Databases Project
2022/2023

(2023/02/25: versão 1.0 do enunciado; pequenos ajustes podem ser realizados nos próximos dias)

Introduction

The goal of this project is to provide students with experience in developing database-centric systems. The project is guided by the industry's best practices for software development; thus, students will experience the main stages of a common software development project, from the very beginning to delivery.

Objectives

After completing this project, students should be able to:

- Understand how a database application development project is organized, planned, and executed
- Master the creation of conceptual and physical data models for supporting application data
- Design, implement, test, and deploy a database system
- Install, configure, manage, and tune a modern relational DBMS
- Understand client and server-side programming in SQL and PL/pgSQL

Groups

The project is to be done in **groups of 3** students.
IMPORTANT: the members of each group **must** be enrolled in PL classes with the same Professor.

Quality attributes for a good project

Your application must make use of:

(a) A transactional relational DBMS (e.g., PostgreSQL)
(b) A distributed database application architecture, providing a REST API
(c) SQL and PL/pgSQL, or similar
(d) Adequate triggers and functions/procedures running on the DBMS side
(e) Good strategies for **managing transactions** and **concurrency conflicts**, and **database security**
(f) Good **error avoidance, detection, and mitigation strategies**
(g) Good documentation

Your application must also respect the functional requirements defined in **Annex A** and execute without "visible problems" or "crashes". To fulfill the objectives of this assignment, you can be as creative as you want, provided you implement the list of required features.

Do not start coding right away - take time to think about the problem and structure your development plan and design.

Plagiarism or any other kind of fraud will not be tolerated

Milestones and deliverables

Midterm presentation (20% of the grade) – 23:55 of March 24th

The group **must present** their work in the PL classes (privately, to the professor of the class). It is necessary to sign up for an available time slot for the defense in *Inforestudante*, before the delivery due date. The list of available slots will be released before the deadline. The following artifacts must be uploaded at *Inforestudante* until the deadline (one member of the group uploads the artifact but all students must be associated with the submission):

- **Presentation** (e.g., powerpoint) with the following information:
    - Course/Discipline identification
    - Name of the project
    - Team members and contacts
    - Brief description of the project
    - Definition of:
        - The main database operations (inserts, updates, deletes)
        - Transactions (where the concept of transactions is particularly relevant, multiple operations that must succeed or fail as a whole)
        - Potential concurrency conflicts (those that are not directly handled by the DBMS) and the strategy to avoid them
    - Development plan: planned tasks, initial work division per team member, timeline
- **ER diagram image**
    - Description of entities, attributes, integrity rules, etc.
- **Relational data model image**
    - The physical model of the database (i.e., the tables)
- **The ONDA project, exported as a JSON file.**

Final delivery (80% of the grade) – 23:55 of May 19th

The project outcomes must be submitted to *Inforestudante* by the deadline. Each group must select a member for performing this task. All submissions must clearly identify the team and the students that compose the group working on the project. Upload the following materials at *Inforestudante*:

- **Final report** with (the following items in a single document):
    - **Installation manual** describing how to deploy and run the software you developed
    - **User manual** describing the submitted Postman collection requests to test the application
    - **Final ER and relational data models**
    - **Development plan:** make sure you specify which tasks were done by each team member and the effort involved (e.g., hours)
    - **All the information, details, and design decisions you consider relevant to understand how the application is built and how it satisfies the requirements of the project**
- **Source code and Scripts:**
    - Include the source code, scripts, executable files ,and libraries necessary for compiling and running the software
    - DB creation scripts containing the definitions of tables, constraints, sequences, users, roles, permissions, triggers, functions, and procedures
    - Postman collection with all the requests required to test the application
    - The ONDA project, exported as a JSON file.

Defense – May 29th to June 2nd

- Prepare a 5-minute live presentation of your software
- Prepare yourself **(individually)** to answer questions regarding all deliverables and implementation details
- Sign up for any available time slot for the defense in *Inforestudante,* before the delivery due date The list of available slots will be released before the deadline for the final delivery

Alternative Deadline

This year, exceptionally, there is an alternative deadline for the "*Final delivery"*, which can be made until **23:55 of the 16th of June**. The defenses for this deadline will occur during the week of the **19th of June.**

IMPORTANT: Notice that you can only submit your work once! Also, if you submit on this alternative deadline, you will not gather the necessary conditions to be approved at the Normal Exam! You will only have such conditions for the Retake Exam (Exame de Recurso).

Assessment

- This project accounts for 8 points (out of 20) of the total grade in the Databases course
- Midterm presentation corresponds to 20% of the grade of the project
- Final submission accounts for the remaining 80% of the grade of the project
- The minimum grade is 35%

Annex A: Music Streaming Platform – Functional Description

This project aims at developing a simplified music streaming platform. The development of the database should fit the required functionalities and business restrictions to ensure effective storage and information processing.

Briefly, the platform has 3 types of users:

- **Consumer**: has permission to listen to music tracks. S/he can be either a *normal* consumer or a *premium* consumer. Consumers are characterized by several personal details (e.g., personal information, address, …) as well as details concerning their activity and subscriptions.
- **Artist**: has permission to publish songs and albums. An artist has personal details like the consumer, but also has specific attributes (e.g., artistic name) and relations with other aspects of the system (e.g., labels/publishers).
- **Administrator**: has permission to create artists, pre-paid cards, (…).

The platform should provide streaming services to both *regular* and *premium* consumers. Regular consumers have free access to the system, while premium consumers pay a subscription. All users need to be registered to use the platform. Everyone can register as a consumer. Artists must be created by an administrator. Administrators are manually added to the database. At any point, consumers can change their subscription from regular to premium. When the subscription period ends, the premium consumer becomes a regular consumer until a new subscription is made. The system should keep the subscription history of every consumer.

A premium consumer can create playlists, either public or private (regular consumers cannot create playlists). Public playlists are available to everyone, while private playlists are available only to the consumer that create them. If a premium consumer becomes a regular one, the public playlists remain available but private playlists are no longer accessible (they become available if the consumer becomes premium again).

The platform should implement a "comments" system, where consumers can leave their comments/feedback on a song. It should be possible for other consumers to comment/reply to existing comments.

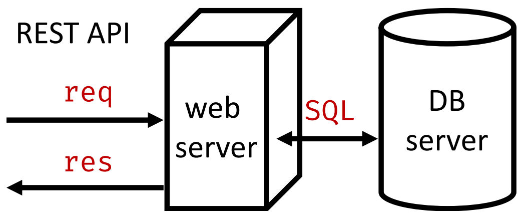
Artists have relationships with record labels (each label can have multiple artists). Artists can publish their songs on the platform. They can also publish albums (sets of songs, with order) and each song can be associated with multiple artists. Still, the artist that published each song must be identifiable. Songs have various details such as ISMN (an identifier that uniquely identifies each song), label/publisher, genre, title, release date, and duration, among others. Artists cannot be consumers.

The system should create and maintain a *top 10* playlist for each consumer (in descending order based on the number of times each music was heard in the last 30 days). This playlist should be updated every time the consumer reproduces a song. This mechanism should be implemented using *triggers*.

The subscriptions are paid through *pre-paid* cards. Only administrators can generate them, and they are issued with a 16-digit unique random identifier and a limit date for their use. The cards have fixed values of 10, 25, or 50 euros. Three subscription plans are available: month, quarter, or semester, which cost 7, 21, and 42 euros, respectively (values may change in the future). When making a subscription the consumer can associate more than one card for the purchase. Because the values of the pre-paid cards do not match the values of the subscriptions, a card can be used more than once but always by the same consumer. The details of the transaction/purchase should be kept for future analysis (e.g., which subscription periods were paid by each card).

Technical Description

The figure represents a simplified view of the system to be developed. The system must be made available through a REST API that allows the user to access it through HTTP requests (when content is needed, JSON must be used). As it can be seen, the user interacts with the web server through a REST request/response exchange, and in turn, the web server interacts with the database server through an SQL interface (e.g., Psycopg2 in the case of Python). This is one of the most used architectures today and supports many web and mobile applications we use daily. Since the course focuses on the design and operation of a data management system and associated functionalities, the development of web or mobile applications is outside the scope of this work. To use or test the REST API, you must use the REST client Postman (postman.com).



To facilitate the development of the project and focus on the data management system, groups should follow/use the demo code that will be made available. An example of how to test the REST API endpoints using Postman is also provided. In the event of missing details, groups should explicitly identify them in the report.

HTTP works as a request-response protocol. For this work, three main methods might be necessary:

- **GET**: used to request data from a resource
- **POST**: used to send data to create a resource
- **PUT**: used to send data to update a resource

REST API responses should follow a simple but rigid structure, always returning a status code (this is a very simplified version of how a real REST API would work):

- 200 – success
- 400 – error: bad request (request error)
- 500 – error: internal server error (API error)

If there are errors, they must be returned in *errors*, and if there are data to be returned, they must be returned in *results* (as can be seen in the details of the endpoints that follow).

Functionalities to be developed

The *endpoints* of the REST API should be **strictly followed**. Any details that are not defined should be explained in the report.

When the user starts using the platform, he/she can choose between registering a new account or logging in using an existing account. The following *endpoints* should be used.

***NOTICE**: This is a simplified scenario. In a real application, a secure/encrypted connection would be used to send the credentials (e.g., HTTPS).*

User Registration. Create a new user, inserting the data required by the data model. Everyone can register as a consumer, which is considered *regular* until they use their pre-paid card to become a *premium* user. Artists can only be created by admins (the admin authentication token should be passed in the request).

```
Req   POST http://localhost:8080/dbproj/user
      {“username”: username, “email”: email, “password”: password, (…)}
Res   {“status”: status_code, “errors”: errors (if any occurs), “results”: user_id (if it
      succeeds)}
```

User Authentication. Login using the *username* and the *password* and receive an *authentication token* (e.g., JSON Web Token (JWT), https://jwt.io/introduction) in case of success. This *token* should be included in the *header* of the remaining requests.

```
req   PUT http://localhost:8080/dbproj/user
      {“username”: username, “password”: password}
res   {“status”: status_code, “errors”: errors (if any occurs), “results”: auth_token (if it
      succeeds)}
```

After the authentication, the user can perform the following operations using the obtained *token* during the user authentication (the token should always be passed in every request, either in the body or in authentication headers):

Add song. Artists can publish their songs on the platform. They can provide additional information about the other artists that also perform in the song.

```
req   POST http://localhost:8080/dbproj/song
      {“song_name”: “name”, “release_date”: “date”, “publisher”: publisher_id,
      “other_artists”: [artist_id1, artist_id2, (…)]}, (…)}
res   {“status”: status_code, “errors”: errors (if any occurs), “results”: song_id (if it
      succeeds)}
```

Add album. Artists can publish their albums on the platform. It is possible to pass details of new songs as well as identifiers of existing songs already in the platform (the system should validate if the artist is associated with the selected existing songs).

```
req   POST http://localhost:8080/dbproj/album
      {“name”: “album_name”, “release_date”: “date”, “publisher”: publisher_id,
      “songs”: [
        {“song_name”: “name”, “release_date”: “date”, “publisher”: publisher_id,
      “other_artists”: [artist_id1, artist_id2, (…)], (…)},
        existing_song_id2,
      (…)]
      }
res   {“status”: status_code, “errors”: errors (if any occurs), “results”: album_id (if it
      succeeds)}
```

Search song. Retrieves all songs that contain the keyword provided by the user. Just **one** *SQL* query should be used to obtain the results data.

```
req   GET http://localhost:8080/dbproj/song/{keyword}
res   {“status”: status_code, “errors”: errors (if any occurs), “results”: [{“title”:
      “song_title”, “artists”: [“artist_name1”, “artist_name2”, (…)], “albums”: [“album_id1”,
      “album_id2”, (…)]}, (…)]}
```

Detail artist. Lists all relevant information about an artist. All the songs, albums, and public playlists where the songs are should be retrieved. Just **one** *SQL* query should be used to obtain the results data.

```
req   GET http://localhost:8080/dbproj/artist_info/{artist_id}
res   {“status”: status_code, “errors”: errors (if any occurs), “results”: {“name”:
      “artist_name”, “songs”: [“song_id1”, “song_id2”, (…)], “albums”: [“album_id1”,
      “album_id2, (…)], “playlists”: [“playlist_id1”, “playlist_id2”, (…)]}}
```

Subscribe to Premium. The consumer provides the pre-paid card number(s) and the period to be subscribed as premium. If the consumer has an active subscription, the new subscription period should be placed at the end of the current subscription. The information on how much was paid from each card for each subscription period should be stored.

```
req   POST http://localhost:8080/dbproj/subcription
      {“period”: “month” | “quarter” | “semester”, “cards”: [card_number1, …]}
res   {“status”: status_code, “errors”: errors (if any occurs), “results”: subscription_id (if
      it succeeds)}
```

Create Playlist. Creates a new playlist of a consumer that is logged in. Only premium consumers can create playlists.

```
req   POST http://localhost:8080/dbproj/playlist
      {“playlist_name”: “name”, “visibility”: “public” | “private”, “songs”: [song_id,
      song_id2, (…)]}
res   {“status”: status_code, “errors”: errors (if any occurs), “results”: playlist_id (if it
      succeeds)}
```

Play song. Consumers can "play" a song (no streaming is expected, just assume this would stream/listen to the music). Played songs should be stored and trigger an update to the top 10 playlist of most played songs (of the logged consumer).

```
req   PUT http://localhost:8080/dbproj/{song_id}
res   {"status": status_code, "errors": errors (if any occurs)}
```

Generate pre-paid cards: the administrator can generate the cards of one of the three possible values: 10, 25, or 50 euros. The number of cards to be created should also be provided. The information of which admin created each card should be stored.

```
req   POST http://localhost:8080/dbproj/card
      {"number_cards": "number", "card_price": 10 | 25 | 50}
res   {"status": status_code, "errors": errors (if any occurs), "results": [id_card1, (…)] (if
      it succeeds)}
```

Leave comment/feedback – It should be possible to leave a comment/feedback on a song. It should be possible to reply/answer to existing comments.

```
req   POST http://localhost:8080/dbproj/comments/{song_id}
      POST http://localhost:8080/dbproj/comments/{song_id}/{parent_comment_id} (if replying to
      an existing comment)
      {"comment": "comment_details"}
res   {"status": status_code, "errors": errors (if any occurs), "results": comment_id (if it
      succeeds)}
```

Generate a monthly report. It should be possible to list the number of songs played per month and genre in the past 12 months. Just **one** *SQL* query should be used to obtain the information.

```
req   GET http://localhost:8080/dbproj/report/topN/{N}/{year-month}
res   {"status": status_code, "errors": errors (if any occurs), "results": [
         {"month": "month_0", "genre": "genre1", "playbacks": total_songs_played},
         {"month": "month_0", "genre": "genre2", "playbacks": total_songs_played},
         {"month": "month_1", "genre": "genre1", "playbacks": total_songs_played},
      (…)
      ]}
```

FINAL REMARKS

- As it is defined in some *endpoints*, the implemented solution should obtain the information using a single SQL *query* on the server side. This does not include potential steps to validate the user authentication. **Nevertheless**, in case you cannot solve it with a single query, it is preferable to use more *queries* than not implementing the *endpoint*.
- The data processing (e.g., order, restrictions) should be done in the *queries* whenever possible (and not in code).
- The transaction control, concurrency, and security will be considered during the evaluation.