ISEL
ADEETC
Área Departamental de Engenharia Electrónica e de Telecomunicações e de Computadores

Licenciatura em Engenharia Informática e de Computadores

Sistemas de Informação

Trabalho Prático
Fase 2

Autores:

Diogo Alexandre Ferreira de Jesus 48302
Miguel José Reys e Sousa Marmelete 48260
Henrique Valente Mareco Ferreira Águas 48929

Grupo 6 - Turma 41D

06/06/2022

Verão 2021 / 2022

Resumo

Para a segunda fase do trabalho no âmbito da disciplina de Sistemas de Informação tivemos que realizar uma aplicação em Java utilizando *JPA* (*Java Persistence API*) para interagir com as funcionalidades implementadas na fase anterior. Funcionalidades estas que incluem funções SQL, triggers, vistas e procedimentos armazenados em *PosgreSQL*.

Abstract

For the second phase of the assignment regarding Information Systems subject we had to build a Java application using *JPA* (*Java Persistence API*) to interact with functionalities implemented in the previous phase. Such functionalities include SQL functions, triggers, views and stored procedures in *PostgreSQL*.

Table of Contents

Lista de Figuras

1 Objetivos do Trabalho e Enunciado

Os objectivos pretendidos com a realização deste trabalho foram os seguintes:

- Desenvolver uma camada de acesso a dados, que use uma implementação de JPA e um subconjunto dos padrões de desenho DataMapper, Repository e UnitOfWork;
- Desenvolver uma aplicação em Java, que use adequadamente a camada de acesso a dados;
- Utilizar corretamente processamento transacional, através de mecanismos disponíveis no JPA;
- Garantir a correta libertação de ligações e recursos, quando estes não estejam a ser utilizados;
- Garantir a correta implementação das restrições de integridade e/ou lógica de negócio;

O Enunciado da segunda fase do trabalho foi o seguinte:

1.
   a) Disponibilizar, através da aplicação Java, acesso às funcionalidades realizadas na fase anterior;
   b) Realizar a funcionalidade 2.h) da fase anterior (criação de um veículo + zona verde [opcional])) sem recorrer a procedimentos armazenados;
   c) Realizar a funcionalidade 2.h) da fase anterior (criação de um veículo + zona verde [opcional])) reutilizando procedimentos armazenados criados na fase anterior.

   Nota: Na alínea a) do exercício 1 reutilizámos procedimentos armazenados e funções realizadas na fase anterior sempre que possível. Desta forma a alínea c) já se encontra incluída na alínea a).

2.
   a) Reimplementar a funcionalidade 2.f) da fase anterior (tratamento de registos inválidos) usando Optimistic Locking;
   b) Apresentar um teste relativo à alínea anterior (2.a)) em que mostramos o comportamento do Optimistic Locking. A exceção provocada por este mecanismo em certas situações deve também ser comunicado ao utilizador da aplicação através de uma mensagem de erro.

2 Organização do Projeto

A organização do projeto Java é a seguinte:

- BusinessLogic
  - Handlers (Contêm os comandos/funcionalidades disponíbilizados pela aplicação separados por alínea)
- DataScope (Disponibiliza operações CRUD ao nível dos dados da base de dados sobre todas as entidades)
- Model
  - Entities (Contêm todas as entidades sob a forma de classe/objeto que facilitando a sua manipulação)
- Presentation
  - UI (Expõe as funcionalidades da aplicação através da linha de comandos)
- Utils (Possui funções utilitárias de forma a evitar repetição no código e abstração)

3 Mapeamento de Entidades

Para mais facilmente manipular dados das tabelas da base de dados em PostgreSQL em Java realizámos mapeamento de entidades (**ORM** – Object-Relational Mapping) através do **JPA**.

Isto é conseguido através de diversas anotações colocadas sobre propriedades e classes.

Para definir o aspecto/assinatura de uma tabela com JPA usámos as anotações **Entity** e **Table**. A classe implementa a interface *Serializable.*

As propriedades são *private* e acessíveis através de *Getters* e *Setters*.

As anotações que utilizámos para as propriedades de cada classe foram:

- **Id** sobre a chave primária;
- **GeneratedValue** sobre sequências (propriedades com o tipo serial em PGSQL);
- **Column** para dar mais informações acerca daquela coluna (Ex: se é admite o valor **NULL**);
- **JoinColumn** juntamente com **ManyToOne**, **OneToMany**, **OneToOne** e **ManyToMany** para caracterizar as relações entre tabelas;

Utilizámos ainda *Lazy Fetching* nas chaves estrangeiras para evitar carregar entidades referenciadas por uma Entidade até que estas sejam requisitadas. Isto foi conseguido através da propriedade *fetch* nas anotações de relações entre tabelas
(Ex: @ManyToOne(fetch = FetchType.LAZY)).

4 Módulo de Acesso a Dados

Para o acesso aos dados utilizámos uma técnica sugerida pelo docente, um Data Scope que é responsável por disponibilizar um conjunto de operações **CRUD** (Create, Read, Update, Delete) sobre as entidades da aplicação.

Tais operações são:

- getAll()
- getSingle(K pk)
- getSingle()
- get(Map queryMap)
- getNative(Map queryMap)
- delete(T item)
- deleteById(K pk)
- update(T item)
- create(T item)

Fizémos este Data Scope de forma genérica para evitar repetição de código. Os parâmetros de tipo são a **Entidade (*T*)** e o **Tipo da Chave Primária (*K*)** da Entidade como é possível observar na figura seguinte.

```
public class DataScope<T extends JPAEntity<K>, K> implements AutoCloseable
```

Figura 1. Assinatura de Data Scope

Para possibilitar ainda a extração da chave primária a partir de uma entidade *T* fizémos todas as Entidades implementarem uma interface á qual chamámos *JPAEntity.*

```
public interface JPAEntity<K> {
    K getPK();
}
```

Figura 2. Interface JPA Entity

Esta interface tem um único método *getPK.* Todas as entidades implementam esta interface.

```
@Entity
@Table(
    name = "gps"
)
public class Gps implements Serializable, JPAEntity<Integer> {
  @Id
  @Column(
      nullable = false
  )
  private int id;

  public Gps() {
  }

  public int getId() { return this.id; }

  public void setId(int id) { this.id = id; }

  @Override
  public Integer getPK() { return getId(); }
}
```

Figura 3. Exemplo de uma entidade a implementar JPAEntity

4.1 Gestão Transacional

Para a gestão de transações utilizámos o **EntityManager** que possui uma transação interna sobre a qual é possível chamar os métodos *begin* (para iniciar uma transação), *commit* (para tornar permanentes os *Statements* executados até então sobre a transação do **EntityManager**), e outros como o *flush, rollback, persist, find, createQuery*.

Utilizámos uma técnica apresentada pelo professor para reutilizar transações entre várias instâncias de Data Scope de forma hierárquica.

Desta forma, através da utilização da funcionalidade *Try With Resources* do Java a realização de um conjunto de operações de forma atómica realiza-se da seguinte forma:

- Abrir vários data scopes (1 por cada tabela/entidade que desejamos manipular)
- Realizar as ações/*statements* sobre estes Data Scopes
- "votar" todos os *DataScopes*. Caso uma sub-transação não vote todas as sub-transações e transações superiores são desfeitas. Assume-se que uma sub-transacção apenas não vota quando ocorre exceção.
- Fechar as transações em cascata (*Try With Resources* automáticamente faz isto através da chamada aos métodos *close* de todos os Data Scopes abertos, uma vez que DataScope implementa AutoCloseable)

A imagem seguinte mostra um exemplo simples de utilização de um Data Scope através do mecanismo *Try With Resources*.

```
try (
        DataScope<EstadosGps, String> ds_estado_gps = new DataScope<>(EstadosGps.class);
) {
    EstadosGps newEstadoGps = new EstadosGps();
    newEstadoGps.setEstado(estado.value.toString());
    ds_estado_gps.create(newEstadoGps);
    // Vote
    ds_estado_gps.validateWork();
}
```

Figura 4. Exemplo de utilização de um DataScope por forma a realizar ações sobre a base de dados

5 Funções Auxiliares

Por forma a tornar a aplicação mais legível e expansível fomos criando funções utilitárias:

- Função que nos permite a chamada a procedimentos armazenados, funções e vistas existentes na base de dados. Para funções que retornam tabelas ou vistas esta função retorna uma lista com os itens das mesmas. Para poder retornar os itens da vista realizada em 2.i) da fase anterior criámos ainda uma entidade a representar as colunas da vista.

- Temos também uma função para desenhar uma tabela na linha de comandos.

- Funções que recolhem da base de dados um conjunto limitado de registos de certas tabelas que podem ser úteis para sugerir a quem está a tentar inserir um Cliente Particular um referenciador. Desta forma quem está a inserir o cliente não tem que saber ao certo o Nif do referenciador. No entanto, como é óbvio, nada garante que o referenciador não é apagado até ao momento da criação (com *commit*) do Cliente Particular.

6 Execução de um Comando

Quando o utilizador escolhe uma opção do menu de operações é chamada a função *run* relativa ao comando escolhido. Esta função vai pedir ao utilizador todos os parâmetros (alguns podem ser opcionais) para depois realizar as ações requiridas para concretizar a operação selecionada.

O fluxo de execução é o seguinte:

1) Definição dos parâmetros que queremos que o utilizador forneça. Podemos indicar se o parâmetro é opcional e se queremos mostrar opções de escolha quando formos pedir um valor ao utilizador. Com estes parâmetros está incluido também um validador, que efetua uma validação para garantir que o valor recebido é válido de acordo com algumas das restrições da base de dados;

2) Passar estes parâmetros a uma função que vai recolher os seus valores perguntando ao utilizador. Efetua a validação referida no ponto 1) sobre todos os valores recebidos.

3) Com os valores para os parâmetros obtidos podemos agora chamar um procedimento armazenado, função ou vista.

4) Caso a funcionalidade que estamos a tentar realizar não esteja já implementada ao nível da base de dados utilizamos Data Scopes para manipular entidades e realizar as ações necessárias.

A figura seguinte ilustra o exemplo da execução de um comando que utiliza Data Scopes para se concretizar.

```java
public class InserirSobreVistaAlarmes {

    public static void run() throws Exception {
        Parameter matricula = EntityParameters.MATRICULA( optional: false,  showOptions: true);
        Parameter nomeCondutor = EntityParameters.NOMECONDUTOR( optional: false);
        Parameter latitude = EntityParameters.LATITUDE( optional: true);
        Parameter longitude = EntityParameters.LONGITUDE( optional: true);
        Parameter marcaTemporal = EntityParameters.MARCATEMPORAL( optional: true);

        Boolean result = UI_Utils.getMultipleInputs(new ArrayList<>() {
            {add(matricula); add(nomeCondutor); add(latitude); add(longitude);
                add(marcaTemporal);}
        });

        if (!result)
            return;

        try (
                DataScope<ListAllAlarmes, NullType> ds_list_all_alarmes = new DataScope<>(ListAllAlarmes.class);
                DataScope<Veiculo, String> ds_veiculo = new DataScope<>(Veiculo.class)
        ) {
            ListAllAlarmes newItem = new ListAllAlarmes();
            Veiculo v = ds_veiculo.getSingle(matricula.value.toString());
            newItem.setMatricula(v);
            newItem.setNome_condutor(nomeCondutor.value.toString());
            if (latitude.value != null)
                newItem.setLatitude(latitude.value.toString());
            if (longitude.value != null)
                newItem.setLongitude(longitude.value.toString());
            if (marcaTemporal.value != null)
                newItem.setMarca_temporal(Timestamp.valueOf(marcaTemporal.value.toString()));

            ds_list_all_alarmes.create(newItem);

            // Vote
            ds_list_all_alarmes.validateWork();
            ds_veiculo.validateWork();
        }
        System.out.println("[DONE] Alarme inserido sobre a vista");
    }
}
```

Figura 5. Código para execução de um comando

7 Controlo de Concorrência Otimista

O Controlo de concorrência otimista consiste na realização de operações sem a posse de *Lock* em que que outras transações não são impedidas de modificar o mesmo conjuntos de dados.

A solução para evitar problemas de consistência nestes casos passa por manter registo da versão de cada registo de uma tabela. Desta forma é possível detetar se houve alterações realizadas a certos registos entre o início e fim da "nossa" transação e dessa forma abortar se necessário.

Para a alínea 2.f) da fase anterior foi nos pedido que utilizássemos esta técnica.
Com **JPA** decidimos realizar *Optimistic Locking* fazendo uso de ***LockModeType.OPTIMISTIC***.

Esta alínea envolve passar os registos da tabela de registos não processados para as tabelas de registos processados ou registos inválidos (caso não cumpram certos requisitos).

Como apenas a tabela Registo têm informações acerca de cada registo e as outras 3 tabelas apenas têm uma chave estrangeira para esta tabela pensámos que apenas fosse necessário "proteger" esta tabela com uma coluna de **versão**.

Caso um registo fosse inválido aquando do inicio do processo de tratamento de registos, mas entretanto outra transação o tornasse válido, isto deveria obrigar a nossa transação a falhar com exceção relacionada com *Optimistic Locking.*

O **JPA** dá-nos a garantia de que quando atualiza os dados de um registo marcado com locking optimista e cuja tabela têm uma coluna de versão o valor dessa coluna é incrementado.

Outras aplicações que possam interagir com a base de dados podem não fazer o mesmo. Isto levaria o *JPA* a pensar que o registo não teria sido modificado desde o início da sua transação e então faria *commit*, que iria potencialmente "esmagar" os valores entretanto alterados, provocando *lost updates*.

Para resolver este problema utilizámos uma solução presente no segundo documento das **Referências**. Criámos um trigger que é ativado em caso de *INSERT* ou *UPDATE* sobre a tabela ***Registo*** e chama uma função que incrementa o valor da versão do registo em causa.

7.1 Teste de Controlo de Concorrência Otimista

Para testar a funcionalidade referida anteriormente fizémos o seguinte:

Configuração:

- Na aplicação Java colocámos uma instrução bloqueante (*Scanner.nextLine()*) antes de realizar *commit* de uma transação que iria observar cada registo da tabela **Registo** e decidir colocá-los na tablela de registos **Processados** ou na tabela de registos **Inválidos** (caso não cumprissem um conjunto de critérios);

- Ainda na aplicação Java, ao iterar pelos registos não processados, a operação de ir á base de dados "buscar" o registo utiliza a função *EntityManager.find(Class, Object, LockModeType)*, passando como *LockModeType* o valor *LockModeType.**OPTIMISTIC***.

Intruções:

1) Iniciámos a aplicação e escolhemos a opção de **Processamento de Registos**;
2) O processamento é realizado, mas o programa fica bloqueado antes de fazer *commit* até que seja premida a tecla *Enter*;
3) No DBeaver atualizamos um campo qualquer de um dos registos não processados aquando do inicio da transação da aplicação Java;
4) Este *UPDATE* dispara o *trigger* mencionado previamente que atualiza a coluna de versão do registo afetado;
5) Na aplicação Java pressionamos a tecla *Enter* provocando *commit*;
6) Ocorre exceção *OptimisticLockException* uma vez que a versão de um dos registos a processar em tempo de *commit* não é a mesma que era no início da transação iniciada pela aplicação Java.

De seguida encontram-se figuras que mostram o processo descrito acima.

```
| -- MAIN MENU OPTIONS -- |
1) Inserir Cliente Particular
2) Remover Cliente Particular
3) Atualizar Cliente Particular
4) Total de alarmes para um ve◆culo
5) Processamento de registos
6) Inserir Veiculo
7) Inserir Estado GPS
8) Remover Estado GPS
9) Listar Estados GPS
10) Inserir sobre a vista de Alarmes
11) Listar Alarmes
12) Apagar Registos Inv◆lidos
13) Inserir Veiculo (Procedimento Armazenado)
14) Processamento de registos (Optimistic Locking)
15) Exit Program

> Choose an Option from 1 to 15:
14

type 'exit' to go back to the menu

```

Figura 6. Escolha da opção e bloqueados até pressionar Enter (before commit)

id_registo
1
2
3
4
5
6
7

Figura 7. Tabela de registos Não Processados

id	id_gps	longitude	latitude	marca_temporal	vers
1	1,001	38.8951	-77.0364	2022-01-22 01:10:25.000	0
2	2,002	[NULL]	38.3491	2016-06-22 07:23:25.000	0
3	3,003	29.3458	-93.2394	2016-06-22 10:10:50.000	0
4	4,004	12.3493	[NULL]	[NULL]	0
5	1,001	32.8341	-77.0364	2016-06-22 19:10:25.000	0
6	2,002	-90.0364	[NULL]	2022-04-24 17:10:15.000	0
7	2,002	[NULL]	23.8951	2016-06-22 14:10:20.000	0

Figura 8. Tabela de Registos

id	id_gps	longitude	latitude	marca_temporal	vers
1	1,001	23. 23	-77.0364	2022-01-22 01:10:25.000	1
2	2,002	[NULL]	38.3491	2016-06-22 07:23:25.000	0
3	3,003	29.3458	-93.2394	2016-06-22 10:10:50.000	0
4	4,004	12.3493	[NULL]	[NULL]	0
5	1,001	32.8341	-77.0364	2016-06-22 19:10:25.000	0
6	2,002	-90.0364	[NULL]	2022-04-24 17:10:15.000	0
7	2,002	[NULL]	23.8951	2016-06-22 14:10:20.000	0

Figura 9. Tabela de Registos após atualização de um campo de um dos registos

```
> Choose an Option from 1 to 15:
14

type 'exit' to go back to the menu



[UNCAUGHT ERROR]: [DB ERROR] OPTIMISTIC LOCK EXCEPTION: Please try again later

Press ENTER to continue...

```

Figura 10. Mensagem de erro mostrada ao utilizador ao haver OptimisticLockException

8 Conclusão

Com a realização deste trabalho aprendemos a construir uma aplicação em Java utilizando *JPA* que interage com os dados da base de dados mantendo consistência dos mesmos através de técnicas como o controlo de concorrência otimista.

Nesta aplicação, para além do mapeamento de objetos a entidades da base de dados tivemos também que ter em conta controlo de concorrência, gestão transacional (através da construção de uma camada de acesso a dados genérica à qual chamámos Data Scope) e desenho geral de uma aplicação de linha de comando com suporte de comandos para a execução das várias funcionalidades pretendidas para a aplicação.

Referências

1: Walter Vieira, SISINF_M1_Transações(v6).pdf
2: Walter Vieira, SISINF_M3_Acesso_a_Dados(v3).pdf

Anexos

(Não existem anexos neste documento)