Gestão de Jogos com JPA

Fase 2

Rafael Pegacho N.º49423
Tiago Neto N.º49431

Sistemas de Informação

Licenciatura em Engenharia Informática e de Computadores
Instituto Superior de Engenharia de Lisboa
Portugal
June 12, 2023

Contents

1 Introdução

Esta fase do trabalho tem como objetivo o desenvolvimento de uma camada de acesso a dados através de uma implementação de JPA e um subconjunto dos padrões DataMapper, Repository e UnitOfWork. Para isto desenvolveremos em Java uma Data Access Layer. É importante garantir a correta implementação de restrições e gerência de ligações e recursos.

É também esperado uma estruturação lógica da arquitetura da aplicação de forma a separar percetívelmente as camadas que acessam a dados da base de dados, as que interagem com o utilizador e as que controlam o fluxo de dados as outras duas camadas.

2 Arquitetura

2.1 Estrutura

Num projeto JPA, existe um certo padrão utilizado que organiza a arquitetura do projeto em camadas. Chama-se a este tipo de arquitetura, arquitetura de três camadas. Sendo estas três camadas as seguintes:

- Presentation: Responsável por suportar interações do utilizador e apresentar-lhe dados. Para isto existe uma interação entre esta camada e as outras duas tornando possível manipular dados.
- BusinessLogic: Nesta camada garantem-se as restrições de integridade e coordena-se o fluxo de dados entre as duas outras camadas.
- Data Access Layer (DAL): Encarregue de fazer as interações com a base de dados é nesta camada onde se incluem componentes como repositórios e mappers com métodos que nos possibilitam operações CRUD e consultas à base de dados.

Para além destas três camadas utilizamos também um módulo "Generic" independente que implementa interfaces genéricas utilizadas na Data Access Layer.

2.2 Módulos

Ao longo do nosso trabalho foram criados vários módulos, cada um deles explicado nesta secção.

2.2.1 BLService

BLService é uma classe que pertence à camada BusinessLogic e tem a função de testar algumas questões maioritariamente transacionais. Cada teste será explicado numa das secções seguintes.

2.2.2 Model

Sendo o módulo Model também pertencente à camada BusinessLogic este representa as entidades da aplicação, definindo a estrutura e relações de entidades sem recorrer a funcionalidades do JPA.

2.2.3 ModelJPA

ModelJPA, ao contrário do Model, é especificamente responsável por fornecer funcionalidades e implementações específicas do JPA. Contendo classes que implementam Entidades da JPA com restrições que recorrem a mecanismos desta mesma API. Estas classes mapeiam a interação entre o Model e a base de dados e dito isto, este módulo pertence então à Data Access Layer.

2.2.4 DataMapper

O padrão DataMapper é utilizado para separar o domínio do objeto da base de dados. Isto permite que os objetos do domínio possam ser manipulados como se não tivessem conexão direta com a base de dados, aumentado a possibilidade de manutenção e reutilização de código.

Cada método neste padrão representa uma operação CRUD (Create, Read, Update, Delete) que pode ser realizada na base de dados. Os métodos create, read, update e delete correspondem respetivamente às operações insert, select, update e delete em SQL

2.2.5 Repository

O padrão Repository adiciona uma camada de abstração entre o domínio do objeto e o acesso a dados, agindo como um meio-termo. Isto permite que as operações do banco de dados sejam manipuladas de uma maneira que esteja de acordo com os conceitos de domínio e regras de negócios.

Este padrão utiliza a sua instância de DataMapper para executar operações do banco de dados e retorna os resultados numa forma que seja útil para o domínio da aplicação.

2.2.6 UnitOfWork

O padrão UnitOfWork gere as operações de transação num conjunto de objetos para garantir que todas as operações sejam concluídas com sucesso ou que nenhuma seja aplicada. Ele rastreia todas as tarefas que o utilizador queira executar durante uma transação de negócios e coordena a gravação de alterações e a resolução de conflitos de concorrência.

Neste caso, o padrão UnitOfWork é implementado por meio da classe DataScope, que controla o ciclo de vida da transação. Cada método no DataMapper e Repository começa criando um novo DataScope e termina validando o trabalho com ds.validateWork(). Se algum erro ocorrer durante a execução, a transação será revertida, garantindo que a base de dados permaneça consistente.

2.2.7 Utils

Sendo o último módulo incluído na DAL, este módulo contém métodos estáticos que realizam a conversão de objetos pertencentes a classes do módulo Model para objetos das entidades JPA e vice-versa. Como exemplo está presente em baixo o código responsável pela conversão de um objeto da classe Partida do Model para um objeto da classe PartidaJPA e vice-versa.

```
public static PartidaJPA toJPA(Partida p) {
        PartidaJPA pj = new PartidaJPA();
        pj.setId(p.getId());
        pj.setJogo(toJPA(p.getJogo()));
        pj.setDataInicio(p.getDataInicio());
        pj.setDataFim(p.getDataFim());
        pj.setRegiao(toJPA(p.getRegiao()));
        return pj;
    }

    public static Partida fromJPA(PartidaJPA pj) {
        Partida p = new Partida();
        p.setId(pj.getId());
        p.setJogo(fromJPA(pj.getJogo()));
        p.setDataInicio(pj.getDataInicio());
        p.setDataFim(pj.getDataFim());
        p.setRegiao(fromJPA(pj.getRegiao()));
        return p;
    }
```

2.2.8 Generic

Neste módulo que não pertence a nenhuma das camadas descritas anteriormente apenas criamos duas interfaces "iRepository" e "iMapper" que vão ser interfaces genéricas implementadas por todas as classes em Mappers e Repository de forma a deixar um código mais organizado e de fácil compreensão.

2.2.9 App

Por fim, no único módulo da camada Presentation, é feita a interação com o utilizador. Neste caso esta interação consiste apenas na escolha pela parte do utilizador do teste que pretende realizar (testes realizados na BusinessLogic).

3 Implementação da DAL

3.1 Implementação da JPA

A Java Persistence API (ou simplesmente JPA) é uma API padrão da linguagem Java que descreve uma interface comum para frameworks de persistência de dados. A implementação da JPA no projeto "GameOn" foi feita para facilitar a gestão de dados, permitindo que operações de banco de dados sejam realizadas de maneira mais orientada a objetos.

3.1.1 Configuração da JPA

O primeiro passo para a implementação da JPA foi a configuração no ficheiro "persistence.xml", que se encontra no diretório "src/main/resources/META-INF". Este arquivo define as propriedades necessárias para estabelecer a conexão com a base de dados, bem como o nome da unidade de persistência (GameOn), que é usado em todo o código para obter o EntityManager.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.2"
             xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">
    <persistence-unit name="GameOn" transaction-type="RESOURCE_LOCAL">
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
            <property name="jakarta.persistence.jdbc.url"
            value="jdbc:postgresql://localhost:5432/?"/>
            <property name="jakarta.persistence.jdbc.user" value="postgres"/>
            <property name="jakarta.persistence.jdbc.password" value="deeznuts"/>
            <property name="jakarta.persistence.jdbc.driver"
            value="org.postgresql.Driver"/>

            <property name="eclipselink.logging.level" value="FINE"/>
            <property name="eclipselink.logging.parameters" value="true"/>
            <property name="eclipselink.logging.level.connection" value="FINEST"/>
            <property name="eclipselink.logging.level.transaction" value="FINEST"/>
            <property name="eclipselink.logging.level.query" value="FINEST"/>
        </properties>
    </persistence-unit>
</persistence>
```

3.1.2 Mapeamento de Entidades

As classes de entidades são representações de tabelas do banco de dados. Cada classe de entidade corresponde a uma linha na tabela correspondente. As classes de entidade foram mapeadas utilizando anotações JPA, como @Entity, @Table, @Id, @Column, entre outras. Isto permite o mapeamento direto de classes para colunas do banco de dados.

Por exemplo, a classe JogadorJPA foi mapeada para a tabela 'jogador' no banco de dados. Cada campo na classe JogadorJPA, como id, email, username, estado e regiao_id foi mapeado para a coluna correspondente na tabela 'jogador' usando a anotação @Column.

```
@Entity
@Table(name = "jogador")
public class JogadorJPA {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
```

```
    @Column(name = "email", nullable = false, unique = true)
    private String email;

    @Column(name = "username", nullable = false, unique = true)
    private String username;

    @Enumerated(EnumType.STRING)
    @Column(name = "estado", nullable = false)
    private Estado estado;

    @ManyToOne
    @JoinColumn(name = "regiao_id", nullable = false)
    private RegiaoJPA regiao;

    public enum Estado {
        Ativo,
        Inativo,
        Banido
    }

    public JogadorJPA() {
    }

    //getters e setters
}
```

3.1.3 Operações de Banco de Dados

A JPA permite a execução de operações de banco de dados de maneira orientada a objetos, utilizando o EntityManager. Este é o ponto de entrada para realizar operações de persistência. Os principais métodos utilizados foram persist() para inserir, merge() para atualizar, remove() para apagar e find() para selecionar.

3.2 DataMapper, Repository

3.2.1 DataMapper

Para a implementação dos Mappers falados na secção da arquitetura do projeto bem como as operações CRUD utilizamos a seguinte interface genérica iMapper falada tambêm nessa secção:

```
public interface IMapper<T, TId> {
    TId create(T e) throws Exception;

    T read(TId id) throws Exception;

    void update(T e) throws Exception;

    void delete(T e) throws Exception;
}
```

Observa-mos assim que, à exceção da operação read todas as operações CRUD de cada repositório são feitas passando como parâmetro um objeto da classe desse mesmo repositório. Sendo a operação read essa exceção onde é apenas passada a chave primária do objeto que se pretende ler esta função retorna um objeto do tipo ¡T¿ sendo também a única função com este retorno. Observamos também que a função create é a unica das que recebe um objeto ¡T¿ que retorna a chave primária desse objeto. Dito isto implementou-se todos os Mappers necessários.

3.2.2 Repository

Sendo que os Repositorys seguem também uma interface genérica iRepository implementada da seguinte forma.

```
    List<Tentity> getAll() throws Exception;

Tentity find(Tkey k) throws Exception;

void add(Tentity entity) throws Exception;

void delete(Tentity entity) throws Exception;

void save(Tentity e) throws Exception;
```

Concluímos então que todos os repositórios precisam de pelo menos essas cinco funções. Reparando nas funções que recebem algum parâmetro e olhando para o tipo deste parâmetro e o seu retorno rapidamente se concluiu que as funções add, delete, save, find podem ser implementadas recorrendo às funções create, delete, update, read presentes nos mappers da entidade em questão.

3.3 Tratamento de erros e gestão transacional

No contexto desta aplicação, o tratamento de erros e a gestão transacional foram implementados com o uso de exceções e do padrão UnitOfWork, como descrito a seguir.

3.3.1 Tratamento de Erros

O tratamento de erros na aplicação é realizado principalmente através do uso de estruturas try-catch. Cada operação do banco de dados é cercada por um bloco try-catch para intercetar qualquer exceção que possa ser lançada durante a execução.

No caso de uma exceção ser lançada, a mensagem de erro é primeiro impressa na consola através do System.out.println(e.getMessage()) e, de seguida, a exceção é relançada com o comando throw e. Isto permite que o erro seja capturado e tratado num nível superior, se necessário, proporcionando uma maior flexibilidade para o controlo de erros.

3.3.2 Gestão Transacional

A gestão de transações é um aspeto crucial da manipulação de dados numa base de dados. Para garantir a atomicidade das operações do banco de dados, a classe DataScope foi implementada utilizando o padrão UnitOfWork.

Cada método em cada DataMapper e Repository começa criando um novo DataScope e termina validando o trabalho com ds.validateWork(). Isto garante que todas as operações do banco de dados sejam agrupadas como uma única transação e que todas sejam concluídas com sucesso ou que nenhuma seja aplicada se um erro ocorrer. Desta forma, a integridade dos dados é mantida mesmo no caso de falhas do sistema.

4 Desenvolvimento de Funcionalidades

4.1 Acesso às funcionalidades (2d a 2l)

Como pedido no enunciado torna-mos possível o acesso às funcionalidades descritas nas alíneas 2d a 2l da primeira fase do projeto.

4.1.1 Consultas Nativas (2d a 2g)

A implementação de funcionalidades através de consultas nativas é direta e permite-nos manipular a base de dados de maneira eficaz. Usamos estas consultas para criar, desativar, e banir jogadores, e também para consultar informações como o número total de pontos e o número total de jogos de um jogador. Para cada uma destas funcionalidades, chamamos a função apropriada (por exemplo, 'criarJogador', 'desativarJogador', 'banirJogador', 'totalPontosJogador' e 'totalJogosJogador') passando os parâmetros necessários e executamos a consulta diretamente na base de dados.

4.1.2 Procedimentos Armazenados (2h a 2k)

Os procedimentos armazenados são usados para executar operações mais complexas que podem necessitar de várias etapas. Eles encapsulam estas operações como rotinas que são armazenadas na base de dados e podem ser chamadas sempre que necessário.

A associação de crachás a jogadores, por exemplo, é realizada através do procedimento armazenado 'associarCracha'. Semelhantemente, permitimos aos jogadores iniciar e juntar-se a uma conversa através dos procedimentos armazenados 'iniciarConversa' e 'juntarConversa', respetivamente. O procedimento 'enviarMensagem' é usado para permitir que os jogadores enviem mensagens durante uma conversa.

Para usar estes procedimentos, chamamos com os parâmetros apropriados, tal como fazemos com as consultas nativas, mas ao contrário das consultas nativas, os procedimentos armazenados permitem executar uma série de operações numa única call.

4.1.3 Vista (2l)

A vista 'jogadorTotalInfo', criada no banco de dados, oferece um acesso simplificado a detalhes de um jogador a partir de múltiplas tabelas. Para a sua manipulação no código, criamos uma entidade correspondente, 'JogadorTotalInfo'.

Essa entidade mapeia os resultados da vista para um objeto manipulável, com atributos alinhados às colunas da vista e anotações JPA para mapeamento apropriado. Isto permite consultas ao 'JogadorTotalInfo' de forma idêntica a outras entidades, facilitando a obtenção de informações do jogador e tornando o código mais eficiente e legível.

4.2 Realização da funcionalidade 2h sem procedimentos armazenados

A funcionalidade 2h consiste na associação de um crachá a um jogador. Esta funcionalidade é realizada sem o uso de procedimentos armazenados, utilizando somente consultas JPQL e operações CRUD básicas do JPA.

A função associarCracha é responsável por realizar esta funcionalidade. Ela inicia abrindo uma nova DataScope para isolar a transação. Depois, ela usa o EntityManager para realizar várias operações na base de dados.

- **JogadorJPA player = em.find(JogadorJPA.class, jogadorId);**: A função primeiro procura o jogador na base de dados. Se o jogador não for encontrado, uma exceção é lançada.
- **JogoJPA jogo = em.find(JogoJPA.class, jogoReferencia);** e **CrachaJPA cracha = em.find(CrachaJPA.class, nomeCracha);**: Da mesma forma, o jogo e o crachá também são procurados na base de dados. Se qualquer um deles não for encontrado, uma exceção é lançada.
- **CrachaJPA crachaExiste = (CrachaJPA) em.createQuery(...)**: A função então verifica se o crachá existe para o jogo especificado. Se não, uma exceção é lançada.

- **Long normalPoints = em.createQuery(...)**: A função calcula a soma de pontos do jogador em partidas normais para o jogo especificado.
- **Long multiPlayerPoints = em.createQuery(...)**: Semelhante mente, a soma de pontos do jogador em partidas multiplayer também é calculada.
- **if (totalPoints ¿= cracha.getLimitePontos()) ...**: Se a soma total de pontos for maior ou igual ao limite de pontos do crachá, o crachá é associado ao jogador.
- **ds.validateWork();**: Finalmente, a transação é validada.

Caso ocorra algum erro durante a execução da função, a exceção é capturada, uma mensagem de erro é impressa e a exceção é lançada novamente. Isto permite que o erro seja tratado adequadamente.

4.3 Realização da funcionalidade 2h com procedimentos armazenados

A funcionalidade 2h, que trata da associação de um crachá a um jogador, pode também ser implementada através do uso de procedimentos armazenados. Isto permite a execução de uma rotina complexa diretamente na base de dados, potencialmente aumentando a eficiência e reduzindo a quantidade de dados transmitidos entre a aplicação e a base de dados.

O método associarCracha é usado para realizar a funcionalidade 2h utilizando um procedimento armazenado chamado associarCracha. A transação é isolada através da criação de uma nova DataScope.

- **StoredProcedureQuery query = em.createStoredProcedureQuery("associarCracha")**: A função começa por criar uma StoredProcedureQuery a partir do EntityManager. Esta query refere-se ao procedimento armazenado associarCracha que está implementado na base de dados.
- **.registerStoredProcedureParameter(...)**: Os parâmetros de entrada do procedimento armazenado são registados. Estes incluem o ID do jogador, a referência do jogo e o nome do crachá.
- **.setParameter(...)**: Os valores dos parâmetros são então definidos.
- **query.execute();**: O procedimento armazenado é então executado.
- **ds.validateWork();**: Finalmente, a transação é validada.

Semelhante à implementação sem o uso de procedimentos armazenados, os erros que ocorrem durante a execução da função são capturados e tratados. Uma mensagem de erro é impressa e a exceção é lançada novamente, permitindo ser tratado adequadamente do erro.

5 Gestão de Concorrência

5.1 Implementação de locking otimista

O locking otimista é uma técnica de controle que permite que várias transações leiam e escrevam dados sem bloquear uns aos outros. Esta estratégia assume que conflitos são raros e, portanto, permite que as transações prossigam sem bloqueios. No entanto, quando um conflito é detetado (ou seja, duas transações tentam modificar os mesmos dados ao mesmo tempo), uma delas será forçada a retroceder.

O método aumentarPontosCrachaOptimistic é usado para implementar o locking otimista. Este método tenta aumentar os pontos de um crachá, dado o nome do crachá e a referência do jogo.

Para o controle otimista, foi adicionada a anotação @Version a um campo na entidade CrachaJPA. Isto indica ao JPA para aumentar automaticamente o valor deste campo sempre que a entidade é atualizada. Quando a entidade é atualizada, o JPA irá verificar o valor do campo @Version na entidade com o valor correspondente no banco de dados. Se eles não coincidirem, significa que outra transação alterou a entidade, e uma OptimisticLockException é lançada.

- **CrachaJPA cracha = em.createQuery(...)**: Uma consulta é realizada para recuperar o crachá com o nome e referência do jogo fornecidos.
- **.setLockMode(LockModeType.OPTIMISTIC)**: O modo de bloqueio otimista é definido para a consulta.
- **cracha.setLimitePontos(...)**: Os pontos do crachá são aumentados.
- **em.merge(cracha)**: O crachá atualizado é devolvido de volta ao EntityManager.
- **ds.validateWork()**: Finalmente, a transação é validada.

Os erros que ocorrem durante a execução da função são capturados e tratados. Se ocorrer um OptimisticLockException, significa que ocorreu um conflito. Isto resulta na impressão de uma mensagem de erro. Para outros tipos de exceções, a mensagem de erro também é impressa, mas a exceção é lançada novamente para permitir o tratamento adicional de erros.

5.2 Teste de alteração concorrente conflituante

5.3 Implementação de locking pessimista

A implementação de controle pessimista tem como objetivo garantir que, ao acessar dados num ambiente concorrente, um único processo ou transação tenha acesso de escrita a um determinado registo de cada vez. Isto é conseguido bloqueando o registo para outros processos enquanto ele está a ser alterado.

A função aumentarPontosCrachaPessimist implementa o locking pessimista usando o método setLockMode com a opção LockModeType.PESSIMISTIC_WRITE. Isto faz com que, ao iniciar a transação, seja adquirido um bloqueio de escrita no registo do crachá, impedindo que outros processos alterem esse registo até que a transação atual seja concluída.

- **CrachaJPA cracha = em.createQuery(...)**: Primeiro, a função realiza uma consulta para obter o registro do crachá que será alterado.
- **.setLockMode(LockModeType.PESSIMISTIC_WRITE)**: De seguida, um bloqueio de escrita é adquirido para esse registo. Isto garante que nenhum outro processo possa alterar este registo enquanto a transação atual estiver sendo processada.
- **cracha.setLimitePontos(...)**: A função então procede para alterar o valor dos pontos do crachá.
- **em.merge(cracha)**: A alteração é estabelecida na base de dados.
- **ds.validateWork()**: Finalmente, a transação é validada e o bloqueio é libertado.

Se ocorrer um erro durante a execução da função, uma mensagem de erro é impressa e a exceção é lançada novamente. Isto permite que o erro seja tratado adequadamente pela função. O bloqueio adquirido no registo também será libertado, permitindo que outros processos possam acessar o registo.

6 Testes

Durante a execução dos testes, foram realizadas diferentes operaçoes. Abaixo está um resumo das ações e resultados obtidos em cada teste.

No Teste 0, foi realizada a criação de um jogo, com referência "JOGO0", nome "jogo0" e URL "jogo0.com". Esta operação foi feita utilizando um Mapper específico para jogos.

No Teste 1, ocorreu a criação de uma região, com nome "regiao1" e a criação de um jogo, com referência "JOGO1", nome "jogo1" e URL "jogo1.com". Ambas as operações foram realizadas dentro de uma mesma transação.

No Teste 2, foram realizadas diversas operações utilizando um Repository para gerir as entidades de região. Primeiramente, uma região com ID 1 e nome "regiao2" foi adicionada. A seguir, a região foi buscada e exibida. O nome da região foi alterado para "nao regiao2" e novamente buscada e exibida. Foi realizada a obtenção de todas as regiões, exibindo os seus IDs e nomes. Todas estas operações foram executadas dentro de uma mesma transação.

No Teste 3, foi informado que não é possível utilizar JPA com a implementação baseada em JDBC.

No Teste 4, ocorreu a utilização do DataScope (UnitOfWork). Foi inserida uma região com ID 1 e nome "regiao4". A região foi buscada e exibida. O nome da região foi atualizado para "nao regiao 4" e a transação foi validada.

No Teste 5, foi inserida uma região com ID 2 e nome "regiao5". De seguida, o Teste 4 foi executado, realizando operações de inserção, busca, atualização e validação numa transação separada. Após a conclusão do Teste 4, a região inserida foi buscada e exibida. O nome da região foi alterada para "nao regiao 5" e a transação foi validada.

Estes testes demonstraram diferentes abordagens na gestão de persistência de dados, utilizando mappers, repositories e UnitOfWork. Cada teste possui um propósito específico e contribui para a compreensão e verificação do funcionamento do sistema.

7 Conclusão

Tendo finalizado esta segunda fase prática do projeto, concluímos a necessidade de uma boa arquitetura e estrutura da aplicação. Mantendo assim a viabilidade e versatilidade de manutenção de código. Viemos também a perceber a utilidade da utilização de interfaces genéricas que implementam outras classes e os tipos de anotações que possibilitam restrições quer a nivel da JPA ou a nível do Domínio (model). Sendo uma matéria com informação e abordagens diferentes, seguimos os exemplos dados pelo docente da disciplina, com algumas alterações que achamos devidas, dando assim forma a um pensamento crítico em relação ao projeto.