Hibernate in 60 Minutes

Eric M. Burke
www.ericburke.com
My JDBC Experience

- Frustration drove me to investigate Hibernate
- Custom JDBC frameworks and idioms
  - Usually a class called JdbcTemplate with static methods
- Custom code generators
  - Database metadata is too cool to resist
- Custom strategy for optimistic locking
- Database-specific ID generation
Typical JDBC Problems

- Massive duplication
- Tied to specific database
- Error-prone try/catch/finally

- Relationships are really hard
  - N+1 selects problem
  - parent/child updates
    - delete all then add

- Screen-specific DAOs for optimization
Hibernate Questions

- How hard is it to learn and use?
- How invasive is it to my code?
- What are the configuration headaches?
  - Yet another framework with XML files, JAR files, additional build complexity?
- Does it simplify my life?
What is Hibernate?

- Java Object/Relational Mapping
  - Open source – LGPL
  - http://www.hibernate.org/ (JBoss Group)

- Lets you avoid SQL

- Works with (almost) all relational DBs
  - DB2, FrontBase, HSQLDB, Informix, Ingres, Interbase, Mckoi, MySQL, Oracle, Pointbase, PostgreSQL, Progress, SAP DB, SQL Server, Sybase, etc...
  - Or you can extend one of the Dialect classes
What is Hibernate? (Cont'd)

- Has a huge feature list
  - Go to their web site – we don't have time

- Maps JavaBeans (POJOs) to tables
  - XML defines the mappings
  - Very few bean requirements

- SQL is generated at app startup time
  - As opposed to bytecode manipulation
  - New database? No problem! Change a few props
Avoiding SQL

Hibernate Query Language approach

```java
public User getByScreenName(String screenName) { 
    try { 
        return (User) getSession().createQuery( 
            "from User u where u.screenName=:screenName"
        ).setString("screenName", screenName)
        .uniqueResult();
    } catch (HibernateException e) { 
        throw new DaoException(e); 
    }
}
```

Code-based query API

```java
public User getByScreenName(String screenName) { 
    return getUnique(Expression.eq("screenName", screenName));
}
```
Sample Application Code

```java
Session session = getSessionFactory().openSession();

// assume we know the root category id is 1
Category rootCategory = (Category) session.load(
    Category.class, new Long(1));

// we can immediately traverse the collection of messages
// without any additional Hibernate code
for (Object msgObj : rootCategory.getMessages()) {
    Message msg = (Message) msgObj; // generics+XDoclet==bad
    System.out.println("Subject: "+ msg.getSubject());
    System.out.println("Body: "+ msg.getBody());
    System.out.println();
}

session.close();
```
Sample Code with Transaction

Session sess = ...  
Transaction tx = sess.beginTransaction();

// find the root category using Criteria
Criteria criteria = sess.createCriteria(Category.class);
criteria.add(Expression.isNull("parentCategory"));
Category rootCategory = (Category) criteria.uniqueResult();

// change something
rootCategory.setDescription("The Root Category");

tx.commit();
sess.close();
Development Options

- Start with the database
  - Use middlegen to generate mapping files
  - Use hbm2java to generate JavaBeans

- Start with JavaBeans 👍
  - Use XDoclet to generate mapping files
  - Use hbm2ddl to generate the database

- Start with the XML mapping files
  - Hand-code the mapping files
  - Use both hbm2java and hbm2ddl
Hibernate Tutorial

- Design the database
- Code some persistent classes
- Write an Ant buildfile
  - Generate the mapping files and hibernate.cfg.xml
- Choose a strategy for sessions and transactions (Spring framework)
- Write and test the DAOs
- Add business logic and helper methods
Database Design

Design your database first

Categories

<table>
<thead>
<tr>
<th>PK</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>description</td>
</tr>
<tr>
<td>FK1</td>
<td>parentCategoryId</td>
</tr>
</tbody>
</table>

Surrogate Keys

Work Best

Resources

<table>
<thead>
<tr>
<th>PK</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>url</td>
</tr>
<tr>
<td></td>
<td>description</td>
</tr>
<tr>
<td></td>
<td>createdOn</td>
</tr>
</tbody>
</table>

CategoriesResources

<table>
<thead>
<tr>
<th>PK,FK1</th>
<th>categoryId</th>
</tr>
</thead>
<tbody>
<tr>
<td>PK,FK2</td>
<td>resourceId</td>
</tr>
</tbody>
</table>

many-to-one

many-to-many
Persistent Objects

- **POJOs - Plain Old Java Objects**
  - No base class or interface requirement

- **Must have public no-arg constructor**

- **Should have getter/setters**
  - Or Hibernate can access fields directly
  - Hibernate uses JavaBeans property names
    - For mappings as well as in queries

- **Serializable is recommended**

- **XDoclet tags are useful but not required**
public class PersistentClass implements Serializable {
    private static final long serialVersionUID = 1;
    private Long id;

    /**
     * @hibernate.id
     *  column="id"
     *  unsaved-value="null"
     *  generator-class="native"
     *  access="field"
     */
    public Long getId() {
        return id;
    }

    private void setId(Long id) {
        this.id = id;
    }
}
public class Category extends NameDescription {
    private static final long serialVersionUID = 1;

    private Set childCategories = new HashSet();
    private Set resources = new HashSet();
    private Category parentCategory;

    public Category() {
    }

    public Category(String name, String description) {
        super(name, description);
    }

    public Set getChildCategories() {
        return childCategories;
    }

    public void setChildCategories(Set childCategories) {
        this.childCategories = childCategories;
    }
}
Category (Cont'd)

```java
public Category getParentCategory() {
    return parentCategory;
}

public void setParentCategory(Category parentCategory) {
    this.parentCategory = parentCategory;
}

public Set getResources() {
    return resources;
}

public void setResources(Set resources) {
    this.resources = resources;
}
```
Mapping Files

Generated by XDoclet, 1 per persistent bean

```xml
<hibernate-mapping>
  <class name="com.ericburke.linkblog.model.Category"
     table="Categories">
    <id name="id" column="id" type="java.lang.Long"
      unsaved-value="null">
      <generator class="native"/>
    </id>

    <set name="childCategories" lazy="true"
        inverse="true" cascade="save-update">
      <key column="parentCategoryId"/>
      <one-to-many class="com.ericburke.linkblog.model.Category"/>
    </set>

    <many-to-one name="parentCategory"
        class="com.ericburke.linkblog.model.Category"
        cascade="none" column="parentCategoryId" not-null="false"/>
  </class>
</hibernate-mapping>
```

Or Java 5 Annotations
Mapping Files (Cont'd)

```xml
<set name="resources" table="CategoriesResources"
     lazy="true" cascade="save-update">
    <key column="categoryId"/>
    <many-to-many column="resourceId"
               class="com.ericburke.linkblog.model.Resource"/>
</set>

<property name="name" type="java.lang.String"
          column="name" length="255"/>

<property name="description" type="java.lang.String"
          column="description" length="2000"/>
</class>
</hibernate-mapping>
```
Xdoclet Tag Example

/**
 * @hibernate.class table="Categories"
 */
public class Category extends NameDescription {

/**
 * @hibernate.set cascade="save-update" inverse="true"
 *    lazy="true"
 * @hibernate.collection-key column="parentCategoryId"
 * @hibernate.collection-one-to-many
 *    class="com.ericburke.linkblog.model.Category"
 */
    public Set getChildCategories() {
    return childCategories;
    }

    // no tags on the setters
public void setChildCategories(Set childCategories) { }

<hibernate-configuration>
    <session-factory>
        <!-- Property definitions here -->
    </session-factory>
</hibernate-configuration>

Generated by XDoclet, or use Spring's XML file
Ant Buildfile

- Invokes XDoclet
  - XDoclet ships with the Hibernate tasks
  - Generates mapping files and hibernate.cfg.xml

- Hibernate includes tasks for
  - hbm2java – generates the persistent classes
  - hbm2ddl – generates the database

- Classpath needs
  - Hibernate Jars, Hibernate extensions, XDoclet, JDBC driver JAR file

Requires the persistent .class files in the taskdef's classpath
HibernateUtil

- Trivial examples show this class
  - Creates a SessionFactory in the static initializer
    
    ```java
    Configuration cfg = new Configuration();
    SessionFactory sessionFactory =
        cfg.configure("/hibernate.cfg.xml")
        .buildSessionFactory();
    ```
  - Provides access to a Session ThreadLocal
    
    ```java
    public static Session getSession() {
        Session s = (Session) threadSession.get();
        if (s == null) {
            s = sessionFactory.openSession();
            threadSession.set(s);
        }
        return s;
    }
    ```
  - APIs to begin/commit/rollback transactions

April 14, 2005
DAOs

- Insulate your code from Hibernate specifics
- CRUD operations
  - I use Java 5 generics heavily
  - You can also use Spring for DAO support
- Locating persistent objects
  - Use Hibernate queries for efficiency
  - From there, walk the graph using normal Java collections semantics
    - Detached objects can be challenging
public class HibernateFilter implements Filter {
    public void init(FilterConfig fc) throws ServletException {
    }

    public void destroy() {
    }

    public void doFilter(ServletRequest req, ServletResponse res,
                         FilterChain filterChain)
            throws IOException, ServletException {
        try {
            filterChain.doFilter(req, res);
            HibernateUtil.commitTransaction();
        } finally {
            HibernateUtil.closeSession();
        }
    }
}
Business Objects

- A layer between your application code and the DAOs
  - Only allow business objects to hit the DAOs
  - Business objects and app code use persistent objects

- Provide a consistent place to use AOP
  - try/catch/finally for transactions

- Risk duplicating DAO code
  - Sometimes business object methods just delegate
Architecture Summary

AOP for Transactions

Business Objects → DAOs → Hibernate → Database

Application Code

Persistent Objects

XML Mappings

Keep an Open Session
Gotchas

- **Must always have an open session**
  - Seemingly innocent code like fetching a lazy collection bombs
  - Swing – open a session on event thread
  - Servlets – use a filter or Spring

- **Hibernate tasks do not abort Ant**
  - No failonerror attribute
  - Watch the build output carefully

- **Test associations and collections heavily**
When Not to Use Hibernate

- A horrible database design is forced on you
  - Many composite keys
- If your application is mostly mass operations
  - See pg. 181 “Hibernate in Action”
  - You might exhaust available heap space
- ??
10 Reasons to Love Hibernate

1. Dynamic UPDATE generation – figure out which columns changed
2. Trying new databases
3. Traversing associations
4. Optimistic Locking
5. ID generation
6. No more DTOs
7. Query result paging
8. Automatic dirty checking
9. Good price
10. Good documentation