Assignment 3 - MongoDB

Submission instructions

The assignment should be submitted in pairs to Moodle. You should submit one zip file. The archived zip file should contain all the project files. Do not include venv or git directories in the zip.

The archive file name should be of form id1_id2.zip.
Deadline: 09.01.2025

Assignment Instructions:

In this assignment, you will be writing a Python script that will manage a game rental system utilizing MongoDB as the NoSQL database. The system will include user registration, login, game recommendation, and other functions. Please download the hw3.py file from Moodle and complete the functions using MongoDB. Only use the attached template and do not change signatures!

a. Function register_user:

```
def register_user(self, username: str, password: str)-> None
```

Add a new user to users collection. U need to hashe the password before storing it with bcrypt library. Use that same salt every time (can be class variable), it can be generated using bcrypt.gensalt() .

Parameters:
- username (str): The username for the new user.
- password (str): The password for the new user.

Returns:
- None

Instructions:
1. Check if the username and password are not empty strings. If either is empty, raise "Username and password are required." ValueError.
2. Check if the length of both username and password is at least 3 characters. If not, raise "Username and password must be at least 3 characters." ValueError.
3. Check if the username already exists in the database. If it does, raise "User already exists: {username}." ValueError.
4. If the validation above has not failed, hash the provided password using bcrypt and create a new user in the database.

b. Function login_user:

```
def login_user (self, username: str, password: str) -> object
```

Log in a user with the provided username and password.

Parameters:
- username (str): The username of the user trying to log in.
- password (str): The password of the user trying to log in.

Returns:
- object: User object if login is successful, otherwise None.

Instructions:

1. Hash the provided password using bcrypt (with same salt as before).
2. Query the MongoDB collection to find a user with the provided username and hashed password.
3. If a user is found, print "Logged in successfully as: {username}" and return the user object.
4. If no user is found or the password doesn't match, raise "Invalid username or password" ValueError.

c. Function load_csv:

def load_csv (self)-> None

Loads data from a CSV file into games collection.
It is assumed that this function will be executed once at first.

Parameters:

- None

Returns:

- None

Instructions:

1. Load the CSV file "NintendoGames.csv".
2. Insure "genres" field will store as list (u can use ast.literal_eval for this).
3. Add "is_rented" field with the values False.
4. Insert all the records into games collection.
5. Insure that using this function few times in a row will not insert duplicate items to the collection.

d. Function rent_game:

def rent_game(self, user: dict, game_title: str) -> str

Rents a game for the user.
Parameters:

- user (dict): The user object.
- game_title (str): The title of the game to be rented.

Returns:

- str: "success" if the rental process was successful, and "failure" if the rental process failed.

Instructions:

1. Query the game collection to find the game with the provided title.
2. If the game is found:
   - Check if the game is not already rented.
   - If not rented, mark the game as rented in the game collection and add it to the user's rented games ids list (as an id, not object. Then, when implementing the following functions, you will need to use the find and aggregation functions to find the games rented by the stored IDs, and not loop through all the existing games.).
   - Return "{game_title} rented successfully".
3. If the game is not found, return "{game_title} not found".
4. If the game is already rented, return "{game_title} is already rented".

e. Function return_game:

def return_game(self, user: dict, game_title: str) -> str
Returns a rented game.

Parameters:

- user (dict): The user object.
- game_title (str): The title of the game to be returned.

Returns:

- str: "success" if the return process was successful, and "failure" if the return process failed.

Instructions:

1. Get the list of games ids rented by the user from the user object.
2. If the game with the provided title is rented by the user:
    - Remove the game id from the user's rented games ids list.
    - Mark the game as not rented in the game collection.
    - Return "{game_title} returned successfully".
3. If the game is not rented by the user, return "{game_title} was not rented by you".

f. Function recommend_games_by_genre:

def recommend_games_by_genre(self, user : dict) -> list

Recommends games based on the user's rented game genre. Don't recommend games that are already owned, it is allowed to recommend games that are rented by others.

Parameters:

- user (dict): The user object.

Returns:

- list of strings: A list containing recommended game titles based on genre.

Instructions:

1. Get the list of games rented by the user from the user object.
2. If no games are rented, return "No games rented".
3. Select a genre randomly from the pool of rented games, taking into account the probability distribution. For instance, if there are three games categorized as "shooter" and one as "RPG," the likelihood of selecting "shooter" would be 75%.
4. Query the game collection to find 5 random games with the chosen genre.
5. Return the titles as a list with 5 random games.

g. Function recommend_games_by_name:

def recommend_games_by_name(self, user: dict) -> str

Recommends games based on random user's rented game name. Don't recommend games that are already owned, it is allowed to recommend games that are rented by others.

Parameters:

- user (dict): The user object.

Returns:

- list of strings: A list containing recommended game titles based on similarity.

Instructions:

1. Get the list of games rented by the user from the user object.
2. If no games are rented, return "No games rented".
3. Choose a random game from the rented games.
4. Compute TF-IDF vectors for all game titles and the chosen title (u can use TfidfVectorizer from sklearn library).
5. Compute cosine similarity between the TF-IDF vectors of the chosen title and all other games (u can use cosine_similarity from sklearn library).
6. Sort the titles based on cosine similarity and return the top 5 recommended titles as a list (from most similar to different - at index 0 the most similar).

h. Function find_top_rated_games:

def find_top_rated_games(self, min_score) -> list

Returns games with a user score higher than or equal to the score received

Parameters:

- min_score(double).

Returns:

- List: list of all games (only titles and user scores) with a user score of at least min_score. Each game is represented as a dictionary and the return format is for example: [{'title': 'A', 'user_score': 9.0}, {'title': 'B', 'user_score': 8.9}]

Instructions:

1. Use the find function to find all games with a user_score of at least min_score
2. Return the list of games so that each game returned only shows its title and user_score

i. Function decrement_scores:

def decrement_scores(self, platform_name) -> None
Lowers user score by 1 for games whose platform is the received platform name. It can be assumed that platform names whose user scores for their games are at least 1 will be received.

Parameters:

- platform_name(str) -the name of the gaming platform.

Returns:

- None

Instructions:

1. Use update_many function.
2. All games whose platform is the platform passed as an argument should be updated to have a user score 1 lower than it was before.

j. Function get_average_score_per_platform:

def get_average_score_per_platform(self) -> dict

Calculate the average user score for games on each platform. The function returns a dictionary that maps each platform to its average user score.

Parameters:

- None.

Returns:

- Dictionary that maps each platform name to its average user score.

Instructions:

1. Use aggregation.
2. Group the records according to the platform it belong to.
3. Computers per platform average user score
4. Returns a dictionary that maps the average user score to each platform.

k. Function get_genres_distribution:

def get_genres_distribution (self) -> dict

Count the number of games in each genre. The function returns a dictionary that maps each genre to its amount games.

Parameters:

- None.

Returns:

- Dictionary that maps each genre to its amount games.

Instructions:

1. Use aggregation.
2. The genera arrays should be broken down into individual documents.
3. Group the records according to the genres to which they belong.
4. For each genre, the number of games in which it appears is considered.
5. Returns a dictionary that maps the number of games to each genre.

Returns a dictionary that maps the amount games to each genre.

Good Luck,
Yarin Benyamin.