
Prog. Orientada a Objetos e Mapeamento Objeto-Relacional – IMD0104

Controle Transacional

João Carlos Xavier Júnior

jcxavier@imd.ufrn.br

Estratégias de Transação

❑ Estratégia JDBC:

- ❖ O controle das transações é de responsabilidade do **java.sql.Connection** da especificação JDBC.
- ❖ Estratégia adequada para persistência de dados com JDBC em **apenas uma base de dados**.
- ❖ Para agrupar mais de uma mudança em uma transação é necessário desativar a opção de **auto commit**.
- ❖ Use `commit()` para gravar permanentemente mudanças ou `rollback()` para desfazer mudanças não comitadas.
 - **`connection.commit();`**
 - **`connection.rollback();`**

Estratégias de Transação

❑ Estratégia JPA:

- ❖ O controle das transações é de responsabilidade do **javax.persistence.EntityManager** da especificação JPA.
- ❖ Estratégia adequada para persistência de dados com JPA em **apenas uma base de dados**.
- ❖ Como um **EntityManager** acessa apenas uma unidade de persistência, não há como fazer o controle transacional de unidades distintas.

Estratégias de Transação

❑ Estratégia JTA (Java Transactional API):

- ❖ Estratégia usada em **ambientes gerenciados**.
- ❖ Ela delega o controle da transação para um **container JEE** (Java Enterprise Edition):
 - Servlets (container para aplicações Web); ou
 - EJBs (container para componentes de negócio).
- ❖ O **JTATransaction** possibilita a inclusão de **várias unidades de persistência** em uma mesma aplicação.

Estratégias de Transação

❑ Exemplo de Transação JTA (persistence.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="bookmark-ds" transaction-type="JTA">
    <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.format_sql" value="true"/>
      <property name="hibernate.transaction.jta.platform"
        value="org.hibernate.service.jta.platform.internal.JBossAppServerJtaPlatform" />
    </properties>
  </persistence-unit>
</persistence>
```

Ambientes não gerenciados

x

Ambientes gerenciados

Estratégias de Transação

❑ Exemplo de persistence.xml para transação local:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence persistence_1_0.xsd" version="1.0">
  <persistence-unit name="acme" transaction-type="RESOURCE_LOCAL">
    <non-jta-data-source>acme</non-jta-data-source>
  </persistence-unit>
</persistence>
```

```
EntityManager em = createEntityManager();
em.getTransaction().begin();
Employee employee = em.find(Employee.class, id);
employee.setSalary(employee.getSalary() + 1000);
em.getTransaction().commit();
em.close();
```

Estratégias de Transação

❑ Exemplo de persistence.xml para transação JTA:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence persistence_1_0.xsd" version="1.0">
  <persistence-unit name="acme" transaction-type="JTA">
    <jta-data-source>acme</jta-data-source>
  </persistence-unit>
</persistence>
```

```
UserTransaction transaction = (UserTransaction)new InitialContext().lookup("java:comp/UserTransaction");
transaction.begin();
EntityManager em = getEntityManager();
Employee employee = em.find(Employee.class, id);
employee.setSalary(employee.getSalary() + 1000);
transaction.commit();
```


Flushing

□ Contexto:

- ❖ JPA trabalha com o conceito de **cache**.
- ❖ Salva sempre todas as entidades no **PersistenceContext** (memória).
- ❖ Objetos em memória são armazenados no banco de dados, quando:
 - A transação é “**commitada**”;
 - Antes de uma **query** ser executada; ou
 - Quando o método **em.flush()** for chamado.

Flushing

❑ Exemplo:

```
public class JPDAO {
    EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("JPAService");
    EntityManager em = emf.createEntityManager();

    public void create() {
        em.getTransaction().begin();
        Student student = new Student();
        student.setId(1);
        student.setName("Joe");
        student.setDateOfBirth(new Date());
        student.setGender(Gender.FEMALE);

        em.persist(student);
        em.flush();
        em.getTransaction().commit();
    }
}
```

Flushing

❑ Observação:

- ❖ Ao chamar **`persist()`**, **`merge()`** ou **`remove()`**, essas alterações **não são sincronizados** com o banco de dados.
- ❖ O gerenciador de entidades (**`EntityManager`**) precisa gravá-las.
 - **`em.getTransaction().commit()`**.
- ❖ Você pode forçar a sincronização a qualquer momento chamando **`em.flush()`**.

Níveis de Isolamento

- ❑ A propriedade do isolamento pode ter diversos graus de execução.
- ❑ Quanto mais isolada for uma transação, mais sincronização é necessário fazer:
 - ❖ Quanto mais **sincronização**, menor a **escalabilidade**.
- ❑ Os banco de dados usam para controle de concorrência o modelo de MVCC (**Multi-version concurrency control**).

Problemas de Isolamento

- ❑ Em cenários standalone, é possível desfazer todo um bloco de procedimentos apenas com um simples **Rollback**.
- ❑ Em **sistemas concorrentes** isso é mais complexo e difícil de garantir.
- ❑ Caso o isolamento não seja garantido alguns problemas podem ocorrer:
 - ❖ Dirty read;
 - ❖ Unrepeatable Reads;
 - ❖ Phantom Read.

Isolamento Adequado

□ Níveis de isolamento:

- ❖ Serializável (**SERIALIZABLE**): cada transação executa com completo isolamento .
- ❖ Leitura repetitiva (**REPEATABLE READ**): cada transação lê apenas **tuplas efetivadas** e nenhuma outra pode atualizar uma tupla que está em uso.
- ❖ Leitura com efetivação (**READ COMMITTED**): cada transação lê apenas **tuplas efetivadas** e qualquer outra pode atualizar uma tupla que está em uso.
- ❖ Leitura sem efetivação (**READ UNCOMMITTED**): cada transação lê tuplas não efetivadas.

Níveis de Isolamento

Velocidade

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read
Read uncommitted	Sim	Sim	Sim
Read committed	Não	Sim	Sim
Repeatable read	Não	Não	Sim
Serializable	Não	Não	Não



Níveis de Isolamento

□ Níveis possíveis:

- ❖ Read uncommitted (1)
- ❖ Read committed (2) - Default
- ❖ Repeatable read (4)
- ❖ Serialize (8);

□ Configurando Hibernate:

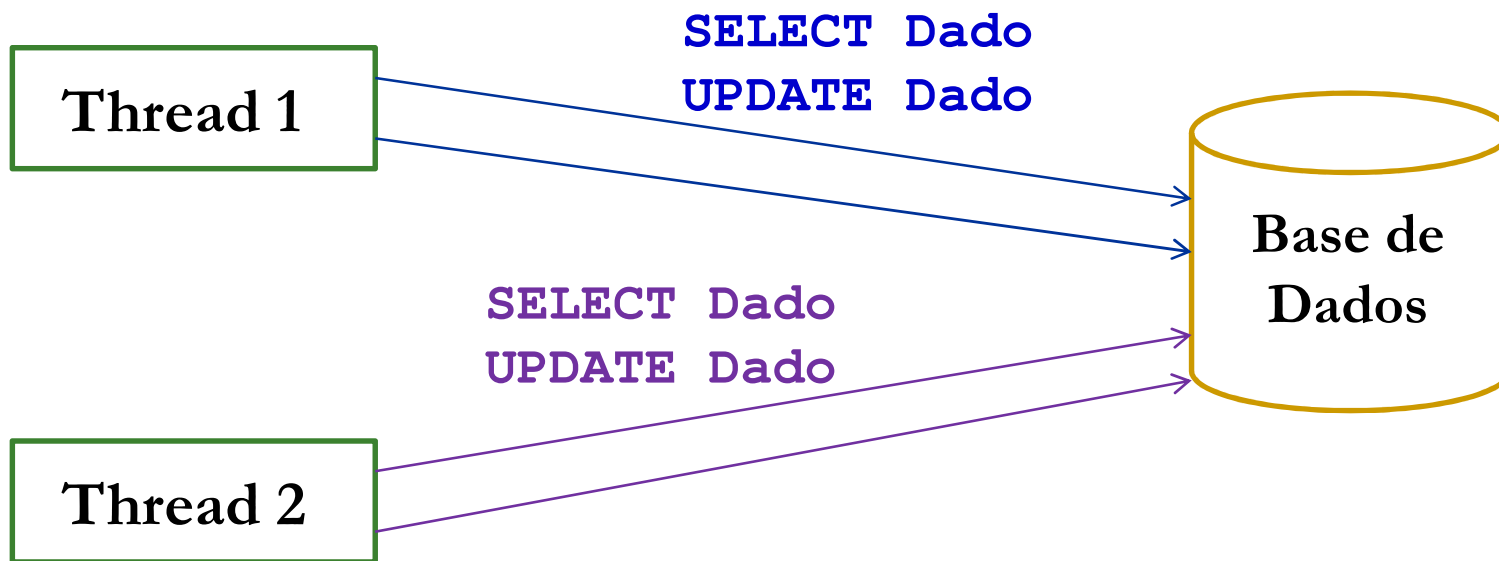
```
<?xml version="1.0"?>
<!DOCTYPE hibernate-configuration PUBLIC "
    -//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory name="">
        <property name="connection.pool_size">2</property>
        <property name="dialect">org.hibernate.dialect.PostgreSQLDialect</property>
        <property name="show_sql">true</property>
        <property name="hbm2ddl.auto">update</property>
        <property name="hibernate.connection.isolation">2</property>
    </session-factory>
</hibernate-configuration>
```

Concorrência

- ❑ Trata a modificação do mesmo dado, ao mesmo tempo.
- ❑ Existem dois tipos:
 - ❖ **Otimista:**
 - Usa mecanismo de versionamento de dado para que, antes de validar operações de escrita, seja checada a versão dos dados.
 - ❖ **Pessimista:**
 - O banco simplesmente trava o dado e só aquele que tem a trava consegue trabalhar com os dados.

Sem Tratamento

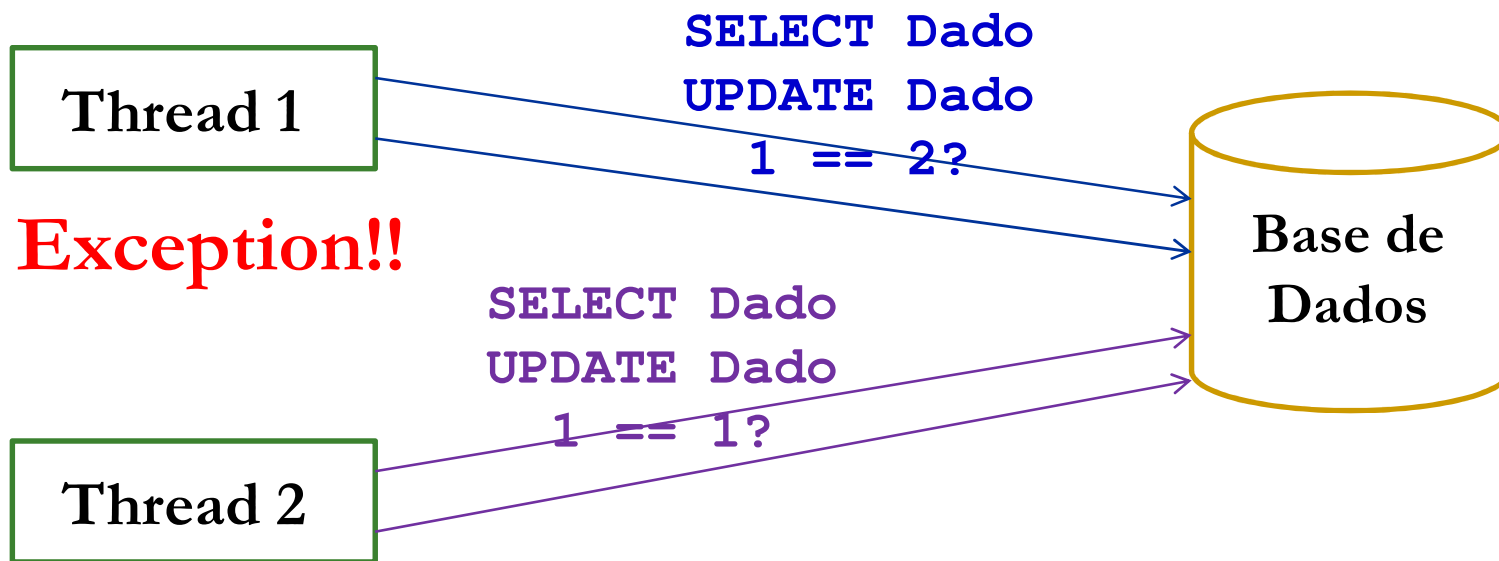
❑ Caso comum:



- ❑ Dois usuário acessando o mesmo produto.
- ❑ Usuário A modifica a **quantidade** e salva antes do usuário B, já o usuário B modifica o **nome** e salva depois.
- ❑ O que acontece?

Lock Otimista

□ Version Number:



Lock Otimista

□ Estratégia Número de Versão:

- ❖ Criar um novo campo, do tipo `int` com nome `version` e anotação: **@Version**.
 - Esse campo pode ser também `long` ou `timestamp`.
- ❖ Atribui-se um valor inicial ao atributo de versão.
- ❖ Ao realizar o update incrementa o valor da versão.
- ❖ Antes de alterar, verifica se o valor da versão do banco alterou, em caso positivo, gera a excessão.

Lock Otimista

❑ Estratégia Número de Versão:

```
// outros imports
import javax.persistence.Version;

@Entity
@Table(name = "usuario")
public class Usuario implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long codigo;
    private String nome;

    @Version
    private Integer versao;

    // getters e setters
    // equals e hashCode

}
```


Lock Otimista

❑ Estratégia Número de Versão:

 Usuário salvo com sucesso.

Salvar Pesquisar

Código	6
Nome	Maria Silva
Versão	0

 Usuário salvo com sucesso.

Salvar Pesquisar

Código	6
Nome	Maria Silva Santos
Versão	1

Lock Pessimista

□ Contexto:

- ❖ O lock pessimista bloqueia o dado até que ele seja atualizado.
- ❖ Alguns banco de dados, como o Oracle e PostgreSQL, utilizam a construção SQL **SELECT FOR UPDATE** para bloquear o dado até que o mesmo seja atualizado.
- ❖ É possível definir o **Lock Mode**:

```
em.getTransaction().begin();  
Aluno a = em.find(Aluno.class, 1,  
                LockModeType.PESSIMISTIC_WRITE);  
a.setNota(8.5);  
em.getTransaction().commit();
```

Tipos de Locks Pessimistas

- ❑ LockModeType.NONE:
 - ❖ Só realiza a consulta ao banco se o objeto não estiver no cache.
- ❑ LockModeType.READ:
 - ❖ Ignora os dados no cache e faz verificação de versão para assegurar-se de que o objeto em memória é o mesmo que está no banco.
- ❑ LockModeType.WRITE:
 - ❖ Passa ambos os níveis de cache e obtêm o lock pessimista do banco.

Dúvidas...





Pool de Conexão



Pool de Conexão

- ❑ A utilização do pool de conexão se faz necessária quando se deseja que o próprio sistema de acesso ao banco de dados gerencie a quantidade máxima de conexões ao banco.
- ❑ Utilização do **c3p0**.
 - ❖ é uma biblioteca “easy-to-use” usada para fornecer a capacidade de pooling de conexões.



Configurando c3p0

❑ Persistence.xml:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.0"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="ConexaoDB" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>br.edu.unirn.dominio.Venda</class>
    <shared-cache-mode>NONE</shared-cache-mode>
    <properties>
      <property name="connection.provider_class"
        value="org.hibernate.connection.C3P0ConnectionProvider" />
      <property name="hibernate.c3p0.acquire_increment" value="4" />
      <property name="hibernate.c3p0.idle_test_period" value="3000" />
      <property name="hibernate.c3p0.max_size" value="100" />
      <property name="hibernate.c3p0.max_statements" value="15" />
      <property name="hibernate.c3p0.min_size" value="5" />
      <property name="hibernate.c3p0.timeout" value="100" />
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.PostgreSQLDialect"/>
      <property name="hibernate.hbm2ddl.auto" value="update" />
      <property name="hibernate.show_sql" value="true" />
    </properties>
  </persistence-unit>
</persistence>
```

Configurando c3p0

- ❑ **connection.provider_class**: qual classe será responsável por gerenciar a conexão.
- ❑ **c3p0.acquire_increment**: quantas conexões serão adquiridas quando a conexão chegar ao limite.
- ❑ **c3p0.idle_test_period**: tempo que será testado para fechar a conexão e caso de não uso.
- ❑ **c3p0.max_size**: número máximo de conexões.
- ❑ **c3p0.min_size**: número mínimo de conexões.
- ❑ **c3p0.max_statements**: número máximo de conexões em cache.
- ❑ **c3p0.timeout**: tempo que será usado para liberar as conexões no pool.