Let’s master Hibernate!

Michał Żmuda
Piotr Turek

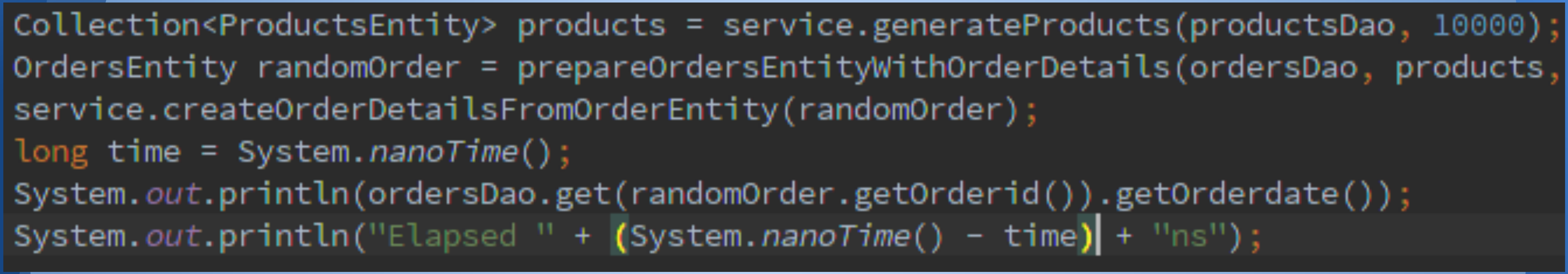
Agenda

- Eager fetch performance
- Cascades do’s and don’ts
- Embracing the power of caching

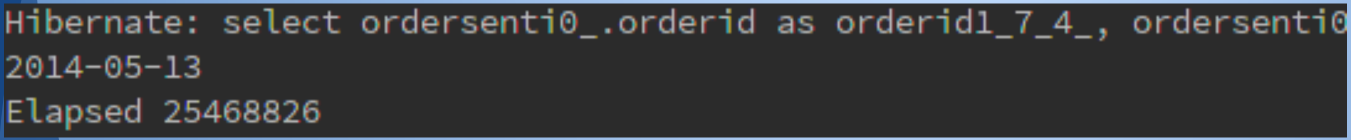
Eager fetch performance

simple example of performance:

(required service can be found in project or implemented more elegant on your own
tip: what ‘batching’ means?)

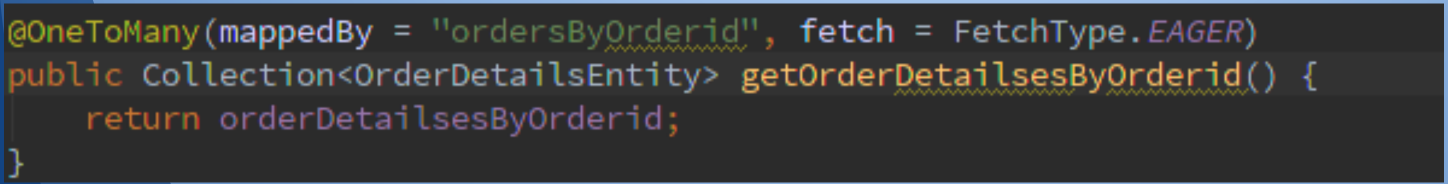


```
Collection<ProductsEntity> products = service.generateProducts(productsDao, 10000);
OrdersEntity randomOrder = prepareOrdersEntityWithOrderDetails(ordersDao, products,
service.createOrderDetailsFromOrderEntity(randomOrder);
long time = System.nanoTime();
System.out.println(ordersDao.get(randomOrder.getOrderid()).getOrderdate());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

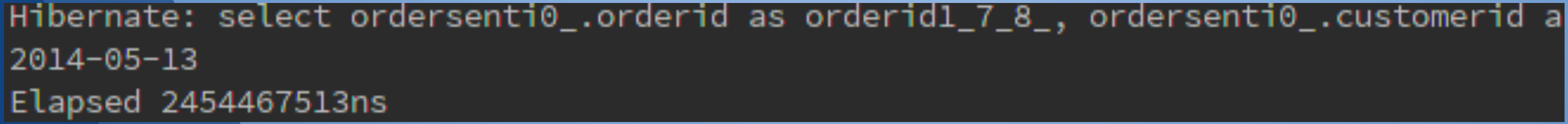


```
Hibernate: select ordersenti0_.orderid as orderid1_7_4_, ordersenti0
2014-05-13
Elapsed 25468826
```

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...

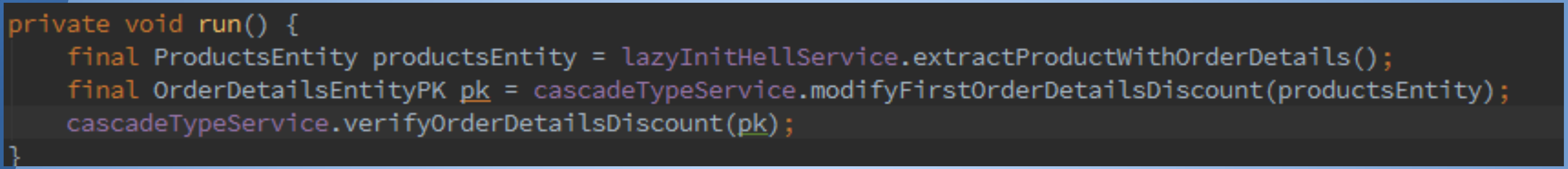
- don’t want ever to use eager fetchnig?

  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

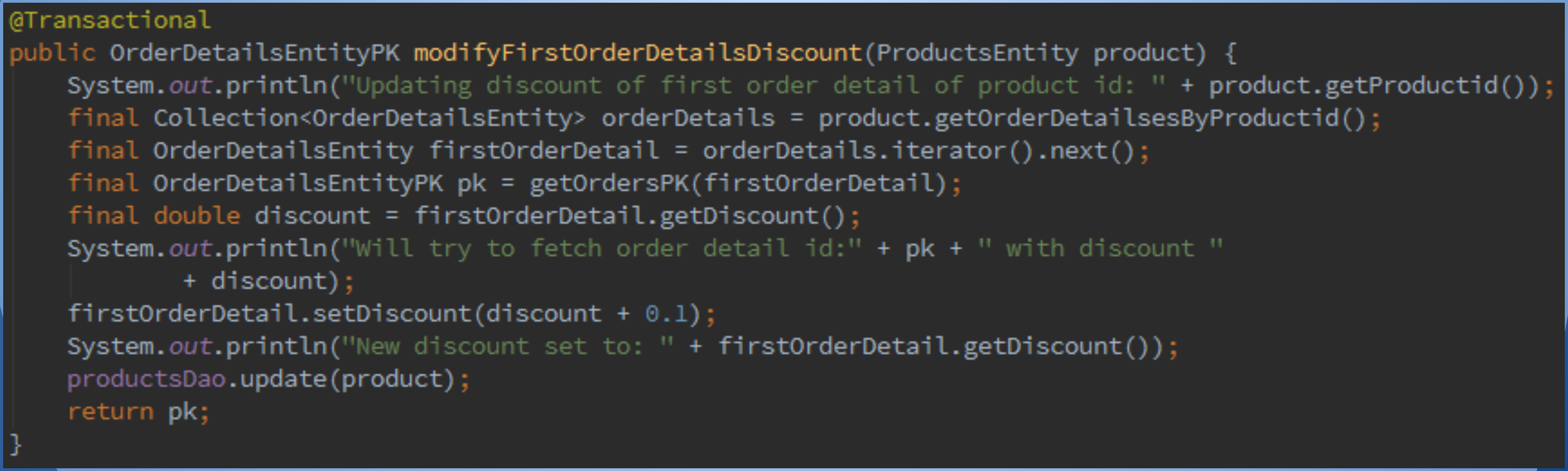
question: if forbidden to change fetching type, how you would tweak reading single value from the db?

Cascades - how not to hurt yourself

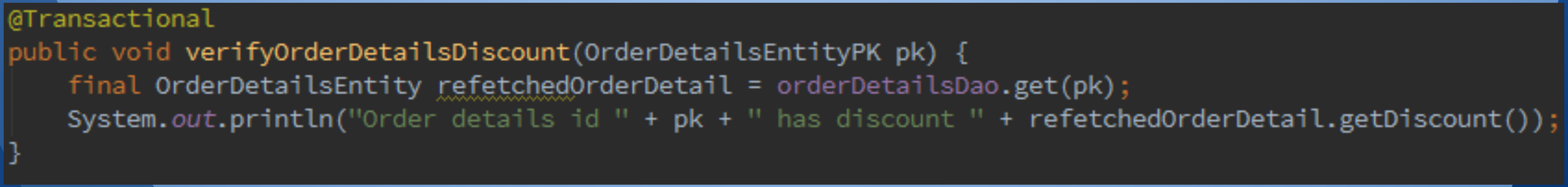
Consider following code: (try it out - try to explain results)



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



```java
@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;
}
```



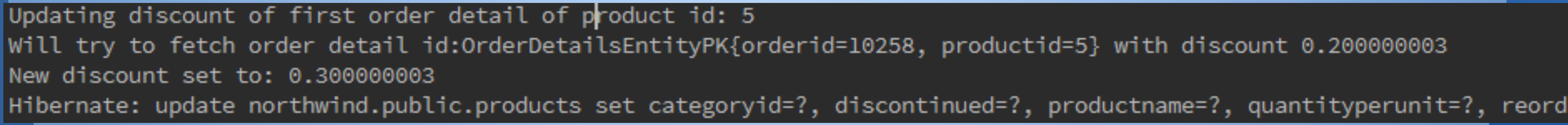
```java
@Transactional
public void verifyOrderDetailsDiscount(OrderDetailsEntityPK pk) {
    final OrderDetailsEntity refetchedOrderDetail = orderDetailsDao.get(pk);
    System.out.println("Order details id " + pk + " has discount " + refetchedOrderDetail.getDiscount());
}
```

Cascades - how not to hurt yourself

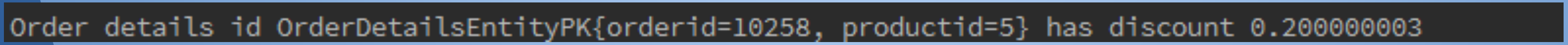
The code:

- 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.

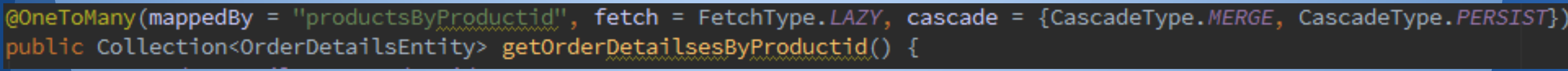
That’s because, *by default, no operations are “propagated” to relations*

Cascades - how not to hurt yourself

cascadeType attribute:

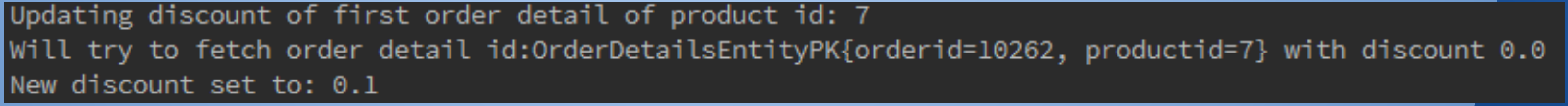
- ALL - Cascade all operations
- DETACH - Cascade detach operation
- MERGE - Cascade merge operation
- PERSIST - Cascade persist operation
- REFRESH - Cascade refresh operation
- REMOVE - Cascade remove operation

Lets use it on our ProductEntity:

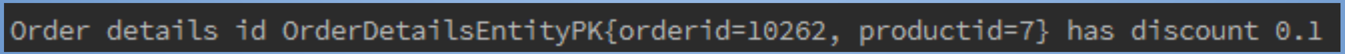


```
@OneToMany(mappedBy = "productsByProductid", fetch = FetchType.LAZY, cascade = {CascadeType.MERGE, CascadeType.PERSIST})
public Collection<OrderDetailsEntity> getOrderDetailsesByProductid() {
```

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*

Caching - when, how and why

Problem:

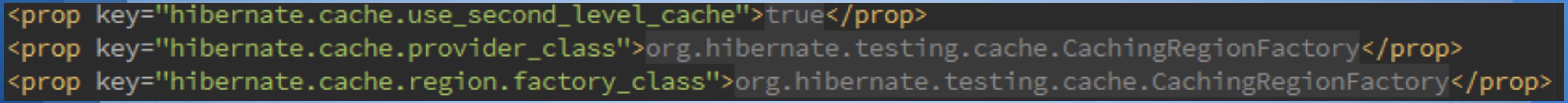
- There exist entities that have much **more reads than writes**
  - think of an online shop and ProductsEntity
  - details rarely updated (maybe even never)
  - thousands of clients read data, every minute
- Going to database for that data each time is an overkill
  - and can literally kill your database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

Solution:

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

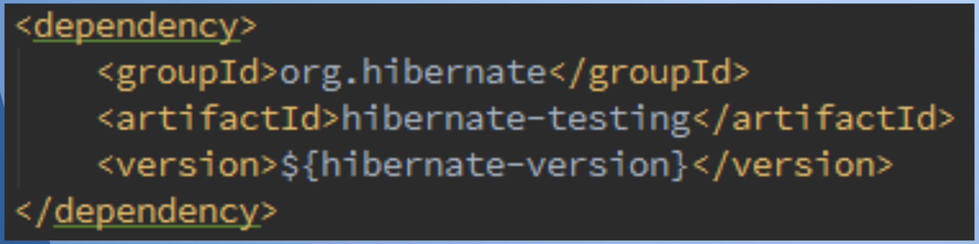
Caching - configuration

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>
```

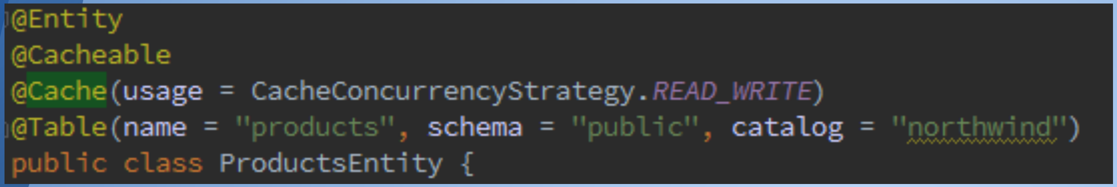
- enable second level caching
- set up caching region provider
  - Hibernate’s ConcurrentHashMap-based one works ok
- add missing dependency



```
<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.



```java
@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

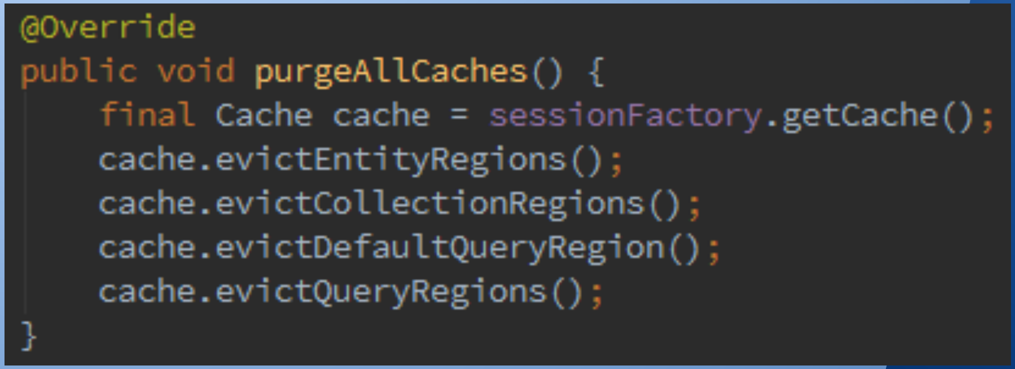
Result:

- 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();
}
```