U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Bases de Dados

Management of gaming streams

2MIEIC01 - Group 106

May 24, 2020

Muriel Pinho up201700132@fe.up.pt

Description

This project is based on the management of a gaming stream service, similar to *Twitch*.

The system is divided into eight classes, these classes are:

- **User:** it stores all data related to the user, like his id, username, password, profilePicture, birthDate, email and age. The **User** can send and receive a **Message**, he can follow multiple channels, own only one **Channel** and he can also follow a specific **Game**.

- **Stream:** the stream is the heart of this service, A stream is a live transmission made on a **Channel** owned by a **User** playing a **Game** for other users to watch. A Stream stores a title, startTime, uptime, ageRestriction and viewerCount. A **Stream** has one **Game** being played at a time, but the user can play multiple games over the course of the stream. A stream can also have Tags, describing what the stream is about, with all streams having at least one **Tag**.

- **Message:** it stores the data from messages sent by the **User**, storing the fields content and dateSent.

- **Channel:** a channel is owned by one user and it is where the user can create an **Stream**, the channel stores a biography and followerCount. A **Channel** can have multiple streams but all streams must be streamed on only one channel.

- **Game:** the game class stores all the information related to the games streamed on the platform, a game has a title, followerCount, description and totalViewers.

- **Tag:** a tag is used to describe a **Stream**, it only has a title field and the same tag can be used on multiple streams at the same time.

- **Viewership:** It is used to generate a list that contains the streams from a specific **Game** ranked in order based on the viewerCount from that **Game**.

- **Genres:** it is identical to the **Viewership** class, the only difference is that the ranking is based on a **Tag** instead of a **Game**.

Relational Diagram and Functional Dependencies

User *(<u>userID</u>, username, password, profilePicture, birthDate, email, age)*

1. {userID} -> {username, password, profilePicture, birthDate, email, age}

Message *(<u>messageID</u>, content, dateSent)*

1. {messageID} -> {*content, dateSent*}

Channel *(<u>channelID</u>, biography, followerCount)*

1. {channelID} -> {biography, followerCount}

Stream *(<u>streamID</u>, title, startTime, uptime, ageRestriction, viewerCount)*

1. {streamID} -> {title, startTime, uptime, ageRestriction, viewerCount}

Tag *(<u>tagID</u>, title)*

1. {tagID} -> {title}
2. {title} -> {tagID}

Game *(<u>gameID</u>, title, followerCount, description, totalViewers)*

1. {gameID} -> {title, followerCount, description, totalViewers}

Viewership *(<u>gameID</u>->Game, <u>streamID</u>->Stream, viewerCount, position)*

1. {streamID, gameID} -> {viewerCount, position}
2. {gameID, position} -> {id}

Genres *(<u>genresID</u>->Tag, <u>streamID</u>->Stream, viewerCount, position)*

1. {streamID, genresID} -> {viewerCount, position}
2. {genresID, position} -> {id}

Sends *(messageID->Message, userID->User)*

Receives *(messageID->Message, userID->User)*

Follows *(userID->User, channelID->Channel)*

Owns *(userID->User, channelID->Channel)*

ChannelStream *(channelID->Channel, streamID->Stream)*

StreamGame *(streamID->Stream, gameID->Game)*

StreamTag *(streamID->Stream, tagID->Tag)*

UserGame *(userID->User, gameID->Game)*

Normal Forms

To identify if there is a violation on the Third Normal Form we can check if the no transitivity rule is broken, in case it is there will be a violation on Boyce-Codd Normal Form as well, because BCNF is a more restrict version of 3NF.

To all enumerated relations there isn’t any violation of the transitivity rule, and all the left side elements from the dependencies are super keys from the relational scheme. As such all relation oblige to the Third Normal Form and Boyce-Codd Normal Form.

Restrictions

The restrictions were used to maintain a secure and accurate database, the ones we used were FOREIGN KEY , NOT NUL and UNIQUE.

- **NOT NULL** was used in most essential attributes in order to maintain the database working correctly, some examples are:

1. The username attribute from the user class, were a user needs a username or else the system isn't able to identify him.
2. The pass attribute from the user class, were a user needs a password or else the user isn't able to be logged in the system.
3. The title attribute from stream, a stream needs to have a title to be able to be identified by viewers.
4. The userID attribute from channel, a channel must have a user that owns it and the userID serves to identify him.
5. All the ID attributes from the classes, all classes need an ID so that each unique entry is able to be identified even if it has the same values as another one.

- **UNIQUE** was used in attributes that were used as keys in the class, some examples are:

1. The name attribute on a channel, each channel has to have a unique name.
2. The title attribute on a game, each game has to have a unique title.
3. The name attribute on a genre, each genre must have an unique name.
4. The username attribute on user, each username must be used by a single user.
5. The email attribute on user, each email must be used by a single user.

- **FOREIGN KEY** was used in attributes that were directly related to other classes and were their value was of use on another class, some examples are:

1. The userID attribute on the channel class, a channel must be owned by an existing user.
2. The channelID attribute on the stream class, a stream must have an existing channel to be streamed.
3. The gameID attribute on the stream class, a stream must have an existing game to be streamed.
4. The userID attribute on the sends class, a message must be sent by an existing user.
5. The streamID on the genres class, a genre must have an existing stream to be added to the genres class.

Queries

The following queries have different complexity levels and facilitate getting information from a database with thousands of entries.

1. **Ranking of the channels by follower count:** counts entries in the follows table and groups them by channel on a decrescent order.

2. **Total hours each game was streamed on the platform:** adds up the total uptime from streams with the same gameID and groups them by title.

3. **Numbers of streams started during the morning:** counts entries in the stream table where a stream was done during the morning , using 8 AM and 12 AM as boundaries.

4. **Hours a streamer has streamed a specific game:** adds up the total time a streamers has streamed a specific game, inner joining the channel, stream and game tables. The values chosen were the game Valorant and the streamer Gaules.

5. **User that has received the most messages:** counts the number of messages received by the user that is the number one receiver.

6. **Data about user age on the platform:** displays the average, maximum and minimum user age on the platform.

7. **Total hours a channel has streamed:** adds up the total uptime from streams done for each streamer.

8. **Top 3 followed games:** ranks the top 3 followed games on the platform using inner join, group by, order by and limit as tools.

9. **Average duration of a nighttime stream:** displays the average duration of a nighttime stream on the platform, using 6 PM and 12 PM and boundaries.

10. **Number of channels each streamer follows:** counts the number of user each streamer follows, using inner join and group by.

Triggers

I implemented 6 triggers that are useful for managing the database, with a pair in each file.

1. calculateAge and validateAge

These triggers serve as a control mechanism for the birthdate and age fields from user, when there is an attempt of adding a user to the database the validateAge trigger checks to see if the user is older than 13 years old, if he is then the calculateAge sets the age field on the user entry.

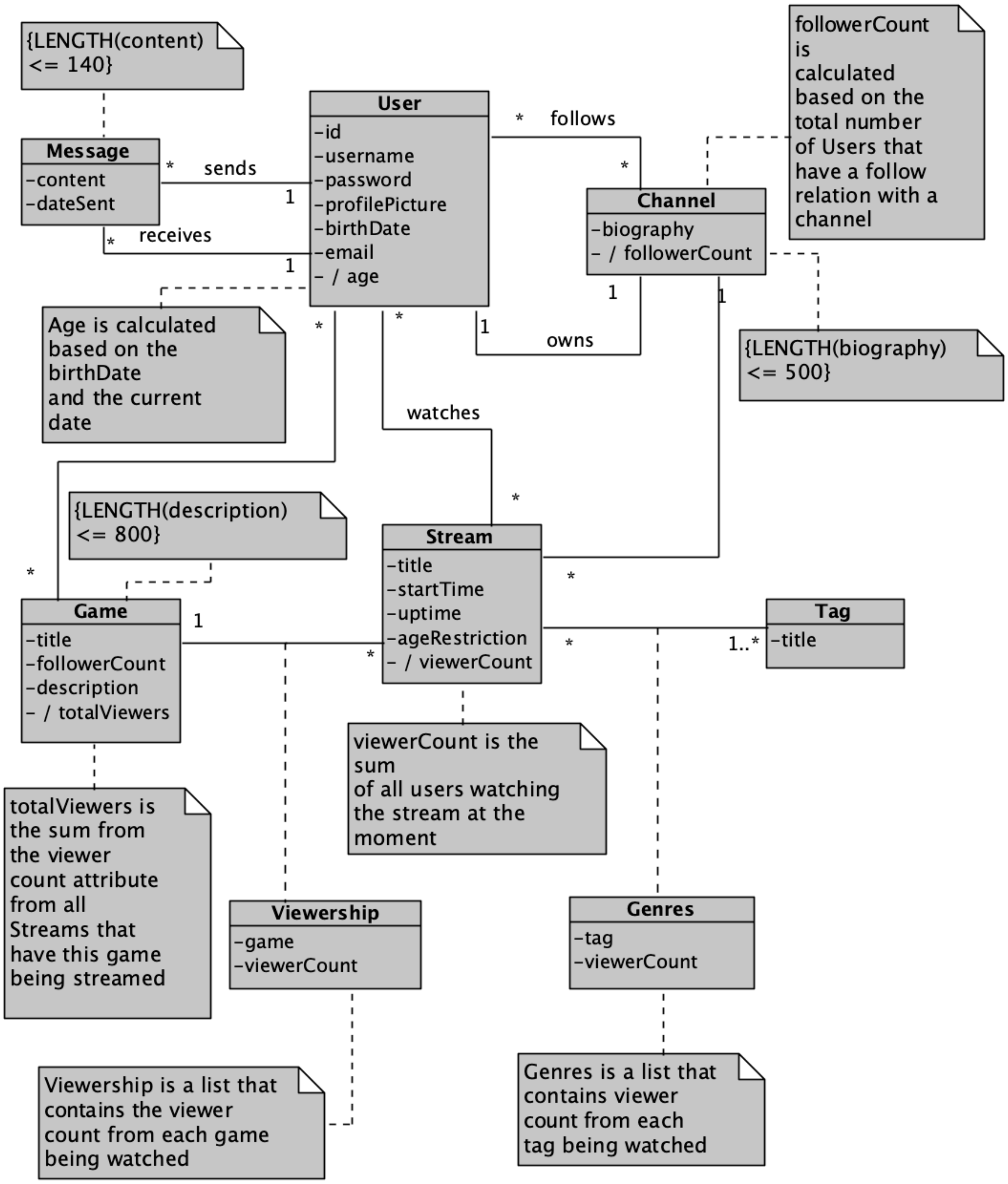
2. addFollower and removeFollower

These triggers control the followerCount field from channel utilising the follows table, either by adding or removing 1 from the count , after an entry is inserted or deleted, respectively, from the table follows.

3. addViewer and removeViewer

These triggers control viewer count on all classes affected directly by it utilising the watches table, it adds or removes 1 from the fields related to this count on the tables: stream, game, viewership and genre, after an entry is added or removed from the watches table.

Class Diagram - UML


{LENGTH(content) <= 140}
Message
-content
-dateSent
sends
receives
User
-id
-username
-password
-profilePicture
-birthDate
-email
- / age
follows
owns
watches
Channel
-biography
- / followerCount
followerCount is calculated based on the total number of Users that have a follow relation with a channel
{LENGTH(biography) <= 500}
Age is calculated based on the birthDate and the current date
{LENGTH(description) <= 800}
Game
-title
-followerCount
-description
- / totalViewers
Stream
-title
-startTime
-uptime
-ageRestriction
- / viewerCount
Tag
-title
1..*
viewerCount is the sum of all users watching the stream at the moment
totalViewers is the sum from the viewer count attribute from all Streams that have this game being streamed
Viewership
-game
-viewerCount
Genres
-tag
-viewerCount
Viewership is a list that contains the viewer count from each game being watched
Genres is a list that contains viewer count from each tag being watched