Faculdade de Engenharia da Universidade do Porto


# Base de dados para um serviço de *streaming* de vídeo ao vivo – 2ª entrega

**Bases de Dados 2023/24 - Licenciatura em Engenharia Informática e Computação**

**Grupo 504 - Turma 2LEIC05**

**Estudantes & Autores:**

Bruno Ricardo Soares Pereira de Sousa Oliveira up202208700@up.pt

Rodrigo Lourenço Ribeiro up202206396@up.pt

João Pedro Martins Mendes up202208783@up.pt

# Índice

# A. Modelo conceptual refinado

Nenhuma sugestão foi feita ao modelo conceptual final da primeira entrega do projeto, pelo que nenhuma alteração foi feita.

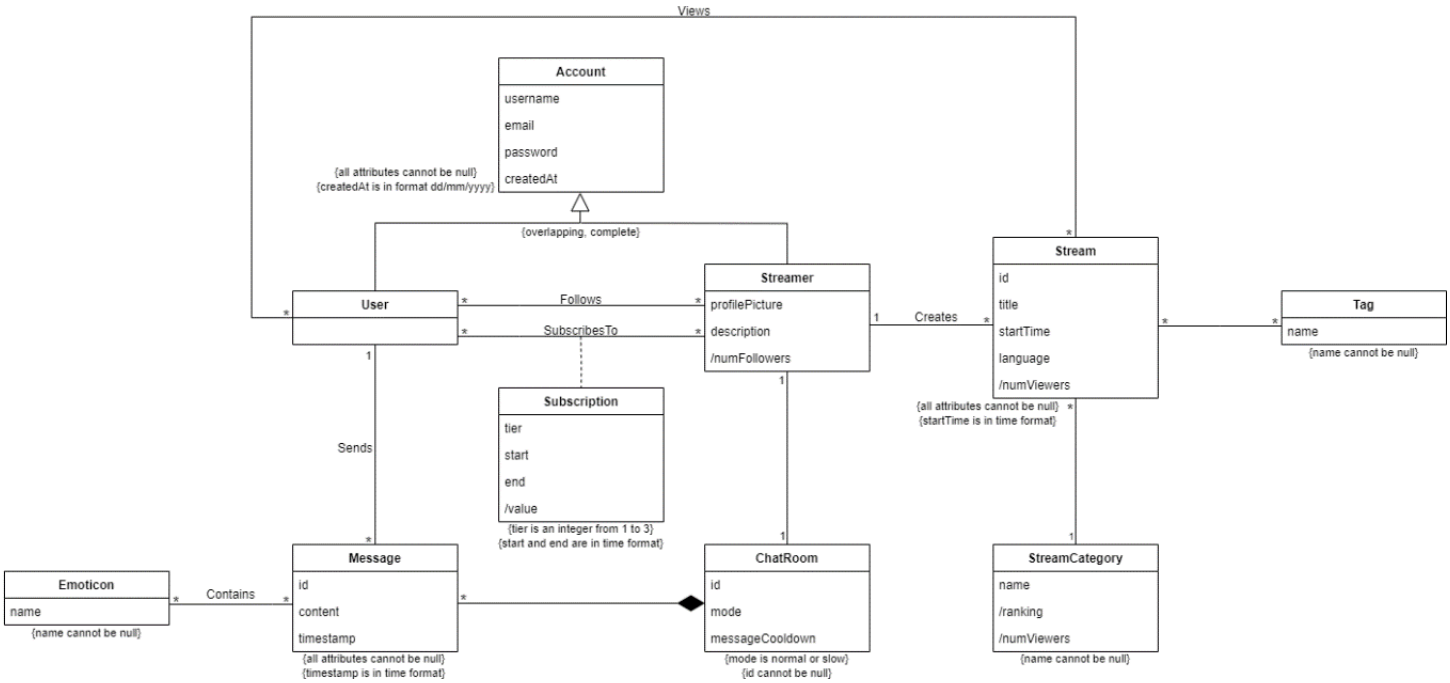


Figura 1 - Modelo conceptual refinado

# B. Definição do modelo relacional

O esquema relacional foi obtido a partir do modelo conceptual, usando as regras de conversão de UML para relações dadas em sala de aula. Nenhuma conversão requer especial atenção, com a exceção da generalização entre as classes `Account`, `User` e `Streamer`.

Para esta generalização, temos três estratégias de conversão possíveis: estilo E/R, orientada a objetos e usando *nulls*. Com o **estilo E/R**, passamos a ter três relações, `Account`, `User` e `Streamer`, similarmente ao modelo conceptual, em que cada uma das relações `User` e `Streamer` tem um novo atributo (`account`), que será chave primária e chave externa para a relação `Account`. Esta estratégia adequa-se ao facto de que a relação é sobreposta, no entanto a relação `User` criada é um pouco redundante, pois só terá o atributo `account` (que é uma chave externa), uma vez que a classe `User` no UML não contém atributos. Consequentemente, interrogações que envolvem a relação `User` (e.g. procurar o usuário com um certo email) ficarão mais complicadas e ineficientes e é gasta mais memória sem grande motivo.

Por sua vez, a estratégia **orientada a objetos**, por causa da generalização ser sobreposta e completa, criará três relações, `User`, `Streamer` e `UserStreamer`. Embora esta alternativa seja mais eficiente em termos de memória, obrigaria a que chaves externas de outras relações que normalmente apontariam para uma das relações das subclasses, por exemplo `Follows.user` que referencia um `User`, também possam referenciar um `UserStreamer`, o que em SQL é inviável.

Por fim, **usando *nulls*** resultaria numa só relação `Account`, com todos os atributos das classes combinados. Esta estratégia, no entanto, não permite manter a informação sobre o tipo de conta a que cada tuplo se associa, impossibilitando garantir que chaves externas para `Account` referenciem uma conta do tipo pretendido, e por ser bastante ineficiente em termos de memória, pois os tuplos relativos a usuários que não são *streamers* teriam todos os atributos da classe `Streamer` (`profilePicture`, `description` e `numFollowers`) com valor `NULL`. Note-se que, num serviço como o deste projeto, o mais frequente é existir muitos mais usuários do que criadores de conteúdo, pelo que a relação teria principalmente tuplos do tipo referido.

Avaliando todas as vantagens e desvantagens, decidimos seguir a abordagem do estilo E/R, pois permite manter informação acerca dos tipos de contas e simplificar a implementação em SQL, à custa de um pouco de redundância.

## 1. Alternativas de conversão da generalização

- **Estilo E/R:**

  Account(<u>username</u>, email, password, createdAt)

  User(<u>account</u>->Account)

  Streamer(<u>account</u>->Account, profilePicture, description, numFollowers)

- **Orientado a objetos:**

  User(<u>username</u>, email, password, createdAt)

  Streamer(<u>username</u>, email, password, createdAt, profilePicture, description, numFollowers)

  UserStreamer(<u>username</u>, email, password, createdAt, profilePicture, description, numFollowers)

- **Usando nulls**

  Account(<u>username</u>, email, password, createdAt, profilePicture, description, numFollowers)

## 2. Modelo relacional (sem integração de ferramenta de IA)

Account (<u>username</u>, email, password, createdAt)
User(<u>account</u>->Account)
Streamer(<u>account</u>->Account, profilePicture, description, numFollowers)
Follows(<u>user</u>->User, <u>streamer</u>->Streamer)
Subscription(<u>user</u>->User, <u>streamer</u>->Streamer, tier, start, end, value)
ChatRoom(<u>id</u>, streamer->Streamer, mode, messageCooldown)
Message(<u>id</u>, user->User, chatRoom->ChatRoom, content, timestamp)
Emoticon(<u>name</u>)
Contains(<u>message</u>->Message, <u>emoticon</u>->Emoticon)
StreamCategory(<u>name</u>, ranking, numViewers)
Stream(<u>id</u>, streamer->Streamer, streamCategory->StreamCategory, title, startTime, language, numViewers)
Views(<u>user</u>->User, <u>stream</u>->Stream)
Tag(<u>name</u>)
HasTag(<u>stream</u>->Stream, <u>tag</u>->Tag)

## 3. Modelo relacional (com integração de ferramenta de IA)

Após o uso da ferramenta generativa de IA, não foram efetuadas nenhumas mudanças. [Ver secção F.1]

Account (username, email, password, createdAt)
User(account->Account)
Streamer(account->Account, profilePicture, description, numFollowers)
Follows(user->User, streamer->Streamer)
Subscription(user->User, streamer->Streamer, tier, start, end, value)
ChatRoom(id, streamer->Streamer, mode, messageCooldown)
Message(id, user->User, chatRoom->ChatRoom, content, timestamp)
Emoticon(name)
Contains(message->Message, emoticon->Emoticon)
StreamCategory(name, ranking, numViewers)
Stream(id, streamer->Streamer, streamCategory->StreamCategory, title, startTime, language, numViewers)
Views(user->User, stream->Stream)
Tag(name)
HasTag(stream->Stream, tag->Tag)

# C. Dependências funcionais e análise das formas normais

Listam-se as seguintes **dependências funcionais** para as relações do modelo anterior:

Account:

```
{username} -> {email, password, createdAt}
{email} -> {username}
```

Streamer:

```
{account} -> {profilePicture, description, numFollowers}
```

Subscription:

```
{user, streamer} -> {tier, start, end}
{tier} -> {value}
```

ChatRoom:

```
{id} -> {streamer, mode, messageCooldown}
{streamer} -> {id}
```

Message:

```
{id} -> {user, chatRoom, content, timestamp}
```

StreamCategory:

```
{name} -> {ranking, numViewers}
```

Stream:

```
{id} -> {streamer, streamCategory, title, startTime, language, numViewers}
```

Analisando as dependências funcionais, podemos verificar que estão quase todas na **Forma Normal de *Boyce-Codd*** e na **Terceira Forma Normal**, uma vez que, para todas as dependências não triviais de cada relação, os atributos do lado esquerdo formam uma superchave da relação. A única exceção é a relação `Subscription`, que admite a dependência `{tier} -> {value}`, onde `{tier}` não é uma superchave da relação nem `value` é um atributo primo, pelo que não está em nenhuma das formas normais.

Ao aplicar o algoritmo da decomposição na Forma Normal de *Boyce-Codd*, obtemos as seguintes relações, que denominámos `Subscription` e `Tier`:

```
Subscription(user->User, streamer->Streamer, tier->Tier, start, end)
Tier(tier, value)
```

Como a relação foi decomposta na Forma Normal de *Boyce-Codd*, e a Forma Normal de *Boyce-Codd* implica a Terceira Forma Normal, as novas relações também estão na Terceira Forma Normal. Assim, conclui-se que o modelo relacional final tem todas as relações em ambas as formas normais.

## 1. Modelo relacional (sem integração da ferramenta de IA)

Account (username, email, password, createdAt)
User(account->Account)
Streamer(account->Account, profilePicture, description, numFollowers)
Follows(user->User, streamer->Streamer)
Tier(tier, value)
Subscription(user->User, streamer->Streamer, tier->Tier, start, end)
ChatRoom(id, streamer->Streamer, mode, messageCooldown)
Message(id, user->User, chatRoom->ChatRoom, content, timestamp)
Emoticon(name)
Contains(message->Message, emoticon->Emoticon)
StreamCategory(name, ranking, numViewers)
Stream(id, streamer->Streamer, streamCategory->StreamCategory, title, startTime, language, numViewers)
Views(user->User, stream->Stream)
Tag(name)
HasTag(stream->Stream, tag->Tag)

## 2. Modelo relacional (com integração de ferramenta de IA)

Após o uso da ferramenta generativa de IA, não foram efetuadas nenhumas mudanças. [Ver secção F.2]

Account (username, email, password, createdAt)
User(account->Account)
Streamer(account->Account, profilePicture, description, numFollowers)
Follows(user->User, streamer->Streamer)
Tier(tier, value)
Subscription(user->User, streamer->Streamer, tier->Tier, start, end)
ChatRoom(id, streamer->Streamer, mode, messageCooldown)
Message(id, user->User, chatRoom->ChatRoom, content, timestamp)
Emoticon(name)
Contains(message->Message, emoticon->Emoticon)
StreamCategory(name, ranking, numViewers)
Stream(id, streamer->Streamer, streamCategory->StreamCategory, title, startTime, language, numViewers)
Views(user->User, stream->Stream)
Tag(name)
HasTag(stream->Stream, tag->Tag)

# D. Criação da base de dados

O ficheiro **create1.sql**, anexado junto a este relatório, contém os comandos em SQL necessários para criar o esquema da base de dados em SQLite. Note-se que foram adicionadas algumas restrições ao ficheiro que não estão explícitas no modelo conceptual, ou porque decorem da forma do esquema relacional (e.g. a restrição `NotFollowingItself` na tabela `Follows`), ou porque se subentendem (e.g. a restrição `StartBeforeEnd` na tabela `Subscription`).

Após o uso da ferramenta generativa de IA, não foram efetuadas mudanças no script de SQL. [Ver secção F.3] Consequentemente, o ficheiro **create2.sql**, que contém os comandos de create1.sql após as melhorias da ferramenta, é idêntico ao inicial.

# E. Carregamento de dados

O ficheiro **populate1.sql** contém comandos em SQL para popular a base de dados em SQLite com uma quantidade significativa de dados. Os tuplos foram principalmente gerados através de dois scripts de Python, que obtiveram grande parte dos dados a partir da API da Twitch, sendo a Twitch a plataforma de *streaming* de vídeo ao vivo que inspirou o tema do projeto. Contudo, alguns dados também foram fabricados, quer através da biblioteca Faker de Python (como palavras-passe e emails), quer usando apenas algoritmos aleatórios (como as subscrições e visualizações), e outros foram obtidos manualmente da Twitch (como as mensagens do chat). Para todos os efeitos, estes dados foram fortemente modificados para se adequarem à base de dados e parecerem o mais realistas possível, mas não devem ser considerados como genuínos.

Após o uso da ferramenta de IA generativa, não se efetuou nenhuma melhoria ao ficheiro anterior. [Ver secção F.4] Assim, o ficheiro **populate2.sql** tem o mesmo conteúdo que o ficheiro populate1.sql.

# F. Integração da ferramenta de IA generativa

Tal como na primeira entrega, a ferramenta que usámos foi o **ChatGPT**, um *chatbot* online de inteligência artificial, que comunica através de diálogos em texto com o utilizador.

## 1. Definição do modelo relacional

Primeiramente, deu-se à ferramenta o seguinte *prompt*:

```
We are doing a project that requires us to transform the conceptual model into a relational schema for a live video streaming service. Our relational model has the following classes:
    Account(username, email, password, createdAt)
    User() (no attributes)
    Streamer(profilePicture, description, numFollowers)
    Subscription(tier, start, end, value)
    Stream(id, title, startTime, language, numViewers)
    StreamCategory(name, ranking, numViewers)
    ChatRoom(id, mode, messageCooldown)
    Message(id, content, timestamp)
    Emoticon(name)
    Tag(name)
User and Streamer are subclasses of Account and there are associations Follows and SubscribesTo between User and Streamer, Views between User and Stream, Contains between Message and Emoticon, Creates between Streamer and Stream, and anonymous ones between Message and ChatRoom, between Streamer and ChatRoom, between Stream and Tag, and between Stream and StreamCategory.
Can you convert the conceptual model into a relational model?
```

De seguida, o ChatGPT converteu o modelo conceptual para um modelo relacional, de forma semelhante ao que nós fizemos. É de notar que o ChatGPT também optou pelo estilo E/R na conversão da generalização. De seguida, deu-se o seguinte *prompt*:

```
We made the following relational schema:
    Account (*username, email, password, createdAt)
    User(*account->Account)
    Streamer(*account->Account, profilePicture, description, numFollowers)
    Follows(*user->User, *streamer->Streamer)
```

```
    Subscription(*user->User, *streamer->Streamer, tier, start, end, value)
    ChatRoom(*id, streamer->Streamer, mode, messageCooldown)
    Message(*id, user->User, chatRoom->ChatRoom, content, timestamp)
    Emoticon(*name)
    Contains(*message->Message, *emoticon->Emoticon)
    StreamCategory(*name, ranking, numViewers)
    Stream(*id, streamer->Streamer, streamCategory->StreamCategory, title,
startTime, language, numViewers)
    Views(*user->User, *stream->Stream)
    Tag(*name)
    HasTag(*stream->Stream, *tag->Tag)
The asterisc (*) marks a primary key.
What changes would you recommend?
```

O ChatGPT apenas sugeriu que adicionássemos um atributo `id` às relações `Follows` e `Subscription` como chave primária alternativa, pois, segundo a ferramenta, chaves primárias compostas podem adicionar complexidade a interrogações que envolvam as duas relações. Não obstante, achamos esta explicação inválida, pois qualquer pesquisa que envolva uma das relações (e.g. que usuários seguem o *streamer* X) jamais usarão diretamente o atributo `id`, determinando o(s) tuplo(s) necessários à interrogação através dos atributos `user` e `streamer`.

## 2. Dependências funcionais e análise das formas normais

O primeiro *prompt* dado foi o seguinte:

```
We have this relational schema:
    Account(*username, email, password, createdAt)
    User(*account->Account)
    Streamer(*account->Account, profilePicture, description, numFollowers)
    Follows(*user->User, *streamer->Streamer)
    Subscription(*user->User, *streamer->Streamer, tier, start, end, value)
    ChatRoom(*id, streamer->Streamer, mode, messageCooldown)
    Message(*id, user->User, chatRoom->ChatRoom, content, timestamp)
    Emoticon(*name)
    Contains(*emoticon->Emoticon, *message->Message)
    StreamCategory(*name, ranking, numViewers)
    Stream(*id, streamer->Streamer, streamCategory->StreamCategory, title,
startTime, language, numViewers)
    Views(*user->User, *stream->Stream)
    Tag(*name)
    HasTag(*stream->Stream, *tag->Tag)
The asterisc (*) marks a primary key.

The relations follow these functional dependencies:
    Account: {username} -> {email, password, createdAt}; {email} -> {username}
    Streamer: {account} -> {profilePicture, description, numFollowers}
    Subscription: {user, streamer} -> {tier, start, end}; {tier} -> {value}
    ChatRoom: {id} -> {streamer, mode, messageCooldown}; {streamer} -> {id}
    Message: {id} -> {user, chatRoom, content, timestamp}
    StreamCategory: {name} -> {ranking, numViewers}
    Stream: {id} -> {streamer, streamCategory, title, startTime, language,
numViewers}

Can you check if each relation is in Boyce-Codd Normal Form or 3rd Normal Form?
```

Então, a ferramenta procedeu à análise das formas normais para cada relação e concluiu que todas as relações estão em ambas as formas normais, o que é falso pois já se verificou anteriormente que a relação `Subscription` não está em nenhuma das formas normais. Então, passámos a dar o seguinte *prompt*:

```
I think you're wrong, the Subscription relation is not in Boyce Codd Normal Form nor 3rd Normal Form. Can you decompose the Subscription relation, so that it doesn't violate Boyce-Codd Normal Form?
```

Com esta informação, o ChatGPT admitiu o erro e decompôs a relação exatamente como nós fizemos. Assim, não foi feita nenhuma alteração ao modelo relacional depois de usar a ferramenta.

## 3. Criação da base de dados

Começámos a conversa com o *prompt* abaixo:

```
I have this relational schema:
    Account (*username, email, password, createdAt)
    User(*account->Account)
    Streamer(*account->Account, profilePicture, description, numFollowers)
    Follows(*user->User, *streamer->Streamer)
    Subscription(*user->User, *streamer->Streamer, tier->Tier, start, end)
    Tier(*tier, value)
    ChatRoom(*id, streamer->Streamer, mode, messageCooldown)
    Message(*id, user->User, chatRoom->ChatRoom, content, timestamp)
    Emoticon(*name)
    Contains(*emoticon->Emoticon, *message->Message)
    StreamCategory(*name, ranking, numViewers)
    Stream(*id, streamer->Streamer, streamCategory->StreamCategory, title,
startTime, language, numViewers)
    Views(*user->User, *stream->Stream)
    Tag(*name)
    HasTag(*stream->Stream, *tag->Tag)
The asterisc (*) marks a primary key.

The tables should have these constraints:
    Profile: all attributes cannot be null, createdAt is in format dd/mm/yyyy
    Subscription: start and end are in time format
    Tier: tier is an integer from 1 to 3
    ChatRoom: mode is normal or slow, id cannot be null
    Stream: all attributes cannot be null, startTime is in time format
    StreamGenre: name cannot be null
    Tag: name cannot be null
    Message: all attributes cannot be null, timestamp is in time format
    Emoticon: name cannot be null

Can you create a script in SQL that creates the tables for this schema in SQLite?
```

Com isto, o ChatGPT escreveu os comandos em SQL para criar cada tabela individualmente. Também implementou a maioria das restrições, porém deixou algumas de

parte e pôs algumas do tipo `NOT NULL` que não especificámos. Além disso, não definiu as ações `ON DELETE` e `ON UPDATE` para as chaves externas e não definiu os comandos `DROP`. Depois, demos-lhe o seguinte *prompt*:

```
Here is the SQL script that we made:
[conteúdo do ficheiro create1.sql]
Do you suggest any improvements?
```

De seguida, o ChatGPT respondeu com algumas sugestões bastante genéricas para a boa criação de tabelas em SQL, sendo que nenhuma delas nos interessou em particular. Assim, não se incluiu nenhuma alteração no script de criação da base de dados.

## 4. Carregamento de dados

Para esta parte, deu-se o *prompt* que se segue:

```
The following script creates a database in SQLite for a live video streaming service:
[conteúdo do ficheiro create2.sql]
Can you create a SQL script to introduce some data into the database?
```

A ferramenta procedeu a criar um pequeno script que insere 2-4 tuplos em cada tabela. Apesar de ser um script simples e com pouca informação e, para alguns casos, genérica, funcionou à primeira com o esquema do create2.sql e estava muito bem construído, com informação bastante coerente para a base de dados num todo. Embora não tenha sido relevante para melhorar o script de carregamento de dados, acreditamos que se exigíssemos mais dados, o ChatGPT seria capaz de criar o ficheiro por si próprio sem muitas alterações.

## 5. Avaliação crítica da ferramenta

As conclusões acerca dos resultados dados pelo ChatGPT são mistas. Por um lado, conseguiu aplicar bem algumas das operações mais algorítmicas e foi surpreendentemente bom na etapa do carregamento de dados. Porém, falhou na etapa da análise das formas normais e não apresentou sugestões muito relevantes quando lhe perguntámos. A nossa avaliação acaba por ser bastante similar à dada na primeira entrega: a ferramenta funciona melhor quando alicerçada ao trabalho de quem a usa, agilizando o processo ao efetuar certas operações que, sem ela, seriam repetitivas e/ou cansativas, mas tudo que ela gera deve ser avaliado criticamente.

# G. Participação dos membros do grupo

O Bruno foi responsável pela totalidade da etapa de carregamento de dados (com a integração da ferramenta de IA), elaborou o relatório e efetuou várias melhorias nas partes dos outros. Em adição, o João fez a análise das formas normais, em conjunto com o Rodrigo, e o script de criação da base de dados. Finalmente, o Rodrigo encarregou-se da conversão do modelo conceptual a relacional, da análise das formas normais com o João e da integração da ferramenta de IA para as três primeiras etapas do projeto.