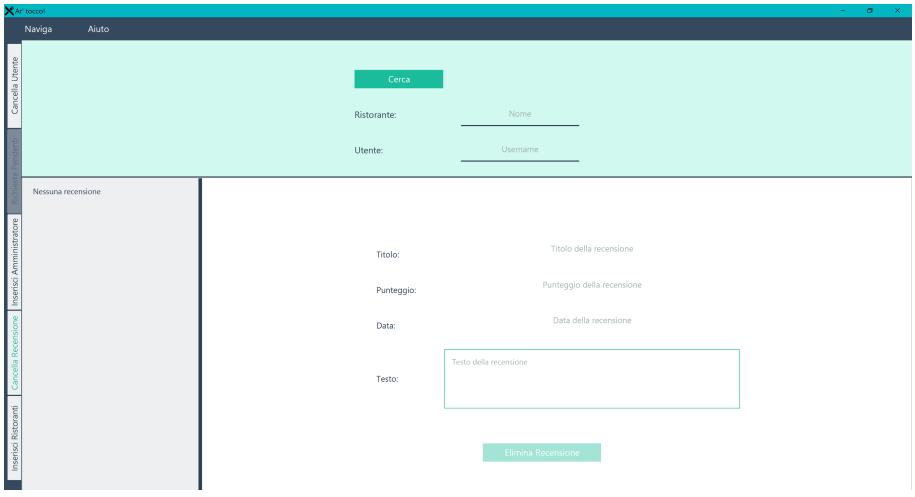


Figure 4.14: Administrator Delete Review Scene

Insert Restaurants

The Administrator can also insert new restaurants in the database in the `Inserisci Ristoranti` section, shown in figure 4.15. First of all, the Administrator must upload the json files in the `./data` folder, then, when he is logged, he can visualize the list of files and select the ones to insert. When the `Inserisci Ristoranti` button is pressed, the selected restaurants are inserted in the database.

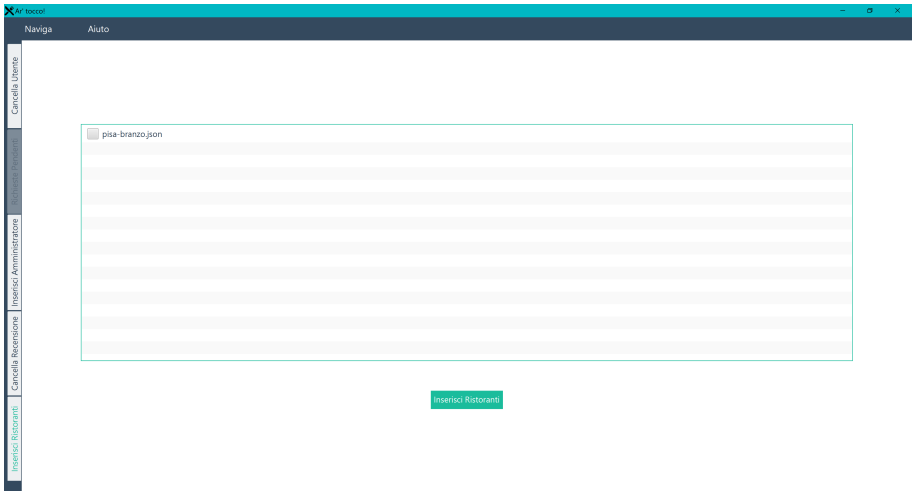


Figure 4.15: Administrator Insert Restaurants