ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Trabalho Fase 2 – Projeto "GameOn"

Vasco Branco – 48259
João Pereira – 48264
Tiago Neves – 48292

Orientadores: Walter Vieira

Relatório final realizado no âmbito de Sistemas de Informação,
do curso de licenciatura em Engenharia Informática e de Computadores
Semestre de Verão 2022/2023

Junho de 2023

Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores

Sistema de gestão de videojogos da empresa Game*On*

Vasco Branco – 48259
João Pereira – 48264
Tiago Neves – 48292

Orientador: Walter Vieira

Relatório final realizado no âmbito da disciplina Sistema de Informação,
do curso de licenciatura em Engenharia Informática e de Computadores
Semestre de Verão 2022/2023

Junho de 2023

Resumo

Nesta segunda fase do trabalho criámos uma camada de acesso aos dados correspondentes ao sistema de base de dados desenvolvido na primeira fase deste trabalho. Agora unimos o utilizador à base de dados através da camada "*Presentation*" (apresentação), sendo esta a única camada que o utilizador tem acesso. As informações inseridas pelo mesmo, as funções de *input*, *output* e de *print* de listas implementadas são tratadas em "*BusinessLogic*" (negócio), utilizando os modelos definidos em "*model*" (dados).

Palavras-chave: Amigo; Comprar; Conversas; Crachás; Estatísticas (jogadores e jogo); Jogadores; Jogos (multijogador e normal); Mensagens; Participar; Região; Tem.

Agradecimentos

Queremos agradecer ao professor Walter Vieira que além de realizar as aulas programadas, sempre se mostrou disponível para a esclarecer qualquer tipo de dúvida e disponibilizou slides e ficheiros de forma a ajudar na compreensão das matérias lecionadas e consequentemente no trabalho.

Índice

1. Introdução

1.1 Introdução à camada de acesso a dados

Como pretendido no enunciado desta fase do trabalho realizámos uma camada de acesso a dados que utiliza uma implementação JPA. Além disto, desenvolvemos também padrões de desenho. Estes padrões podem ser consultados na pasta *java* presente no projeto de trabalho e com eles fomos capazes de mapear e realizar operações de forma correta o modelo do nosso projeto em *java*.

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.

1.3 ORM Mapeamento de entidades

Nesta fase do trabalho foi requisitado que realizássemos procedimentos que trabalhassem com *Optimistic* e *Pessimistic Locking,* para tal recorremos de expressões e anotações que providenciassem este tipo de controlo de concorrência.

2. Formulação do Problema

2.1 Caso Em Estudo

O trabalho proposto consiste no desenvolvimento de um sistema de criação, alteração e gestão de videojogos. Também, tendo em conta toda a informação que diz respeito aos videojogos: jogadores, crachás, chat's entre outros. Assim, possibilitando uma camada de acesso a dados para que o utilizador consiga interagir com a base de dados.

2.2 Complementos à primeira fase

A única alteração das soluções implementadas na primeira fase foi no procedimento "associarcracha" que primeiramento não utilizava qualquer função ou procedimento previamente implementado mas para a resolução correta da alínea 1 c) da segunda fase prática, foi necessário substituir parte do seu código pela função "pontosjogoporjogador". Assim foi possível reutilizar a função chamada dentro do procedimento "associarcracha".

3. Organização geral

3.1 Estruturação da solução

Primeiramente desenvolveu-se o mapeamento das entidades em si e o mapeamento das suas relações, esta implementação localiza-se no package model tendo um ficheiro correspondente para cada tabela e se essa mesma tabela possuir uma chave composta foi criado um ficheiro adicional com o sufixo de “Id” com o objectivo de representar essa mesma chave.

Consequentemente, foram implementados os padrões de desenho Mapper e Unit Of Work, tendo um contrato definido em cada uma das suas interfaces “IUnitOfWork” e “IDataMapper” que são implementadas no package “DataAcess” nos seus ficheiros respetivos. A implementação do “IUnitOfWork” é igual para todas as entidades, no entanto existem diversas implementações do “IDataMapper” uma vez que varia de entidade para entidade, ambos serão descritos com maior detalhe nos próximos subcapítulos.

Posteriormente, foram implementadas todas funções de acesso a procedures e queries já definidas na base de dados e também os tratamentos de concorrência pessimista e optimista, tudo isto no package de “accessFunctionalities” dentro do ficheiro “accessFunctionality”, isto encontra-se dentro do package “businessLogic”.

Mais tarde, estas mesmas implementações foram alteradas para utilizar o “DataScope” que se encontra definido dentro do “DataScopes” com a classe abstracta “AbstractDataScope” que é extendida pela classe "DataScope”, basicamente “DataScope” trata-se de uma classe que cria e gerencia uma sessão de banco de dados usando o JPA. Ela garante que as transações sejam tratadas corretamente e fornece métodos para validar ou cancelar o trabalho realizado na sessão.

3.2 Mappers

Os Mappers utilizam interfaces que extendem a interface “IDataMapper”, esta implementa as seguintes funções:

Tid Create(T entity)

T Read(Tid id)

Tid update(T entity)

Tid delete(T entity)

Resumidamente, os Mappers implementam as funções CRUD, tendo como base um contrato já definido que necessita de ser implementado.

3.3 Unit Of Work

O Unit Of Work à semelhança dos Mappers também é uma interface, sendo que implementa as seguintes **funções**:

void beginTransaction()
void commit()
void rollback()
void flush()

void connect()

void close()

Resumidamente, o Unit Of Work simplifica a gestão do ciclo de vida das entidades e facilita, sem obrigar, a utilização desligada (as ligações à base de dados apenas são estabelecidas durante a leitura das entidades e quando se pretende propagar valores para a base de dados, ficando, entretanto, as entidades armazenadas no objeto que implementa o padrão.

4. Detalhes de implementação

4.1 Acesso a procedimentos e funções

No exercício 1 alínea a), é pedido para que o grupo implemente uma solução que permita aceder às funcionalidades 2d a 2l, descritas na fase 1 deste trabalho.

Assim, foi criada uma classe chamada “accessFunctionality” que tem como objetivo possuir os métodos através dos quais é possível aceder às funcionalidades já construídas na fase 1 deste trabalho.

Para todos os métodos que acedem a alguma funcionalidade e que causam alguma mudança na base de dados seguiu-se o seguinte método para o seu desenvolvimento:

-Criação de um mêtodo que recebe os mesmos parâmetros que a funcionalidade que pretendemos aceder.

-Criação de uma nova instância de DataScope.

-Utilização do mêtodo “createNativeQuery” que recebe como parâmetro uma String. que representa a chamada ao procedimento ou função que se pretende interagir.

-Após a criação da query é lhe colocado os respetivos parâmetros, sendo estes os parâmetros do método em que a mesma se encontra, esta ação é feita a partir do método "setParameter" que recebe como primeiro parâmetro a a sua posição e como segundo parâmetro qual será o valor parametrizado em concreto.

-De seguida, é utilizado o método "executeUpdate" que resumidamente executa na base de dados a query que foi criada anteriormente.

-Por fim é utilizado o método "validateWork" da classe DataScope que marca a transação como terminada , permitindo que a as alterações sejam confirmadas ao fechar o scope.

Todo este código descrito encontra-se dentro de um bloco "try", pois caso ocorra alguma exceção será "apanhada" pelo bloco "catch" que se segue.

Exemplo:

```
  public static void criar_jogador(String email, String user-
name, String nome_regiao) throws Exception {
    try (DataScope ds = new DataScope()) {
        EntityManager em = ds.getEntityManager();
        Query query = em.createNativeQuery("CALL criar_joga-
dor(?, ?, ?)");
        query.setParameter(1, email);
        query.setParameter(2, username);
        query.setParameter(3, nome_regiao);
        query.executeUpdate();
        ds.validateWork();
    } catch (Exception e) {
        System.out.println(e.getMessage());
        throw e;
    }
}
```

Caso o método queria aceder a alguma funcionalidade mas não causa nenhuma mudança na base de dados então o processo que se seguiu para concluir a implementação dos mesmos foi equivalente ao que foi anteriormente descrito, no entanto apresenta algumas mudanças:

-Não utiliza o método "executeUpdate", pois não necessita de causar mudanças na base de dados

-Utiliza o método getSingleResult ou getResultList para obter ou o valor, por exemplo um Integer, ou a tabela com as colunas desejadas no retorno.

Exemplo:

```
  public static Integer total_pontos_jogador(Integer id_player)
throws Exception {
    try (DataScope ds = new DataScope()) {
        EntityManager em = ds.getEntityManager();
        Query query = em.createNativeQuery("SELECT totalpon-
tosjogador(?)");
        query.setParameter(1, id_player);
        Integer totalPoints = (Integer) query.getSingleResult();
        ds.validateWork();
```

```
            String message = "Total de pontos: " + totalPoints;
            System.out.println("\n" + message + "\n");
            return totalPoints;
        } catch (Exception e) {
            System.out.println(e.getMessage());
            throw e;
        }
    }
```

4.2 Realizar funcionalidade sem utilizar procedimentos armazenados / funções

No exercício 1 alínea b), é pedido para que o grupo implemente uma solução que replique a alínea 2h já implementada, no entanto sem recorrer a utilização de procedimentos armazenados ou funções.

Para tal implementação, foi necessário a utilização do método "createQuery", "setParameter" e de "getSingleResult" (conceitos já descritos previamente) com especial menção à utilização do Mapper de Cracha para realizar a leitura.

Após obter todos os dados necessários, sendo estes o totalPontos do jogador e o limitePontos do crachá é verificado se os pontos do jogador igualam ou superam o limite de pontos do crachá.

Se a condição verificar-se verdadeira, então é executado um insert na tabela "tem".

No final toda o código que se encontra dentro da transação é validado pelo método "validateWork" proveniente do "DataScope".

4.3 Realizar funcionalidade sem utilizar procedimentos armazenados / funções reutilizando os procedimentos armazenados e funções que a funcionalidade original usa

No exercício 1 alínea c), é pedido para que o grupo implemente uma solução que reutilize os procedimentos armazenados e funções que sejam utilizados pelo procedimento "associarcracha", neste caso a única função utilizada é "pontosjogoporjogador" à semelhança das implementações anteriormente descritas a função auxiliar foi invocada através da criação de uma NativeQuery que posteriormente recebeu os parâmetros correspondentes e retornou após a sua execução.

O seu valor foi usado para a restante resolução desta alínea usando sempre métodos previamente já introduzidos como: "setParameter", "getSingleResult", "executeUpdate", "validateWork", entre outros.

4.4 Realizar uma funcionalidade utilizando Optimistic Locking e Pessimistic Locking

O controlo de concorrência otimista e pessimista são duas estratégias utilizadas em sistemas de base de dados para lidar com situações de concorrência.

No caso do controlo otimista, baseia-se na suposição de que as operações concorrentes não entraram em conflito com frequência.
Nesta estratégia cada transação verifica se houve alterações concorrentes nos dados antes de efetuar as alterações.

Já no controlo pessimista, adota uma abordagem mais cautelosa, assumindo que conflitos de concorrência são prováveis.
Nesta estratégia os recursos são bloqueados para garantir que apenas uma transação tenha acesso exclusivo a eles.

Tendo em conta os conceitos visados no início deste subcapítulo, o grupo desenvolveu as seguintes soluções:

No exercício 2 alínea a), é pedido para que o grupo implemente uma solução que
incremente os pontos associados a um crachá por 20% do seu valor atual, assim desenvolveu-se o seguinte método:

Primeiramente, verifica-se se os parâmetros estão dentro dos supostos requerimentos.
De seguida, a partir de uma leitura com o método “find”, obteve-se o crachá de onde se irá retirar os pontos que são necessários para realizar o incremento por 20%.
Após a aritmética ter sido feita, fez-se um update para colocar o valor atualizado na base de dados.

No que diz respeito a utilizar o controlo de concorrência otimista, colocou-se uma anotação *@OptimisticLocking* na tabela “Cracha” que recebe como parâmetros *OptimisticLockingType.changed_columns*, esta anotação serve para definir o tipo de otimização de bloqueio otimista a ser aplicado a uma entidade em um contexto JPA.

Já no exercício 2 alínea c), é pedido para que o grupo implemente uma solução que incremente os pontos associados a um crachá por 20% do seu valor atual, assim desenvolveu-se o seguinte método:

O método criado é igual ao descrito na alínea a) no entanto o método "find” recebe nos seus parâmetros “*LockModeType.PESSIMISTIC_WRITE*”. Com este parâmetro é garantido que no processo de obter o objeto pretendido o utilizador realiza um *lock* por completo sobre o objeto, não deixando assim que outros utilizadores possam realizar qualquer *read* ou *write* sobre este objeto.

Ambos os métodos utilizam o método “*validateWork*” do “*DataScope*” que está dentro do scope de um bloco “*Try*” seguido de um bloco “*Catch*” com as respetivas exceções e finalmente um bloco “*Finally*” que independentemente do que acontecer utiliza o método “validateWork”.

4.5 Teste

No exercício 2 alínea b), é pedido para que o grupo implemente um teste para alínea 2 a).

Resumidamente, são criadas duas threads que tentam executar o método *optimisticCrachaUpdate* em concorrência, logo vai ser lançada uma exceção devido a esse mesmo acesso concorrente.

Ambas a threads obtêm a informação correta sobre o objeto crachá que vai ser atualizado, porém apenas uma conseguirá realizar a sua atualização, pois quando a segunda thread tenta realizar a atualização encontra uma exceção *OptimisticLocking* devido a objeto que pretende atualizar não ser o mesmo que estava inicialmente quando a thread efetuou a operação read. Como este é o conceito que a concorrência otimista segue, a atualização é impedida.

5. Tratamento de erros

Ao longo do trabalho em diversos locais do projeto, colocou-se blocos “Try Catch”

Para que fosse possível identificar as possíveis exceções que pudessem ocorrer dentro de certos métodos assim é criado um mecanismo “defesa” contra as falhas de implementação ou até mesmo falhas exteriores como falha de conexão à base de dados.

6. Conclusão

Com a resolução deste trabalho os elementos do grupo adquiriram uma nova precessão sobre o desenvolvimento de uma base de dados de “raiz” e de como se possibilita o seu acesso ao exterior através de diferentes mecanismos como o JBDC que é uma API Java para interagir com bancos de dados relacionais. Com o JBDC, fomos capazes de estabelecer conexões com os bancos de dados, criar tabelas, executar consultas SQL e gerenciar transações. Em seguida, descobrimos que o JPA é uma especificação Java para o mapeamento objeto-relacional.

Assim foi possível realizar todas as alíneas especificadas no enunciado.

Com especial menção à dificuldade que o grupo teve no início de estruturar todo o projeto, mas após diversas horas de pesquisa foi possível entender a grande parte dos componentes e os seus propósitos.

Referências

1: Moodleisel 2022 / 2023 --- Slides da turma 42D , Professor Walter Vieira