Java and Databases

PPJDG, May 2004
Chris Smith, MindIQ
Overview

- The Persistence Problem
- Object/Relational Mapping
- Persistence in EJBs
- Hibernate
- JDO
- Object Databases
- Wrap-up
The Persistence Problem
Why Worry About Persistence?

- Persistence is one of the most common requirements of modern applications.
- More money is often spent on persistence than the rest of the application put together!
- Persistence is generally the major performance limiter for enterprise applications.
- Java developers spend a lot of time doing the same thing over and over.
Basic Persistence Approaches

- Persistence can be solved by:
  - Avoiding a database
    - Can be made easy to do (serialization)
    - Fine for transient local data, but doesn’t scale
  - Writing JDBC by hand
    - A lot of repetition in “CRUD” operations
    - Possible ad-hoc automation of common tasks
  - Third-party frameworks and services
    - Easier, and often a better theoretical basis
    - Often some performance cost, but not much
Persistence Frameworks

- Born of the concept that much persistence work can be automated...

- Buyer (or free software user) beware:
  - Easy to write a 90% functional framework
  - Easy to think you can solve this problem
  - Hard to really solve persistence problems
Goals and Concerns

- Performance
- Scalability
- Flexibility
- Transparency
- Fidelity to Relational Model
- Fidelity to Object Model
- Legacy Data
- Cross-application and cross-platform access
Enabling Technologies

- Explicit Persistence
- Runtime Introspection of Code (Reflection)
- Code Generation
- Bytecode Postprocessing
- Proxies and Polymorphism
Object/Relational Mapping
Definition: O/R Mapper

- An O/R mapper bridges between the relational and object models of data.
  - Loads from relational database to objects
  - Saves from objects to relational database

- **Relational Model:** A strict mathematical model of information used by most DBMS packages.
- **Object Model:** A looser, more familiar information model found in application programming languages.
The Relational Model - Basics

- Two kinds of relational terminology:
  - Snobby pretentious words
  - Practical everyday usage words

<table>
<thead>
<tr>
<th>Snobby Word</th>
<th>Normal Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relation</td>
<td>Table</td>
</tr>
<tr>
<td>Tuple</td>
<td>Row</td>
</tr>
<tr>
<td>Attribute</td>
<td>Column or Field</td>
</tr>
<tr>
<td>Domain</td>
<td>Type</td>
</tr>
</tbody>
</table>
The Relational Model - Joins

- Relational entities are associated by joining tables.
  - Fast for arbitrary searching of data
  - Slow for navigating intrinsic associations

- This favors a very course entity model
  - In other words, keep as much information as possible in one table.
The Object Model

- Object models vary between languages.
  - Not as strict as the relational model.
- In general, though, objects have:
  - Identity
  - State
  - Behavior
- Don’t worry about modeling behavior, but identity and state are both critical!
Mapping State

- In general, it’s easiest to map an object’s state to fields in a table.

- Potential problems:
  - Granularity: object models should generally be finer-grained than relational models
  - Associations: relationships between entities are expressed very differently
    - Collections or Foreign Keys
Mapping Identity

- Objects have identity that’s independent of their attributes or fields.
- This is foreign to the relational world.

**Solutions:**
- Give up identity, and use a natural key
- Invent a surrogate key to represent identity
Another Stick in the Gears

A very important distinction:

- Databases model a complete set of data.
- Applications use a subset of data.

A good O/R mapper will let you build a simplified map to just that data that your application needs.
Persistence in EJBs
Entity Bean Persistence

- EJB provides two options for persisting entity beans:
  - BMP - Bean Managed Persistence
    - a.k.a., “If you want it done right…”
    - Not a persistence framework
  - CMP – Container Managed Managed Persistence
    - A somewhat rigid and inflexible persistence framework for entity beans.
EJB-CMP Vital Signs

- Persistence only for EJB entity beans
- Concrete classes are generated at runtime from abstract bean classes
- Associations declared in deployment descriptors and managed with CMR
- EJB-QL query language for data
EJB-CMP Advantages

- Easy to use, if you’re already in an entity bean environment.
- Integrated into your application server.
  - No redundant declarations for persisting bean relationships.
  - Able to take advantage of container-private knowledge about transaction management.
- If you’ve already bought WebLogic, you may as well use it!
EJB-CMP Disadvantages

- No standard way to map beans to legacy data from an existing database.
  - Some containers provide this ability
  - Some containers don’t
- No inheritance/polymorphic relationships
- Portability issues moving between application servers.
EJB-CMP Disadvantages II

- Little flexibility in data representation.
- And the biggie:
  - You have to be writing an EJB entity bean!
EJB-CMP Performance

- This is a raging debate.
  - App Server vendors claim (unsurprisingly) that CMP performs fine. BEA says better than is possible with BMP. (huh?!?)
  - General experience is that CMP performance is bad, but that’s possibly related to memories of earlier versions.
- As always, the truth is in the middle.
  - And will depend on your app server!
Bean Managed Persistence is the other option for EJB entity beans.

BMP is not a persistence framework. You can:

- Write your own JDBC by hand
- Use another persistence framework that works with EJB-BMP (such as Hibernate)

The latter approach has a nice separation-of-concerns ring to it.
Hibernate
Hibernate Vital Signs

- Data persistence service for “ordinary” Java objects and classes.
- Associations via Java Collections API
- O/R mapping defined in an XML file
- Persistent operations via an external `net.sf.hibernate.Session` object
- HQL – An object-relational query language
- Basic tool set for common tasks
Persistent Classes

- Look a lot like ordinary Java classes.
  - Some special considerations, though.

- Requirements:
  - JavaBeans accessors and mutators
  - No-argument constructor
  - Use of Collections interfaces
    - List, not ArrayList
  - Don’t rely on null collections or elements
Entities and Components

- Entities in Hibernate represent entity tables in the database.
  - Recall that the relational model encourages entities to be course-grained.

- Components are dependent objects
  - A customer record in a database may have columns called ‘address’, ‘city’, ‘state’, etc.
  - The Hibernate mapping may use a component class called Address to encapsulate those fields.
Collections for Associations

- Many kinds of collections:
  - Set, Map, List, array, and “Bag” (using List)
  - Even SortedSet and SortedMap
  - All implemented with foreign keys and/or association tables in the database

- Hibernate is strict about collections:
  - Sometimes common to use List as a “convenience” for unordered collections
  - This won’t work in Hibernate; Set is better
Mapping with XML Documents

- Each Hibernate persistent class should have an XML mapping document.
- Defines:
  - Persistent properties
  - Relationships with other persistent classes
- Writing these mapping files is the “core” Hibernate persistence task.
  - … but there are tools for other approaches
Using the Session

- Persistent operations are available via a `net.sf.hibernate.Session` object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Session.load</td>
<td>Loads an object from the database</td>
</tr>
<tr>
<td>Session.get</td>
<td>Same, but the object may not exist</td>
</tr>
<tr>
<td>Session.save</td>
<td>Adds a new object to the database</td>
</tr>
<tr>
<td>Session.delete</td>
<td>Deletes an object from the database</td>
</tr>
<tr>
<td>Session.find</td>
<td>Search for objects by an HQL query</td>
</tr>
<tr>
<td>Session.filter</td>
<td>Get a subset of some collection</td>
</tr>
<tr>
<td>Session.flush</td>
<td>Flush local changes out to the database</td>
</tr>
</tbody>
</table>
HQL – Hibernate Query Language

- HQL is used for building queries to find or filter data from the database.
  - Looks like SQL’s SELECT at first glance

- Differences from SQL:
  - It’s only used for searching, not updating.
  - It understands inheritance polymorphism, and object-oriented ownership of associations.
  - Most pieces (even the SELECT clause) are optional in at least some situations!
HQL - Basics

- Simplest possible HQL query:
  - `from Employee`
  - Returns all employees in the database

- HQL implements the four ANSI join types, plus Oracle-style Cartesian joins.

- Clauses:
  - `SELECT` (optional)
  - `FROM` (required, except with `Session.filter`)
  - `WHERE` (optional)
  - Other: `ORDER BY`, `GROUP BY`, `HAVING`, …
HQL – Complex Queries

- HQL supports subqueries and correlated subqueries...
  - If the underlying database does.
  - That means no MySQL (for now?)
- Named or positional parameters:
  - Use `createQuery` to build the query
  - Methods on `Query` set the parameters
    - `from Employee where salary > :minSalary`
Hibernate Tools

- Plain Hibernate means writing XML mapping.
- Other options include:

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hbm2ddl</td>
<td>Generate DDL for database directly from Hibernate mapping files</td>
</tr>
<tr>
<td>hbm2java</td>
<td>Generate Java source file directly from Hibernate mapping files</td>
</tr>
<tr>
<td>Middlegen</td>
<td>Generate Hibernate mapping from an existing database schema</td>
</tr>
<tr>
<td>AndroMDA</td>
<td>Generate Hibernate mapping from UML</td>
</tr>
<tr>
<td>XDoclet</td>
<td>Generate Hibernate mapping from annotations in source code</td>
</tr>
</tbody>
</table>
Hibernate Sample
Hibernate Summary

- XML files define the O/R mapping.
- Ordinary objects, but with some rules.
- Associations via the Collections API.
- HQL language for queries.
Java Data Objects (JDO)
JDO Vital Signs

- API for “ordinary” Java object persistence
- Successor to ODMG API
- Generally uses bytecode enhancement to provide persistence (with controversy…)
- Persistence provided through a PersistenceManager instance
- Managed (via JCA) or local environments
- Mapping to database is not specified!
JDO Status

- JDO is an official Java Specification
  - JSR-12: Java Data Object (JDO)
- But, not adopted by database vendors
  - Limited to ODBMS vendors only
  - Most deployments use third-party O/R mappers
- IBM, BEA, and Oracle recently banded together to kill the 2.0 release of the spec.
  - But they failed
JDO Step-by-Step

- Write persistent classes to represent data
- Enhance the classes with a JDO enhancer
- Obtain an instance of PersistenceManager
- Obtain PersistenceCapable instances
- Modify JDO objects obtained
- Store and commit results
Persistent Classes

- Implement the PersistenceCapable interface, either:
  - Because they were enhanced, OR
  - Because they were written that way (rare)
- Can’t contain references to classes that are not persistence-capable
  - Certain system classes are considered capable, though.
- May use Set – but List and Map collection types are optional for the implementation!
First/Second Class Objects

- **First-class objects (FCOs):**
  - Are independently persistent entities
  - Can exist on their own in the database
  - Can be associated with several other entities

- **Second-class objects (SCOs):**
  - Are part of their containing FCO
  - Cannot exist on their own on the database
  - Must belong to one specific FCO
  - Reminds you of Hibernate’s components?
  - Often there’s no guarantee of SCO vs. FCO
Persistent Fields

- Fields are persisted – not JavaBeans properties as in Hibernate.
- May be FCOs or SCO.
- May be primitives, class-typed or interface-typed references.
- References must be to persistence-capable instances...
  - including system classes that are persistence-capable from the JDO implementation.
Bytecode Enhancement

- Classes are made persistence-capable by bytecode enhancement.
  - Helps objects determine what fields changed.
  - Provides support for instance-pooling.
  - And more, of course.

- There are alternatives:
  - Preprocessing: enhance before compiling
  - Code generation: create pre-enhanced code

- Enhanced classes implement PersistenceCapable
Getting a PersistenceManager

- A JDO application starts by obtaining a persistence manager instance.
- This is done via a PersistenceManagerFactory.
  - Can be obtained with JDOHelper’s getPersistenceManagerFactory method
  - Pass a properties object with parameters
  - Required parameter: javax.jdo.PersistenceManagerFactoryClass
The PersistenceManager

- The entry point to persistence operations

- Used to:
  - Add/delete/detach objects from the database
  - Retrieve specific objects by ID
  - Build “extents” of objects from the database
  - Build queries to filter objects from extents
  - Obtain the current transaction for operations
  - Manage the persistent object cache
Using the PersistenceManager

<table>
<thead>
<tr>
<th>makePersistent(Object)</th>
<th>deletePersistent(Object)</th>
</tr>
</thead>
<tbody>
<tr>
<td>getObjectById(Object,boolean)</td>
<td>makeTransient(Object)</td>
</tr>
<tr>
<td>getObjectId(Object)</td>
<td>currentTransaction()</td>
</tr>
<tr>
<td>refresh(Object)</td>
<td>retrieve(Object)</td>
</tr>
<tr>
<td>evict(Object)</td>
<td>currentTransaction()</td>
</tr>
<tr>
<td>getExtent(Class,boolean)</td>
<td>newQuery()</td>
</tr>
<tr>
<td>newQuery(Object)</td>
<td>newQuery(Class)</td>
</tr>
<tr>
<td>newQuery(...)</td>
<td>close()</td>
</tr>
</tbody>
</table>
PersistenceCapable Interface

- PersistenceCapable is intended as an internal interface
  - not used by application
- Instead, use JDOHelper static methods:
  - getPersistenceManager
  - makeDirty/isDirty
  - getObjectId
  - isPersistent/isTransactional/isNew/isDeleted
- Changes to objects saved automatically!
JDOQL

- JDO uses query language JDOQL.
- Based more on Java than SQL…
  - But some SQL syntax is still there.

- JDOQL fragments fit in specific locations:
  - Query.setFilter (like SQL WHERE clause)
  - Query.declareImports, declareVariables and declareParameters
  - Query.setOrdering (like SQL ORDER BY)
JDOQL Example

- Sample JDOQL with parameter:

```java
Extent ext = pm.getExtent(BreakfastItem.class, false);
Query q = pm.newQuery(ext, "carbs <= maxc");
q.declareParameters("Integer maxCarbs");
Collection items = (Collection) q.execute(new Integer(15));
```
JDO Sample
JDO Summary

- O/R Mapping (and lots of functionality) is implementation-specific.
- Ordinary objects, with restrictions.
- JDOQL for query language.
- Complex because of level of abstraction.
Object Databases
Object Databases

- An alternative to relational databases...

- Object databases!
  - Not new; they have been around for ages
  - Not as much standardization as relational
  - Sometimes lack scalability and data integrity of relational databases.

- Interfaces are all over the board:
  - JDO is frequently used.
  - Proprietary interfaces are also common.
Pick One

- For the purposes of this presentation, we choose one object database.

- Since you’ve already seen JDO, I choose one with a proprietary interface:
  - db4o (= database for objects)
  - Available for Java and .NET
db4o Vital Signs

- Persistence for ordinary objects
  - The only solution that doesn’t restrict your object model to fit database needs
- Uses pure reflection for persistence
- Opaque non-relational database; no O/R mapping or schema management
- Query-by-example or “S.O.D.A.” querying
Weaknesses of db4o

- Doesn’t scale well in my testing to arbitrary queries on very large data, as in data mining.
  - But it’s very fast for simple persistence needs
  - Perhaps 1000 times the speed of some databases
- Poor referential integrity checking
  - Deleting an elsewhere-referenced object doesn’t give an error message, but causes database contents to become potentially invalid.
  - No provision for defining referential integrity constraints on the data.
  - (Validation for non-referential constraints can happen in Java mutator methods.)
Steps in Using db4o

- Create an ObjectContainer.
- Use the ObjectContainer for data manipulation.
- Commit between transactional boundaries.
- Close the ObjectContainer
  - Careful – there’s an automatic commit here!
## Basic ObjectContainer Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get(Object)</code></td>
<td>Queries by example</td>
</tr>
<tr>
<td><code>query()</code></td>
<td>Creates a S.O.D.A. query</td>
</tr>
<tr>
<td><code>set(Object)</code></td>
<td>Stores an object in the database</td>
</tr>
<tr>
<td><code>delete(Object)</code></td>
<td>Deletes a database object</td>
</tr>
<tr>
<td><code>activate(Object, int)</code></td>
<td>Fills in fields of this object</td>
</tr>
<tr>
<td><code>commit()</code></td>
<td>Commits the current transaction</td>
</tr>
<tr>
<td><code>rollback()</code></td>
<td>Rolls back the current transaction</td>
</tr>
<tr>
<td><code>ext()</code></td>
<td>Accesses advanced functions</td>
</tr>
<tr>
<td><code>close()</code></td>
<td>Closes the database</td>
</tr>
</tbody>
</table>
Query by Example

- Perhaps the most unique aspect of db4o
- Fill in desired fields in an example object
  - Including relationships and their properties
  - Null references or default primitives ignored
- Call `ObjectContainer.get(Object)`
- Nice but…
  - Only equality comparisons are possible
  - Can’t look for null or default values
S.O.D.A. Queries

- The alternative to query-by-example.
- “Simple Object Database Access”
- Queries are built using an object-based query graph.
  - Nice for auto-generating queries
  - Hard to include queries in properties files!
S.O.D.A. Example

Query q = db.query();
q.constrain(Employee.class);
q.constrain(new Evaluation() {
    public void evaluate(Candidate c) {
        c.include(((Employee) c.getObject()).isRetired())
    }
});
q.descend("department")
    .descend("payPeriodDays")
    .constrain(new Integer(14));
ObjectSet results = q.execute();
Many object databases use JDO…
  - But some are proprietary

**db4o** is an example of a proprietary object database system.
  - Very close to the Java object model.
  - Not suitable for large-scale systems or high reliability environments.
  - Query by example or S.O.D.A.
Concluding Thoughts
Putting It All Together

- EJB-CMP
- JDBC
- JDO
- Hibernate
- db4o

Relational Object
Resources

- http://www.ambysoft.com/mappingObjects.html
  More about designing an object-relational database, but there’s good info here.

- http://www.hibernate.org/
- http://access1.sun.com/jdo/
- http://www.db4o.com
Resources

- **Java Persistence for Relational Databases**
  by Richard Sperko

- **Java Database Best Practices**
  George Reese

- **White Papers by Scott Ambler**
Other Options

- We can’t cover everything in one night.
- Oracle TopLink
- Castor JDO (confusingly not JDO-based)
- CocoBase
- Apache OJB
- And more…

Questions?