



ISEL
INSTITUTO SUPERIOR DE
ENGENHARIA DE LISBOA

Nível Aplicacional da Base de Dados Para a Empresa GameOn

José Alves

Alexandre Severino

Diogo Carichas

Orientadores Walter Vieira

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

Maio de 2023

Resumo

A camada aplicacional de uma base de dados procura disponibilizar uma interface para um eventual programador fazer uso dos dados de maneira fácil, segura e organizada. Através de uma base de dados PostgreSQL e uma aplicação Java que utiliza a biblioteca JPA para criar um modelo com classes e aceder à base de dados remota. Este modelo é organizado em diferentes níveis e permite facilmente interagir com os dados.

Palavras-chave: aplicacional;interface;programador;PostgreSQL;Java;JPA;modelo;dados

Abstract

The applicational layer of a database intends to make an interface available for an eventual programmer to make use of the data in an easy, safe and organized way. Through a PostgreSQL database and a Java application which utilizes the JPA library to create a model with classes and access the remote database. This model is organized in different levels and allows easy interaction with the data.

Keywords: applicational;interface;programmer;PostgreSQL;Java;JPA;model;data

Índice

| | | |
|----------|---|-----------|
| 1 | Introdução | 1 |
| 1.1 | Objetivos | 1 |
| 2 | Modelo de dados | 2 |
| 2.1 | Embeddables | 2 |
| 2.2 | Relations | 2 |
| 2.3 | Tables | 3 |
| 3 | Funcionalidade da Aplicação | 4 |
| 3.1 | Acesso às Funcionalidades da Base de Dados | 4 |
| 3.1.1 | Funções | 4 |
| 3.1.2 | Procedimentos armazenados | 6 |
| 3.1.3 | Vistas | 9 |
| 3.2 | Realização da Funcionalidade 2h | 9 |
| 3.3 | Funcionalidade Adicional | 11 |
| 3.3.1 | Testes da Funcionalidade | 12 |
| 3.4 | Aplicação | 13 |
| 4 | Conclusão | 14 |
| 4.1 | Aspetos a melhorar | 14 |
| A | Código das Classes do <i>Package</i> model.embeddables | 16 |
| B | Código das Classes do <i>Package</i> model.relations | 22 |
| C | Código das Classes do <i>Package</i> model.tables | 26 |
| D | Código das Classes do <i>Package</i> businessLogic | 39 |
| E | Código das Classes do <i>Package</i> businessLogic.BLServicesUtils | 45 |
| F | Código das Classe App | 48 |

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Uma base de dados por ela mesma tem o seu mérito, quando bem estruturada e implementada é uma mais valia para qualquer empresa que procure organizar a sua informação. No entanto esta não deve permanecer sozinha, se for acompanhada por uma aplicação capaz de comunicar numa outra linguagem de programação, a base de dados pode ser utilizada por aplicações e programas de computador com mais flexibilidade no seu uso.

Por forma a cumprir este objetivo foi desenvolvida uma aplicação *Java* que faz uso da biblioteca JPA, através da qual irá criar um modelo de dados e comunicar com a base de dados remota.

1.1 Objetivos

Em primeiro lugar é necessário implementar o modelo de dados ao nível aplicacional. Para este efeito a biblioteca JPA permite usar anotações do *Java* para determinar o tipo de cada campo de uma classe e como este deve interagir com os tipos de *PostgreSQL* [1].

Esta aplicação deverá ser capaz de permitir a um programador que a utilize de aceder a todas as funcionalidades presentes na fase 1, logo deverá implementar funções que façam uso das funções, procedimentos armazenados, vistas e gatilhos presentes no modelo de base de dados.

Para além de apresentar uma interface para acesso à base de dados esta aplicação deve também permitir ao programador escolher certas funcionalidades entre *optimistic locking* e *pessimistic locking*.

Capítulo 2

Modelo de dados

Antes de passar à implementação das funcionalidades da base de dados é necessário primeiro organizar as tabelas e relações em classes de *Java* correspondentes. Este modelo está dividido em três *packages* diferentes, todos dentro do *package model*. Dentro deste encontra-se **embeddables**, o qual contém classes de valores embebidos, ou seja, identificadores complexos que podem englobar mais que um atributo. Encontram-se também o *package relations*, que contém as tabelas que representam relações na base de dados e o *package tables*, o qual contém as tabelas principais.

Cada uma destas classes contém as anotações que sejam necessárias. Realça-se o facto de que as classes dentro do *package relations* têm a anotação **@Entity**. Esta característica deve-se ao facto de todas as tabelas da base de dados, incluindo as que representam relações, serem tratadas como entidades.

2.1 Embeddables

As classes dentro deste *package* servem como auxílio à implementação das classes mais complexas, pois estas permitem ter identificadores complexos que sejam representados por mais que um atributo. Todas estas classes incluem o uso da anotação **@Embeddable**.

A implementação destas classes encontra-se no anexo A.

2.2 Relations

Este *package* contém as classes que identificam relações do modelo que foram implementadas como tabelas na base de dados. Estas estão colocadas separadamente pois não representam uma entidade no modelo ER, mas sim uma relação. Todas contém as anotações **@Entity** e **@Table**, com o valor do campo **name** igual à tabela correspondente da base de dados PostgreSQL.

Como todas estas são representadas através de identificadores de outras entidades, todas têm um campo com a anotação **@EmbeddedId**, sendo este campo do tipo correspondente do *package embeddables*. A implementação destas classes encontra-se no anexo B.

2.3 Tables

Neste *package* encontram-se as tabelas da base de dados que representam entidades no modelo ER. Estas fazem uso das classes presentes em relations para ser mais fácil de aceder a certas informações, como, por exemplo, a lista de jogadores que compraram um jogo ou a lista de jogos que um jogador comprou. Estes estão presentes como campos das classes **Jogo** e **Jogador**, respetivamente.

Existem ainda funções que acompanham estas implementações pois todos estes campos estão declarados como **private**, sendo apenas possível aceder fazendo uso de um *getter* e alterando o seu valor com um *setter*.

A implementação destas classes encontra-se no anexo C.

Capítulo 3

Funcionalidade da Aplicação

Uma vez o modelo feito é agora possível proceder à implementação das funcionalidades desejadas do trabalho. Estas requerem primeiro acesso às funções e procedimentos armazenados criados na primeira parte, os quais vêm em formato das alíneas do primeiro enunciado 2d até 2l.

Com o acesso a estas funcionalidades obtido, procede-se à implementação da alínea 2h como na fase 1 mas sem usar qualquer procedimento armazenado ou função `pgSql`, ou seja, limitando as possibilidades para usar apenas interações entre JPA e PostgreSQL. Após esta implementação é realizada outra vez mas sem a limitação, o qual permite facilmente comparar as duas implementações e concluir qual será melhor.

Esta aplicação também permite aumentar em 20% o número de pontos associados a um crachá, o qual foi implementado com *optimistic locking* e *pessimistic locking*. A versão que utiliza *optimistic locking* vem acompanhada de testes que verificam que uma mensagem de erro é levantada quando existe uma alteração concorrente que inviabilize a operação.

Business Logic O *package* `businessLogic`, cujo código das suas classes está presente no anexo D, contém todas as funções representadas neste capítulo. Dentro deste *package* encontra-se também `BLServiceUtils`, presente no anexo E, o qual contém funções úteis para a classe `BLService`.

3.1 Acesso às Funcionalidades da Base de Dados

O primeiro requisito é disponibilizar as funcionalidades presentes na base de dados, começando com o exercício 2d da primeira fase até ao exercício 2l. Estas são apresentadas na forma de funções da aplicação.

3.1.1 Funções

Nesta secção encontram-se as funções chamadas tal como na base de dados PostgreSQL.

Exercício 2d Este exercício consiste em duas funções distintas, tal como na base de dados, pois estas efetuam operações diferentes. As duas funções são `createPlayer` e `setPlayerState`. Nas listagens 1 e 2 encontram-se as funções `createPlayer` e `setPlayerState`, respetivamente. Realça-se nestas implementações que a forma de chamar um função em PostgreSQL é como realizar uma *query* para a função, tal como a *string* criada na primeira linha de cada função.

```
1 public String createPlayer(String username, String email, String regiao) {
2     String query = "SELECT createPlayer(?1, ?2, ?3)";
3     Query functionQuery = em.createNativeQuery(query)
4         .setParameter(1, username)
5         .setParameter(2, email)
6         .setParameter(3, regiao);
7     return (String) functionQuery.getSingleResult();
8 }
```

Listing 1: Código da função `createPlayer`

```
1 public String setPlayerState(String playerName, String newState) {
2     int idJogador = modelManager.getPlayerByEmail(playerName, em).getId();
3     String query = "SELECT setPlayerState(?1, ?2)";
4     Query functionQuery = em.createNativeQuery(query)
5         .setParameter(1, idJogador)
6         .setParameter(2, newState);
7     return (String) functionQuery.getSingleResult();
8 }
```

Listing 2: Código da função `setPlayerState`

Exercício 2e Este exercício apenas requer a implementação de uma função. Esta também tem apenas como objetivo chamar uma função da base de dados e não requer nível de isolamento acima do que está por definição em PostgreSQL. Tal como nas funções em 2d, esta apenas realiza uma *query* para a função na base de dados. A implementação encontra-se na listagem 3.

```
1 public Long totalPontosJogador(String email) {
2     int idJogador = modelManager.getPlayerByEmail(email, em).getId();
3     String query = "SELECT totalPontos from totalPontosJogador(?1)";
4     Query functionQuery = em.createNativeQuery(query);
5     functionQuery.setParameter(1, idJogador);
6     return (Long) functionQuery.getSingleResult();
7 }
```

Listing 3: Código da função `totalPontosJogador`

Exercício 2f Tal como as outras funções, esta é chamada através de uma *query*. Este exercício pretende permitir saber quantos pontos um jogador obteve no total das suas partidas em vários jogos.

```
1 public Long totalJogosJogador(String email) {
2     int idJogador = modelManager.getPlayerByEmail(email, em).getId();
3     String query = "SELECT totalJogos from totalJogosJogador(?)";
4     Query functionQuery = em.createNativeQuery(query);
5     functionQuery.setParameter(1, idJogador);
6     return (Long) functionQuery.getSingleResult();
7 }
```

Listing 4: Código da função totalJogosJogador

Exercício 2g Esta função é mais complexa que as outras devido ao facto de retornar uma tabela e não apenas um valor ou o resultado de uma operação e não há entidade para este resultado. Para resolver o problema é possível seguir várias opções: usar a função `getResultList` e colocar numa lista de arrays de objetos da classe `Object`, fazer duas queries, uma para os jogadores e outra para as pontuações e depois agrupá-las num objeto da classe `Map`, ou criar uma entidade apenas para os resultados desta função. A solução escolhida foi a primeira, tal como implementado na listagem 5

```
1 public Map<Integer, BigDecimal> PontosJogosPorJogador(String gameName) {
2     String idJogo = modelManager.getGameByName(gameName, em).getId();
3     String queryString = "SELECT jogadores, pontuacaoTotal from PontosJogosPorJogador(?)";
4     Query query = em.createNativeQuery(queryString);
5     query.setParameter(1, idJogo);
6     Map<Integer, BigDecimal> map = new java.util.HashMap<>(Map.of());
7     List<Object[]> list = query.getResultList();
8     for (Object[] obj : list) {
9         Integer idJogador = (Integer) obj[0];
10        BigDecimal points = (BigDecimal) obj[1];
11        map.put(idJogador, points);
12    }
13    return map;
14 }
```

Listing 5: Código da função PontosJogosPorJogador

3.1.2 Procedimentos armazenados

Esta secção cobre a chamada aos procedimentos armazenados da base de dados. Estes requerem um tratamento diferente pois podem ter níveis de transação específicos.

Exercício 2h Este procedimento da base de dados permite associar um crachá a um jogador dados o identificador do jogador, o identificador do jogo e o nome do crachá. Esta transação necessita de um nível de isolamento `TRANSACTION_REPEATABLE_READ`, tal como se identifica na linha 6 da listagem 6. No final da transação é necessário fazer `transaction.commit()`, tal como na linha 12. Caso falhe, irá fazer `transaction.rollback()`, presente na linha 15.

```
1 public void associarCracha(int idJogador, String gameName, String nomeCracha) {
2     EntityTransaction transaction = transactionManager.startTransaction();
3     Connection cn = em.unwrap(Connection.class);
4     String idJogo = modelManager.getGameByName(gameName, em).getNome();
5     try {
6         transactionManager.setIsolationLevel(cn, Connection.TRANSACTION_REPEATABLE_READ, transaction);
7         try (CallableStatement storedProcedure = cn.prepareCall("call associarCracha(?,?, ?)") {
8             storedProcedure.setInt(1, idJogador);
9             storedProcedure.setString(2, idJogo);
10            storedProcedure.setString(3, nomeCracha);
11            storedProcedure.executeUpdate();
12            transaction.commit();
13        }
14    } catch (Exception e){
15        if(transaction.isActive()) transaction.rollback();
16    }
17 }
```

Listing 6: Código da função associarCracha

Exercício 2i Assim como no exercício 2h, este procedimento armazenado necessita de um nível de isolamento `TRANSACTION_REPEATABLE_READ`. Este procedimento cria uma nova conversa e coloca o jogador que a iniciou dentro dela. Este procedimento coloca o identificador da conversa no seu terceiro argumento, o qual é criado na linha 4 da listagem 7. Este valor é depois retornado. Caso a transação falhe, retorna explicitamente `null`, pois não se garante que `idConversa` permaneça com o valor `null` em caso de falha e consequente `rollback`.

```
1 public Integer iniciarConversa(int idJogador, String nomeConversa) {
2     EntityTransaction transaction = transactionManager.startTransaction();
3     Connection cn = em.unwrap(Connection.class);
4     Integer idConversa = null;
5     try {
6         transactionManager.setIsolationLevel(cn, Connection.TRANSACTION_REPEATABLE_READ, transaction);
7         try (CallableStatement storedProcedure = cn.prepareCall("call iniciarConversa(?,?, ?)") {
8             storedProcedure.setInt(1, idJogador);
9             storedProcedure.setString(2, nomeConversa);
10            storedProcedure.registerOutParameter(3, Types.INTEGER);
11            storedProcedure.executeUpdate();
12            idConversa = storedProcedure.getInt(3);
13            transaction.commit();
14        }
15    } catch (Exception e){
16        if(transaction.isActive()) transaction.rollback();
17        return null;
18    }
19 }
```

```

14     }
15 } catch(Exception e){
16     if(transaction.isActive()) transaction.rollback();
17     return null;
18 }
19 return idConversa;
20 }

```

Listing 7: Código da função iniciarConversa

Exercício 2j Este procedimento é dos mais simples, sendo que apenas necessita de um nível de isolamento `TRANSACTION_READ_COMMITTED`. Com o identificador de um jogador e o identificador de uma conversa, colocará esse par na tabela `jogador_conversa`, a qual indica que jogadores pertencem a cada conversa.

```

1 public void juntarConversa(int idJogador, int idConversa) {
2     EntityTransaction transaction = transactionManager.startTransaction();
3     Connection cn = em.unwrap(Connection.class);
4     try {
5         try (CallableStatement storedProcedure = cn.prepareCall("call juntarConversa(?,?)")) {
6             storedProcedure.setInt(1, idJogador);
7             storedProcedure.setInt(2, idConversa);
8             transaction.commit();
9         }
10    } catch(Exception e){
11        if(transaction.isActive()) transaction.rollback();
12    }
13 }

```

Listing 8: Código da função juntarConversa

Exercício 2k Neste exercício o procedimento é, tal como em 2j, mais simples e requer apenas um nível de isolamento `TRANSACTION_READ_COMMITTED`. Usando o identificador de um jogador, o qual envia a mensagem, o identificador da conversa para onde a mensagem é enviada e o conteúdo da mensagem em questão. Esta função apenas cria uma transação e realiza `commit` em caso de sucesso e `rollback` em caso de falha.

```

1 public void enviarMensagem(int idJogador, int idConversa, String content) {
2     EntityTransaction transaction = transactionManager.startTransaction();
3     Connection cn = em.unwrap(Connection.class);
4     try {
5         transactionManager.setIsolationLevel(cn, Connection.TRANSACTION_READ_COMMITTED, transaction);
6         try (CallableStatement storedProcedure = cn.prepareCall("call enviarMensagem(?,?, ?)")) {
7             cn.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
8             storedProcedure.setInt(1, idJogador);
9             storedProcedure.setInt(2, idConversa);

```

```

10         storedProcedure.setString(3, content);
11         storedProcedure.executeUpdate();
12         transaction.commit();
13     }
14 } catch(Exception e){
15     if(transaction.isActive()) transaction.rollback();
16 }
17 }

```

Listing 9: Código da função enviarMensagem

3.1.3 Vistas

Esta última subsecção representa a implementação da vista presente na base de dados.

Exercício 21 Esta vista permite obter a informação de todos os jogadores com exceção dos que têm o estado 'banido'. Para obter os valores desta tabela optou-se por criar uma entidade que representa estes resultados, pois uma vista funciona tal como uma tabela e, devido a esta característica, é efetuada uma *query* normal. Após obter os resultados esta função coloca-os no *standard output*.

```

1 public void jogadorTotalInfo(){
2     String query = "Select * from jogadorTotalInfo";
3     Query getAllInfo = em.createNativeQuery(query, JogadorTotalInfo.class);
4     List<JogadorTotalInfo> allInfo = getAllInfo.getResultList();
5     for(JogadorTotalInfo jogador: allInfo){
6         System.out.println(
7             jogador.getEstado() + " " +
8             jogador.getEmail() + " " +
9             jogador.getUsername() + " " +
10            jogador.getJogosParticipados() + " " +
11            jogador.getPartidasParticipadas() + " " +
12            jogador.getPontuacaoTotal() + " "
13        );
14    }
15 }

```

Listing 10: Código da função jogadorTotalInfo

3.2 Realização da Funcionalidade 2h

Originalmente este procedimento armazenado é chamado diretamente. No entanto implementou-se a função `associarCrachaModel`, a qual, com os mesmos parâmetros que a função `associarCracha`, efetua o mesmo procedimento mas fazendo uso de ferramentas da aplicação ao invés de usar funções `pgsql`.

Esta função começa por criar uma **TypedQuery**, na qual pretende determinar se o cracha pretendido existe, presnete na listagem 11.

```
1 String idJogo = modelManager.getGameByName(gameName, em).getId();
2 TypedQuery<String> crachaExistsQuery =
3     em.createQuery(
4         "select c.id.jogo from Cracha c where c.id.jogo = :idJogo and c.id.nome = :nomeCracha",
5         String.class
6     );
7
8 crachaExistsQuery.setParameter("idJogo", idJogo);
9 crachaExistsQuery.setParameter("nomeCracha", nomeCracha);
10 String foundIdJogo = crachaExistsQuery.getSingleResult();
```

Listing 11: Excerto da função `associarCrachaModel` para determinar se o cracha existe

O excerto da listagem 11 obtém um id de um jogo, o qual necessita de ser igual ao dado como parâmetro. Uma vez sabendo esta informação, é agora possível efetuar a *query* `getLimitPoints`, a qual permite saber qual o limite de pontos necessários para que um jogador obtenha esse crachá. O excerto da listagem 12 contém o código necessário para obter esta informação.

```
1 TypedQuery<Integer> getLimitPoints = em.createQuery(
2     "select c.limite from Cracha c where c.id.nome = :nomeCracha",
3     Integer.class
4 );
5 getLimitPoints.setParameter("nomeCracha", nomeCracha);
6 Integer limit = getLimitPoints.getSingleResult();
```

Listing 12: Excerto da função `associarCrachaModel` para determinar o limite de pontos do crachá

Após saber o limite do crachá, é preciso garantir que o jogador cumpre este limite, começando por verificar se este sequer comprou o jogo em questão. A classe `ModelManager` contém o método `ownsGame`, o qual recebe como parâmetros o identificador de um jogador e o identificador de um jogo e retorna um valor booleano.

```
1 BigDecimal totalPoints = modelManager.getPlayerPoints(idJogo, idJogador);
```

Listing 13: Excerto da função `associarCrachaModel` para determinar se o jogador cumpre o limite de pontos

A próxima verificação determina se este jogador cumpre o limite, calculando o total de pontos nesse jogo com o método do `ModelManager`, `getPlayerPoints`, com o jogador como parâmetro.

Para evitar erros, verifica-se ainda se o jogador já tem o crachá em questão, através da função `ModelManager`, `ownsBadge`, a qual determina se um jogador já tem o crachá em questão. Finalmente, após todas as verificações necessárias, é criada uma nova entrada na tabela `CrachasAdquiridos`, através do modelo. Este código encontra-se na listagem 14.

```
1 CrachasAdquiridos crachaAdquirido = new CrachasAdquiridos();
2 CrachasAdquiridosId crachasAdquiridosId = modelManager.setCrachaAdquiridoId(idJogo, nomeCracha, idJogador);
3 crachaAdquirido.setId(crachasAdquiridosId);
4
5 Cracha cracha = em.find(Cracha.class, modelManager.setCrachaId(idJogo, nomeCracha));
6 crachaAdquirido.setCracha(cracha);
7
8 Jogador jogador = em.find(Jogador.class, idJogador); // Fetch the Jogador entity by ID
9 crachaAdquirido.setJogador(jogador);
```

Listing 14: Excerto da função `associarCrachaModel` que cria a nova entrada em `CrachasAdquiridos`

3.3 Funcionalidade Adicional

A segunda fase do trabalho prático vem acompanhada de uma funcionalidade adicional, a qual consiste em aumentar a quantidade de pontos necessários para obter um crachá por 20%. Esta vem no formate de uma função do modelo de negócio e começa por criar uma transação e colocá-la com nível de isolamento a `TRANSACTION_REPEATABLE_READ`. Este nível +e necessário para obter o cracha pretendido. O código desta query e transação encontra-se na listagem 15.

```
1 EntityTransaction transaction = transactionManager.startTransaction();
2 Connection cn = em.unwrap(Connection.class);
3 try {
4     transactionManager.setIsolationLevel(cn, Connection.TRANSACTION_READ_COMMITTED, transaction);
5     String selectQuery = "SELECT c FROM Cracha c WHERE c.id.nome = ?1 AND c.id.jogo = ?2";
6     TypedQuery<Cracha> selectTypedQuery = em.createQuery(selectQuery, Cracha.class);
7     selectTypedQuery.setParameter(1, nomeCracha);
8     selectTypedQuery.setParameter(2, idJogo);
9     selectTypedQuery.setLockMode(lockType);
10    Cracha cracha = selectTypedQuery.getSingleResult();
```

Listing 15: Excerto da função `aumentarPontosCracha20` para obter o crachá

Nota-se também que na linha 9 da listagem 15 encontra-se a chamada à função `setLockMode`. Aqui encontra-se a diferença entre *locking* otimista e pessimista.

O objetivo da próxima *query* da transação é alterar o valor do limite de pontos do crachá encontrado na porção anterior. A listagem 16 representa esta *query*.

```

1  transactionManager.setIsolationLevel(cn, Connection.TRANSACTION_REPEATABLE_READ, transaction);
2  String query =
3      "UPDATE Cracha c SET c.limite = c.limite * 1.2 WHERE c.id = :crachaId";
4  cracha.setVersion(cracha.getVersion() + 1);
5  Query updateQuery = em.createQuery(query);
6  if (lockType == LockModeType.PESSIMISTIC_READ) {
7      updateQuery.setLockMode(LockModeType.PESSIMISTIC_WRITE);
8  } else {
9      updateQuery.setLockMode(lockType);
10 }
11 updateQuery.setParameter("crachaId", cracha.getId());
12 updateQuery.setParameter("crachaVersion", cracha.getVersion());
13 int updatedCount = updateQuery.executeUpdate();

```

Listing 16: Excerto da função `aumentarPontosCracha20` - alterar o valor do limite do crachá. Na listagem 16 também se altera o tipo de lock para esta *query* com base no parâmetro do método `aumentarPontosCracha20`.

Após a execução da *query* verifica-se se a quantidade de linhas alteradas é 0. Neste caso verifica-se qual o tipo de lock e lança-se uma exceção correspondente, tal como se verifica na listagem 17.

```

1  if (updatedCount == 0) {
2      switch (lockType) {
3          case OPTIMISTIC -> throw new OptimisticLockException("Concurrent update detected for Cracha");
4          case PESSIMISTIC_READ -> throw new PessimisticLockException("Concurrent update detected for Cracha");
5          default -> throw new Exception("Concurrent update detected for Cracha");
6      }
7  }

```

Listing 17: Excerto da função `aumentarPontosCracha20` - verificar o sucesso da transação. No final, em caso de sucesso, a função simplesmente faz `commit`. Já em caso de falha, irá realizar `rollback`.

Devido à forma como esta função foi implementada, executar com *optimistic locking* basta a função `aumentarPontosOptimistic`, a qual passa como parâmetro `LockModeType.OPTIMISTIC`. Para obter a versão com *pessimistic locking* basta chamar a outra função, `aumentarPontosPessimistic`, que passa como parâmetro `LockModeType.PESSIMISTIC_READ`.

3.3.1 Testes da Funcionalidade

Para verificar o bom funcionamento desta nova funcionalidade, é necessário criar testes para a versão que faz uso de *optimistic locking*. Esta é vulnerável a erros de concorrência por não efetuar *lock* às tabelas afetadas antes de executar.

3.4 Aplicação

Para testar o bom funcionamento de cada função foi criada a classe **App**, a qual corre todas as funções e apresenta os resultados no *standard output*. Contém uma interface funcional **ITest** a qual apenas corre uma função sem retorno nem argumentos, assim é possível seleccionar o teste pretendido.

Capítulo 4

Conclusão

Enquanto que uma base de dados por si só apresenta muito potencial, acompanhada de uma aplicação torna-se mais compreensível e fácil de usar. A segunda fase do trabalho prático serviu para desenvolver esta aplicação e possibilitar o seu uso num ambiente de desenvolvimento Java.

Os objetivos foram todos cumpridos e permitem concluir que uma base de dados, ainda que apresente muito potencial, com o desenvolvimento de uma aplicação que a acompanhe demonstra o seu verdadeiro potencial. Afirma-se também que há diferentes formas de realizar certas operações, sendo que a escolha depende apenas de cada situação, não havendo uma solução que sirva para todos os casos.

4.1 Aspetos a melhorar

O modelo SQL desenvolvido na primeira fase do trabalho sofreu alterações devido à aplicação. Estas deveriam ter sido identificadas logo na fase 1. Um bom modelo SQL deve procurar ser fácil de entender e de utilizar. A alteração em questão foi a adição da restrição **unique** à coluna **email** da tabela de jogadores, desta forma é possível identificar um jogador através do seu *email*.

Referências

- [1] Thomas G. Lockhart Peter Eisentraut, Julien Rouhaud. Postgresql 15.2 documentation. [Online; Accessed 01-May-2023].

Anexo A

Código das Classes do *Package* `model.embeddables`

Neste anexo encontra-se a implementação das várias classes do *package* `Embeddables`

```
1 package model.embeddables;
2
3 import jakarta.persistence.Embeddable;
4 import java.io.Serializable;
5
6 // Composite key for table Compra
7 @Embeddable
8 public class CompraId implements Serializable {
9     private int jogador;
10    private String jogo;
11
12    public CompraId(){}
13    public void setJogoId(String jogo) {
14        this.jogo = jogo;
15    }
16
17    public String getJogoId() {
18        return jogo;
19    }
20
21    public void setJogadorId(int jogador) {
22        this.jogador = jogador;
23    }
24
25    public int getJogadorId() {
26        return jogador;
27    }
28 }
```

Listing 18: Código da classe `CompraId`

```
1 package model.embeddables;
2
3 import jakarta.persistence.Embeddable;
4
5 import java.io.Serializable;
6
7 // Composite jey for table cracha
8 @Embeddable
9 public class CrachaId implements Serializable {
10     private String nome;
11     private String jogo;
12
13     public CrachaId(){}
14     public String getNome() {
15         return nome;
16     }
17
18     public void setNome(String nome) {
19         this.nome = nome;
20     }
21
22     public String getJogo() {
23         return jogo;
24     }
25
26     public void setJogo(String jogo) {
27         this.jogo = jogo;
28     }
29
30 }
```

Listing 19: Código da classe CrachaId

```
1 package model.embeddables;
2
3 import jakarta.persistence.Embeddable;
4
5 import java.io.Serializable;
6
7 @Embeddable
8 public class CrachasAdquiridosId implements Serializable {
9     private int jogo;
10     private String jogador;
11     private String cracha;
12
13     public CrachasAdquiridosId(){}
14     public void setJogo(int jogo) {
15         this.jogo = jogo;
16     }
17 }
```

```

17
18     public int getJogo() {
19         return jogo;
20     }
21
22     public void setJogador(String jogador) {
23         this.jogador = jogador;
24     }
25
26     public String getJogador() {
27         return jogador;
28     }
29
30     public void setCracha(String cracha) {
31         this.cracha = cracha;
32     }
33
34     public String getCracha() {
35         return cracha;
36     }
37 }

```

Listing 20: Código da classe CrachasAdquiridosId

```

1  package model.embeddables;
2
3  import jakarta.persistence.Embeddable;
4  import jakarta.persistence.GeneratedValue;
5  import jakarta.persistence.GenerationType;
6
7  import java.io.Serializable;
8
9  //Composite key for table mensagem
10 @Embeddable
11 public class MensagemId implements Serializable {
12     @GeneratedValue(strategy = GenerationType.IDENTITY)
13     private int id;
14     private int conversa;
15
16     public MensagemId(){ }
17     public void setid(int id) {
18         this.id = id;
19     }
20
21     public int getid() {
22         return id;
23     }
24
25     public int getConversa() {
26         return conversa;

```



```

27     }
28     public void setConversa(int conversa) {
29         this.conversa = conversa;
30     }
31
32 }

```

Listing 21: Código da classe MensagemId

```

1  package model.embeddables;
2
3  import jakarta.persistence.Embeddable;
4
5  import java.io.Serializable;
6
7  //Composite key for jogador_conversa relation participa
8  @Embeddable
9  public class ParticipaId implements Serializable {
10     private int jogador;
11     private int conversa;
12
13     public ParticipaId(){}
14
15     public void setJogadorId(int jogador) {
16         this.jogador = jogador;
17     }
18
19     public int getJogadorId() {
20         return jogador;
21     }
22
23     public void setConversaId(int conversa) {
24         this.conversa = conversa;
25     }
26
27     public int getConversaId() {
28         return conversa;
29     }
30 }

```

Listing 22: Código da classe ParticipaId

```

1  package model.embeddables;
2
3  import jakarta.persistence.Embeddable;
4  import jakarta.persistence.GeneratedValue;
5  import jakarta.persistence.GenerationType;
6
7  import java.io.Serializable;

```

```

8
9 // Composite Primary key for Partida
10 @Embeddable
11 public class PartidaId implements Serializable {
12     @GeneratedValue(strategy = GenerationType.IDENTITY)
13     private int partida;
14     private String jogo;
15
16     public PartidaId(){}
17     public void setPartidaId(int partida) {
18         this.partida = partida;
19     }
20
21     public int getPartidaId() {
22         return partida;
23     }
24
25     public void setJogoId(String jogo) {
26         this.jogo = jogo;
27     }
28
29     public String getJogoId() {
30         return jogo;
31     }
32 }

```

Listing 23: Código da classe PartidaId

```

1 package model.embeddables;
2
3 import jakarta.persistence.Embeddable;
4
5 import java.io.Serializable;
6
7 // Composite Primary key for Partida_Normal and Partida_MultiJogador
8 @Embeddable
9 public class PartidaNMId implements Serializable {
10     int partida;
11     String jogo;
12     int jogador;
13
14     public PartidaNMId(){}
15
16     public void setPartida(int partidaId) {
17         this.partida = partidaId;
18     }
19
20     public int getPartida() {
21         return partida;
22     }

```

```
23
24     public String getJogo() {
25         return jogo;
26     }
27
28     public int getJogador() {
29         return jogador;
30     }
31
32     public void setJogador(int jogadorId) {
33         this.jogador = jogadorId;
34     }
35
36     public void setJogo(String jogoId) {
37         this.jogo = jogoId;
38     }
39 }
```

Listing 24: Código da classe PartidaNMId

Anexo B

Código das Classes do *Package* model.relations

Estas listagens contêm código correspondente às classes presentes no *package relations*

```
1 package model.relations;
2
3 import jakarta.persistence.*;
4 import model.tables.Jogador;
5 import model.tables.Jogo;
6 import model.embeddables.CompraId;
7
8 import java.io.Serial;
9 import java.io.Serializable;
10 import java.util.Date;
11
12 @Entity
13 @Table(name="compra")
14 public class Compra implements Serializable {
15
16     @Serial
17     private static final long serialVersionUID = 1L;
18
19     public Compra() { }
20
21     @EmbeddedId
22     private CompraId id;
23     private Date data;
24     private Double preco;
25
26
27     // N:N relation Compra between jogo and jogador
28     @ManyToOne
29     @MapsId("jogo")
30     @JoinColumn(name = "jogo")
31     private Jogo jogo;
32
33     @ManyToOne
```

```

34     @MapsId("jogador")
35     @JoinColumn(name = "jogador")
36     private Jogador jogador;
37
38     //getter and setters
39     public Jogador getJogador() { return jogador; }
40
41     public void setJogador(Jogador jogador) { this.jogador = jogador; }
42
43     public Jogo getJogo() { return jogo; }
44
45     public void setJogo(Jogo jogo) { this.jogo = jogo; }
46
47     public CompraId getId() { return this.id; }
48
49     public Date getData() { return data; }
50
51     public void setData(Date data) { this.data = data; }
52
53     public void setPreco(Double preco) { this.preco = preco; }
54     public Double getPreco() { return preco; }
55
56 }

```

Listing 25: Código da classe Compra

```

1  package model.relations;
2
3  import jakarta.persistence.*;
4  import model.tables.Cracha;
5  import model.tables.Jogador;
6  import model.embeddables.CrchasAdquiridosId;
7
8  import java.io.Serializable;
9
10 @Entity
11 @Table(name="crachá_jogador")
12 public class CrchasAdquiridos implements Serializable {
13
14     @EmbeddedId
15     private CrchasAdquiridosId id;
16
17
18     // N:N realtion CrchasAdquiridos
19     @ManyToOne
20     @MapsId("jogador")
21     @JoinColumn(name="jogador", referencedColumnName = "jogador")
22     private Jogador jogador;
23
24

```

```

25     @ManyToOne
26     @JoinColumns({
27         @JoinColumn(name = "cracha_nome", referencedColumnName = "nome"),
28         @JoinColumn(name = "cracha_jogador", referencedColumnName = "jogador")
29     })
30     private Cracha cracha;
31
32
33     public CrachasAdquiridosId getId() {
34         return id;
35     }
36
37     public void setId(CrachasAdquiridosId id) {
38         this.id = id;
39     }
40
41     public Cracha getCracha() {
42         return cracha;
43     }
44
45     public void setCracha(Cracha cracha) {
46         this.cracha = cracha;
47     }
48
49     public Jogador getJogador() {
50         return jogador;
51     }
52
53     public void setJogador(Jogador jogador) {
54         this.jogador = jogador;
55     }
56 }

```

Listing 26: Código da classe CrachasAdquiridos

```

1  package model.relations;
2
3  import jakarta.persistence.*;
4  import model.tables.Conversa;
5  import model.tables.Jogador;
6  import model.embeddables.ParticipaId;
7
8  import java.io.Serial;
9  import java.io.Serializable;
10
11  @Entity
12  @Table(name = "jogador_conversa")
13  public class Participa implements Serializable {
14
15      @Serial

```

```

16     private static final long serialVersionUID = 1L;
17
18     public Participa(){ }
19     @EmbeddedId
20     private ParticipaId id;
21
22     @ManyToOne
23     @MapsId("jogador")
24     @JoinColumn(name="jogador")
25     private Jogador jogador;
26
27     @ManyToOne
28     @MapsId("conversa")
29     @JoinColumn(name="conversa")
30     private Conversa conversa;
31
32
33     //getters and setters
34     public ParticipaId getId() {return id;}
35
36     public void setId(ParticipaId id) { this.id = id; }
37
38     public Jogador getJogador() { return jogador; }
39
40     public void setJogador(Jogador jogador) { this.jogador = jogador; }
41
42     public Conversa getConversa() { return conversa; }
43
44     public void setConversa(Conversa conversa) { this.conversa = conversa; }
45 }

```

Listing 27: Código da classe Participa

Anexo C

Código das Classes do *Package* model.tables

```
1 package model.tables;
2
3 import jakarta.persistence.*;
4 import model.relations.Compra;
5
6 import java.io.Serial;
7 import java.io.Serializable;
8 import java.util.ArrayList;
9 import java.util.List;
10
11 @Entity
12 @Table(name="jogo")
13 public class Jogo implements Serializable {
14
15     @Serial
16     private static final long serialVersionUID = 1L;
17
18     @Id
19     @GeneratedValue(strategy = GenerationType.UUID)
20     private String id;
21     private String nome;
22     private String url;
23
24     @OneToMany(mappedBy = "jogo")
25     private List<Compra> compras = new ArrayList<>();
26
27     public void addCompra(Compra compra){ this.compras.add(compra); }
28
29     public List<Compra> getCompras() { return compras; }
30
31
32     // 1:N pertence
33     @OneToMany(mappedBy="jogo",cascade=CascadeType.PERSIST, orphanRemoval=true)
34     private List<Cracha> crachas = new ArrayList<>();
35
```



```

36     public List<Cracha> getCrachas() {
37         return crachas;
38     }
39
40     public void addCracha(Cracha cracha) {
41         this.crachas.add(cracha);
42     }
43     @OneToMany(mappedBy = "jogo", cascade = CascadeType.PERSIST, orphanRemoval=true)
44     private List<Partida> partidas = new ArrayList<>();
45
46     public List<Partida> getPartidas() {
47         return partidas;
48     }
49
50     public void addPartida(Partida partida){
51         partidas.add(partida);
52     }
53
54     public String getId(){ return this.id; }
55     public String getNome(){ return this.nome; }
56     public void setNome(String nome) { this.nome = nome; }
57
58     public String getUrl(){ return this.url; }
59     public void setUrl(String url) { this.url = url; }
60
61 }

```

Listing 28: Código da classe Jogo

```

1  package model.tables;
2
3  import jakarta.persistence.*;
4  import model.relations.Compra;
5  import model.relations.CrachasAdquiridos;
6  import model.relations.Participa;
7
8  import java.io.Serial;
9  import java.io.Serializable;
10 import java.util.ArrayList;
11 import java.util.List;
12
13 @Entity
14 @Table(name="jogador")
15 public class Jogador implements Serializable {
16     @Serial
17     private static final long serialVersionUID = 1L;
18
19     public Jogador(){ }
20
21

```

```

22     @Id
23     @GeneratedValue(strategy = GenerationType.IDENTITY)
24     private int id;
25     private String email;
26     private String username;
27     private String estado;
28     private String regiao;
29     @OneToMany(mappedBy = "jogador")
30     private List<Compra> compras = new ArrayList<>();
31     public void addCompra(Compra compra){ this.compras.add(compra); }
32     public List<Compra> getCompras() { return compras; }
33
34
35     //N:N Crachas
36
37     @OneToMany(mappedBy = "jogador")
38     private List<CrachasAdquiridos> crachasAdquiridosList = new ArrayList<>();
39
40     public List<CrachasAdquiridos> getCrachasList() {
41         return crachasAdquiridosList;
42     }
43
44     public void adquirirCracha(CrachasAdquiridos crachasAdquiridos) {
45         this.crachasAdquiridosList.add(crachasAdquiridos);
46     }
47
48
49     //N:N Participa Relation
50     @OneToMany(mappedBy = "jogador")
51     private List<Participa> participaList = new ArrayList<>();
52     public List<Participa> getParticipaList() { return this.participaList; }
53     public void addParticipacao(Participa participa){this.participaList.add(participa);}
54
55     @OneToMany(mappedBy = "jogador", cascade = CascadeType.PERSIST)
56     private List<Partida_Normal> partidasNormais = new ArrayList<>();
57
58     public List<Partida_Normal> getPartidasNormais() {
59         return partidasNormais;
60     }
61
62     public void addPartida_Normal(Partida_Normal partida_normal){
63         this.partidasNormais.add(partida_normal);
64     }
65
66     @OneToMany(mappedBy = "jogador", cascade = CascadeType.PERSIST)
67     private List<Partida_MultiJogador> partidasMultiJogador = new ArrayList<>();
68
69     public void addPartidaMultiJogador(Partida_MultiJogador partida){
70         this.partidasMultiJogador.add(partida);
71     }
72     public List<Partida_MultiJogador> getPartidasMultiJogador() {

```

```

73         return partidasMultiJogador;
74     }
75
76
77     //getter and setters
78     public int getId(){
79         return this.id;
80     }
81
82     public String getEmail(){
83         return this.email;
84     }
85
86     public void setEmail(String email){
87         this.email = email;
88     }
89
90     public String getUsername(){
91         return this.username;
92     }
93
94     public void setUsername(String username){
95         this.username = username;
96     }
97
98     public String getEstado(){
99         return this.estado;
100     }
101
102     public void setEstado(String estado){
103         this.estado = estado;
104     }
105
106     public String getRegiao(){
107         return this.regiao;
108     }
109
110     public void setRegiao(String regiao){
111         this.regiao = regiao;
112     }
113
114 }
115

```

Listing 29: Código da classe Jogador

```

1 package model.tables;
2
3 import jakarta.persistence.*;
4 import model.embeddables.PartidaId;

```

```

5
6 import java.io.Serial;
7 import java.io.Serializable;
8 import java.util.Date;
9
10 @Entity
11 @Table(name="partida")
12 public class Partida implements Serializable {
13
14     @Serial
15     private static final long serialVersionUID = 1L;
16
17     public Partida() { }
18
19     @EmbeddedId
20     private PartidaId id;
21
22     String estado;
23     String nome_regiao;
24
25     Date dtinicio;
26
27     Date dtfim;
28
29     // 1:N
30     @ManyToOne
31     @MapsId
32     @JoinColumn(name = "id")
33     private Jogo jogo;
34
35     public void setJogo(Jogo jogo) {
36         this.jogo = jogo;
37     }
38
39     public Jogo getJogo() {
40         return jogo;
41     }
42
43     public PartidaId getId() {
44         return id;
45     }
46
47     @OneToOne(mappedBy = "partida")
48     private Partida_MultiJogador partida_multiJogador;
49
50     public Partida_MultiJogador getPartida_multiJogador() {
51         return partida_multiJogador;
52     }
53
54     public void setPartida_multiJogador(Partida_MultiJogador partida_multiJogador) {
55         this.partida_multiJogador = partida_multiJogador;
56     }
57 }

```

```

56     }
57
58     @OneToOne(mappedBy = "partida")
59     private Partida_Normal partida_normal;
60
61     public Partida_Normal getPartida_normal() {
62         return partida_normal;
63     }
64
65     public void setPartida_normal(Partida_Normal partida_normal) {
66         this.partida_normal = partida_normal;
67     }
68
69     public void setId(PartidaId id) {
70         this.id = id;
71     }
72
73     public String getEstado() {
74         return estado;
75     }
76
77     public void setEstado(String estado) {
78         this.estado = estado;
79     }
80
81     public String getNome_regiao() {
82         return nome_regiao;
83     }
84
85     public void setNome_regiao(String nome_regiao) {
86         this.nome_regiao = nome_regiao;
87     }
88
89     public Date getDtinicio() {
90         return dtinicio;
91     }
92
93     public void setDtinicio(Date dtinicio) {
94         this.dtinicio = dtinicio;
95     }
96
97     public void setDtfim(Date dtfim) {
98         this.dtfim = dtfim;
99     }
100
101     public Date getDtfim() {
102         return dtfim;
103     }
104 }

```

Listing 30: Código da classe Partida

```

1 package model.tables;
2
3 import jakarta.persistence.*;
4 import model.embeddables.PartidaMId;
5 import model.embeddables.PartidaNId;
6
7 import java.io.Serial;
8 import java.io.Serializable;
9
10 @Entity
11 @Table(name="partida_multiJogador")
12 public class Partida_MultiJogador implements Serializable {
13     @Serial
14     private static final long serialVersionUID = 1L;
15
16     public Partida_MultiJogador() { }
17
18     @EmbeddedId
19     private PartidaMId partida_multiId;
20
21     int pontuacao;
22
23     @OneToOne
24     @JoinColumns({
25         @JoinColumn(name = "partida", referencedColumnName = "partida", insertable = false, updatable = false),
26         @JoinColumn(name = "jogo", referencedColumnName = "jogo", insertable = false, updatable = false)
27     })
28     private Partida partida;
29
30
31     @ManyToOne
32     @MapsId
33     @JoinColumn(name = "jogador", referencedColumnName = "jogador")
34     private Jogador jogador;
35
36     public Jogador getJogador() {
37         return jogador;
38     }
39
40     public void setJogador(Jogador jogador) {
41         this.jogador = jogador;
42     }
43
44     public Partida getPartida() {
45         return partida;
46     }
47
48     public void setPontuacao(int pontuacao) {
49         this.pontuacao = pontuacao;
50     }

```

```

51
52     public void setPartida_multiId(PartidaMId partida) {
53         this.partida_multiId = partida;
54     }
55
56     public PartidaMId getPartida_multiId() {
57         return partida_multiId;
58     }
59
60     public int getPontuacao() {
61         return pontuacao;
62     }
63
64 }

```

Listing 31: Código da classe Partida_MultiJogador

```

1  package model.tables;
2
3  import jakarta.persistence.*;
4  import model.embeddables.PartidaNId;
5
6  import java.io.Serial;
7  import java.io.Serializable;
8
9  @Entity
10 @Table(name="partida_normal")
11 public class Partida_Normal extends Partida implements Serializable {
12
13     @Serial
14     private static final long serialVersionUID = 1L;
15
16     public Partida_Normal() { }
17
18     @EmbeddedId
19     private PartidaNId partida_normalId;
20
21     private int pontuacao;
22
23     @OneToOne
24     @JoinColumns({
25         @JoinColumn(name = "partida", referencedColumnName = "partida", insertable = false, updatable = false),
26         @JoinColumn(name = "jogo", referencedColumnName = "jogo", insertable = false, updatable = false)
27     })
28     private Partida partida;
29
30     public Partida getPartida() {
31         return partida;
32     }
33

```

```

34     @ManyToOne
35     @MapsId
36     @JoinColumn(name = "jogador", referencedColumnName = "jogador")
37     private Jogador jogador;
38
39     public Jogador getJogador() {
40         return jogador;
41     }
42
43     public void setJogador(Jogador jogador) {
44         this.jogador = jogador;
45     }
46
47     public PartidaId getPartida_normalId() {
48         return partida_normalId;
49     }
50
51     public void setPartida_normalId(PartidaId partida) {
52         this.partida_normalId = partida;
53     }
54
55     public int getPontuacao() {
56         return pontuacao;
57     }
58
59     public void setPontuacao(int pontuacao) {
60         this.pontuacao = pontuacao;
61     }
62
63
64 }

```

Listing 32: Código da classe Partida_Normal

```

1  package model.tables;
2
3
4  import jakarta.persistence.*;
5  import model.embeddables.CrachaId;
6  import model.relations.CrachasAdquiridos;
7
8  import java.io.Serial;
9  import java.io.Serializable;
10 import java.util.ArrayList;
11 import java.util.List;
12
13 @Table(name="cracha")
14 @Entity
15 public class Cracha implements Serializable {
16

```



```

17     @Serial
18     private static final long serialVersionUID = 1L;
19
20     public static long getSerialVersionUID() {
21         return serialVersionUID;
22     }
23
24     @EmbeddedId
25     private CrachaId id;
26
27     int limite;
28
29     String url;
30
31     @ManyToOne
32     @MapsId
33     @JoinColumn(name="jogo")
34     private Jogo jogo;
35
36     public Jogo getJogo() {
37         return jogo;
38     }
39
40     public void setJogo(Jogo jogo) {
41         this.jogo = jogo;
42     }
43
44     @OneToMany(mappedBy = "cracha")
45     private List<CrachasAdquiridos> crachasAdquiridosList = new ArrayList<>();
46
47     public List<CrachasAdquiridos> getCrachasList() {
48         return crachasAdquiridosList;
49     }
50
51     public void addCrachas(CrachasAdquiridos crachasAdquiridos) {
52         this.crachasAdquiridosList.add(crachasAdquiridos);
53     }
54
55
56     public CrachaId getId() {
57         return this.id;
58     }
59
60     public void setId(CrachaId id) {
61         this.id = id;
62     }
63
64     public int getLimite() {
65         return limite;
66     }
67

```

```

68     public void setLimite(int limit) {
69         this.limite = limit;
70     }
71
72     public String getUrl() {
73         return url;
74     }
75
76     public void setUrl(String url) {
77         this.url = url;
78     }
79
80 }

```

Listing 33: Código da classe Cracha

```

1  package model.tables;
2
3  import jakarta.persistence.*;
4  import model.relations.Participa;
5
6  import java.io.Serial;
7  import java.io.Serializable;
8  import java.util.ArrayList;
9  import java.util.List;
10
11  @Entity
12  @Table(name="conversa")
13  public class Conversa implements Serializable {
14
15      @Serial
16      private static final long serialVersionUID = 1L;
17
18      public Conversa() { }
19
20      @Id
21      @GeneratedValue(strategy = GenerationType.IDENTITY)
22      private int id;
23      private String nome;
24
25      // N:N relation participa
26      @OneToMany(mappedBy = "conversa")
27      private List<Participa> participantes = new ArrayList<>();
28
29      public List<Participa> getParticipantes(){ return this.participantes; }
30
31      public void addJogador(Participa participante){
32          this.participantes.add(participante);
33      }
34

```

```

35     // 1:N relation contem
36     @OneToMany(mappedBy = "conversa", cascade = CascadeType.PERSIST, orphanRemoval = true)
37     private List<Mensagem> mensagens = new ArrayList<>();
38
39     public List<Mensagem> getMensagens() {
40         return mensagens;
41     }
42
43     public void addMensagem(Mensagem mensagem) {
44         this.mensagens.add(mensagem);
45     }
46
47     //getter and setters
48     public int getId() { return id; }
49     public void setNome(String nome){ this.nome = nome; }
50     public String getNome(){ return this.nome; }
51 }

```

Listing 34: Código da classe Conversa

```

1  package model.tables;
2
3  import jakarta.persistence.*;
4  import model.embeddables.MensagemId;
5
6  import java.io.Serial;
7  import java.io.Serializable;
8  import java.sql.Date;
9
10 @Entity
11 @Table(name = "mensagem")
12 public class Mensagem implements Serializable {
13
14     @Serial
15     private static final long serialVersionUID = 1L;
16     public Mensagem() { }
17
18     @EmbeddedId
19     private MensagemId mensagemId;
20
21     String conteudo;
22
23     Date data;
24
25
26     @ManyToOne
27     @MapsId
28     @JoinColumn(name="conversa")
29     private Conversa conversa;
30

```

```
31     public MensagemId getMensagemId() {
32         return mensagemId;
33     }
34
35     public void setMensagemId(MensagemId mensagemId) {
36         this.mensagemId = mensagemId;
37     }
38
39     public Date getData() {
40         return data;
41     }
42
43     public void setData(Date data) {
44         this.data = data;
45     }
46
47     public String getConteudo() {
48         return conteudo;
49     }
50
51     public void setConteudo(String conteudo) {
52         this.conteudo = conteudo;
53     }
54 }
```

Listing 35: Código da classe Mensagem

Anexo D

Código das Classes do *Package* businessLogic

```
1  package businessLogic;
2
3  import java.math.BigDecimal;
4  import java.sql.CallableStatement;
5  import java.sql.Connection;
6  import java.sql.Types;
7  import java.util.List;
8  import java.util.Map;
9
10
11 import businessLogic.BLserviceUtils.ModelManager;
12 import businessLogic.BLserviceUtils.TransactionManager;
13 import jakarta.persistence.*;
14 import model.embeddables.CrachasAdquiridosId;
15 import model.relations.CrachasAdquiridos;
16 import model.tables.Cracha;
17 import model.tables.Jogador;
18 import model.views.JogadorTotalInfo;
19
20 /**
21  * Hello world!
22  *
23  */
24 public class BLService
25 {
26     EntityManagerFactory emf = Persistence.createEntityManagerFactory("JPAex");
27     EntityManager em = emf.createEntityManager();
28
29     private final ModelManager modelManager = new ModelManager(em);
30     private final TransactionManager transactionManager = new TransactionManager(em);
31
32     /**
33      * 1. (a)
34      * Access to the functionalities 2d to 2l
35      */
```

```

36 // These 2 functions constitute the exercise 2d, these do not require a transaction level above
37 // read uncommitted since they only preform an update to the respective tables once
38 public String createPlayer(String username, String email, String regiao) {
39     String query = "SELECT createPlayer(?1, ?2, ?3)";
40     Query functionQuery = em.createNativeQuery(query)
41         .setParameter(1, username)
42         .setParameter(2, email)
43         .setParameter(3, regiao);
44     return (String) functionQuery.getSingleResult();
45
46 }
47
48 public String setPlayerState(String playerName, String newState) {
49     int idJogador = modelManager.getPlayerByEmail(playerName).getId();
50     String query = "SELECT setPlayerState(?1, ?2)";
51     Query functionQuery = em.createNativeQuery(query)
52         .setParameter(1, idJogador)
53         .setParameter(2, newState);
54     return (String) functionQuery.getSingleResult();
55 }
56
57 // Exercise 2e
58 public Long totalPontosJogador(String email) {
59     int idJogador = modelManager.getPlayerByEmail(email).getId();
60     String query = "SELECT totalPontos from totalPontosJogador(?1)";
61     Query functionQuery = em.createNativeQuery(query);
62     functionQuery.setParameter(1, idJogador);
63     return (Long) functionQuery.getSingleResult();
64 }
65
66 // Exercise 2f
67 public Long totalJogosJogador(String email) {
68     int idJogador = modelManager.getPlayerByEmail(email).getId();
69     String query = "SELECT totalJogos from totalJogosJogador(?1)";
70     Query functionQuery = em.createNativeQuery(query);
71     functionQuery.setParameter(1, idJogador);
72     return (Long) functionQuery.getSingleResult();
73 }
74
75 // Exercise 2g (Temporary implementation, not optimized)
76
77 public Map<Integer, BigDecimal> PontosJogosPorJogador(String gameName) {
78     String idJogo = modelManager.getGameByName(gameName).getId();
79     String queryString = "SELECT jogadores, pontuacaoTotal from PontosJogosPorJogador(?1)";
80     Query query = em.createNativeQuery(queryString);
81     query.setParameter(1, idJogo);
82     Map<Integer, BigDecimal> map = new java.util.HashMap<>(Map.of());
83     List<Object[]> list = query.getResultList();
84     for (Object[] obj : list) {
85         Integer idJogador = (Integer) obj[0];
86         BigDecimal points = (BigDecimal) obj[1];

```

```

87         map.put(idJogador, points);
88     }
89     return map;
90 }
91
92
93 // Exercise 2h
94 public void associarCracha(int idJogador, String gameName, String nomeCracha) {
95     EntityTransaction transaction = transactionManager.startTransaction();
96     Connection cn = em.unwrap(Connection.class);
97     String idJogo = modelManager.getGameByName(gameName).getNome();
98     try {
99         transactionManager.setIsolationLevel(cn, Connection.TRANSACTION_REPEATABLE_READ, transaction);
100        try (CallableStatement storedProcedure = cn.prepareCall("call associarCracha(?, ?, ?)")) {
101            storedProcedure.setInt(1, idJogador);
102            storedProcedure.setString(2, idJogo);
103            storedProcedure.setString(3, nomeCracha);
104            storedProcedure.executeUpdate();
105            transaction.commit();
106        }
107    } catch (Exception e) {
108        if (transaction.isActive()) transaction.rollback();
109    }
110 }
111
112 // Exercise 2i
113 public Integer iniciarConversa(int idJogador, String nomeConversa) {
114     EntityTransaction transaction = transactionManager.startTransaction();
115     Connection cn = em.unwrap(Connection.class);
116     Integer idConversa = null;
117     try {
118         transactionManager.setIsolationLevel(cn, Connection.TRANSACTION_REPEATABLE_READ, transaction);
119        try (CallableStatement storedProcedure = cn.prepareCall("call iniciarConversa(?, ?, ?)")) {
120            storedProcedure.setInt(1, idJogador);
121            storedProcedure.setString(2, nomeConversa);
122            storedProcedure.registerOutParameter(3, Types.INTEGER);
123            storedProcedure.executeUpdate();
124            idConversa = storedProcedure.getInt(3);
125            transaction.commit();
126        }
127    } catch (Exception e) {
128        if (transaction.isActive()) transaction.rollback();
129        return null;
130    }
131    return idConversa;
132 }
133
134 // Exercise 2j
135 public void juntarConversa(int idJogador, int idConversa) {
136     EntityTransaction transaction = transactionManager.startTransaction();
137     Connection cn = em.unwrap(Connection.class);

```

```

138     try {
139         try (CallableStatement storedProcedure = cn.prepareCall("call juntarConversa(?,?)")) {
140             storedProcedure.setInt(1, idJogador);
141             storedProcedure.setInt(2, idConversa);
142             transaction.commit();
143         }
144     } catch (Exception e) {
145         if (transaction.isActive()) transaction.rollback();
146     }
147 }
148
149 // Exercise 2k
150 public void enviarMensagem(int idJogador, int idConversa, String content) {
151
152     EntityTransaction transaction = transactionManager.startTransaction();
153     Connection cn = em.unwrap(Connection.class);
154     try {
155         transactionManager.setIsolationLevel(cn, Connection.TRANSACTION_READ_COMMITTED, transaction);
156         try (CallableStatement storedProcedure = cn.prepareCall("call enviarMensagem(?,?, ?)")) {
157             cn.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
158             storedProcedure.setInt(1, idJogador);
159             storedProcedure.setInt(2, idConversa);
160             storedProcedure.setString(3, content);
161             storedProcedure.executeUpdate();
162             transaction.commit();
163         }
164     } catch (Exception e) {
165         if (transaction.isActive()) transaction.rollback();
166     }
167 }
168
169 public void jogadorTotalInfo(){
170     for(JogadorTotalInfo jogador: modelManager.getPlayerTotalInfo()){
171         System.out.println(
172             jogador.getEstado() + " " +
173             jogador.getEmail() + " " +
174             jogador.getUsername() + " " +
175             jogador.getJogosParticipados() + " " +
176             jogador.getPartidasParticipadas() + " " +
177             jogador.getPontuacaoTotal() + " "
178         );
179     }
180 }
181
182
183 public void associarCrachaModel(int idJogador, String gameName, String nomeCracha){
184     String idJogo = modelManager.getGameByName(gameName).getId();
185     TypedQuery<String> crachaExistsQuery =
186         em.createQuery(
187             "select c.id.jogo from Cracha c where c.id.jogo = :idJogo and c.id.nome = :nomeCracha"
188             String.class

```



```

189         );
190
191         crachaExistsQuery.setParameter("idJogo", idJogo);
192         crachaExistsQuery.setParameter("nomeCracha", nomeCracha);
193         String foundIdJogo = crachaExistsQuery.getSingleResult();
194
195         if(foundIdJogo.equals(idJogo)){
196             TypedQuery<Integer> getLimitPoints = em.createQuery(
197                 "select c.limite from Cracha c where c.id.nome = :nomeCracha",
198                 Integer.class
199             );
200             getLimitPoints.setParameter("nomeCracha", nomeCracha);
201             Integer limit = getLimitPoints.getSingleResult();
202
203             if(modelManager.ownsGame(idJogador, idJogo)){
204                 BigDecimal totalPoints = modelManager.getPlayerPoints(idJogo, idJogador);
205                 if(limit <= totalPoints.intValue() ){
206
207                     if(!modelManager.ownsBadge(idJogador, idJogo)){
208                         CrachasAdquiridos crachaAdquirido = new CrachasAdquiridos();
209                         CrachasAdquiridosId crachasAdquiridosId = modelManager.setCrachaAdquiridoId(idJogo, nomeCracha, crachaAdquirido);
210                         crachaAdquirido.setId(crachasAdquiridosId);
211
212                         Cracha cracha = em.find(Cracha.class, modelManager.setCrachaId(idJogo, nomeCracha));
213                         crachaAdquirido.setCracha(cracha);
214
215                         Jogador jogador = em.find(Jogador.class, idJogador); // Fetch the Jogador entity by ID
216                         crachaAdquirido.setJogador(jogador);
217                     }
218                 }
219             }
220
221         }
222     }
223 }
224 }
225
226 private void aumentarPontosCracha20(String nomeCracha, String idJogo, LockModeType lockType) {
227     EntityTransaction transaction = transactionManager.startTransaction();
228     Connection cn = em.unwrap(Connection.class);
229     try {
230         transactionManager.setIsolationLevel(cn, Connection.TRANSACTION_READ_COMMITTED, transaction);
231         String selectQuery = "SELECT c FROM Cracha c WHERE c.id.nome = ?1 AND c.id.jogo = ?2";
232         TypedQuery<Cracha> selectTypedQuery = em.createQuery(selectQuery, Cracha.class);
233         selectTypedQuery.setParameter(1, nomeCracha);
234         selectTypedQuery.setParameter(2, idJogo);
235         selectTypedQuery.setLockMode(lockType);
236         Cracha cracha = selectTypedQuery.getSingleResult();
237         transactionManager.setIsolationLevel(cn, Connection.TRANSACTION_REPEATABLE_READ, transaction);
238         String query =
239             "UPDATE Cracha c SET c.limite = c.limite * 1.2 WHERE c.id = :crachaId";

```

```

240         cracha.setVersion(cracha.getVersion() + 1);
241         Query updateQuery = em.createQuery(query);
242         if (lockType == LockModeType.PESSIMISTIC_READ) {
243             updateQuery.setLockMode(LockModeType.PESSIMISTIC_WRITE);
244         } else {
245             updateQuery.setLockMode(lockType);
246         }
247         updateQuery.setParameter("crachaId", cracha.getId());
248         updateQuery.setParameter("crachaVersion", cracha.getVersion());
249         int updatedCount = updateQuery.executeUpdate();
250         if (updatedCount == 0) {
251             switch (lockType) {
252                 case OPTIMISTIC -> throw new OptimisticLockException("Concurrent update detected for Cracha");
253                 case PESSIMISTIC_READ -> throw new PessimisticLockException("Concurrent update detected for Cracha");
254                 default -> throw new Exception("Concurrent update detected for Cracha");
255             }
256         }
257         transaction.commit();
258     } catch (Exception e) {
259         if (transaction.isActive()) transaction.rollback();
260         System.out.println("Error Message: " + e.getMessage());
261     }
262 }
263
264 /**
265  * 2.(a), 2.(b), 2.(c)
266  */
267
268 // Exercise 2 (a)
269 public void aumentarPontosOptimistic(String nomeCracha, String idJogo) {
270     aumentarPontosCracha20(nomeCracha, idJogo, LockModeType.OPTIMISTIC);
271 }
272
273 // Exercise 2 (b) -> Inside test/java/ConcurrencyErrorTest.java
274
275 // Exercise 2 (c)
276 public void aumentarPontosPessimistic(String nomeCracha, String idJogo) {
277     aumentarPontosCracha20(nomeCracha, idJogo, LockModeType.PESSIMISTIC_READ);
278 }
279 }

```

Listing 36: Código da classe BLService

Anexo E

Código das Classes do *Package* businessLogic.BLServiceUtils

```
1 package businessLogic.BLserviceUtils;
2
3 import jakarta.persistence.EntityManager;
4 import jakarta.persistence.EntityTransaction;
5 import jakarta.persistence.Query;
6 import jakarta.persistence.TypedQuery;
7 import model.embeddables.CrachaId;
8 import model.embeddables.CrachasAdquiridosId;
9 import model.relations.Compra;
10 import model.tables.Jogador;
11 import model.tables.Jogo;
12
13 import java.math.BigDecimal;
14 import java.sql.Connection;
15 import java.sql.SQLException;
16
17 public class ModelManager {
18     public ModelManager(EntityManager em){
19         this.em = em;
20     }
21
22     private final EntityManager em;
23
24
25     public Jogo getGameByName(String gameName, EntityManager em){
26         TypedQuery<Jogo> query = em.createQuery("SELECT j FROM Jogo j WHERE j.nome = ?1", Jogo.class);
27         query.setParameter(1, gameName);
28         return query.getSingleResult();
29     }
30
31     public Jogador getPlayerByEmail(String email, EntityManager em){
32         TypedQuery<Jogador> query = em.createQuery("SELECT j FROM Jogador j WHERE j.email = ?1", Jogador.class);
33         query.setParameter(1, email);
34         return query.getSingleResult();
35     }
36 }
```

```

36
37     public CrachasAdquiridosId setCrachaAdquiridoId(String idJogo, String nomeCracha, int idJogador){
38         CrachasAdquiridosId crachasAdquiridosId = new CrachasAdquiridosId();
39         crachasAdquiridosId.setJogo(idJogo);
40         crachasAdquiridosId.setCracha(nomeCracha);
41         crachasAdquiridosId.setJogador(idJogador);
42         return crachasAdquiridosId;
43     }
44
45     public CrachaId setCrachaId(String idJogo, String nomeCracha){
46         CrachaId crachaId = new CrachaId();
47         crachaId.setJogo(idJogo);
48         crachaId.setNome(nomeCracha);
49         return crachaId;
50     }
51
52     public BigDecimal getPlayerPoints(String idJogo, int idJogador){
53         String jogadoresQuery = "SELECT pontuacaoTotal from PontosJogosPorJogador(?) WHERE jogadores = ?2";
54         Query pontuacaoQuery = em.createNativeQuery(jogadoresQuery);
55         pontuacaoQuery.setParameter(1, idJogo);
56         pontuacaoQuery.setParameter(2, idJogador);
57         return (BigDecimal) pontuacaoQuery.getSingleResult();
58     }
59
60     public Boolean ownsGame(int idJogador, String idJogo){
61         TypedQuery<Compra> getCompra = em.createQuery(
62             "select c from Compra c " +
63             "where c.jogador.id = :idJogador and c.jogo.id = :idJogo",
64             Compra.class
65         );
66         getCompra.setParameter("idJogador", idJogador);
67         getCompra.setParameter("idJogo", idJogo);
68         return getCompra.getResultList().isEmpty();
69     }
70
71     public Boolean ownsBadge(int idJogador){
72         Query hasCracha = em.createQuery("SELECT ca.id.jogo from CrachasAdquiridos ca WHERE ca.id.jogador = ?1");
73         hasCracha.setParameter(1, idJogador);
74         return !hasCracha.getResultList().isEmpty();
75     }
76 }

```

Listing 37: Código da classe ModelManager

```

1 package businessLogic.BLserviceUtils;
2
3 import jakarta.persistence.EntityManager;
4 import jakarta.persistence.EntityTransaction;
5
6 import java.sql.Connection;

```

```
7  import java.sql.SQLException;
8
9  public class TransactionManager {
10
11      public TransactionManager(EntityManager em){
12          this.em = em;
13      }
14
15      private EntityManager em;
16      public EntityTransaction startTransaction(){
17          EntityTransaction transaction = em.getTransaction();
18          transaction.begin();
19          return transaction;
20      }
21
22      public void setIsolationLevel(Connection cn, Integer isolationLevel, EntityTransaction transaction) throws
23          transaction.rollback();
24          cn.setTransactionIsolation(isolationLevel);
25          transaction.begin();
26      }
27 }
```

Listing 38: Código da classe TransactionManager

Anexo F

Código das Classe App

```
1  /*
2   Walter Vieira (2022-04-22)
3   Sistemas de Informação - projeto JPAAulas_ex3
4   Código desenvolvido para iulustração dos conceitos sobre acesso a dados, concretizados com base na especifica
5   Todos os exemplos foram desenvolvidos com EclipseLink (3.1.0-M1), usando o ambiente Eclipse IDE versão 2022
6
7   Não existe a pretensão de que o código esteja completo.
8
9   Embora tenha sido colocado um esforço significativo na correção do código, não há garantias de que ele não con
10  acarretar problemas vários, em particular, no que respeita à consistência dos dados.
11
12  */
13
14  package presentation;
15
16  import java.util.Scanner;
17
18
19  import businessLogic.*;
20
21
22  /**
23   * Hello world!
24   *
25   */
26
27  public class App {
28      protected interface ITest {
29          void test();
30      }
31
32      public static void main( String[] args ) {
33          BLService srv = new BLService();
34          Scanner imp = new Scanner(System.in);
35          ITest[] tests = new ITest[] {
36              () -> {
37                  System.out.println("\n\n\n ----- CREATE PLAYER -----")
```

```

38         System.out.println("Introduza um username");
39         String playerName = imp.nextLine();
40         System.out.println("Introduza um email");
41         String email = imp.nextLine();
42         try { System.out.println( srv.createPlayer(playerName, email, "EU") ); }
43         catch(Exception e) { System.out.println(e); }
44     },
45     () -> {
46         System.out.println("\n\n\n ----- BAN PLAYER -----");
47         System.out.println("Introduza um email de um jogador que queira banir");
48         String email = imp.nextLine();
49         try { System.out.println(srv.setPlayerState(email, "Banido")); }
50         catch(Exception e) { System.out.println(e); }
51     },
52     () -> {
53         System.out.println("\n\n\n ----- PONTOS POR JOGADOR -----");
54         System.out.println("Introduza um email de um jogador que queira saber");
55         String email = imp.nextLine();
56         try { System.out.println(srv.totalPontosJogador(email)); }
57         catch(Exception e) { System.out.println(e); }
58     },
59     () -> {
60         System.out.println("\n\n\n ----- TOTAL JOGOS JOGADOR -----");
61         System.out.println("Introduza um email de um jogador que queira saber");
62         String email = imp.nextLine();
63         try { System.out.println(srv.totalJogosJogador(email)); }
64         catch(Exception e) { System.out.println(e); }
65     },
66     () -> {
67         System.out.println("\n\n\n ----- PONTOS JOGOS POR JOGADOR -----");
68         try { System.out.println(srv.PontosJogosPorJogador("Valorant")); }
69         catch (Exception e){ System.out.println(e);}
70     },
71     () -> {
72         System.out.println("\n\n\n ----- ASSOCIAR CRACHA -----");
73         try { srv.associarCracha(2, "Genshin Impact", "Master"); }
74         catch (Exception e){ System.out.println(e); }
75     },
76     () -> {
77         System.out.println("\n\n\n ----- INICIAR CONVERSA -----");
78         try { srv.iniciarConversa(1, "newConvo"); }
79         catch (Exception e){ System.out.println(e); }
80     },
81     () -> {
82         System.out.println("\n\n\n ----- JUNTAR CONVERSA -----");
83         try { srv.juntarConversa(2, 2); }
84         catch (Exception e){ System.out.println(e); }
85     },
86     () -> {
87         System.out.println("\n\n\n ----- ENVIAR MENSAGEM -----");
88         try { srv.enviarMensagem(2, 1, "Hello"); }

```

```

89         catch (Exception e){ System.out.println(e); }
90     },
91     () -> {
92         System.out.println("\n\n\n ----- JOGADOR TOTAL INFO -----");
93         try { srv.jogadorTotalInfo(); }
94         catch (Exception e) { System.out.println(e); }
95     },
96     () -> {
97         System.out.println("\n\n\n ----- ASSOCIAR CRACHA MODEL -----");
98         try { srv.associarCrachaModel(1, "Valorant", "Pro player");}
99         catch (Exception e){ System.out.println(e); }
100     },
101 };
102
103 while(true){
104     Scanner opt = new Scanner(System.in);
105     System.out.printf("Choose a test (1-%d)? -1 to break ",tests.length);
106     int option = opt.nextInt();
107     if (option >= 1 && option <= tests.length)
108         tests[--option].test();
109     if(option == -1) break;
110 }
111 }
112 }

```

Listing 39: Código da classe ModelManager