Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Sistema de Gestão de Jogos - F2

47206 : Tiago Alexandre Figueiredo Pardal (47206@alunos.isel.pt)

47202 : Manuel Maria Penha Gonçalves Cavaco Henriques (47202@alunos.isel.pt)

47198 : Ricardo Filipe Matos Rocha (47198@alunos.isel.pt)

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

Professor : Doutor Nuno Leite

12 – 06 – 2023

Resumo

No âmbito da segunda fase do trabalho prático da cadeira, este relatório tem como propósito explicitar as decisões tomadas no desenho da aplicação de acesso à base de dados desenvolvida na primeira fase do mesmo.

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, bastando apenas mencionar trechos do mesmo.

Abstract

In the scope of this assignment's second phase, this report has the purpose of exposing the design philosophy of the application, specifically the one developed in the first phase of this assignment.

This report parts with the assumption that the reader has access to the code developed, not being necessary to describe it to its full extent, only mentioning a few parts of it.

Índice

Lista de Figuras

Lista de Código

1

Introdução

1.1 Camada Acesso a Dados

A aplicação foi desenvolvida de forma a tentar modular ao máximo o acesso aos dados e as escritas sobre a base de dados.

Sendo carregado os dados do modelo da base de dados pelo *Entity Manager*. Esta class pertencente à biblioteca do **JPA**, sendo inicializada através do *Entity Manager Factory*, vai conter os métodos necessários para iniciar as transações à base de dados e permitir interagir com a mesma.

As escritas são apenas efetuadas após a obtenção dos valores a inserir, atualizar ou até apagar, e os termos verificados quanto ao seu tipo e formato de modo a garantir que não se tentam efetuar escritas com valores errados, evitando assim acessos à base de dados desnecessários.

1.2 Jakarta Persistence

Toda a aplicação foi desenvolvida com base na *Java Persistence API*.

O **JPA** é responsável pelo acesso a dados, o mapeamento dos mesmos através do *Object-Relational Mapping* (**ORM**) e as escritas dos mesmos na base de dados.

1.2.1 ORM Mapeamento Entidades

A camada **ORM** é estabelecida através do *EclipseLink*, tendo sido criadas interfaces e classes para cada entidade da base de dados, com uso das anotações respetivas, responsáveis por dar a informação relevante ao *EclipseLink*, de forma a que este mapeie cada tabela e as suas respetivas colunas para um objecto.

1.2.2 Repositórios JPA

Os repositórios JPA são classes responsáveis por realizar o acesso à camada de dados e permitem fazer consultas sobe as informações guardadas na base de dados.

1.2.3 Mappers JPA

Mappers no contexto do JPA servem o propósito de realizar operações **CRUD**.

1.2.4 Abordagem Optimista versus Pessimista

Estas abordagens referem se ao sistema de *locking*.

A abordagem otimista tal como o próprio nome indica, parte do princípio otimista de que os dados sobre o quais vai efetuar uma transação não serão alterados, seguindo a seguinte ordem de operações:

Leitura -> Transação -> *Update* -> Procurar conflito -> Verificação conflito

Se não existiu conflito efetua *commit* da transação, se existiu conflito cancela transação e efetua *rollback*.

Por conflito entenda se uma inconsistência entre os dados observados no início e os presentes na base de dados.

Algoritmos otimistas fazem por vezes várias tentativas até conseguirem efetuar a transação com sucesso.

Já a abordagem pessimista, parte do princípio de que vão existir acessos concorrentes aos dados sobre os quais está a tentar efetuar a transação.

Assim sendo a sua solução é reservar os recursos, e só os libertar no final da operação.

Lock -> Leitura -> Transação -> *Commit* -> Libertação *Locks*

2

Problema

Relembramos aqui o caso em estudo além de mencionar alterações feitas ao modelo.

Estas alterações deveram se apenas ao facto de nos termos apercebido no início da implementação da aplicação em java que algumas das decisões tomadas não foram as mais corretas.

Sendo assim as adendas efetuadas à estrutura da base de dados encontram se aqui explicadas.

2.1 Caso em Estudo

No presente trabalho é nos proposto o desenvolvimento de um sistema de jogos e jogadores onde seja possível guardar os jogos adquiridos por cada jogador, as suas partidas individuais ou com outros jogadores sobre estes mesmos jogos e as conversas com outros jogadores, além de um mecanismo de amizade para ligar contas para mais fácil interação.

2.2 Adendas sobre Base de dados implementada na primeira fase

Por mero lapso tínhamos um script de remoção dos conteúdos da base de dados inconsistente com o da criação do mesmo.

Isto deveu se a uma implementação tardia das estatísticas sobre vistas em vez de entidades que não se refletiu numa atualização do script *DropDatabase.sql*.

Tomamos também a decisão errada de não separar pontos de partida_multi_jogador, indo assim contra a normalização do modelo.

Corrigimos esta situação criando uma nova entidade jogador_partida_multi_jogador que retém agora a informação dos pontos dos jogadores em partidas multijogador.

Esta alteração levou à necessidade de várias alterações de procedimentos que envolvessem pontos de partidas multijogador.

Depois de alguma conversa com o professor, percebeu-se que as funções **PLpgSQL** não necessitam de ter um nível de isolamento, sendo atómicas. Assim, retirou-se os procedimentos que procuravam fazer *SET* ao nível de isolamento nas funções das alíneas e), f) e g), não sendo necessário de todo.

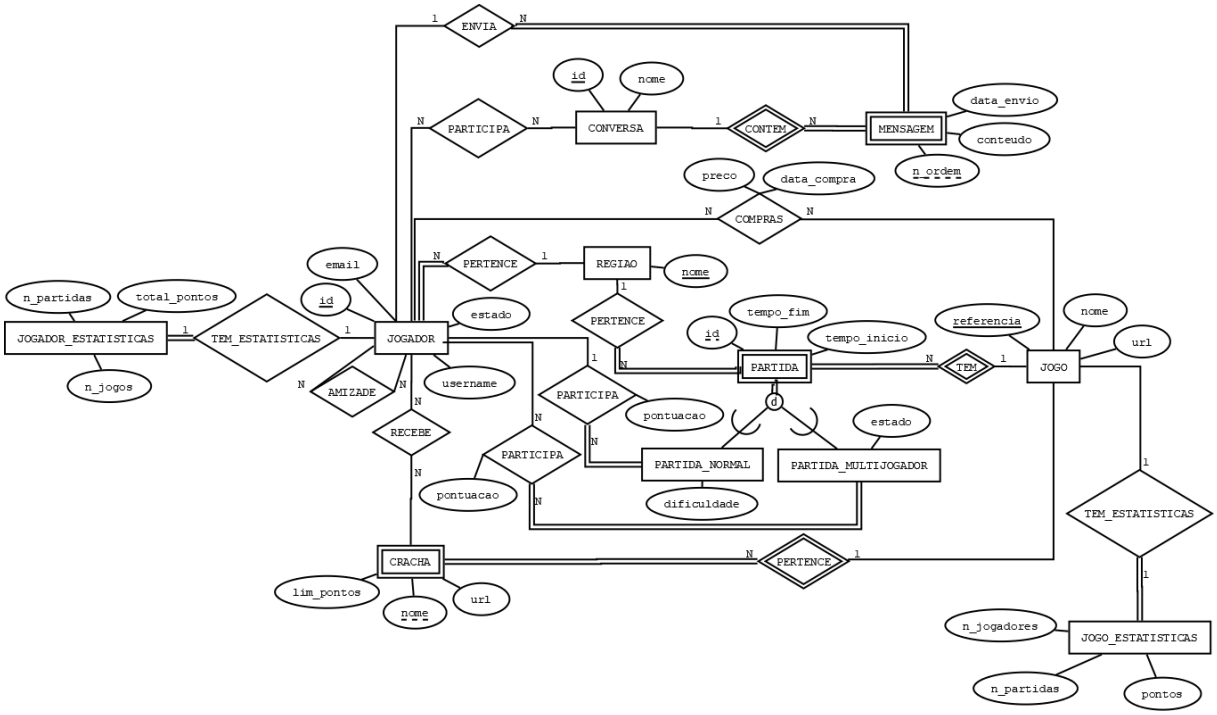

Figura 2.1: Diagrama EA

3

Organização modelo JPA

3.1 Organização geral

Na nossa implementação fizemos uso de *Repositories* e *Mappers*, usando interfaces para as ligar com o *JPAContext*. Para o gerenciamento centralizado das transações, utilizou-se a classe abstrata dada em aula, a classe *DataScope*. Um objeto é criado a partir de uma implementação desta classe criada por nós e é a partir daqui que todas as operações presentes nos *Mappers*, *Repositories* e *JPAContext* ocorrem, recorrendo ao *EntityManager* gerido por este.

Para o acesso à base de dados, utiliza-se uma variável de ambiente.

Esta variável de ambiente, **PERSISTENCE_NAME**, tem como objectivo definir definir o nome do *Persistence*, isto devido a termos elaborado o trabalho em diferentes máquinas e com bases de dados locais e não locais, ou seja tendo configurações de *Persistence* distintas entre máquinas.

3.2 *Repositories*

Os *Repositories*, através das interfaces para cada entidade, estendem a interface *IRepository*.

```
1  package si.isel.t43dg01.data_repositories.interfaces;
2
3  import java.util.List;
4
5  public interface IRepository<T,TCol,TK> {
6      T findByKey(TK key);
7      TCol find(String jpql, Object... params);
8      List<T> findAll();
9
10 }
```

Listagem 3.1: *IRepository*

As funções *findByKey* e *findAll* utilizam *NamedQueries* presentes nas classes das entidades a que acedem, por via da anotação *@NamedQuery*. Isto resolve a dificuldade de aceder a entidades a partir do *JPA* cuja chave é gerada pelo sistema.

Quando o **ID** é gerado pelo sistema, à partida não é fácil o acesso ao objeto a partir do mesmo, visto em diversos casos sabermos alguns ou até todos os parâmetros do mesmo, com exceção do próprio **ID**, devido ao carater altamente imprevisível do mesmo, principalmente devido a situações de concorrência em que os dados não persistiram, mas o valor foi incrementado.

3.3 *Mappers*

Os *Mappers*, à semelhança dos *Repositories*, utilizam interfaces para cada entidade, que extendem a interface *IDataMapper*.

```
1 package si.isel.t43dg01.data_mappers.interfaces;
2
3 public interface IDataMapper<T, TK> {
4     TK create(T entity);
5     T read(TK id);
6     TK update(T entity);
7     TK delete(T entity);
8 }
```

Listagem 3.2: *IDataMapper*

Os *Mappers* limitam-se a implementar as funções **CRUD**, explicando a simplicidade do seu retorno e parâmetros.

4

Detalhes de Implementação

4.1 Notas Prévias

Assim como é evidente nos outros capítulos deste relatório, uma vez mais, são aqui apenas referidos detalhes de implementação que achámos relevantes para a compreensão de como este trabalho foi efetuado e a razão das decisões tomadas no mesmo.

Não sendo assim mencionado aqui alíneas inteiras pedidas no relatório, por acharmos que a lógica das mesmas é trivial ou semelhante à de outras alíneas anteriores ou posteriores já explicadas.

4.2 Associações com Chaves compostas em JPA

Este trabalho notou-se particularmente complexo, principalmente devido aos mapeamentos em JPA entre entidades que possuem chaves compostas.

Para efetuarmos estes mapeamentos foi necessário a criação do objeto pk para cada entidade que tem uma chave composta.

Mapeando no objeto pk os atributos presentes na mesma, e efetuando a ligação destes com as relações correspondentes, caso sejam chaves estrangeiras, caso contrário estarão apenas presentes no objeto pk.

Utilizamos a anotação @embeddable em cada classe pk de forma a identificar como uma chave composta, e usando a anotação @EmbeddedId na entidade que faz uso

desta chave composta para identificar a sua chave, substituindo o uso da anotação @Id nestas classes.

4.3 Associar Crachá

Para a implementação de associação de crachá foi nos pedida 3 implementações distintas:

1. Com o procedimento armazenado
2. Sem procedimentos armazenados nem qualquer função **PLpgSQL**
3. Sem o procedimento principal, mas utilizando os procedimentos usados pelo mesmo

4.3.1 Sem procedimentos armazenados

O create, update e delete foram implementados manualmente, sem uso a *procedures*, embora estes tenham sido desenvolvidos na primeira fase do trabalho.

Assim, fazemos uso do *Mapper* de JogadorCracha para associar o crachá a jogador caso o mesmo tenha alcançado pontos suficientes para o obter.

Esta verificação efetua se através do uso de *queries* simples e da comparação dos resultados obtidos pelas mesmas.

4.3.2 Sem o procedimento armazenado principal

A implementação é similar à implementação sem procedimentos, no entanto, esta implementação aproveita-se de uma função **PLpgSQL** presente no uso do procedimento, a função *pontosJogoPorJogador*. Esta função é utilizada para obter o total de pontos do jogador a que se quer associar um crachá.

4.3.3 Com procedimento armazenado

A chamada ao procedimento é feita recorrendo ao uso da função *createNativeQuery*, em que se passa a *string*, *CALL associarCrachaLogica(?1, ?2, ?3)*. Devido à necessidade do uso da definição "*CALL*", é necessário recorrer a uma *Native Query*.

4.4 Teste à Abordagem Otimista

Tal como é pedido no enunciado realizamos um teste concorrente para a abordagem otimista que demonstra a falha do mesmo num cenário de acessos concorrentes aos mesmos dados. No nosso cenário de teste este acesso concorrente foi efetuado por meio de 2 *threads* a tentarem efetuar a operação de aumento de pontos com implementação otimística.

No nosso teste, por forma a repormos o estado original da base de dados, são eliminados os objetos criados para a realização deste teste. Optámos por apanhar as exceções e imprimir apenas a mensagem das mesmas.

O teste resulta assim na seguinte mensagem de erro:

```
jakarta.persistence.OptimisticLockException: Exception [EclipseLink-5006]
(Eclipse Persistence Services - 4.0.1.v202302241130):
org.eclipse.persistence.exceptions.OptimisticLockException
Exception Description: The object [Cracha[nome = test cracha, jogo = 'zzzzzxxxx1',
lim_pontos = 1200, url = 'www.imgs/test_cracha']] cannot be updated because it
has changed or been deleted since it was last read.
Class> si.isel.t43dg01.orm.Cracha Primary Key> [[test cracha, zzzzzxxxx1]:
27073120]
```

Comprovando assim que estamos a usar corretamente uma abordagem otimista.

Foi também implementado um teste pessimista que simula exatamente a mesma situação concorrente e que devido à natureza da abordagem pessimista não resulta em qualquer erro. Provamos assim que temos ambas as abordagens implementadas.

Infelizmente por um erro que não conseguimos compreender, não nos é possível correr com sucesso ambos os testes em simultâneo, no entanto se corridos independentemente funcionam e é evidente que o comportamento obtido é o esperado, tal como já foi explicado.

Supomos que o motivo de falha derive da forma como efetuamos o teste de forma concorrente.