

Agenda

Transaktionen, Locking

Strategien Transaktionen

Optimistic Locking

Pessimistic Locking

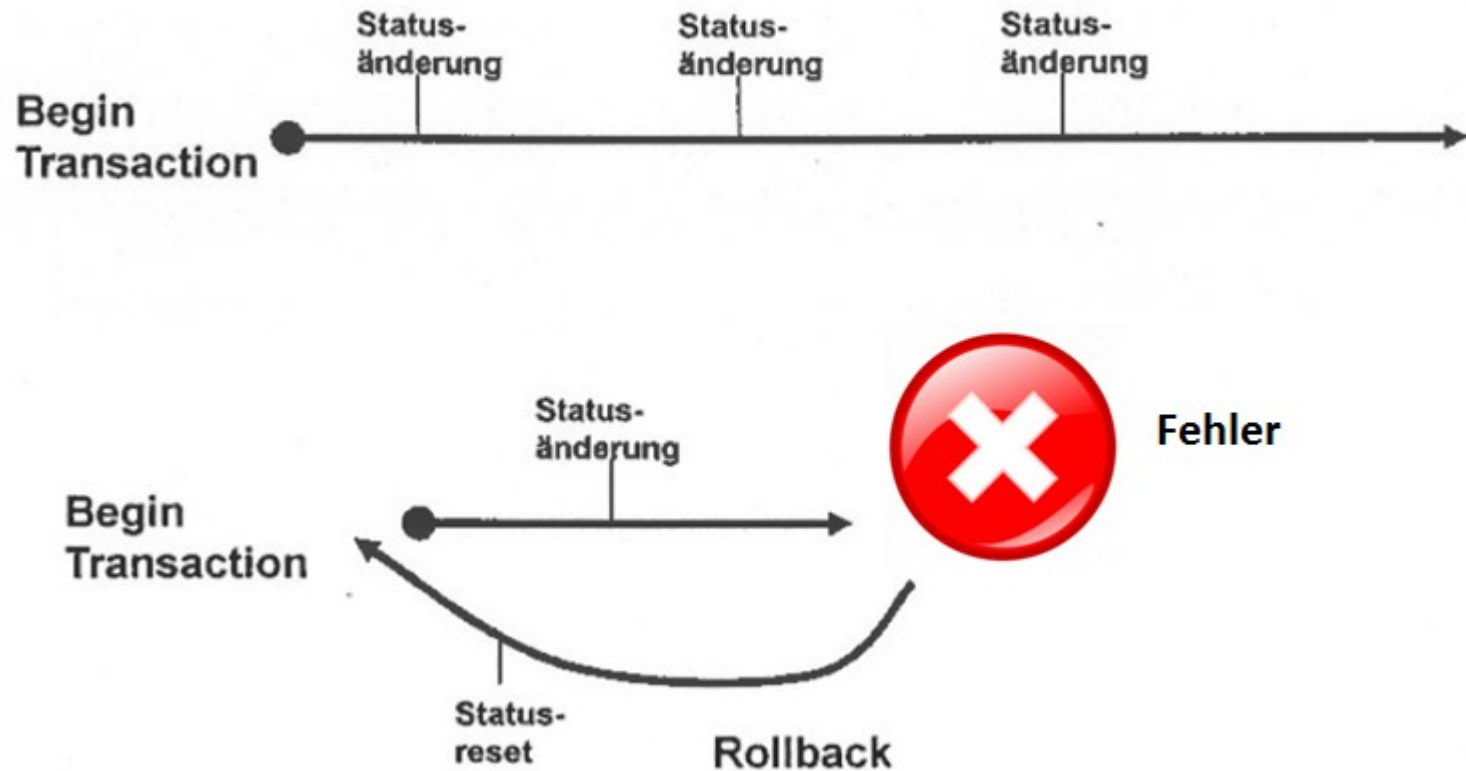
Transaktionen

Transaktionen, Locking

Strategien Transaktionen

Optimistic Locking

Pessimistic Locking



Transaktionseigenschaften – ACID

■ ACID-Eigenschaften

- ◆ **Atomic (atomar)** - eine Transaktion ist Reihe von "primitiven" unteilbaren Operationen. Es werden entweder alle Operationen innerhalb einer Transaktion ausgeführt oder gar keine.
- ◆ **Consistent (konsistent)** - die Änderungen einer Transaktion hinterlassen die Datenbank in einem konsistenten Zustand.
- ◆ **Isolated (isoliert)** - gleichzeitige Transaktionen beeinflussen sich nicht.
- ◆ **Durable (dauerhaft)** - abgeschlossen Transaktion sind dauerhaft.

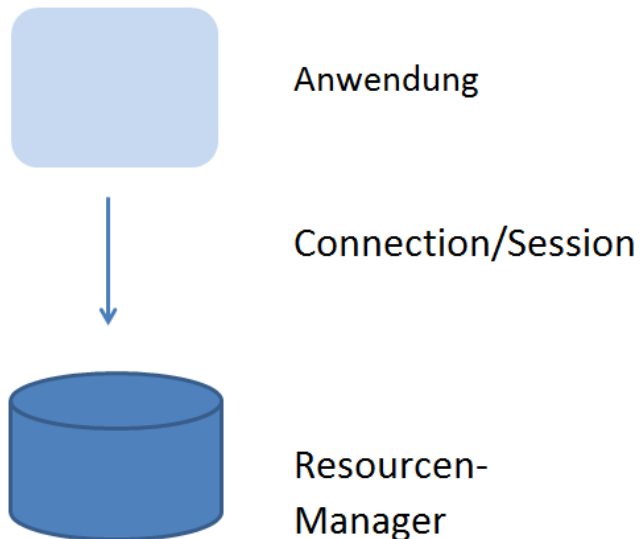
Lokale Transaktionen

Strategien Transaktionen

Optimistic Locking

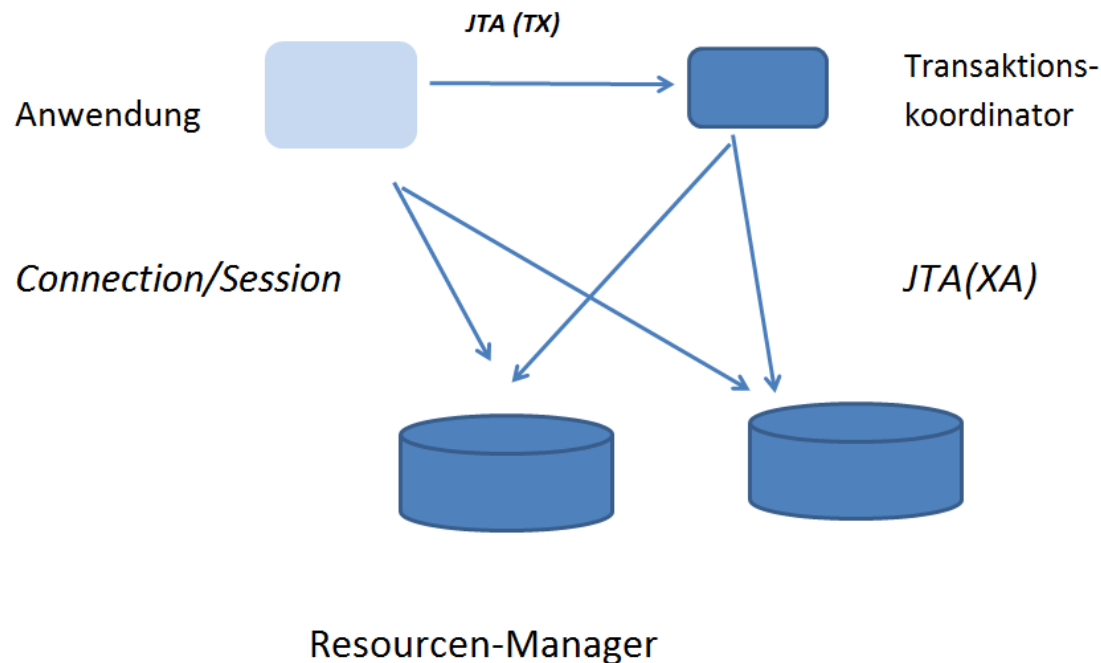
Pessimistic Locking

- Einsatz eines Ressourcen-Managers
- "Out of the Box" Unterstützung bei JDBC, JMS
- Performant



Globale Transaktionen

- Einsatz mehrerer Ressourcen-Manager möglich
- Transaktionskoordinator und Two Phase Commit notwendig



Phänomene bei konkurrierendem Zugriff

Transaktionen, Locking

Strategien Transaktionen

Optimistic Locking

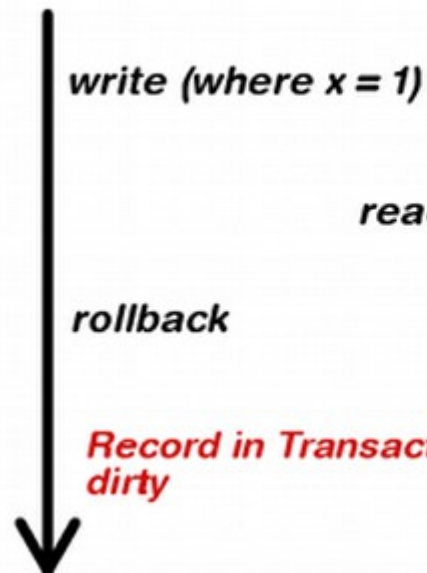
Pessimistic Locking

- Dirty Reads
- Non-repeatable Reads
- Phantom Read

Dirty Reads

- Innerhalb einer Transaktion (A) wird ein Datensatz verändert.
- Dieser veränderte Datensatz wird innerhalb einer zweiten Transaktion (B) gelesen, bevor A abgeschlossen wurde (Commit).
- Wird nun A mit einem Rollback abgebrochen, arbeiten die Operationen innerhalb von B mit einem ungültigen Wert.

Transaction A



Transaction B

read (where $x = 1$)



Record in Transaction B is now dirty

Non-repeatable Reads

- Innerhalb einer Transaktion (A) wird ein bestimmter Datensatz gelesen.
- Direkt nach dem Lesen von A, aber noch vor einem Commit von A, verändert eine zweite Transaktion (B) diesen Datensatz und wird mit einem Commit beendet.
- Liest nun A diesen Datensatz erneut, wird ein anderer Inhalt zurückgegeben als zu Beginn, obwohl aus der Sicht von A der Datensatz nicht verändert wurde.

Transaction A

read (where x = 1)

write (where x = 1)

commit

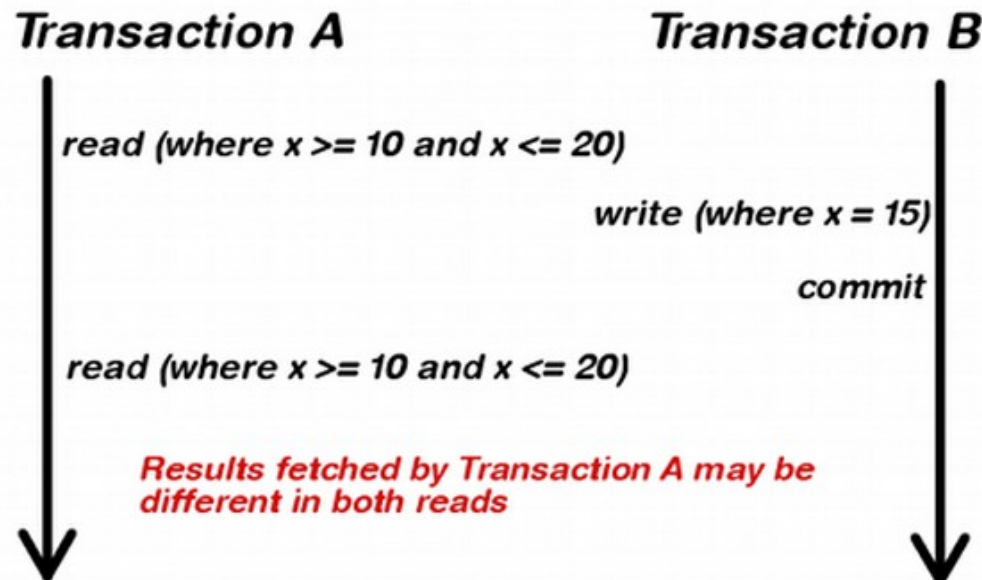
read (where x = 1)

*Transaction A might get a record
with different values between
reads*













Transaction B

Phantom Read

- Innerhalb einer Transaktion (A) wird eine Abfrage an die Datenbank gestellt, welche eine bestimmte Anzahl von Ergebnisdatensätzen liefert.
- Eine zweite Transaktion (B) ändert den Inhalt der Datenbank, indem es neue Datensätze einfügt und wird mit einem Commit beendet.
- Führt nun A die gleiche Abfrage erneut aus, so werden mehr Ergebnisdatensätze als beim ersten Mal gefunden.



Isolation Level

	Dirty Read	Non-repeatable Read	Phantom Read
Read uncommitted			
Read committed			
Repeatable Read			
Serializable			

Transaktionen in Java SE

- Abstraktion um JDBC-Transaktionen

```
EntityTransaction tx = em.getTransaction() ;
tx.begin();
try {
    // em...
    tx.commit();
} catch (Exception e) {
    tx.rollback() ;
}
finally {
    if (em != null) {
        em.close() ;
    }
}
```

Container-Managed Transactions (CMT)

- Standard bei EJB-Session-Beans

@Stateless

```
public class UserService {
```

```
    @PersistenceContext
```

```
    EntityManager em;
```

```
    @TransactionAttribute(
```

```
        TransactionAttributeType.REQUIRED)
```

```
    public void save (String username, ...) {
```

```
        em.persist (new User (username, ...));
```

```
    }
```

```
    . . .
```

```
}
```

Entity-Managed Transactions (EJB)

- JTA Transaktionen : javax.transaction.UserTransaction

@Stateless

```
public class UserService {  
    @Resource UserTransaction tx;  
    public void save (String username, ...) {  
        tx.begin();  
        try {  
            em.persist (new User (username,  
                                ...));  
            tx.commit();  
        } catch (Exception e) {  
            tx.rollback() ;  
        }  
    }  
}
```

Spring Transaction Managers

- **DataSourceTransactionManager**
 - ◆ Transaktion nur über eine Datenbank
- **HibernateTransactionManager**
 - ◆ Beim Einsatz von Hibernate
- **JdoTransactionManager**
 - ◆ Bei JDO als Persistenzmechanismus
- **JtaTransactionManager**
 - ◆ Nutzt JTA TransactionManager für verteilte Transaktionen
- **PersistenceBrokerTransactionManager**
 - ◆ Beim Einsatz von Apache OJB

Transaction Attributes in Spring

- Propagation Verhalten
 - ◆ Beschreibt die Grenzen der Transaktion

Transaktionsattribut	Vorher...	Nachher...
Required	none	T2
	T1	T1
RequiresNew	none	T2
	T1	T2
Mandatory	none	error
	T1	T1
NotSupported	none	none
	T1	none
Supports	none	none
	T1	T1
Never	none	none
	T1	error
Nested	none	T1
	T1	T1.1

Konfiguration in Spring

- **Deklarative Konfiguration (komplett XML)**
 - ◆ Kombination aus XML-Namensräumen AOP und TX
 - ◆ AOP: Deklaration von Pointcut und Advisor
 - ◆ TX: Deklaration der Advices und Transaktionsattribute
- **Konfiguration mittels Annotations**
 - ◆ @Transactional
 - ▶ Transaktionsgrenze
 - ▶ Isolationslevel
 - ▶ Nur Lesen?
 - ▶ Rollback-Verhalten (bei bestimmten Exceptions)
 - ◆ Bei AOP Proxies nur auf öffentliche Methoden anwendbar
 - ◆ Aktivierung mittels XML Konfigurationseintrag:
`<tx:annotation-driven />`

Konfiguration (Spring >= 2.0)

Strategien Transaktionen

Optimistic Locking

Pessimistic Locking

```
<aop:config>
  <aop:pointcut id="servicePointcut"
    expression="execution(* de.example.fspring.service..*.*(..))" />
  <aop:pointcut id="daoPointcut"
    expression="execution(* de.example.fspring.dao..*.*{..}))" />
  <aop:advisor advice-ref="serviceAdvice" pointcut-ref="servicePointcut" />
  <aop:advisor advice-ref="daoAdvice" pointcut-ref="daoPointcut" />
</aop:config>
<tx:advice id="serviceAdvice" transaction-manager="txManager">
  <tx:attributes><tx:method name="*" propagation="REQUIRED"/>
</tx:attributes>
</tx:advice>
<tx:advice id="daoAdvice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="get*" propagation="MANDATORY" read-only="true" />
    <tx:method name="*" propagation="MANDATORY" />
  </tx:attributes>
</tx:advice>

<bean id="txManager" class="...DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource" />
</bean>
```

Konfiguration mit Annotation (Spring > 3.0)

```
@Transactional(propagation=Propagation.REQUIRED)
public class CardServiceImpl implements ICardService {
    public List getCards() {
        . . .
    }
}
```

```
@Transactional(propagation=Propagation.MANDATORY)
public class CardDAOImpl implements IDAOImpl {
    @Transactional (readOnly=true)
    public List getCards() {
        . . .
    }
}
```

```
<tx:annotation-driven transaction-
    manager="transactionManager" />
```

Agenda

Transaktionen, Locking

Strategien Transaktionen

Optimistic Locking

Pessimistic Locking

- **Last-Wins**

- ◆ Letzte Transaktion schreibt in die DB

- **Optimistic:**

- ◆ Nachteil: Implementierung muß im Fehlerfall reagieren
- ◆ Vorteil: Performance
- ◆ Sinnvoll wenn konkurrierende Zugriffe auf ein- und denselben Datensatz selten bis gar nicht auftreten.

- **Pessimistic**

- ◆ Deadlocks möglich
- ◆ Konsistenz ist besser gewährleistet
- ◆ Einsatz bei wenig Änderungen

Konfiguration Versionsspalte

Strategien Transaktionen

Optimistic Locking

Pessimistic Locking

@Version

```
private Date version; // Timestamp
```

oder

@Version // Zahlenwert

```
public long getVersion() {  
    return version;  
}  
  
public void setVersion(long v){  
    this.version = v;  
}
```

Mehr Möglichkeiten in Hibernate

- Strategien für optimistic locking mit **optimistic-locking**
 - ◆ version prüft version/timestamp Spalte
 - ◆ all prüft alle Spalten auf Veränderung
 - ◆ dirty prüft veränderte Spalten
 - ◆ none benutzt kein optimistic locking
- Eine Spalte für Versionierung wird empfohlen
 - ◆ Spalte mit Versionsnummer
 - ◆ Spalte mit Timestamp

Konfiguration der Versioning

```
private long version;

public long getVersion() {
    return version;
}

public void setVersion(long v){
    this.version = v;
}

<version column="version_column" name="propertyName"
    type="typename" unsaved-value="null|negative|undefined"/>
<timestamp column="timestamp_column" name="propertyName"
    unsaved-value="null|undefined"/>
```

- Die folgenden *LockModeTypes* stehen zur Verfügung:
 - ◆ **LockModeType.NONE**: Keine Blockierung verwenden
 - ◆ **LockModeType.OPTIMISTIC**: Optimistic Locking verwenden; das ist die Voreinstellung
 - ▶ *perform a version check on locked Entity before commit, throw an OptimisticLockException if Entity version mismatch*
 - ◆ **LockModeType.OPTIMISTIC_FORCE_INCREMENT**: Wie *OPTIMISTIC*, allerdings wird das Versionsattribut inkrementiert bzw. aktualisiert, ohne dass zwingend eine Änderung der Daten vorgenommen wurde
 - ▶ *perform a version check on locked Entity before commit, throw an OptimisticLockException if Entity version mismatch, force an increment to the version at the end of the transaction, even if the entity is not modified.*



API's for Locking

- APIs's um ein Lock für eine Entity zu spezifizieren:
- EntityManager Methoden: lock, find, refresh
- Query Methode: setLockMode
- NamedQuery Annotation: lockMode element

Optimistic Locking Beispiel

Version Updated when transaction commits

OptimisticLockException if mismatch

```
tx1.begin();
//Joe's employee id is 5
//e1.version == 1
e1 = findPartTimeEmp(5);

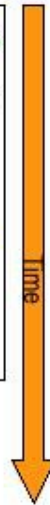
//Joe's current rate is $9
e1.raise(2);

tx1.commit();
//e1.version == 2 in db
//Joe's rate is $11
```

```
tx2.begin();
//Joe's employee id is 5
//e1.version == 1
e1 = findPartTimeEmp(5);

//Joe's current rate is $9
if(e1.getRate() < 10)
    e1.raise(5);

//e1.version == 1 in db?
tx2.commit();
//Joe's rate is $14
//OptimisticLockException
```



https://blogs.oracle.com/carolmcdonald/entry/jpa_2_0_concurrency_and

- **Transaktion links** commitet als erste und führt beim Update auch zu einer Versionsnummererhöhung.
- **Transaktion rechts** wirft ein OptimisticLockException, da die Versionsnummer nicht mit der ursprünglich gelesenen übereinstimmt.
 - ◆ Dies führt zu einem Roll Back.

OPTIMISTIC (READ)

Locking

Transaktionen, Locking

Strategien Transaktionen

Optimistic Locking

Pessimistic Locking

- **Änderung** des **employee** soll nicht commitbar sein, falls **Department** nach dem Lesen geändert wurde.
- Hier kommt **OPTIMISTIC lock** zum Einsatz:
 - ◆ **em.lock(dep, OPTIMISTIC).**
- In Transaktion 2 wird vor dem Commit ein Versionscheck auf dem **dep** Entity durchgeführt.
 - ◆ **OptimisticLockException**, da das Versionsattribut von **dep** höher ist als zum Zeitpunkt des Lesens von **dep**.

Optimistic Lock (READ):

perform a **version check** on entity **before commit**

OptimisticLockException if mismatch

```
tx1.begin();
dep = findDepartment(depId);

//dep's original name is
//"Eng"
dep.setName("MarketEng");
tx1.commit();

tx2.begin();
emp = findEmp(eId);
dep = emp.getDepartment();
em.lock(dep, OPTIMISTIC);
if (dep.getName().equals("Eng"))
    emp.raiseByTenPercent();

//Check dep version in db
tx2.commit();
//ei gets the raise he does
//not deserve
//Transaction rolls back
```



https://blogs.oracle.com/carolmcdonald/entry/jpa_2_0_concurrency_and

OPTIMISTIC_FORCE_INCREMENT

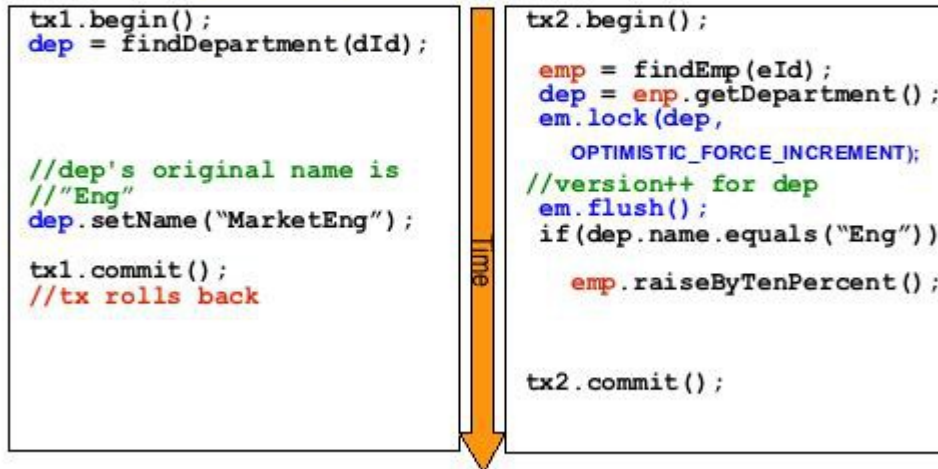
(write)

OPTIMISTIC_FORCE_INCREMENT (write) lock:

perform a **version check** on entity

OptimisticLockException if mismatch

increment **version** before commit



https://blogs.oracle.com/carolmcdonald/entry/jpa_2_0_concurrency_and

- Transaktion2 rechts möchte sicher sein, das dep name nicht während der Transaktion geändert wird.
- Der Aufruf von em.flush() inkrementiert das **dep's** Versionsattribut in der Datenbank.

Aufgabe



Transaktionen, Locking

Strategien Transaktionen

Optimistic Locking

Pessimistic Locking

- Aufgabe 13 (optional):
- Optimistic Locking

Agenda

Transaktionen, Locking

Strategien Transaktionen

Optimistic Locking

Pessimistic Locking

Explizites (Pessimistic) Locking

- Sperren des Datensatzes für andere Bearbeiter direkt in der Datenbank
- PESSIMISTIC_READ, PESSIMISTIC_WRITE, PESSIMISTIC_FORCE_INCREMENT
- // keine dirty oder nonrepeatable Reads möglich
`manager.lock(t1, LockModeType.PESSIMISTIC_WRITE);`
- Lock Timeout ist nur ein Hinweis an JPA-Provider (Umsetzung ist optional)

```
TypedQuery<User> u = em.createQuery("SELECT ...");  
q.setLockMode(LockModeType.PESSIMISTIC_FORCE_INCREMENT);  
q.setHint("javax.persistence.lock.timeout", 5000) ;
```

- ◆ **LockModeType.PESSIMISTIC_READ**: Der Eintrag in der Tabelle wird mit einem **Shared Lock** versehen
 - ▶ *lock the database row when reading*
- ◆ **LockModeType.PESSIMISTIC_WRITE**: Der Tabelleneintrag wird mit einem **Exclusive Lock** gesperrt
- ◆ **LockModeType.PESSIMISTIC_FORCE_INCREMENT**: Wie *PESSIMISTIC_WRITE*, allerdings mit gleichzeitiger Erhöhung bzw. Aktualisierung des Versionsattributs
 - ▶ *lock the database row when reading, force an increment to the version at the end of the transaction, even if the entity is not modified.*



Explizites (Pessimistic)

Locking (Hibernate)

- bei Session mittels explizitem lock() oder beim Laden
- WRITE, UPGRADE, UPGRADE_NOWAIT, READ, NONE

// keine dirty oder nonrepeatable Reads möglich

```
session.lock(t1, LockMode.READ);
```

```
session.load(t2, 5L, LockMode.WRITE);
```

Pessimistic Locking Beispiele

- Lesen der Entity / Locking später

```
//Read then lock:  
Account acct = em.find(Account.class, acctId);  
// Decide to withdraw $100 so lock it for update  
em.lock(acct, PESSIMISTIC);  
int balance = acct.getBalance();  
acct.setBalance(balance - 100);
```

Lock after read, risk
stale, could cause
**OptimisticLock
Exception**

https://blogs.oracle.com/carolmcdonald/entry/jpa_2_0_concurrency_and

- Lesen der Entity / gleichzeitiges Locking

```
//Read and lock:  
Account acct = em.find(Account.class, acctId, PESSIMISTIC);  
// Decide to withdraw $100 (already locked)  
int balance = acct.getBalance();  
acct.setBalance(balance - 100);
```

Locks longer,
could cause
bottlenecks,
deadlock

https://blogs.oracle.com/carolmcdonald/entry/jpa_2_0_concurrency_and

Pessimistic Locking Beispiele

- Lesen der Entity / Refresh später mit Lock

```
// read then lock and refresh
Account acct = em.find(Account.class, acctId);
// Decide to withdraw $100 - lock and refresh
em.refresh(acct, PESSIMISTIC);
int balance = acct.getBalance();
acct.setBalance(balance - 100);
```

https://blogs.oracle.com/carolmcdonald/entry/jpa_2_0_concurrency_and

- Trade-offs:
- Je länger Lock gehalten wird, desto grösser das Risiko einer schlechteren Skalierbarkeit / Deadlocks.
- Umso später der Lock erfolgt, desto grösser das Risiko veralteter Daten, die zu einer Optimisticklockexception führen können, falls Entity nach dem Lesen, aber vor dem Lock geändert wird.

Locking Hints

- Das Verhalten von Pessimistic Locking kann mithilfe folgender Hints beeinflusst werden:
 - ◆ Mit ***javax.persistence.lock.timeout*** kann eine maximale Wartezeit in Millisekunden angegeben werden.
 - ▶ Ohne Angabe des Hints gilt eine evtl. für die DB konfigurierte Timeoutzeit als Vorgabe.
 - ◆ Mit ***javax.persistence.lock.scope*** kann bestimmt werden, ob ein *Pessimistic Lock* mehr als nur das jeweils bearbeitete Objekt blockiert.



- Die folgenden Werte können angegeben werden:
 - ◆ *PessimisticLockScope.NORMAL*: Nur die zum Objekt gehörenden Tabelleneinträge werden blockiert.
 - ◆ Das ist die Voreinstellung.
 - ◆ *PessimisticLockScope.EXTENDED*: Es werden zusätzlich die Einträge der Tabellen blockiert, die Element Collections darstellen.
 - ▶ Zudem erstreckt sich die Blockierung auch auf die Einträge in Verknüpfungstabellen, die zu Relationen gehören, deren Eigentümer das bearbeitete Objekt ist.
 - ▶ Die dadurch referenzierten Einträge werden allerdings nicht blockiert.

Locking Hints

- Hints
 - ◆ können als Parameter den folgenden Methoden übergeben werden:
 - ▶ **EntityManager.find**
 - ▶ **EntityManager.lock,**
 - ▶ **EntityManager.refresh**
 - ▶ **Query.setHint**
 - ◆ können bei der Deklaration einer **Named Query** eingetragen werden.
 - ◆ **javax.persistence.lock.timeout** kann zudem als Property im Deskriptor persistence.xml eingetragen werden.



Locking Hints

- Achtung:
 - ◆ Hints sind nur Hinweise, die vom Provider nicht beachtet werden müssen.
 - ◆ Für eine portable Anwendung sollten Sie auf die Hints nicht angewiesen sein.





- Demo 6: Pessimistic Locking (Projekt: 09-Locking)