

.nl.
jug



J-Spring

15 april 2009 Spant!



Hoe bouw ik een ECHTE applicatie met JPA?

Vincent Partington
CTO XebiaLabs



Introductie Vincent Partington

- Sinds 1996 bezig met Java, eerste aanraking met servlets in 1997.
- Interessegebieden: middleware, performance, security. Als het maar Java én moeilijk is. 😊
- Sinds 2003 werkzaam bij Xebia IT Architects.
- Sinds 2008 verantwoordelijk voor de ontwikkeling van het nieuwe deployment lifecycle management product van XebiaLabs: Deployit.



Wat is JPA?

- JPA = Java Persistence API
 - Standaard interface voor object-relational mapping (ORM) tools.
- Onderdeel van EJB 3.0 en dus van Java EE 5
- Geïmplementeerd door:
 - Hibernate
 - TopLink (van Oracle)
 - EclipseLink (open source branch van TopLink)
 - OpenJPA (open source branch van Kodo)
 - En meer...



JPA in twee slides (1/2)

@Entity

```
public class OrderLine {
```

```
    @Id
```

```
    @GeneratedValue
```

```
    private int id;
```

```
    @Column
```

```
    private String description;
```

```
    ...
```



JPA in twee slides (2/2)

```
@PersistenceContext
```

```
protected EntityManager entityManager;
```

```
Order o = new Order();
```

```
o.set(...);
```

```
entityManager.persist(o);
```

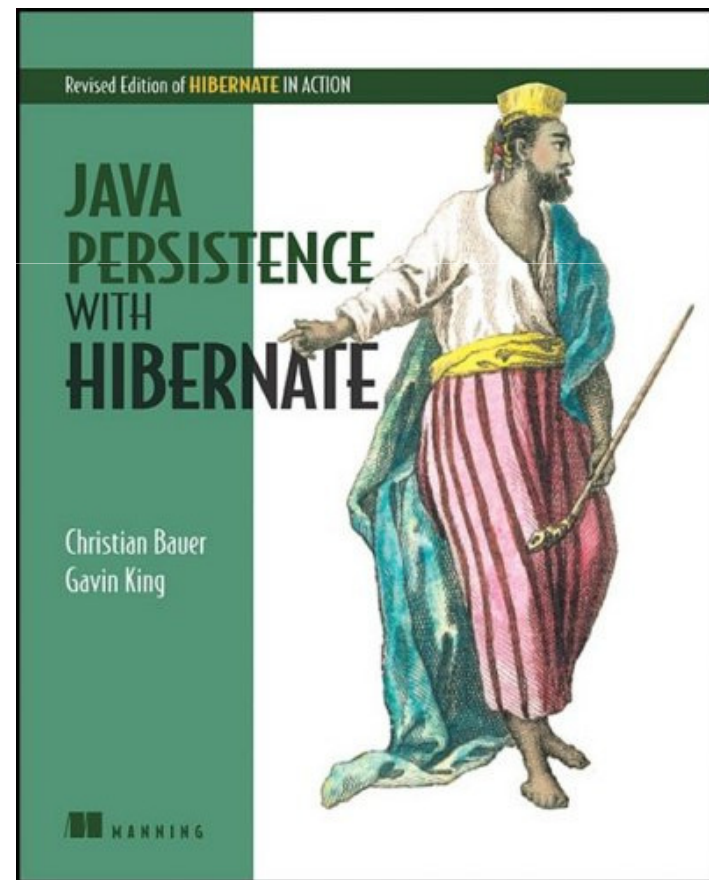
```
Order o = entityManager.find(
```

```
    "OrderLine" , 15);
```



Waarom JPA?

- Annotaties zorgen ervoor dat code en metadata bij elkaar staan:
 - Eenvoudiger te begrijpen.
 - Makkelijker refactoreren.
- Waarom niet?
 - Functionaliteit is grootste gemene deler van ORM tools.
 - JPA evolueert langzamer.





Waarom deze presentatie?

- We gebruiken JPA bij de ontwikkeling van Deployit.
 - Java EE app met services die via BlazeDS en Hessian benaderd worden.
 - Flex frontend en text interface via Hessian.
 - Geen HTML frontend.
- Het gebruiken van JPA in deze Java EE applicatie bleek toch lastiger dan gedacht.
- Boeken geven technische details maar weinig samenhang.
- Dus: zoektocht naar "JPA implementation patterns".



Pattern 1

Type-safe generic DAO



- Waarom eigenlijk?
 - `EntityManager` heeft toch simpele methods.
 - Verwisselen van JPA providers is toch al mogelijk.
 - En switchen naar iets anders dan JPA niet realistisch.
- Maar:
 - Hoe dwing je het juiste gebruik van de `EntityManager` methods door de ontwikkelaar af?
 - Waar laat je je queries?
 - Waar laat je je cross cutting concerns (transacties, logging, etc.)?
 - Single Responsibility Principle (SRP)



Voorbeeld entity - Order

```
@Entity
@Table(name = "ORDERS")
public class Order {
    @Id
    @GeneratedValue
    private int id;
    private String customerName;
    private Date date;

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    ...
}
```



Type-safe generic DAO (1/6)

- Generieke DAO interface met minimale methods:

```
public interface Dao<K, E> {  
    void persist(E entity);  
    void remove(E entity);  
    E findById(K id);  
}
```



Type-safe generic DAO (2/6)

- Subinterfaces voor specifieke entities:

```
public interface OrderDao
    extends Dao<Integer, Order>
{
    List<Order>
        findOrdersSubmittedSince(Date date);
}
```



Type-safe generic DAO (3/6)

- Standaard implementatie van generieke DAO interface:

```
public abstract class JpaDao<K, E>  
    implements Dao<K, E>  
{  
    protected Class<E> entityClass;  
  
    @PersistenceContext  
    protected EntityManager entityManager;  
  
    ...  
}
```



Type-safe generic DAO (4/6)

...

```
public JpaDao() {  
    ParameterizedType s =  
        (ParameterizedType)  
        getClass().getGenericSuperclass();  
    this.entityClass = (Class<E>)  
        s.getActualTypeArguments()[1];  
}
```

...



Type-safe generic DAO (5/6)

...

```
public void persist(E entity) {  
    entityManager.persist(entity);  
}
```

```
public void remove(E entity) {  
    entityManager.remove(entity);  
}
```

```
public E findById(K id) {  
    return entityManager.find(entityClass, id);  
}
```

```
}
```



Type-safe generic DAO (6/6)

```
public class JpaOrderDao
    extends JpaDao<Integer, Order>
    implements OrderDao {

    public List<Order>
    findOrdersSubmittedSince(Date date) {
        Query q = entityManager.createQuery(
            "SELECT e FROM " + entityClass.getName() +
            " e WHERE date >= :date_since");
        q.setParameter("date_since", date);
        return (List<Order>) q.getResultList();
    }
}
```



Pattern 2

Bidirectional associations



Bidirectionele relaties

- `@ManyToOne` geeft aan dat een entity naar een andere entity kan verwijzen.
- `@OneToMany` geeft aan dat een entity naar meerdere andere entities kan verwijzen; een collection.
- Bij `@ManyToMany` gaat het om NxN met een join table.
- En bij `@OneToOne` gaat om 1x1.



Voorbeeld entity - OrderLine

```
@Entity
public class OrderLine {
    @Id
    @GeneratedValue
    private int id;
    private String description;
    private int price;

    @ManyToOne
    private Order order;
```



Orderlines ophalen

- OrderLineDao:

```
public interface OrderLineDao
    extends Dao<Integer, OrderLine>
{
    public List<OrderLine>
        findOrderLinesByOrder (Order o);
}
```



Uitbreidingen in Order

```
public class Order {  
    ...  
    @OneToMany(mappedBy = "order")  
    private Set<OrderLine> orderLines =  
        new HashSet<OrderLine>();  
  
    public Set<OrderLine> getOrderLines() {  
        return orderLines;  
    }  
    public void setOrderLines(  
        Set<OrderLine> orderLines) {  
        this.orderLines = orderLines;  
    }  
}
```



Beheren bidirectionele relaties

- EJB 2.0 had Container Managed Relationships (CMR), maar JPA regelt dat niet.
- De gebruiker van de Order en OrderLine classes moet de relaties bidirectioneel houden.
- Foutgevoelig.
 - De JPA provider kan een hoop maskeren.
 - Beeld in Java objecten vs. beeld in de database.



Nieuwe methods in OrderLine

```
public Order getOrder() {  
    return order;  
}
```

```
public void setOrder(Order order) {  
    if (this.order != null)  
        this.order.internalRemoveOrderLine(this);  
    this.order = order;  
    if (order != null)  
        order.internalAddOrderLine(this);  
}
```



Nieuwe methods in Order

```
public Set<OrderLine> getOrderLines() {  
    return Collections.unmodifiableSet (orderLines) ;  
}
```

```
public void internalAddOrderLine (OrderLine line) {  
    orderLines.add(line) ;  
}
```

```
public void internalRemoveOrderLine (OrderLine l) {  
    orderLines.remove (l) ;  
}
```



Nog meer methods in Order

```
public void addOrderLine(OrderLine line) {  
    line.setOrder(this);  
}
```

```
public void removeOrderLine(OrderLine line) {  
    line.setOrder(null);  
}
```



Pattern 3

Saving (detached) entities



Entities opslaan

- Hibernate heeft `saveOrUpdate`, JPA heeft `persist` en `merge`.

- **Wie kent deze niet?**

```
javax.persistence.PersistenceException:  
org.hibernate.PersistentObjectException:  
detached entity passed to persist:  
com.xebia.jpaij.order.Order;
```



merge vs. saveOrUpdate

| | persist | merge | saveOrUpdate |
|--|------------------------------------|---|---------------------------------------|
| Nieuwe entity | Entity wordt toegevoegd aan db | Nieuwe entity wordt aangemaakt in db, waarden worden gekopieerd, nieuwe entity is resultaat | Entity wordt toegevoegd aan db |
| Bestaande entity maar nog niet geladen in deze context | <code>EntityExistsException</code> | Entity wordt geladen uit db, waarden gekopieerd, geladen entity is resultaat | Entity wordt geupdate in db |
| Entity bestaat én is al geladen | <code>EntityExistsException</code> | Waarden gekopieerd, al geladen entity als resultaat | <code>NonUniqueObjectException</code> |



Persisten of mergen?

- Kopieergedrag van merge kan onverwachte bijwerken hebben, zoals het breken van bidirectionele associaties.
- Vuistregels:
 - Gebruik persist net nadat je een object hebt aangemaakt: persist-on-new.
 - Bij maken van wijzigingen: niets doen, gaat automatisch!
 - Bij binnenkrijgen van nieuwe data: merge of "DIY merge". Laatste is ook logisch als je een DTO gebruikt.



Pattern 4

Retrieving entities



Entities inlezen

- Twee manieren om een entity in te lezen:
 - `EntityManager.find`
 - `EntityManager.createQuery`
 - Geen NamedQueries > queries in de DAO.
- *Finding vs. getting:*
 - `findById` **vs.** `findById`
 - `findOrderSubmittedAt` **vs.** `getOrderSubmittedAt`
 - Maar alleen `findOrdersSubmittedSince`



findById vs. getById

```
public E findById(K id) {  
    return entityManager.find(entityClass, id);  
}
```

```
public E getById(K id)  
    throws EntityNotFoundException  
{  
    E e = entityManager.find(entityClass, id);  
    if (e == null)  
        throw new EntityNotFoundException();  
    return e;  
}
```



findOrdersSubmittedAt vs. ...

```
public Order findOrderSubmittedAt(Date date)
    throws NonUniqueResultException
{
    Query q = entityManager.createQuery(
        "SELECT e FROM " + entityClass.getName() +
        " e WHERE date = :date_at");
    q.setParameter("date_at", date);
    try {
        return (Order) q.getSingleResult();
    } catch (NoResultException exc) {
        return null;
    }
}
```



getOrdersSubmittedAt

```
public Order getOrderSubmittedAt(Date date)
    throws NonUniqueResultException,
           NoResultException,

{
    Query q = entityManager.createQuery(
        "SELECT e FROM " + entityClass.getName() +
        " e WHERE date = :date_at");
    q.setParameter("date_at", date);
    return (Order) q.getSingleResult();
}
```



findOrdersSubmittedSince

```
public List<Order> findOrdersSubmittedSince(  
    Date date)  
{  
    Query q = entityManager.createQuery(  
        "SELECT e FROM " + entityClass.getName() +  
        " e WHERE date >= :date_since");  
    q.setParameter("date_since", date);  
    return (List<Order>) q.getResultList();  
}
```



Pattern 5

Removing entities



Entities verwijderen

- Op zich simpel:

```
public void remove(E entity) {  
    entityManager.remove(entity);  
}
```

- Behalve bij bidirectionele associates.
 - `OrderLineDao.remove()` verwijdert het verwijderde `OrderLine` object niet uit de set van `OrderLines` die in het `Order` object zit.

.nl.
jug



@PreRemove

- Maar we kunnen gebruik maken van de @PreRemove lifecycle hook:

```
@PreRemove  
public void preRemove() {  
    setOrder(null);  
}
```



Orphans

- Maar wat als ik het andersom wil doen?

```
parentOrder.removeOrderLine(orderLineToRemove);
```

- Bovenstaande regel verandert de database niet.
- Tenzij je deze Hibernate specifieke optie gebruikt:

```
@OneToMany(mappedBy = "order")  
@org.hibernate.annotations.Cascade(  
    org.hibernate.annotations.CascadeType.DELETE_ORPHAN)  
private Set<OrderLine> orderLines =  
    new HashSet<OrderLine>();
```



Pattern 6

Lazy loading



Lazy loading

- In Hibernate worden by default alle referenties met lazy loading geladen.
- In JPA:
 - de default voor `@*ToOne` is **EAGER**
 - de default voor `@*ToMany` is **LAZY**.
- Deze heb je vast ook wel eens gezien:
`org.hibernate.LazyInitializationException:
failed to lazily initialize a collection - no
session or session was closed`



Lazy loading efficiënt inzetten

- Pas op voor het n+1 select probleem.
- Gebruik het "Open Entity Manager In View" pattern:
 - `OpenEntityManagerInViewFilter` (Servlet API)
 - `OpenEntityManagerInViewInterceptor` (Spring Web MVC)
- Of gebruik DTO's en een service facade.
 - Meer eerst meer over DTO's en de service facade.



Pattern 7

Data transfer objects



- Hebben we toch niet meer nodig?
 - `toString` en `OpenSessionInViewInterceptor` om makkelijk te renderen in de presentation layer.
 - Data binding om request parameters om te zetten in domein objecten (`PropertyEditors`).
- Of toch wel?
 - Bovenstaande werkt alleen goed in applicaties met een Java web frontend in dezelfde JVM.
 - En zorgt voor vermenging van presentation layer en rest van applicatie.
 - Maakt duidelijk wanneer je moet mergen etc.



DTO Factory

- Creeërt DTO's aan de hand van domein objecten.
- Kan gebruik maken van DAO's om meer informatie op te halen.

- Voorbeeld uit onze applicatie:

```
public interface DTOFactory {  
    public ChangePlanView  
        createChangePlanView(ChangePlan cp);  
    public ChangeView createChangeView(Change c);  
    public StepView createStepView(Step s);  
}
```



Pattern 8

Service facade



Service facade

- Verantwoordelijkheden:
 - Zet binnenkomende DTO's om in domein objecten.
 - Roept juiste service(s) aan.
 - Zet resultaat om in DTO's m.b.v. DTOFactory.
 - Start de transactie.
- Los van alle andere redenen om een service facade te hebben.



Service facade voorbeeld

- Voorbeeld uit onze applicatie:

```
public ChangePlanView findResolvedCurrentChangePlan() {
    ChangePlan cp =
        changePlanService.findCurrentChangePlan();

    if (cp == null)
        return null;

    if (cp.canBeResolved())
        changePlanService.resolveChangePlan(cp);

    return dtoFactory.createChangePlanView(cp);
}
```



Pattern 9

Transaction management



Transactie management

- In een EJB 3.0 applicatie geef je op welke classes en methods transactie boundaries zijn d.m.v. de `@TransactionAttribute` annotatie.
- In Spring heb je de `@Transactional` annotatie.
- Zes mogelijke waarden;
 - `REQUIRED` – maakt een transactie als er nog geen is.
 - `NOT_SUPPORTED` – suspend de TX als er een is.
 - `SUPPORTS` – doet eigenlijk niets.
 - `REQUIRES_NEW` – maakt altijd een nieuwe transactie.
 - `MANDATORY` – gooit een exception als er geen TX is.
 - `NEVER` – gooit een exception als er *wel* een TX is.



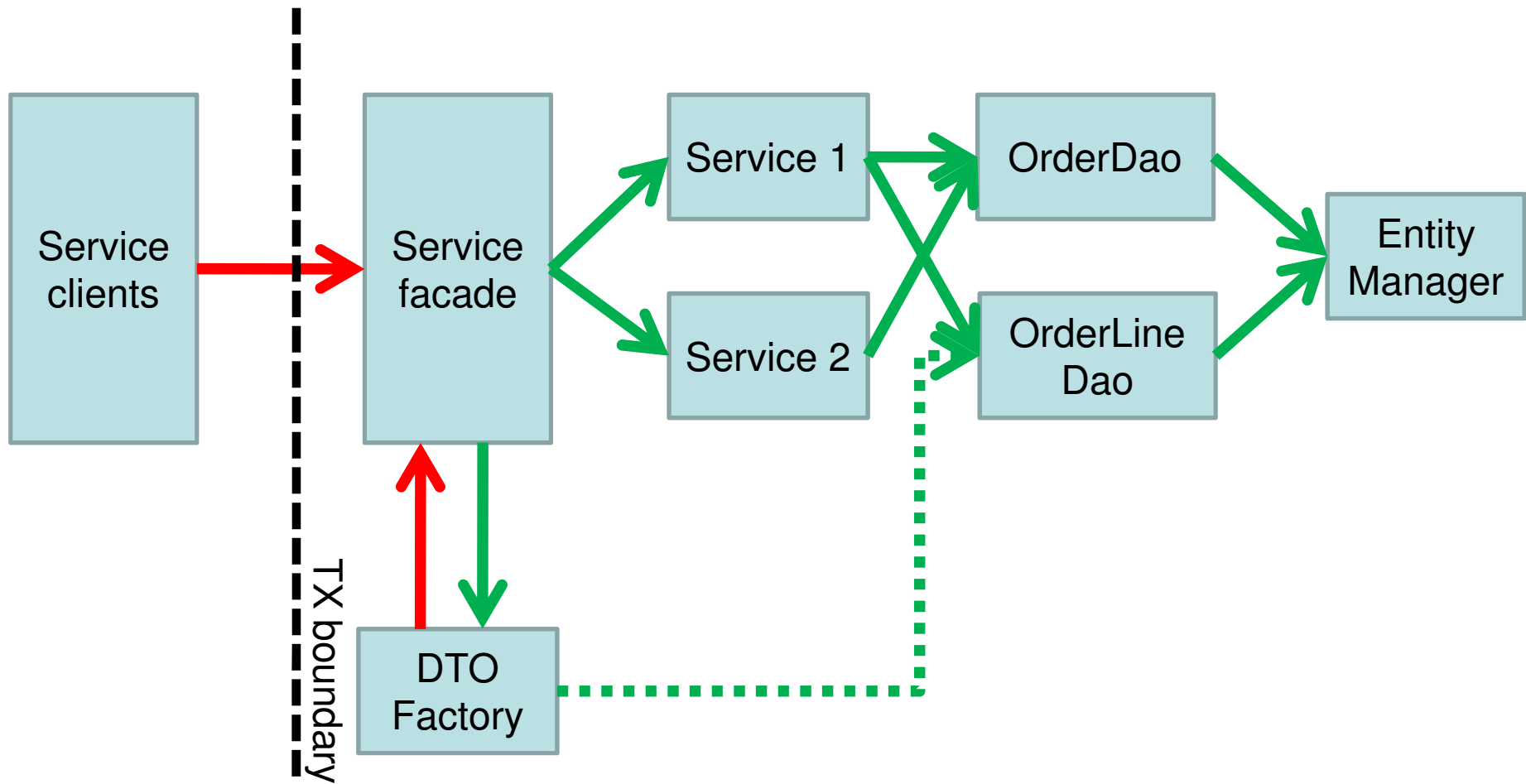
Transactie management

- Welke instellingen moet je waar gebruiken?
 - REQUIRED op service facade is belangrijkste.
 - REQUIRED ook op services en DTO factory (ook een service).
 - MANDATORY op DAO's om rechtstreeks benaderen tegen te gaan.



Pattern 10

JPA application architecture





De patterns

- Pattern 1: Type-safe generic DAO
- Pattern 2: Bidirectional associations
- Pattern 3: Saving (detached) entities
- Pattern 4: Retrieving entities
- Pattern 5: Removing entities
- Pattern 6: Lazy loading
- Pattern 7: Data transfer objects
- Pattern 8: Service facade
- Pattern 9: Transaction management
- Pattern 10: JPA application architecture



Niet behandelde patterns

- Testen
 - @Transactional tests testen niet alles: veel dingen zie je pas in een grotere context, of op zijn minst buiten de transactie.
- Inheritance
 - Verschillende strategiën (`JOINED`, `TABLE_PER_CLASS` (`UNION`), `SINGLE_TABLE`) maar bij diepe inheritance hiërarchiën stuit je op limieten: te veel tabellen in `JOIN` of `UNION`.
- Toegang tot services
 - Simpel gebruik van technologie leidt snel tot "anemic domain model".



Meer informatie

- JPA 1.0 en 2.0 specificaties.
 - Resp. JSR-220 en JSR-317.
- Java Persistence with Hibernate
- Lopende serie op de blog van Xebia:
<http://blog.xebia.com/category/jpa/implementation-patterns/>
- Shameless plug:
<http://www.xebialabs.com/>