TP

1 General instructions

This TP consists of 8 × slots of 1h30. In two-person teams, you will have to follow and answer the instructions that will be given to you later. The evaluation consists of:

- a small PDF report answering the questions that require so.
- the project source code.
- a 10min discussion with the teacher on October 9th (last TP slot).

2 Goal

Good news! You have just been hired as a junior data business analyst by a company that offers a flight price comparison service. To get you started, your senior manager has asked you to work on a simplified view of their company's data.

Your goal is to help them by creating an **REST API** (Application Programming Interface) to handle their data and access them. A REST API is a way to communicate with a server and access resources located in the server. For this REST API, all queries made to communicate to the server and for the responses will be in **HTTP**.

You will implement a **Flask** application to render the server, and **sqlite3** to handle the database (a skeleton of such an application will be provided to you).

3 Introduction

The flight price comparison company also offers its users the option of booking direct flights with various airlines directly on the website. Users have a username and password. They can book one or more direct flights. An airline has an ICAO code, a name, and a country of origin, and can organize one or more flights. An aircraft type has an ICAO code and a name that can be used during a flight. An airport has an ICAO code, a name, a type, a latitude, a longitude, and is linked to a city and a country. It can serve as the departure or arrival point for a flight. Countries have a name and an ISO code. Finally, a flight is organized

by an airline, necessarily departs from one airport to reach another by plane, has a price, a departure and arrival time.

4 Database design

Question 1 (Answer required)

From the model description above,

1. propose an ER diagram;
2. translate the ER diagram into a collection of relational tables (i.e., the physical model). For each table, specify the primary and foreign keys[1]. The proposed model should be **3NF**.

Ask your supervisor to validate the schema.

5 Data processing

In the **./data/** directory, you will find a big CSV file containing the data extracted from:

- Airports and countries data: `https://ourairports.com/`
- Cities: `https://www.kaggle.com/datasets/thedevastator/airport-city-mappings-and-aggregat`
- Airlines: `https://en.wikipedia.org/wiki/List_of_airline_codes`
- Airplanes: `https://github.com/jpatokal/openflights`

The flight data was generated.

Question 2

Given the data provided in the CSV file by the supervisors, identify the main entities and reorganize the data accordingly.

You may implements the function `split_data()` located in **./db/__init__.py** to return the separated data.

A little tip: you should probably use the pandas library to make your job easier.

Question 3

There is some useless columns, implement the function `transform_data()` to remove them.

[1] Warning: you may want to use ICAO or ISO codes as primary keys, but this is not a good idea as these attributes are partly linked to the place names given and may change over time.

6 Database initialisation

Question 4

Complete the function `load_config()` in **utils.py** to load the configuration of the database from the configuration file located in **./config/config**.

Question 5

Complete the function called `create_database()` in **./db/__init__.py** to create a database based on the model proposed before.

About **column types**:

- IDs must have the type INTEGER.
- Password must have the type BINARY(256).
- Any boolean must have the type INTEGER with either a 0 or a 1.

Then run the file **test_create_db.py** and verify the file **./data/flight.db** is created.

Question 6

Complete the Python function called `populate_database()` in **./db/__init__.py** to populate the database with csv files.

Question 7

Update `init_database()` in **./db/__init__.py** to populate the database during its creation.

7 First communications with the database

Question 8

Read the code and the comments of function `insert_user()` in **./db/users.py** to learn how to run a parameterized query on the database and how to handle the sqlite3.IntegrityError.

Question 9 (Answer required)

The file **test_users.py** is a file to test the functions implemented in **./db/users.py**. Run it and check whether Hubert has been added to the database by decommenting `print_users()` to list all the users in the database.

- How do you explain that you do not see Hubert in the list?

- By looking at code that we used to create the tables, how would you modify the code in **./db/users.py** in order to fix the problem?
- Rerun the code after the modification and verify that Hubert is now listed by `print_users()` .

Question 10 (Answer required)

Rerun the code. Which error do you get now? And why do you get it?

Question 11

Complete the function `update_password()` to update the password of a user.

Question 12 (Answer required)

What happens when you try to update the password of a user that is not in the database?

To help you understand, you can use the functions in **test_user.py**:

- `test_update_password_existing_user()`;
- `test_update_password_non_existing_user()`.

8 Database manipulation

Now that you can create new users, update their password and retrieve all information about them from the database, you can start to manipulate the database to handle deputies, bills, votes, and so on.

Question 13

Implement:

1. in **./db/countries.py**:
   - `get_all_countries(cursor: Cursor) -> list[dict]`: returns all countries.
   - `get_all_countries_sorted_by_name(cursor: Cursor, ascending: bool) -> list[dict]`: returns all countries from the database, sorted by their name.
   - `get_country_name_by_code(cursor: Cursor, iso_code: str) -> str | None`: returns the city name according to the ISO code, returns `None` if it does not exist.
2. in **./db/cities.py**:

- `insert_city(cursor: Cursor, city_name: str, iso_country: str) -> bool`: adds a new city linked to a country with the ISO code, returns `True` on success, `False` otherwise.
- `get_city_id_by_name_and_country_code(cursor: Cursor, name: str, iso_code: str) -> int | None` Retrieves the ID of a city based on its name and the ISO code of its country.
- `get_city_name_by_id(cursor: Cursor, id: str) -> str | None`: returns the name of city according to its id, returns `None` if it does not exist.
- `update_city_name(cursor: Cursor, id: str, new_name: str) -> bool`: updates the city name according to the id, returns `True` on success, `False` otherwise.

3. in **./db/airport.py**:
   - `get_airport_id_by_icao(cursor: Cursor, icao: str) -> int | None`: retrieves the airport ID corresponding to the given ICAO code. Returns the ID if found, otherwise `None`.
   - `get_all_airport_info_by_id(cursor: Cursor, id: int) -> dict | None`: retrieves all available information about an airport based on its ID. Returns a dictionary with the airport details, or `None` if the airport does not exist.
4. in **./db/airlines.py**:
   - `get_airlines_by_country(cursor: Cursor, iso_code: str) -> list[dict] | None`: retrieves a list of airlines associated with the given country ISO code, returns `None` if no airlines are found.
   - `get_airline_by_id(cursor: Cursor, id: str) -> dict | None`: returns a dictionary with the airline's information for the given ID, returns `None` if the airline does not exist.
   - `remove_airline_by_id(cursor: Cursor, id: str) -> bool`: removes an airline with the specified ID from the database, returns `True` on success, `False` otherwise.
5. in **./db/flights.py**:
   - `insert_flight(cursor: Cursor, departure: str, arrival: str, price: float, src_icao: str, dest_icao: str, plane_icao: str, airline_icao: str) -> bool`: inserts a new flight with the specified details. Returns `True` on success, `False` otherwise.
   - `get_flight_by_id(cursor: Cursor, flight_id: int) -> dict | None`: retrieves a flight's details by its ID, including departure/arrival times, price, source/destination airport, plane, and airline. Returns `None` if not found.

- `remove_flight_by_id(cursor: Cursor, flight_id: int) -> bool:` removes a flight by its ID. Returns `True` if a flight was deleted, `False` otherwise.
- `get_flights_by_src_city_and_dest_city(cursor: Cursor, src_city: str, dest_city: str) -> list[dict] | None)` retrieves all flights departing from `src_city` and arriving at `dest_city`. Returns a list of dictionaries containing flight details (departure/arrival times, price, source/destination airport, plane, and airline) or `None` if no flights are found.

6. in **./db/bookings.py**

   - `create_booking(cursor: Cursor, user_name: str, flight_id: int) -> bool:` creates a booking for a user on a given flight. Returns `True` if the booking is successfully created, `False` otherwise.
   - `get_usernames_by_flight(cursor: Cursor, flight_id: int) -> list[str] | None:` retrieves all usernames for a specific flight. Returns a list of str or `None` if no bookings exist.
   - `get_bookings_by_user(cursor: Cursor, user_name: str) -> list[dict] | None:` returns a list of dictionaries containing the details of the flights booked by the user, or `None` if the user has no bookings. Each dictionary includes the flight price, departure and arrival times, airline name, and the names of the departure and arrival cities.
   - `delete_booking(cursor: Cursor, user_name: str, flight_id: int) -> bool:` deletes a specific booking for a user on a flight. Returns `True` if the deletion succeeds, `False` otherwise.

9 Create the first API routes

In this section, you have to create the first routes of the API.

Question 14 (Answer required)

Take a look at the file **app.py** and read the code and comments. To start it, run:

```
$ python app.py
```

What does the application do when you run it?

Question 15 (Answer required)

So far, the application only render routes to handle information about the users. The routes are partially defined in **./routes/users.py** as well as the code executed when you query one of them.

- What is the path of the route to get all users?
- What is the type of HTTP request you can make to access it?

Question 16

Create a script in Python to make the following query to the database using the library **request**:

```
GET localhost:5000/users/
```

In the request, "localhost" is the address and 5000 is the port number

Question 17

Complete the functions in **./routes/users.py**:

- `get_user()` to retrieve a user based on its username as well as the list of their flights.
- `patch_password()` to update the password of a user.
- `add_user()` to add a new user to the database.

10 Help to debug

Question 18

In the file **./routes/data.py**, you can find the prototypes of functions to help you debug the database that will be useful in the following questions. Implement them based on the specifications given in the function comments.

11 Remaining routes

Question 19

Implement all functions to handle booking **./routes/booking.py** and flights **./routes/flights.py**. Implement the remaining functions from **./routes/users.py**.

12 Securing the API

With the previous question, you have now an API to handle booking, flights and users. However, it lacks of security. First, the **password** are stored in plain text, which is definitely not a good practice. Moreover, the routes are not secured as **no authentication** is required to make calls to the API you created. Finally, all **communications are in clear** because we use the HTTP protocol which is not secure. In a real life API, all communications between the

client and the API server must be encrypted using the HTTPS protocol. In this section, we will focus on the problem of password stored in plain text and the authentication of the user making the queries.

Question 20

A good practice when creating a database is to store the password hashes rather than the password itself. To do this, Python provide the `bcrypt` library you can install in your virtual environment using:

```
$ python -m install bcrypt
```

Now, using the documentation of **bcrypt**, complete the functions in **utils.py**:

- `hash_password()` to generate the hash of a password
- `check_password()` to check whether a password provided as plain text is conform to a hash value
- `check_user()` to check whether the user username supplied with a plain text password corresponds to an user in the database and whether the password is correct.

Question 21

Update function **insert_user()** in **./db/users.py** to store the hash of the password in the database instead of the plain text value.

Question 22

The second security issue you need to tackle is the authentication. A good practice to handle this issue is to use tokens. The idea is to make an initial request with your password and username to recover a token. Then, add the token to the headers of your HTTP request. For example, to query the user list using a token, the query is:

```
GET localhost:5000/users/ HTTP/1.1
Authorization: Bearer <token>
```

First, add the PyJWT library to the virtual environment:

```
$ python -m install PyJWT
```

Then, implement in **utils.py** the functions `generate_token()` and `check_token()`.

Question 23

To enable a user to get a token, implement the route "/login" in **./routes/auth.py**.

Question 24

Look at the code in **./routes/auth.py** for the function `token_required()`. The function creates a decorator that can be added to routes to ensure a token is provided and verified before rendering the data.
Add this decorator to the routes you think requires protection and justify it.