Database Systems 236363, Winter 2023/24

Homework 2

- TA in charge: Erez Koifman
- Due date: 13/03/2024
- Questions should be asked in the dedicated forum in the course's Piazza: https://piazza.com/technion.ac.il/winter2024/236363
- Administrative questions should be directed to Shir Rotman

1. Introduction

In this assignment you will develop part of an innovative app called “airdbnb” that allows apartment owners to rent their apartments to customers.
In this program, users with admin privileges (you), can add apartment owners, customers, apartments, reservations, and reviews that were received from customers.
Your mission is to design the database and implement the data access layer of the system. Typically, the data access layer facilitates the interaction of other components of the system with the database by providing a simplified API that carries out a predefined desired set of operations. A function in the API may receive business objects as Input arguments. These are regular Python classes that hold special semantic meaning in the context of the application (typically, all other system components are familiar with them). The ZIP file that accompanies this document contains the set of business objects to be considered in the assignment, as well as the full (unimplemented) API. Your job is to implement these functions so that they fulfill their purpose as described below.

Please note:
1. The database design is your responsibility. You may create and modify it as you see fit. You will be graded for your database design, so bad and inefficient designs will suffer from points reduction.
2. Every calculation involving the data, like filtering and sorting, must be done by querying the database. You are prohibited from performing any calculations on the data using Python. Furthermore, you cannot define your own classes, and your code must be contained in the functions given, except for the case of defining basic functions to avoid code duplication. Additionally, you may only use the material learned in class when writing your queries.
3. It is recommended to go over the relevant Python files and understand their usage.
4. All provided business classes are implemented with a default constructor and getter\setter to each field.
5. You can use only **one** SQL query in each function implementation, not including views. Create/Drop/Clear functions are not included!

2. Business Objects

In this section we describe the business objects to be considered in the assignment.
Unless specified otherwise, all attributes cannot be NULL.
Unless specified otherwise, all IDs are positive (>0).

Owner
Attributes:

Description	Type	Comments
Owner ID	Int	The owner's ID
Name	String	The owner's name

Constraints:

1. Owner IDs are unique

Apartment
Attributes:

Description	Type	Comments
ID	Int	The apartment's ID
Address	String	The apartment's address
City	String	The city the apartment is in
Country	String	The country the apartment is in
Size	Int	Size in square meters

Constraints:

1. There can't be two apartments in the same city at the same address

Customer

Attributes:

Description	Type	Comments
Customer ID	Int	The customer's ID
Customer name	String	The name of the customers

Constraints:

1. Customer IDs are unique

3. Dates in SQL and Python

3.1 Dates in SQL

Postgres has a type called DATE for storing dates.

Creating a DATE that represents the 6th of may 2023:

This line returns a table with a single row and a single column that has type date and a value of the date 2023-5-6.

```
SELECT DATE('2023-5-6');
```

The returned table:

	date date
1	2023-05-06

We can extract fields from a date object:

This line returns a table with a single row and a single column that has type numeric and a value of the day from the given date (6)

```
SELECT EXTRACT(DAY FROM DATE('2023-5-6'));
```

The returned table:

	extract numeric
1	6

The next two lines will return similar tables with values 5 and 2023.

```
SELECT EXTRACT(MONTH FROM DATE('2023-5-6'));
SELECT EXTRACT(YEAR FROM DATE('2023-5-6'));
```

3.2 Dates in Python

This assignment will be done in python. Because you will be working with dates, you will need to use the date object from the datetime module in python.

How to use date:

Import:

```
from datetime import date, datetime
```

Basic constructor:

```
# format: date(year: int, month: int, day: int)
# example: 24/12/2023
d = date(2023, 12, 24)
```

Basic getters:

```
print(d.year) # prints “2023”
print(d.month) # prints “12”
print(d.day) # prints “24”
```

date.strftime:

A function for getting a string of the date object in a specified format. The function uses codes for different representations of year, month and day.

A link to a list of all codes: [Python strftime reference cheatsheet]

Usage example:

```
print(d.strftime('%Y-%m-%d')) # prints “2023-12-24”
print(d.strftime('%A %B %d')) # prints “Sunday December 24”
```

datetime.strptime:

This function is the inverse of strftime. It takes a format and a string and converts it into a datetime object. It uses the same codes as strftime.

Usage example:

```
t = datetime.strptime('2023-12-24', '%Y-%m-%d').date()
print(t) # prints “2023-12-24”
print(type(t)) # prints “<class 'datetime.date'>”
```

Further reading: [datetime — Basic date and time types — Python 3.12.1 documentation]

4. API

In this part you will implement the API for the system. You should use views whenever possible to avoid code duplication. We will deduce points for code duplication where views could have been used.

Note: for get_apartment_rating and get_owner_rating you **must** use a view (one view that will be used for both).

3.1 Return Type

For the return value of the API functions, we have defined the following enum type:

ReturnValue (enum):

- OK
- NOT_EXISTS
- ALREADY_EXISTS
- ERROR
- BAD_PARAMS

In case of conflicting return values, return the one that appears first on each section.

3.2 CRUD API

This part handles the CRUD - Create, Read, Update and Delete operations of the business objects in the database. Implementing this part correctly will lead to easier implementations of the more advanced APIs.

Python's equivalent to NULL is None.

You can assume the arguments to the function will not be None, the inner attributes of the argument might consist of None.

ReturnValue add_owner(owner: Owner)

Add an owner to the database.

Input: An Owner object

Output: ReturnValue with the following conditions:

- OK in case of success
- BAD_PARAMS if any of the params are illegal
- ALREADY_EXISTS if a owner with the same ID already exists
- ERROR in case of database error

Hotel get_owner(owner_id: int)

Get an owner from the database.

Input: The id of the requested owner

Output: The object of the requested owner if the owner exists, bad_owner() otherwise

ReturnValue delete_owner(owner_id: int)

Delete an owner from the database.

Input: The id of the owner to delete

Output: ReturnValue with the following conditions:

- OK in case of success
- BAD_PARAMS if any of the params are illegal
- NOT_EXISTS if the owner does not exist
- ERROR in case of database error

ReturnValue add_apartment(apartment: Apartment)

Add an apartment to the database.

Input: An Apartment object

Output:

ReturnValue with the following conditions:

- OK in case of success
- BAD_PARAMS if any of the params are illegal
- ALREADY_EXISTS if an apartment in the same location (address and city) already exists
- ERROR in case of database error

Room get_apartment(apartment_id: int)

Get an apartment from the database.

Input: The address, city and country of the requested apartment

Output: The object of the requested apartment if the apartment exists, bad_apartment() otherwise

ReturnValue delete_apartment(apartment_id: int)

Delete an apartment from the database.

Input: The id of the apartment to delete

Output: ReturnValue with the following conditions:

- OK in case of success
- BAD_PARAMS if any of the params are illegal
- NOT_EXISTS if the apartment does not exist
- ERROR in case of database error

ReturnValue add_customer(customer: Customer)

Add a customer to the database.

Input: A Customer object

Output: ReturnValue with the following conditions:

- OK in case of success
- BAD_PARAMS if any of the params are illegal
- ALREADY_EXISTS if a customer with the same ID already exists
- ERROR in case of database error

Room get_customer(customer_id: int)

Get a customer from the database.

Input: The id of the requested room

Output: The object of the requested customer if the customer exists, bad_customer() otherwise

ReturnValue delete_customer(customer_id: int)

Delete a customer from the database.

Input: The id of the customer to delete

Output: ReturnValue with the following conditions:

- OK in case of success
- BAD_PARAMS if any of the params are illegal
- NOT_EXISTS if the customer does not exist
- ERROR in case of database error

ReturnValue customer_made_reservation(customer_id: int, apartment_id: int, start_date: date, end_date: date, total_price: float)

Customer made a reservation of **apartment** from **start_date** to **end_date** an paid **total_price**

Input:

- customer_id: the id of the customer that made the reservation
- apartment_id: the id of the apartment the customer made a reservation at
- start_date: the date the reservation starts at
- end_date: the date the reservation ends at, has to be **after** start_date
- total_price: the total price of the reservation, has to be **positive**

Output:

- OK in case of success
- BAD_PARAMS if any of the params are illegal or the apartment isn't available at the specified date (there is already a reservation for it)
- NOT_EXISTS if the customer or the apartment don't exist
- ERROR in case of database error

ReturnValue customer_cancelled_reservation(customer_id: int, apartment_id: int, start_date: date)

Remove a reservation from the database.

Input:

- customer_id: the id of the customer that made the reservation
- apartment_id: the id of the apartment the customer made a reservation at
- start_date: the date the reservation starts at

Output:

- OK in case of success
- BAD_PARAMS if any of the params are illegal
- NOT_EXISTS if the customer or the apartment don't exist or there isn't a reservation at the given apartment on the given date
- ERROR in case of database error

ReturnValue customer_reviewed_apartment(customer_id: int , apartment_id: int, review_date: date, rating: int, review_text: str)

Customer reviewed **apartment** on date **review_date** and gave it **rating** stars, with text **review_text**.

Add a given review to the database. The customer must have a reservation at the apartment that ended before review_date in order to review the apartment. A customer may only review an apartment once.

Input:

- customer_id: the id of the customer that wrote the review
- apartment_id: the id of the reviewed apartment
- review_date: the date the review was published at
- rating: the rating the customer gave the apartment. Must be between 1 and 10
- review_text: the text the the customer wrote in the review

Output:

- OK in case of success
- BAD_PARAMS if any of the params are illegal
- NOT_EXISTS if the customer or the apartment don't exist or the customer doesn't have a reservation as described above
- ERROR in case of database error

ReturnValue customer_updated_review(customer_id: int, apartment_id:int, update_date: date, new_rating: int, new_text: str)

Customer decided to update their review of **apartment** on **update_date** and changed his rating to **new_rating** and the review text to **new_text**

Input:

- customer_id: the id of the customer that wants to update their review
- apartment_id: the id of the reviewed apartment
- update_date: the date the customer wanted to update the review. Must be after review_date in the original review.
- new_rating: the new rating the customer gave the apartment. Must be between 1 and 10
- new_text: the new text the the customer wants to appear in the review

Output:

- OK in case of success
- BAD_PARAMS if any of the params are illegal

- NOT_EXISTS if there is no review of the given apartment by the given customer or a review was given after update_date
- ERROR in case of database error

ReturnValue owner_owns_apartment(owner_id: int, apartment_id: int)

Owner owns **apartment**.

Input:

- owner_id: the id of the owner
- apartment_id: the id of the apartment

Output:

- OK in case of success
- BAD_PARAMS if any of the params are illegal
- NOT_EXISTS if the owner or the apartment don't exist
- ALREADY_EXISTS if the apartment already has an owner
- ERROR in case of database error

ReturnValue owner_drops_apartment(owner_id: int, apartment_id: int)

Owner dropped **apartment** and does not own it anymore.

Input:

- owner_id: the id of the owner
- apartment_id: the id of the apartment

Output:

- OK in case of success
- BAD_PARAMS if any of the params are illegal
- NOT_EXISTS if the owner or the apartment don't exist or the owner doesn't own the apartment
- ERROR in case of database error

Owner get_apartment_owner(apartment_id: int)

Get the owner of **apartment**.

Input: the id of the requested apartment

Output: the Owner object of the specified apartment, or bad_owner() if no one owns the apartment or it doesn't exist

List[Apartment] get_owner_apartments(owner_id: int)

Get a list of all **apartments** owned by **owner**.

Input: the id of an owner

Output: a list of Apartment objects of all apartments owned by the owner or an empty list if the owner doesn't exist

3.3 Basic API

float get_apartment_rating(apartment_id: int)

Get the average **rating** across all reviews of **apartment**.

Input: the id of an apartment

Output: the average rating of the given apartment

Note: You **must** use a view for this function and get_owner_rating

float get_owner_rating(owner_id: int)

Get the average of averages of ratings from all reviews of apartments owned by **owner**.

Input: the id of an owner

Output: the average of the all the ratings for all apartments owned by this owner, 0 if the owner doesn't exist or doesn't own any apartments

Note: You **must** use a view for this function and get_apartment_rating

Customer get_top_customer()

Get the **customer** that made the most **reservations**.

Input: None

Output: the customer that made the most reservations. If there is more than one customer with the maximal number of reservations, return the one with the **lower** id.

List[Tuple[str, int]] reservations_per_owner()

Input: none

Output: a list of tuples of (owner_name, total_reservation_count) of all owners in the database.

3.4 Advanced API

List[Owners] get_all_location_owners()

Return all owners that own an apartment in every city there are apartments in.

Input: None

Output: a list of all owners that own an apartment in every city there are apartments in (every city that appears with an apartment in our db)

Apartment best_value_for_money()

Get the apartment that has the best reviews compared to its average nightly price.

Input: none

Output: the apartment that has the height rating average divided by average reservation price per night (**not** weighted by the number of nights).

Example**:**

An apartment has one reservation from the 10/8 to the 15/8 with a total price of 1000, And has received one rating of 6. To calculate the value we first get the price per night which is 200. There is only one reservation so this is the average. We then divide the rating by that price to get: 6 / 200 = 0.03. We want the apartment that maximizes this value.

List[Tuple[int, float] profit_per_month(year: int)

The profit of the app each month in the given year.

Input: The year to sum the profit in

Output: the app's profit from a reservation is 15% from the total price. The function should return the profit per month in the specified year, **including months where no profit was made**. The month of the reservation should be determined by the end_date.

List[Tuple[Apartment, float]] get_apartment_recommendation(customer_id: int)

In this query you will need to approximate what the given customer will rate an apartment they haven't been in, based on their and other users' reviews.

You will use the following method for the approximation:

For every customer (other than the one you were given) that has reviewed an apartment the given customer also reviewed (or multiple apartments), you will calculate the ratio between their ratings (or the average of ratios if there is more than one apartment reviewed by both).

Then, for every customer you calculated a ratio for, the approximated ratio for an apartment they reviewed would be the ratio multiplied by the rating they gave the other apartment. If you get multiple approximations for the same apartment, return their average.

Generate an approximation for all apartments where it is possible.

Example: Itay reviewed apartment a1 and rated it 6 out of 10. Shir also reviewed apartment a1 and rated it 3 out of 10. Shir also reviewed apartment a2 and rated it 2 out of 10. When we run the query for Itay, the expected output would be the apartment a2 with a rating of 4. This is because the ratio between Itay and Shir based on a1 is 2, and Shir rated a2 2 out of 10. Multiplied by the ratio, we get an approximation of 4.

Input: customer_id: the id of the customer to get recommendations for

Output: A list of tuples of apartments and their projected rating. If the customer hasn't reviewed any apartments or doesn't exist (or if for any other reason no approximations could be calculated), return an empty list.

5. Database

5.1 Basic Database Functions

In addition to the above, you should also implement the following functions:

void createTables()

Creates the tables and views for your solution.

void clearTables()

Clears the tables for your solution (leaves tables in place but without any data).

void dropTables()

Drops the tables and views from the DB.

Make sure to implement them correctly.

5.2 Connecting to the Database Using Python

Each of you should download, install and run a local PosgtreSQL server from https://www.postgresql.org. You may find the guide provided helpful.

To connect to that server, we have implemented for you the DBConnector class that creates a *Connection* instance that you should work with to interact with the database.

For establishing successful connection with the database, you should provide a proper configuration file to be located under the folder Utility of the project. A default configuration file has already been provided to you under the name database.ini. Its content is the following:

[postgresql]
host=localhost
database=postgres
user=postgres
password=password
port=5432

Make sure that port (default: 5432), database name (default: cs236363), username (default: username), and password (default: password) are those you specified when setting up the database.

To get the Connection instance, you should create an object using:

```
conn = Connector.DBConnector()
```

(after importing "import Utility.DBConnector as Connector" as in Example.py).

To submit a query to your database, simply perform:

```
conn.execute("query here")
```

This function will return the number of affected rows and a ResultSet object.
You can find the full implementation of ResultSet in DBConnector.py.
Here are a few examples of how to use it:

```
conn = Connector.DBConnector()
query = "SELECT * FROM customers"
# execute the query, result will be a ResultSet object
rows_affected, result = conn.execute(query)

# get the first tuple in the result as a dictionary, the keys are the column
# names, the value are the values of the tuple
tuple = result[0]

# get the id column from the first tuple in the result
customer_id = result[0]['id']

# get the id column as a list
id_column = result['id']

# get the id of the first tuple
tuple_id = column[0]

# iterate over the rows of the result set. tuple is a dictionary, same as the
# example above
for tuple in result:
    # do something
    pass
```

This will return a tuple of (number of rows affected, results in case of SELECT).
Make sure to close your session using:

```
conn.close()
```

5.3 SQL Exceptions

When preparing or executing a query, an SQL Exception might be thrown. It is thus needed to use the try/catch (try/except in python) mechanism to handle the exception. For your convenience, the DatabaseException enum type has been provided to you. It captures the error codes that can be returned by the database due to error or inappropriate use. The codes are listed here:

NOT_NULL_VIOLATION (23502), *FOREIGN_KEY_VIOLATION*(23503), *UNIQUE_VIOLATION*(23505), *CHECK_VIOLIATION* (23514);

To check the returned error code, the following code should be used inside the except block: (here we check whether the error code *CHECK_VIOLIATION* has been returned)

```
except DatabaseException.CHECK_VIOLATION as e:
    # Do stuff
    print(e)
```

Notice you can print more details about your errors using print(e).

5.4 Tips

1. Create auxiliary functions that convert a record of ResultSet to an instance of the corresponding business object.
2. Use the enum type DatabaseException. It is highly recommended to use the exceptions mechanism to validate Input, rather than use Python's “if else”.
3. Devise a convenient database design for you to work with.
4. Before you start programming, think which Views you should define to avoid code duplication and make your queries readable and maintainable. (Think which sub-queries appear in multiple queries).
5. Use the constraints mechanisms taught in class to maintain a consistent database. Use the enum type DatabaseException in case of violation of the given constraints.
6. Remember - you are also graded on your database design (tables, views).

7. Please review and run Example.py for additional information and implementation methods.
8. AGAIN, USE VIEWS!

6. Submission

Please submit the following:

A zip file named <id1>-<id2>.zip (for example 123456789-987654321.zip) that contains the following files:

1. The file Solution.py that should have all your code (your code will also go through manual testing).
2. The file <id1>_<id2>.pdf in which should include detailed explanations of your database design and the implantation of the API (explain each function and view). You should explain your database design choices in detail. You are NOT required to draw a formal ERD but it is recommended.

You must use the check_submission.py script to check that your submission is in the correct format. Usage example:

```
python check_submission.py 123456789-987654321.zip
```

If your zip file is in the correct format you will get:

```
Success, IDs are: 123456789, 987654321
```

Or if you are submitting alone:

```
python check_submission.py 123456789.zip
```

And you should get:

```
Success, ID is: 123456789
```

Any other type of submission will fail the automated tests and result in 0 on the wet part.

You will not have an option to resubmit in that case!

Note that you can use the unit tests framework (unittest) as explained in detail in the PDF about installing IDE, but no unit test should be submitted.