UNIVERSITÀ DI PISA

Large-Scale and Multi-structured Databases

NYSleep

Application Report

Developed by:
Salvatore Patisso
Denny Meini
Domenico D'Orsi

INTRODUCTION

NYSleep is an application for reserving accommodations in the city of New York. Customers can browse all the accommodations presented by renters of New York or search them and decide to reserve them and leave reviews for the accommodation that they reserved. People who have accommodations can rent them and become a renter by uploading information on their accommodation, like amenities, neighborhood, price and also pics about the accommodation.

The application business logic is developed in Java, for the creation of the GUI we used JavaFX and data is stored using MongoDB as a document database and Neo4J as a graph Database. MongoDB contains the main part of the data used in the application like: accommodation details, users account info,review info, reservations info, accommodation reserved, customer who reserved the accommodation ecc. Neo4J is used to map network-like relations that occur between customers who review accommodation that are owned by renters. Graph databases allow us to perform some statistical analysis on these relations in an efficient way.

REQUIREMENTS ANALYSIS

Actors

The main actors involved in this application are all reliable to human interaction with the system, so they are:

- Customer
- Renter
- Admin
- Unregistered User

Functional Requirements

We present the functional requirements for each actor and then it will be shown the use-cases diagram.

Unregistered user's requirements:

- An unregistered user can register by inserting his/her name, email and password.
- An unregistered user can login as a registered user.
- An unregistered user can browse all the accommodations in NY shown in the homepage.
- An unregistered user can find an accommodations in NY by dates, number of rooms, neighborhood, price
- An unregistered user can display information about an accommodation including: name, neighborhood, number of rooms, number of beds, photos, price, number of reviews, rating
- An unregistered user can display reviews about a selected accommodation

- An unregistered user can display informations about the renter of a selected accommodation

Customer's requirements:

- A customer can modify his/her account information like username, password, payment method,profile pic… etc
- A customer can logout.
- A customer can browse all the accommodations in NY shown in the homepage.
- A customer can find accommodations in NY by dates, number of rooms, neighborhood, price.
- A customer can display reviews about a selected accommodation
- A customer can insert a review for an accommodation that must include a rate and may specify a comment.
- A customer can display his/her own reviews
- A customer can display informations about the renter of a selected accommodation
- A customer can reserve, an accommodation for a specific date
- A customer can view his/her own past and active reservations.
- A customer can cancel his/her own reservations.

Renter's requirements:

- A renter can modify his/her account information like username, password,profile pic… etc
- A renter can logout.
- A renter can browse all the accommodations in NY shown in the homepage.
- A renter can find accommodations in NY by dates, number of rooms, neighborhood, price.
- A renter can add an accommodation.
- A renter can view a list of his/her accommodations.

- A renter can remove his/her own accommodations.
- A renter can modify his/her own accommodations.
- A renter can display informations about the renter of a selected accommodation
- A renter can view a list of reservations about his/her own accommodations.
- A renter can cancel reservations about his/her own accommodations.
- A renter can view reviews about his/her own accommodations.

***Admin's requirements**:*

- An admin can modify his/her account information like username, password,profile pic… etc.
- An admin can logout.
- An admin can browse all the accommodations in NY shown in the homepage.
- An admin can find accommodations in NY by dates, number of rooms, neighborhood, price.
- An admin can display information about the renter of a selected accommodation.
- An admin can display information about an accommodations including: name, description, neighborhood, number of rooms, sleeps, photos, price, availability for selected dates, comments and rating, renter's contact.
- An admin can display reservations about an accommodation.
- An admin can cancel a reservation about an accommodation.
- An admin can remove an accommodation.
- An admin can remove reviews.
- An admin can view statistics about accommodations, renters and customers:
    - customer who has spent the most
    - most reserved accommodation for each neighborhood
    - customer with highest average expense
    - most reserving country for neighborhood
    - most and least expensive accommodations for property type
    - average rating by country

- most active user
- renter with most accommodations
- best reviewed renter
- renter with most accommodations for neighborhood
- neighborhood rented by the most number of country
- most reserved accommodation for season

Use cases diagram

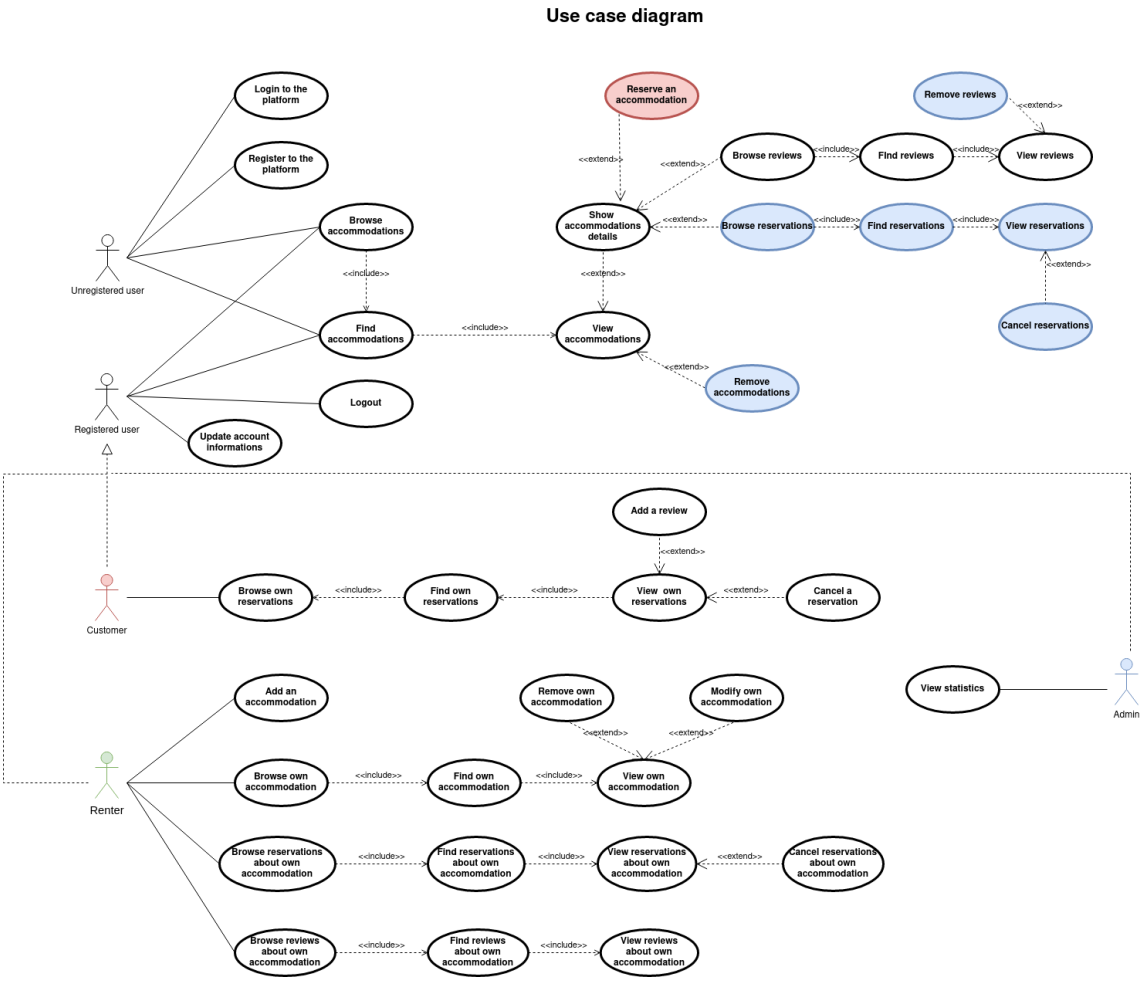


Figura 1

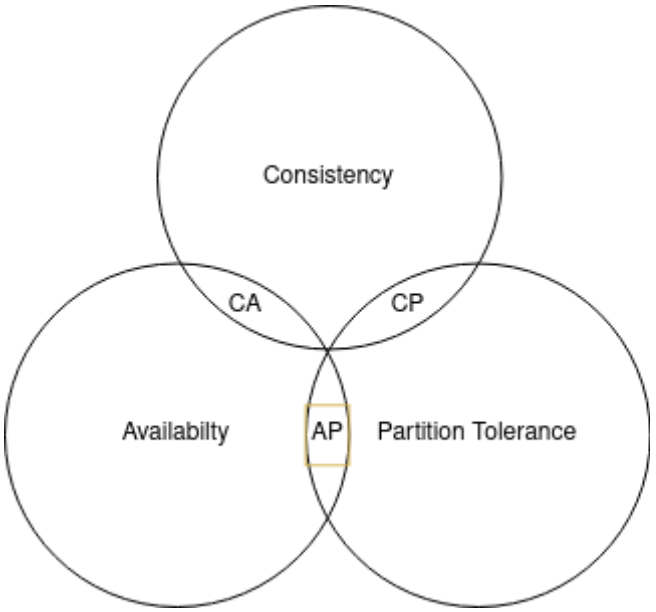
Non-functional requirements

Non functional requirements explain what technical requirements the application must satisfy:

- The application must be implemented in a client-server based structure
- The application must be developed using java technologies
- The code must be readable, easy to understand stratified in different level
- Eventual consistency must always be guaranteed
- GUI must be easy to use for a customer and renter that should use the application
- Interaction between the application and the database must not bring high latency to the application response
- Operation on the database must be atomic
- The system should be available 24/7

Handling CAP theorem issue

For this application, the expectation is to have a lot of read operations, so it's priority is to guarantee high availability and low latency, with a system still available under partitioning. Considering the CAP theorem, the application is more oriented to the AP side of the triangle, favoring Availability (A) and Partition Tolerance (P) in spite of data Consistency (C).
In order to respect the non functional requirements, we decided to guarantee high availability of data, even if an error occurs on the network layer, accepting that the content returned to the user cannot always be consistent in all the replicas. We ensured that all the users will see the same view of the accommodations data: all the customers will read data for accommodation from the primary replica of mongo which is the most recent updated one.



CAP triangle

DATA MODELING

Class analysis

The class analysis for this application's reality lead us to model entity that corresponds to:

- Review
- Accommodation
- Registered user
- Reservation

"Registered user" also must be specialized in different roles (customer, renter, admin) for each we have different relationships to other entities, ex: only a customer can write a review, renter and admin cannot.
Due to this consideration, we can design the UML diagrams like in fig. (2).
As we can see, renter, customer and admin are specialized w.r.t. Registered user for some attribute: for Customer we have address, country, phone which are useful and interesting information to store only for a customer; for what regard a Renter we are interested to store information about his personal contact that can allow customer to contact renter for the accommodation on which they are interested; for Admin we have only one attribute that describe the role that admin cover for the company NySleep.

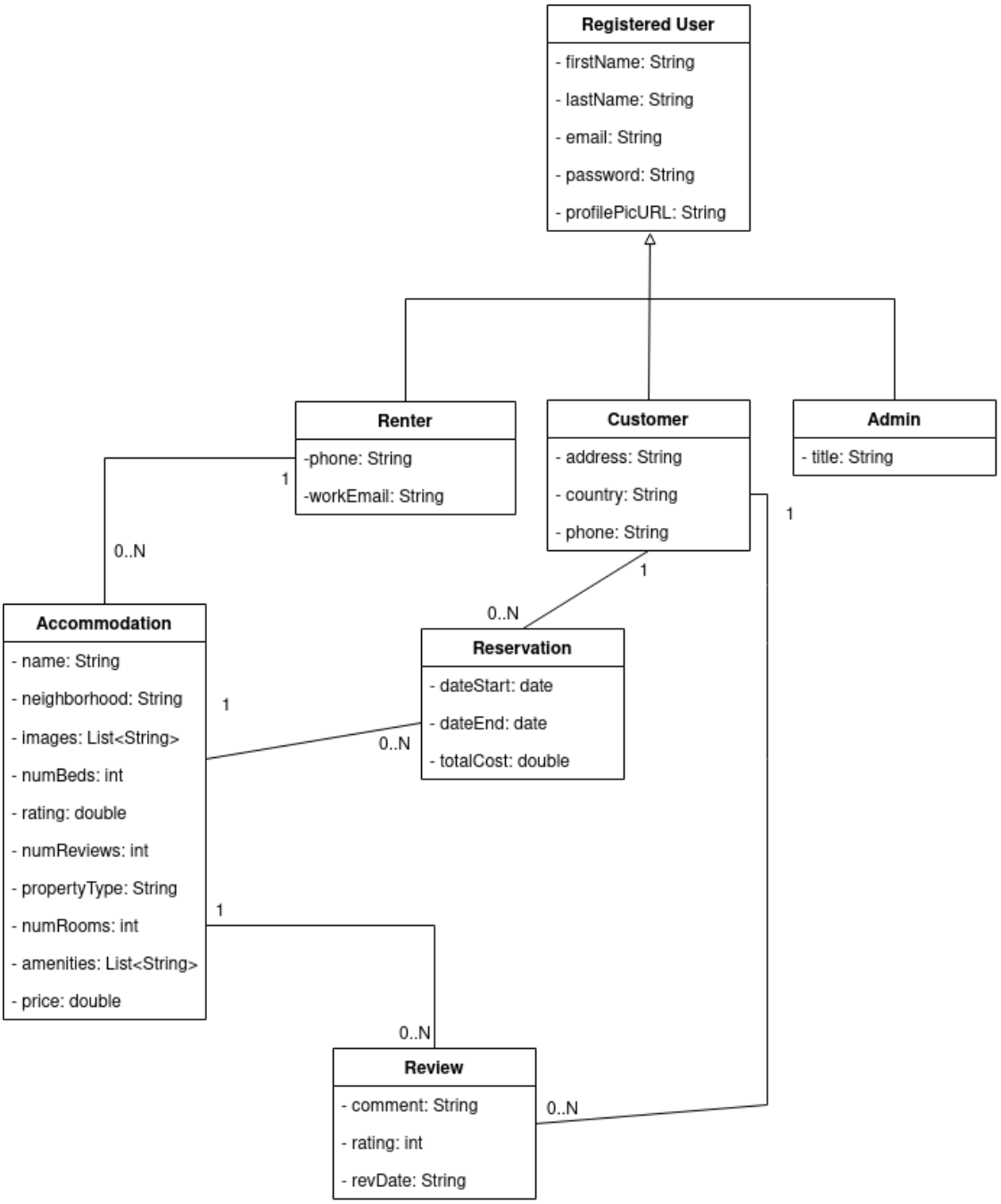


Figure 2

Data model

To model the classes presented in the UML diagram we used two NoSql DBMS: MongoDB as document db and Neo4J as graph DB

MongoDB

We choose to create a collection for every entity and map the relationships within them using nested documents that contain only attributes useful for the queries purposes.
For what regard the registered user and the child classes, we choose to create only one collection for all the documents of registered users. Every document will have, in addition to the common attributes, an attribute that specifies the type of registered user and based on the type there will be other specific attributes related to customer, admin or renter.

```
_id: 12500
first_name: "Ayşe"
last_name: "Nebioğlu"
email: "ayse.nebioglu@example.com"
password: "gerard"
url_profile_pic: "https://randomuser.me/api/portraits/thumb/women/74.jpg"
type: "admin"
title: "analyst"
```

admin document

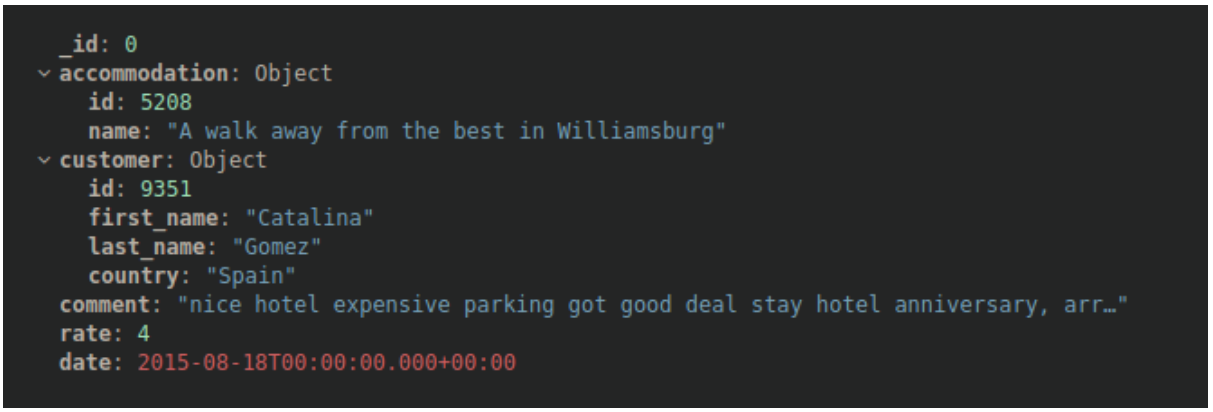
```
_id: 10000
first_name: "Ömür"
last_name: "Kocabıyık"
email: "omur.kocabiyik@example.com"
password: "reddevil"
url_profile_pic: "https://randomuser.me/api/portraits/thumb/women/27.jpg"
type: "renter"
phone: "(742)-470-2654"
work_email: "Ömür.Kocabıyık@NYSleep.com"
```

renter document

```
_id: 1
first_name: "Charlie"
last_name: "Sanchez"
email: "charlie.sanchez@example.com"
password: "yogibear"
url_profile_pic: "https://randomuser.me/api/portraits/thumb/men/13.jpg"
type: "customer"
address: "1938 Rue Gasparin"
country: "France"
phone: "01-19-73-33-93"
```

customer document

All documents in the reviews collection contain attributes relative to a specific review such as comment,rate and date. We also choose to insert nested documents related to the reviewed accommodation: only the id and the name for the accommodation because queries that are performed on reviews in mongoDB needs only this information.There is no need to put all the accommodation information, in this way we can limit the amount of redundancy.

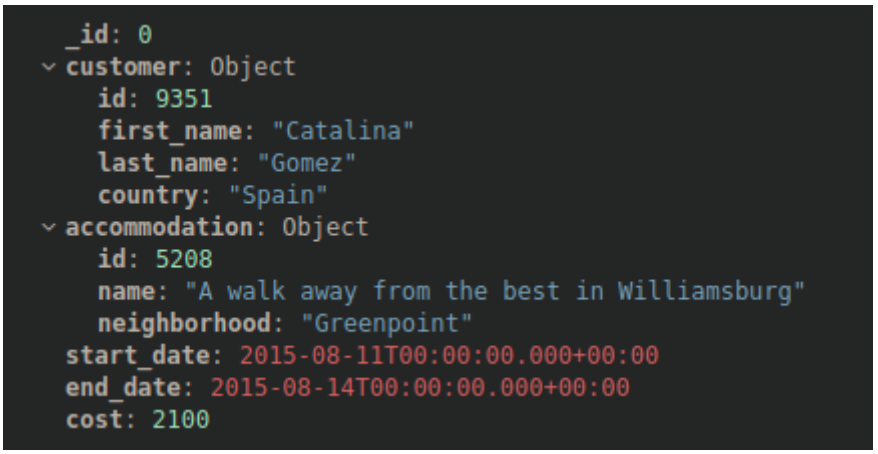


review document

Similarly for the customer nested document we put only info about first/last name and country to easily and efficiently perform queries and aggregations like getting the average rating by country.

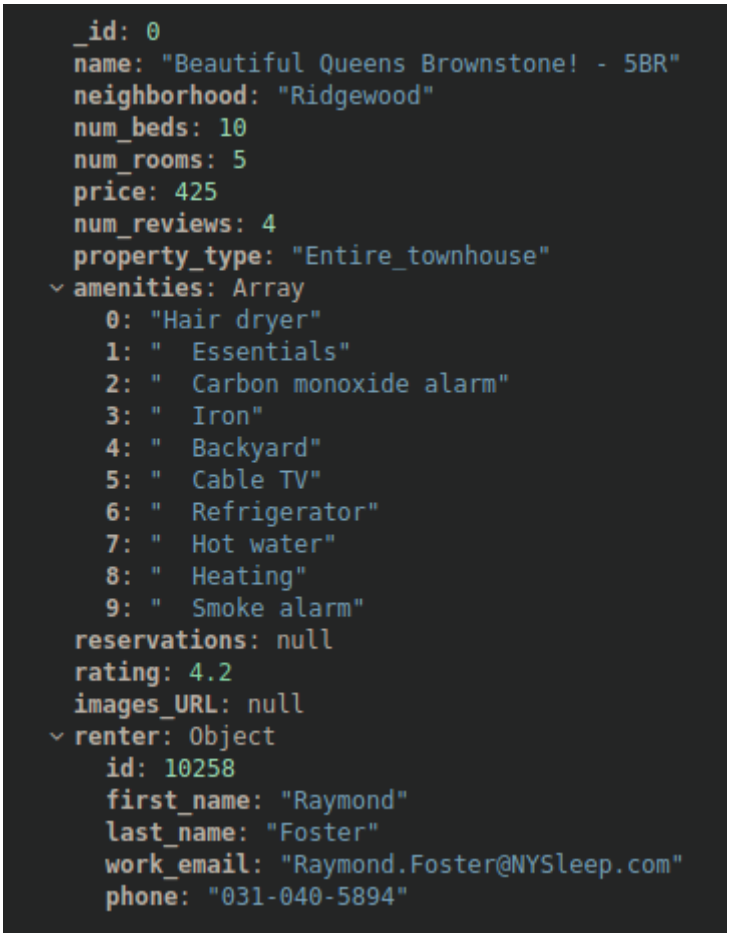
Regarding reservations documents, we can easily see that the nested documents are similar to review documents. In addition we added a neighborhood attribute to accommodation to perform analytics like most reserving country for each neighborhood. Other analytics required for the reservation are:

- Customer who spent the most
- Customer with the high average expense
- Most reserved accommodation for each neighborhood



reservation document

In the end there is the accommodation document which look like this:



accommodation document

Renter information in the nested documents can be interesting for a customer who wants to reserve an accommodation.
Reservations array is an array of documents that contains only info about the start date and the end date for each reservation on the accommodation. It is used to search accommodation based on the availability for a specific date.

MongoDB replica

We have three replicas of mongodb, one of which is the primary replica that acts as the server that takes client requests. They run on three virtual machines provided by the university of Pisa.
As we said previously we choose to pick the AP side of the triangle,for this reason the writeConcern option is set to 'majority': it writes on the majority of the replicas, always including the primary replica. Regarding the readConcern option, it is set to "nearest" except for reading operations on the accommodation, for which we impose the option "primary".
In the case of failure of the primary node, one of the two secondary ones is elected as the new primary one.

```
lsmdb [primary] NYSleep> db.getMongo().getReadPref()
ReadPreference {
  mode: 'nearest',
  tags: undefined,
  hedge: undefined,
  maxStalenessSeconds: undefined,
  minWireVersion: undefined
}
```

Read concern

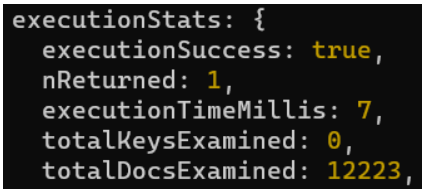
```
lsmdb [primary] NYSleep> db.adminCommand({getDefaultRWConcern: 1 })
{
  defaultReadConcern: { level: 'local' },
  defaultWriteConcern: { w: 'majority', wtimeout: 0 },
  defaultWriteConcernSource: 'implicit',
  defaultReadConcernSource: 'implicit',
  localUpdateWallClockTime: ISODate("2023-02-06T10:48:16.601Z"),
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1675680489, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1675680489, i: 1 })
```

Write Concern

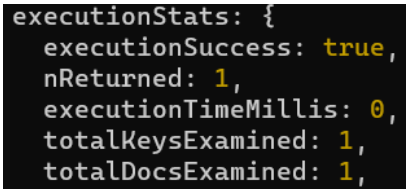
MongoDB indexes

We choose to put indexes to improve the efficiency of some read operations and queries that are frequently performed by the system like:

- ➔ Login of users: email of the collection users are indexed this result in the following improvement of performance

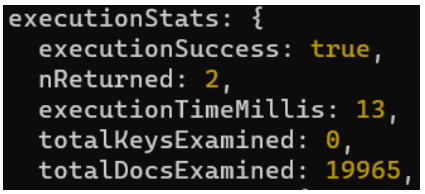


Without index

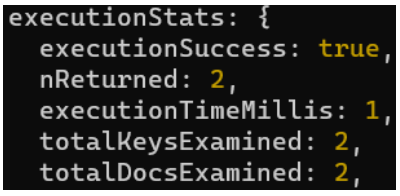


With index

Show reviews of an accommodation: id of accommodation are indexed in the reviews collection because this query is expected to be frequent given that customers always want to consult reviews for an accommodation.

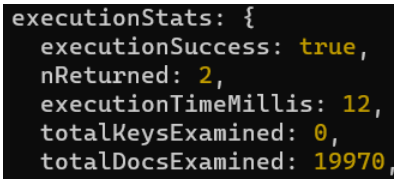


Without index

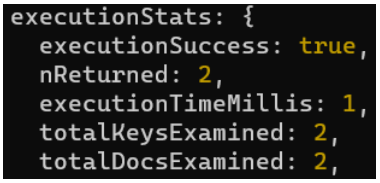


With index

Show reservations for an accommodation: we indexed the id of accommodation in the collection of the reservations



Without index



With index

Search accommodation for a specific date: for this query we create two index on start_date and end_date of the nested document reservation in the accommodations collection

```
executionStats: {
  executionSuccess: true,
  nReturned: 15568,
  executionTimeMillis: 107,
  totalKeysExamined: 0,
  totalDocsExamined: 16479,
```

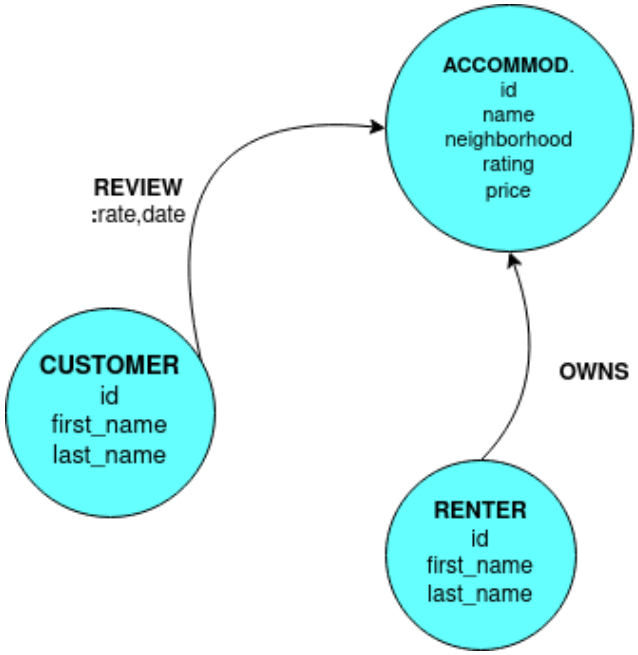

Without index


```
executionStats: {
  executionSuccess: true,
  nReturned: 15568,
  executionTimeMillis: 95,
  totalKeysExamined: 15570,
  totalDocsExamined: 15568,
```


With index

Neo4J

The structure of the graph DB is represented in the following figure:



GraphDB structure

We use graphDB for queries that are easily implemented and efficiently performed in a network structure like this one. To avoid creating a too heavy graph db we choose to represent the entities with a reduced number of attributes (we keep only the attribute used in queries).

On the graph we calculate the average rating of each accommodation by considering only the reviews written in the last year. All the reviews are used in analytics such as “find the best reviewed renter” or “find the most active user”

The graph is not so affected by the probable high number of reviews because they are represented as edges in a simpler form(only date and rate, comments are useless for the purposes of the GraphDB).

An important queries that we can perform with Neo4J is suggesting an accommodation for a customer. It is based on a plugin of Neo4J called GDS(Graph Data Science) that allows us to define a similarity measure between customers. It suggests accommodations based on similarity between customers and filters them based on the rate of the review written by similar customers(i.e. greater or equal than 4).

Sharding proposal

Sharding in MongoDB with this low amount of data is not convenient, but we can still present proposals for future implementations.

Collection	Sharding key	Note
Users	_id	Hash-based
Reviews	date	Range-based on couple of year
Reservations	start_date,end_date	Range-based on a year
Accommodations	_id	Hash-based

CRUD

MongoDB CRUD operations

Users collection

Read Operations	**Query**
Get User Information	db.users.find({_id: $eq:{*id*} })
Get email	db.users.find({email: $email })

Create Operations	**Query**
Create user	db.users.insertOne(document)

Update Operations	**Query**
Modify account info	db.users.updateOne({_id:{$eq:*id*}} , doc)

Delete Operations	**Query**
Delete user	db.users.deleteOne({_id:{$eq: *id*} })

Accommodations collection

Create operations	**Query**
Create accommodation	db.accommodations.insertOne(document)

Read operations	**Query**
Get Accommodations HomePage	db.accommodations.find()
Get Accommodation	db.accommodations.find({_id:{$eq: *id* }})

Get searched accommodation	db.accommodations.find({$and:[{ $or: [{ "reservations.start_date": { $not: { $lte: ISODate("2023-04-20T00:00:00.000Z") } } }, { "reservations.end_date": { $not: { $gte: ISODate("2023-04-15T00:00:00.000Z") } } }] }, {neighborhood: "South Slope"}, {num_beds: 2}, {price: {$lt: 200}}]})
Get searched acc. for a renter	db.accommodations.find({“renter.id”:{ $eq:*renterId*}})

Update operations	**Query**
Update accommodation	db.accommodations.updateOne({_id:{$eq:*id* }},{$set: *newDoc*})
Update accommodation renter	db.accommodations.updateOne({_id:{$eq:*id* }},{$set: {renter: *newRenterDoc*}})
Update rating	db.accommodations.updateOne({_id:{$eq:*id* }},{$set:{rating: *newRating*}})
Decrease number of review	db.accommodations.updateOne({_id:{$eq:*id* }},{$inc: {num_reviews:-1}})
Increment number of review	db.accommodations.updateOne({_id:{$eq:*id* }},{$inc: {num_review:1}})
Insert reservation	db.accommodations.updateOne({_id:{$eq:*id*}}, {$push:*resDoc*})
Delete reservation	db.accommodations.updateOne({_id:{$ eq:*id*}}, {$pull:*resDoc*})

Delete operations	**Query**
Delete accommodation	db.accommodations.deleteOne({_id:{$e

	q:*id*}})

Reservations collection

Create operations	**Query**
Create reservation	db.reservations.insertOne(*resDoc)*

Read operations	**Query**
Get customer reservations	db.reservations.find({“customer.id”:{$eq:id}})
Get accommodation reservations	db.reservations.find({“accommodations.id”:{$eq:id}})

Update operations	**Query**
Modify reservation	db.reservations.updateOne({_id:{$eq:*id*}}, {$set:{*newDoc*}})

Delete operations	**Query**
Delete reservation	db.reservations.deleteOne({_id:{$eq:*id*}})

Review collection

Create operations	**Query**
Create review	db.reviews.insertOne(*revDoc*)

Read operations	Query
Get reviews for acc	db.reviews.find({“accommodation.id”:{$eq:*id*}})
Get reviews for customer	db.reviews.find(“customer.id:{$eq:*id*}}”)

Update operations	Query
Modify review	db.reviews.updateOne({_id:{$eq:*id*}} ,{$set:newDoc})

Delete operations	Query
Delete reservation	db.review.deleteOne({_id:{$eq:*id*}})
Delete accommodation review	db.review.deleteOne({“accommodation.id:{$eq:*id*}})
Delete customer review	db.review.deleteOne({“customer.id”:{$eq:*id*}})

Neo4J CRUD Operations

Customers

Read operations	Query
Get User Information	CREATE (cc: customer {id: *id*, first_name: *firstName*, last_name: *lastName*})
Most active user	MATCH (cc:customer)-[r:REVIEWS]->(a:accommodation) RETURN cc.id AS id, cc.first_name AS first_name, cc.last_name AS last_name, COUNT(r) as num_reviews ORDER BY num_reviews DESC LIMIT 1
Get suggested accommodation	MATCH

	(cc:customer)-[r:REVIEWS]->(aa:accommodation)<-[o:OWNS]-(rr:renter)-[so:OWNS]->(sa:accommodation) WHERE r.rate>3 AND cc.id=$id RETURN sa

Create operations	**Query**
Create user	CREATE (cc: customer {id: *id*, first_name: *firstName*, last_name: *lastName*})

Update operations	**Query**
Modify account info	MATCH (cc: customer {id: *oldId* }) SET cc.first_name = *newFirstName*, cc.last_name = *newLastName*

Delete operations	**Query**
Delete user	MATCH (cc: customer {id: *id* }) DELETE cc

Renter

Read operations	**Query**
Get User Information	CREATE (rr: renter {id: *id*, first_name: *firstName*, last_name: *lastName*})
Renter with most accommodation	MATCH (rr:renter)-[o:OWNS]->(a:accommodation)RETURN rr.id AS id, rr.first_name AS first_name, rr.last_name AS last_name,COUNT(o) as num_accommodations ORDER BY num_accommodations DESC LIMIT 1
bestReviewedRenter	MATCH (cc:customer)-[r:REVIEWS]->(aa:accommodation)<-[o:OWNS]-(rr:renter) WITH COUNT(r) as num_reviews, rr,

	AVG(r.rate) as avg_rate WHERE num_reviews >= *lim* RETURN rr.id AS id, rr.first_name AS first_name, rr.last_name AS last_name, avg_rate ORDER BY avg_rate DESC LIMIT 1
Renter with most accommodation for neighborhood	MATCH (rr:renter)-[o:OWNS]->(aa:accommodation) WHERE aa.neighborhood= *neighborhood* RETURN rr.id AS id, rr.first_name AS first_name, rr.last_name AS last_name, COUNT(o) as num_accommodations_neighborhood ORDER BY num_accommodations_neighborhood DESC LIMIT 1

Create operations	**Query**
Create user	CREATE (rr: renter {id: *id*, first_name: $firstName, last_name: *lastName*})

Update operations	**Query**
Modify account info	MATCH (rr: renter {id: *oldId*}) SET rr.first_name = *newFirstName* rr.last_name = *newLastName*

Delete operations	**Query**
Delete user	MATCH (rr: renter {id: *id* }) DELETE rr

Accommodation

Read operations	**Query**
Show renter accommodation	MATCH(rr:renter)-[o:OWNS]->(aa:accommodation) WHERE rr.id= *id* RETURN aa.id AS id, aa.name AS name, aa.neighborhood AS neighborhood,

	aa.rating AS rating
Show suggested accommodation	MATCH (cc:customer)-[r:REVIEWS]->(aa:accommodation) WHERE cc.id=*id2* AND r.rate>=4 RETURN aa.id AS id, aa.name AS name, aa.neighborhood as neighborhood, aa.rating as rating
Recompute rate	MATCH (aa:accommodation)<-[r:REVIEWS]-() WHERE aa.id=*id* AND r.date>date({year: *year*, month: *month*, day: *day*}) RETURN AVG(r.rate) AS rate
Show accommodation of liked renter	MATCH (cc:customer)-[r:REVIEWS]->(aa:accommodation)<-[o:OWNS]-(rr:renter)-[so:OWNS]->(sa:accommodation) WHERE r.rate>3 AND cc.id=*id* RETURN sa.id AS id, sa.name AS name, sa.neighborhood as neighborhood, sa.rating as rating

Create operations	**Query**
Create accommodation	CREATE (aa:accommodation {id: *id*, name: *name*, neighborhood: *neighborhood*, rating: *rating*})

Update operations	**Query**
Update rating	MATCH (aa: accommodation {id: *id* }) SET aa.rating = *rating*
Update accommodation	MATCH (aa: accommodation {id: *oldId* }) SET aa.name = *newName*

Delete operations	**Query**
Delete accommodation	MATCH (aa: accommodation {id: *id* }) DELETE aa

Review

Create operations	**Query**
Create review	MATCH (cc:customer) WHERE cc.id = *idc* MATCH (aa:accommodation) WHERE aa.id = *ida* CREATE (cc)-[:REVIEWS {rate: *rate*, date: *date*}]->(aa)

Delete operations	**Query**
Delete review	MATCH (cc:customer { id: *idc*})-[r:REVIEWS]->(aa:accommodation { id: *ida*}) DELETE r

Aggregations

Name	**Query**

Most expensive and least expensive accommodation for property type	db.accommodations.aggregate(
	[
	{ $project: { "id": 1, "name": 1,"neighborhood":1 ,"property_type":1, "price":1} },
	{$sort: {"price": -1}},
	{$group:
	{
	_id: "$property_type",
	least_expensive: {$last:"$name"},
	lowest_cost: {$last:"$price"},
	least_expensive_neigh: {$last: "$neighborhood"},
	most_expensive_name: {$first:"$name"},
	most_expensive_neigh: {$first:"$neighborhood"},
	highest_cost: {$first:"$price"}
	}
	}
	]
	)

Cust who has spent the most	db.reservations.aggregate([{$group: {_id:{ neighborhood : "$accommodation.neighborhood", acc: "$accommodation.id", accName:"$accommodation.name"} , numRes: {$sum:1}}}, {$sort: {numRes:-1}}, {$group:{_id: "$_id.neighborhood",most_res_acc:{$fir st:"$_id.accName"},num_reservation:{$f irst: "$numRes"}} }])

Neighborhood rented by most number of countries	db.reservations.aggregate(
	
	[
	{$group:
	{
	_id:{neighborhood:
	"$accommodation.neighborhood",
	country: "$customer.country"}
	}
	},
	{$group:
	{
	_id:{neighborhood:
	"$_id.neighborhood"}, num_countries:
	{$sum: 1}
	}
	},
	{$sort:
	{num_countries: -1}
	},
	{$limit:1}
	]
	)

Most reserved acc. for each neighborhood	db.reservations.aggregate([{$group: {_id:{ neighborhood : "$accommodation.neighborhood", acc: "$accommodation.id", accName:"$accommodation.name"} , numRes: {$sum:1}}}, {$sort: {numRes:-1}}, {$group:{_id: "$_id.neighborhood",most_res_acc:{$first:"$_id.accName"},num_reservation:{$first: "$numRes"}} }])
Customer with highest avg expense	db.reservations.aggregate([{$group: {_id:{cust_id:"$customer.id",first_name:"$customer.first_name",last_name:"$customer.last_name",country:"$customer.country"}, avg_cost: { $avg: "$cost" }, num_res:{$sum:1} } }, {$match: {num_res:{$gt:5}}}, { $sort: {"avg_cost": -1 }}, { $limit: 1}])

Most reserved country for neigh.	db.reservations.aggregate(
	[
	{$group: {
	_id:{ neighborhood :
	"$accommodation.neighborhood",
	country: "$customer.country"},
	num_res: {$sum:1}
	}
	},
	{$sort: {num_res:-1}},
	{$group:
	{
	_id: "$_id.neighborhood",
	
	most_res_country:{$first:"$_id.country"},
	num_reservation:{$first:
	"$num_res"}
	}
	}
	]
	)

Most reserved accommodation for season	db.reservations.aggregate(
	[
	{
	$project:
	{
	acc: "$accommodation.id",
	month: { $month: "$start_date" },
	neighborhood:
	"$accommodation.neighborhood",
	accName: "$accommodation.name"
	}
	},
	{
	$match:
	{
	$or:
	[{ "month": 12 }, { "month": 1 }, {
	"month": 2 }]
	}
	},
	{
	$group:

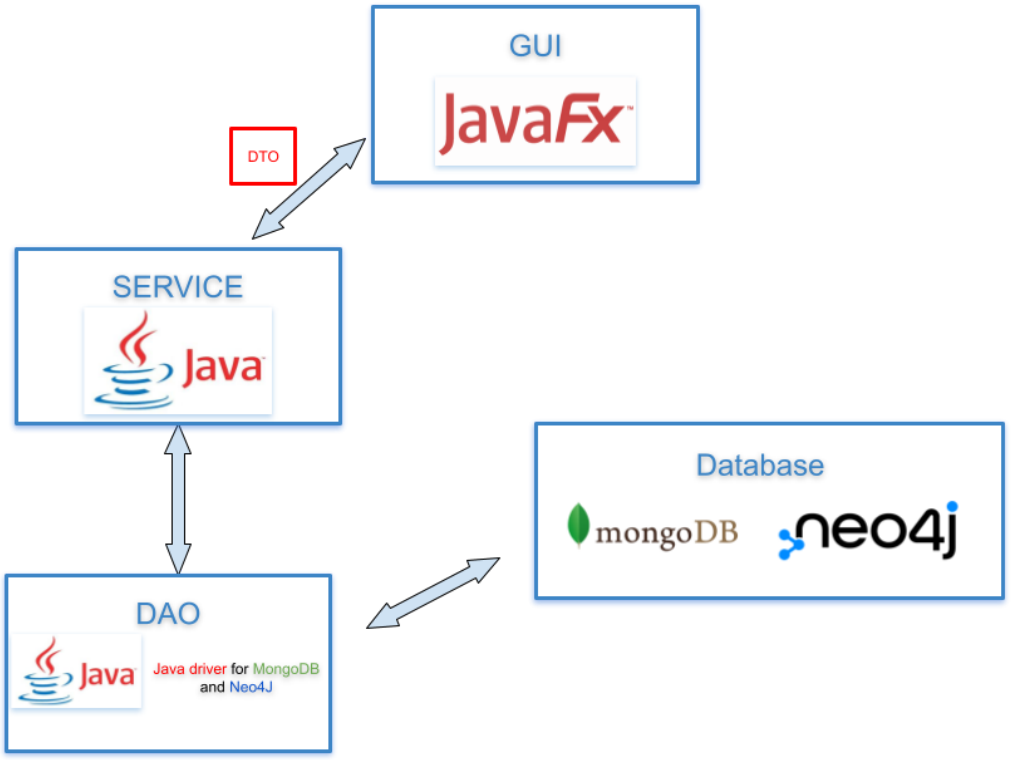
	{
	_id: "$acc",
	num_res: {$sum:1},
	accommodation:
	{$addToSet:{name: "$accName",
	neighborhood: "$neighborhood"}}
	}
	},
	{
	$sort:
	{num_res:-1}
	},
	{$limit: 1}
	]
	)

Average rating by country	db.reviews.aggregate(
	[
	{ $group:
	{
	_id: { country: "$customer.country"},
	average_rate: { $avg: "$rate" }
	}
	}
	]
	)

IMPLEMENTATION

System architecture

The application is implemented in a client-server structure. On the client side we have GUI that allow users to interact with the Service layer. On the other side, Service contains all methods to manage the write and read operations based on the business logic constraint. All CRUD operations are defined in the Data Access Object (DAO) layer, on which the access to the several collections in Mongo and entities in Neo4J is defined. The architecture structure is the following:



Database population

Users information

User information was created through https://randomuser.me. Some users have been turned into renters, others into admins and others into customers. These informations was then saved in both the Document Database and the Graph Database

Accommodation Information

Accommodation info are retrieved from http://insideairbnb.com/get-the-data/ and by preprocessing this data we retrieved only the useful information like: "id", "name", "neighborhood_cleansed", "beds", "bedrooms", "price", "number_of_reviews", "property_type", "amenities" and as property type we extracted only this nine: 'Entire apartment', 'Private room in apartment', 'Private room in house', 'Private room in townhouse', 'Entire condominium', 'Entire house', 'Entire loft', 'Entire townhouse', 'Entire rental unit'.

Reservation Information

Reservation info is generated artificially using the reviews. Assuming that a review can exist only if there is a related reservation, we created the reservations for each review by setting the start_date as one week before the reviews date and the end date as four days before the review date.

Review Information

Review info is retrieved from a kaggle dataset and then associated artificially with generated customers
https://www.kaggle.com/datasets/andrewmvd/trip-advisor-hotel-reviews?resource=download

Java

We have structured the server project in different packages such as:

- DTO (Data Transfers Object): object used to transfer data from the GUI to the Service Layer and vice versa
- DAO (Data Access Object): Classes that describe the connection and access to the two different databases. They also implement all the CRUD operations, aggregations and queries used in the application.
- Business: here are present all the classes used to define the use cases and manage the business logic.
- RMI (Remote Method Invocation)
- Model: Classes that map a representation of the entities of the databases as java objects.

For what regard DAO classes we can notice that they are divided into MongoDB DAO and Neo4JDAO. Both are based on two base classes that define the basic operations used in the DAO for each entity, like: reads documents, update documents, write a document, define connection, define db, define session etc..
Each DAO implements all the CRUD operations, Aggregations and queries required by the system.

For what regard Business classes we can notice that each class is composed of methods that represent the use cases for each actor. All the classes are extended to UserService.java which implements all the common methods between users.
Every use case that requires write operations on both the databases use the mongodb java driver session which allows us to start, commit and (if it's necessary) abort a transaction. In this way we guarantee consistency between the neo4j and MongoDB common entities: every time the application requires writing in both DBs the business starts a session that allows the execution of an abort operation in the case a writing operation in MongoDB or in Neo4J will fail.
Every reading operation also is managed by the business transferring only the required data by parsing documents or records obtained from the DAO objects.

Read operation example:

We take as an example the method showSearchedAcc. This method use "getSearchedAcc" method of MongoAccommodationDAO.java that get all the accommodation from the collection of accommodations which respect some search filter such as: available in range between "start_date" and "end_date", number of people that can accommodate, number of rooms available and so on. This method also checks if the user has specified all the necessary parameters.

```
public List<Document> getSearchedAcc(LocalDate startDate,LocalDate endDate,int numPeople,String neighborhood,
                                      double price, int skip, int limit) throws BusinessException {
    Document searchQuery;

    if(startDate == null || endDate == null){
        throw new BusinessException("Select start date and end date");
    }

    if(numPeople == 0){
        throw new BusinessException("Select a number of people");
    }

    if(startDate.isAfter(endDate)){
        throw new BusinessException("Start date must come before end date");
    }

    if(price < 0){
        throw new BusinessException("Maximum price must be a positive value");
    }
```

Then it proceeds to construct the search query based on what additional parameters are specified

```
try {
    if (neighborhood.equals("") && price == 0) {
        searchQuery = new Document("$and", Arrays.asList(new Document("$or", Arrays.asList(new Document("reservations.start_date",
                                new Document("$not",
                                        new Document("$lte", endDate))),
                        new Document("reservations.end_date",
                                new Document("$not",
                                        new Document("$gte", startDate))))),
                new Document("num_beds", numPeople)));

    } else if (neighborhood.equals("")) {
        searchQuery = new Document("$and", Arrays.asList(new Document("$or", Arrays.asList(new Document("reservations.start_date",
                                new Document("$not",
                                        new Document("$lte",
                                                endDate))),
                        new Document("reservations.end_date",
                                new Document("$not", new Document("$gte", startDate))))),
                new Document("num_beds", numPeople),
                new Document("price",
                        new Document("$lt", price))));
    }
    else if (price == 0) {
        searchQuery = new Document("$and", Arrays.asList(new Document("$or", Arrays.asList(new Document("reservations.start_date",
                                new Document("$not",
                                        new Document("$lte",
                                                endDate))),
                        new Document("reservations.end_date",
                                new Document("$not",
                                        new Document("$gte",
                                                startDate))))),
                new Document("neighborhood", neighborhood),
                new Document("num_beds", numPeople)));

    }
    else{
        searchQuery = new Document("$and", Arrays.asList(new Document("$or", Arrays.asList(new Document("reservations.start_date",
                                new Document("$not",
                                        new Document("$lte",
                                                endDate))),
                        new Document("reservations.end_date",
                                new Document("$not",
                                        new Document("$gte",
                                                startDate))))),
                new Document("neighborhood", neighborhood),
                new Document("num_beds", numPeople),
                new Document("price",
                        new Document("$lt", price))));
    }
```

This function return a list of document there are converted into a PageDTO of AccommodationDTO by the business logic layer

```
public PageDTO<AccommodationDTO> showSearchAcc  (LocalDate startDate, LocalDate endDate, int numPeople, String neighborhood,double price,
                                                 int skip, int limit) throws BusinessException, RemoteException {
  try{
    documentAccDAO = new MongoAccommodationDAO();
    LinkedList<Document> results = (LinkedList<Document>) documentAccDAO.getSearchedAcc(startDate,endDate, numPeople,neighborhood, price, skip, limit);
    LinkedList<AccommodationDTO> accDTOList = new LinkedList<>();
    for (Document doc : results) {
        AccommodationDTO accDTO = new AccommodationDTO();
        //LinkedList<String> picsURL = (LinkedList<String>) doc.get("images_URL");
        accDTO.setId((int) doc.get("_id"));
        accDTO.setName((String) doc.get("name"));
        accDTO.setNeighborhood((String) doc.get("neighborhood"));
        accDTO.setRating((double) doc.get("rating"));
        accDTO.setPrice((double) doc.get("price"));
        accDTOList.add(accDTO);
    }
    PageDTO<AccommodationDTO> accommodations = new PageDTO<>();
    accommodations.setEntries(accDTOList);
    return accommodations;
  }catch(Exception e){
      throw new BusinessException(e);
  }finally{
      documentAccDAO.closeConnection();
  }
}
```

Write operations example:

As a write operations example we show the “insertReview” method of the CustomerService.class in the business logic layer. This method takes care of creating a new review in the collection of reviews in mongoBD and to maintain consistency it also adds this review in the graph, updates the rating in both DBs by recomputing it and increments the “number review” attribute. As we can see all the operations are present after the startTransaction() method. In this way all the operations that come after can be aborted. Once an exception is generated the transaction for MongoDB is aborted.

```java
public void insertReview(AccReviewDTO accReviewDTO, CustomerReviewDTO customerReviewDTO) throws BusinessException, RemoteException {
    try {
        documentRevDAO = new MongoReviewDAO();
        documentAccDAO = new MongoAccommodationDAO();
        graphRevDAO = new NeoReviewDAO();

        Review review = new Review();
        review.setId(documentRevDAO.getLastId(documentRevDAO.getCollection()));
        Accommodation acc = new Accommodation();
        acc.setId(customerReviewDTO.getAccommodationId());
        acc.setName(customerReviewDTO.getAccommodationName());
        review.setAccommodation(acc);

        Customer cus = new Customer();
        cus.setId(accReviewDTO.getCustomerId());
        cus.setFirstName(accReviewDTO.getCustomerFirstName());
        cus.setLastName(accReviewDTO.getCustomerLastName());
        cus.setCountry(accReviewDTO.getCustomerCountry());
        review.setCustomer(cus);
        review.setComment(accReviewDTO.getComment());
        review.setRate(accReviewDTO.getRate());
        review.setDate(LocalDate.now());
        documentRevDAO.startTransaction();

        documentRevDAO.createReview(review);
        documentAccDAO.incrementNumReview(review.getAccommodation());

        graphRevDAO.createReview(review);
        double rate=graphAccDAO.recomputeRate(review.getAccommodation());

        documentAccDAO.updateRating(review.getAccommodation(), rate);
        graphAccDAO.updateRating(review.getAccommodation(), rate);

        documentRevDAO.commitTransaction();  //IF commit will fail MongoDB driver will retry the commit operation one time
        documentAccDAO.commitTransaction();

    }catch(Exception e) {
        documentRevDAO.abortTransaction();
        throw new BusinessException(e);
    }finally{
        documentRevDAO.closeConnection();
        documentAccDAO.closeConnection();
    }
}
```

Neo4J suggested accommodation

The Node Similarity algorithm compares a set of nodes based on the nodes they are connected to. Two nodes are considered similar if they share many of the same neighbors. Node Similarity computes pair-wise similarities based on either the Jaccard metric, also known as the Jaccard Similarity Score, or the Overlap coefficient, also known as the Szymkiewicz–Simpson coefficient.
Given two sets A and B, the Jaccard Similarity is computed using the following formula:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

The Overlap coefficient is computed using the following formula:

$$O(A,B) = \frac{|A \cap B|}{min(|A|, |B|)}$$

The input of this algorithm is a bipartite, connected graph containing two disjoint node sets. Each relationship starts from a node in the first node set and ends at a node in the second node set.
The Node Similarity algorithm compares each node that has outgoing relationships with each other such node. For every node n, we collect the outgoing neighborhood N(n) of that node, that is, all nodes m such that there is a relationship from n to m. For each pair n, m, the algorithm computes a similarity for that pair that equals the outcome of the selected similarity metric for N(n) and N(m).
We implemented the suggesting accommodation mechanism in the method "showSuggestedAccommodation()" in NeoAccommodationDAO.java.
This method consist in two step:

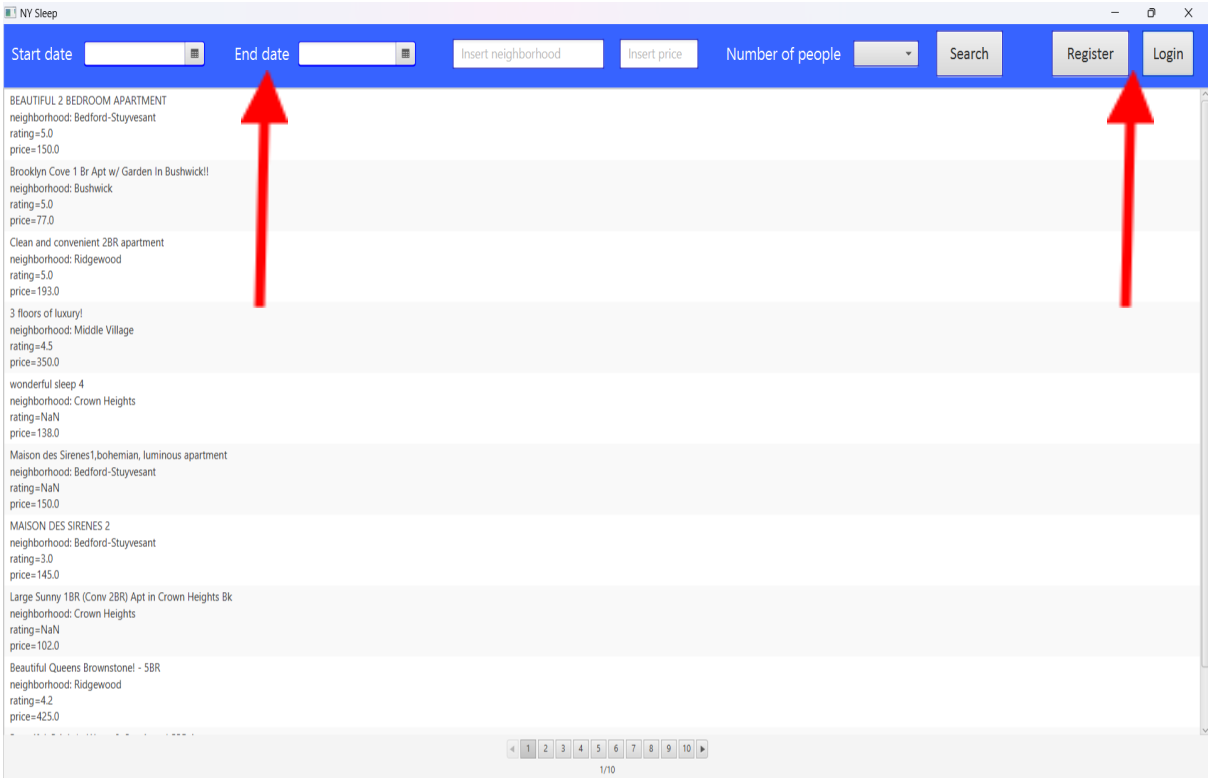
1. It calculates the most similar customer
2. It retrieves all the accommodations that the most similar customer has reviewed with a rate greater or equal than 4

```
public List<Record> showSuggestedAccommodation(Customer customer){
    driver = initDriver(driver);
    List<Record> recordList = new LinkedList<>();
    int id2=-1;
    try(Session session = driver.session())
    {
        //deletes the projection, if it already exists
        session.run("CALL gds.graph.drop( 'reviews', false);");
        //creates a new projection
        session.run("CALL gds.graph.project( 'reviews', ['customer','accommodation'], ['REVIEWS'])" +
                "YIELD graphName AS graph, relationshipProjection AS knowsProjection, nodeCount AS nodes, relationshipCount AS rels;");
        //calculates the most similar customer
        Result result1 = session.run("CALL gds.nodeSimilarity.stream('reviews') "+
                        "YIELD node1, node2, similarity " +
                        "WHERE gds.util.asNode(node1).id = $id " +
                        "RETURN gds.util.asNode(node1).id AS Customer1, gds.util.asNode(node2).id AS Customer2, similarity " +
                        "ORDER BY similarity DESCENDING, Customer1, Customer2 LIMIT 1"
                , parameters("id", customer.getId()));
        while(result1.hasNext()) {
            Record record= result1.next();
            id2=record.get("Customer2").asInt();
        }
        //retrieves the accommodation that the similar customer had rated well (4 or more)
        Result result2 = session.run("MATCH (cc:customer)-[r:REVIEWS]->(aa:accommodation) WHERE cc.id=$id2 AND r.rate>=4 "+
                        "RETURN aa.id AS id, aa.name AS name, aa.neighborhood as neighborhood, aa.rating as rating"
                , parameters("id2", id2));
        while(result2.hasNext()) {
            Record record= result2.next();
            recordList.add(record);
        }
        return recordList;
    }finally {
        close(driver);
    }
}
```

User Manual

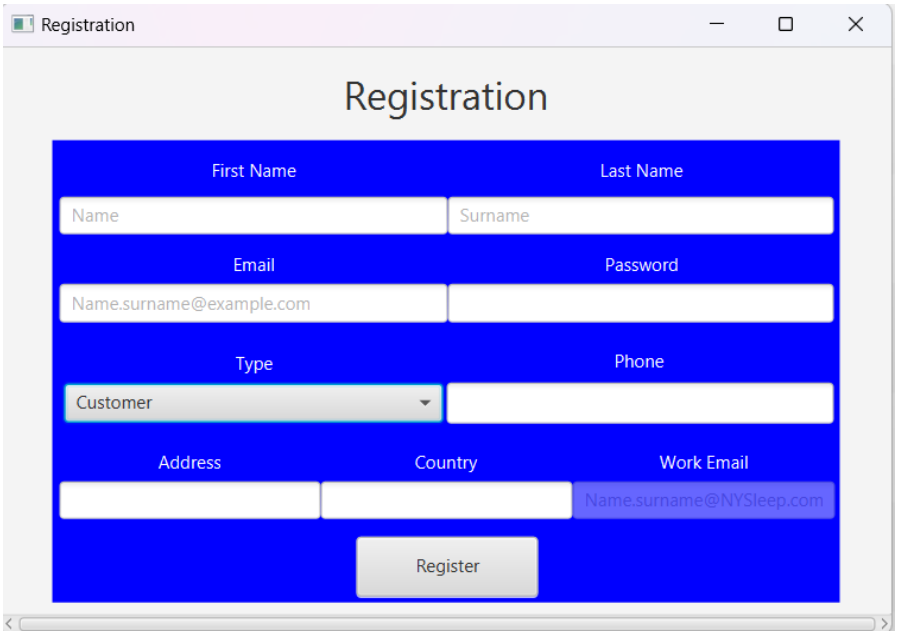
Homepage - Unregistered User

We have a view of some accommodations and the user can search for specific accommodation or decide to login or register



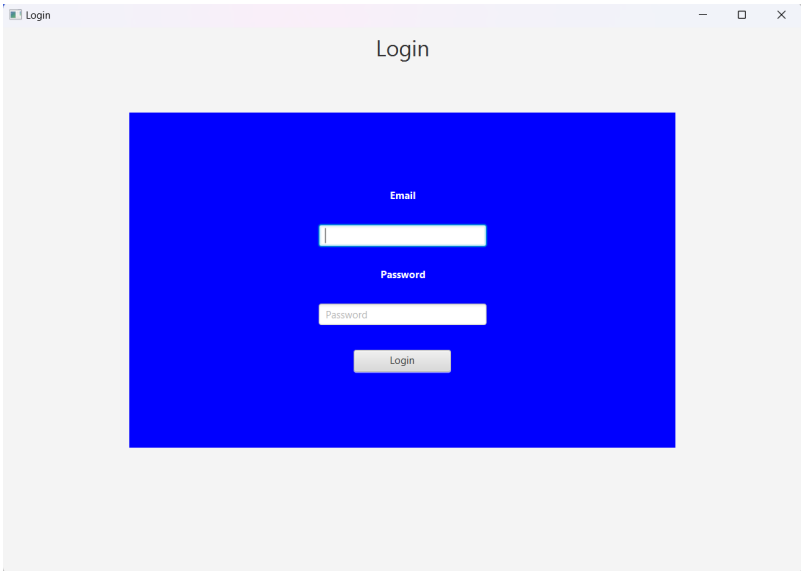
Register

Register page allow unregistered user to create an account



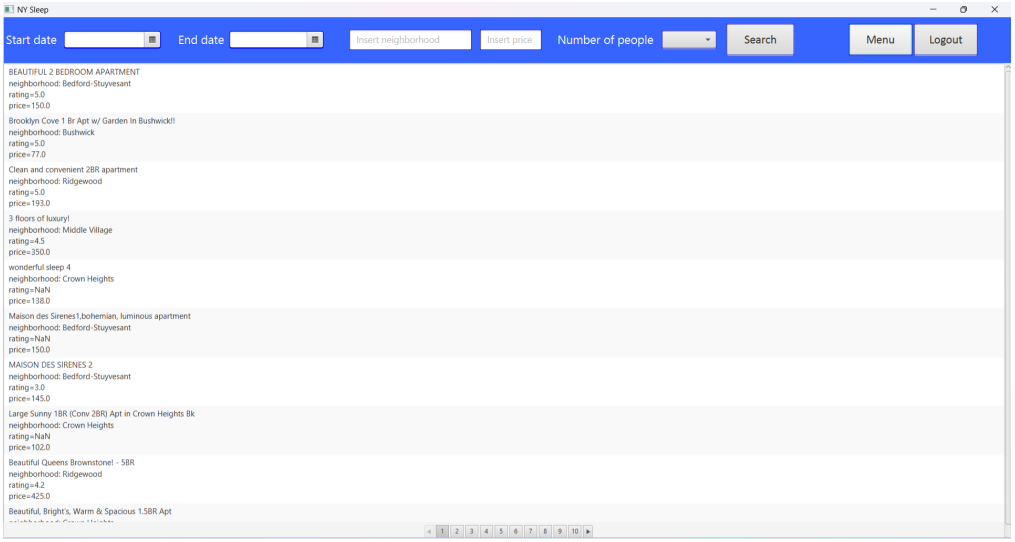
Login

The login page where the user can log as customer renter or admin by typing his credentials.

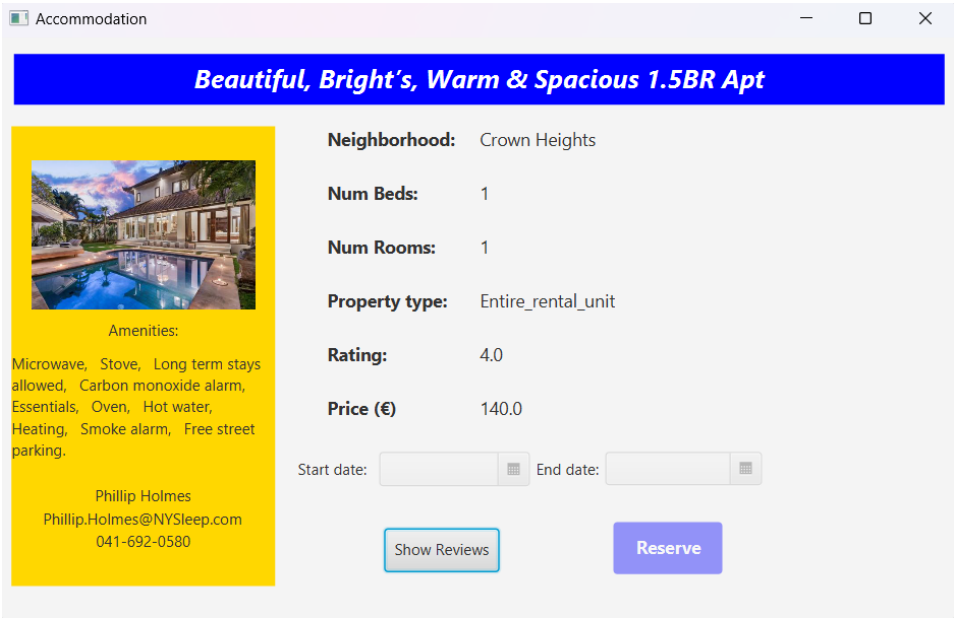


Home page - Registered User

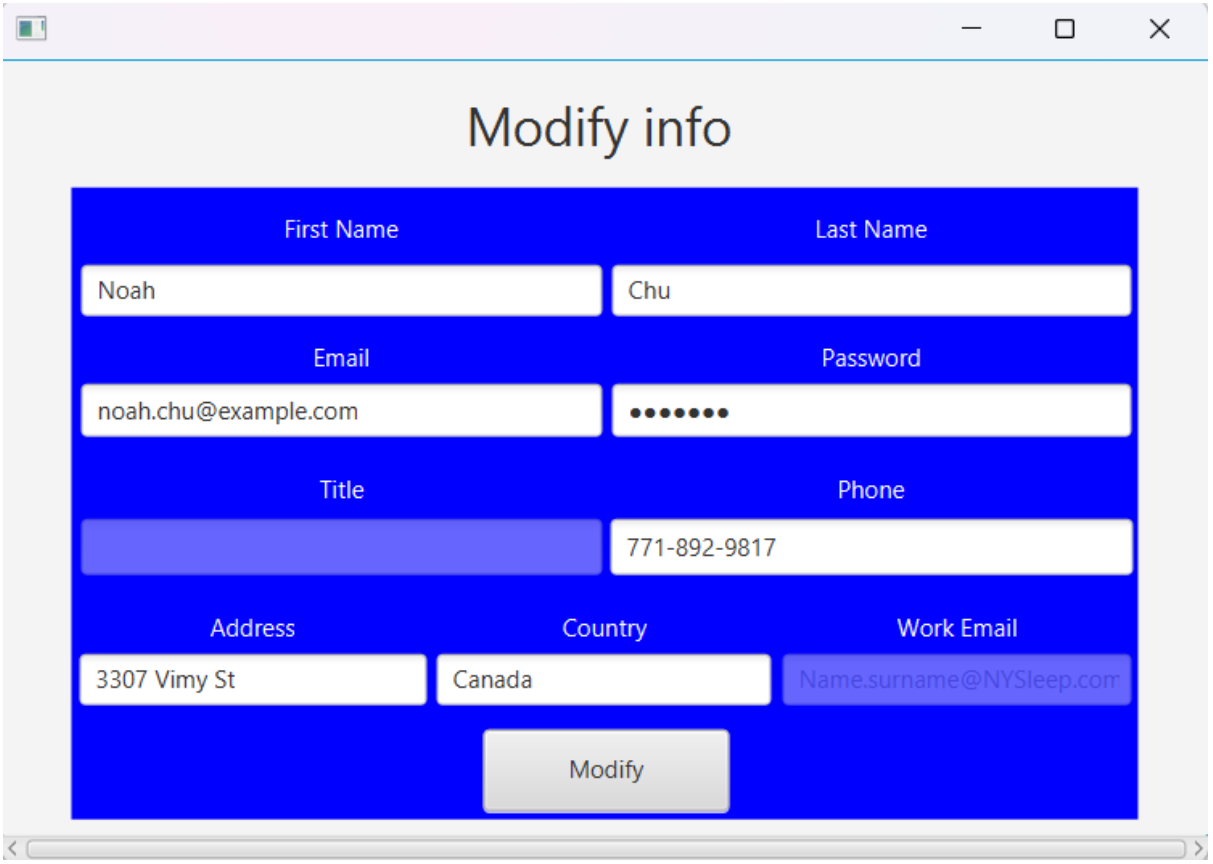
Once a user has logged in, the application shows the accommodation homepage for a registered user. In this section a registered user can search for accommodation and logout. The button menu allow the user to see his account information, his reservation details exc.,



Accommodation details

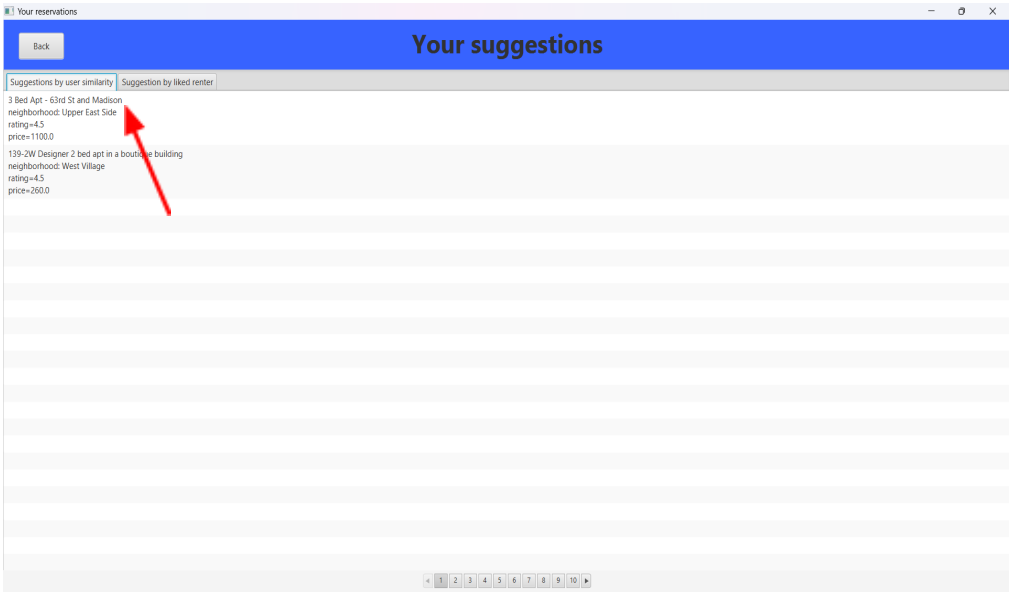


Modify account info



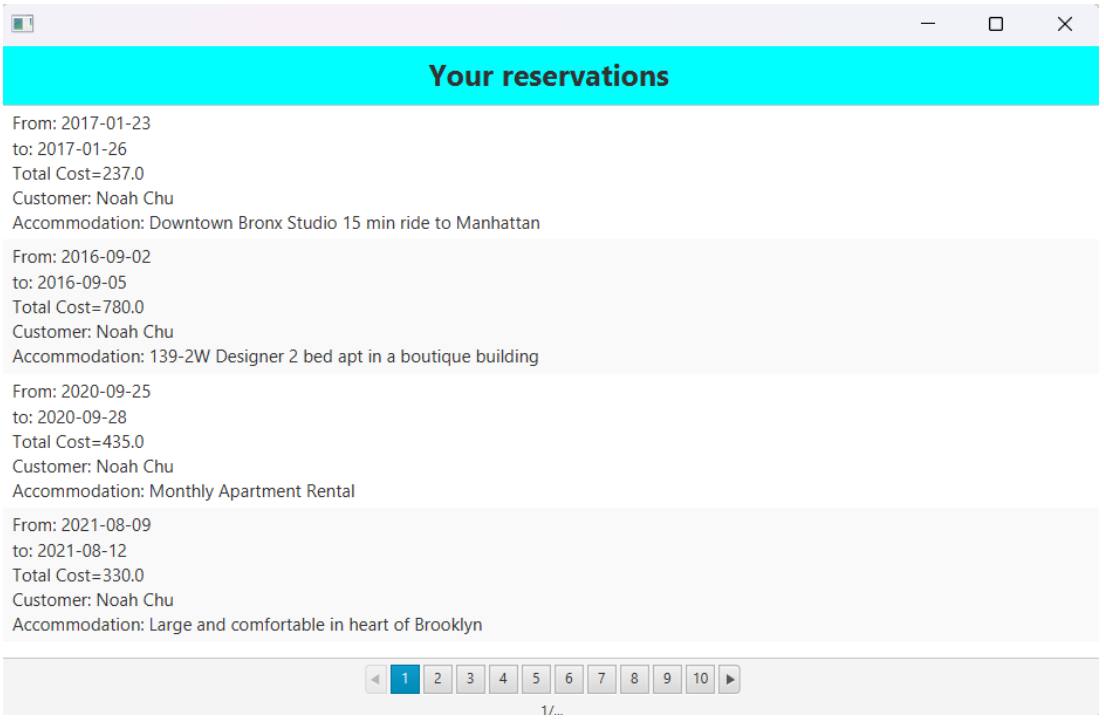
Suggested Accommodation

A customer can show suggested accommodation based on user similarity or by liked renter

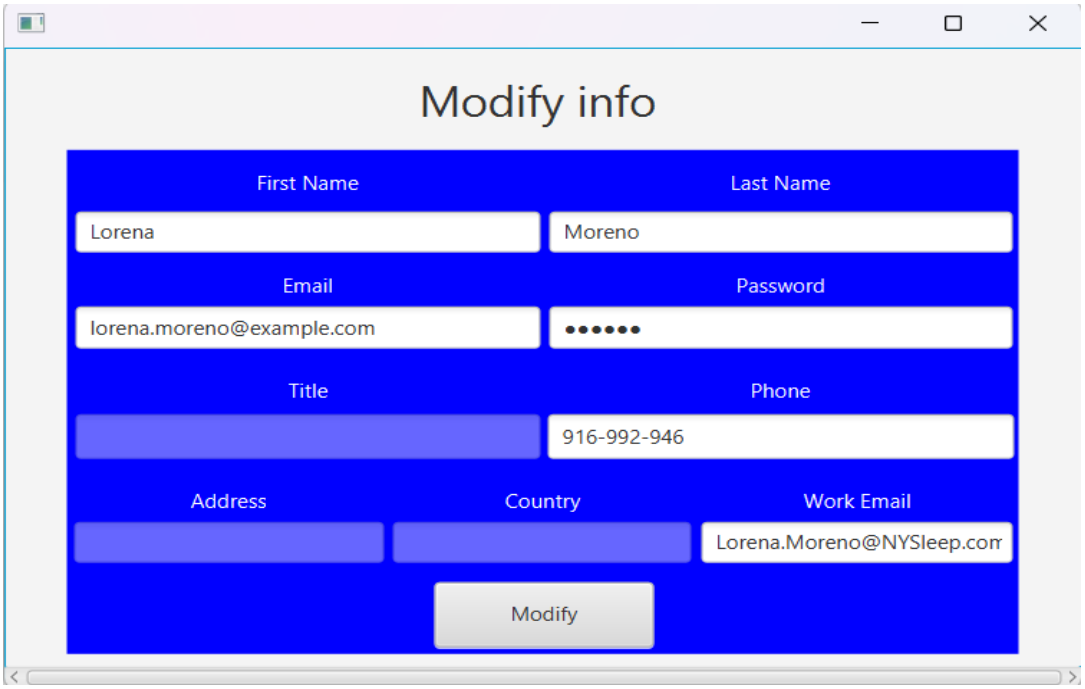


Customer's reservations

A customer can view his her own reservation listed in pages

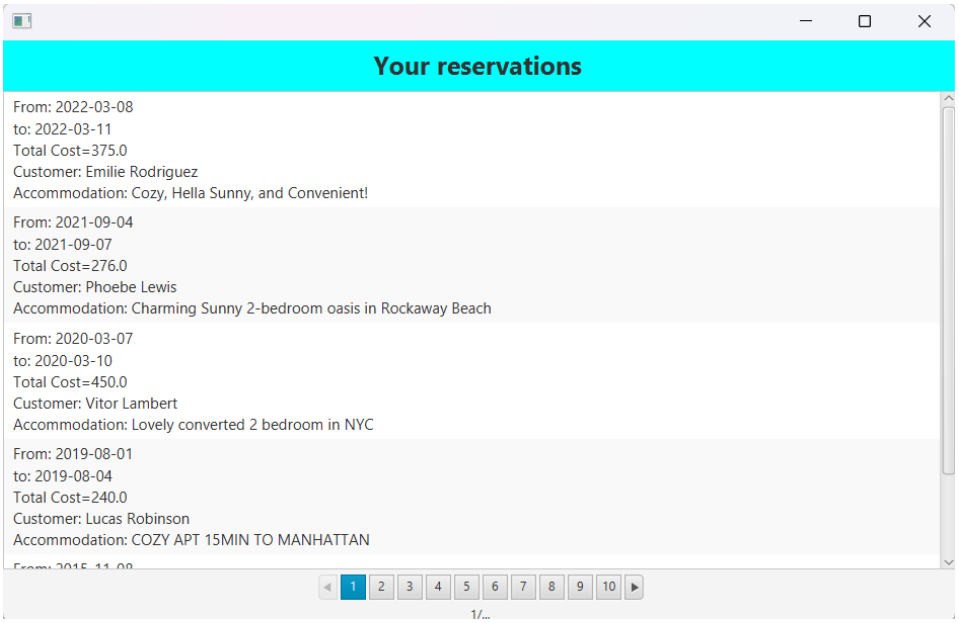


Modify account info - Renter



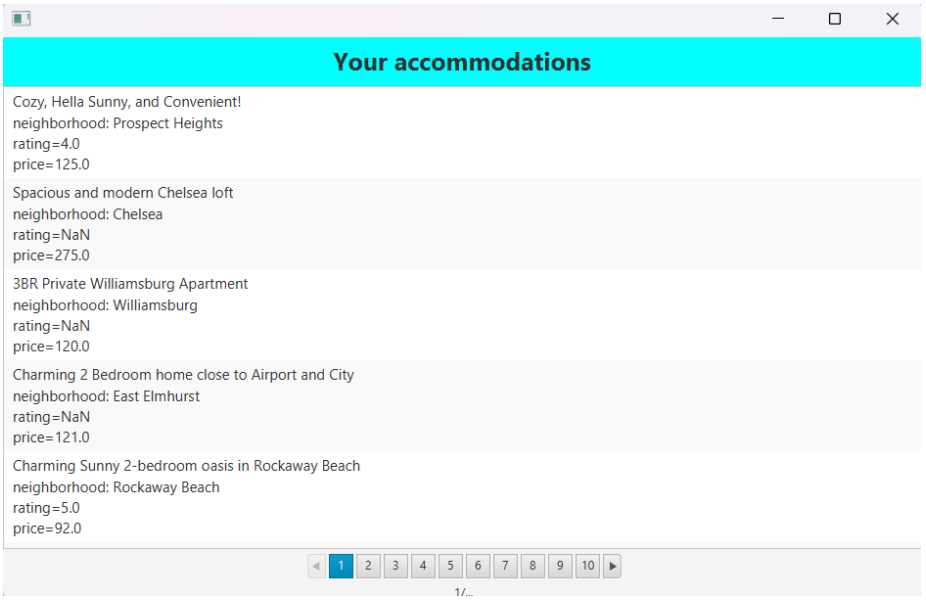
Renter's reservations

A page where a renter that logged in the application can see reservations about his/her own accommodations.

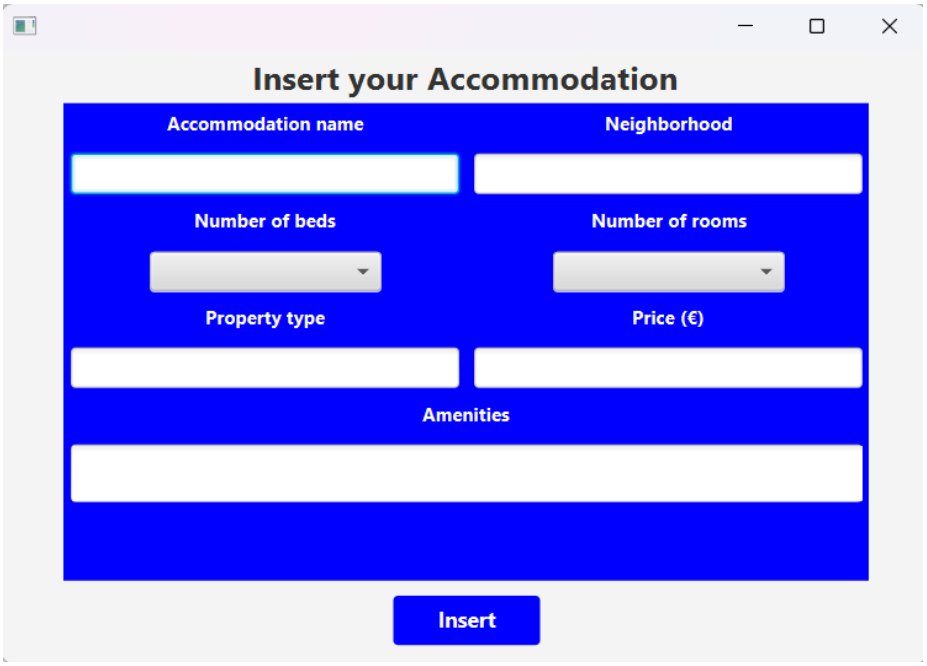


Renter’s accommodations

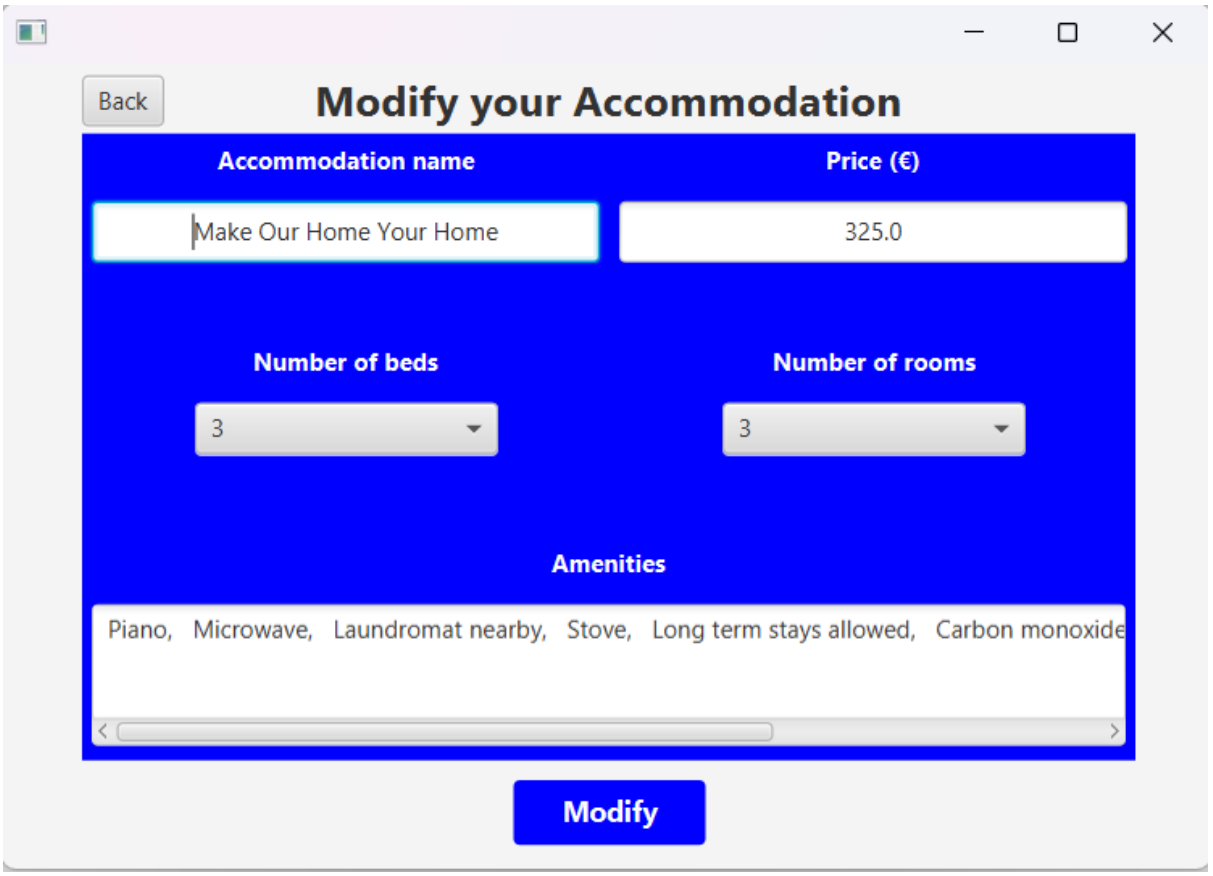
A page where a renter can see his/her own accommodations



Insert accommodations



Modify accommodations



Admin analytics