
**Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores**

# GameOn - F1

48253 : Carlos Guilherme Cordeiro Pereira (A48253@alunos.isel.pt)

48281 : Adolfo Miguel Martins Morgado (A48281@alunos.isel.pt)

48335 : Rodrigo Henriques Correia (A48335@alunos.isel.pt)

Relatório para a Unidade Curricular de Sistemas de Informação
da Licenciatura em Engenharia Informática e de Computadores

Professor : Mestre Walter Jorge Mendes Vieira

**16 – 04 – 2023**

# Resumo

No âmbito da primeira fase do trabalho prático da cadeira, este relatório tem como propósito a elaboração de uma base de dados para o problema proposto da empresa fictícia '*GameOn*'.

Através do trabalho realizado pretendemos demonstrar conhecimento sobre desenho da base de dados, e sua implementação bem como a criação de funcionalidades para a manipulação de dados da mesma através da forma como resolvemos os problemas propostos.

Este relatório parte do pressuposto do acesso por parte do leitor ao código desenvolvido no âmbito do mesmo, não sendo assim necessário enunciá-lo em extensão, sendo somente mencionado trechos do mesmo.

# Abstract

As part of the first phase of the course's work assignment, this report aims to create a database for the proposed problem of the fictitious company '*GameOn*'.

Through the work carried out, we intend to demonstrate knowledge about the design of the database, and its implementation, as well as the creation of functionalities for the manipulation of its data through the way in which we solve the proposed problems.

This report is based on the assumption that the reader has access to the code developed within the scope of the same, therefore it is not necessary to state it in length, just mentioning partly from it.

# Índice

# Lista de Figuras

# Lista de Listagens

# 1

# Modelo Entidade-Associação

## 1.1 Caso em Estudo

O caso de estudo proposto foi da empresa *GameOn* esta empresa pretende desenvolver um sistema para gerir jogos, jogadores e as partidas que estes efetuam, tento ainda uma funcionalida que permite que haja amizade entre jogadores e conversas entre os mesmos.

Os jogadores para além da sua informação pessoal podem ainda ter a sua própria estatistica como por exemplo a pontuação total dos jogos que estes jogaram assim como uma lista dos seus amigos. Podem ainda comprar jogos, e para isso é necessário registar o preço e a data da compra. Os jogos têm um url, uma descricao, um id alfanumerico e o proprio nome, permitindo aos jogadores jogar partidas do jogo em questão. Estes possuem uma estatística que guarda, o número de jogadores que o jogaram, o número de vezes que o mesmo foi jogado e o total de pontos feitos. Os jogadores só podem jogar em partidas da sua região, e as partidas podem ser normais ou multi-jogador. Para as partidas normais, interesa registar a dificuldade. Já para as partidas multi-jogador, interesa registar o estado que pode ser um dos seguintes: 'Por iniciar', 'A aguardar jogadores', 'Em curso' ou 'Terminada'. Terminada a partida, os jogadores têm a possibilidade de ganhar crachás únicos para cada jogo, se atigirem uma certa pontuação. Este crachá tem uma imagem, um nome e um límite de pontos mínimo requerido. Quanto ao sistema de conversas, para cada conversa contém um nome e cada mensagem possui um número de ordem, o texto, a data e o identificador do jogador

que a enviou.

## 1.2 Restrições de Integridade

Após o estudo dos requerimentos do sistema, retiramos as sequintes restrições de integridade:

- **RI1** - atributo com valor único e obrigatório, para o *username* e email do jogador bem como para o nome do jogo;
- **RI2** - O jogador apenas pode tomar um dos seguintes estados ('Ativo', 'Inativo' ou 'Banido');
- **RI3** - O identificador do jogo tem de ser uma sequência alfanumérica de dimensão 10;
- **RI4** - A dificuldade da partida normal tem de ser um valor entre 1 a 5;
- **RI5** - O estado da partida multi-jogador toma os valores: 'Por iniciar', 'A aguardar jogadores', 'Em curso' ou 'Terminada';
- **RI6** - A data de início mais antiga que data de fim para detalhes da partida;
- **RI7** - Data fim tem de ser mais recente que data início e pode ser nulable, quando a partida ainda nao tiver terminado;
- **RI8** - O jogador tem de participar na conversa para enviar mensagens, sendo uma restrição óbvia aos membros do grupo, a discusao da mesma não o é, com isto queremos dizer que é difícil de perceber se esta é uma restrição de integridade do modelo EA, uma regra de negócio ou ambas;
- **RI9** - O jogador tem de pertencer à região onde a partida está a ser jogada, tal como na restrição anterior, o grupo chegou a um consenso de colocar esta restrição tanto em modelo EA como implementar a mesma como regra de negócio.

## 1.3 Mudanças ao EA ao longo do tempo

Numa primeira abordagem, o grupo tinha construído um modelo onde as estatísticas tanto de jogo como de jogador eram fraca de ambas as suas classes mãe, no entanto após algum debate com o professor, este fez-nos perceber que a solução não passava

por esse caminho, mas sim por usar as Superclasses e Subclasses, pois a alternativa que tínhamos era uma associação de 1:1 com obrigatoriedade do lado de estatística, sendo esta fraca da sua classe mãe.

Inicialmente, também tínhamos tanto as entidades crachá como partida, como simples, mas rapidamente nos apercebemos que ambas seriam fracas de jogo pois sem este não pode existir nenhum dos dois.

Posteriormente, tínhamos também na associação entre partida e jogador o atributo pontuação, mas em grupo conseguimos perceber que isso não seria uma solução viável uma vez que existem partidas de multijogador. Apercebemo-nos ainda que a região teria de ter uma associação para partida, algo que não tinha até agora, para que pudéssemos posteriormente implementar a restrição que garante que os jogadores e partida pertencem à mesma região.

Somente mais tarde, é que percebemos ainda que tal como partida e conversa, mensagem e jogador teriam de ter a mesma abordagem.

Chegámos então ao seguinte modelo Entidade-Associação:

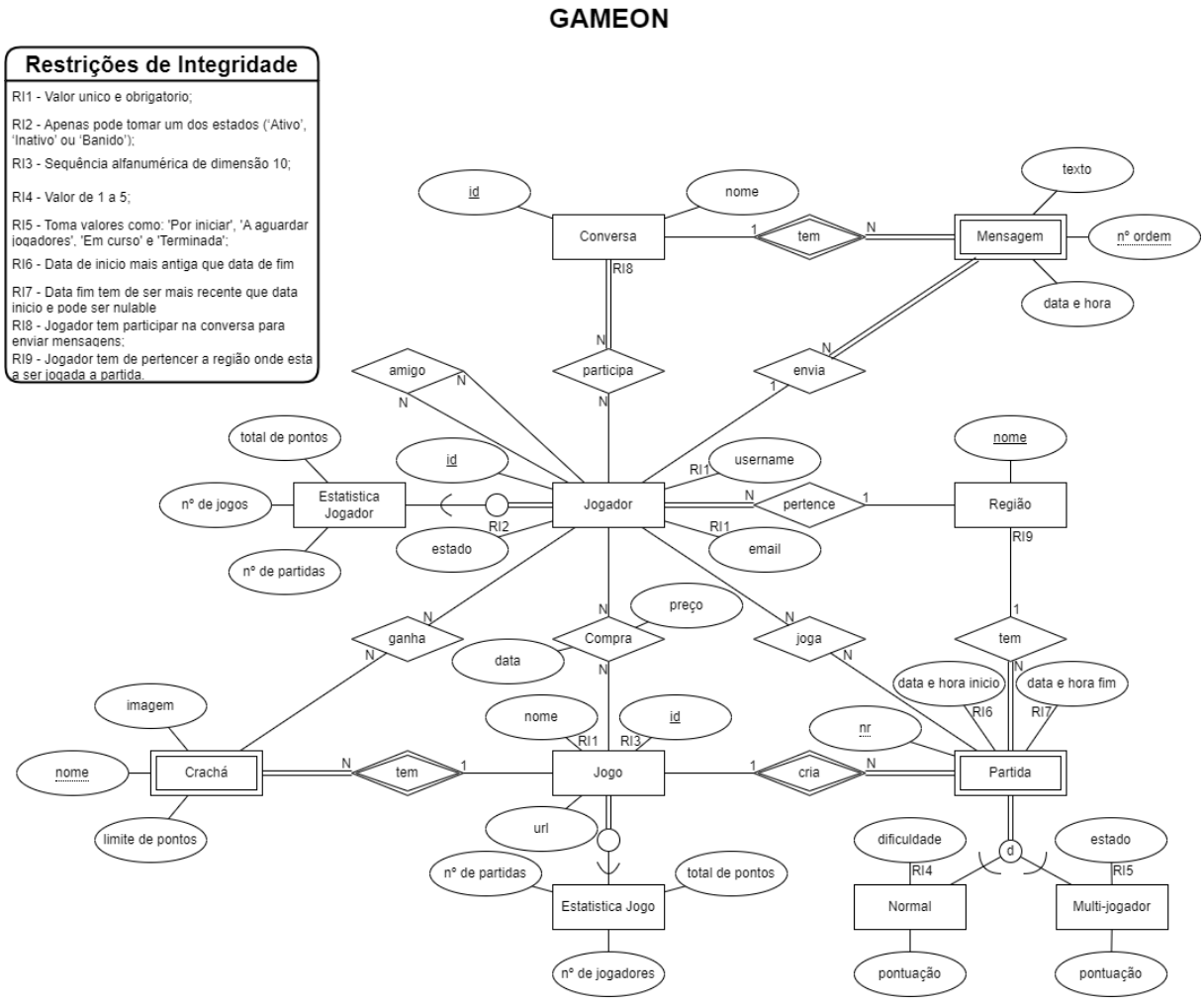


Figura 1.1: Diagrama EA

# 2

# Modelo Relacional

## 2.1 Passagem do modelo EA para Relacional

Foi feita a conversão do modelo EA para o modelo relacional usando as regras aprendidas previamente na cadeira de Introdução a Sistemas de Informação, entre elas o mapeamento das associações N:N para uma nova entidade, que resultou no seguinte modelo:

Região(nome)

PK: Região(nome); FK:

Jogador(id, email, username, status, Região(nome))

PK: Jogador(id); FK: Região(nome)

Jogo(id, url, nome)

PK: Jogo(id); FK:

Conversa(id, nome)

PK: Conversa(id); FK:

Estatística Jogador(nº de jogos, nº de partidas, total de pontos, Jogador(id))

PK: ; FK: Jogador(id)

Estatística Jogo(nº de jogadores, nº de partidas, total de pontos, Jogo(id))

PK: ; FK: Jogo(id)

Partida(nr, data e hora início, data e hora fim, Jogo(id), Região(nome))

PK: Partida(nr); FK: Jogo(id), Região(nome)

Normal(dificuldade, pontuação, Partida(nr))

PK: Partida(nr); FK:

Multijogador(estado, pontuação, Partida(nr))

PK: Partida(nr); FK:

Crachá(nome, imagem, limite de pontos, Jogo(id))

PK: Crachá(nome); FK: Jogo(id)

Mensagem(nº de ordem, texto, data e hora, Conversa(id), Jogador(id))

PK: Mensagem(nº de ordem); FK: Conversa(id), Jogador(id)

Joga (Jogador(id), Partida(nr))

PK: Jogador(id), Partida(nr); FK: Jogador(id), Partida(nr)

Compra(data, preço, Jogador(id), Jogo(id))

PK: Jogador(id), Jogo(id); FK: Jogador(id), Jogo(id)

Ganha(Crachá(nome), Jogador(id))

PK: Crachá(nome), Jogador(id); FK: Crachá(nome), Jogador(id)

Participa(Jogador(id), Conversa(id))

PK: Jogador(id), Conversa(id); FK: Jogador(id), Conversa(id)

Amigo(Jogador(id), Jogador(id))

PK: Jogador(id), Jogador(id); FK: Jogador(id), Jogador(id)

# 3

# Modelo Físico

O modelo físico foi implementado com a linguagem PostgreSQL, com uso à extensão SQL para programação, a linguagem plpgSQL, que permite que se escrevam procedimentos armazenados, funções e triggers, que são um tipo especial de procedimento armazenado que é automaticamente executado pelo banco de dados quando um determinado evento ocorre, como a inserção, atualização ou exclusão de dados em uma tabela. Esta extensão permite que se executem tarefas mais complexas do que as que podem ser realizadas com simples consultas SQL.

## 3.1 Domínios

Para a implementação da nossa solução sentimos a necessidade de criamos domínios, para criar um tipo de variável mais apropriada para certos campos, como para o URL, EMAIL e ALFANUMERIC.

```
1 CREATE DOMAIN ALPHANUMERIC AS VARCHAR(10) CHECK (VALUE ~* '^[A-
    Z0-9]+$');
```

Listagem 3.1: Domínio *ALPHANUMERIC*

## 3.2 Tabelas

A tabela jogo, embora não tenha muitas restrições (*Constrains*) implementa dois dos três tipos criados para a implementação da base de dados.

```
1  -- Jogo
2  CREATE TABLE IF NOT EXISTS jogo(
3      id           ALPHANUMERIC,
4      nome         VARCHAR(50) NOT NULL,
5      url          URL NOT NULL,
6
7      UNIQUE (nome),
8
9      CONSTRAINT pk_jogo PRIMARY KEY (id)
10 );
```

Listagem 3.2: Tabela jogo

## 3.3 Regras de Negócio

As regras de negócio foram garantidas através da execução de triggers, executados pelo banco de dados. A rotina ***checkJogadorPartidaRegiao*** é uma das duas rotinas criadas, que cobre as restrições de integridade que não conseguem ser implementadas. Esta verifica apenas se o jogador pode jogar uma partida, isto é apenas aceita o jogador na partida se este pertencer à mesma região que a partida está a ser jogada.

```
1  CREATE FUNCTION checkJogadorPartidaRegiao()
2      RETURNS TRIGGER LANGUAGE plpgsql
3  AS
4  $$
5      DECLARE
6          regiao_nome VARCHAR(50);
7      BEGIN
8          SELECT partida.nome_regiao INTO regiao_nome FROM partida
       WHERE partida.nr == NEW.nr_partida;
9          IF (regiao_nome != (SELECT jogador.nome_regiao FROM
      jogador WHERE jogador.id == NEW.id_jogador)) THEN
10             RAISE EXCEPTION 'O jogador nao pertence a regiao da
      partida';
```

```
11          END IF;
12      END;
13  $$;
14
15  CREATE TRIGGER checkJogadorPartidaRegiao BEFORE INSERT ON joga
16      FOR EACH ROW
17      EXECUTE FUNCTION checkJogadorPartidaRegiao();
```

Listagem 3.3: Rotina checkJogadorPartidaRegiao

## 3.4 Funcionalidades

Esta secção destina-se apenas às funcionalidades requeridas diretamente no projeto, onde apenas se irá enunciar um exemplo de cada tipo, para efeitos demonstrativos.

### 3.4.1 Funções

Um dos requerimentos foi a função **PontosJogoPorJogador**, que tem como objetivo retornar uma tabela com duas colunas (identificador de jogador, total de pontos) em que cada linha contém o identificador de um jogador e o total de pontos que esse jogador teve nesse jogo.

```
1  CREATE FUNCTION PontosJogoPorJogador(jogo_id ALPHANUMERIC)
2      RETURNS TABLE (jogador_id INT, total_pontos INT) LANGUAGE
       plpgsql
3  AS
4  $$
5      BEGIN
6          RETURN QUERY SELECT joga.id_jogador, totalPontosJogador(
       joga.id_jogador) FROM joga WHERE joga.id_jogador IN (
7          SELECT joga.id_jogador FROM joga WHERE joga.nr_partida
       IN (
8          SELECT partida.nr FROM partida WHERE partida.id_jogo ==
       jogo_id));
9      END;
10 $$;
```

Listagem 3.4: Função PontosJogoPorJogador

### 3.4.2 Procedimentos

O seguinte procedimento armazenado, tem como objetivo mudar o estado do jogador, no entanto para criar algum tipo de controlo de verificação achámos que seria interessante avisar o utilizador, caso o id fornecido não for o id de um utilizador conhecido. Acrescentamos também, a garantia que o estado atual é diferente do estado futuro, caso isso se confirme não realizamos a alteração e notificamos o utilizador do sucedido.

```
1  CREATE PROCEDURE updateEstadoJogador(id_jogador INT, new_estado
       VARCHAR(10))
2       LANGUAGE plpgsql
3  AS
4  $$
5      BEGIN
6          -- Checks
7          IF (id_jogador NOT IN (SELECT jogador.id FROM jogador))
       THEN
8              RAISE NOTICE 'jogador not found';
9          END IF ;
10         IF ((SELECT jogador.estado FROM jogador WHERE jogador.id
        == id_jogador) == new_estado) THEN
11             RAISE NOTICE 'jogador already has this estado';
12         END IF ;
13         -- expected
14         UPDATE jogador SET estado = new_estado WHERE jogador.id
       = id_jogador;
15     END ;
16 $$;
```

Listagem 3.5: Procedimento updateEstadoJogador

### 3.4.3 Gatilhos

Este gatilho foi criado dada a necessidade da instrução *DELETE* sobre a vista jogadorTotalInfo permita colocar os jogadores envolvidos no estado "Banido".

```
1  CREATE FUNCTION banirJogador()
2      RETURNS trigger LANGUAGE plpgsql
3  AS
```

```
4  $$
5      DECLARE
6          jogador_id INT;
7      BEGIN
8          SELECT jogador.id INTO jogador_id FROM jogador WHERE
       jogador.username == OLD.username;
9          UPDATE jogador SET estado = 'Banido' WHERE jogador.id ==
        jogador_id;
10     END;
11 $$;
12
13 CREATE TRIGGER banirJogador INSTEAD OF DELETE ON
       jogadorTotalInfo
14     FOR EACH ROW
15     EXECUTE FUNCTION banirJogador();
```

Listagem 3.6: Gatilho banirJogador

### 3.4.4 Vistas

Esta vista, foi criada para permitir aceder à informação do jogador, como o identificador, estado, email, *username*, número total de jogos em participou, número total de partidas e número total de pontos que já obteve de todos os jogadores cujo estado seja diferente de "Banido". Não foi usada a tabela de estatísticas, pois foi proibído o seu uso para esta funcionalidade.

```
1  CREATE VIEW jogadorTotalInfo AS
2      SELECT jogador.id, jogador.estado, jogador.email, jogador.
       username, totalJogosJogador(jogador.id) AS total_jogos,
3      totalPartidasJogador(jogador.id) AS total_partidas,
       totalPontosJogador(jogador.id) AS total_pontos
4      FROM jogador WHERE jogador.estado != 'Banido';
```

Listagem 3.7: Vista jogadorTotalInfo

## 3.5 Detalhes

# 4

# Validação

## 4.1 Introdução

De forma a validar o bom funcionamento de todas as funcionalidades do nosso trabalho, recoremos a testes não intrusivos, pelo facto de não ser boa política danificar o conteúdo de uma base de dados, mesmo que para este caso como não existe um consumidor direto, não haveria qualquer problema em fazer testes intrusivos, no entando é uma boa prática de programação que decidimos seguir. No entanto, alguns dos testes não estão a funcionar. Embora não esteja tudo testado, o grupo conseguiu perceber o proposito dos mesmos e demonstrar as suas capacidades na boa realização de testes.

## 4.2 Controlo transacional

O controlo transacional neste trabalho foi feito apenas com recurso a diferentes nível de isolamento do *Read Commited (default)*, com isto apenas algumas das funcionalidades é que têm um nível diferente do mesmo, com *Repeatable Read*.

```
1 CREATE OR REPLACE PROCEDURE test_i()
2     LANGUAGE plpgsql
3 AS
4 $$
5     DECLARE
```

```
6           region_name1 varchar := 'TestRegion61';
7           player_name1 varchar := 'TestPlayer61';
8           player_email1 varchar := 'testplayer61@gmail.com';
9           player_id1 INT;
10          chat_name varchar := 'TestChat61';
11          chat_id INT :=0;
12      BEGIN
13          insert into regiao(nome) values(region_name1);
14          CALL create_jogador(region_name1, player_name1,
    player_email1);
15          SELECT id INTO player_id1 FROM jogador WHERE email =
    player_email1;
16          RAISE NOTICE 'Exercise 2i';
17          RAISE NOTICE 'Test data created';
18          RAISE NOTICE 'Testing the creation of a chat';
19      BEGIN
20          CALL iniciarconversa(player_id1, chat_name, chat_id);
21              IF (SELECT COUNT(*) FROM conversa WHERE id = chat_id
    ) = 1 THEN
22                  RAISE NOTICE 'Test 1 succeeded';
23              ELSE RAISE EXCEPTION 'Test 1 failed'; END IF;
24      END;
25          DELETE FROM mensagem WHERE id_conversa = chat_id;
26          DELETE FROM participa WHERE id_conversa = chat_id;
27          DELETE FROM conversa WHERE id = chat_id;
28          DELETE FROM jogador WHERE id = player_id1;
29          DELETE FROM regiao WHERE nome = region_name1;
30      END;
31  $$;
32
33  CALL test_i();
```

Listagem 4.1: Teste de criar um *chat*

# 5

# Conclusão

Ao longo deste projeto, desenvolvemos um sistema de gerenciamento de jogos online utilizando um banco de dados relacional. Através da implementação do modelo Entidade-Associação, Modelo Relacional e Modelo Físico, garantimos a integridade dos dados. O projeto também incluí a criação de funções, procedimentos armazenados, gatilhos e vistas para melhorar a funcionalidade e a flexibilidade do sistema. As alterações feitas no modelo Entidade-Associação ao longo do tempo demonstram a importância da adaptação e evolução do projeto conforme as necessidades que surgem. O uso da linguagem PostgreSQL e da extensão plpgSQL permitiu a implementação de um sistema robusto e eficiente para gerenciar informações. Contudo ainda não foi possível, devido à capacidade do grupo, definir níveis de isolamento para as funcionalidades acima ilustradas.

Em suma, apesar de algumas adversidades e imperfeições o grupo conseguiu desenvolver os modelos de uma base dados e implementá-la com sucesso.