ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Trabalho Prático

(Fase 2)

Alunos: Guilherme Cepeda 47531

Rafael Coelho 47578

Tiago Martinho 48256

Docentes: Nuno Leite

Walter Vieira

Relatório final da 2ª Fase realizado no âmbito de Sistemas de Informação, do curso de licenciatura em Engenharia Informática e de Computadores
Semestre de Verão 2022/2023

Junho de 2023

<< Esta página foi intencionalmente deixada em branco >>

Resumo

Durante esta segunda fase do trabalho foi nos pedido para desenvolvermos uma camada de acesso a dados, que usou uma implementação de JPA.

Adicionalmente também foi desenvolvida uma aplicação programada em Java, que faz a ligação entre o utilizador e a camada de acesso a dados.

Também foi implementado os padrões de desenho DataMapper, Repository e UnitOfWork.

Índice

Lista de Figuras

1. Introdução

Durante primeira fase do trabalho de SI foi nos pedido para implementar um sistema de informação para a gestão de jogos, jogadores e as partidas que estes efetuam para a empresa **“GameOn”**.

Para esta segunda fase foi nos pedido então que desenvolvessemos duas coisas:

Primeiro uma camada de acesso a dados para tratar com esse mesmo sistema de informação que foi desenvolvido durante a primeira fase do trabalho e , segundo, uma applicação em java para um utilizador poder manipular essa mesma base de dados.

Tendo isso em conta também nos foi pedido para na nossa camada de acesso a dados usar uma implementação de JPA que usasse os subconjunto dos padrões de desenho DataMapper, Repository e UnitOfWork.

Aprofundaremos ao longo deste trabalho então como implementamos cada uma dos padrões de desenho e o restante do que foi pedido no enunciado, cada um na sua secção.

2. Arquitetura do sistema

Para este trabalho utilizamos uma arquitetura de sistema com 4 camadas, duas das quais realizadas pela JPA application, sendo as outras a base de dados em PostgreSql e a interface do utilizador.

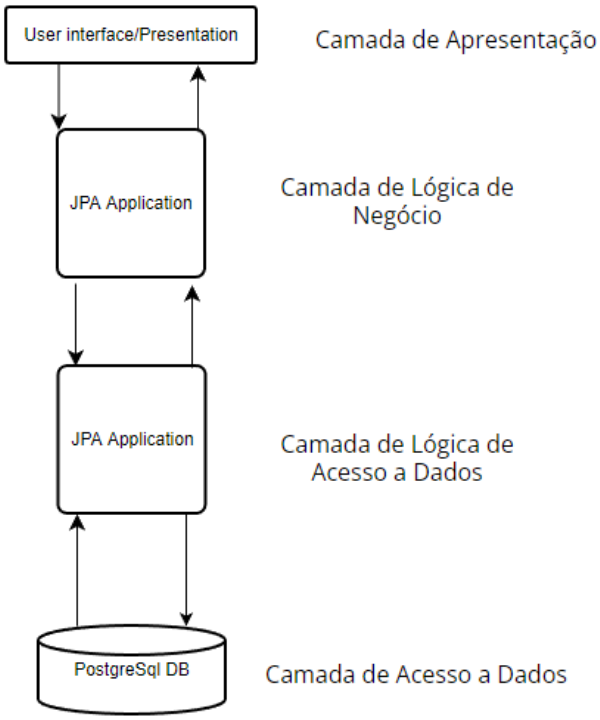


Figura 1 - Arquitetura do Sistema

Falemos então de algumas alterações que foram realizadas à base de dados inicial

2.1 Alterações ao Modelo Físico

A única alteraçao feita ao modelo da 1ª fase do trabalho prático foi na tabela **BADGE** foi acrescentado o tuplo "version" para fazer o suporte de *optimistic locking.*

Onde a adição desta coluna serve para o controlo da versão da tabela **BADGE**, dado que em modo optimistic locking, o JPA incrementa este valor ao dar commit de uma transação que faça alterações sobre o tuplo. Deste modo garante que quaisquer leituras e atualizações realizadas estão consistentes se o valor for igual ao que era no inicio da transação.

2.2 Utilização de JPA e persistência de dados

Jakarta Persistence API ou JPA é o ponto focal do trabalho.

Esta ferramenta, permite-nos como desenvolvedores obter persistência em termos de código, criamos classes que refletem as tabelas da base de dados onde iremos aceder, criar, remover e atualizar objetos que representam as entradas nas tabelas do modelo.

O objetivo principal do JPA é permitir ao desenvolvedor ignorar a necessidade de pensar relacionalmente, permitindo, ao invês, que o programador decida o que e como é que a informação persiste, dentro das ideias de código de Java.

Isto é conseguido expandindo nas funcionalidades do API Java DataBase Connectivity ou JDBC, especialmente quando se usa uma base de dados relacional, para o programador, ao obter a informação da base de dados da sua escolha, poder guardar essa informação em objetos, que estão colocados num mapa, onde estão definidas as relações entre si e quais persistem (ou não).

2.3 Padrões de Desenho (DataMapper, Repository e UnitOfWork)

Para a realização deste trabalho também nos foi pedido para utilizarmos 3 padrões de desenho. Iremos então falar um pouco de cada um:

Data Mapper:

Este padrão de desenho consiste na transferência de dados de forma bidirecional, entre a base de dados e uma estrutura de dados persistente na memoria do programa que se mantém consistente com os dados na base de dados, permitindo que mesmo que o programa pare, este tenha acesso aos dados ao reiniciar. Este padrão disponibiliza as operações Create, Read, Update e Delete (CRUD) para facilitar o acesso à base de dados.

Neste caso quem cria esta persistência é o JPA, o mesmo que nos dá as ferramentas para criar este data mapping..

Repository:

Este padrão de desenho consiste em armazenar a lógica de acesso a dados numa seção do código de forma a se esconder a lógica de acesso a dados da lógica de negócio, para evitar dar informação indesejada, permitindo assim ao negócio aceder as dados sem saber como a informação é tratada.

Neste caso é uma parte integral do código pois é usado diretamente junto com o padrão de Data Mapper para termos um código organizado e seguro quando desejamos utilizar o JPA.

Unit of Work:

Este último padrão de desenvolvimento trata de otimizar as mudanças na base de dados.

Para o pudermos aplicar o Unit of Work todas as tecnologias mencionadas até agora são uma grande ajuda, porque são estas que permitem que o Unit of Work seja implementado facilmente.

Este trata-se de ao invês de enviar um update cada vez que acontece uma alteração nos dados, enviar um grande número de updates de uma vez, tornando assim mais rentável e eficiente o tempo de conexão e os recursos dispendidos na mesma, sendo que este processo é largamente facilitado pela existência de um **Data Mapper** e de um **Repository,** porque como estes criam uma espécie de cópia da informação da base de dados, torna mais fácil acumular updates sem correr o risco de obter erros nos dados.

2.4 Organização e funcionamento da aplicação JPA

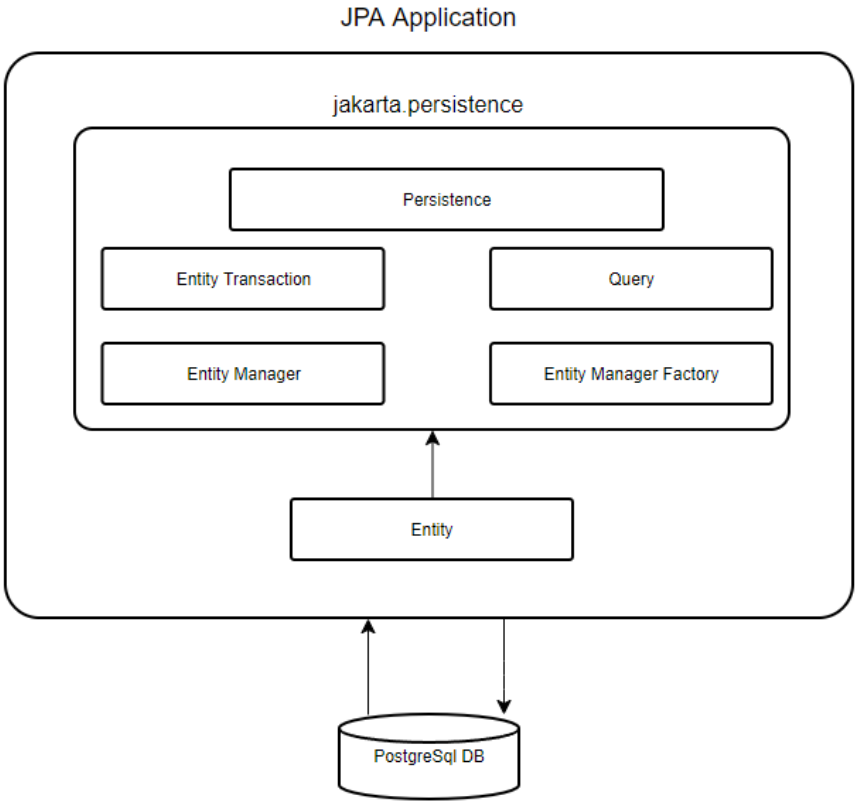


Figura 2 - JPA application

Uma aplicação JPA permite através da linguagem de programação ***JAVA*** a utilização direta dos objetos em vez de declarações diretas na linguagem SQL, como por exemplo na aplicação efetuada no semestre anterior na UC de ISI.

A implementação usada nesta aplicação é a mais comum e de referência o ***EclipseLink.*** Nesta aplicação JPA conseguimos guardar, atualizar e mapear dados existentes na Base de Dados criada na primeira fase deste trabalho para objetos java que posteriormente são usados nas operações realizadas.

Para a conexão da Base de Dados foi utilizado o ficheiro persistence.xml para mapear as entidades da Base de Dados local nos objetos java que representam essas entidades, neste ficheiro temos o JPA provider que é utilizado para fazer as várias operações existentes na BD corretamente.

Nas entidades utilizamos anotações para definir metodos existentes na BD , assim como a chave primária, nome da entidade, chaves estrangeiras, colunas existentes na tabela e associações entre tabelas (ManyToOne, OneToMany, OnetoOne, ManyToMany).

Em alguns casos na nossa BD temos chaves primárias como ***SERIAL*** e essas são representadas com a anotação *@GeneratedValue* de modo a serem geradas automaticamente sempre que existir uma inserção na respetiva tabela.

A logica da aplicação é executada no ***JPAContext*** onde temos varios ***Repositories e Mappers*** de cada uma das entidades existentes, nos quais temos acesso a metodos que nos permitem executar as operações e manter a persistencia de dados.

2.5 Detalhes de implementação

Começando por explicar os ***Repositories*** criados, estes são utilizados como referido no capitulo anterior onde por exemplo para cada entidade temos 5 métodos : ***getAll() , find() , add(), delete() e save()*** como por exemplo na figura 3.

```
public interface IRepository<Tentity,Tkey> {

    1 usage   5 implementations
    List<Tentity> getAll() throws Exception;;
    5 implementations
    Tentity find(Tkey k) throws Exception;;
    5 implementations
    void add(Tentity entity) throws Exception;;
    5 implementations
    void delete(Tentity entity) throws Exception;;
    no usages   5 implementations
    void save(Tentity e) throws Exception;;

}
```

Figura 3 - Repository

Onde de acordo com o tipo de chave no ***find()*** conseguimos aceder ao objeto especifico da respetiva entidade e aceder aos seu membros.

o método **getAll()** obtém os tuplos todos da tabela associada à entidade representada pelo tipo genérico **TEntity** sobre forma de lista

O método ***add()*** adiciona ao modelo de persistência o objeto do tipo TEntity passado por parâmetro (analogamente, adiciona um tuplo à tabela associada).

O método ***delete()*** remove o tuplo representado pelo objeto.

O método ***save()*** atualiza em persistência o objeto especificado, atualizando todos os seus membros exceto keys.

Já nos ***Mappers*** estes tambem têm metodos de *Create, Update, Delete e Read* que através do uso da classe **DataScope** têm acesso á **EntityManager** que por sua vez irá permitir fazer queries para controlo e acesso a base de dados, como por exemplo na figura 5. Como está explicado na class **AbstractDataScope** no ponto 2.5.1.

```java
public interface IMapper<Tentity,Tkey>{

    6 implementations
    Tkey create(Tentity e) throws Exception;

    6 implementations
    Tentity read(Tkey k) throws Exception; // acesso dada a chave

    6 implementations
    void update(Tentity e) throws Exception;

    6 implementations
    void delete(Tentity e) throws Exception;
}
```

Figura 4- Mapper

```java
public class MapperPlayer implements IMapper <Player, Integer>  {
    @Override
    public Integer create(Player e) throws Exception {
        try (DataScope ds = new DataScope()) {

            EntityManager em = ds.getEntityManager();
            em.persist(e);
            ds.validateWork();
            return e.getId();

        }
    }
```

Figura 5 – Exemplo Mapper Create Player

```java
@Override
public Player read(Integer id) throws Exception {
    try (DataScope ds = new DataScope()) {
        EntityManager em = ds.getEntityManager();
        em.flush();  // • necess•rio para o pr•ximo find encontrar o registo ca
        Player p = em.find(Player.class, id, LockModeType.PESSIMISTIC_WRITE);
        ds.validateWork();
        return p;
    }
}
```

Figura 6 - Exemplo Mapper Read Player

```java
@Override
public void update(Player e) throws Exception {
    try (DataScope ds = new DataScope()) {
        EntityManager em = ds.getEntityManager();
        em.flush();  // É necessário para o próximo find encontrar o registo caso ele tenha
        Player p1 = em.find(Player.class, e.getId(), LockModeType.PESSIMISTIC_WRITE);
        if(p1 == null)
            throw new java.lang.IllegalAccessException("Entidade inexistente");

        // Set values of persistent data (except id)
        p1.setEmail(e.getEmail());
        p1.setActivityState(e.getActivityState());
        p1.setUsername(e.getUsername());
        p1.setRegionName(e.getRegionName());

        ds.validateWork();

    }
}
```

Figura 7 - Exemplo Mapper Update Player

```java
@Override
public void delete(Player e) throws Exception {
    try (DataScope ds = new DataScope()) {
        EntityManager em = ds.getEntityManager();
        em.flush(); // É necessário para o próximo find encontrar o registo caso ele t

        Player p1 = em.find(Player.class, e.getId(), LockModeType.PESSIMISTIC_WRITE);
        if (p1 == null)
            throw new java.lang.IllegalAccessException("Entidade inexistente");
        em.remove(p1);

        ds.validateWork();

    }
}
```

Figura 8 - Exemplo Mapper Delete Player

No método **create()**, através de um objeto do tipo ***EntityManager*** conseguimos manter a persistência dos dados com método *persist()* criar um novo player, exemplo figura 5. Com o método remove retiramos o player com aquela chave da tabela, aplicado no método ***delete()*** este irá ser eliminado. Embora exista um caso especifico onde assim que chamarmos este método o player não irá ser eliminado mas só irá ter o campo state atualizado de acordo com o enunciado e com o trigger feito no Ex L na primeira fase deste trabalho, exemplo figura 8.

O método ***update()*** acede ao cliente e altera os campos que este quiser alterar e que sejam permitidos, exemplo figura 7.

O método ***read()*** meramente acede aos dados da entidade, exemplo figura 6.

Quanto aos modos de leitura escolhemos o **PESSIMISTIC_WRITE** porque ao iniciar a escrita na base de dados após o commit não permite que esta seja atualizada ou eliminada mantendo esta num "lock" e protegida de outras transações que possam ocorrer em simultâneo.

Quanto ao **Optimistic Locking** este baseia-se na verificação de mudanças na BD, e de transações concorrentes, nomeadamente nas entidades através da verificação da versão dos seus atributos é mais eficiente numa BD onde ocorrem mais leituras do que atualizações e eliminações. A utilização esperada no ex 2b era de que esta fizesse alterasse o campo *points_limit* da entidade **BADGE** e impedisse que locks fossem criados para a leitura desta de modo a que este procedimento (ex h da fase 1) pudesse ser chamado periodicamente sem problemas dado que não é vulnerável a "deadlocks" da BD por consequência de não fazer lock nas leituras. Como se pode ver no exemplo abaixo na figura 10.

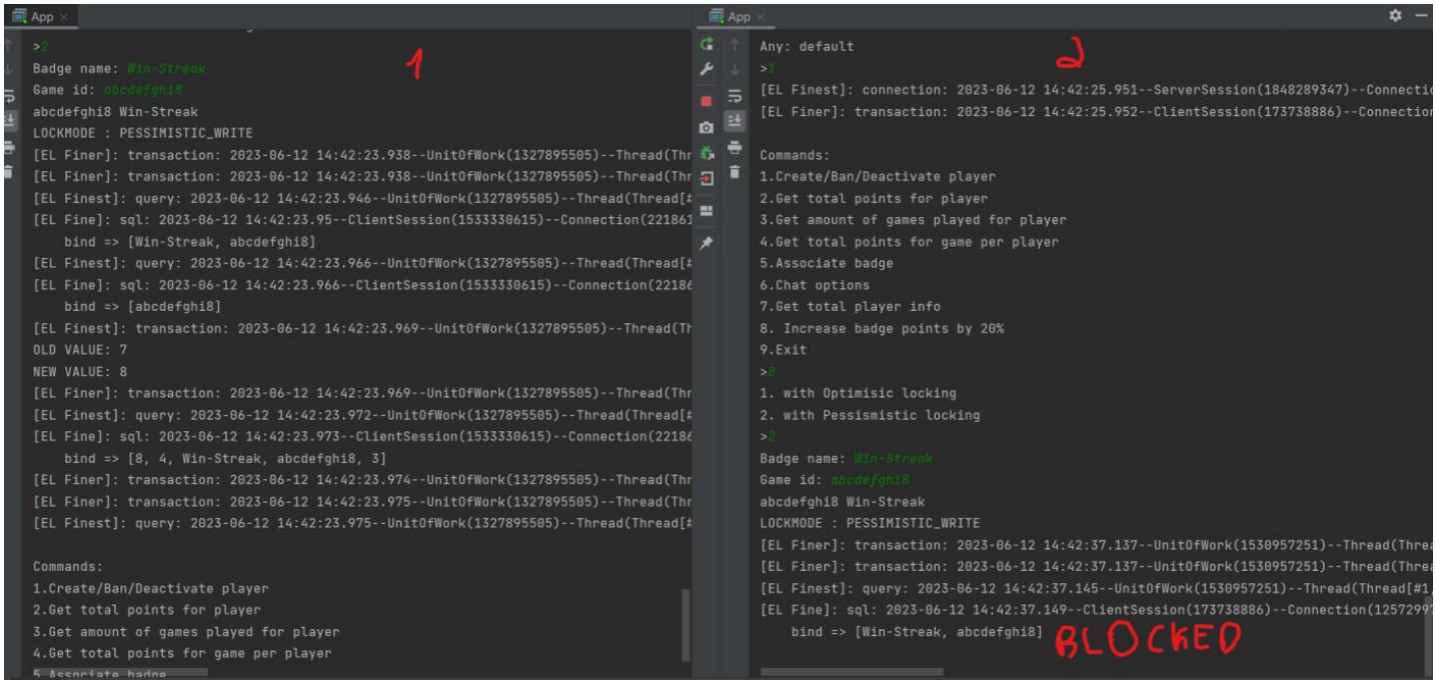


Figura 9 - Pessimistic Write Exemplo

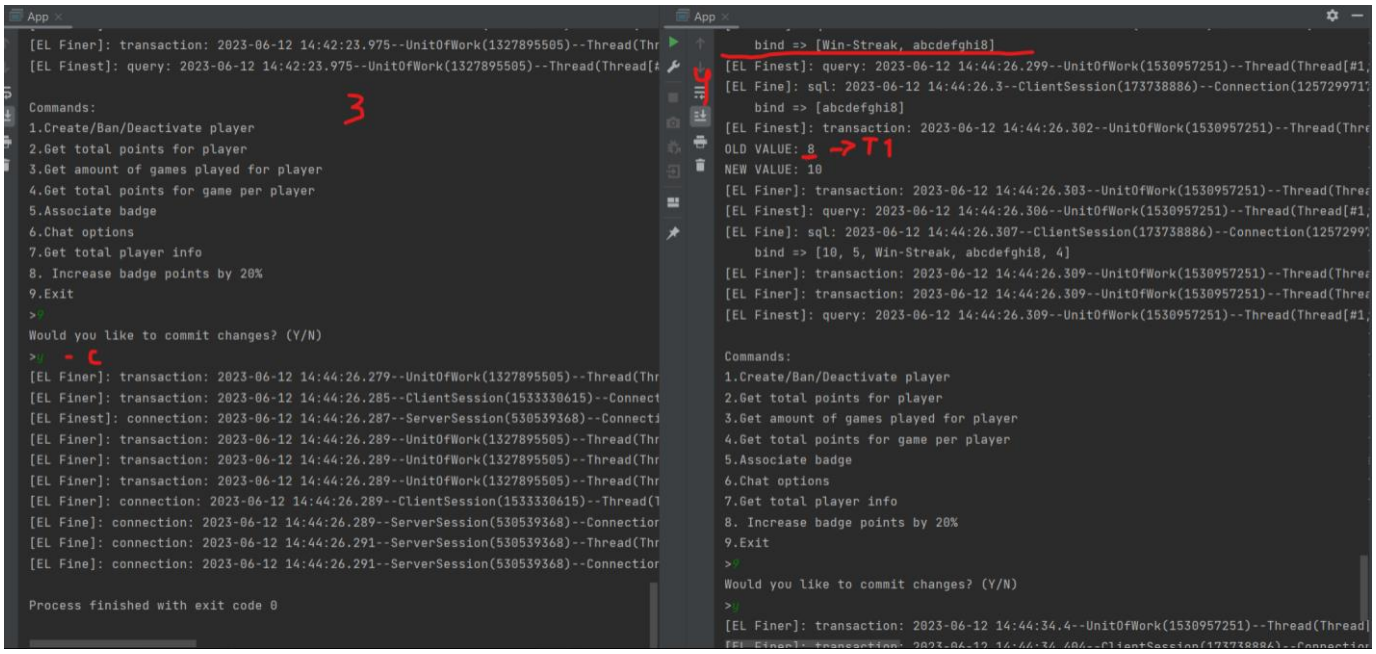
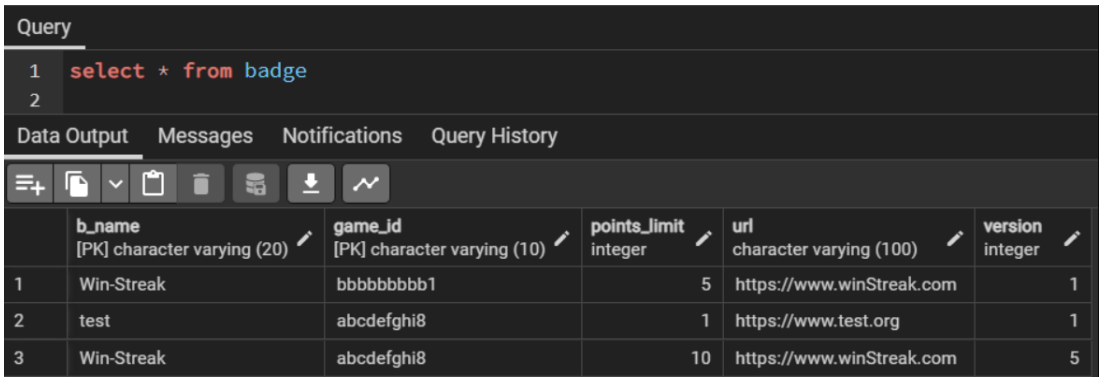


Figura 10 - Continuação do Pessimistic Exemplo



	b_name [PK] character varying (20)	game_id [PK] character varying (10)	points_limit integer	url character varying (100)	version integer
1	Win-Streak	bbbbbbbbb1	5	https://www.winStreak.com	1
2	test	abcdefghi8	1	https://www.test.org	1
3	Win-Streak	abcdefghi8	10	https://www.winStreak.com	5

Figura 11 - Resultado do Pessimistic write

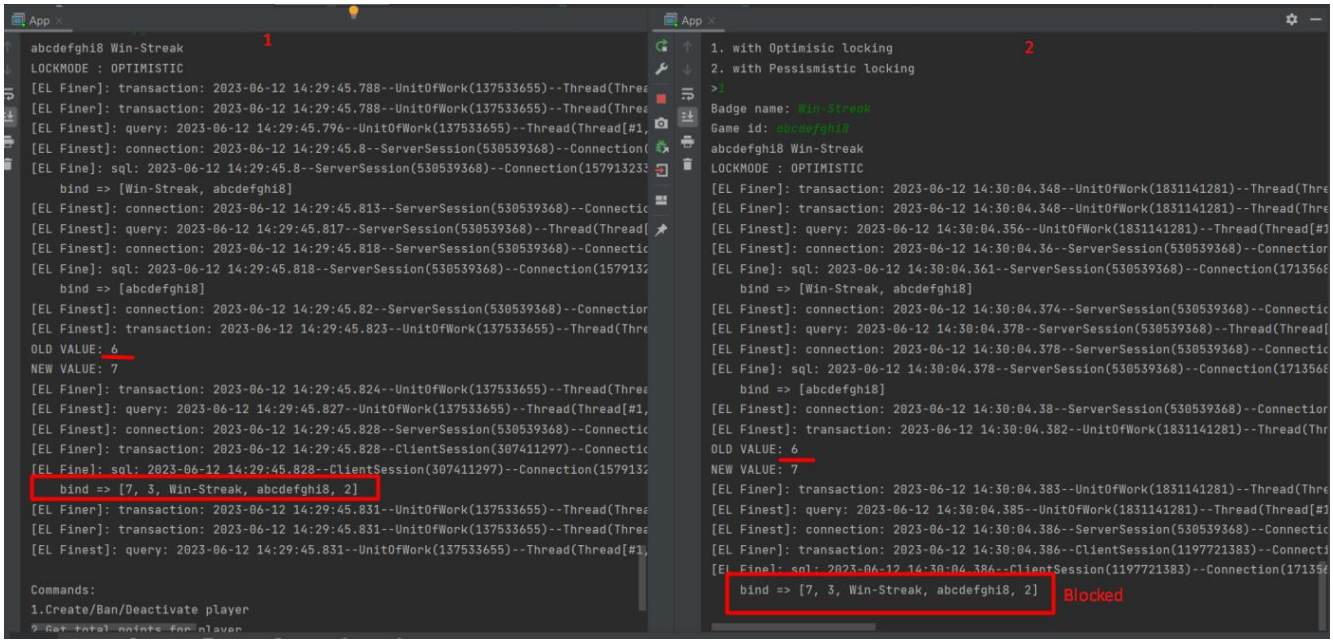


Figura 12 - Optimistic Locking Exemplo

2.5.1 AbstractDataScope e DataScope Classes

A classe AbstractDataScope faz uso de Thread Local Storage (TLS) para armazenar o modelo de persistência e a instância de transação associada. A classe DataScope expande esta com a possibilidade de mudar o nivel de isolamento da transação. Instanciações subsequentes (ou aninhadas) de um objeto DataScope apenas acedem aos elementos previamente guardados. Quando o método close do objeto que colocou os elementos na TLS é chamado, são então removidos os elementos da TLS e é feito então o commit ou rollback da transação associada caso o trabalho tenha sido marcado como válido ou inválido, respetivamente. Permite também a qualquer momento abortar a transação corrente e iniciar uma nova (com o mesmo nível de isolamento) sobre o mesmo padrão de desenho.

Só na operação de commit é que as várias operações que alteram o estado da base de dados vão realmente ocorrer, deste modo agrupamos um número x de updates de uma só vez. Assim se garante o padrão de desenho UnitOfWork.

2.5.2 Exception Handling

Foi criada a classe ServiceWrapper que tem como objetivo evitar a repetição de código escrito de duas formas:

- Evitando a utilização do padrão try-with-resources para instanciar o objeto do tipo DataScope;
- Evitar o handling de erros repetitivo providenciando um handling correto e genérico o suficiente à maior parte dos casos.

3. Conclusão

Ao longo deste trabalho tivemos a opurtunidade de desenvolver uma camada de acesso a dados, usando um stack tecnológico e padrões de desenvolvimento atualizados e relevantes, tendo produzido uma boa solução para o pedido em termos de enunciado, o que nos dá a confiança que numa situação onde o fosse pedido iriamos conseguir desenvolver um bom sistema de informação.

Aprendemos também que o Optimistic locking é mais eficiente em BD que usem mais leituras do que updates e delete e que o pessimistic locking permite um melhor controlo de transações com o lock em transações concurrentes.

Acreditamos assim ter atingido com sucesso os objetivos de aprendizagem do trabalho.

4. Referências

Repository:

https://cubettech.com/resources/blog/introduction-to-repository-design-pattern/

Data Mapper:

https://designpatternsphp.readthedocs.io/en/latest/Structural/DataMapper/README.html#data-mapper

Unit Of Work:

https://java-design-patterns.com/patterns/unit-of-work/

JPA application:

https://www.infoworld.com/article/3379043/what-is-jpa-introduction-to-the-java-persistence-api.html

https://www.vogella.com/tutorials/JavaPersistenceAPI/article.html#:~:text=JPA%20permits%20the%20developer%20to,is%20defined%20via%20persistence%20metadata.

UnitOfWork:

https://java-design-patterns.com/patterns/unit-of-work/

Optimistic locking:

https://www.baeldung.com/jpa-optimistic-locking

Pessimistic locking:

https://www.baeldung.com/jpa-pessimistic-locking

a. Anexos

Modelo EA no diagrama no dia.

Scripts em sql em anexo.