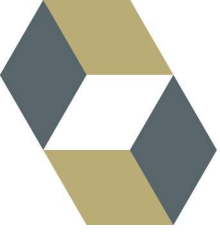


Hibernate

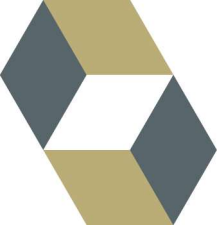
Object/Relational Persistence for
idiomatic Java

Gavin King

Christian Bauer

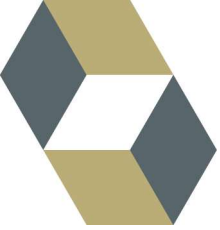


**Why we need
object / relational mapping
and why Hibernate is the best solution.**



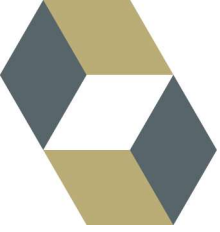
Key Topics

- Why object/relational mapping?
- Solving the mismatch with tools
- Basic Hibernate features
- Hibernate query options
- *Detached Objects*



The structural mismatch

- Java types vs. SQL datatypes
 - user-defined types (UDT) are in SQL:1999
 - current products are proprietary
- Type inheritance
 - no common inheritance model
- Entity relationships
- Collection semantics



Behavioral aspects

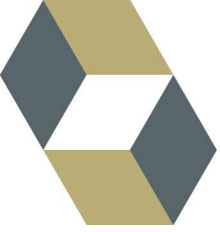
- Java object identity, equality, primary keys

`a == b`

`a.equals(b)`

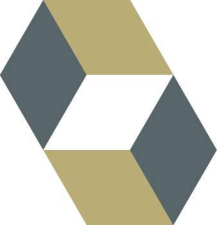
?

- Polymorphism
- Joining tables vs. navigating associations



Modern ORM Solutions

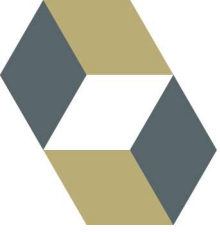
- Transparent Persistence
- Automatic dirty checking
- Transitive Persistence
- Inheritance mapping strategies
- Smart fetching and caching
- Development tools



Defining Transparent Persistence

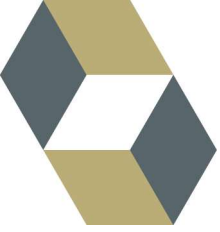
- Any class can be a *persistent* class
- No interfaces have to be implemented
- No persistent superclass has to be extended

- Persistent classes can be used outside of the “persistence” context (Unit Tests, Batch Processing)
- Full portability without any dependency



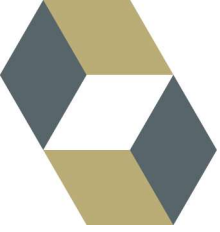
Why ORM?

- Structural mapping more robust
- Less error-prone code
- Optimized performance all the time
- Vendor independence



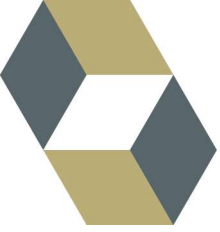
Data integrity is the first rule

- Even so, the relational model is important
- Current implementations are the problem
- Always ensure data integrity using the database
- The data in your SQL database will be around much longer than your application!



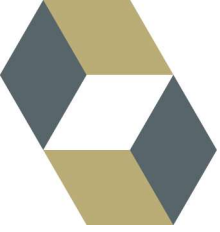
The Goal

Take advantage of the things SQL databases do well, without leaving the Java language of objects and classes.



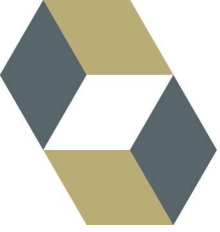
The Real Goal

Do less work and have a happy DBA.



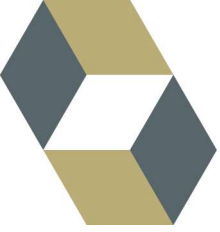
Hibernate

- Open Source (LGPL)
- Mature software driven by user requests
- Popular (15.000 downloads/month)

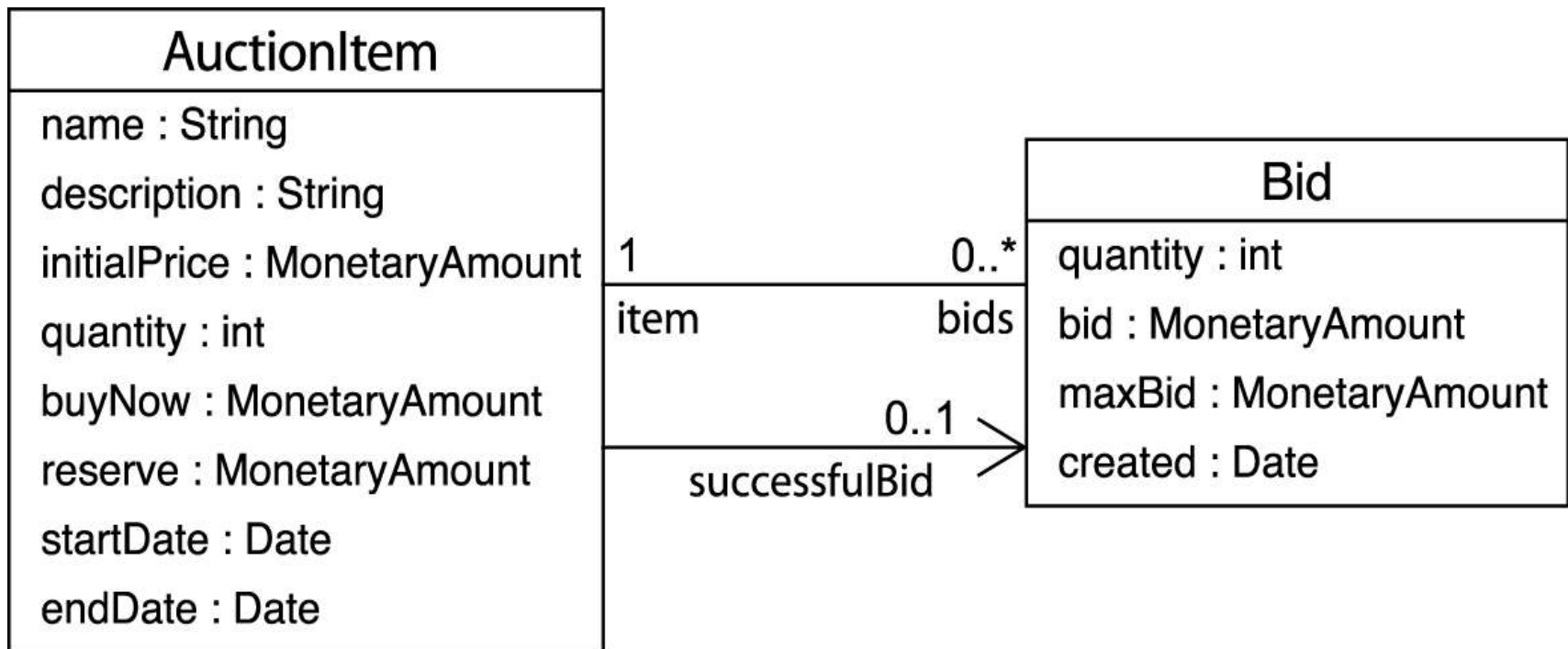


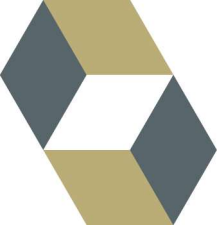
Features

- Persistence for POJOs (JavaBeans)
- Flexible and intuitive mapping
- Support for fine-grained object models
- Powerful, high performance queries
- Dual-Layer Caching Architecture (HDLCA)
- Toolset for roundtrip development
- Support for *detached* objects (no DTOs)



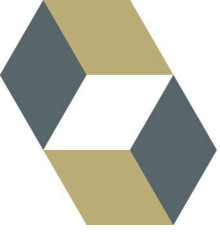
An example object model





Persistent classes

- JavaBean specification (or POJOs)
- Accessor methods for properties
- No-arg constructor
- Collection property is an interface
- Identifier property optional



XML Mapping Metadata

```
<class name="AuctionItem" table="AUCTION_ITEM">

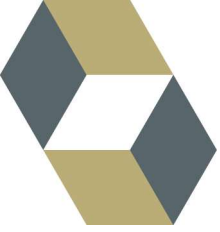
  <id name="id" column="ITEM_ID">
    <generator class="native"/>
  </id>

  <property name="description" column="DESCR"/>

  <many-to-one name="successfulBid"
    column="SUCCESSFUL_BID_ID"/>

  <set name="bids" cascade="all" lazy="true">
    <key column="ITEM_ID"/>
    <one-to-many class="Bid"/>
  </set>

</class>
```



Automatic dirty object checking

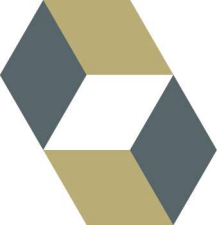
Retrieve an `AuctionItem` and change the description:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

AuctionItem item =
    (AuctionItem) session.get(AuctionItem.class, itemId);

item.setDescription(newDescription);

tx.commit();
session.close();
```



Transitive Persistence

Retrieve an `AuctionItem` and create a new persistent `Bid`:

```
Bid bid = new Bid();
```

```
bid.setAmount(bidAmount);
```

```
Session session = sessionFactory.openSession();
```

```
Transaction tx = session.beginTransaction();
```

```
AuctionItem item =
```

```
    (AuctionItem) session.get(AuctionItem.class, itemId);
```

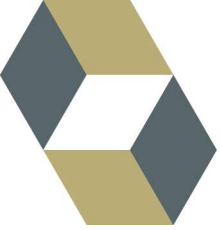
```
bid.setItem(item);
```

```
item.getBids().add(bid);
```

```
tx.commit();
```

```
session.close();
```

**Application managed
associations!**



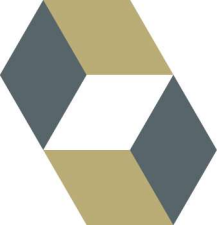
Detached objects

Retrieve an `AuctionItem` and change the description:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
AuctionItem item =
    (AuctionItem) session.get(AuctionItem.class, itemId);
tx.commit();
session.close();

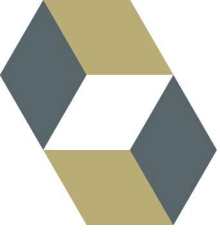
item.setDescription(newDescription);

Session session2 = sessionFactory.openSession();
Transaction tx = session2.beginTransaction();
session2.update(item);
tx.commit();
session2.close();
```



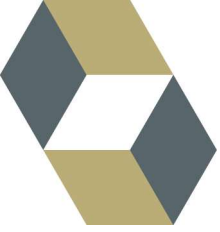
Hibernate query options

- Hibernate Query Language (HQL)
 - object-oriented dialect of ANSI SQL
- Criteria queries (QBC)
 - extensible framework for query objects
 - includes Query By Example (QBC)
- Native SQL queries
 - direct passthrough with automatic mapping
 - named SQL queries in metadata



Hibernate Query Language

- Make SQL “object-oriented”
 - classes and properties instead of tables and columns
 - supports polymorphism
 - automatic association joining
 - *much* less verbose than SQL
- Full support for relational operations
 - inner/outer/full joins, cartesian product
 - projection, ordering, aggregation and grouping
 - subqueries and SQL functions

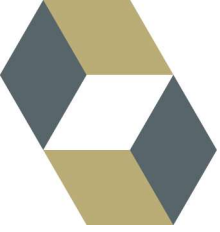


Simplest HQL query

```
from AuctionItem;
```

i.e. get all the AuctionItems:

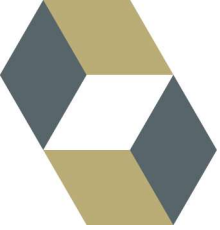
```
List allAuctions =  
    session.createQuery("from AuctionItem").list();
```



A more realistic HQL example

```
select item
  from AuctionItem item
  join item.bids as bid
 where item.description like "Hibernate%"
       and bid.amount > 100
```

i.e. get all the `AuctionItems` with a `Bid` worth more than 100 and an item description that starts with "Hibernate".



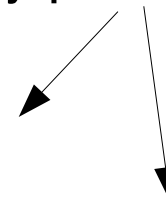
Criteria queries

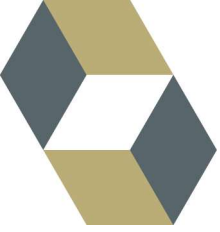
```
List auctionItems =
    session.createCriteria(AuctionItem.class)
        .setFetchMode("bids", FetchMode.EAGER)
        .add( Expression.like("description", desc) )
        .createCriteria("successfulBid")
            .add( Expression.gt("amount", minAmount) )
        .list();
```

Equivalent HQL:

```
from AuctionItem item
    left join fetch item.bids
where item.description like :description
    and item.successfulbid.amount > :minAmount
```

named query parameters





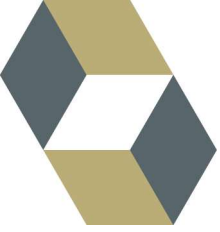
Example queries

```
Bid exampleBid = new Bid();  
exampleBid.setAmount(100);
```

```
List auctionItems =  
    session.createCriteria(AuctionItem.class)  
        .add( Example.create(exampleBid) )  
        .createCriteria("bid")  
            .add( Expression("created", yesterday)  
            .list();
```

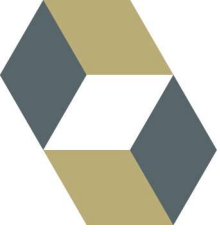
Equivalent HQL:

```
from AuctionItem item  
    join item.bids bid  
where bid.amount = 100  
    and bid.created = :yesterday
```



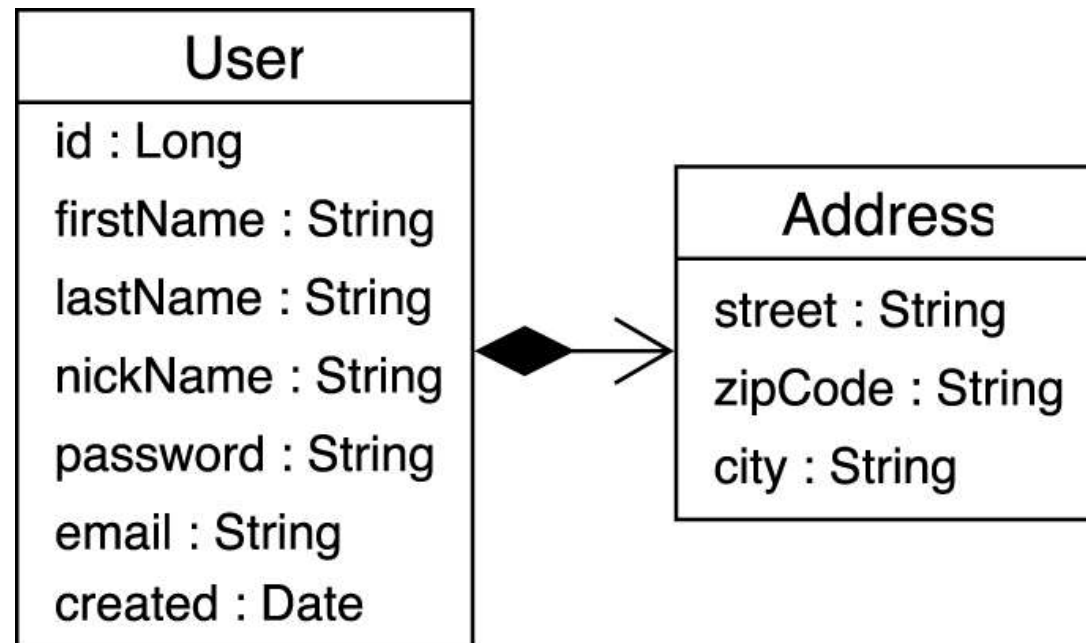
Fine-grained persistence

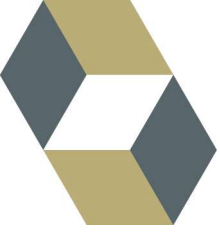
- “More classes than tables”
- Fine-grained object models are good
 - greater code reuse
 - easier to understand
- Hibernate defines
 - Entities (lifecycle and relationships)
 - Values (no identity, “embedded” state)



Composing objects

- Address of a User
- Address depends on User

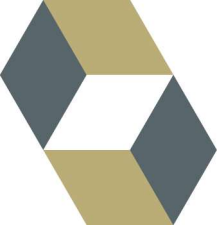




Mapping components

In the mapping metadata of the containing class:

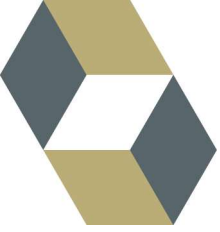
```
<class name="User" table="USER">
  ...
  <component name="address">
    <property name="street" column="STREET"/>
    <property name="zipCode" column="ZIPCODE"/>
    <property name="city" column="CITY"/>
  </component>
</class>
```



DTOs are Evil

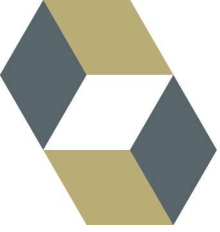
- “Useless” extra LoC
- Only state, no behavior
- Results in parallel class hierarchies
- Shotgun changes are bad

Solution: Detached Object Support



Detached Object Support

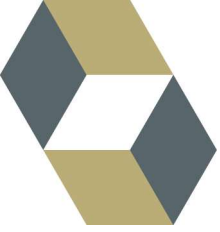
- For applications using Servlets and Session Beans
- You don't need DTOs anymore
- You may serialize objects to the web tier, then serialize them back to the EJB tier in the next request
- Hibernate lets you *selectively* reattach a subgraph!



Step 1: Retrieve objects

in a Session Bean:

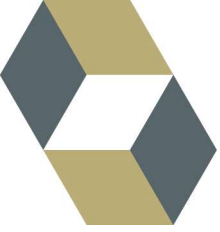
```
public List getItems() throws ... {  
  
    Query q =  
        getSession().createQuery("from AuctionItem");  
  
    return q.list();  
}
```



Step 2: Manipulate objects

in a Servlet, set user information:

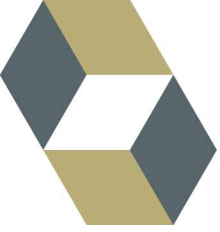
```
item.setDescription(newDescription);
```



Step 3: Make changes persistent

back in the Session Bean:

```
public void updateItem(AuctionItem item) throws ... {  
    getSession().update(item);  
}
```

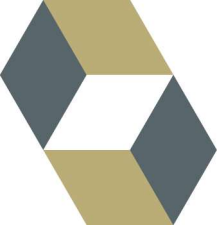


Even with transitive persistence!

```
Session session = sf.openSession();
Transaction tx = session.beginTransaction();
AuctionItem item =
    (AuctionItem) session.get(AuctionItem.class, itemId);
tx.commit();
session.close();
```

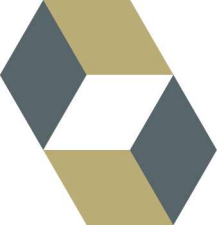
```
Bid bid = new Bid();
bid.setAmount(bidAmount);
bid.setItem(item);
item.getBids().add(bid);
```

```
Session session2 = sf.openSession();
Transaction tx = session2.beginTransaction();
session2.update(item);
tx.commit();
session2.close();
```



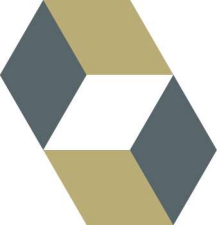
The Big Problem

- Detached Objects & Transitive Persistence
- How do we distinguish between newly instantiated objects and detached objects that are already persistent?



Solution

- Hibernate uses the “version” property, if there is one
- Hibernate uses the identifier value
 - no identifier value means a new object
 - doesn't work for natural keys, only for Hibernate managed surrogate values!
- Write your own strategy with
`Interceptor.isUnsaved()`



Q & A

<http://www.hibernate.org>