Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

GameOn - F2

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 segunda fase do trabalho prático da cadeira, este relatório tem como propósito a elaboração de uma aplicação com a base de dados criada para a primeira fase 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 second phase of the course's work assignment, this report aims to create a application with the databese made for phase one 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 Listagens

1

Introdução

Este trabalho foi realizado com o intuito de estudar e desenvolver uma camada de acesso a dados, aplicada ao modelo de dados desenvolvido na fase anterior. Esta camada é utilizada numa aplicação na linguagem Java, com recurso à tecnologia JPA (Jakarta Persistence API).

Inicialmente, implementou-se o modelo de dados, previamente desenhado na primeira fase do projeto, em Java, com recurso a JPA. Para cada entidade do modelo, foi implementada uma classe Java que representa essa mesma entidade. Cada classe tem como propriedades os atributos pertencentes à entidade, e alguns métodos de encapsulamento dos mesmos. Com o modelo de dados implementado em Java, começou-se o desenvolvimento da camada de acesso a dados. Nesta camada, são implementados padrões de acesso a dados lecionados nas aulas, entre eles Mapper, Repository e PersistenceManager. Estes padrões são responsáveis por realizar operações sobre as respetivas entidades, como por exemplo operações CRUD (Create, Read, Update, Delete). Com a camada de acesso a dados implementada, desenvolveram-se as funcionalidades descritas na fase 1 do projeto em Java. Estas funcionalidades foram implementadas em procedimentos armazenados, funções, gatilhos e vistas.

Posteriormente, re-implementou-se uma das funcionalidades utilizando optimistic locking, que é uma técnica usada em aplicações com bases de dados SQL para lidar com o problema da concorência na manipulação de dados. Esta, não mantém locks de linha durante leituras, updates ou deleções, tomando uma abordagem otímistica em como a concorrência será baixa então não o travamos. Em vez disso, quando uma transação

tenta fazer commit, o sistema verifica se o registo foi alterado por outra transação. Se sim, a transação falha. Numa fase final, as peças de software foram devidamente testadas, comprovando o seu correto funcionamento. Apresenta-se também no documento, a discussão das alternativas de modelação e as razões de escolha das soluções apresentadas. Após a realização do trabalho, concluiu-se que os objetivos de aprendizagem foram alcançados, produzindo os resultados pretendidos e conhecimentos da matéria em estudo foram adquiridos com sucesso.

2

Mapeamento para JPA

Tendo como requesito a utilização da JPA *Jakarta Persistence API* que é a *API* que permite a persistência de entidades da aplição java para a base de dados em PostgreSQL.

2.1 Mapeamento de Tipos

Para mapear tipos personalizados da base de dados para o JPA, o grupo para simplificar, em vez de os mapear diretamente com as anotações que o JPA oferece, decidiu implementar novamente a mesma lógica de negocio. Também tendo em mente a portabilidade da aplicação o grupo também decicidu criar novos tipos, estes nao estando em base de dados, tais como os estados de uma partida multijogador bem como os estados possíveis de um jogador.

2.2 Mapeamento Entidades

Sendo de longe a parte mais trabalhosa do nosso trabalho, o grupo teve em mente o seguinte pensamento: "Se de hoje a amanhã mudarmos a tecnologia que impacto teria na manutenção?", com isto achamos à conclusão que seria interesante criar uma camada de abstação das entidades/tabelas da base de dados e como tal para cada Entidade/Tabela criamos uma interface. Não criamos interfaces para as chaves propositadamente, uma vez que achamos que, neste caso em concreto foi um *trade-off* da tecnologia usada

como requerimento. Internamente e relativamente às classes para os tipos criados em db.

2.3 Mapeamento Associações

Tendo pararelismo com as entidades, as associações também foram abstraidas com interfaces.

3

Acesso a dados

O acesso a dados está restringido a *Class JPAContex* uma vez que usamo-lo como o nosso *Unit of Work*. Para a manipulação e pesquisa de entidades foram utilizados tanto Repositorios com Mappers.

3.1 *JPAContex*

O Unit of Work (Unidade de Trabalho) é um padrão de projeto utilizado no JPA (Jakarta Persistence API) para gerenciar operações de persistência em bancos de dados. Ele representa uma transação lógica que agrupa operações relacionadas, como inserções, atualizações e exclusões de entidades. O Unit of Work rastreia as alterações nas entidades, sincroniza-as com o banco de dados e lida com operações de consulta. Ele garante a integridade dos dados, simplifica o desenvolvimento de aplicativos orientados a objetos e oferece controle fino das operações de persistência, mantendo a consistência dos dados ao longo das transações. Em suma, o Unit of Work é essencial no JPA, fornecendo um contexto confiável para a manipulação de entidades e transações.

3.2 Repositorios

O repositório base implementado do qual todos os outros repositórios sao extendidos tem as funções básicas de procura, procura pela chave, procura total ou procura pelos paramentos requesitados.

```java
import java.util.List;

/**
 * Interface that represents a repository
 *
 * @param <T>    Entity
 * @param <TCol> Collection of entities
 * @param <TK>   Key of the entity
 */
public interface Repository<T, TCol, TK> {
    /**
     * Finds an entity by its key
     *
     * @param key Key of the entity
     * @return Entity
     */
    T findByKey(TK key);

    /**
     * Finds a Collection of entities by the given query
     *
     * @param jpql   query to be executed
     * @param params parameters of the query
     * @return Collection of entities
     */
    TCol find(String jpql, Object... params);

    /**
     * Finds all entities
     *
     * @return Collection of entities
     */
    List<T> findAll();
}
```

Listagem 3.1: Repositorio

Apos alguma reflexão o grupo também achou que seria uma boa ideia procurar por parâmetros como o nome e como tal criamos a função *findByName* que retorna se existir

a entidade.

3.3 *Mappers*

Os *Mappers*, semelhante aos repositorios foram implementados a partir de um repositorio genérico e extendidos para as suas respetivas classes.

```
1  /**
2   * Interface that defines the methods that a DataMapper must
      implement to be able
3   * to perform CRUD operations on the database.
4   *
5   * @param <T>  Entity type
6   * @param <TK> Entity key type
7   */
8  public interface DataMapper<T, TK> {
9      /**
10      * Creates a new entity in the database.
11      *
12      * @param entity Entity to be created
13      * @return The key of the created entity
14      */
15     TK create(T entity);
16
17     /**
18      * Reads an entity from the database.
19      *
20      * @param id Key of the entity to be read
21      * @return The entity with the given key
22      */
23     T read(TK id);
24
25     /**
26      * Updates an entity in the database.
27      *
28      * @param entity Entity to be updated
29      * @return The key of the updated entity
30      */
31     TK update(T entity);
```

```
32
33      /**
34       * Deletes an entity from the database.
35       *
36       * @param id Key of the entity to be deleted
37       */
38      void delete(TK id);
39  }
```

Listagem 3.2: *Mapper*

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

4

Implementação das funcionalidades

No nosso trabalho implentámos uma *UI* com simples verificação de input, chamando seguidamente as funcionalidades requesitadas com descrito posteriormente, como por exemplo criar um jogador com as suas informações obrigatórias, após os *inputs* verificados, chamamos o procedimento ou função associada a essa funcionalidade onde ainda seram feitas verificações tanto na aplicação como na base de dados (ja faladas na fase 1 ou implementadas na fase 1).

4.1 *User Interface*

A *UI* e a camada que verifica todo o tipo de inputs passados a aplicação de modo a dar mais rebustez a mesma, esta verifica e valida os inputs tais como identificadores alfanumericos com base em *regular expresions*(*regex*).

4.2 Funcionalidades

Estas funcionalidade foram todas implementadas com o pensamento que de que esta seria uma aplicação cliente e como tal nao deveriamos de pedir por nenhum id, e tendo como *trade-offs* a lentidao de nao pesquisar pela chave das suas entidades.

4.2.1 Criacao de jogador

Esta funcionalidade é acedida na camada de acesso a dados tal como todas as outras funcionalidades. Nesta função apenas estamos a fazer a verificação em aplicação de garantir que a região existe antes de criar o jogador, passando todos os parâmetros validos ao procedimento em base de dados.

4.2.2 *Update* de estado de jogador

Nesta função e após a validação de input pela camada de *UI* seguidamente verificamos se o jogador de facto existe para atualizar o seu estado e so então e que atualizamos o estado do jogador passado os parâmetros ao procedimento correspondente.

4.2.3 Total de pontos de um jogador

Nesta funcionalidade com recurso ao nome de utilizador disponibilizamos os pontos de todas as partidas que o jogador jogou chamando a funcionalidade correspondente em base de dados. Verificamos antes de fazer a chamada a funcionalidade em base de dados se o jogador existe.

4.2.4 Total de Jogos de um Jogador

Esta funcionalidade é análoga à "Total de pontos de um jogador".

4.2.5 Pontos Jogo Por Jogador

Esta funcionalidade dá a informação dos pontos ganhos por cada jogador no jog, mostrando em aplicação os seus respetivos pontos e identificadores.

4.2.6 Associar Crachá

Esta funcionalidade associa um jogador a um cracha, verificando primeiro se o cracha existe antes de o associar.

4.2.7 Iniciar Conversa

Esta funcionalidade inicia um conversa associando o jogador à mesma criando uma mensagem indicadora de que o jagador iniciou a mesma devolvendo também o identificador da mesma.

4.2.8 Juntar jogador a uma Conversa

Esta funcionalidade adiciona um jogador a uma conversa ja existente dando uma mensagem identificadora de que o jogador entrou na conversa, para esta funcionalidade antes de adicionar o jogador a conversa verificamos se a mesma existe.

4.2.9 Enviar Mensagem

Esta funcionalidadde envia uma mensagem numa conversa dado a própria mensagem bem como o nome de utilizador e o identificador da mensagem

4.2.10 Jogador Total Info

Esta funcionalidade disponibiliza a informacao total de um jogador que nao esteja banido, por informacao total de um jogador entende se: identificador, nome de utilizador, email, estado, total de pontos do mesmo, total de partidas participadas bem como o total de jogos jogados.

4.2.11 Associar Crachá Manual

Este procedimente é análogo ao procedimento já explicado anteriormente no entanto em vez de por a lógica de negócio na base de dados esta foi feita o mais possível na aplicação java, infelizmente nao foi possível fazer a funcionalidade toda na aplicação uma vez que precisávamos de insformações que estavam apenas na base de dados, no entanto tentámos minimizar ao máximos todas a *queries* a ela associada.

4.2.12 Aumentar em 20% os pontos de um Cracha com *Optimistic Locking*

Nesta funcionalidade aumentamos o número de pontos requeridos para ganhar um cracha em 20% e porque o JPAContext para as transações já tem como predefinido o

optimistic locking não foi preciso utilizar explicitamente.

4.2.13 Aumentar em 20% os pontos de um Cracha com *Pessimistic Locking*

Nesta funcionalidade tivemos de utilizar o pessimistic locking para aceder à funcionalidade já descrita anteriormente.

5

Validação

A validação feita em aplicação é mínima uma vez que já foi feita quase na totalidade em base de dados, no entanto uma coisa que esta na grande maioria das funcionalidades e verificação de existência de entidades antes de serem feitos pedidos.

Quanto ao testes implementados foram feitas duas classes, um *RepoTest.java* que serve para verificar se os repositorios funcionam bem como verificar se a aplicação consegue fazer pedidos a base de dados. A segunda classe verifica se é possível aumentar em 20% os pontos de um cracha apresentando uma mensagem de erro adequada em caso de alteração concorrente conflituante que inviabilize a operação, neste teste foram inicializadas *Threads* de fora a gerar situacões conflituantes.

5.1 Teste de *Optimistic Locking*

No contexto do projeto, a implementação do optimistic locking permite aumentar em 20% o número de pontos associados a um crachá. Isso é feito ao utilizar transações com bloqueio otimista para evitar conflitos de atualização concorrentes. Caso ocorra uma alteração concorrente conflituante que inviabilize a operação, uma mensagem de erro apropriada é apresentada. Para testar essa funcionalidade, foi criado um cenário de teste em que duas transações simultâneas tentam alterar o mesmo crachá, sendo tratadas corretamente com base em exceções específicas. Essa abordagem garante a consistência dos dados e permite um aumento seguro nos pontos dos crachás.

6

Conclusão

Ao longo desta fase do projecto desenvolvemos uma aplicaçao java para dar suporte a uma base de dados PostgreSQL, para tal foi necessário passar por varios passos:

1. Mapear as tabelas presentes na base de dados e neste caso utilizando JPA;
2. Criar todo um sistema de manipulação e pesquisa de entidades/tuplos na aplicação/base de dados;
3. Criar algum tipo de *UI* neste caso bastante simples;
4. Finalmente ligar todos os componentes com controlo transacional.

Também foi possível melhorar o codigo relativamente a primeira parte do trabalho uma vez que tivemos alguns problemas com testes que comprovaram alguns erros na implentação.

Em suma e apesar de bastantes adversidades principalmente no mapeamento de entidades devido a complexidade do nosso modelo com esta implementação estamos confiantes que cumprimos com todos os requesitos do trabalho.

Melhorias relativamente a 1º fase

Relativamente a 1ª fase houveram alterações e subsequentemente melhorias relativamente ao codigo *PostgreSQL*.

A.1 Alterações

- Onde antes estava o tipo *Date* encontra se agora o tipo *Timestamp*, esta alteração deveu se apos o grupo ter identificado que o tipo *Date* não tinha nenhuma hora associada;

A.2 Melhorias

- Agora todas as funções funcionam o que antes não acontecia uma vez que os valores estavam em campos errados, foi um erro de distração por parte do grupo que nesta fase foi corrigido, subsequentemente os testes agora passam todos;
- Antes para atualizar as estatistícas de jogadores e jogos tinhamos 1 gatilho para cada propriedade agora temos apenas um para cada tabela (de 6 gatilhos para 2).