JUDCon

JBoss Users & Developers Conference

2012:Boston

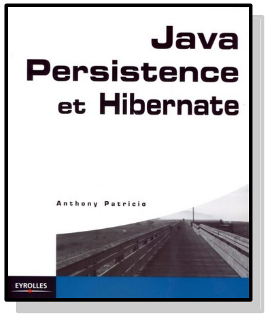
Painless Persistence

Some guidelines for creating persistent Java applications that work

The Authors

Anthony Patricio – Senior JBoss Certification Developer

- Highest volume poster on early Hibernate forums
- 5 years as 3rd level Hibernate support
- Author of two Hibernate books



Greg Kable – JBoss Certification Manager

- More than 20 years enterprise application experience
- More than 15 years enterprise Java experience
- Lived through the evolution of Java persistence from JDBC 1.0 to JPA 2

Anti-pattern – We're Special!

Forces

- Your problem domain is special
- You have leet SQL/JDBC skills
- You don't need no stinking ORM
  - they're heavy, slow and hard to use
  - nobody needs all those features

Results

- Lots and lots and lots of custom, low-level code
  - Constantly re-inventing the wheel
- Very poor maintainability
- Worse performance overall

Solution

Leverage an ORM

- ~~Your problem domain is special~~ Very unlikely!
- ~~You have leet SQL/JDBC skills~~ Very expensive!
- ~~You don't need no stinking ORM~~ You do!
  - they're fast, easy to use and cheap
  - use the features as and when you need them

Results

- Good encapsulation of data concerns
- Easier to maintain and much less code
- Better performance overall with excellent results from targeted optimisation

Anti-pattern – ORM Apathy

Forces

- Hibernate is very good at what it does so nobody needs to understand the DB
- Pressure to deliver NOW!
- Efficient design is HARD!

Results

- Use of default ORM behaviour throughout
- No concern for performance
- Occasional unpredictable behaviours and bugs
- Works in testing but not in production

Solution

Learn how to use JPA

- Be prepared to have some developers who understand ORM and the DBMS
- Design the data and service layers:
  - What does the DBMS look like?
  - What representations does the business logic need?
  - What do you need to do with the data?
- Avoid nice but expensive features (e.g. cascade)
- Monitor performance and work with the DBAs to address hot spots

Results

- It works!

Anti-pattern – Skinny Objects

<u>Forces</u>

- Data focused development (often with a legacy DB)
- Misunderstanding of ORM

<u>Results</u>

- No encapsulation
- Very poor maintainability
- Very fragile implementations

Example

```
@Entity
class Cafe
{
     private int key;

     private Chain chain;
     private Integer longitude;
     private Integer latitude;
     ....

     @Id @GeneratedValue
     public void setKey(int key) {...}
     public int getKey() {...}

     @ManyToOne
     public void setChain(Chain chain) {...}
     public Chain getChain() {...}

     public void setLongitude(Integer longitude) {...}
     public Integer getLongitude() {...}

     public void setLatitude(Integer latitude) {...}
     public Integer getLatitude() {...}
}
```

Solution

Design your entities

- Encapsulate behaviours where appropriate
- Do NOT externalise the entity's internal consistency
- Do NOT expose implementation details
- Fail early, fail often

Results

- Less “wrapper” code
- More reliable business logic
- Faster and more accurate detection of business logic and design errors

A Better Way

```
@Entity
public class Cafe
{
     private int key;

     private Chain chain;
     private Integer longitude;
     private Integer latitude;
     ....

     @Id @GeneratedValue
     private void setKey(int key) {...}
     public int getKey() {...}

     @ManyToOne @Column(nullable = false)
     private void setChain(Chain chain) {...}
     public Chain getChain() {...}

     @Column(nullable = false)
     public Integer getLongitude() {...}
     private void setLongitude(Integer longitude) {...}

     @Column(nullable = false)
     public Integer getLatitude() {...}
     private void setLatitude(Integer latitude) {...}

     @Transient
     public void setLocation(Integer longitude, Integer latitude) {...}
}
```

Even Better

```
@Entity
class Cafe
{
     @Id @GeneratedValue
     private int key;

     @ManyToOne @Column(nullable = false)
     private Chain chain;
     @Embedded
     private Location location;
     ....

     protected Cafe() {}
     public Cafe(Chain chain, Location location) throws NPE {...}

     public int getKey() {...}

     public Location getLocation(Location location) {...}
     public void setLocation(Location location) throws NPE {...}

     public Chain getChain() {...}
}
```

Anti-pattern – OO Purity

Forces

- Heavy focus on OO principles
- Poor attention to DBMS design
- No service layer

Results

- Poor performance (probably fatally so)
- Unpredictable behaviour under load
- Often buggy in very strange ways

Solution

- Design for and use a service layer
- OO + ORM != OODB
  - There is a good reason OODBs have never taken off
- “you can” != “you should”
  - avoid bi-directional associations unless they are required by the business logic
  - be careful mapping inheritance
  - avoid cascade unless you REALLY know the implications
  - make sure entities and actions are well defined and separated

Anti-pattern – DAO Heaven

Forces

- Lack of overall application design
- Poor understanding of ORM

Results

- DAO takes over
- Poor encapsulation
- Overly complex coding and duplicated effort

Solution

<u>Design your data access</u>

- Don't confuse the DAO and the service layer
  - DAO exists to abstract common persistence actions
  - DAO must not understand or be involved in transactions
- EM is a perfectly adequate DAO for small scale
- A single generic DAO works for medium scale
- One DAO per domain model works well for large scale
- Use @NamedQuery and generic query execution

<u>Results</u>

- Clean separation of concerns
- Simpler, more reliable business logic

Anti-pattern – False Identity

Forces

- Inexperience
- Time pressures

Results

- Very difficult bugs
- Eventual maintenance nightmare

Solution

- Use autogenerated keys wherever possible
- ALWAYS declare `equals()` and `hashcode()`

Identity, Equality & Hibernate

	==	ID	Business
Compound Key	No	Yes	Yes
New Instances	Yes	No	Yes
Out of session	No	Yes	Yes
Collection Integrity	Yes	No	Yes

- 1st level cache uses identity
- Everything else uses `equals()`
- Be careful about `equals/hashcode` and hibernate proxies

Identity, Equality & Hibernate

As Generated by Eclipse...

```
@Entity
class Cafe
{
     @ManyToOne @Column(nullable = false)
     private Chain chain;
     ....

     @Override
     public boolean equals(Object obj) {
           ....
           if (getClass() != obj.getClass())
                return false;
           Cafe other = (Cafe) obj;
           if (chain == null) {
                if (other.chain != null)
                      return false;
           } else if (!chain.equals(other.chain))
                return false;
           ....
           return true;
     }
}
```

Identity, Equality & Hibernate

What works...

```
@Entity
class Cafe
{
     @ManyToOne @Column(nullable = false)
     private Chain chain;
     ....

     @Override
     public boolean equals(Object obj) {
           ....
           if (!(obj.instanceof(Cafe)))
                return false;
           Cafe other = (Cafe) obj;
           if (chain == null) {
                if (other.getChain() != null)
                     return false;
           } else if (!chain.equals(other.getChain()))
                return false;
           ....
           return true;
     }
}
```

Recommended Practices

- Use auto-generated primary keys but always declare a business key for Java equivalence
- Include entity version on all tables and use `@Version`
- Don't be afraid to use JP-QL
- Use native JDBC for heavy, read only queries such as reporting – `createNativeQuery()` and `@NamedNativeQuery`
- Second level cache is for read frequently only
- Consider using Seam/Weld's conversation context

5 Steps to Painless Persistence

1. Invest in some JPA skills
2. Design your persistent objects
3. Create a services layer (DAOs are not sufficient)
4. Avoid cool but expensive features (e.g. Cascade) and always work with the DBAs
5. Don't blindly do anything – always think before you code!

Questions