Hotel Booking Platform

Cloud Data Flexibility and Portability Across Multiple Providers

San Jose State University
CMPE 272: Enterprise Software Platforms
Noam Smilovich 016065699
Spring 2023, Prof. Bhaskar

Table of Contents

TABLE OF CONTENTS **2**

1. INTRODUCTION **3**

2. LITERATURE REVIEW **4**

CHOOSING THE RIGHT CLOUD DATABASE: CONSIDERATIONS FOR PERFORMANCE AND SCALABILITY 4

LEVERAGING MULTI-CLOUD ARCHITECTURES TO IMPROVE PERFORMANCE AND REDUCE VENDOR LOCK-IN 5

3. SYSTEM MODEL **6**

SYSTEM OVERVIEW 6

SYSTEM PROCESS FLOW DESCRIPTION 6

4. IMPLEMENTATION **8**

JAVA SERVER 8

Message Structure and Parsing *8*

Factory Method Design Pattern Implementation in the Server *8*

Processing of Client Requests in the Client Handler *9*

Migrate Feature *10*

THE DATABASES 11

MongoDB *11*

MySQL *12*

PYTHON CLIENT 14

The GUI *14*

Migrating from MySQL to MongoDB *15*

5. RESULTS AND CONCLUSIONS **16**

RESULTS 16

CONCLUSIONS 16

6. SETUP INFORMATION **17**

PRE-REQUISITES 17

SETUP 17

7. REFERENCES **18**

1. Introduction

The hotel booking platform project aims to provide a solution for companies who face challenges when using different cloud providers that have varying database architectures. In today's world, has become an essential part of the business world. However, many businesses face the issue of being locked into a specific cloud platform, which makes data portability between different cloud providers a significant challenge. This project focuses on addressing this problem by providing a solution that allows data portability between multiple cloud providers, reducing the risk of being locked into a specific cloud platform.

The project utilizes a Python GUI to provide an easy-to-use interface for hotel bookings. The Java server is designed to connect to multiple cloud databases, depending on the requests received from the Python client. This provides businesses with the flexibility to choose between different cloud providers and architectures based on their specific requirements. The server also includes a feature that allows for the migration of data from one database to another.

By using multiple cloud providers simultaneously, businesses can benefit from the unique strengths of each provider while avoiding vendor lock-in. This project's architecture allows for seamless switching between providers while maintaining data consistency and minimizing downtime.

The report will discuss the design and implementation of the project, including the challenges faced during development and the solutions that were implemented. It will also discuss the benefits of the project, including data portability, fault tolerance, and scalability, as well as the data migration feature. Finally, the report will provide recommendations for future improvements to the project, including additional features and optimizations.

2. Literature Review

The emergence of cloud computing has led to a vast array of cloud databases that offer unique features and benefits. However, each cloud database has its own advantages and disadvantages, making it challenging to choose the most appropriate database for a particular application. Moreover, being locked into a specific cloud platform can be a common problem when developing software for the cloud, making it difficult to switch providers or use multiple clouds simultaneously.

Choosing the Right Cloud Database: Considerations for Performance and Scalability

SQL databases have been around for several decades and are well known for their strong consistency and reliability. However, they may not be suitable for applications that require horizontal scaling due to their rigid schema design. On the other hand, NoSQL databases are designed for horizontal scaling and can handle large amounts of unstructured data. MongoDB is a popular NoSQL database that offers high performance for basic operations such as insertion and retrieval. However, it does not support aggregate functions natively, which can result in performance issues for applications that require such functions.

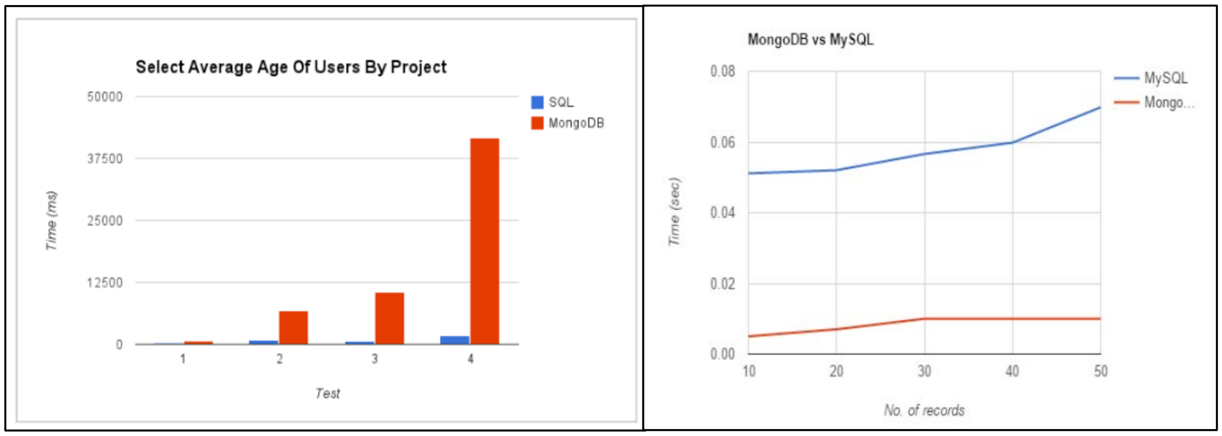


Figure 1: MySQL and MongoDB performance for two different cases [1][2]

The conflicting results from the plots in Figure 1, highlight the importance of considering the specific use case and workload when selecting a cloud database. As shown in the right plot, MongoDB outperforms MySQL. However, as shown in the left plot, MySQL shows better performance for the aggregate function. This is because each database has its own strengths and weaknesses and may be optimized for different types of data and workloads. Therefore, a thorough analysis of the specific requirements of the project and the available cloud database options is necessary to make an informed decision. Additionally, the use of multiple cloud databases can be a viable solution to leverage the strengths of each database and mitigate their weaknesses.

Leveraging Multi-Cloud Architectures to Improve Performance and Reduce Vendor Lock-In

To overcome the limitations of using a single cloud provider, many organizations are moving towards multi-cloud architectures. A multi-cloud architecture involves using multiple cloud providers to distribute the workload and minimize the risk of vendor lock-in. By leveraging the strengths of each cloud provider, a multi-cloud architecture can improve performance, scalability, and availability.

Data portability is a critical aspect of a multi-cloud architecture. Moving data between cloud providers can be challenging due to differences in data formats and storage architectures.

In conclusion, choosing the right cloud database is a critical decision that can have a significant impact on the performance and scalability of an application. By utilizing multiple cloud databases, organizations can leverage the strengths of each provider and improve overall performance. Furthermore, a multi-cloud architecture can provide additional benefits such as increased availability and reduced vendor lock-in. However, data portability remains a challenge, and organizations must carefully consider the trade-offs when designing a multi-cloud architecture.

3. System Model

System Overview

Figure 2 shows the proposed system overview. It comprises MongoDB and MySQL databases, a Java server and several Python clients. The Java server is responsible for connecting to both MongoDB and MySQL databases and receiving requests from the Python clients. The Python clients communicate with the Java server, which then communicates with the appropriate database to execute the requested operation.

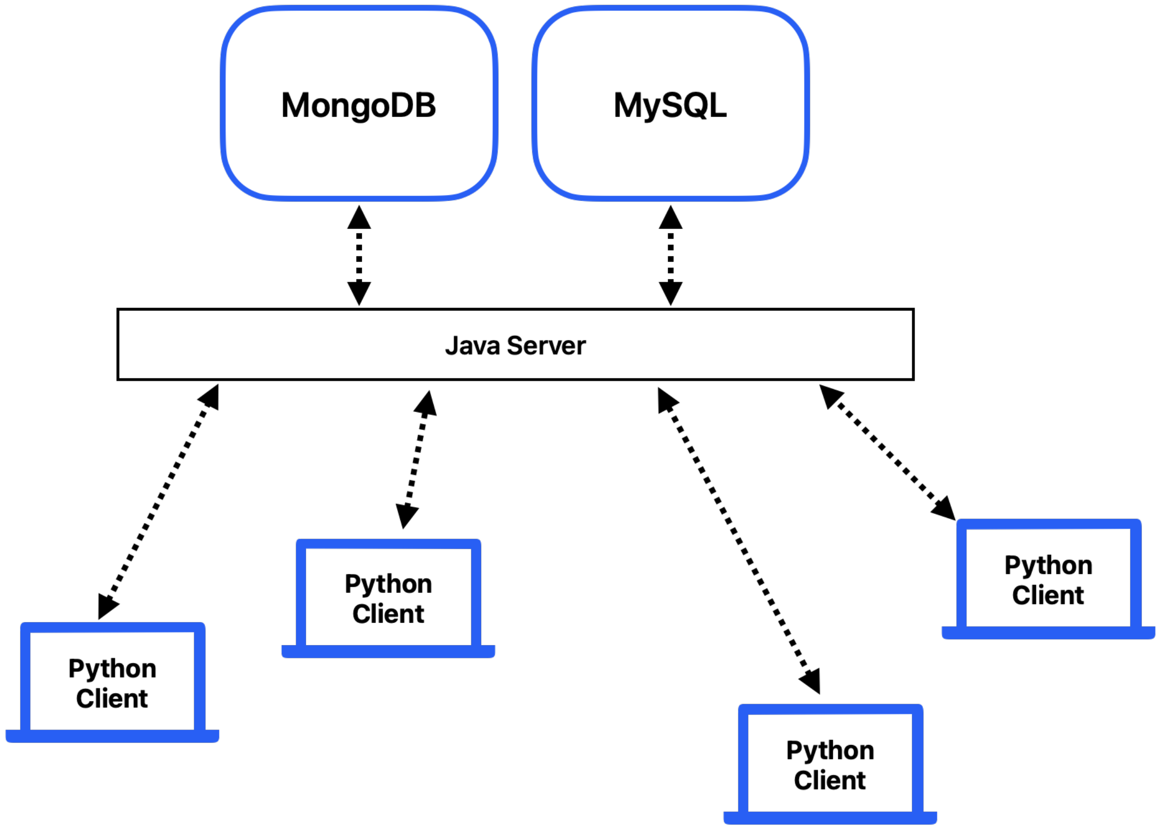


Figure 2: System Overview

The system architecture represented in the diagram allows for greater flexibility and ease of use, as the clients do not need to worry about the specifics of the database architecture, and can rely on the Java server to handle the communication and execution of operations on the appropriate database.

System Process Flow Description

Figure 3 shows a system flow diagram. The flow of the system is as follows:

1. Python client initiates a connection request to the Java server.
2. The "Listening Server", which is a connection requsts dedicated thread within the Java server, allocates a thread from a threadpool to a "Client Handler", then continues to listen to more connection requests.
3. The handoff to the client handler results in a socket connection formation between the Python client and the "Client Handler" portion within the Java server.

4. Messages are sent to the Java server from the Python client via the formed socket connection, they are handled by the client handler which parses them. Then queries are formed based on the received messages and submission to correct database is performed according to the aforementioned messages.
5. Queries sent to the correct databases, either MongoDB or MySQL, and results are returned to the java server.
6. Results from the databases are parsed and sent back to the Python client.

The message structure and parsing is explained in the implementation section.

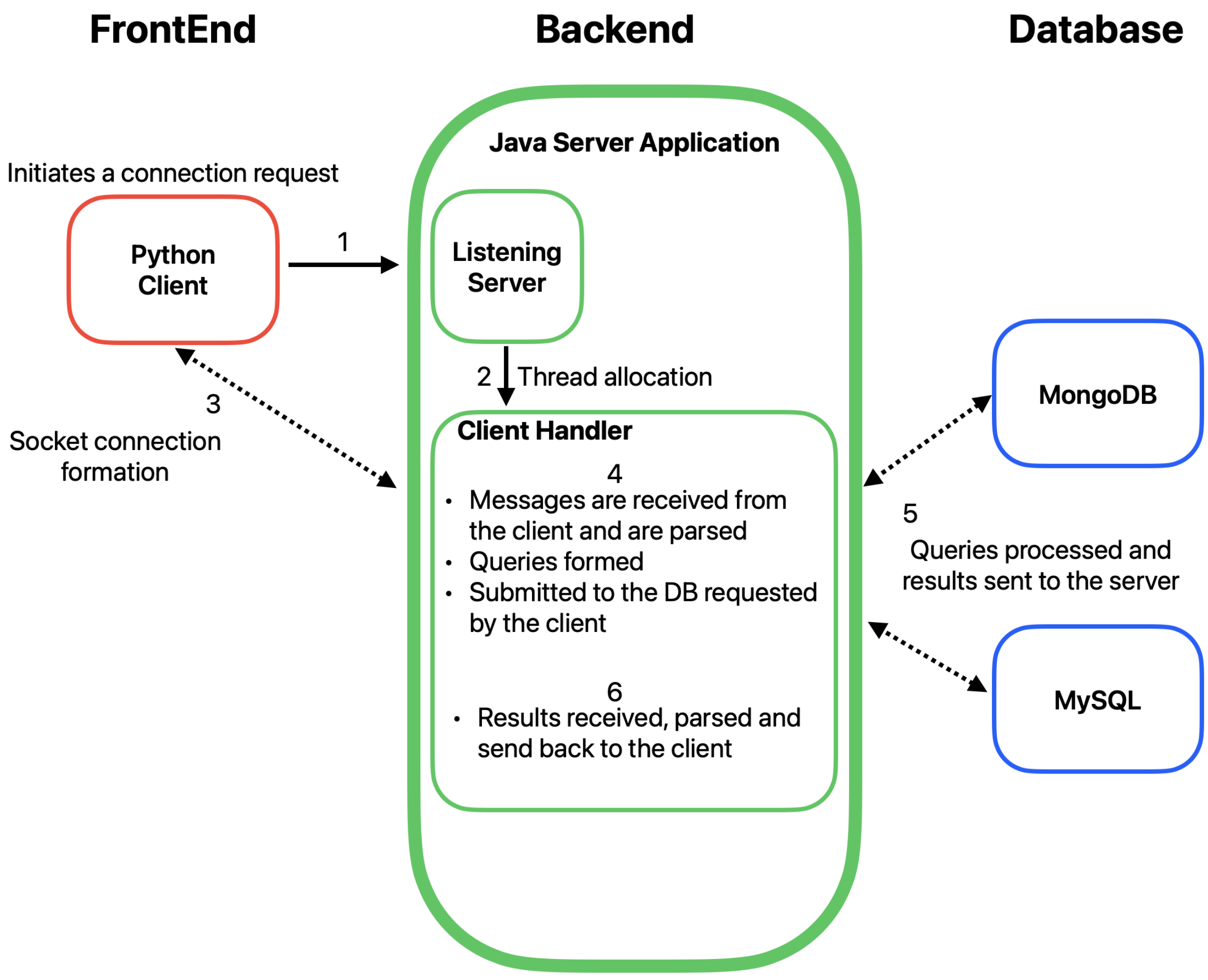

FrontEnd
Backend
Database
Java Server Application
Initiates a connection request
Python Client
1
Listening Server
2 Thread allocation
3
Socket connection formation
Client Handler
4
• Messages are received from the client and are parsed
• Queries formed
• Submitted to the DB requested by the client
6
• Results received, parsed and send back to the client
MongoDB
5
Queries processed and results sent to the server
MySQL

Figure 3: System Flow Diagram

4. Implementation

Java Server

Message Structure and Parsing

The communication between the Python client and the Java server is based on a simple message format that is easy to parse and can be extended to support additional functionality. The messages are sent and received in JSON format, which is a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate.

The messages sent from the client to the Java server have the following format:

{'client_type': client_type, 'database': database, 'collection': collection, 'operation': operation, 'filter': filter}

The message contains five fields:

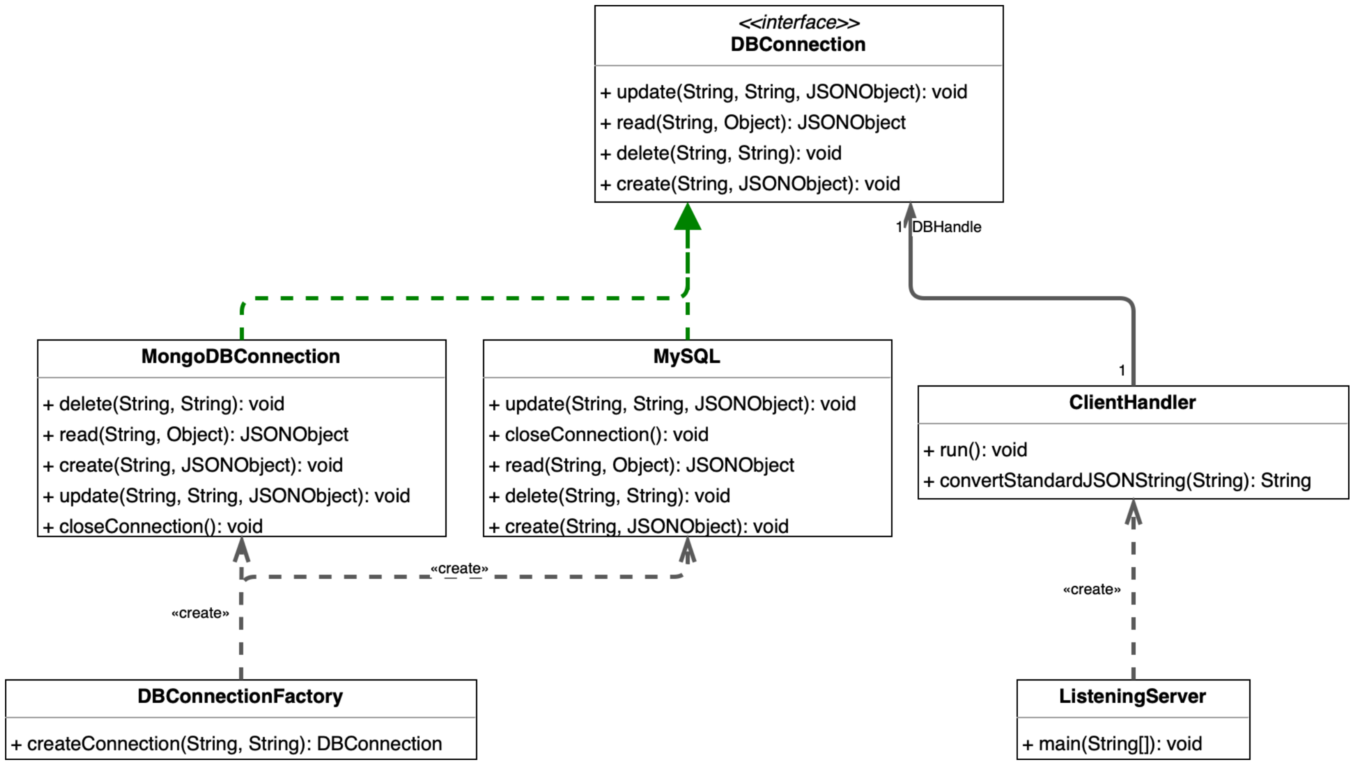
• client_type:	The type of database, which can be either "MySQL" or "MongoDB".
• database:	The name of the database within the cloud provider.
• collection:	The name of the collection within that database.
• operation:	One of the CRUD operations (Create, Read, Update, or Delete).
• filter:	An optional argument that is necessary for some CRUD operations, such as filtering data.

When the Java server receives the message from the client, it parses the JSON and extracts the fields to determine the type of operation to perform and the database and collection to perform it on. Once the server has processed the request and obtained the result from the database, it sends a response back to the client in a similar JSON format.

In order to ensure interoperability between the client and server, it is important to follow the message format as described above.

Factory Method Design Pattern Implementation in the Server

The Factory Method design pattern is a creational pattern that is used to provide an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. Figure 4 shows the UML class diagram for the Java server.



<<interface>>
DBConnection
+ update(String, String, JSONObject): void
+ read(String, Object): JSONObject
+ delete(String, String): void
+ create(String, JSONObject): void
1
DBHandle
1
MongoDBConnection
+ delete(String, String): void
+ read(String, Object): JSONObject
+ create(String, JSONObject): void
+ update(String, String, JSONObject): void
+ closeConnection(): void
MySQL
+ update(String, String, JSONObject): void
+ closeConnection(): void
+ read(String, Object): JSONObject
+ delete(String, String): void
+ create(String, JSONObject): void
ClientHandler
+ run(): void
+ convertStandardJSONString(String): String
«create»
«create»
«create»
DBConnectionFactory
+ createConnection(String, String): DBConnection
ListeningServer
+ main(String[]): void

Figure 4: Java Server UML Class Diagram

In the context of the Java server, the DBConnectionFactory class is a factory that creates DBConnection objects. The DBConnectionFactory class has a 'createConnection' method that receives a string as an argument and returns a DBConnection object according to that string.

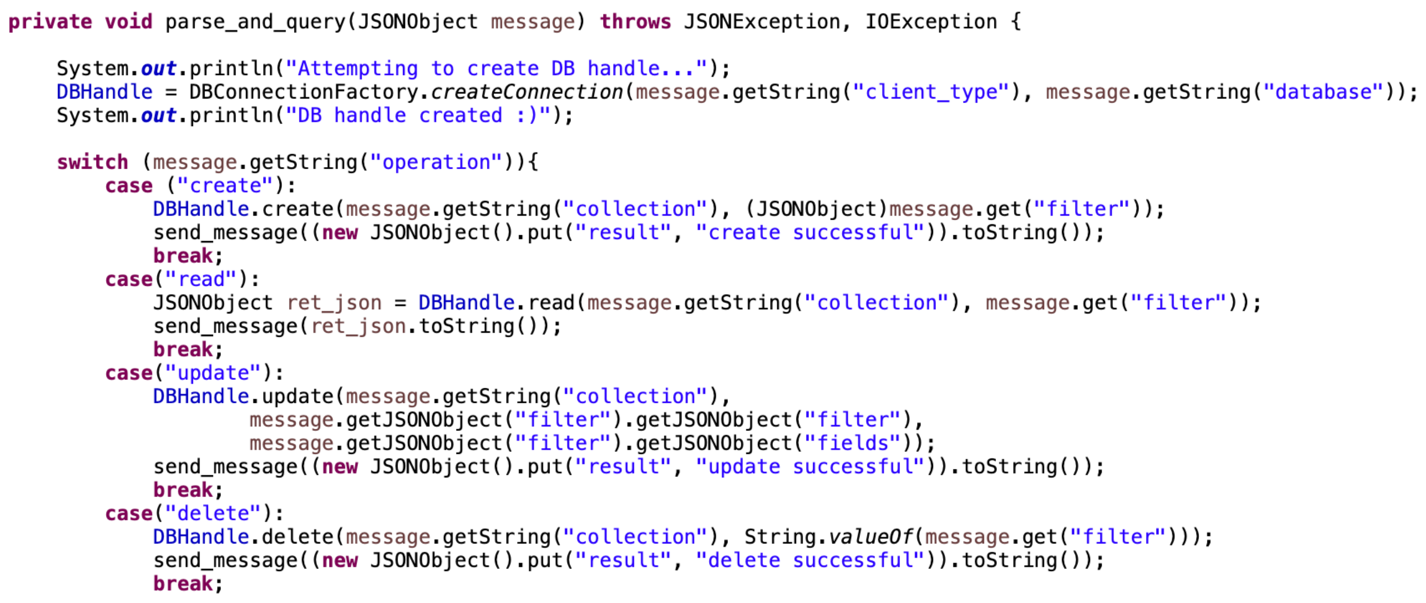
The DBConnection interface is implemented by two classes: MongoDBConnection and MySQLConnection. The DBConnection interface defines four methods that must be implemented by each subclass: create, read, update, and delete. This allows for flexibility in the implementation of the connection to different types of databases, as long as they conform to the interface.

The UML class diagram shows the relationship between the DBConnectionFactory class, the DBConnection interface, and the MongoDBConnection and MySQLConnection classes. The diagram also shows some of the methods defined in each class and interface.

By using the Factory Method design pattern, the Java server can create connections to different types of databases, namely MongoDB and MySQL, without having to know the details of how those connections are made. This allows for greater flexibility and modularity in the design of the Java server and makes it easier to add support for new types of databases in the future.

Processing of Client Requests in the Client Handler

Figure 5 shows the implementation of the ‘parse_and_query’ method. The method received the message received from the client in JSON format. Then according to the value in the JSONObject variable, which corresponds to the key ‘client_type’, it creates the proper DBConnection object used to interact with the appropriate database.



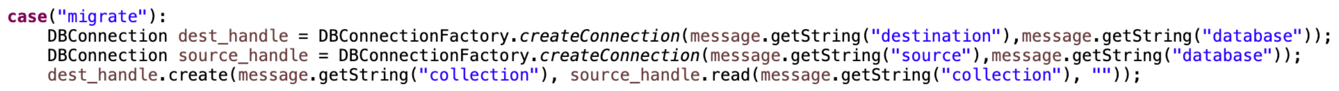
```java
private void parse_and_query(JSONObject message) throws JSONException, IOException {

    System.out.println("Attempting to create DB handle...");
    DBHandle = DBConnectionFactory.createConnection(message.getString("client_type"), message.getString("database"));
    System.out.println("DB handle created :)");

    switch (message.getString("operation")){
        case ("create"):
            DBHandle.create(message.getString("collection"), (JSONObject)message.get("filter"));
            send_message((new JSONObject().put("result", "create successful")).toString());
            break;
        case("read"):
            JSONObject ret_json = DBHandle.read(message.getString("collection"), message.get("filter"));
            send_message(ret_json.toString());
            break;
        case("update"):
            DBHandle.update(message.getString("collection"),
                    message.getJSONObject("filter").getJSONObject("filter"),
                    message.getJSONObject("filter").getJSONObject("fields"));
            send_message((new JSONObject().put("result", "update successful")).toString());
            break;
        case("delete"):
            DBHandle.delete(message.getString("collection"), String.valueOf(message.get("filter")));
            send_message((new JSONObject().put("result", "delete successful")).toString());
            break;
```

Figure 5: Client Handler 'parse_and_query' method

After creating the correct DBConnection object, one of its methods is invoked, in the switch statement, according to the value in the JSONObject variable which corresponds to the key 'operation'. A DBConnection object must implement the four CRUD operations methods.

Migrate Feature

The Java server also has a migrate feature. This feature involves reading data from the MySQL database using the 'read' operation, and then creating new data in the MongoDB database using the 'create' operation, with the MySQL data as the input. Figure 6 shows the implementation of the migrate feature.



```java
case("migrate"):
    DBConnection dest_handle = DBConnectionFactory.createConnection(message.getString("destination"),message.getString("database"));
    DBConnection source_handle = DBConnectionFactory.createConnection(message.getString("source"),message.getString("database"));
    dest_handle.create(message.getString("collection"), source_handle.read(message.getString("collection"), ""));
```

Figure 6: Migrate Feature Code

The code creates two database handles: one for the source database and one for the destination database. The source handle is used to read the information, while the destination handle is used to insert the data into the destination database. It is important to note that the data does not pass through the client during this process.

The Databases

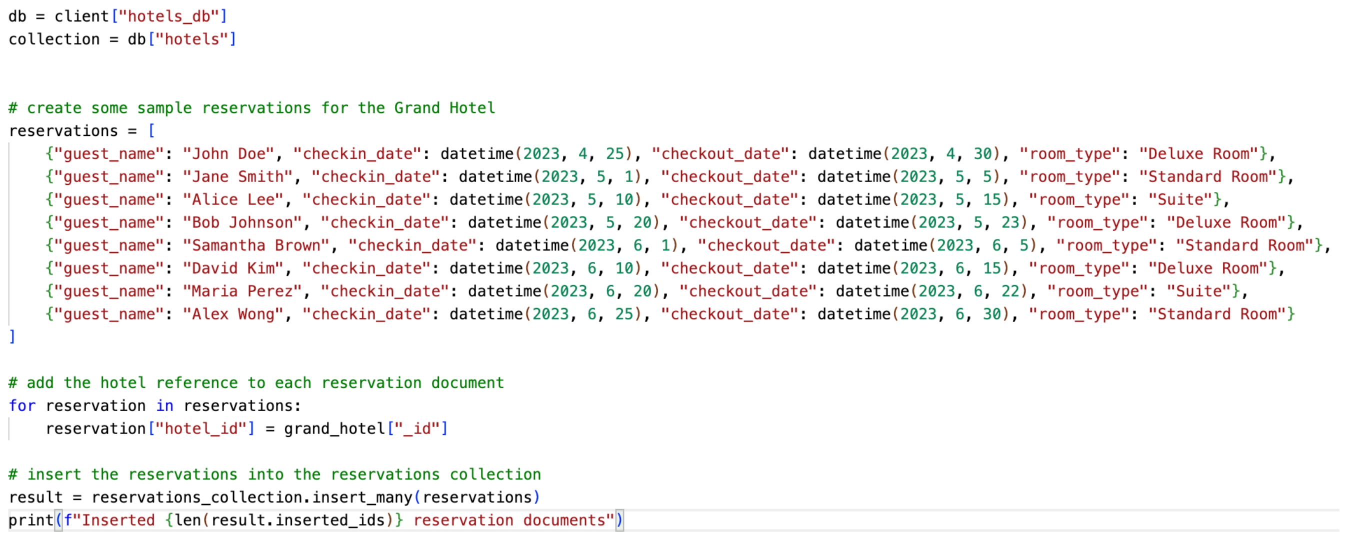
For this project, two databases were utilized: MongoDB and MySQL. The data in both databases was populated with mock information that includes five hotels with varying details about each hotel, such as room types, address, and amenities. In addition, reservation information was also included in the mock data.

MongoDB

MongoDB is a popular NoSQL database system that uses JSON-like documents. It is known for its flexibility, scalability, and performance when it comes to handling unstructured data. MongoDB supports various programming languages and platforms and provides powerful querying and indexing capabilities. It also has built-in sharding and replication features for high availability and horizontal scaling.

For MongoDB, I utilized their free trial available on their website to host the database in the cloud. This allowed for easy access and management of the data from anywhere with an internet connection.

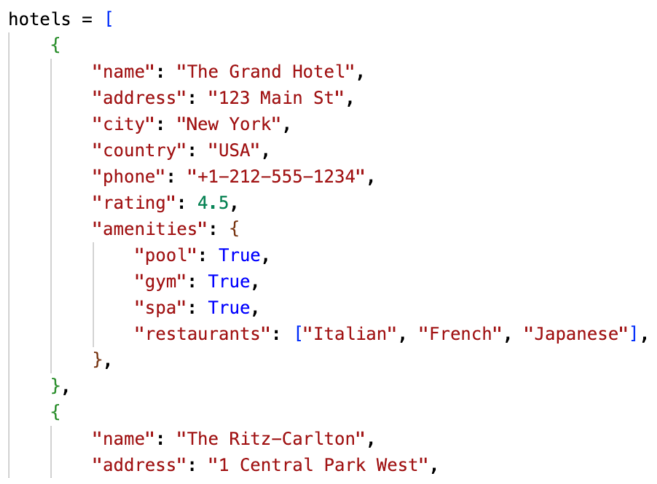
Figure 7 shows the Python script used to create and insert mock data into MongoDB.



```python
db = client["hotels_db"]
collection = db["hotels"]


# create some sample reservations for the Grand Hotel
reservations = [
    {"guest_name": "John Doe", "checkin_date": datetime(2023, 4, 25), "checkout_date": datetime(2023, 4, 30), "room_type": "Deluxe Room"},
    {"guest_name": "Jane Smith", "checkin_date": datetime(2023, 5, 1), "checkout_date": datetime(2023, 5, 5), "room_type": "Standard Room"},
    {"guest_name": "Alice Lee", "checkin_date": datetime(2023, 5, 10), "checkout_date": datetime(2023, 5, 15), "room_type": "Suite"},
    {"guest_name": "Bob Johnson", "checkin_date": datetime(2023, 5, 20), "checkout_date": datetime(2023, 5, 23), "room_type": "Deluxe Room"},
    {"guest_name": "Samantha Brown", "checkin_date": datetime(2023, 6, 1), "checkout_date": datetime(2023, 6, 5), "room_type": "Standard Room"},
    {"guest_name": "David Kim", "checkin_date": datetime(2023, 6, 10), "checkout_date": datetime(2023, 6, 15), "room_type": "Deluxe Room"},
    {"guest_name": "Maria Perez", "checkin_date": datetime(2023, 6, 20), "checkout_date": datetime(2023, 6, 22), "room_type": "Suite"},
    {"guest_name": "Alex Wong", "checkin_date": datetime(2023, 6, 25), "checkout_date": datetime(2023, 6, 30), "room_type": "Standard Room"}
]

# add the hotel reference to each reservation document
for reservation in reservations:
    reservation["hotel_id"] = grand_hotel["_id"]

# insert the reservations into the reservations collection
result = reservations_collection.insert_many(reservations)
print(f"Inserted {len(result.inserted_ids)} reservation documents")
```

Figure 7: MongoDB Reservations Mock Data Script

Similarly, mock data for the hotels themselves was also inserted in the same manner. Figure 8 shows a partial view of the hotels array which was inserted into MongoDB. The full script is on GitHub (Access in: Setup Information).



```
hotels = [
    {
        "name": "The Grand Hotel",
        "address": "123 Main St",
        "city": "New York",
        "country": "USA",
        "phone": "+1-212-555-1234",
        "rating": 4.5,
        "amenities": {
            "pool": True,
            "gym": True,
            "spa": True,
            "restaurants": ["Italian", "French", "Japanese"],
        },
    },
    {
        "name": "The Ritz-Carlton",
        "address": "1 Central Park West",
```

Figure 8: MongoDB Hotels Mock Data Partial Script

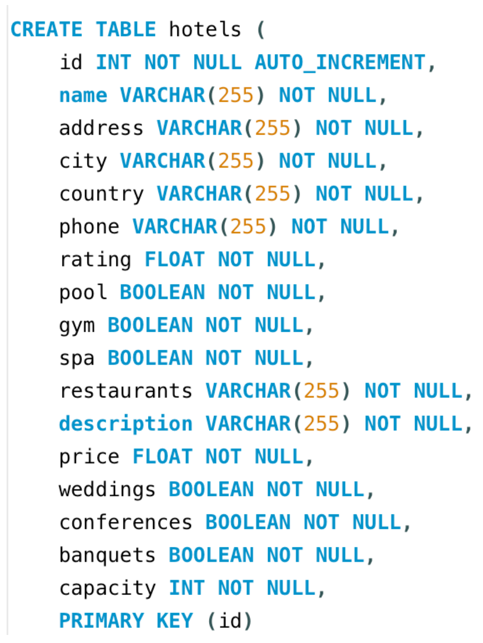
One of the advantages of using MongoDB with Python is the ease of inserting documents to the database. Both MongoDB and Python use JSON notation, which makes it very easy to create and insert documents as JSON objects. In this project though, this advantage was only used in the insertion of mock data.

MySQL

MySQL is a widely used open-source relational database management system. It is known for its robustness, reliability, and ease of use. MySQL supports SQL as its query language, and provides features such as transactions, views, and stored procedures. It is commonly used in web applications, e-commerce systems, and other data-driven applications.

For MySQL, I set up a local server using MySQL Workbench. This provided a reliable and secure way to manage and manipulate the data locally, without the need for external hosting.

Figure 9 shows the SQL 'hotels' table definition.



```sql
CREATE TABLE hotels (
    id INT NOT NULL AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    address VARCHAR(255) NOT NULL,
    city VARCHAR(255) NOT NULL,
    country VARCHAR(255) NOT NULL,
    phone VARCHAR(255) NOT NULL,
    rating FLOAT NOT NULL,
    pool BOOLEAN NOT NULL,
    gym BOOLEAN NOT NULL,
    spa BOOLEAN NOT NULL,
    restaurants VARCHAR(255) NOT NULL,
    description VARCHAR(255) NOT NULL,
    price FLOAT NOT NULL,
    weddings BOOLEAN NOT NULL,
    conferences BOOLEAN NOT NULL,
    banquets BOOLEAN NOT NULL,
    capacity INT NOT NULL,
    PRIMARY KEY (id)
```

Figure 9: MySQL ‘hotels’ Table Definition

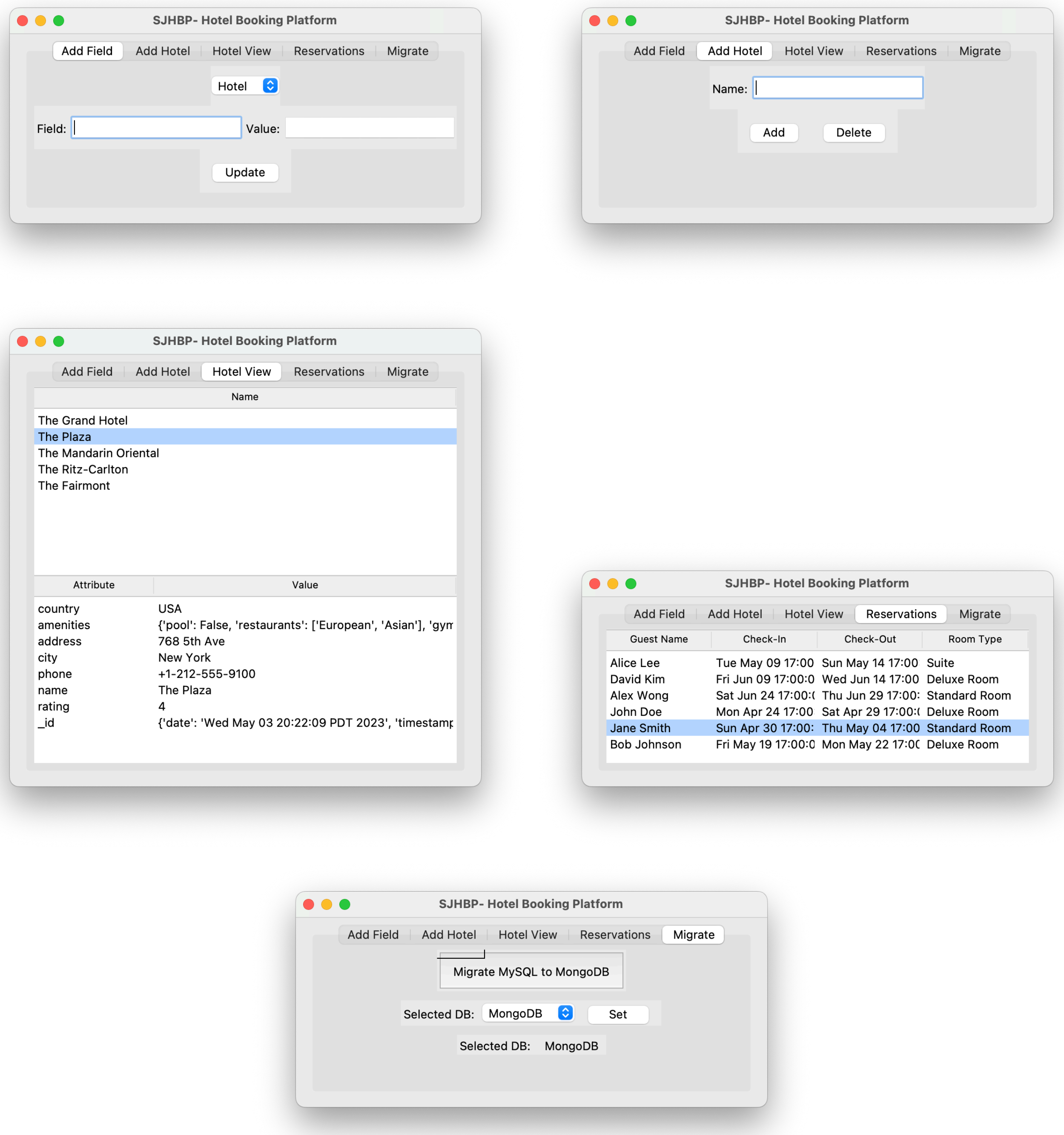
In contrast to MongoDB, the data in MySQL is "flattened" in a single table with columns for each field. While this can make queries more straightforward, it can also lead to data redundancy and make it more difficult to query complex data structures like nested objects and arrays. To represent the data in a similar way to MongoDB, one could create multiple tables and use relationships between them to represent the relationships between objects and arrays. However, this approach would require careful planning and implementation, and may not always be practical.

The full SQL script is available on GitHub (Access in: Setup Information).

Python Client

The GUI

Figure 10 shows the Python client GUI. The GUI allows easy user interaction. The GUI is a single window which consists of five tabs.



Name
The Grand Hotel
The Plaza
The Mandarin Oriental
The Ritz-Carlton
The Fairmont

Attribute	Value
country	USA
amenities	{'pool': False, 'restaurants': ['European', 'Asian'], 'gym
address	768 5th Ave
city	New York
phone	+1-212-555-9100
name	The Plaza
rating	4
_id	{'date': 'Wed May 03 20:22:09 PDT 2023', 'timestamp

Guest Name	Check-In	Check-Out	Room Type
Alice Lee	Tue May 09 17:00	Sun May 14 17:00	Suite
David Kim	Fri Jun 09 17:00:0	Wed Jun 14 17:00	Deluxe Room
Alex Wong	Sat Jun 24 17:00:(	Thu Jun 29 17:00:	Standard Room
John Doe	Mon Apr 24 17:00	Sat Apr 29 17:00:(	Deluxe Room
Jane Smith	Sun Apr 30 17:00:	Thu May 04 17:00	Standard Room
Bob Johnson	Fri May 19 17:00:0	Mon May 22 17:0(	Deluxe Room

Selected DB: MongoDB

Selected DB: MongoDB

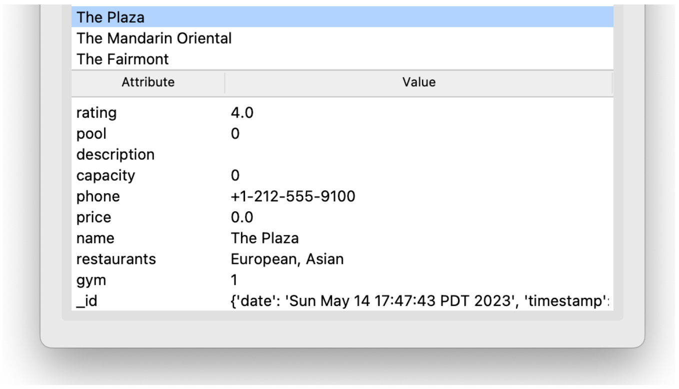
Figure 10: Python Client GUI

The GUI of the hotel management system utilizes the following CRUD operations to interact with the database:

- Add Field: allows the user to update an existing hotel field or create a new one if it does not exist. This operation is implemented using the **'update'** operation.
- Add Hotel: allows the user to add a new hotel or delete an existing one from the database. These operations are implemented using the **'create'** and **'delete'** operations respectively.
- Hotel View: displays a list of all the hotels in the database. This operation is implemented using the **'read'** operation.
- Reservations: displays a list of all the reservations in the database. This operation is also implemented using the **'read'** operation.
- Migrate: Allows to switch from MongoDB to MySQL. In addition, it includes the 'Migrate MySQL to MongoDB' feature, which transfers all the data in the MySQL database to MongoDB.

Migrating from MySQL to MongoDB

Figure 11 shows the outcome of migrating from MySQL to MongoDB.



The Plaza
The Mandarin Oriental
The Fairmont

Attribute	Value
rating	4.0
pool	0
description	
capacity	0
phone	+1-212-555-9100
price	0.0
name	The Plaza
restaurants	European, Asian
gym	1
_id	{'date': 'Sun May 14 17:47:43 PDT 2023', 'timestamp':

Figure 11: Outcome of Migrating from MySQL to MongoDB

Due to the way the 'hotels' table was defined in MySQL (Figure 9), the hotel data looks like it was flattened when comparing it to what it looked like in Figure 10.

5. Results and Conclusions

Results

Figure 12 shows the read performance of the Java server from MongoDB and MySQL. The read operation read the entire database of hotels, which consists of 5 hotels total.

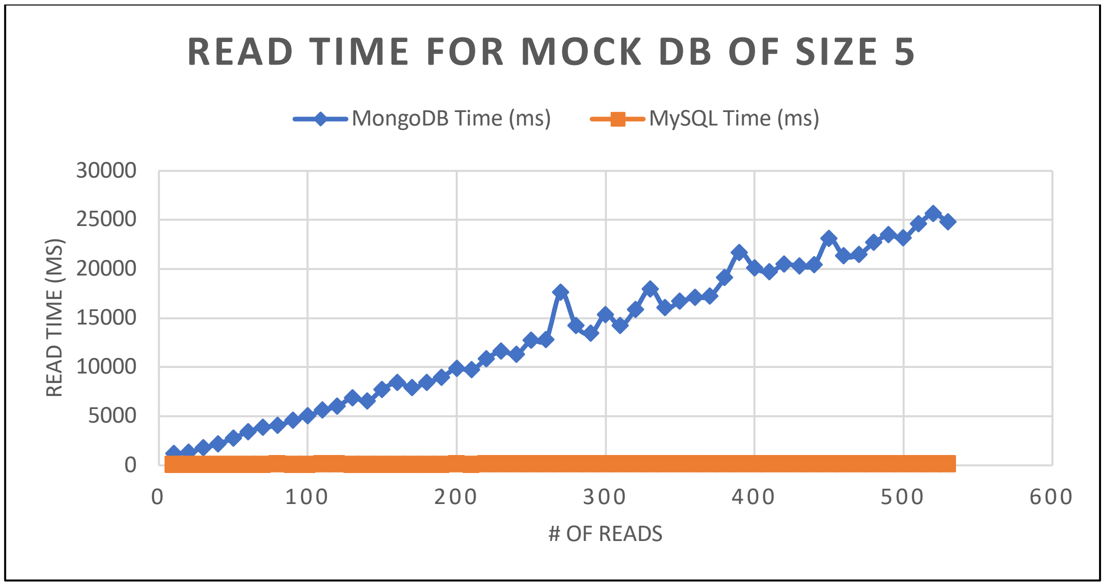


Figure 12: Java Server Performance for MongoDB and MySQL

Based on the results shown in the graph, it is evident that in the setup used for this project, MySQL outperformed MongoDB in terms of read time. However, it is important to note that this may not always be the case, as factors such as network latency and cloud provider performance could impact the results. The implementation of the actual read operation within the Java server also has an impact here.

Conclusions

The use of multiple database cloud providers can provide many benefits to applications but requires careful planning and development. While the migration process between providers can be facilitated by various tools and features offered by cloud providers, further research is necessary to identify the best practices for implementing such systems. Additionally, the development of a standard for cloud databases and migrations between them, similar to what 3GPP does for 5G, could potentially simplify the process and reduce the complexity of multi-provider setups.

In conclusion, this project has shown that using multiple databases can provide many benefits, such as improved performance and flexibility. However, it also requires careful planning and implementation to ensure that the different databases work together seamlessly. Additionally, it is important to consider the cost and complexity of using multiple databases, as well as the potential security risks. Overall, this project highlights the importance of understanding the strengths and weaknesses of different database technologies and choosing the appropriate ones for specific use cases. As technology continues to evolve, it is likely that new database technologies and migration tools will emerge, providing even more options and opportunities for developers and businesses.

6. Setup Information

The setup was tested on an M1 Mac running MacOS Ventura 13.2.1.

Pre-requisites

- Java JDK 17
- MySQL 8.0.30
- MySQL Workbench 8.0.29 or higher
- Python 3.9 or higher

It is crucial to have the exact specifications above. Java server testing on Java JDK 20 failed.

Setup

1. Clone the GitHub repository to the desired folder using the following command:

```
git clone https://github.com/NoamSmilovich/CMPE272-Hotel-Booking-Platform.git
```

2. In order to set a virtual environment, installation of virtualenv platform is required. This is not mandatory, but it is recommended.
   Use the following commands to create a new working virtual environment with all the required dependencies.

```
cd CMPE272-Hotel-Booking-Platform/PythonClient
python -m virtualenv .
.\Scripts\activate
pip install -r requirements.txt
```

3. Navigate to the JavaServer folder and run the following commands to start the server:

```
java -cp .:"Dependencies/*.jar" -jar javaserver.jar
```

4. Navigate to the PythonClient folder and run the following commands to start the client:

```
python GUI.py
```

Now the GUI should start, and you can explore the functionality of the application. Note that the GUI will not start if the server is not running first.

Important: you will not be able to explore the functionality of the MySQL portion unless you start a local MySQL server and run the 'hotels_mock_data.sql' script in the main folder first.

7. References

[1] Parker, Zachary, et al. “Comparing NoSQL Mongodb to an SQL DB.” Proceedings of the 51st ACM Southeast Conference, 2013, https://doi.org/10.1145/2498328.2500047.

[2] M. M. Patil, A. Hanni, C. H. Tejeshwar, and P. Patil, “A qualitative analysis of the performance of MongoDB vs MySQL database based on insertion and retriewal operations using a web/android application to explore load balancing — Sharding in MongoDB and its advantages,” 2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), Feb. 2017, doi: https://doi.org/10.1109/ismac.2017.8058365.

[3] Mahdi Negahi Shirazi, Ho Chin Kuan, and Hossein Rezaei Dolatabadi, “Design Patterns to Enable Data Portability between Clouds’ Databases,” International Conference on Computational Science and Its Applications, Jun. 2012, doi: https://doi.org/10.1109/iccsa.2012.29.

[4] F. Ravat, J. Song, O. Teste, and C. Trojahn, “Efficient querying of multidimensional RDF data with aggregates: Comparing NoSQL, RDF and relational data stores,” International Journal of Information Management, p. 102089, Jun. 2020, doi: https://doi.org/10.1016/j.ijinfomgt.2020.102089.

[5] B. Jose and S. Abraham, “Exploring the merits of nosql: A study based on mongodb,” IEEE Xplore, Jul. 01, 2017. https://ieeexplore.ieee.org/document/8076778.

[6] M. Diogo, B. Cabral, and J. Bernardino, “Consistency Models of NoSQL Databases,” Future Internet, vol. 11, no. 2, p. 43, Feb. 2019, doi: https://doi.org/10.3390/fi11020043.

[7] R. Gunawan, A. Rahmatulloh, and I. Darmawan, “Performance Evaluation of Query Response Time in The Document Stored NoSQL Database,” IEEE Xplore, Jul. 01, 2019. https://ieeexplore.ieee.org/document/8898035.

[8] S. Bradshaw, K. Chodorow, and Eoin Brazil, MongoDB : the definitive guide : powerful and scalable data storage. Sebastopol, Ca: O’reilly, 2019.

[9] A. Davoudian, L. Chen, and M. Liu, “A Survey on NoSQL Stores,” ACM Computing Surveys, vol. 51, no. 2, pp. 1–43, Apr. 2018, doi: https://doi.org/10.1145/3158661.

[10] M. Ha and Y. Shichkina, "The Query Translation from MySQL to MongoDB Taking into Account the Structure of the Database," 2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus), St. Petersburg, Moscow, Russia, 2021, pp. 383-386, doi: 10.1109/ElConRus51938.2021.9396591.

[11] I. Mearaj, P. Maheshwari and M. J. Kaur, "Data Conversion from Traditional Relational Database to MongoDB using XAMPP and NoSQL," 2018 Fifth HCT Information Technology Trends (ITT), Dubai, United Arab Emirates, 2018, pp. 94-98, doi: 10.1109/CTIT.2018.8649513.

[12] A. Kanade, A. Gopal and S. Kanade, "A study of normalization and embedding in MongoDB," 2014 IEEE International Advance Computing Conference (IACC), Gurgaon, India, 2014, pp. 416-421, doi: 10.1109/IAdCC.2014.6779360.