Let’s master Hibernate!

Michał Żmuda
Piotr Turek

Agenda

- Transactions
  - Why do you need them - example
  - ACID
  - Building complex bussiness logic - how to use propagation
  - Transaction Manager - go declarative!
  - Propagation levels

Agenda

- Review of JPA compatibility
  - What is it about Hibernate vs JPA - a bit of history
  - Interoperobality and replacements
  - Hibernate vs JPA - which one to choose?

Agenda

- Things to consider designing ORM software
  - Eager fetching performance
    - case showing disastrous performance
    - actual/typical overheat
  - Lazy initialization pitfalls
  - Caching - When, How, Why, plus examples
  - Cascades - how not to hurt yourself
  - Golden rules of hibernate based orm layer design

Transactions - none

```
OrdersDao ordersDao = (OrdersDao) ctx.getBean("ordersDao");
OrderDetailsDao orderDetailsDao = (OrderDetailsDao) ctx.getBean("orderDetailsDao");
OrdersEntityFactory ordersEntityFactory = (
        OrdersEntityFactory) ctx.getBean("ordersEntityFactory");

OrdersEntity randomOrder = ordersEntityFactory.createRandomOrder();
ordersDao.create(randomOrder);

OrderDetailsEntity detailsEntity = new OrderDetailsEntity();
detailsEntity.setOrderid(randomOrder.getOrderid());
setOtherOrderDeatilsFields(orderDetailsDao, detailsEntity);
orderDetailsDao.create(detailsEntity);//what happens when if fails?
```

- imagine that creating OrderDetails fails
  (excercise: make it fail! tip: violate constraint)
- we have only Order record but we do not have any data what send anf for how much
- how to recover from such failure
- solution: perform both creates or any at all

Transactions

```
// session created by openSession() will be used by dao's - see GenericDao!
Transaction transaction = sessionFactory.openSession().beginTransaction();
OrdersEntity randomOrder = ordersEntityFactory.createRandomOrder();
ordersDao.create(randomOrder);

OrderDetailsEntity detailsEntity = new OrderDetailsEntity();
detailsEntity.setOrderid(randomOrder.getOrderid());
setOtherOrderDeatilsFields(orderDetailsDao, detailsEntity);
orderDetailsDao.create(detailsEntity);
transaction.commit();
```

- now sequence of creates is protected by orm transaction mechanism
- transactions provide easy to use yet powerful help in implementing business logic
- they are analogous to database transactions
- they follow ACID acronym

Transactions

- are limited to one seesion
  see session opened in example and how DAOs are designed
- may be used by Transaction Manager
  discussed later - manual management better serves examples
  TM used by annotating methods @Transactional
- manually handled by struct like:

```
Transaction transaction = null;
try {
    transaction = session.beginTransaction();
    //operations
    transaction.commit();
} catch (Exception e) {
    if (transaction != null) {
        transaction.rollback();
    }
}
```

ACID - Atomicity

- previous example shows need of atomicity
- grouping set of changes in transaction guarantees that they all be executed or any of them at all
- custom fallback policies may be handled with savepoints
- savepoint is used to group changes into atomic groups inside an transaction
- savepoints are provided by Transacion Manager which will be discussed later
- question: do you see any application of savepoints in our northwind example?

ACID - Consistency

- imagine that in db exists trigger that order has at most 10 order detail records
- when it would be violated inside a transaction, it would be all rollbacked
- this guarantees consistency - all data after transaction fulfill validation rules

optional excersise: try adding such trigger to the db and prepare runner violating it inside transaction

ACID - Isolation

```
executor.submit((Runnable) () -> {
        try {
            Thread.currentThread().sleep(4000);
            printTotalValueStatistic(orderDetailsDao,ordersDao);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
});
printTotalValueStatistic(orderDetailsDao, ordersDao);
OrdersEntity randomOrder = ordersEntityFactory.createRandomOrder();
ordersDao.create(randomOrder);
Thread.currentThread().sleep(6000);//simulating time window
OrderDetailsEntity detailsEntity = new OrderDetailsEntity();
detailsEntity.setOrderid(randomOrder.getOrderid());
setOtherOrderDeatilsFields(orderDetailsDao, detailsEntity);
orderDetailsDao.create(detailsEntity);
printTotalValueStatistic(orderDetailsDao,ordersDao);
```

- do you see what might occur?
- do you believe that without sleep() such situation won't never happen?

ACID - Isolation

```
Total value of 1.1200001376859595E13 is in 858 orders
Total value of 1.1200001376859595E13 is in 859 orders
Total value of 1.6800001376859595E13 is in 859 orders
```

- middle result was generated when order was already create but it’s details not
- when statistics and order addition executed in different transactions it is guanranteed that changes from one transaction won’t be visible to another until it’s completed

  (this is overall description - isolation leevels later)

- question: what middle result will look like if transactions were used?
- excercise: introduce transactions

ACID - Isolation

for better isolation we pay with performance - choose of isolation level is given

- READ UNCOMMITTED - no protection, see ‘dirty reads’
- READ COMMITTED - when reading row you might get different values due update
  - see ‘non-repeatable reads’
- REPEATABLE READ - when selecting range of rows you might get different result sets
  - see ‘phantom read’
- SERIALIZABLE - whole tables are locked - safest

- isolation levels may be set declaratively with Transaction Manager or programaticly with TransactionTemplates
- typically we annotate method to be handled by TM with annotation where we can set isolation level

```
@Transactional(isolation = Isolation.SERIALIZABLE)
```

question: which level is required to make our example work?

ACID - Durability

- durability means that after transaction completion all changes made within it are now persisted
- this is core requirement for persistence layer and won't be disscused further

Transaction Manager

- allows to declaratively use transactions
- configured once in applicationContext.xml

```
<tx:annotation-driven/>
<bean id="transactionManager" class=
        "org.springframework.orm.hibernate4.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

- by annotating method with @Transactional we are guaranteed that its instructions will be executed in transaction
- it may be new transaction or one already opened in method invoking our method
- this behavoiur is controlled by propagation levels

TM - propagation levels

```
@Service
public class OrdersValueService {
    @Transactional(propagation = Propagation.NESTED
    public void printTotalValueStatistic() {
```

```
@Transactional
public void addNewOrder() throws InterruptedException {
    OrdersEntity randomOrder = ordersEntityFactory.createRandomOrder();
    ordersDao.create(randomOrder);
    ordersValueService.printTotalValueStatistic(); //during
    OrderDetailsEntity detailsEntity = new OrderDetailsEntity();
    detailsEntity.setOrderid(randomOrder.getOrderid());
    setOtherOrderDeatilsFields(orderDetailsDao, detailsEntity);
    orderDetailsDao.create(detailsEntity);
}
```

```
public static void main(String[] args) throws InterruptedException {
    ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("META-INF/a
    ((OrdersValueService)ctx.getBean("ordersValueService")).printTotalValueStatistic();
    ((PropagationLevelsRunner) ctx.getBean("propagationLevelsRunner")).run();
    ((OrdersValueService)ctx.getBean("ordersValueService")).printTotalValueStatistic();
}
```

TM - propagation levels

- execution of example yelds

```
Total value of 5.6000013768635 91E12 is in 861 orders
Total value of 5.6000013768635 91E12 is in 862 orders
Total value of 1.1200001376863 59E13 is in 862 orders
```

- this result is similar to one we would get without transactions but we use them (each method is using TM thanks to annotation)
- see propagetion level - this particular used in the exapmle causes that actually we have only one transaction!

TM - propagation levels

MANDATORY Support a current transaction, throw an exception if none exists.

NESTED Execute within a nested transaction if a current transaction exists, behave like PROPAGATION_REQUIRED else.

NEVER Execute non-transactionally, throw an exception if a transaction exists.

NOT_SUPPORTED Execute non-transactionally, suspend the current transaction if one exists.

REQUIRED Support a current transaction, create a new one if none exists.

REQUIRES_NEW Create a new transaction, suspend the current transaction if one exists.

SUPPORTS Support a current transaction, execute non-transactionally if none exists.

question: which levels would fix our example? try it out!

Hibernate vs JPA

What JPA is?

- a specification

What JPA isn't?

- an implementation

What Hibernate is?

- an implementation

What Hibernate isn't?

- a specification

Hibernate provides a superset of JPA functionality!

… just like a musician can play more than one song or symphony

Hibernate vs JPA

Hibernate vs JPA

equals

SessionFactory vs EntityManagerFactory

equals

implementation coupling vs implementation independence

equals

powerfull features vs limited functionality

Hibernate vs JPA

Hibernate	JPA
SessionFactory	EntityManagerFactory
Session	EntityManager
sessionFactory.getCurrentSession(). [method]()	entityManager.[method]()
saveOrUpdate()	persist()
Query.setInteger/String/Entity()	Query.setParameter()
list()	getResultList()
uniqueResult()	getSingleResult()
uniqueResult() returns null	getSingleResult() throws NoResultException
CriteriaQueries – yes	CriteriaQueries – no

- You can unwrap EntityManager to obtain Session
  - entityManager.getDelegate()
  - entityManager.unwrap(Session.class)

Hibernate vs JPA - which to choose

Advantages of using JPA:

- designing and coding to API
- standardization
- increased portability (!)

Maybe one day you will want to change the provider to something different than Hibernate?

… quite likely however, that will never ever happen

Therefore …

Hibernate vs JPA - which to choose

… considering:

- much better exception translation
- more power, more features

Use Hibernate's Session and Session Factory

Good compromise is to use **JPA annotations with Hibernate Session**

… but, most probably you do that anyway already

Eager fetch performance

simple example of performance:
(required service can be found in project or implemented more elegant on your own)

```
Collection<ProductsEntity> products = service.generateProducts(productsDao, 10000);
OrdersEntity randomOrder = prepareOrdersEntityWithOrderDetails(ordersDao, products,
service.createOrderDetailsFromOrderEntity(randomOrder);
long time = System.nanoTime();
System.out.println(ordersDao.get(randomOrder.getOrderid()).getOrderdate());
System.out.println("Elapsed " + (System.nanoTime() - time) + "ns");
```

```
Hibernate: select ordersenti0_.orderid as orderid1_7_4_, ordersenti0
2014-05-13
Elapsed 25468826
```

by default hibernate lazy initialisation, change to eager fetching

```
@OneToMany(mappedBy = "ordersByOrderid", fetch = FetchType.EAGER)
public Collection<OrderDetailsEntity> getOrderDetailsesByOrderid() {
    return orderDetailsesByOrderid;
}
```

```
Hibernate: select ordersenti0_.orderid as orderid1_7_8_, ordersenti0_.customerid a
2014-05-13
Elapsed 2454467513ns
```

Eager fetch performance

- that’s 100 times slower!
- moreover it’s over 2s to read simple value!
- and it could be more than related records 10000...

- don’t want ever to use eager fetchnig?

  you may be wrong - it prooves quite usefull when you have relations @OneToOne or @ManyToOne and it’s used by default with such relations by default by JPA

question: if forbidden to change fetching type, how you would tweak reading single value from the db?

Lazy initialization hell

Eager fetching is bad.

Lazy fetching is treacherous.

```
@OneToMany(mappedBy = "productsByProductid", fetch = FetchType.LAZY)
public Collection<OrderDetailsEntity> getOrderDetailsesByProductid() { return orderDetailsesByProductid; }
```

```
private void run() {
    final ProductsEntity product = lazyInitHellService.extractProduct();
    displayOrderDetails(product);
}
```

```
@Transactional
public ProductsEntity extractProduct() {
    return productsDao.getAll().get(0);
}
```

```
private void displayOrderDetails(ProductsEntity product) {
    final Collection<OrderDetailsEntity> orderDetails = product.getOrderDetailsesByProductid();
    final OrderDetailsEntity firstDetail = orderDetails.iterator().next();
    System.out.println("Discount for a product: " + firstDetail.getDiscount());
}
```

Simple. What can possibly go wrong… (try it out)

Lazy initialization hell

… well,

```
Exception in thread "main" org.hibernate.LazyInitializationException: failed to lazily initialize a
 collection of role: pl.agh.turek.bazy.hibernate.model.ProductsEntity.orderDetailsesByProductid, could
 not initialize proxy - no Session <5 internal calls>
    at pl.agh.turek.bazy.hibernate.runners.LazyInitHellRunner.displayOrderDetails(LazyInitHellRunner
.java:32)
    at pl.agh.turek.bazy.hibernate.runners.LazyInitHellRunner.run(LazyInitHellRunner.java:27)
    at pl.agh.turek.bazy.hibernate.runners.LazyInitHellRunner.main(LazyInitHellRunner.java:22)
  <5 internal calls>
```

Explanation:

- Hibernate returns a HibernateProxy, not a real ProductEntity
- HibernateProxy instance belongs to a Session that created it
- When transaction ends, session gets closed
- When accessing lazy collection, Hibernate tries to read it lazily
  - but the session had already been closed (!)

Lazy initialization hell

```java
@Transactional
public ProductsEntity extractProductWithOrderDetails() {
    final ProductsEntity product = extractProduct();
    System.out.println("... Product lazy proxy retrieved ...");
    product.getOrderDetailsesByProductid().size();
    return product;
}
```

We need to “unpack” the collection while the session is open

- we can do that f.e. by calling size() method on it

```
... Product lazy proxy retrieved ...
Hibernate: select orderdetai0_.productid as producti1_8_6_, orderdetai0_.productid as producti1_6_6_, orderdetai0_.orderid as
  orderdetai0_.discount as discount3_6_5_, orderdetai0_.quantity as quantity4_6_5_, orderdetai0_.unitprice as unitpric5_6_5_,
  ordersenti1_.employeeid as employee3_7_0_, ordersenti1_.freight as freight4_7_0_, ordersenti1_.orderdate as orderdat5_7_0_,
  ordersenti1_.shipcity as shipcity8_7_0_, ordersenti1_.shipcountry as shipcoun9_7_0_, ordersenti1_.shipname as shipnam10_7_0_
  ordersenti1_.shippostalcode as shippos12_7_0_, ordersenti1_.shipregion as shipreg13_7_0_, customerse2_.customerid as custome
  customerse2_.companyname as companyn4_3_1_, customerse2_.contactname as contactn5_3_1_, customerse2_.contacttitle as contact
  customerse2_.phone as phone9_3_1_, customerse2_.postalcode as postalc10_3_1_, customerse2_.region as region11_3_1_, employee
  as birthdat3_4_2_, employeese3_.city as city4_4_2_, employeese3_.country as country5_4_2_, employeese3_.reportsto as reports
  employeese3_.hiredate as hiredate8_4_2_, employeese3_.homephone as homephon9_4_2_, employeese3_.lastname as lastnam10_4_2_,
  as photopa13_4_2_, employeese3_.postalcode as postalc14_4_2_, employeese3_.region as region15_4_2_, employeese3_.title as ti
  employeese4_.employeeid as employee1_4_3_, employeese4_.address as address2_4_3_, employeese4_.birthdate as birthdat3_4_3_,
  as reports16_4_3_, employeese4_.extension as extensio6_4_3_, employeese4_.firstname as firstnam7_4_3_, employeese4_.hiredate
```

HibernateProxy is lazily fetching data from database.

Problem Solved :)

Caching - when, how and why

Problem:

- There exist entities that have much **more reads than writes**
  - think of an online shop and ProductsEntity
  - details rarely updated (maybe even never)
  - thousands of clients read data, every minute
- Going to database for that data each time is an overkill
  - and can literally kill your database (and app as well)

Solution:

- Add some caching …
  - which can give some pretty amazing results

Caching - configuration

Hibernate properties:

```
<prop key="hibernate.cache.use_second_level_cache">true</prop>
<prop key="hibernate.cache.provider_class">org.hibernate.testing.cache.CachingRegionFactory</prop>
<prop key="hibernate.cache.region.factory_class">org.hibernate.testing.cache.CachingRegionFactory</prop>
```

- enable second level caching
- set up caching region provider
  - Hibernate's ConcurrentHashMap-based one works ok
- add missing dependency

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-testing</artifactId>
    <version>${hibernate-version}</version>
</dependency>
```

Caching - setting up caches

Selected entities need to be marked as cacheable.

```
@Entity
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
@Table(name = "products", schema = "public", catalog = "northwind")
public class ProductsEntity {
```

Two annotations:

- @Cacheable - marker annotation
- @Cache(...)
  - region(), *The cache region. This attribute is optional, and defaults to the fully-qualified class name of the class, or the qually-qualified role name of the collection.*

Caching - setting up caches

- @Cache(...)
  - include(), *Whether or not to include all properties.*
  - *usage(),*
    - *read-only - entity has read-only access*
    - *read-write - support for writing, requires locking*
    - *nonstrict-read-write - allows writing without locking (avoid)*
    - *transactional - full transactional support*

Read-only caches are to be preferred!

Caching - deadly trap

Lets try an excercise:

- Configure caching for ProductsEntity
- Run LazyInitHellRunner
- Modify data manually in a database (in Products table)
  - For example remove the first row
- Run the runner again

Result:

- Data integrity goes to hell(!)
- which can and most probably will, break your business logic
- … and cost you money

Caching - deadly trap (how to avoid)

Conclusion:

- **NEVER, EVER** modify data externally, when using caching
  - no manual inserts/updates/deletes
  - no external systems modifying db state in the background
- **IF** for some reason such events can occur:
  - **notify** your system about this fact
  - **evict** the caches:

```
@Override
public void purgeAllCaches() {
    final Cache cache = sessionFactory.getCache();
    cache.evictEntityRegions();
    cache.evictCollectionRegions();
    cache.evictDefaultQueryRegion();
    cache.evictQueryRegions();
}
```

Cascades - how not to hurt yourself

Consider following code: (try it out - try to explain results)

```
private void run() {
    final ProductsEntity productsEntity = lazyInitHellService.extractProductWithOrderDetails();
    final OrderDetailsEntityPK pk = cascadeTypeService.modifyFirstOrderDetailsDiscount(productsEntity);
    cascadeTypeService.verifyOrderDetailsDiscount(pk);
}
```

```
@Transactional
public OrderDetailsEntityPK modifyFirstOrderDetailsDiscount(ProductsEntity product) {
    System.out.println("Updating discount of first order detail of product id: " + product.getProductid());
    final Collection<OrderDetailsEntity> orderDetails = product.getOrderDetailsesByProductid();
    final OrderDetailsEntity firstOrderDetail = orderDetails.iterator().next();
    final OrderDetailsEntityPK pk = getOrdersPK(firstOrderDetail);
    final double discount = firstOrderDetail.getDiscount();
    System.out.println("Will try to fetch order detail id:" + pk + " with discount "
            + discount);
    firstOrderDetail.setDiscount(discount + 0.1);
    System.out.println("New discount set to: " + firstOrderDetail.getDiscount());
    productsDao.update(product);
    return pk;
}
```

```
@Transactional
public void verifyOrderDetailsDiscount(OrderDetailsEntityPK pk) {
    final OrderDetailsEntity refetchedOrderDetail = orderDetailsDao.get(pk);
    System.out.println("Order details id " + pk + " has discount " + refetchedOrderDetail.getDiscount());
}
```

Cascades - how not to hurt yourself

The code:

- fetches a product
- adds 0.1 to the discount of the first order
- updates the product
- verifies the value of discount

When you run it without any changes to the model:

```
Updating discount of first order detail of product id: 5
Will try to fetch order detail id:OrderDetailsEntityPK{orderid=10258, productid=5} with discount 0.200000003
New discount set to: 0.300000003
Hibernate: update northwind.public.products set categoryid=?, discontinued=?, productname=?, quantityperunit=?, reord
```

```
Order details id OrderDetailsEntityPK{orderid=10258, productid=5} has discount 0.200000003
```

As you can see, even though the product has been updated, the discount stayed unchanged.

That's because, *by default, no operations are "propagated" to relations*

Cascades - how not to hurt yourself

cascadeType attribute:

- ALL - Cascade all operations
- DETACH - Cascade detach operation
- MERGE - Cascade merge operation
- PERSIST - Cascade persist operation
- REFRESH - Cascade refresh operation
- REMOVE - Cascade remove operation

Lets use it on our ProductEntity:

```
@OneToMany(mappedBy = "productsByProductid", fetch = FetchType.LAZY, cascade = {CascadeType.MERGE, CascadeType.PERSIST})
public Collection<OrderDetailsEntity> getOrderDetailsesByProductid() {
```

Result:

```
Updating discount of first order detail of product id: 7
Will try to fetch order detail id:OrderDetailsEntityPK{orderid=10262, productid=7} with discount 0.0
New discount set to: 0.1

Order details id OrderDetailsEntityPK{orderid=10262, productid=7} has discount 0.1
```

Merge operation has been "propagated" to relations

Cascades - how not to hurt yourself

However:

- Cascading must be used with great care
- … and great deal of thought

Otherwise:

- We can literally implode our database with a single remove()

Thankfully:

- Database constraints can work as a safety measure against such scenario

In general:

- Avoid using *CascadeType.ALL*