JB297

Hibernate: A JBoss Technology

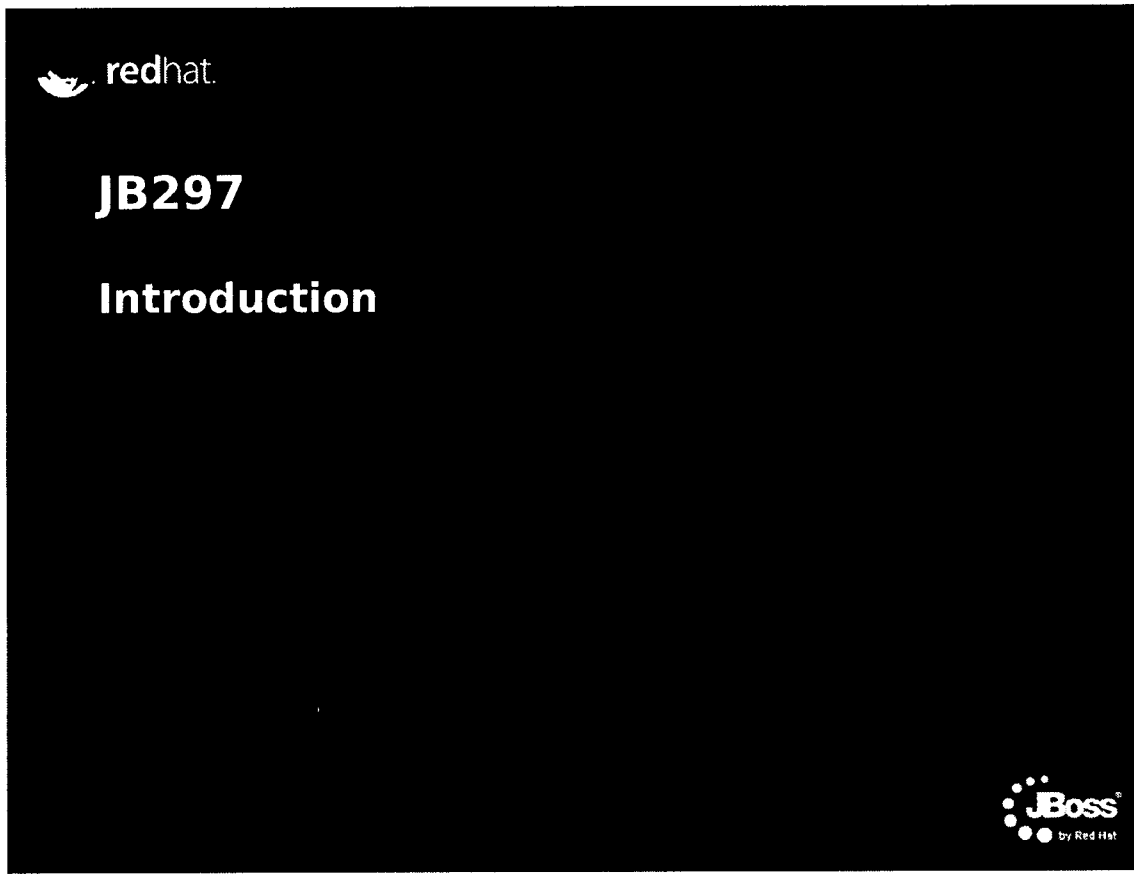
JB297-JBOSS-en-2-20110401

RED HAT **TRAINING**

JB297

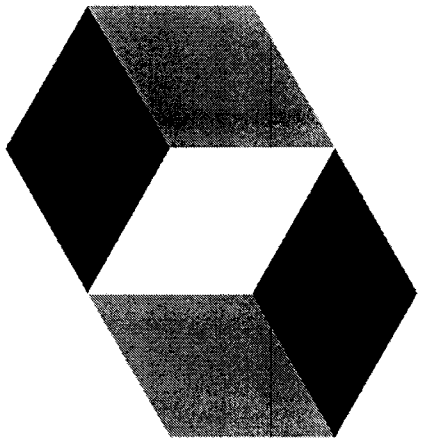
Hibernate: A JBoss Technology

JB297-JBOSS-en-2-20110401



JBoss by Red Hat

Roadmap



- **Who is Everyone ?**
  - **Background**
  - **Previous Experience**
  - **Hibernate Experience (O/R Mapping, JPA, Hibernate)**
  - **Interesting Projects**

Expectation of JB297 ?

- **Unique Question(s)**
- **Hands-On Practice**
- **Discussion Topics**

JBoss Hibernate - Agenda

Day 1:

- Understanding Java Persistence
  - Essentials
  - Paradigm Mismatch
  - Hibernate 3 Framework
  - Hibernate, Java Persistence and EJB 3.0
- Getting Started
  - First Hibernate Application
  - Essential Hibernate Mapping
  - Lab 1 (Mapping a simple Component, create query, add tx, persist, delete)
- Hibernate Projects and Tools
  - Hibernate Projects
  - Hibernate Tools
  - Lab 2

JBoss Hibernate - Agenda

Day 2:

- Advanced Hibernate Mapping
  - Entity and Value Types
  - Identity
  - Further Mappings
  - User-defined Types
  - Lab 3
- Entity Relations and Inheritance
  - Entity Relations
  - Inheritance Strategies
  - Lab 4
- Persistence Context, Session and Transaction
  - Cascading
  - State
  - Session and Transactions
  - Lab 5

JBoss Hibernate - Agenda

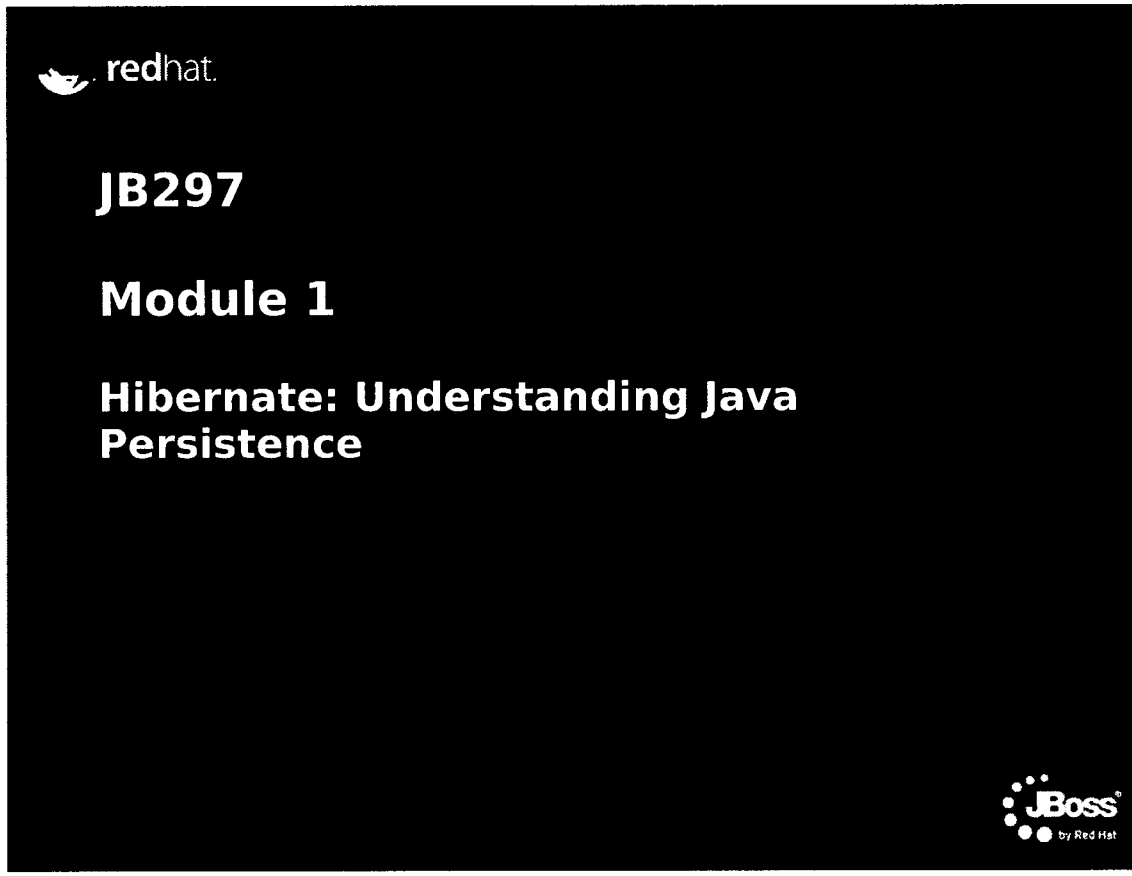
Day 3:

- Querying Data
  - Lab 6
- Hibernate Application Design
  - Layers
  - Use Case Examples
  - Data Validation
  - DAOs and DTOs
  - Conversations with Hibernate
  - Audit Logging, Hibernate Event System
  - Open-Session-In-View Pattern
- Performance Tuning and Caching
  - Bulk Operations
  - Caching
  - Efficiently Querying Data
  - Lab 7

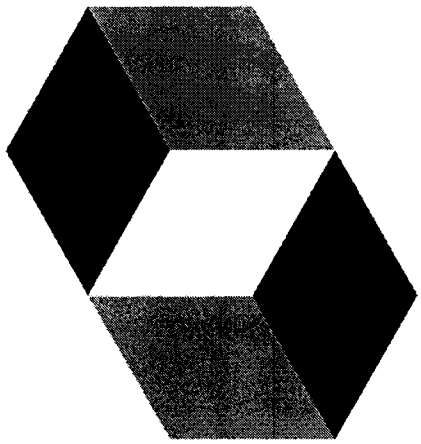
JBoss Hibernate - Agenda

Appendix:

- Testing Frameworks
  - Testing
  - DB Unit
- Advanced Frameworks (Optional)
  - Full Text Search
  - Data Historization / Multi Version Storage



Roadmap



- Essentials
- Paradigm Mismatch
- Hibernate 3 Framework
- Hibernate, Java Persistence and EJB 3.

Hibernate Essentials: Persistence...

- Short description
  - Hibernate helps you to persist Java Objects in relational data bases
  - Hibernate provides a query language to load objects from the database
  - Basically: You work with Java objects and the Hibernate API. You don't care too much about the database

Hibernate Essentials: Persistence...

- Persistence
  - Persistent data exists outside of an application's active memory.
  - In Java, persistence is generally discussed in the context of managing data in a relational database using SQL.
  - Persistent data is fundamentally necessary for most applications.

Hibernate Essentials: Persistence

- In Object-Oriented Applications
  - Domain Model vs. Tabular Data for Entity interaction and abstraction.
  - Object graphs - *a single or vast network of interconnected objects* - can be persisted in this manner.
  - "Subgraph" persistence is needed since most Java applications mix "transient" and persistence objects.

Persistence : Choices & Alternatives

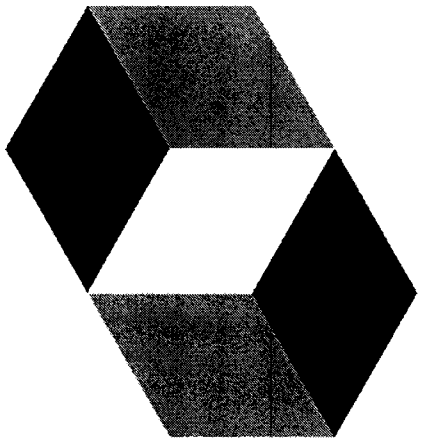
- JDBC - SQL Layer
  - JDBC powerful toolset for SQL-savvy developers, also well associated with DAO pattern.
  - Manual process consumes & can extend development effort.
- Serialization
  - Serialization is the Java-based persistence utility that offers an object-graph snapshot, albeit accessed as a whole (marshall/unmarshall).
- XML
  - XML is an alternative to byte-stream serialization, however it introduces the object/hierarchical mismatch

Object/Relational Mapping (ORM) - Distilled

- Automatic persistence of Java objects to RDBMS tables
  - Metadata describes the object-to-relational tables mapping
  - Metadata handling is abstracted by the ORM solution

- A Full ORM Strategy Snapshot:
  - An API to define queries pointing to classes/class properties
  - Mechanism for metadata mapping
  - API for standard CRUD operations on persistent class instances
  - A strategy for ORM interaction with transaction-aware objects

Roadmap



- Essentials
- **Paradigm Mismatch**
- Hibernate 3 Framework
- Hibernate, Java Persistence and EJB 3.

Object Relational Paradigm Mismatch

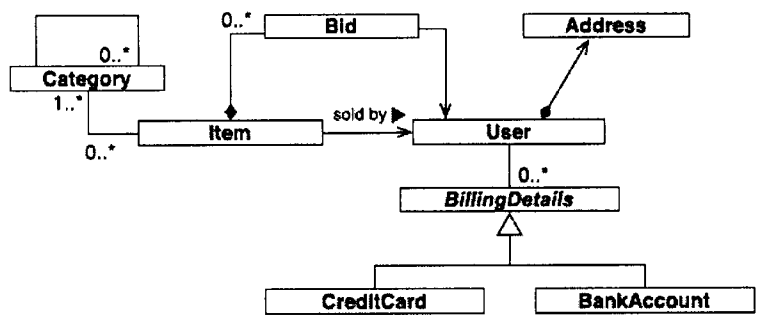
- Tabular Data – Relational Databases (RDBMS)
  - "Tables of Data"
  - ANSI SQL query results represented in tabular format
- Good Points
  - Straightforward - usage, syntax, constructs.
  - Excellent for set operations.
- Bad Points
  - Business logic is in the RDBMS (PL/SQL).
  - Difficult to reuse.

Object Relational Paradigm Mismatch

- Domain Object Model - Idiomatic Java
  - Object Oriented in nature.
  - Rich vs. Anemic Domain Model.
- Good Points
  - Reusable polymorphic patterns (e.g. Composite, Strategy, Composite).
  - Rich associations & lifecycle relationships.
  - Excellent reusability (even across projects).
- Bad Points
  - Inheritance & Polymorphism remain crucial.
  - JDBC/SQL require considerable upkeep effort.
  - Some operations naturally more tabular.

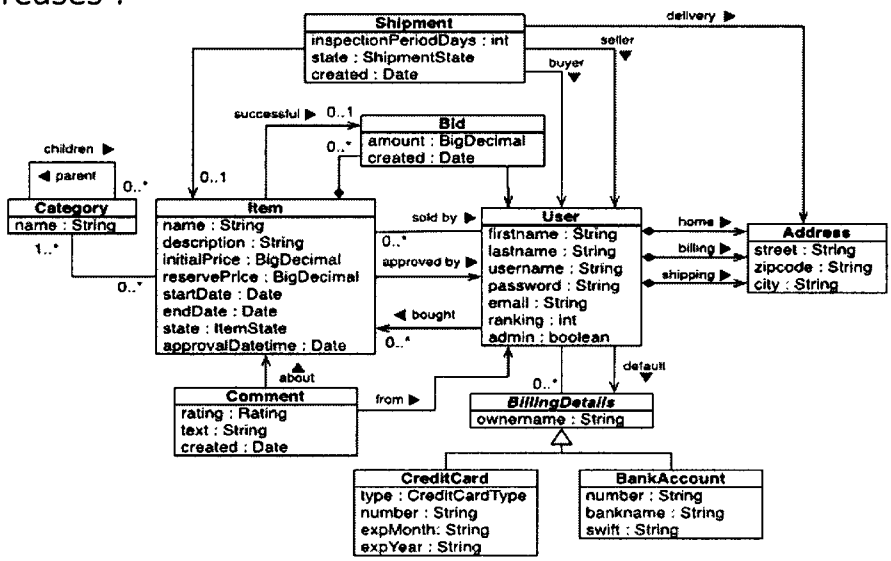
Paradigm Mismatch – Considerations

- Relationship between “Entities”?



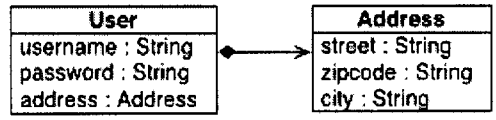
Paradigm Mismatch – Considerations

- What happens when the number of Entities & Relationships increases ?



O/R Mismatch: Issue of Granularity

- Granularity refers to relative "size" of a *type*.



- Can a new SQL datatype (UDT) be created?

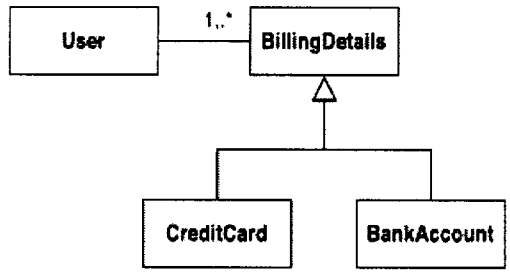
```
create table USER (USERNAME varchar not null primary key,
                   PASSWORD varchar not null,
                   ADDRESS address not null )
```

- We usually require built-in datatypes.

```
create table USER (USERNAME varchar not null primary key,
                   PASSWORD varchar not null, ADDRESS_STREET varchar not null,
                   ADDRESS_CITY varchar not null, ADDRESS_ZIP varchar not null)
```

O/R Mismatch: Issue of Subtypes

- Java understands notions of Inheritance (super-classes & subclasses).



- RDBMS tables are not types – no polymorphic associations represented.
  - Possible but DBMS specific/conceptually questionable

- Can one relation extend another relation?

```
create table USER (. . .)
create table BILLING_DETAILS (. . .)
create table CREDIT_CARD extends BILLING_DETAILS
```

O/R Mismatch: Identity of Identity

- Java understands 2 notions of "sameness":
  - Equality of Value – `equals()`
  - Object Identity – `x == y`

- RDBMS sameness is understood as the `primary-key` value

- Neither `equals()` nor object identity is "naturally" equal to PK value.

```
a.getId().equals.b.getId()
```

O/R Mismatch: Issue of Navigation

- Java understands navigation as "walking the object graph" via dot notation:

```
anItem.getSeller( ).getAddress( )
```

- RDBMS navigation is best handled via multiple joins:

```
select * from EMPLOYEE e where e.EMP_ID = 345
```

- Joins can also be used to navigate subsets of the object graph.

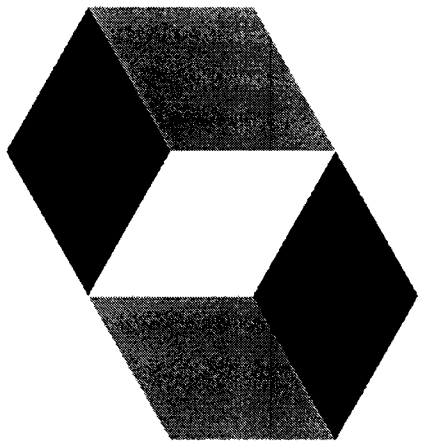
O/R Mismatch: Issue of Associations

- Associations describe relationships between entities.
- Object-references (Pointers)/Collections of Objects:
  - O-O style association representation (Java).
- Foreign Key Column/PK Value Copies:
  - RDMBS association representation (SQL Schema).
- The Mismatch:
  - Pointers are Directional
  - Many-to-many Multiplicity via “link tables”

JB297 - Hibernate
Copyright © 2011 – Red Hat Inc.

JBoss by Red Hat

Roadmap



- Essentials
- Paradigm Mismatch
- **Hibernate 3 Framework**
- Hibernate, Java Persistence and EJB 3.

Hibernate: Framework – Application Platform

- Hibernate Core APIs:
  - Session / SessionFactory API
  - Transaction API
  - Hibernate Query API
- Hibernate EntityManager -
  - EJB 3.0 Persistence specified lifecycles and API
  - Wrapper for EJB 3.0 with Hibernate Annotations
- Hibernate Annotations
  - Logical Mapping Category
  - Physical Mapping Category
- Hibernate Tools API

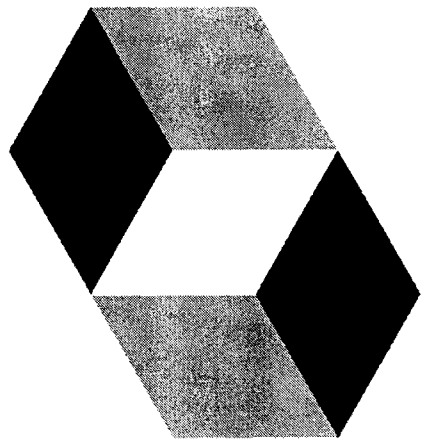
JB297 - Hibernate
Copyright © 2011 – Red Hat Inc.

Hibernate: Framework – Application Platform

- Hibernate Validation
  - Data validation: length, integer range, many more, customizable
- Hibernate Search
  - Full text search engine
  - Transparent integration with Lucene
  - Hibernate Search API
- Hibernate Envers
  - Historization of data
  - Multi version storing
- Hibernate Shards
  - Data Historization / Multi Version Storage

JBoss by Red Hat

Roadmap



- Essentials
- Paradigm Mismatch
- Hibernate 3 Framework
- **Hibernate, Java Persistence and EJB 3.**

EJB 3.0 & Hibernate: A Clarification of Standards

Enterprise JavaBeans 3.0

- Industry Standard
- EJB Programming Model
- Full/Partial EJB 3.0 implementations supported
- Java Persistence API (JPA)
- Core Contracts for EJB 2.1 compatibility
- JPA 2.1
  - O/R Mapping Metadata
  - Persistence Manager APIs
  - A Query Language (JPQL)

Hibernate Application Platform

- Full O/R tooling with ORM benefits
- Super-set of Java Persistence API
- Enterprise support
- Hibernate Core & Extended APIs
- Vendor Independence - Productivity
- Performance – Maintenance
- Java Containers
  - Application Servers
  - Web Containers
  - Java EE Runtimes
  - Java SE Runtimes

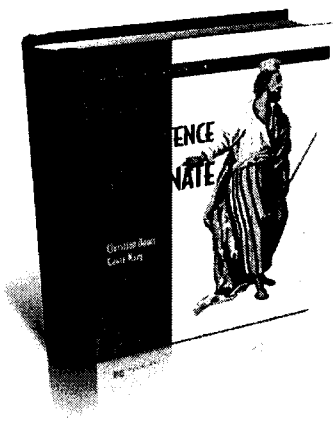
JBoss by Red Hat

Hibernate: Java EE Containers

- Hibernate is the persistence engine of the JBoss Application Server:
  - Full EJB 3.0 container.
  - Implements other JEE 5.0 standards.

- EJB 3.0 containers are no longer the monolithic beasts of the EJB 2.x era:
  - Easy deployment and management.
  - Available in an embeddable version, runs in any Java application or even Tomcat.

Resources: Hibernate & Java Persistence



- Main Hibernate Wiki
  - https://www.hibernate.org/
- Java Persistence with Hibernate
  - http://manning.com/bauer2/
- JBoss Hibernate Resource
  - http://www.jboss.com/products/hibernate/
- JSR-000317 Java Persistence 2.0
  - http://jcp.org/en/jsr/detail?id=317

JBoss by Red Hat

Summary

- In this lesson, you learned about:
  - Essentials of object relational persistence with Hibernate.
  - Choices and feature sets for Java persistence.
  - Understanding of Object Relational Mapping (ORM) .
  - The paradigm mismatch between relational/tabular and domain models and the challenges Hibernate solves for developers.

Lab: Class Lab Environment Set Up

Requirements

- JDK 6.0 on any operating system
- JBoss Developer Studio IDE
- Time: 15 Minutes

Goal

This lab is a "Fog-on-the-Mirror" test. It determines if students' workstations are configurable and have proper resources to successfully complete lab activities.

Description

In this lab you will configure your development environment for use with and without an IDE. Throughout this lab, we refer to the instructor's "coursematerials" folder. On Linux, this is linked on the desktop. On Windows, it can be found at **\\instructor\CourseMaterial\coursemateral**.

Project Instructions

TASKS: Install, Deploy, and Test JBoss Developer Studio IDE with Proper JDK

1. Install Java (Sun JDK-1.6.0)
   i. Linux: run **sudo yum -y install java-1.6.0-sun-devel** from a terminal window
   ii. Windows: Copy the Java 1.6 installer from the **installers** directory under the **coursematerals** directory and run on your local system. Select the default values, except for the following options:
      a) On the Custom Setup screen, change the install location to **C:\Java\jdk1.6.0**
      b) On the Java Setup – Destination Folder screen, change the install location to **C:\Java\jre6**
      c) If a registration window opens, close it. No need to register.

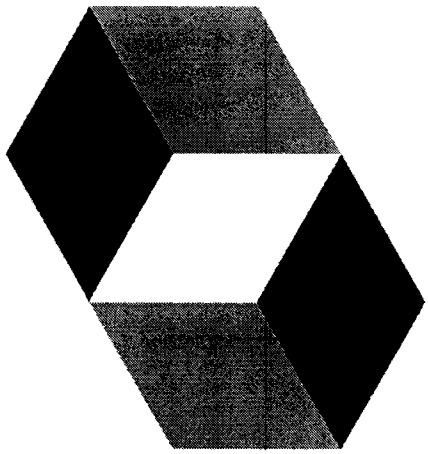
2. Copy the **JB297** folder from the **coursematerials** folder:
   i. Linux: Copy to **/home/student**
   ii. Windows: Copy to **C:**

3. Install JBoss IDE (JBDS Version 4.0)
   i. Copy the installer to your system from the **coursematerials** directory:
      a) Linux: jbdevstudio-linux-gtk-4.0.0.GA.jar
      b) Windows: installers\jbdevstudio-win32-4.0.0.GA.jar
   ii. Run the installer from a command prompt using java -jar jbdevstudio*.jar
      a) Windows users may also simply double click the installer jar file.
   iii. Accept all defaults, except the following:
      a) On the Select Java VM screen, set to:
         1. Linux: **/etc/alternatives/java_sdk**
         2. Windows: **C:\Java\jdk1.6.0**
   iv. Start JBDS and create an Eclipse workspace
      a) The workspace location should be:
         1. Linux: **/home/student/workspace**
         2. Windows: **C:\workspace**
      b) After installation, a "Welcome" Screen appears – click the Workbench arrow to enter IDE Workspace
      c) Change "Perspective" (top-right corner) to Java or JavaEE
      d) Add a variable to the classpath for Java apps
         1. Open **Window > Preferences**
         2. Expand **Java > Build Path** and click **Classpath Variables**
         3. Click **New...**
         4. Enter the name **M2_REPO** and the path is

/home/student/JB297/labs/libraries/M2_REPO (Linux) or
C:\JB297\labs\libraries\M2_REPO (Windows)

JBoss by Red Hat

Roadmap



- **First Hibernate Application**
- Essential Hibernate Mapping
- Lab 1

The First Persistent Class

```
public class Message {
  private Long id;
  private String text;
  private Sender sender;

  protected Message() {}
  public Message(String text, Sender sender) {
       this.text = text;
       this.sender = sender;
  }

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

  public String getText() {return text;}
  public void setText(String text) {
    this.text = text;
  }

  public Sender getSender() {return sender;}
  public void setSender(Sender sender) {
    this.sender = sender;
  }
}
```

- POJO model for persistent classes
  - Utilize JavaBeans Programming Model
  - Naming conventions – Accessors/Mutators
  - "id" Property has unique value for Message instances
  - "No-Args" constructor required, at least protected visibility to allow Hibernate instantiation via Reflections

The First Mapped Class

```
@Entity
@Table(name = "messages")
public class Message {

@Id
@GeneratedValue
private Long id;

@Column(nullable = false)
private String text;

@ManyToOne(cascade = {CascadeType.MERGE,
CascadeType.PERSIST})
@JoinColumn(name = "message_sender_id")
private Sender sender;

...
```

- *A* Mapped Class – Annotated.
  - `@Entity` – Specifies that the class is an entity.
  - `@Table` – Specifies the primary table for the annotated entity.
  - `@Id` – defines an association between Entities/Collections.
  - `@GeneratedValue` – Specifiy `PK` value generation strategies.
  - `@Column` – specify a mapped column for a persistent property or field.
  - `@ManyToOne` – Specify a mapped column for joining an entity association.
  - `@JoinColumn` – Specify a mapped column for joining an entity association. `<CascadeType>` – Operations that are propagated to the associated entity.

Configuration

- Configuration file *persistence.xml*
  - Mapping Details for Domain Objects
  - Database Connection Details

```
<persistence  xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">
 <persistence-unit name="samplePU" transaction-type="RESOURCE_LOCAL">
  <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <non-jta-data-source/>
  <properties>
   <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver"/>
   <property name="javax.persistence.jdbc.user" value="sa"/>
   <property name="javax.persistence.jdbc.password" value=""/>
   <property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:."/>
   <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
   <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
   <property name="cache.provider_class" value="org.hibernate.cache.NoCacheProvider"/>
 </properties>
 </persistence-unit>
</persistence>
```


Configuration

- Default: classpath is scanned for annotated classes
- Be explicit if scanning takes to long

```
<persistence-unit name="samplePU" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <non-jta-data-source/>
    <class>com.jboss.sample.Message</class>
    <class>com.jboss.sample.Sender</class>
    <exclude-unlisted-classes/>
...
```

Store Persistent Objects With Hibernate. . .

```
public class HelloWorld {

 public static void main(String[] args) {
   // Start EntityManagerFactory
   EntityManagerFactory emf = Persistence
       .createEntityManagerFactory("samplePU");

   // First unit of work
   final EntityManager em =
      emf.createEntityManager();
   em.getTransaction().begin();
   em.persist(new Message("Hello world",
      new Sender("Holger")));
   em.getTransaction().commit();
   em.close();
  }
}
```

- Introducing Persistence Context
  - EntityManager
    - Factory
    - JPA 2.0
    - Hibernate
  - Handle Persistence Context
    - Unit-of-work
    - Finite Persistent States
  - Load/Store with Hibernate
    - Transactions
    - Fetching (Eager/Lazy)

Store Persistent Objects With Hibernate. . .

```
final EntityManager em = emf.createEntityManager();
try {
 em.getTransaction().begin();
 em.persist(new Message("Hello world",
  new Sender("Holger")));
 em.getTransaction().commit();
} catch (RuntimeException relevantException) {
 if (em.getTransaction().isActive()) {
  try {
   em.getTransaction().rollback();
  } catch (PersistenceException rollBackException) {
   logger.warn("Rollback of open transaction failed",
     RollBackException);

//less noisy without second stacktrace
//logger.warn("Rollback of open transaction failed");
  }
  throw relevantException;
 }
} finally {
  em.close();
}
```

- Exception Handling
  - Discard the entity manager
  - Try to rollback transactions to release resources on the database
  - Rollback may fail as well => log it to inform administrators
  - Close the entity manager
- Recommendation
  - JEE container handle the exceptions for you
  - AOP (JBoss Weld, Google Guice, ...) helps to solve this as well

Fetch Persistent Objects with Hibernate

```
final EntityManager em =
emf.createEntityManager();
em.getTransaction().begin();

Sender reloaded = em.find(Sender.class,
  sender.getId());

final List<Message> resultList =
  em.createQuery(
    "select m from Message m order by m.text",
    Message.class)
  .getResultList();
System.out.println(String.format(
   "Found %s messages", resultList.size()));
for (Message message : resultList) {
    System.out.println(message);
}

em.getTransaction().commit();
em.close();
```

- Querying data
  - By a given id using *find*
  - HQL
  - Criteria (JPA)
  - Criteria (Hibernate)
  - SQL

Fetch Persistent Objects with Hibernate

```
final TypedQuery<Sender> query = em.createQuery(
"select s from Sender s where s.name like :name and
   s.birthday < :birthday",
   Sender.class);

final Calendar tenyearsago = Calendar.getInstance();
tenyearsago.add(Calendar.YEAR, -10);

query.setParameter("name", "P%")
  .setParameter("birthday",tenyearsago.getTime());
final List<Sender> list = query.getResultList();
for (Sender sender : list) {
  System.out.println(sender);
}
```

- Where conditions
  - Mostly like SQL
  - You always refer to class attribute names not to database columns

Updating objects

```
sender.setName("Correct name");


final EntityManager em =
emf.createEntityManager();
em.getTransaction().begin();
Sender newSenderInstance = em.merge(sender);
em.getTransaction().commit();
em.close();


// Hibernate
Session session = (Session) em.getDelegate();
session.update(sender);
// alternatively
session.buildLockRequest(LockOptions.NONE).lock
(sender);
```

- JPA
  - merge
  - Merges the data of the current instance into a new instance
- Hibernate is more flexible
  - merge
  - update
  - lock

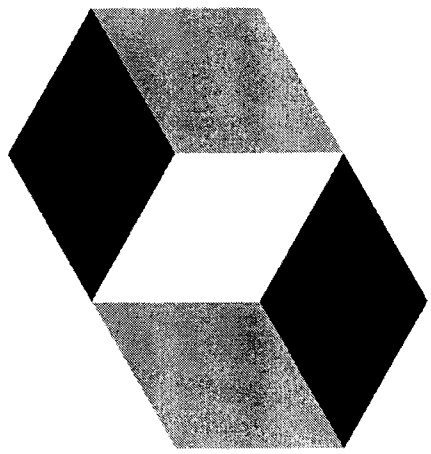
Deleting objects

```
sender.setName("Correct name");


final EntityManager em =
emf.createEntityManager();
em.getTransaction().begin();
Sender newSenderInstance = em.merge(sender);
em.getTransaction().commit();
em.close();


// Hibernate
Session session = (Session) em.getDelegate();
session.update(sender);
// alternatively
session.buildLockRequest(LockOptions.NONE).lock
(sender);
```

- JPA
  - remove
  - Entity must be in persistent state (just merged or loaded with a query)
- Hibernate
  - delete

JBoss by Red Hat

Roadmap



- First Hibernate Application
- **Essential Hibernate Mapping**
- Lab 1

Basic Mappings

```
@Entity
public class Sender {

// name is not nullable, database column
// name is 'column_name',
// table generation uses the following
// constraints: char(10) unique not null
@Column(
  nullable = false,
  unique = true,
  name = "column_name",
  length = 10)
private String name;

// a java.util.Date stored as date
// TIME and TIMESTAMP exists as well
@Temporal(TemporalType.DATE)
private Date birthday;

public enum Sex { MALE, FEMALE }
@Enumerated
private Sex sex;
```

- Default: every property is mapped
- But
  - Specify constraints (unique, nullable)
  - Hints for DDL generation (length)
  - Data validation (length, numeric range, email, custom validation) → is discussed later

Basic Mappings - Components

```
@Entity
public class Sender {
@Embedded
private Address address;
...
}

@Embeddable
public class Address {

private String street;
private String city;
private String country;

public String getStreet() {
return street;
}

public void setStreet(String street) {
this.street = street;
}
...
```

- Entities can contain components
  - Sample: Sender contains an Address class
- Fine grained entities (Recommended)
- Good object oriented design
- Reduces complexity of entities
  - 4 components with 10 attributes versus 40 attributes in a single class

Illustration: Hibernate Application "Setup"

- Using Standard Layout
  - Standard Layout
  - Ant Frameowork
  - `build.xml` project's build directory
- Using Hypersonic (in-memory) RDBMS
  - Proof-of-concept database
  - http://hsqldb.org
- Using Standard Enterprise Application Layout
  - `src/java`
  - `src/etc`
  - `src/webapp`

Lab 1

Simple Class & Property Mappings with Hibernate & JPA

- Details and instructions for the lab can be found in the lab guide.
- The instructor will answer any questions and will determine the stop time.

Summary

- In this lesson, you learned about:
  - All components of a first Hibernate application
  - Fundamentals of Hibernate POJOs / Entities.
  - Working with objects
  - Basic mappings

Lab 1: Simple Class & Property Mappings with Hibernate & JPA

Requirements

- JB297-Hibernate Lab Archives
- Pre-Configured JBoss Developer Studio IDE
- Time: 30 Minutes

Goal

This lab demonstrates basic mappings and working with entities

Description

- In this lab you need to complete mappings and code. There are a number of tasks and each tasks can be validated running the Junit tests.

- The tasks are marked with TODO markers. Search each of the source files listed below for Tasks

  Example Task:
  /* TODO: A Task Title
  * Step 1: Code Something Here, As Instructed - Tags
  */

- You can find a detailed description below.
- Import JB297's **lab-1** project as an existing project, copied to the workspace
- The working/full source code is available in the lab-1/solution/ directory. View this directory for help, or backup the lab-1/src/ directory, rename the solution folder to src.
- Start the database using the **runDatabase** script found in **JB297/labs/hsqldb**

Project Instructions

---

1. Complete the configuration
   i. The dialect of Hibernate is not properly configured. Check the *persistence.xml* and set the dialect for the HSQL database
   ii. You can find all available dialects by looking for sub classes of *org.hibernate.dialect.Dialect* under the hibernate-core-3.6.0.Final.jar in Referenced Libraries.
   iii. Run the test *checkConfiguration* in the class *HelloWorldTest* to validate your changes. To run the test from JBDS, expand the **com.jboss.sample.HelloWorldTest** class under **src/test/java**, right-click the **checkConfiguration** method and select **Run As... > JUnit Test**. *Don't forget to start the database.*

2. Map persistent classes with basic JPA annotations:
   i. The entity *Sender* has three attributes which are not yet stored: name, birthday and sex.
   ii. Remove the Transient annotation from these three attributes.
   iii. Name should be mapped to the column *column_name* and should not be nullable. Birthday should be stored as date. Sex should be stored as Enumerated.
   iv. Complete the mappings in *Sender* to store the attributes.
   v. Run the *persistSender* test to validate your changes.

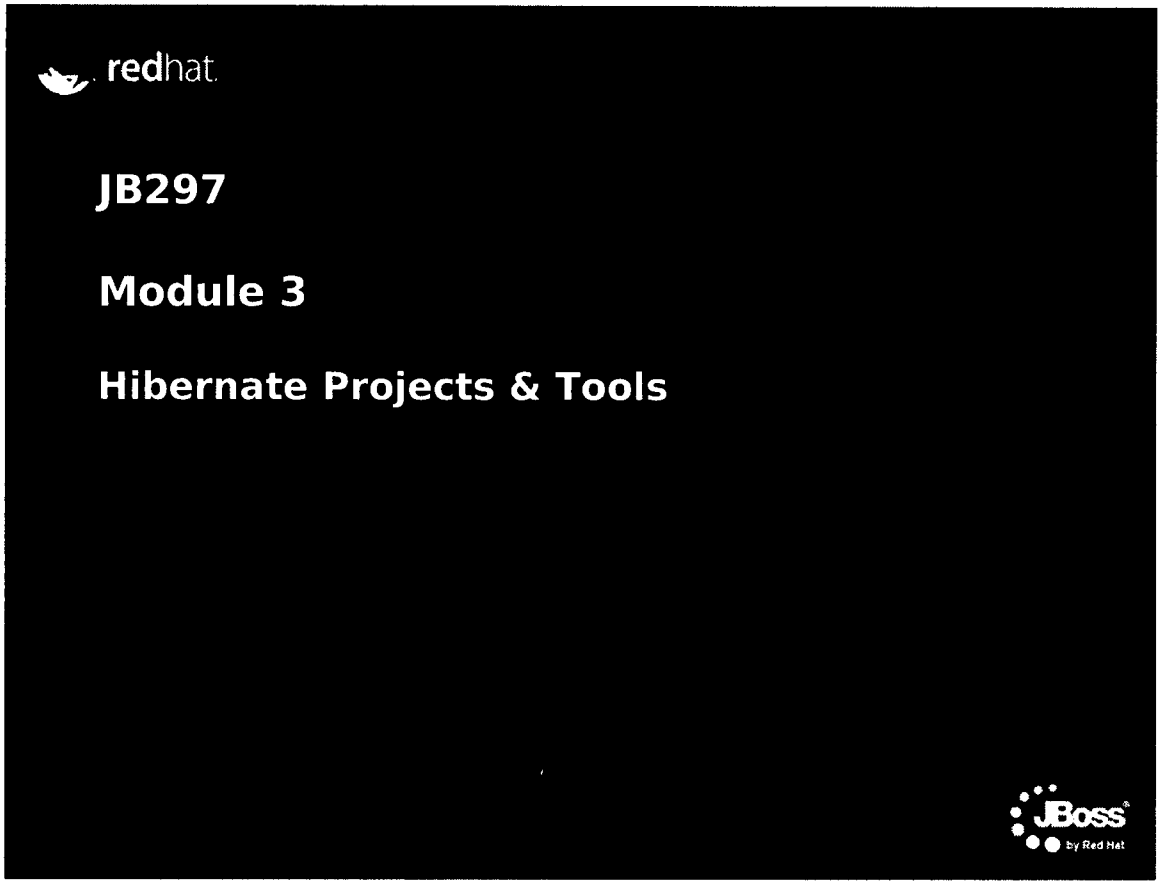
3. Map a component
   i. The entity *Sender* has an attribute *Address.* It is a component and currently not stored. Your task is to store the address.
   ii. Remove the Transient annotation from the address attribute.
   iii. Complete the mapping in *Sender* and *Address* to store the attributes via embedding.
   iv. Run the *persistSenderAddress* test to validate your changes.

4. Map a relation
   i. The entity *Message* has a relation to the *Sender.* Make sure that the relation is stored in the database.
   ii. Complete the mapping in *Message.*
   iii. Run the *createRelation* test to validate your changes.

5. Update a sender

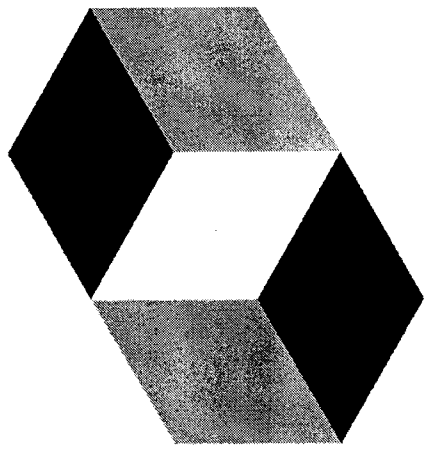
i. A sender was created and its name was changed from *Andrea* to *Davorka*. You need to add code to save the new name in the database.
ii. Complete the code in the *updateSender* method in *HelloWorldTest*.
iii. Run the *updateSender* test to validate your changes.
iv. The sex needed to be corrected as well and should be changed to *Sex.FEMALE*. Update the method *updateSender*.
v. Validate that you have used merge correctly and that both, the new name and the corrected sex, is stored in the database.
vi. Run the *updateSender* test to validate your changes.

6. Write queries
   i. Edit the *querySender* method.
   ii. Write a query to select all *Sender*.
   iii. Write a query to select all *Sender* named 'Holger'.
   iv. Run the entire class as a JUnit test to confirm your work.
   Note: the JUnit tests destroy the database each time in order to allow for a clean database each time. If you run the *querySender* method on it's own, it will not show you anything.



JBoss by Red Hat

Roadmap



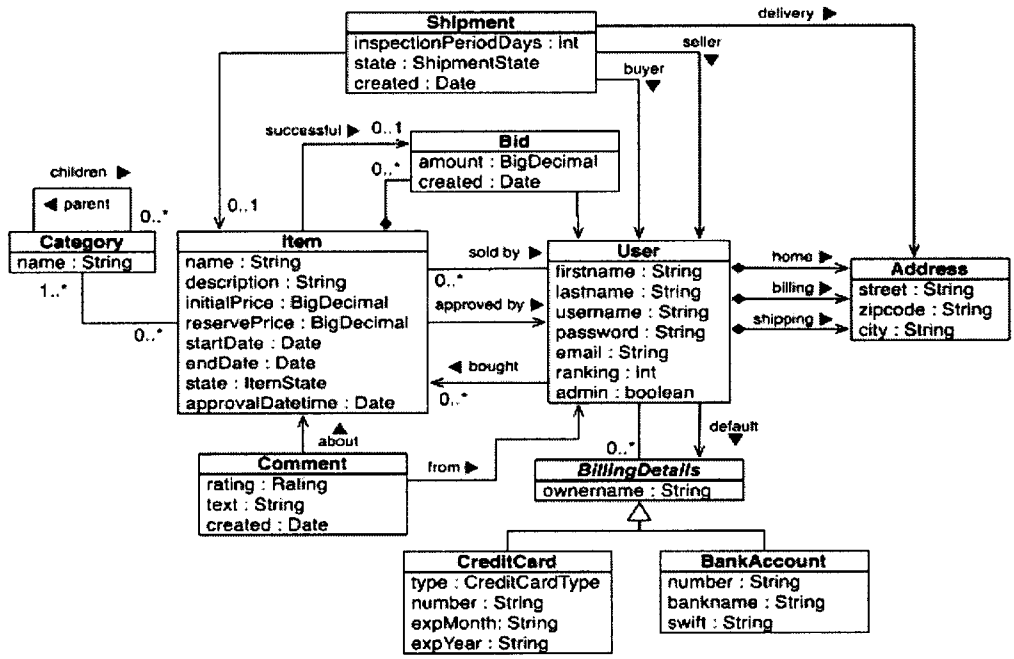
- **Hibernate Project Overview**
  - CaveatEmptor
  - Domain Analysis
  - Domain Model Classes
  - Persistence Requirements
  - Entities
  - Value Objects
- Hibernate Tools

CaveatEmptor - Project Example

- A Non-trivial Hibernate Project
  - The JB297 Hibernate Auction application
  - Non-trivial Project based on Hibernate's Caveat Emptor
  - Hibernate Core Reference Implementation

- Superset of Features within Hibernate
  - Integration and Development Toolkit
  - Hibernate Additional Features (over JPA 2.0 standard / RI)
  - Ease of Use
  - Elegant Pattern Implementation
  - Best Practices – New Development / Legacy Integration

The JB297 Example Application



Project Design Strategies

- Top down
  - Implement a Java (JavaBeans) object model
  - Write a mapping document by hand, or generate it from XDoclet tags, or use JDK 5.0 annotations
  - Export the database tables: hbm2ddl (SchemaExport) tool

- Bottom up
  - Start with an existing data model
  - Hibernate Tools will generate mapping documents (or annotated Java source)
  - hbm2java (CodeGenerator) tool to generate stub Java code
  - Fiill in the business logic by hand

Project Design Strategies

- Middle out
  - Express your conceptual object model directly as a mapping document
  - hbm2java tool to generate skeletal Java code
  - Fill in the business logic by hand
  - Export the database tables using the hbm2ddl tool

- Meet in the middle
  - Start with an existing data model and existing Java classes
  - Write a mapping document to adapt between the two models (you may require some minor refactorings of the object and / or data models)

First Task – Analyzing the Domain

- Experts identify the main business entities for the software system.
- Entities – Lightweight persistent domain objects – are the "nouns" understood by a system's users.
- A business model provides a conceptual view of the business.
- An object-oriented domain model captures the entities (Can be elaborate UML or mental model).

Domain Model Classes - Definitions

- Persistent Class
  - Persistent Class - An instance of this class can be made persistent.
- POJO (Plain Old Java Object)
  - POJO – The class has been implemented following Best Practices (JavaBeans convention).
- Entity Class
  - Entity – The class has been implemented according to the EJB 3.0 / Java Persistence 2.0 Specification.

Projects: Domain Model Considerations

- Separation of the business concerns from cross-cutting concerns (e.g. Transactions vs. Persistence)
- What manner of persistence is required (automatic or transparent)
- Is there a particular programming strategy to follow?
- Note:
  - Domain object models do not inherently encapsulate all the behaviors of the business that could change.
  - Domain object models may include functionality that is not likely to change.

JBoss by Red Hat

Project: Separation of Concerns

- The Domain Model implementation is a central, organizing component.
- Ensure that concerns that are not business aspects do not "leak" into the domain model implementation.
  - Example:
    - Code within the domain model should not be used to perform JNDI lookups or calls to the database.
- Code within the domain model should be able to be unit tested without special infrastructure.
- What about governances for persistence?

Persistent Class Requirements

- Hibernate & Entity (EJB 3.0) Rules:
  - No final Classes or methods
  - A no-args constructor is required
    - Constructor must be public or protected
  - Collection properties must be interfaces (not implementations)
  - Entity class must be top-level
  - An Enum or interface may not be a persistent class
  - Abstract or concrete classes can be persistent

JBoss by Red Hat

Persistent Class Requirements

POJO Class Best Practices

- POJO Classes:
  - Consider JavaBeans convention
  - Provide public mutators/accessors
  - Provide private fields
  - Declare Serializable for detached usage

- Note: Business logic is implemented thorough collaboration of domain objects.

Entity Example

```
@Entity
@Table(name = "USER")
public class User implements Serializable{

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private Long id;

    private Address addr;
    public User( ){ }

    public String getUsername() {return username;    }

    public Address getAddress() {return Address addr; }

    //Business Method
    public MonetaryAmount calcShipCosts(Address from){
     . . . }
 . . .
}
```

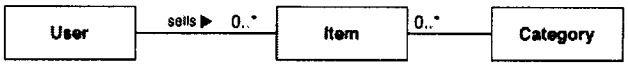
Projects: Pain-points & Persistence

- Transparent Persistence:
  - No interfaces to implement
  - No superclasses to extend
  - No intrusive programming model for persistent classes of the Domain Model
- Automated Persistence:
  - SQL is automatically generated
  - Low-level data set handling is automated
  -
- Note: Persistence concerns are automatically externalized to a generic persistence manager interface (Session and EntityManager)

JB297 - Hibernate
Copyright © 2011 – Red Hat Inc.

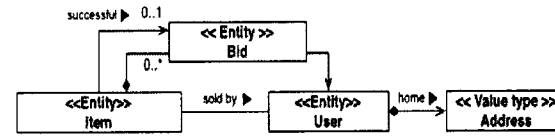
Projects: Entities – Objects of Importance

- Some of the types within an application are "more important" and others "less important".
- Persistent entities are coarse-grained types representing business entities:
  - Independent lifecycle
  - Has its own database identity
  - Has its own primary key value
  - Exists independently of any other instance
  - Supports shared references
- CaveatEmptor examples:
  - Item
  - User
  - Category



Projects: Value Objects – Finer Granularity

- Value objects – powerful, fine-grained object model components providing multiple objects per record within a database table.
- Value Objects – (a/k/a Value Types)
  - Are compositional in nature (UML Model)
  - No database identity
  - Dependent life cycle via its owning Entity
  - Does not support shared references



CaveatEmptor examples:

Address

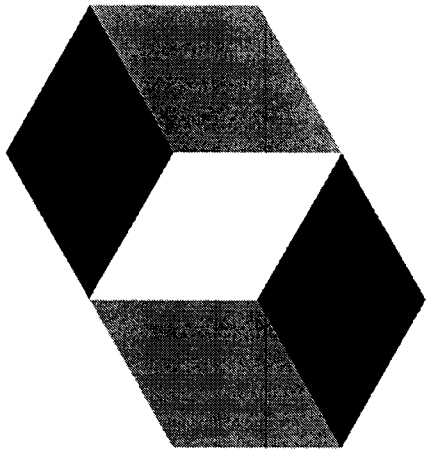
MonetaryAmount

Date

String

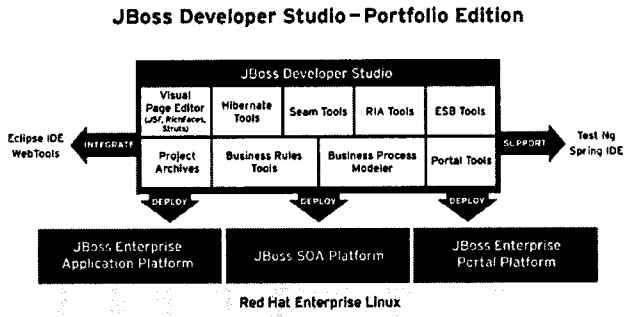
JBoss by Red Hat

Roadmap



- Hibernate Projects
- **Hibernate Tools & Tasks**

Quickstart: JBoss Developer Studio IDE (JBDS)

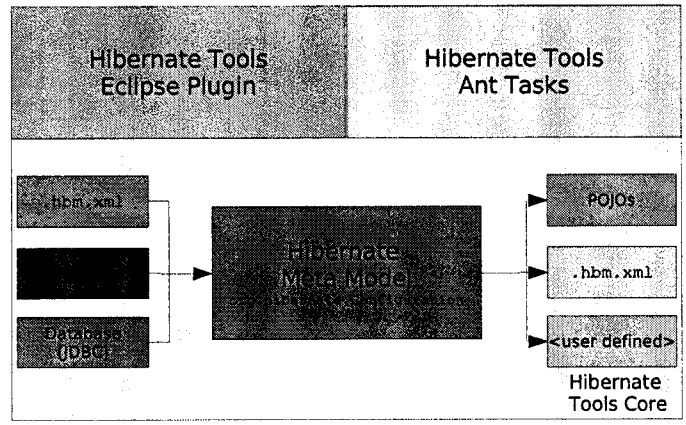


- JBoss Developer Studio (JBDS) is a set of Eclipse-based tools pre-configured for JBoss Platform.
- What is the benefit?
  - An installer
  - JBoss/RedHat support access
  - Eclipse/Web Tools pre-configuration
  - JBossEAP – JBAS & Seam
  - 3rd Party plugins pre-bundled/configured
- Supported platforms:
  - Linux
  - Mac OS X
  - Windows

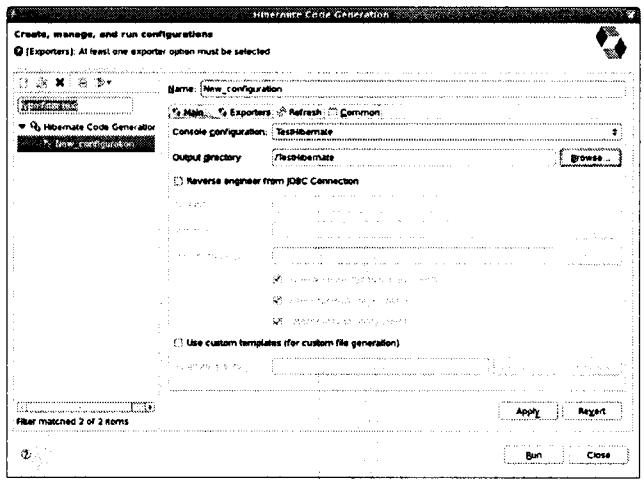
Quickstart: JBDS – JBoss Tools Modules

- JBoss Tools is an umbrella project for the JBoss-developed plugins within the Eclipse-based JBDS IDE
  - JBoss AS Tooling
  - Hibernate Tools
  - Seam Tools
  - Visual Page Editor
  - JST Tools
  - JBPM Tools
- *Note: JBoss Tools standalone project remain unsupported – use JBDS*

Quickstart: Hibernate Tools – Details - Internals



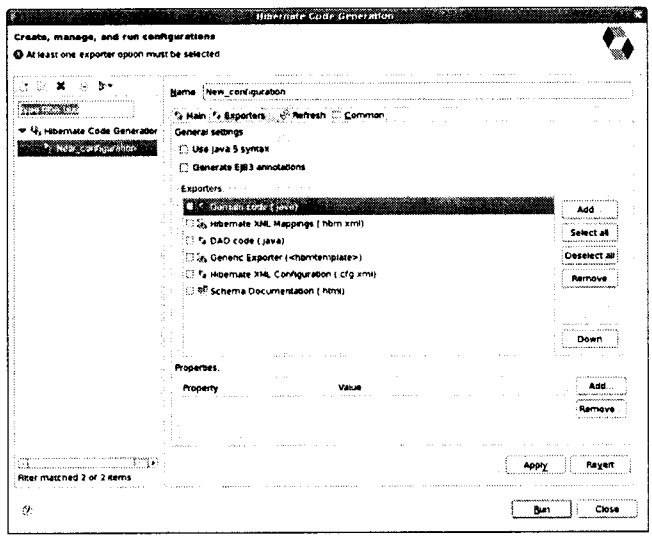
Quickstart: JBDS - Hibernate Tools – Wizards



JBoss Hibernate Tools - Code Generation Wizard

- Hibernate Tools/JBDS Plugins
  - Mapping Editor
  - Hibernate Console
  - Configuration Wizard
  - Code Generator
  - Eclipse JDT Integration

Quickstart: JBDS – Hibernate "Click & Generate"

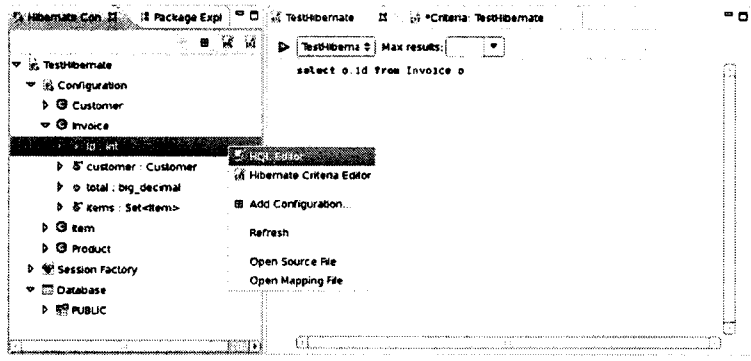


JBoss/Hibernate Tools - Code Exporters

- Hibernate provides a "Click & Generate" reverse engineering & code creation utility.
  - Creates artifacts from:
  - Legacy/Pre-existing RDBMS
  - Existing Hibernate configuration - (XML-based or annotated mappings)
- Within Hibernate perspective:
  - Click "Hibernate Code Generation" (see toolbar) ->Provide meaningful content/names
  - Click Exporters tab

Quickstart: JBDS – Query Prototyping

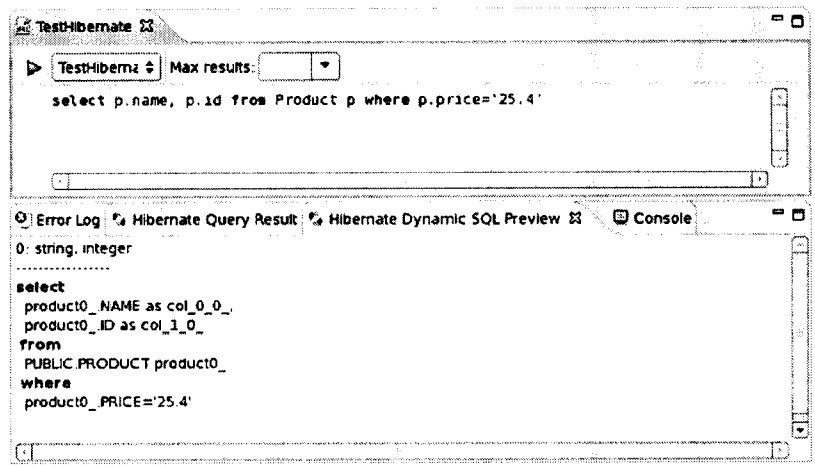
- Queries can be prototyped within the HQL editor.
- The HQL Editor is opened by right-clicking the Console configuration and select "HQL Scratchpad" (If disabled then create a `SessionFactory`)
- Executing the query is done by clicking the green run button in the toolbar or pressing Ctrl+Enter.
- *Note: HQL queries are executed using `list()` and without any limit of the size of the output. Could cause out of memory errors.*



Hibernate Query Editor

Quickstart: JBDS – Dynamic Query Translator

- If the "Hibernate Dynamic Query Translator" view is visible while writing in the HQL editor it will show the generated SQL for a HQL query.
- The translation is done each time you stop typing into the editor, if there are an error in the HQL the parse exception will be shown embedded in the view.



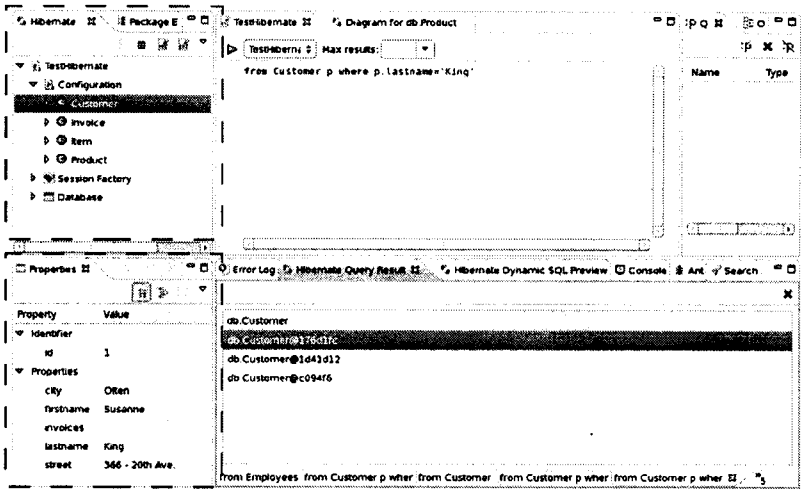
Hibernate Dynamic Query Translator View

Quickstart: JBDS – Hibernate's Perspective

- The IDE provides a a combined set of "views"
  - Hibernate Console - for observing mapped Entities/classes, structure, queries (view, prototype, test)
  - Properties views

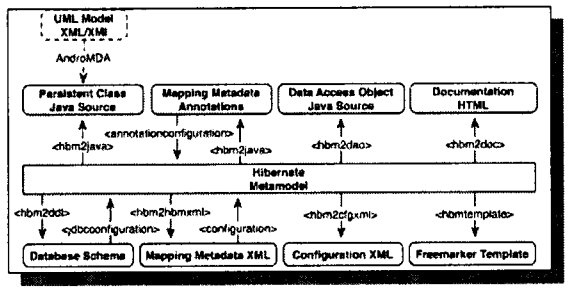
Console View: Mappings, structure, queries...

Properties view for selected entry



Quickstart: JBDS – More Control – Ant Tools

- The `hibernate-core.jar` (IDE plugins dir) contains Hibernate Tools core & Hibernate Ant functionality:
  - Ant Tasks (see build.xml)
  - Configuration (Supports 4 types)
  - Exporters (Meta-model conversion)
  - Fine-Grained Control
    - Reverse Engineering Strategies
    - POJO Code Generation



Hibernate Tool Tasks for Ant

Lab 2

Using Hibernate Tools to generate classes and mappings

- Details and instructions for the lab can be found in the lab guide.
- The instructor will answer any questions and will determine the stop time.

JBoss by Red Hat

Summary

- In this lesson, you learned about:
  - Understanding Hibernate Projects
  - Rich Domain Model
  - Hibernate Entity Classes
  - Hibernate Value-Objects
  - Available tools
    - Setting up projects from existing tables

Lab 2: Hibernate Tools

Requirements

- JB297-Hibernate Lab Archives
- Pre-Configured JBoss Developer Studio IDE
- Time: 30 Minutes

Goal

This lab demonstrates the use of Hibernate tools to generate entity classes, mappings and DAO to ramp up your persistence layer in short time.

Description

- In this lab you need to use Hibernate Tools to generate the classes. There are a number of tasks and each tasks can be validated running the Junit tests.

- The tasks are marked with TODO markers. Search each of the source files listed below for Tasks

  Example Task:
  /* TODO: A Task Title
  * Step 1: Code Something Here, As Instructed - Tags
  */

- You can find a detailed description below.
- Import JB297's **lab-2** project as an existing project, copied to the workspace
- The working/full source code is available in the lab-2/solution/ directory. View this directory for help, or backup the lab-2/src/ directory, rename the solution folder to src.

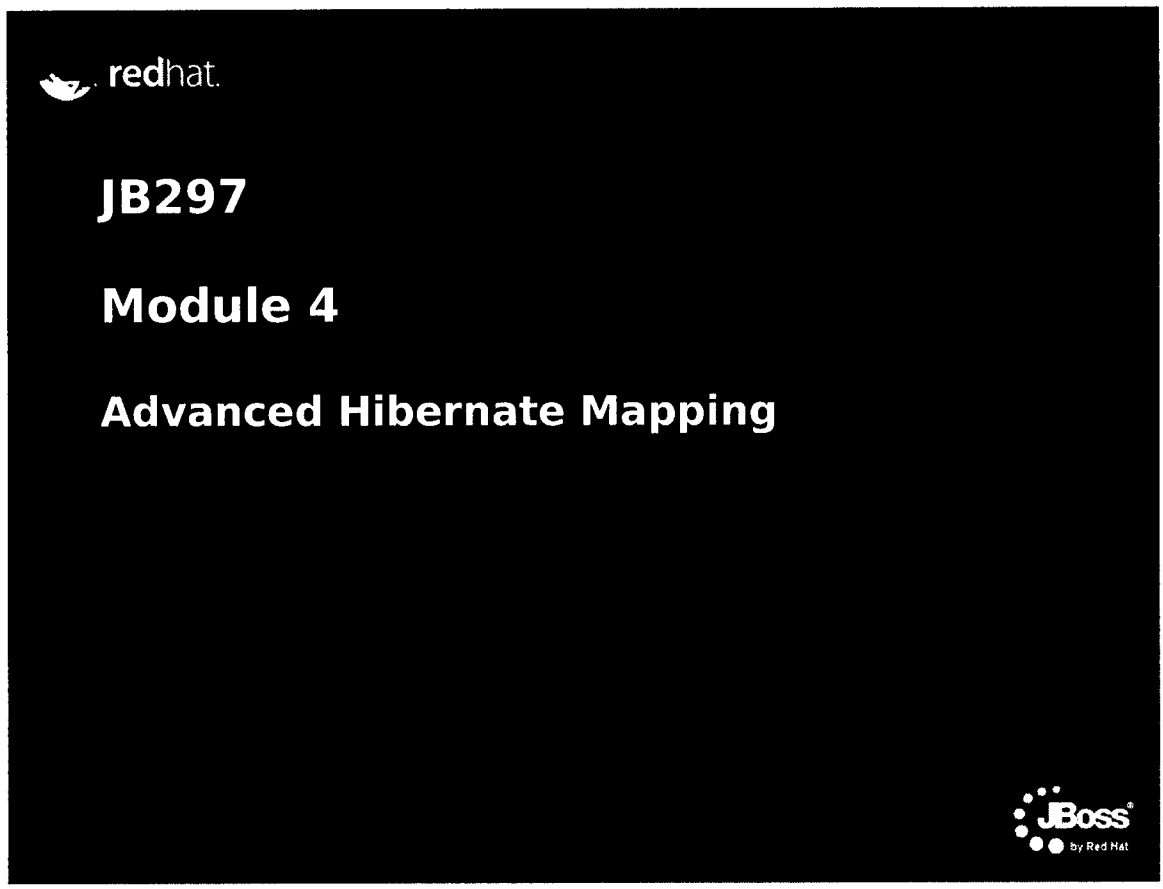
Project Instructions

1. In this lab, we will reverse engineer our Entities from an existing database. Therefore, we need to set up the database first.
   i. Start the database and the SQL-client using the runDatabase and runGuiClient scripts in the JB297/labs/hsqldb directory.
   ii. Copy the schema.sql content from the src/main/resources directory into the SQL-client and execute it.
   iii. You should see a full set of tables.

2. Generate classes with annotation mapping
   i. Open the Hibernate perspective from Window > Open Perspective... > Other...
   ii. In the Hibernate Configurations window, create a new 'Hibernate Console Configuration'
      a) Settings: type=JPA, project: lab-2, database connection=[JPA configured connection], Persistence Unit=samplePU
   iii. Select the icon 'Hibernate-Code-Generation-Configuration' in the top icon bar.
   iv. Create a new configuration
      a) Chose your console configuration
      b) Output directory: lab-2/src/main/java
      c) Select 'Reverse engineer from JDBC'
      d) Set 'auction.model' as package.
      e) On the exporters tab select
         Select 'Use Java 5'
         Select 'Generate EJB3 annotations'
         Select 'Domain code'
   v. Run the code generator.
   vi. Your application should compile.
      a) If it does not, a common issue is that the Address generated class has two instances of Users associated with it (and similarly between Users and Address). Simply delete one wherever it is seen in the code, and recompile. JBDS will assist in finding the duplicates.
      b) The lesson learned is that generation isn't always perfect!
   vii. Run the test *rampUpTest* in the class *MappingTest* to validate your generated classes.

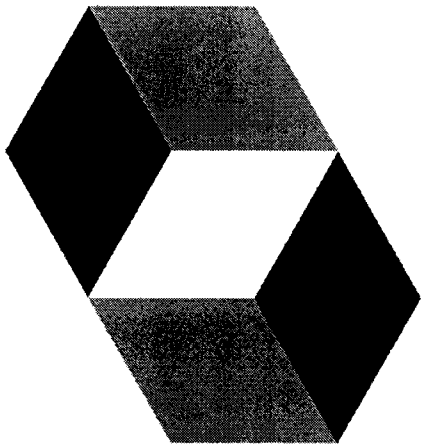
3. Component support
   i. The user is supposed two have two components: home address and billing address.

ii. Have a look at the generated classes. Can you find any address components? Discuss why or why not.



JBoss by Red Hat

Roadmap



- **Entity and Value Types**
- Identity
- Further Mappings
- User-defined Types
- Lab 3

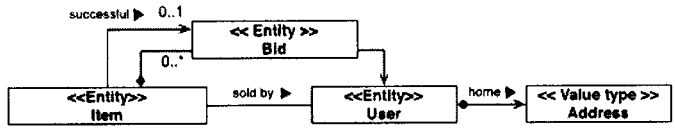
Entities

- Annotated with *@Entity*
- Id property
  - `EntityManager.find(SomeClass.class, 4711);`
- Independent lifecycle
  - `EntityManager.persist(myInstance),`
    `EntityManager.remove(myInstance);`
- Supports shared references

Value Types

- Components
  - Simple component, *@Embedded*
  - Collection of component classes, *@ElementCollection*
- Hibernate supported types
  - Basic JDK types
    - integer, long, short, float, double, big_decimal, big_integer, character, string, byte, boolean, yes_no, true_false
  - Date and time types
    - date, time, timestamp, calendar, calendar_date
  - Binary and large objects
    - binary, text, serializable, clob, blob
  - Other JDK types
    - class, locale, timezone, currency
- No id property, No independent lifecycle, No shared references
  - Belongs to an entity

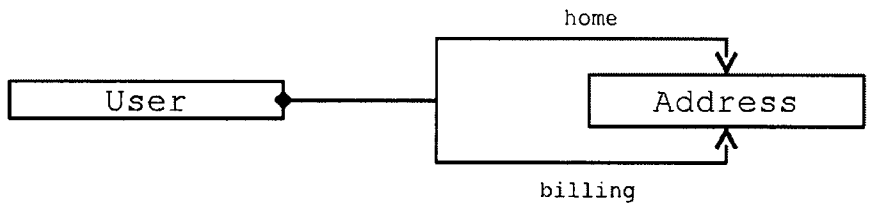
In Practice...



- Use stereotypes in your UML diagram for type demarcation.
- *Note: Bid has a dependent lifecycle but also supports shared references!*

Single Value Component

- Stored in the same table
  - Classes have a composition relationship:



Single Value Component

```
@Embedded
@AttributeOverrides( {
 @AttributeOverride(name   = "street",
  column = @Column(name="HOME_STREET", length = 255)
),
  @AttributeOverride(name   = "zipcode",
  column = @Column(name="HOME_ZIPCODE", length = 16)
),
 @AttributeOverride(name   = "city",
  column = @Column(name="HOME_CITY", length = 255) )
 })
private Address homeAddress;
```

- @AttributeOverride allows to modify the mapping

Single Value Component

- When inside an embeddable object, one of the properties may be defined as a pointer back to the owner element.

```
@Embeddable
public class Address {
. . .

@Parent
private User user;
```

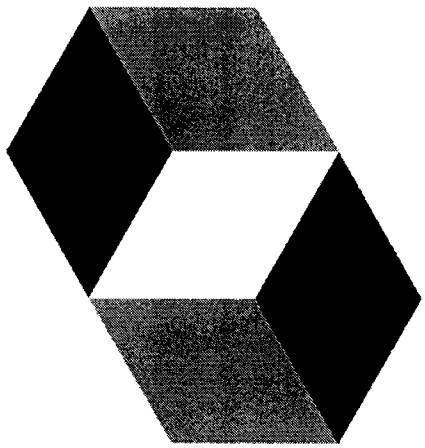
Collection of Component

```
@Entity
public class Sender {

@ElementCollection
@JoinTable(name = "sender_addresses",
joinColumns = @JoinColumn(name = "sender_fk"))
@AttributeOverrides(value =
{@AttributeOverride(name = "street", column =
@Column(name = "street_column"))})
private Set<Address> addresses = new
HashSet<Address>();
```

- @ElementCollection since JPA 2.1
- Hibernate offered @CollectionOfElements
  - Now deprecated
- Address is stored in a separate table
- @JoinTable configures the table

JBoss by Red Hat

Roadmap



- Entity and Value Types
- **Identity**
- Complex Mappings
- User-defined Types
- Lab 3

Identity: Objects vs. Databases - Revisited

- Java Object Identity
  - Test: `a==b`
- Database Identity
  - Test: `a.getId().equals(b.getId())`

- *When are both tests returning true?*
- *Why the need for two different notions?*
  - An application can concurrently access the same persistent state in two different Sessions
  - However, an instance of a persistent class is never shared between two Session instances
  - *Note: The ORM service guarantees a scope of object identity*

Identity Implementation: Object Equality Routine

- Objects outside the scope of identity – Essentially detached objects may be evaluated for equality.
- If Detached instances are tested:
  - Implement a custom (overridden) `equals()` & `hashCode()` - if two references are `equal()` they must have the same `hashCode()` value.
  - Once an object is within a hash-based collection (`HashMap`, `HashSet`), the hash code cannot change.
  - *Note: Recommended implementation of `equals()` is with comparison of business keys, not a comparison of identifiers (transient objects have no identifier value, it changes when they become persistent)*

Equality: Business Key Example

- Business Key is a property (or combination of properties) that is unique for each instance with the same database identity.
  - Business Key values rarely/never change.
  - Business Key is likely what people want in real-world use to identify a particular object (see Natural Key).
  - Business Key equality means that the `equals()` compares only properties that form the business key.

A Business Key Example

```
public class User {
    ...
 public boolean equals(Object other){
    if(this == other) return true;

    if(!(other instanceof User)) return false;
    final User that = (User)other;

    return
      this.username.equals(that.getUsername());
  }
  public int hashCode(){
    Return username.hashCode();
  }
}
```

- Good Candidates:
  - `Immutable/Unique` (database constraint) attribute – `Date/Time` value attributes (consider precision) – Database identifiers of *associated* objects: `bid.getItem().getId()` *never* changes.

Examining the Database Identifier Property

```
class Item {

 private Long id;

 public Long getId() {
   return this.d;
 }

 private void setId(Long id) {

   this.id=id;
 }

}
```

Identifier Property Example

- Database Identifier Property Guidelines:
  - Identifier can by any value object/type
  - Identifier property is a primary key value
  - Hibernate & JPA require an identifier property
    - At least a private field
  - Add public/private accessor method if desired

Id Property

```
// simple generated
@Entity
public class Sender {
@Id @GeneratedValue
private Long id;
...
}
// a more detailed configuration
@Id @GeneratedValue(strategy =
GenerationType.SEQUENCE, generator = "myGen")
@SequenceGenerator(name = "myGen",
allocationSize = 20)
private Long id;
```

- Common types are *Integer* or *Long*
- Can be generated
  - Various strategies are available
    - *GenerationType.SEQUENCE*
    - *GenerationType.TABLE*
    - *GenerationType.IDENTITY*
    - *GenerationType.AUTO*
- Can be assigned as well
  - Natural primary keys

Identifiers within Hibernate – Guidelines

- The property name should always be the same for consistency
  - Example: Id

- Should the Id remain public or private?
  - ID is a convenient "handle" to a particular instance
  - Lookup by Id is an especially efficient operation
  - A Hibernate Best Practice – expose a public getter/setter for Id
  - Identifier Generation Strategy is customizable – more on this later

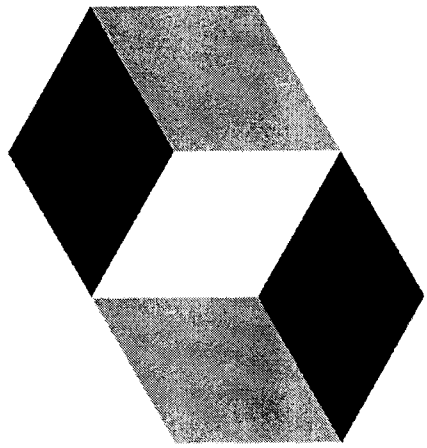
- *Note: In legacy databases where tables contain composite keys, user-defined tables with properties can be used in Hibernate – composite Ids.*

Hibernate Id Generators - Details

- A Generator names a Java class used to generate unique identifiers for instances of the persistent class.
  - All generators implement the interface `org.hibernate.id.IdentifierGenerator`.
  - 2 new generators address portability & optimization.
    - `org.hibernate.id.enhanced.SequenceStyleGenerator`
      - Replaces `Sequence`, better to port than `Native`
    - `org.hibernate.id.enhanced.TableGenerator`
      - Defines a table able to holding many different increment values simultaneously by using multiple distinctly keyed rows.

JBoss by Red Hat

Roadmap



- Entity and Value Types
- Identity
- **Further Mappings**
- User-defined Types
- Lab 3

@Formula

```
@Formula("(1+vat)*price")
double price;

@Formula("select sum(p.total) from
InvoicePosition p where p.id=id")
double total;
```

- A SQL snippet for a property
- Can even be a correlated subquery
  - If supported by your database

Mapping Entities: Override Annotated Fields

```
@Entity
class Item {
  ...

  @Basic(optional = false)
  @Column(name = "DESCRIPTION"
nullable = false)
  private String description;

  . . .
}
```

Annotation Override Example

Note: *Both annotations would result in a* `NOT NULL` *column if the schema is auto-exported via Hibernate toolset.*

- All non-static non-transient properties (field or method) of an Entity are considered persistent:
  - `@Transient` – Makes the property non-persistent
  - `@Basic` – Permits to declaration of the fetching strategy for a property
  - `@Columns` – defines column(s) used for a property mapping – it is used to override default values and is used at the property level for properties that are:
    - annotated with `@Basic`
    - annotated with `@Version`
    - annotated with `@Lob`
    - annotated with `@Temporal`
    - annotated with `@org.hibernate.annotations.CollectionOfElements` (for Hibernate only)
    - not annotated at all

Mapping Entities: Annotations

```
@Entity(access = AccessType.FIELD)
class Item implements Serializable
{

    @Id @GeneratedValue
    private Long id;

}
```

Annotation Example

Note: *Entities – Item, Category, User*

- Mapping Guidelines:
  - The position of the `@Id` defines whether the field or method access is the default .
- All annotations are expected on fields if field-access is enabled.
- Table & column names default to class & field/accessor (getters) names.
- *Note: Persistent fields/properties of an Entity are also called "attributes" of a class.*

Mapping Entities: Enabling Accessor Methods

```
@Entity
@Table(name = "ITEM")
class Item {
    private Long id;

 @Id @GeneratedValue
 @Column(name = "ITEM_ID")
 public Long getId() { …}

 private void setId(Long id) { ...}

}
```

Accessor Method Example

- Mapping Guidelines:
  - All properties are now accessed through accessors/mutators (getter/setters) methods.
  - All annotations are expected on accessor (getter) methods – not fields.

Mapping Entities: Custom Property Access

```
@Entity
class Item {
    ...
    @org.hibernate.annotations.
                    AccessType("field")
    private String description;
     . . .
}
```

Custom Property Access Example

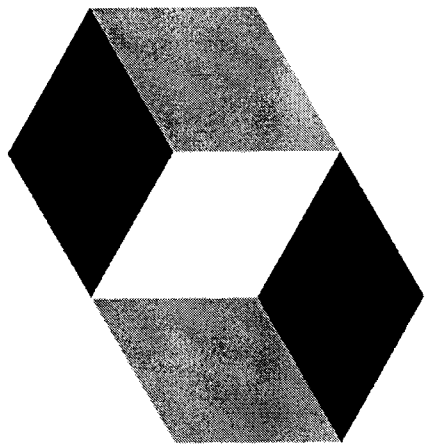
Note: *Access type is overridden for the annotated element. On classes, all the properties of the given class inherit the access type.*

For root entities, the access type is considered to be the default one for the whole hierarchy

- Hibernate supports overriding the access type via `@AccessType` annotation:
  - Custom access type strategy usage
  - Refine the access type at the class level or at the property level
  - access type can be overridden upon:
    - Entities
    - Superclasses
    - Embeddable objects
    - Properties/Fields

JBoss by Red Hat

Roadmap



- Entity and Value Types
- Identity
- Further Mappings
- **User-defined Types**
- Lab 3

Advanced Mappings

- Custom user-defined types and converters
  - Implementing international pricing and currency handling
  - Alternatively we write a custom UserType:

```
@Entity
@Table(name = "BID")
@org.hibernate.annotations.Entity(mutable = false)
public class Bid {
 . . .
    @org.hibernate.annotations.Type(type = "monetary_amount_usd")
    @org.hibernate.annotations.Columns( columns = {
        @Column( name="BID_AMOUNT", length = 2, updatable = false),
        @Column( name="BID_AMOUNT_CURRENCY", length = 3, updatable =
false)
        }
)
    private MonetaryAmount amount;
```

Annotations

  - Our UserType implementation knows how to store and load MonetaryAmount objects into and from two columns.

Advanced Mappings

- When to use custom types:
  - User-defined mapping types are powerful, we...
    - Can implement "unnatural" conversion, e.g. from `String` to `int`
    - Can handle validation
    - Might read and write data from/to an LDAP directory
    - Can read and write data to a different database
    - Can create typesafe enumerations (Typesafe Enum pattern)
  - Hibernate provides different interfaces you can combine.
    - `UserType`: The basic interface
    - `CompositeUserType`: Exposes properties to Hibernate queries
    - `ParameterizedType`: Metadata arguments for custom types
    - `EnhancedUserType`: For discriminator values and XML marshalling
    - `UserVersionType`: For version properties
    - `UserCollectionType`: For custom collection semantics

JB297 - Hibernate
Copyright © 2011 – Red Hat Inc.

Advanced Mappings

- Loading and storing money
  - First, the basic implementation of a `UserType`:

```
public class MonetaryAmountSimpleUserType implements UserType {

 public int[] sqlTypes() {
   return new int[]{ StandardBasicTypes.BIG_DECIMAL.sqlType(),
     StandardBasicTypes.STRING.sqlType() };
 }
 public Class<MonetaryAmount> returnedClass() { … }

 public Object nullSafeGet(ResultSet resultSet,String[] names,Object owner)throws
              HibernateException, SQLException {
   BigDecimal valueInUSD = resultSet.getBigDecimal(names[0]);
   if (resultSet.wasNull()) return null;
     Currency currency = Currency.getInstance(resultSet.getString( names[1] ) );
     return new MonetaryAmount(valueInUSD, currency);
   }

   public void nullSafeSet(PreparedStatement statement,Object value,int index)throws
            HibernateException, SQLException {
     if (value == null) {
       statement.setNull(index, StandardBasicTypes.BIG_DECIMAL.sqlType());
       statement.setNull(index+1, StandardBasicTypes.CURRENCY.sqlType());
     } else {
       MonetaryAmount monetaryAmount = (MonetaryAmount)value;
       statement.setBigDecimal(index, monetaryAmount.getValue());
       statement.setString(index+1, monetaryAmount.getCurrency().getCurrencyCode());
     }
   }
//  ********** More Method Details ****************** //
```

Advanced Mappings

- Using a composite type
  - A composite type exposes individual properties for Hibernate queries.

```
public class MonetaryAmountCompositeUserType
        implements CompositeUserType {
 public String[] getPropertyNames() {
   return new String[] { "value", "currency" };
 }

 public Type[] getPropertyTypes() {
   return new Type[] { Hibernate.BIG_DECIMAL,
                       Hibernate.CURRENCY };
 }
...
```

  - We can now use the component properties in HQL.

```
entityManager.createQuery("from Bid b where b.amount.value > :value")
        .setParameter("value", new BigDecimal(50))
```

Advanced Mappings

- Using a composite type
  - Definition of a custom type can be outside of a class mapping
  - Can be re-used in many class/property mappings

```
@org.hibernate.annotations.TypeDefs({
    @org.hibernate.annotations.TypeDef(
        name="monetary_amount_usd",
        typeClass = MonetaryAmountType.class,
        parameters = { @org.hibernate.annotations.Parameter(name="convertTo", value="USD") }
    ),
    @org.hibernate.annotations.TypeDef(
        name="monetary_amount_eur",
        typeClass = MonetaryAmountType.class,
        parameters = { @org.hibernate.annotations.Parameter(name="convertTo", value="EUR") }
    )
})

@org.hibernate.annotations.FilterDefs({
    @org.hibernate.annotations.FilterDef(
        name="limitItemsByUserRank",
        parameters = {
            @org.hibernate.annotations.ParamDef(
                name = "currentUserRank", type = "int"
            ) } ) })
package auction.persistence;
import auction.persistence.MonetaryAmountType;
```

JBoss by Red Hat

Lab - 3

Advanced Class & Property Mappings: Hibernate, User-Defined Types

- Details and instructions for the lab can be found in the lab guide.
- The instructor will answer any questions and will determine the stop time.

Summary

- In this lesson, you have learned about:
  - Difference between entity and value types
  - Advanced mappings
  - Hibernate identities
  - Custom types

Lab 3: Advanced Mappings

Requirements

- JB297-Hibernate Lab Archives
- Pre-Configured JBoss Developer Studio IDE
- Time: 30 Minutes

Goal

This lab demonstrates advanced component mappings and user types

Description

– In this lab you need to complete mappings and code. There are a number of tasks and each tasks can be validated running the Junit tests.

– The tasks are marked with TODO markers. Search each of the source files listed below for Tasks

Example Task:
```
/* TODO: A Task Title
*  Step 1: Code Something Here, As Instructed - Tags
*/
```

– You can find a detailed description below.
– Import JB297's **lab-3** project as an existing project, copied to the workspace
– The working/full source code is available in the lab-3/solution/ directory. View this directory for help, or backup the lab-3/src/ directory, rename the solution folder to src.

Project Instructions

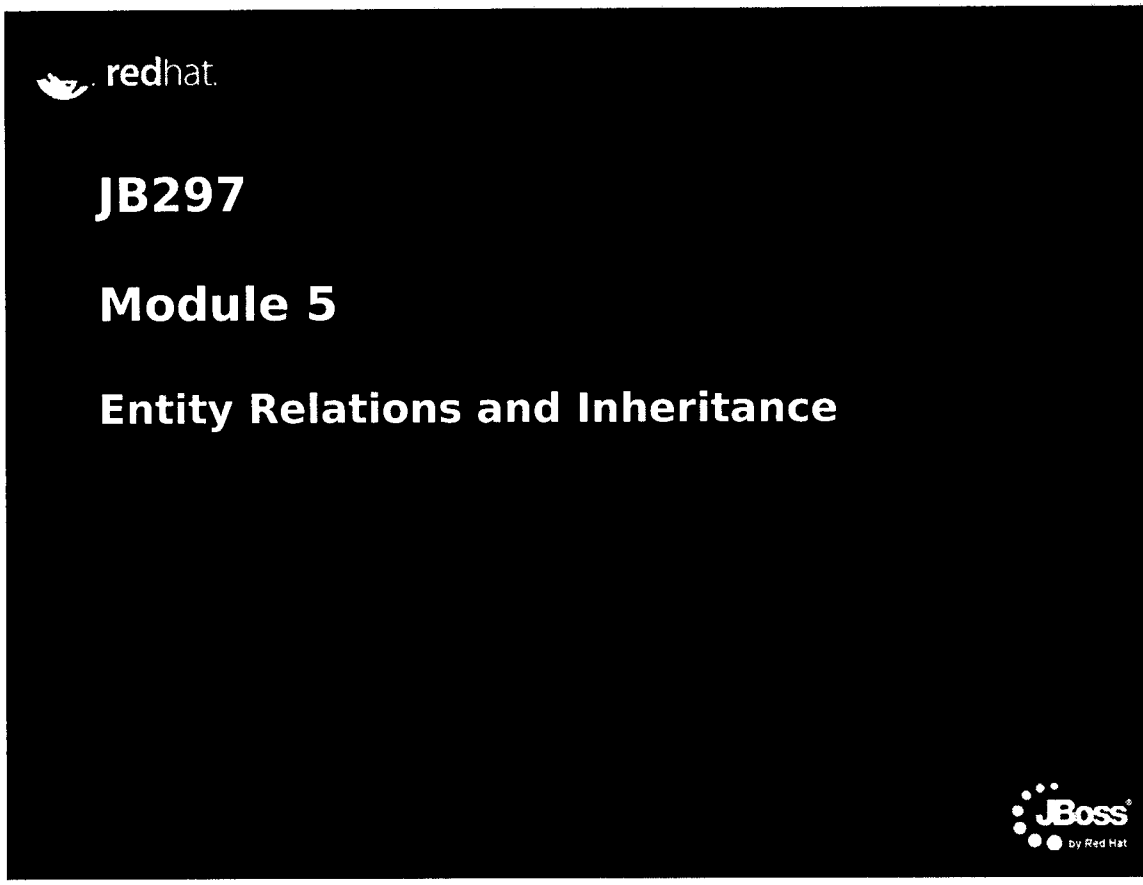
---

1. Restart your database, to get an empty database schema.

2. The user has two address components: home and billing address. By default the components are persisted in the user table using the column names as mapped in the embeddable *Address* class.
   i. Change the mapping of the two components with the @AttributeOverrides annotation. Add the prefix HOME_ to all columns of the home address and the prefix BILLING_ to all columns of the billing address.
   Hint: Examine the Address class to determine column information.
   ii. Run the test *persistAddress* in the class *MappingTest* to validate your changes. Don't forget to start the database.

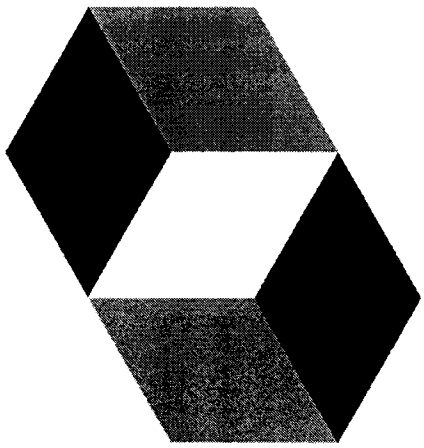
3. A bid has a monetary amount. It is currently mapped as a component. We want to change the mapping now to use a user type, instead.
   i. Complete the user type *MonetaryAmountSimpleUserType* in the *auction.persistence* package.. It is missing the methods which converts the database columns to the class and back.
   ii. Complete the mappings in *Bid* to make use of your user type. The type is already configured in the *package-info.java* file in *auction/persistence*.
   iii. Disable the *@Embeddable* annotation in the class *MonetaryAmount*.
   iv. Run the *persistSimpleUserType* test to validate your changes.

4. The query in *useCompositeUserType* fails. It tries to use individual properties of the user type. What kind of user type may help us for this requirement?
   i. Check if the class *MonetaryAmountUserType2* is the user type you need. If so, change the mapping in the class *Bid* to use that type.
   ii. Run the *useCompositeUserType* again to validate your changes.



JBoss by Red Hat

Roadmap



- **Entity Relations**
- Inheritance Strategies
- Lab 4

JBoss by Red Hat

Associations: Parent-Child Relationships

- A "parent/child" instance has a collection of references to *many* child instances
- A child instance has a single" reference to *one* parent instance
- The term - "parent/child" implies a transitive lifecycle; deleting one parent instance deletes all referenced child instances

Understanding the Association - Multiplicity

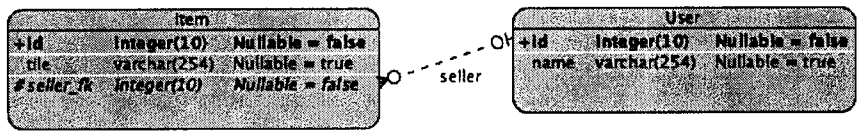
- Hibernate (JPA) does *not* manage bi-directional references.
  - Essentially no container managed associations.
- The association from Item to Bid is a different one than the association from Bid to Item.
- Mapping a *one-to-many* and *many-to-one* is supported.
  - Hibernate will need to be "told" that both associations map to the same foreign key column.
- Association Types
  - Uni-directional association

    vs.
  - Bi-directional association

Associations and Collections

```
@Entity
public class Item {
 @ManyToOne(fetch = FetchType.LAZY)
 @JoinColumn(name = "SELLER_FK",
    nullable = false,
    updatable = false)
 private User seller;

// Usage
```

- Many to One
  - Unidirectional
  - Adding a foreign key to the item table



Associations and Collections

```
@Entity
public class User {
    @OneToMany(mappedBy = "seller")
    private Collection<Item> itemsForSale
          = new ArrayList<Item>();
...
}
@Entity
public class Item {
  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "SELLER_ID",
    nullable = false, updatable = false)
  private User seller;
...
}
```

- Making it bidirectional
  - The "many" side should use *mappedBy*
  - Parent/child relationships (with cascading) are common
  - Why bidirectional ?
    - All associations could be *@ManyToOne* or *@OnetoOne*
    - Hibernate can't optimize much => subselect/batch collection fetching
    - Write a query when all items of a seller is needed.

Associations and Collections

- One-to-One association mapping
  - We can map a one-to-one using two different techniques:
    - A shared primary key, the problem is generation of values
    - A foreign key, we need a unique constraint
  - We can also map it using a link table, but this is exotic...

Associations and Collections

```
@GenericGenerator(name = "foreign_id",
   strategy = "foreign", parameters =
@Parameter(name = "property",   value =
"user") })

@Entity
public class Address {
 @Id
 @GeneratedValue(generator =
  "foreign_id")
 private Integer id;

 @OneToOne(optional=false)
 @PrimaryKeyJoinColumn
 private User user;
```

- One-to-one: Using a primary key association
  - Two rows have the same PK values
  - Id generator is a Hibernate extension
  - Can be bidirectional
    - Cascading helps us to make one of the two "more important".

Associations and Collections

```
@Entity
public class Address {
 @OneToOne
 @JoinColumn(name="user_id")
 private User user;
...
```

- One-to-one: Using a foreign key
  - We can use a (unique) foreign key in ADDRESS to the USERS table

Associations and Collections

- Many-to-Many entity associations
  - A many-to-many association between Category and Item
    - Many-to-many associations are rare,
    - We almost always need some more information on the "link", e.g. "who added this item to this category and when?"
    - But we'll first map a real many-to-many, we hide the join table in the Java domain model.

Associations and Collections

```
@Entity
public class Category {
 @ManyToMany
 @JoinTable(
   name = "CATEGORY_ITEM",
   joinColumns =
    @JoinColumn(name = "CATEGORY_ID"),
   inverseJoinColumns =
    @JoinColumn(name = "ITEM_ID")
 )

 @IndexColumn(name = "DISPLAY_POSITION")
 private List<Item> items =
   new ArrayList<Item>();

...
```

- Many-to-Many: A unidirectional association
  - A unidirectional many-to-many association is trivial
  - The association table CATEGORY_ITEM has two columns, CATEGORY_ID and ITEM_ID, both form the primary key.

Associations and Collections

```
@Entity
public class Item {

 @ManyToMany(mappedBy="items")
 private Set<Category> categories
    = new HashSet<Category>();
...


/* Usage, don't forget to set the relation on
   Both sides.
*/
item.getCategories().add(category);
category.getItems().add(item);
```

- Mapping it bidirectional
  - We have to deactivate one end using mappedBy
  - We are free to set either end to inverse.
  - Remember that indexed collections can't be inverse.
  - Be careful with the cascading semantics
    - Shared references!!
  - Always set the relation on both sides !!

Associations and Collections

```
public class CategorizedItem {
    private Category category;
    private Item item;
    private String username;
    private Date dateAdded;
    // don't forget equals() and hashCode()!
}
```

- Extra columns on the link table
  - A class that represents the join table with the extra information

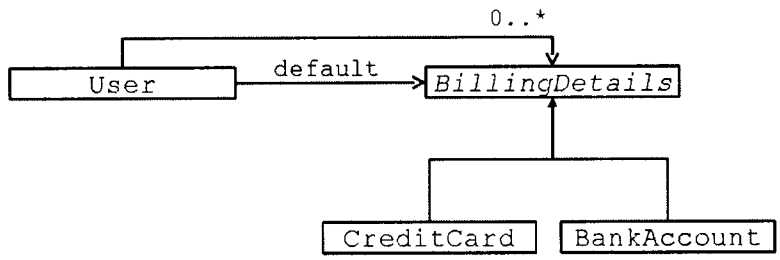
Associations and Collections

```
@Entity
public class Category{

  @ElementCollection
  @JoinColumn
  private Set<CategorizedItem> items =
          new HashSet<CategorizedItem>();
...
}

@Embeddable
public class CategorizedItem {
  @ManyToOne
  @JoinColumn
  private Item item;

  private String username;
...
}
```

- Using a composite element mapping
  - A composite element is perfect in this situation
  - Creating a many-to-many link is very easy, only instantiate a `CategorizedItem`. Dito for all other lifecycle changes.

Associations and Collections

- Polymorphic associations
  - A polymorphic association, User to a single BillingDetails




```
@OneToMany
@JoinColumn
private Set<BillingDetails> allBillingDetails = new
HashSet<BillingDetails>();
```

Associations: More Hibernate Features

- Hibernate-generated Foreign key constraints have a fairly unreadable name. Override the constraint name by use `@ForeignKey`

```
@Entity
@Table(name = "BID")
@org.hibernate.annotations.Entity(mutable = false)
public class Bid implements Serializable, Comparable<Object> {
 . . .

    @ManyToOne
    @JoinColumn(name = "ITEM_ID", nullable = false, updatable = false,
       insertable = false)
    @org.hibernate.annotations.ForeignKey(name="FK_ITEM_ID")
    private Item item;

    @ManyToOne
    @JoinColumn(name = "BIDDER_ID", nullable = false, updatable = false)
    @org.hibernate.annotations.ForeignKey(name="FK_BIDDER_ID")
    private User bidder;
```

Annotations

Associations: More Hibernate Features

- `@IndexColumn` – Hibernate Annotations supports true List and Array. This annotation describes the column that will hold the index.
- The index value may be declared in DB that represent the first element (aka as base index). The usual value is 0 or 1.

```
@Entity

public class Item  {
 . . .
    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JoinColumn(name = "ITEM_ID", nullable = false)
    @org.hibernate.annotations.IndexColumn(name = "BID_POSITION")
    @org.hibernate.annotations.BatchSize(size = 10)
    private List<Bid> bids = new ArrayList<Bid>();

}
```

Annotations

Associations: More Hibernate Features

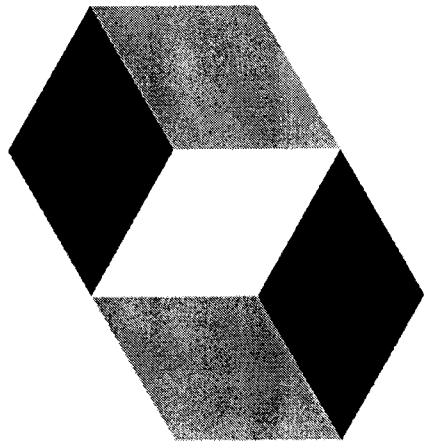
- `@OrderBy` – This annotation describes the field in a OneToMany collection that will be used for ordering
- Key attribute: clause describes the SQL clause

```
@Entity
@Table
@org.hibernate.annotations.Entity(mutable = false)
public class Category implements Serializable, Comparable<Object> {
 . . .

    @OneToMany(mappedBy="parentCategory",
               cascade={CascadeType.PERSIST, CascadeType.MERGE})
    @org.hibernate.annotations.OrderBy(clause="CATEGORY_NAME asc")
    private List<Category> childCategories = new ArrayList<Category>();
```

Annotations

JBoss by Red Hat

Roadmap



- Entity Relations
- **Inheritance Strategies**
- Lab 4

Hibernate: Inheritance Overview

- Inheritance
  - Normal occurrence with Java to define an inheritance relationship between classes.
  - Several well-established strategies may be employed according to relevance.
  - Inheritance (and Polymorphism) defining features of O-O Domain Models.
  - Hibernate/JPA allows users to determine how to persist classes in the inheritance tree.
  - Also inside each class (Entity) "how" information is persisted is determined as well.

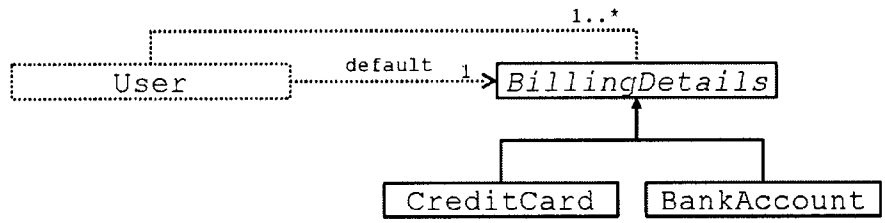
Inheritance: Understanding Entities

- Entities *may* inherit from another entity.
- Abstract and Concrete classes may be defined entities.
- Some special cases:
  - Abstract Entity Classes - Abstract class mapped as an "entity", and commonly used in the Table per Class strategy. Uses all of the common annotations.
  - @MappedSuperclass Classes – Entity *may* inherit from a (concrete/abstract) non-entity superclass that, by design, defines common entity state and mapping details - it's not queryable and is not passed as a parameter to `EntityManager` or `Query` functions. This is Implicit Polymorphism, discussed in more detail later in the module.
  - Non-Entity Class (within Inheritance Chain) – The (concrete/abstract) non-Entity class offers behavior mostly, is not persistent, is *unmanaged* by Hibernate/JPA, no associated annotations.

Basic Inheritance Mapping Strategies

- Mismatch: Java value-types provides *"is a"* and *"has a"* Entity relationships (RDBMS understands only a "has a" Entity).
- Table per Class Hierarchy:
  - All classes mapped to a single table
  - The table contains a *Discriminator Column*
- Table per Subclass:
  - Root of class hierarchy is represented by a single table
  - Subclasses represented by tables containing fields specific to the subclass (non-inherited properties)
  - May also use a discriminator column
- Table per Concrete Class:
  - Only subclass is mapped to separate table
  - *All* class properties (inherited/non-inherited) are mapped to columns of the corresponding table
  - Superclass may be implied

Class Hierarchy for Inheritance Examples



Mapping Strategy: Table per Class Hierarchy

Features

- Benefits:
  - Winner in Performance & Simplicity
  - Best way to represent polymorphism
    - polymorphic/non-polymorphic queries perform well
  - Good support for polymorphic relationships between "Entities"
  - Good support for Queries ranging over the class hierarchy
  - Schema evolution is clear & concise
  - Can perform Ad-hoc reporting w/o complex joins/unions
- Pain Points:
  - Requires columns corresponding to subclass state be nullable
  - Normalization Challenges
    - Violates Third Normal Forms

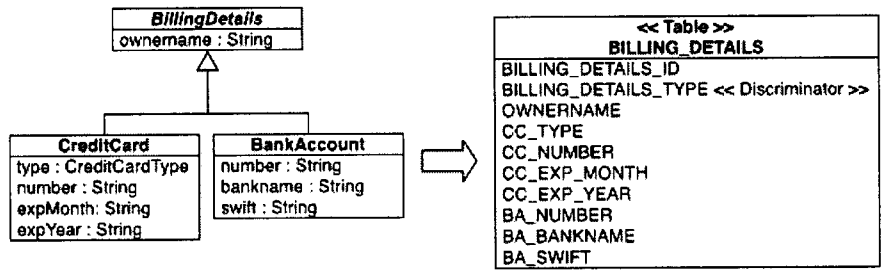
Details

- All of the data representing the properties of every class in the Hierarchy are stored in one table:
  - One table for three Java classes
  - Requires that columns representing fields in subclasses allow null
  - Uses ***discriminator*** column to determine the type of object represented a row
    - Each subclass must declare the discriminator value for the type

Third Normal Form essentially states that null values in a row aren't adequately normalized, since every non-key attribute in a row should provide information concerning the key.

Strategy Example: Table per Class Hierarchy

- Queries are simple SELECT statements
- Superclass implements shared properties
- Subclasses implement detailed properties
- InheritanceType.SINGLE_TABLE used in annotations



Annotations used:

@Inheritance in superclass (can leave off the InheritanceType because SINGLE_TABLE is the default).

@DiscriminatorColumn in superclass
@DiscriminatorValue on subclasses

Subclasses have no @Id field (inherited from superclass)

All the usual @Entity and @Column annotations will still be used to define the fields.

Mapping Strategy: Table per Class Hierarchy SQL Queries

Annotation:

```
@Entity
@Inheritance
@DiscriminatorColumn(name="BILLING_TYPE",
                     discriminatorType = DiscriminatorType.STRING)
public class BillingDetails implements Serializable, Comparable { . . . }

@Entity
public class CreditCard extends BillingDetails{ . . }
```

Resulting SQL Queries:

```
Retrieving a one-to-many collection:
 - HQL: from User u join u.billingDetails bd
 - SQL: select … from USER u
          inner join BILLING_DETAILS bd
          on u.USER_ID = bd.OWNER_ID

Retrieving objects of a concrete class:
 - HQL: from BankAccount
 - SQL: select ba.BA_ACCOUNT, ba.BA_SWIFT… from BILLING_DETAILS ba
```

JBoss by Red Hat

Mapping Strategy: Table per Subclass

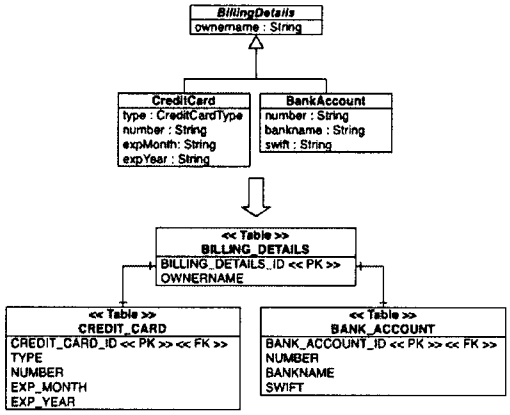
Features

- Benefits:
  - Offers fully-normalized SQL schema
  - Schema evolution & integrity constraints are clear & concise
  - Polymorphic associations work well
  - Strategy very simple to implement
- Painpoints:
  - Polymorphic queries *may* run slowly
  - Difficult to hand-code strategy
    - Can encounter complex *ad-hoc* queries
    - Challenging if Hibernate code will be mixed with hand-written SQL
  - Performance *may* be unacceptable for complex class hierarchies overall

Details

- Essentially represents - *"is a"* and *"has a"* relationships as foreign key associations - Each subclass in the hierarchy gets its own table
  - Parent class fields are in one table
  - Primary key in subclass tables refer back to the parent class table
  - Use joined-subclass configuration
  - Three tables for three classes

Strategy Example: Table per Subclass



- Normalized Model
- Supports polymorphic associations even to particular subclasses.
- Performance of OUTER JOIN can be an issue.
- Ad hoc Reporting may be more complex.

InheritanceType.JOINED is used with the @Inheritance in the superclass

@SecondaryTable must be used to define the relation in the subclass.

Essentially a one-to-one mapping exists between rows in the BILLING_DETAILS table and a row in the CREDIT_CARD table, and between the BILLING_DETAILS and BANK_ACCOUNT tables.

SQL queries will result in join statments.

Strategy Example: Table per Subclass

Java Annotations:

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class BillingDetails implements Serializable, Comparable { . . . }

@Entity
public class CreditCard extends BillingDetails{ . . }
```

Resulting SQL Queries

```
Retrieving one-to-many collection:
 - HQL: from User u join u.billingDetails bd
 - SQL: select … from USER u
         inner join BILLING_DETAILS bd
           on u.USER_ID = bd.OWNER_ID
          left outer join CREDIT_CARD cc
            on cc.CREDIT_CARD_ID = bd.BILLING_DETAILS_ID

Retrieving objects of a concrete class:
 - HQL: from CreditCard
 - SQL: select … from CREDIT_CARD cc
          inner join BILLING_DETAILS bd
            on cc.CREDIT_CARD_ID = bd.BILLING_DETAILS_ID
```

Mapping Strategy: Table per Concrete Class

Features

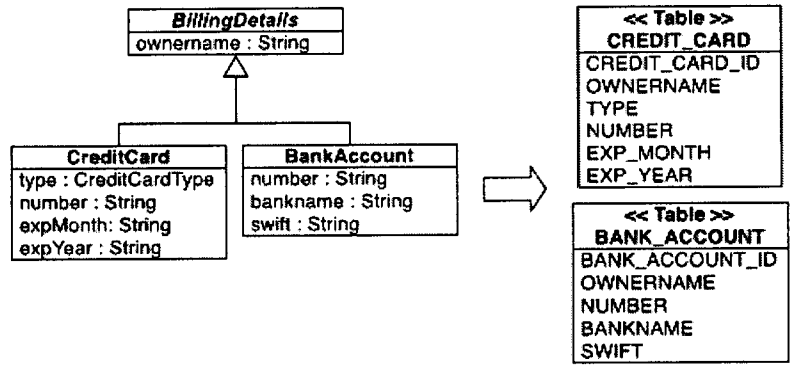
- Benefits:
  - Ideal strategy for *only* for top-level hierarchy classes – no notion of Inheritance at the database level
    - Superclass modifications are not likely
    - Polymorphism is not required
  - No special steps needed by Hibernate to achieve polymorphic behavior
  - No detailed inheritance mapping
  - Default runtime polymorphic (query & loading) behavior
- Pain Points:
  - Schema evolution is complex
  - No support for full polymorphic queries in JPA – only *mapped* Entities may be part of JPA query

Details

- Each concrete Entity class in the hierarchy receives its own table
- Two tables for three classes
- Two Approaches
  - Union Subclass
  - Implicit polymorphism
- The parent class fields are *duplicated* in each of the tables.
- Superclass properties are ignored / not-persistent by default:
  - Superclass must annotate if properties are to be persistent
  - Annotated super-type may *not* be an interface in JPA – However, this *is* permitted in Hibernate
  - Superclass field names must be the same in both subclass tables

Strategy Example: Table per Concrete Class

- Polymorphic queries result in a UNION (instead of several selects) for InheritanceType.TABLE_PER_CLASS.
- Many-valued associations back to superclass result in a UNION (but must be bi-directional).
- Single-valued associations back to a superclass can be problematic – no foreign key constraint can be created.

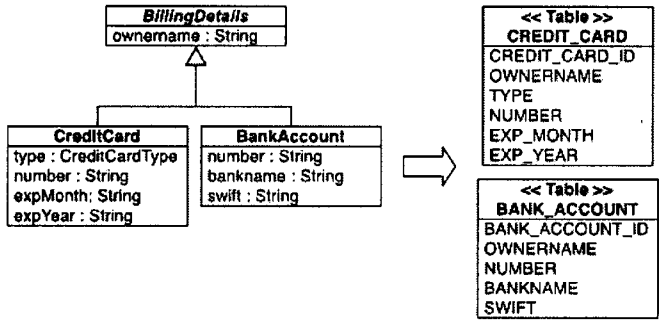


This is the Union Subclass example.

Superclass annotated with @Inheritance(strategy=InheritanceType.TABLE_PER_CLASS), but the superclass does not map to a database table. Fields in the superclass should be the same name in all subclass tables.

Strategy Example: Implicit Polymorphism

- Is a form of Table per Concrete Class
- Each concrete Entity class in the hierarchy receives its own table
- Superclass is, again, not mapped to a table
  - Annotate with superclass `@MappedSuperclass`
  - Each of the subclasses is mapped as a normal class, including Id field
  - Map unique properties in the subclasses
  - Map common properties in the superclass



Mixing Strategies: Table per Class and Subclass

- Map the Parent class as normal for Table per Class
  - Specify parent class properties
  - Map a discriminator column
  - `@Inheritance(strategy = InheritanceType.SINGLE_TABLE)`
- Map each of the subclasses
  - For those fitting Table per Class, `@DiscriminatorValue` is all that is needed
  - For those fitting Table per Subclass, combine `@DiscriminatorValue` with `@SecondaryTable`

Another way to mix strategies is to use implicit polymorphism along with table per subclass, but this is a complicated topic. Refer to the Hibernate Documentation, section 9.1.7, for more details.

Lab - 4

Inheritance Strategies and Entity Relations

- Details and instructions for the lab can be found in the lab guide.
- The instructor will answer any questions and will determine the stop time.

JBoss by Red Hat

Summary

- In this lesson, you learned about:
  - What Inheritance is and how it poses a challenge to ORM implementations.
  - Strategies for dealing the inheritance of entities.
  - What associations are and how to handle the associations of Entities

Lab 4: Entity Relations and Inheritance

Requirements

- JB297-Hibernate Lab Archives
- Pre-Configured JBoss Developer Studio IDE
- Time: 30 Minutes

Goal

This lab demonstrates the use of entity relations and inheritance

Description

- In this lab you need to complete mappings and code. There are a number of tasks and each tasks can be validated running the Junit tests.

- The tasks are marked with TODO markers. Search each of the source files listed below for Tasks

  Example Task:
  /* TODO: A Task Title
  * Step 1: Code Something Here, As Instructed - Tags
  */

- You can find a detailed description below.
- Import JB297's **lab-4** project as an existing project, copied to the workspace
- The working/full source code is available in the lab-4/solution/ directory. View this directory for help, or backup the lab-4/src/ directory, rename the solution folder to src.

Project Instructions

1. An *item* is sold by a *user*. In our model this is a bi-directional one-to-many relation.
   i. Add the missing annotations to the attribute *seller* of the class *Item* and the attribute *itemsForSale* of the class *User.*
   ii. Run the test *itemSellerTest* in the class *MappingTest* to validate your changes. Check the data in your database as well.

2. In this task, we try to save an item and a seller in the database.
   i. Run the test *createItemSellerTest* in the class *MappingTest.* Is the relation stored in the database?
   Hint: check with the GUI (runClientGui script in the hsqldb directory).
   ii. If the relation is not stored, repair the code in the method *createItemSellerTest.*

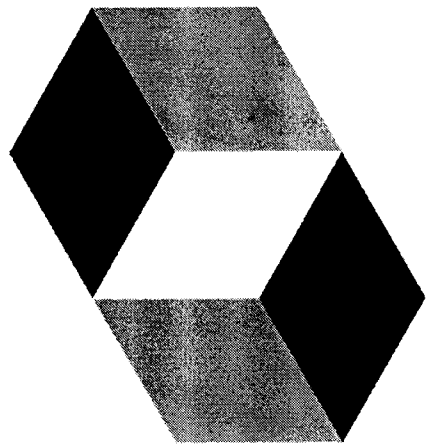
3. A category can have multiple items and an item might be in multiple categories. This is a bi-directional many to many relation.
   i. Add the missing annotations to the class *Item* and the class *Category.*
   ii. Run the test *addItemToCategoryTest* in the class *MappingTest.*

4. In this task, we try to delete a saved item.
   i. Run the test *deleteItemTest* in the class *MappingTest.* It fails with a constraint violation.
   ii. When creating bi-directional associations, we need to set the associations on both sides. The opposite is true if we delete an item which is part of an association. In that case you need to separate the association.
   iii. Repair the code in the method *deleteItemTest* to properly delete the item.

5. *BillingDetails* is the parent class of a inheritance structure. There are two types of billing details: credit card and bank account.
   i. Map the inheritance structure using the single table approach.
   ii. Run the test *inheritanceTest* in the class *MappingTest* to validate that your mappings are working.
   Have a look at the resulting SQL queries.
   iii. Change the inheritance strategy to table per class.
   iv. Run the test again and compare the resulting SQL queries to the single table strategy.

JBoss by Red Hat

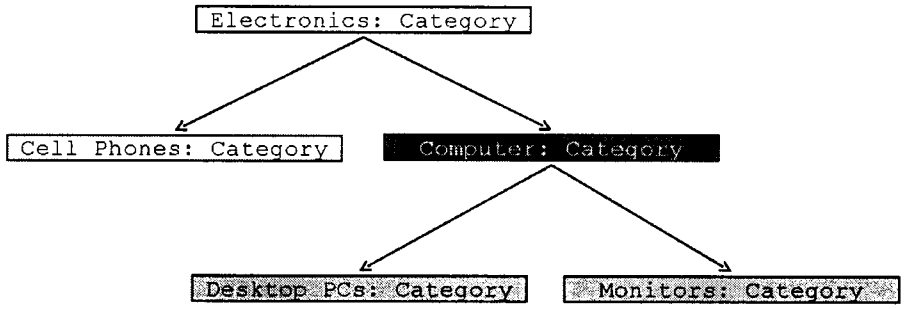
Roadmap



- **Cascading Persistent State**
- Object State and Persistence
- Session and Transactions

Cascading Persistent State

- Parent/ Child relationship suggest less code needed to manage relationships
  - Things are managed automatically



Transient Persistent Persistent by Reachability

Hibernate Cascade Options - Annotations

- Java Persistence Operations for Cascade:
  - MERGE
  - PERSIST
  - REMOVE
  - REFRESH
  - DETACH
- Corresponds to EntityManager API

- Hibernate Operations for Cascade:
  - MERGE
  - PERSIST
  - REMOVE
  - REFRESH
  - DELETE
  - SAVE_UPDATE
  - REPLICATE
  - DELETE_ORPHAN
  - LOCK
  - EVICT
  - (See documentation for CascadeType class values)
- Corresponds to Hibernate API

Cascading State Changes

- Instead of working with only single objects, Hibernate can cascade
  - State changes to associated entity instances.
  - Cascading options can be declared on a per-association basis
  - Recommendation
    - Use JPA Cascade Type if you use the EntityManager
    - Use Hibernate CascadeType if you use the Hibernate Session

```
@OneToMany(cascade={
  javax.persistence.CascadeType.PERSIST,
  javax.persistence.CascadeType.MERGE})
@org.hibernate.annotations.Cascade(value = {
 CascadeType.SAVE_UPDATE,
 CascadeType.PERSIST,
 CascadeType.MERGE})
private Set<Item> items = new HashSet<Item>();
```

Annotations

Enabling Cascading Deletion

- Deleting an `Item` instance also deletes all referenced `Bid` instances.
- Enabled with cascade option for removal.

```
@OneToMany( cascade = {CascadeType.REMOVE} )
public Collection<Bids> getBids()
```

Annotations

- When to use cascade 'all' or 'remove'?
  - Think about the object oriented terms: aggregation versus composition.

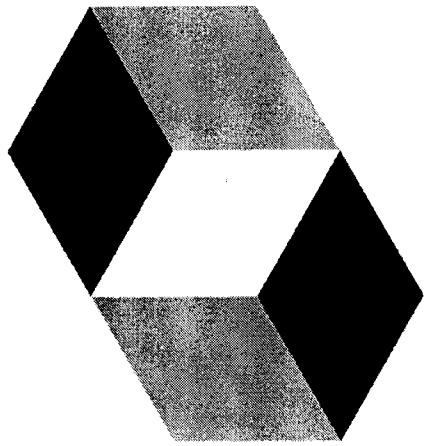
Associations: Understanding Orphan Deletion

```
/*
 Since JPA 2.0
*/
@OneToMany(orphanRemoval = true)
@JoinColumn(name = "item_fk")
private List<Bid> bids = new ArrayList<Bid>();

/*
With Hibernate or JPA < 2.0
*/
@OneToMany
@JoinColumn(name = "item_fk")
@Cascade(org.hibernate.annotations.CascadeType.
DELETE_ORPHAN)
private List<Bid> bids = new ArrayList<Bid>();
```

- Removing the reference to a `Bid` instance will delete the bid
- Rationale:
  - "...when I remove an element from this collection, assume it is the only & last reference to this particular Entity instance!"
- Only for One-to-Many Associations

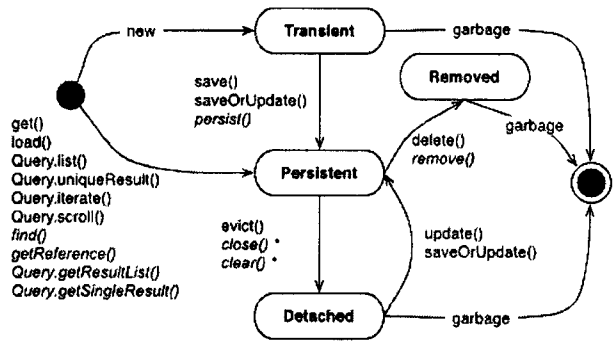
JBoss by Red Hat

Roadmap



- Cascading Persistent State
- **Object State and Persistence**
- Session and Transactions

Understanding Object State Management



* Hibernate & JPA, affects all instances in the persistence context

- The Persistence Lifecycle
  - With JDBC & SQL programming you ask: "What statements are executed; and how/what do they look like?'
  - With full ORM engines, you must ask: "What is the state of my objects; and how do I change their state?"
  - *Note: The state of an object can be manipulated explicitly (calling Persistence Manager APIs) or implicitly (composition - cascading)*

Object State Management - Concepts

- The Persistence Context – Hibernate Session
  - A ***Cache*** *of* objects managed by the ORM service
    - A True First-level Cache
    - Begins when the *Session (EntityManager)* is opened
    - Ends when the *Session (EntityManager)* is closed
    - Guarantees Scope of Object Identity
    - Extends the Persistence Context for a Conversation
    - Unit of Work - Atomic group of operations that *trigger* state changes of objects
  - Automatic Dirty Checking
  - Flush – Execution of SQL DML (Insert, Update, Delete) to ***Synchronize*** persistence context (State of the cache) with the database

State Management: Transient-to-Persistent

- Transient objects
  - Transient instances
    - Instances of a persistent class instantiated with the new operator
    - Transient, they have no persistent state
    - Garbage collected if de-referenced by the application
    - Have no database identity
  - Transient instances may be made persistent by
    - calling `save(o)` or `persist(o)`
    - creating a reference from another instance that is already persistent

State Management: Retrieve Persistent Objects

- Persistent objects
  - Persistent instances
    - include any instance retrieved with a query, lookup by identifier or navigation
    - are managed, changes are automatically flushed to the database
    - are transactional, changes can be rolled back in the database only
    - have database identity
  - Persistent instances may be
    - made transient with remove `(o)`
    - made transient by de-referencing from other persistent instances with "orphan delete" enabled (later)
    - made detached with `evict(o)`, `clear()`, or `close()`
    - refreshed (loaded from the database) with `refresh(o)`

State Management: Detached State

- Detached instances
  - Are instances with identity that are not associated with any open persistence context
  - Are no longer managed by Hibernate
  - Represent database state, that is potentially stale
- State of a detached instance can be `merge()`ed into an already persistent instance.
- Hibernate extensions
  - Detached instances can be made persistent by
  - `lock(o)`, `update(o)`, `saveOrUpdate(o)`, `replicate(o)`
  - Creating a reference from another instance that is already persistent

JBoss by Red Hat

Manage the Persistence Context

- `evict(o)`
  - Manually detaches an instance, essentially evicting it from the cache.
- `clear()`
  - Detaches all persistent instances from the context.
- Hibernate Session supports in addition
  - `setReadOnly(o,true)`
    - Disables automatic dirty checking during flush.
  - `setReadOnly(o,false)`
    - Enables automatic dirty checking during flush.

Synching the Persistence Context

- Default – `FlushMode.AUTO` – Enables synchronization:
  - When a Hibernate Transaction is committed
  - Before and explicit HQL, Criteria, or SQL query is executed
  - When the application calls `flush()` explicitly – (see Conversations)
  - Possibly other times – only order of SQL operations is guaranteed

- Advanced Use Cases:
  - `FlushMode.COMMIT` – Flush upon Transaction commit
  - `FlushMode.MANUAL` – Flush only when explicitly invoked via `flush()`

JBoss by Red Hat

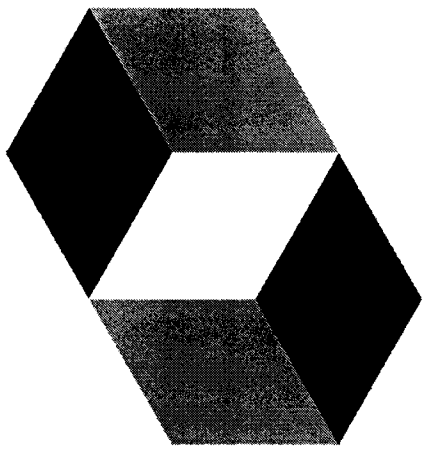
Automatic Dirty Checking

- The Persistence Context allows ORM engines to:
  - Automatically check all instances in persistent state for modifications.
  - Generate SQL statements that synchronize the cached changes with the DBMS.
  - Avoid conflicting representations, at most a single object in context for any DBMS row.

- Transactional Write-behind:
  - Execution of SQL DML occurs as late as possible.
  - Efficient use of JDBC Batching (more later) by grouping operations.
  - Keeping lock times in the RDBMS short – by executing DML towards the end of the transaction.

JBoss by Red Hat

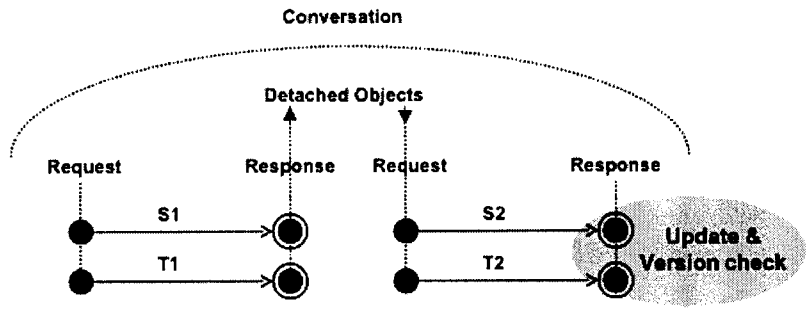
Roadmap



- Cascading Persistent State
- Object State and Persistence
- **Session and Transactions**

Use Cases: Detach/Reattach - Conversations

- Detached objects allow implementation of *Conversations*



- *Note: Objects are held (during use think time) & modified between requests – then merged or reattached to make changes permanent*

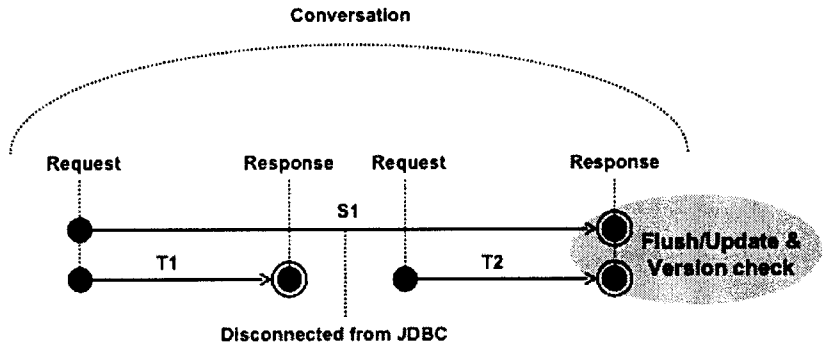
Detached Objects and Automatic Versioning

- Hibernate uses a smart algorithm (during flush) to detect transient/detached instances
  - Always consider using `saveOrUpdate()`
    - General purpose method that either saves a transient instance by generating a new identifier or updates/reattaches the detached instances associated with its current identifier vs. `save()` or `update()`.

JB297 - Hibernate
Copyright © 2011 – Red Hat Inc.

Extending the Persistence Context & Versioning

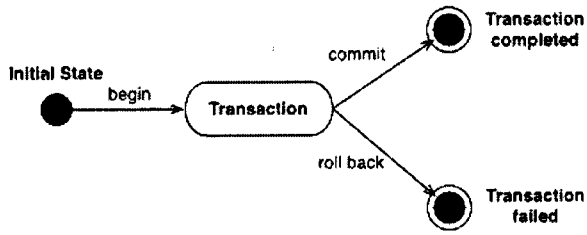
- Conversation using extended Sessions
  - FlushMode is set to manual - `FlushMode.MANUAL`
  - Session is stored and only flushed at the end
  - Session can be disconnected from a (committed) JDBC connection and reconnected to a new connection later on



Hibernate Transactions: The Essentials

- A Unit of Work – Application functionality require that several different tasks be executed simultaneously
  - Example: Three distinct tasks are performed upon auction completion:
    - Mark the winning (highest) bid
    - Charge the seller auction fees/costs
    - Notify the seller & successful bidder (winner)

ACID Criteria – Transaction Guarantee



- ***Atomicity*** - If one step in the unit-of-work fails, the whole unit must fail.
- ***Consistency*** - Data is left in a consistent state (database constraints).
- ***Correctness*** - Data is left in a *correct state* – from business perspective (never charge the customer twice).
- ***Isolation*** - A transaction is not visible to other concurrent transactions.
- ***Durability*** - Changes made are not lost even if the system fails afterwards.

Setting Transaction Boundaries

- Programatically begin, commit, or rollback transactions in code with:
  - Hibernate's Transaction API (JavaSE and JavaEE compatible)
  - JPA's EntityTransaction API (JavaSE compatible)
  - Standard JTA UserTransaction API (JavaEE Compatible)
  - Plain JDBC API (JavaSE compatible, not recommended)
- Declaratively create a Transaction assembly with:
  - Standard transactional EJB components (JavaEE compatible)
  - Custom transaction interceptors (AOP compatible)
- *Note: Programatic transaction demarcation requires extra coding*
  - *Declarative demarcation requires extra runtime infrastructure – see next section*

Transaction Demarcation – A Unit of Work

```
final EntityManager em =
emf.createEntityManager();
try {
 em.getTransaction().begin();
 em.persist(new Message("Hello world",
  new Sender("Holger")));
 em.getTransaction().commit();
} catch (RuntimeException
relevantException) {
 if (em.getTransaction().isActive()) {
  try {
   em.getTransaction().rollback();
  } catch (PersistenceException
rollBackException) {
   logger.warn("Rollback of open
transaction failed",
     RollBackException);
//less noisy without second stacktrace
//logger.warn("Rollback of open
transaction failed");
  }
  throw relevantException;
 }
} finally {
  em.close();
}
```

Unit of Work

- We always wrap every read or write data access inside a Transaction
  - Do not forget to close the `entity manager` – more on this later.

Automatic Session Binding & Propagation

```
public class ItemDao {
  private SessionFactory sessionFactory;

  public void setSessionFactory(
    SessionFactory sessionFactory) {
    this.sessionFactory = sessionFactory;
  }

  public void save(Item item){
    final Session session =
        sessionFactory.getCurrentSession();
    session.save(item);
  }
}
```

Unit of Work

- Hibernate specific
  - The "current" `Session` in bound to the Java thread & automatically closed upon Hibernate commit/rollback.
  - Enable within Hibernate configuration – `hibernate.cfg.xml` with `hibernate.current_session_context_class="thread"`.
  - Can be achieved with any AOP Framework as well.
    - See our Weld integration

Transaction Exception Handling

- Hibernate throws typed-exceptions – all subtypes of `RuntimeException`, which helps to identify errors:
  - Generic `HibernateException`
  - Subtypes of `JDBCException`, invoke `getSQL()`, `getSQLException()`, `getErrorCode()`
    - To return more information, see Best Practices module.
  - Other `RuntimeException` occurrences

- Rules/Guidelines for Hibernate Exception Handling:
  - Never use/reuse a `Session` again after an Exception has occurred.
  - Rollback the database or system transaction (JTA).
  - Do NOT use typed exceptions for data input validation.
    - See Validation Beans.

JB297 - Hibernate Copyright © 2011 – Red Hat Inc.

System Transactions in JavaEE

- A `TransactionManager` can manage multiple resources within the same transaction (e.g. databases & messaging systems).
- A `ResourceManager` connects & enlists resources with a `TransactionManager`.
- Programatic system transaction demarcation with CMT & EJB components.

Hibernate Transaction Configuration for JavaEE

- Name of managed datasource within JNDI – no Hibernate connection pool.
- *hibernate.transaction.factory_class* for JTA or CMT.
- *hibernate.transaction.manager_lookup_class* for your JTA provider/application server.

JB297 - Hibernate
Copyright © 2011 – Red Hat Inc.

Programatic Transaction Boundaries in JavaEE

- *A JTA service:*
  - *Can be installed separately or*
  - *Is available in all JavaEE application servers*

```
UserTransaction utx = (UserTransaction) new InitialContext()
                      .lookup("java:comp/UserTransaction");
EntityManager em = null;
  try {
      utx.begin();
      em = get().createEntityManager();
      concludeAuction(session);
      em.flush();
      utx.commit();
    } catch (RuntimeException ex){
       try{
           utx.rollback();
          }catch (RuntimeException rbEx){
           log.error("Couldn't Rollback Transaction", rbEx);
           }
       throw ex;
    } finally {
       em.close();
    }
```

UserTransaction Boundary

Declarative Transaction Assembly via EJBs

- Transaction and Persistence contexts are propagated (e.g. EJB X calls EJB Z)
- Transaction Scope for "current" Session
- Any RuntimeException triggers a rollback
- Automatically enabled if you configure Hibernate for CMT and EJBs

```
@Stateless
public class ManageAuctionBean implements ManageAuction{
  @PersistenceContext
  Private EntityManager em;

  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public void endAuction( Item item) {
      /*****************************/
      /* Reattach the Item instance */
      /* ***************************/
      em.merge(item);

    /* Charge Buyer/Winner     */
    /* Notify Winner & Seller */
    }
}
```

Declarative EJB Assembly

JBoss by Red Hat

Lab - 5

Cascading and Object State

- Details and instructions for the lab can be found in the lab guide.
- The instructor will answer any questions and will determine the stop time.

JBoss by Red Hat

Summary

- In this lesson, you learned about:
  - What the Cascading Persistent State is.
  - How to handle transitive persistence, cascading deletion and orphan deletion in associations.
  - Object state management and how to retrieve and maintain persistent objects.
  - How transactions are handled in Hibernate and how they can be configured.

JB297 - Hibernate
Copyright © 2011 – Red Hat Inc.

Lab 5: Cascading and Object State

Requirements

- JB297-Hibernate Lab Archives
- Pre-Configured JBoss Developer Studio IDE
- Time: 30 Minutes

Goal

This lab demonstrates the use of entity relations and inheritance

Description

- In this lab you need to complete mappings and code. There are a number of tasks and each tasks can be validated running the Junit tests.

- The tasks are marked with TODO markers. Search each of the source files listed below for Tasks

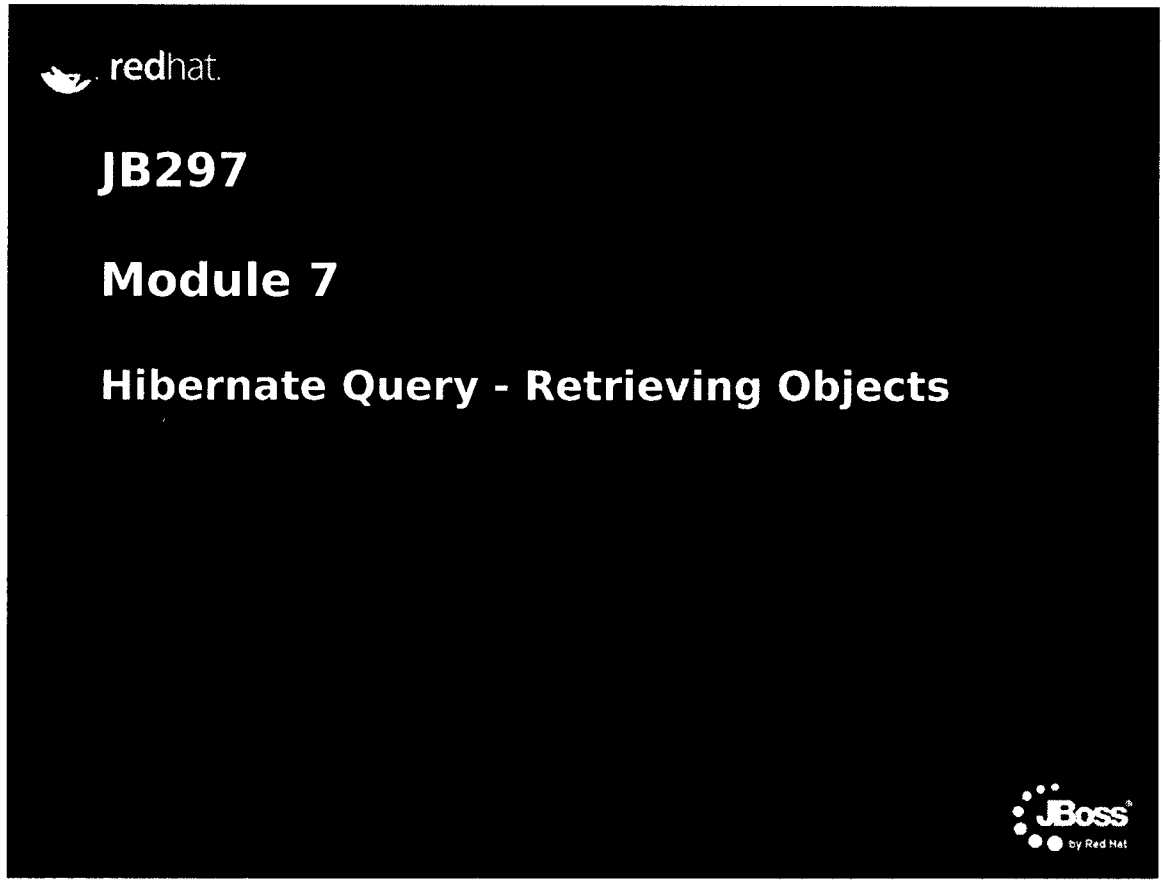
  Example Task:
  ```
  /* TODO: A Task Title
  *  Step 1: Code Something Here, As Instructed - Tags
  */
  ```

- You can find a detailed description below.
- Import JB297's **lab-5** project as an existing project, copied to the workspace
- The working/full source code is available in the lab-5/solution/ directory. View this directory for help, or backup the lab-5/src/ directory, rename the solution folder to src.

Project Instructions

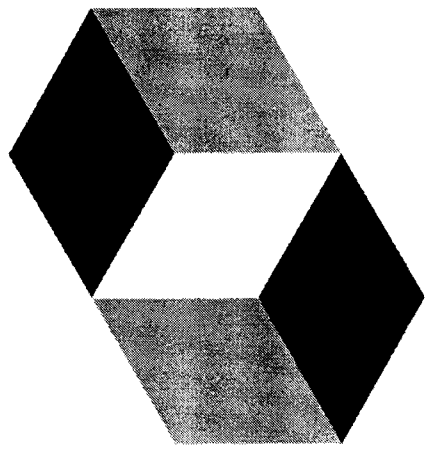
1. A *category* contains a *subcategory* which itself contains *items*. In this lab we try to persist an object tree as efficiently as possible.
   i. Have a look into the test *persistData* in the class *MappingTest*.
   There are currently 4 calls to persist. Try to use cascading to remove at least two of the calls.
   ii. Verify the inserts using SQL logging of Hibernate. Set the priority for the package *org.hibernate.SQL* in *log4j.xml* to debug.
   iii. Run the test again to see if the data is still persisted after your changes.

2. An *item* may have *comments*. If a *comment* is removed from an *item*, the comment should be deleted, but the item should not.
   i. Have a look in the test *removeComment*. Run the test to see that it fails. Adjust the cascading in order have the comment deleted. You don't have to adjust the *removeComment* method.



JBoss by Red Hat

Roadmap



- **Query options**
- Java Persistence queries
- Type safe criteria queries
- Collection filter
- Dynamic data filters
- Lab 6

Query Options

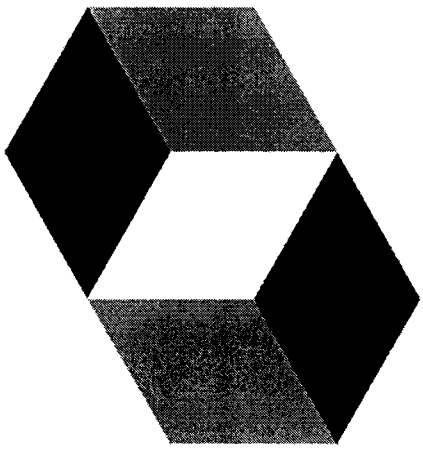
- Java Persistence
  - Java Persistence Query Language
  - Type safe Criteria Queries
    - Since Java Persistence 2.0
    - Uses a static meta model -> Java 6 annotation processor
    - Very verbose
  - SQL queries
    - Result set mapping possible = transform the resultset into Java objects
- Hibernate extensions
  - Hibernate Query Language
    - Same as JPA QL
    - Some extensions to JPA QL
  - Type-safe and extensible Criteria query API
    - Less verbose as compared to JPA criterias

Query Options

- Java Persistence and Hibernate Query language
  - Understandable for SQL aware developers
  - Refers to entity and properties instead of table names and columns
- JPA and Hibernate Criteria
  - Well suited to build queries at runtime
  - Can be verbose
- SQL queries
  - Many projects don't need them
  - Should only be used for corner cases
  - Allows to migrate SQL based applications incrementally

JBoss by Red Hat

Roadmap



- Query options
- **Java Persistence queries**
- Type safe criteria queries
- Collection filter
- Dynamic data filters
- Lab 6

Java Persistence QL

```
EntityManager em =  emf.createEntityManager();
Query query = em.createQuery(
  "select m from Message m where m.id < :id");
query.setParameter("id", 5);
List result = query.getResultList();
for (Object o : result) {
  Message message = (Message) o;
  System.out.println(message.getText());
}

/* generic alternative without cast */
TypedQuery<Message> query = em.createQuery(
  "select m from Message m where m.id < :id",
  Message.class);
query.setParameter("id", 5);
List<Message> result = query.getResultList();
for (Message message : result) {
  System.out.println(message.getText());
}

/* Method chaining is possible as well */
List<Message> typedResult = em.createQuery(
   "select m from Message m where m.id < :id",
   Message.class)
   .setParameter("id", 5)
   .getResultList();
```

- Sample query
  - Find all messages with id lower than 5
- Create a query object
  - Queries refer to
    - Entity names
    - Entity attributes
  - You don't care about database tables and columns!
- Add parameter for where conditions
- Execute the query

Java Persistence QL - Parameters

```
Date tenYearsAgo = getDate();
List<Sender> typedResult = em.createQuery(
  "select s from Sender s where s.name = ?1 and "+
  "s.birthday > ?2", Sender.class)
 .setParameter(1, "Jim")
 .setParameter(2, tenYearsAgo)
 .getResultList();

List<Sender> typedResult = em.createQuery(
 "select s from Sender s where s.name = :name " +
 "and s.birthday > :date", Sender.class)
 .setParameter("name", "Jim")
 .setParameter("date", tenYearsAgo)
 .getResultList();
```

- Always use parameters
  - Think of SQL injection
- Named parameters
  - :foo
- Positional parameters
  - :1
  - Starts with 1

Java Persistence QL - Restrictions

```
/* Select invoices having the given ids */
em.createQuery("select i from Invoice i where
i.id in (:ids)").setParameter("ids",
Arrays.asList(1,2,3)).getResultList();


/* Select invoices without invoice positions */
em.createQuery(
   "select i from Invoice i where "+
   "i.invoicePositions is empty");


/* Select invoices with more than 3 invoice
positions */
em.createQuery("select i from Invoice i "+
    "where size(i.invoicePositions) > 3");


/* Find the invoice for the given invoice
position */
em.createQuery("select i from Invoice i "+
  "where :p in elements (i.invoicePositions)")
  .setParameter("p", position)
  .getResultList();
```

- Works as in SQL
  - like, >, >=, <>, <, <=
  - is null, is not null
  - in, not in
- Some collection specific expressions
  - is empty
  - size
  - elements
  - member of

Joining Associations – SQL

- The `join` SQL operation combines the data from two tables

```
from ITEM_TABLE I inner join BID_TABLE B on I.ITEM_ID =
B.ITEM_ID
```

ITEM_ID	PRICE	NAME	BID_ID	ITEM_ID	AMOUNT
1	2.00	foo	1	1	10
1	2.00	foo	2	1	20
2	50.00	bar	3	2	15

```
from ITEM_TABLE I left outer join BID_TABLE B
   on I.ITEM_ID = B.ITEM_ID
```

ITEM_ID	PRICE	NAME	BID_ID	ITEM_ID	AMOUNT
1	2.00	foo	1	1	10
1	2.00	foo	2	1	20
2	50.00	bar	3	2	15
3	50.00	bazz	null	null	null

Joins – Inner Joins with HQL

- Inner join examples
  - Implicit inner join using dot notation in HQL

```
select b from Bid b
  where b.item.description like :param
```

  - Dot notation is possible for any singled ended association
    - Many to one
    - One to one
    - Component
  - Collections require an alias

```
select i from Item i join i.bids b where b.amount > 100
```

  - Explicit theta-style join with a condition

```
from User u, LogRecord l where u.username = l.username
```

Joins – Select

- You can select one or multiple alias of the query
  - Select a list of items using dot notation

```
select b.item from Bid b where b.amount = :param
```

  - Select a list of invoice positions

```
select p from Invoice i join i.invoicePositions p
  where i.number is null and p.total > :param")
```

  - Select invoice and positions into an object array

```
List result = em.createQuery(
 "select i,p from Invoice i join i.invoicePositions p "+
 "where i.number is null and p.total > 15").getResultList();

for (Object o : result) {
  Object array[] = (Object[]) o;
  System.out.println(String.format(
    "Invoice: %s, Position: %s", array[0], array[1]));
}
```

Joins – Outer Joins

- Dynamic fetching example using outer joins
  - HQL ignores the metadata fetch settings

```
em.createQuery(
  "select i from Item i left join fetch i.bids");
```

  - Careful with multiple collections
    → Cartesian products

```
em.createQuery(
  "select i from Item i left join fetch i.bids "+
  "left join fetch i.comments");
```

  - A query with inner join fetch can make sense, usually it doesn't.

Options for Joins

- Join options
  - The `with` keyword can be used for extra outer-join conditions

```
List results = em.createQuery(
   "select i from Item as i left outer join "+
   "i.bids as b "+
   "with b.amount > 42 ").getResultList();
```

  - At this time extra join conditions for outer joins are not supported on the Criteria API.

Externalizing Named Queries

- Calling a named query

```
em.createNamedQuery("findItemsByDescription",
 Item.class).setParameter("param", description)
 .getResultList();
```

- Named HQL or SQL queries (more about SQL later) in metadata

```
@NamedQueries({
 @NamedQuery(name = "findItemsByDescription", query =
   "select i from Item i where i.title like :param")
})
@Entity
public class Item {
...
```

Polymorphic Queries

- Queries to entities and super classes are possible
  - Sample: *BillingDetail* is the parent class of *CreditCard and BankAccount*
  - Query all kind of billing details

```
select b from BillingDetail b
```

  - Query just for credit cards

```
select c from CreditCard c
```

  - Query for everything in the database

```
select o from java.lang.Object o
```

  - And of course you should try

```
delete from java.lang.Object
```

Ordering Results with HQL

- Ordering results
  - The `order by` clause is used in HQL

```
from User user order by user.username desc

from Item item order by item.successfulBid.amount desc, item.id asc
```

HQL String Matching

- String matching examples
  - We can use wildcards for string pattern matching

```
from User user where user.firstname not like "%Foo B%"
```

  - We can also call arbitrary SQL functions in the where clause

```
from User user where lower(user.email) =
     'foo@hibernate.org'
```

HQL – Report Queries

- Report queries HQL examples:
  - Projection returning a collection of `Object[]`...

```
select item.id, item.description, bid.amount
    from Item item join item.bids bid
    where bid.amount > 100
```

  - Projection returning a collection of custom types with matching constructor

```
select new ItemRow(item.id, item.description, b.amount)
    from Item item join item.bids b
    where b.amount > 100
```

  - This custom class does not need to be mapped, it can be any Java type.

HQL - Aggregation

- Aggregation in HQL examples:
  - The aggregation functions are
    - `count(), min(), max(), sum(), avg()`
  - Count all items with a successful bid

```
select count(item.successfulBid) from Item item
```

  - The total of all successful bids

```
select sum(item.successfulBid.amount) from Item item
```

  - The result is an ordered pair of Floats
    - Note the special id property!

```
select min(bid.amount), max(bid.amount) from Bid bid
where bid.item.id = 1
```

HQL - Grouping

- Grouping in HQL examples:
  - If we aggregate, we have to group every property in the select

```
select bid.item.id, avg(bid.amount) from Bid bid
    group by bid.item.id
```

  - We can further restrict the result on the group:

```
select item.id, count(bid) ), avg(bid.amount)
    from Item item
        join item.bids bid
    where item.successfulBid is null
    group by item.id
        having count(bid) > 10
```

  - Use report queries (projection, aggregation, grouping, dynamic instantiation)
    - To optimize performance
    - Only retrieve required data.

JB297 - Hibernate Copyright © 2011 – Red Hat Inc.

HQL - Subqueries

- Examples:
  - Correlated subquery, total `items` sold by users, if more than `10`

```
from User user where 10 < (
  select count(item) from user.items where
   item.successfulBid is not null )
```

  - Uncorrelated subquery, all bids that are `1` within the maximum
    - Can always be written as several queries.

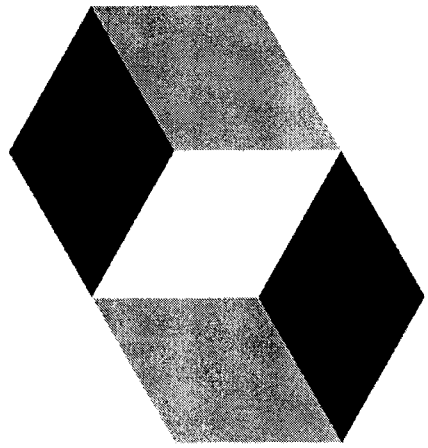
```
from Bid bid where bid.amount + 1 >= (
    select max(b.amount) from Bid b )
```

  - We can use SQL quantifiers, `ANY/ALL/SOME/IN`

```
from Item item where 100 > all (select b.amount from item.bids b)
from Item item where 100 < any (select b.amount from item.bids b)
from Item item where 100 = some (select b.amount from item.bids b)
from Item item where 100 in (select b.amount from item.bids b)
```

JBoss by Red Hat

Roadmap



- Query options
- Java Persistence queries
- **Type safe criteria queries**
- Collection filter
- Dynamic data filters
- Lab 6

Java Persistence Criteria Queries

```
CriteriaBuilder builder =
   em.getCriteriaBuilder();

CriteriaQuery<Message> query =
   builder.createQuery(Message.class);

Root<Message> root =
   query.from(Message.class);

query.where(builder.lessThan(
   root.get(Message_.id), 5L));

List<Message> list =
   em.createQuery(query).getResultList();

for (Message message : list) {
  System.out.println(message);
}
```

- Sample query
  - Find all messages with id lower than 5
- Create a query object
  - It defines the return type
- Define what you want to select
- Add where conditions
  - You can use String constants
  - Or the typesafe metamodel *Message_.id*
- Execute the query

Java Persistence Criteria Queries - Motivation

```
List<Message> search(Integer mLength,
  String sender, Date sent) {
EntityManager em = getEntityManager();
CriteriaBuilder b = em.getCriteriaBuilder();
CriteriaQuery<Message> q = b.createQuery(Message.class);
Root<Message> r = q.from(Message.class);
List<Predicate> p = new ArrayList<Predicate>();
if(mLength != null){
 Predicate gt = b.gt(b.length(r.get(Message_.text)), mLength);
 p.add(gt);
}
if(sent != null){
 Predicate equal =b.equal(root.get(Message_.sendDate),sent);
 p.add(equal);
}
if(sender != null){
 Join<Message,Sender> se =r.join(Message_.sender);
 Predicate l = b.like(se.<String>get(Sender_.name), sender);
 p.add(l);
}
query.where(predis.toArray(new Predicate[0]));
return em.createQuery(query).getResultList();
}
```

- Dynamic Queries
  - Query is build on runtime
  - Use cases:
    - Partially filled search form
    - Data dependent queries
- Sample:
  - Shows a search method
  - Adds only conditions if the parameter is not null

Java Persistence Criteria Queries - Restrictions

```
/* Select invoices having the given ids */
query = builder.createQuery(Message.class);
query.from(Message.class);
Predicate inP = r.get(Message_.id).in(Arrays.asList(1L, 2L, 3L));
query.where(inP);
em.createQuery(query).getResultList();
/* messages without a text */
r = query.from(Message.class);
Predicate notNull = r.get(Message_.text).isNotNull();
query.where(notNull);
/* Select invoices without invoice positions */
Root<Invoice> root = query.from(Invoice.class);
Predicate predicate = builder.isEmpty(root.get
  (Invoice_.invoicePositions));
query.where(predicate);
/* Select invoices with more than 3 invoice positions */
Root<Invoice> root = query.from(Invoice.class);
Predicate gt = builder.gt(builder.size(root.get
  (Invoice_.invoicePositions)), 3);
query.where(predicate);
/* Find invoice for the given invoice position */
Root<Invoice> root = query.from(Invoice.class);
ListJoin<Invoice, InvoicePosition> positions =
  root.join(Invoice_.invoicePositions);
Predicate equal = builder.equal(positions, aPos);
q.where(equal);
```

- SQL like restrictions are supported
  - like, >, >=, <>, <, <=
  - is null, is not null
  - in, not in
- Additional collection specific expressions
  - is empty
  - size

Java Persistence Criteria Queries - Joins

```
q = builder.createQuery(Invoice.class);
root = q.from(Invoice.class);
ListJoin<Invoice, InvoicePosition> iPositionJoin =
root.join(Invoice_.invoicePositions);
Join<InvoicePosition, Article> articleJoin =
  iPositionJoin.join(InvoicePosition_.article);

Predicate predicate =
  builder.equal(articleJoin.get(Article_.name),
  "Hibernate");


/* chaining is possible */
Join<InvoicePosition, Article> articleJoin = root
 .join(Invoice_.invoicePositions)
 .join(InvoicePosition_.article);
Predicate predicate =
  builder.equal(articleJoin.get(Article_.name),
  "Hibernate");
```

- Use the *join* method of root elements or joined elements
- Sample
  - Find invoices of articles on Hibernate

JB297 - Hibernate
Copyright © 2011 – Red Hat Inc.

Java Persistence Criteria Queries - Ordering

```
/* Invoice ordered by number and id */
q = builder.createQuery(Invoice.class);
Root<Invoice> root = q.from(Invoice.class);
q.orderBy(
  builder.desc(root.get(Invoice_.number)),
  builder.asc(root.get(Invoice_.id)));

/* Select Messages ordered by the sender's name */
query = builder.createQuery(Message.class);
Root<Message> root = query.from(Message.class);
final Join<Message, Sender> senderJoin =
root.join(Message_.sender);
query.orderBy(builder.asc(senderJoin.get(Sender_.na
me)));
```

- Ordering
  - Ascending, descending
  - *Orderby* can take multiple sort orders
  - Ordering of root and joined elements is possible

Java Persistence Criteria Queries – Reporting Queries

```
/* Select id and number of invoices  */
CriteriaQuery<Object[]> query = b.createQuery(Object[].class);
root = query.from(Invoice.class);
Path<Long> idPath = root.get(Invoice_.id);
Path<String> numberP = root.get(Invoice_.number);
query.multiselect(idPath, numberP);
List<Object[]> resultList = em.createQuery(query).getResultList();
for (Object[] objects : resultList) {
  System.out.println(String.format(
    "id: %s, number: %s", objects[0], objects[1]));
}
/* same query using tuples */
CriteriaQuery<Tuple> query = b.createTupleQuery();
root = query.from(Invoice.class);
Path<Long> idPath = root.get(Invoice_.id);
Path<String> numberP = root.get(Invoice_.number);
query.multiselect(idPath.alias("id"), numberPath.alias("number"));
List<Tuple> tuples = em.createQuery(tupleQuery).getResultList();
for (Tuple tuple : tuples) {
  System.out.println(String.format(
    "id: %s, number: %s",
    tuple.get(0), tuple.get(1)));
  System.out.println(String.format(
    "id: %s, number: %s",
    tuple.get("id"), tuple.get("number")));
}
```

- Selection of one or multiple scalar values
- Two approaches
  - Selecting object array on criteria query
  - Tuple queries
    - Allow to access objects using String alias
  - Reporting classes
    - See next page

Java Persistence Criteria Queries – Reporting Queries

```
/* The reporting class  */
public class InvoiceReport {
  private Long id;
  private String number;

  public InvoiceReport(Long id, String number) {
    this.id = id;
    this.number = number;
  }
  public Long getId() {
    return id;
  }
  public String getNumber() {
    return number;
  }
}

CriteriaQuery<InvoiceReport> q =
  builder.createQuery(InvoiceReport.class);
root = q.from(Invoice.class);
idPath = root.get(Invoice_.id);
numberPath = root.get(Invoice_.number);
q.multiselect(idPath, numberPath);
List<InvoiceReport> invoiceReports =
  em.createQuery(q).getResultList();
```

- Reporting classes
  - Return a class as result of a query for scalar values
  - Requires a matching constructor

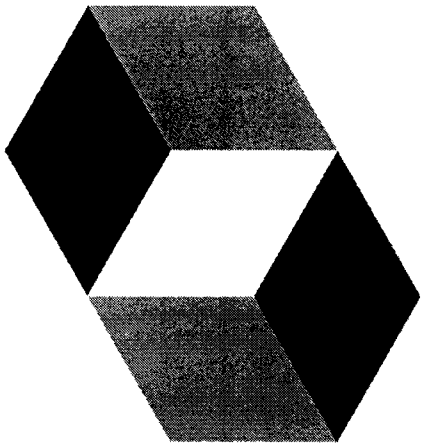
Java Persistence Criteria Queries – Aggregation

```
CriteriaQuery<Tuple> tupleQuery =
  builder.createTupleQuery();
root = tupleQuery.from(Invoice.class);
Expression<Long> count =
  builder.count(root.get(Invoice_.id));
Path<String> numberPath =
  root.get(Invoice_.number);
tupleQuery.groupBy(numberPath);

tupleQuery.multiselect(count.alias("count"),
  numberPath.alias("number"));
List<Tuple> tuples =
  em.createQuery(tupleQuery).getResultList();
```

- Aggregation like min, max, count
- SQL rules apply
  - Group by is required for additional properties

JBoss by Red Hat

Roadmap



- Query options
- Java Persistence queries
- Type safe criteria queries
- **Collection filter**
- Dynamic data filters
- Lab 6

Filters

- Collection filters
  - Hibernate Extension
  - We can filter the collection of bids of an already loaded `Item`

```
Query q = session.createFilter( item.getBids(),
          "order by this.amount asc" );
```

  - HQL filter queries have an implicit from and where clause

```
// This is valid...
Query q = session.createFilter( item.getBids(), "" );

// and useful!
List result =
q.setFirstResult(50).setMaxResults(100).list()
```

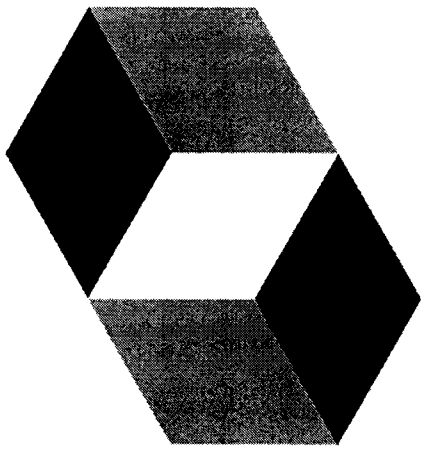
  - A filter doesn't have to return objects from the collection

```
session.createFilter( item.getBids(),
           "select elements(this.bidder.bids)" );
```

    - In fact, it never touches the original collection...

JB297 - Hibernate Copyright © 2011 – Red Hat Inc.

Roadmap



- Query options
- Java Persistence queries
- Type safe criteria queries
- Collection filter
- **Dynamic data filters**
- Lab 6

Understanding Data Filters

- Ability to define a restriction, like a where clause, for a particular unit of work.
  - Hibernate extension !
- Filters are powerful and can be used in many scenarios
  - Temporal views
  - Data-level authorization
  - etc...
- Think about database views, but
  - At the application layer
  - More dynamic (parameters as runtime argument)
- For example, users should only see things they are allowed to see

```
<filter-def name="accessFilter">
    <filter-param name="userLevel" type="int"/>
</filter-def>
```

Attaching Filters to Classes

- Defined filters can be enabled for persistent classes (or collections)

```
@FilterDef(name = "accessFilter", parameters =
  {@ParamDef(name = "regionName", type = "string")})
@Filters({@Filter(name = "accessFilter",
  condition = ":userLevel >= access_lvl")    })
public class Desert implements Serializable {
```

Annotations

```
<class name="Category" ...>
    ...
    <property name="accessLevel" type="int"
              column="access_lvl"/>
    <filter name="accessFilter">
      <![CDATA[ :userLevel >= access_lvl ]]>
    </filter>
</class>
```

XML

- The access level of the user is set at runtime...

Filtering a Session

- First, enable the `Filter` for a particular `Session`

```
Filter f = session.enableFilter("accessFilter");
```

- Bind arguments to `Filter` parameters

```
f.setParameter("userLevel", currentUser.getLevel() );
```

- Now all data in this `Session` is filtered

```
List filtered = session.createQuery("from Category").list();
Iterator iter = category.getSubcategories().iterator();
```

Lab

Lab 6 – Hibernate Query

- Details and instructions for the lab can be found in the lab guide.
- The instructor will answer any questions and will determine the stop time.

JBoss by Red Hat

Summary

- In this lesson, you learned about:
  - Different approaches to query data with Hibernate
  - Binding techniques, externalizing queries, using HQL, and using joins and subqueries.
  - Creating queries dynamically
  - What data filters are and how to use them.

JB297 - Hibernate
Copyright © 2011 – Red Hat Inc.

Lab 6: HQL and Criteria queries

Requirements

- JB297-Hibernate Lab Archives
- Pre-Configured JBoss Developer Studio IDE
- Time: 30 Minutes

Goal

This lab demonstrates the use of entity relations and inheritance

Description

- In this lab you need to write a lot of queries using both HQL and criterias. There are a number of tasks and each tasks can be validated running the Junit tests.

- The tasks are marked with TODO markers. Search each of the source files listed below for Tasks

  Example Task:
  /* TODO: A Task Title
  * Step 1: Code Something Here, As Instructed - Tags
  */

- You can find a detailed description below.
- Import JB297's **lab-6** project as an existing project, copied to the workspace
- The working/full source code is available in the lab-6/solution/ directory. View this directory for help, or backup the lab-6/src/ directory, rename the solution folder to src.

Project Instructions

1. Use HQL for the tasks 2 to 4.

2. Select all users having a *username* starting with 'jo' and who are not an *admin.*
   i. Run the test *selectUser* in the class *MappingTest* to validate your changes. Don't forget to start the database.

3. Select all categories with less than 3 items.
   i. Run the test *selectItems* in the class MappingTest to validate your changes.

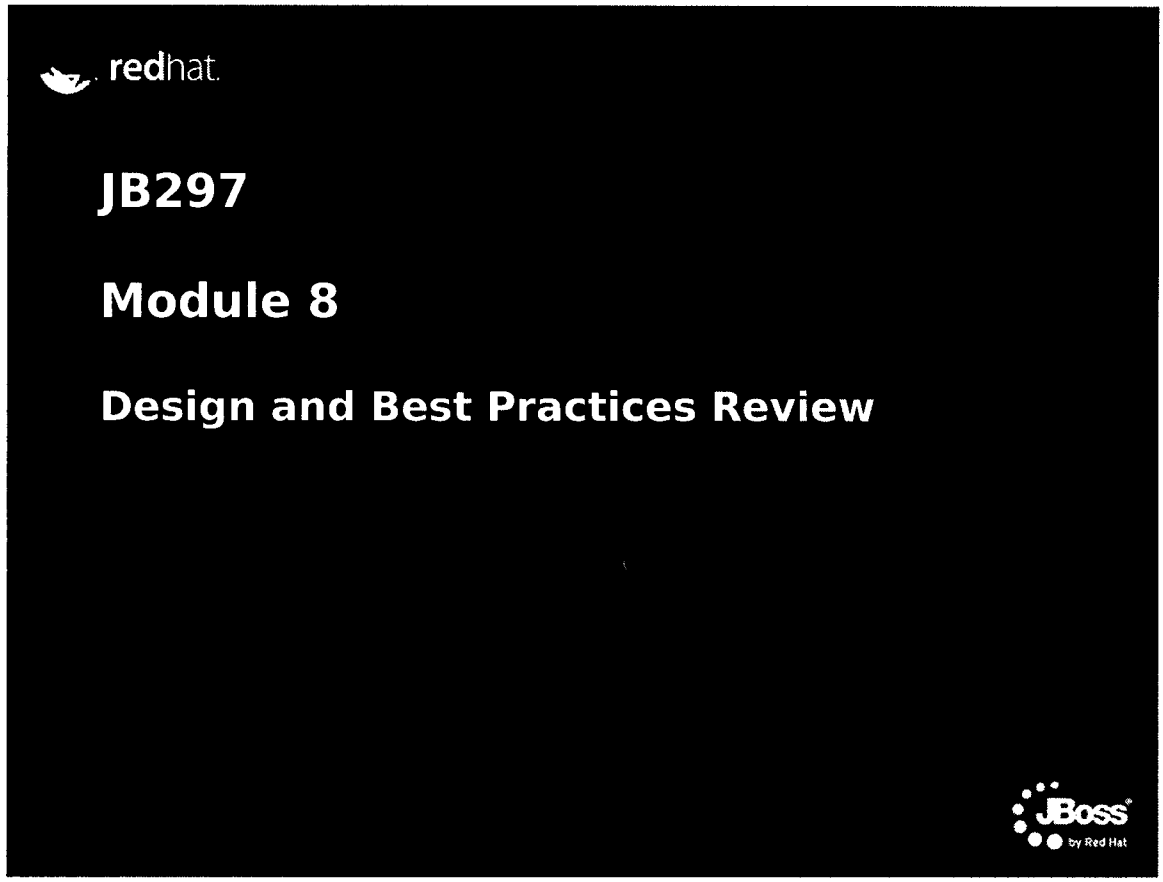
4. Select all categories having an item sold by a user with the firstname 'John'.
   i. Run the test *selectJohnsCategories* in the class MappingTest to validate your changes.

5. Reporting Query: The code in *categoryReport* causes a lot of SQL queries and loads complete object graphs into memory. Try to load the data much more efficiently using a reporting query.
   i. Run the test *categoryReport* in the class MappingTest to validate your changes.

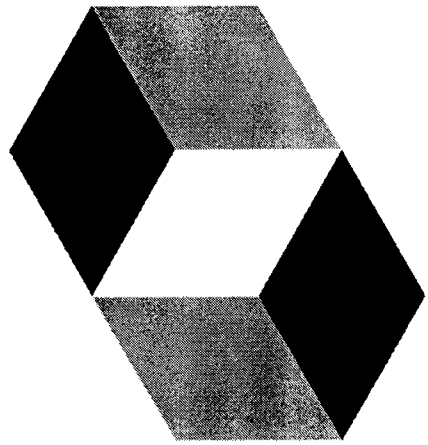
6. Use criteria queries for the task 7 and 8.

7. Select all users having a *username* starting with 'jo' and who are not an *admin.*
   i. Run the test *selectUserWithCriteria* in the class *MappingTest* to validate your changes. Don't forget to start the database.

8. Select all categories having an item sold by a user with the username 'John'.
   i. Run the test *selectJohnsCategoriesWithCriteria* in the class MappingTest to validate your changes.



Roadmap



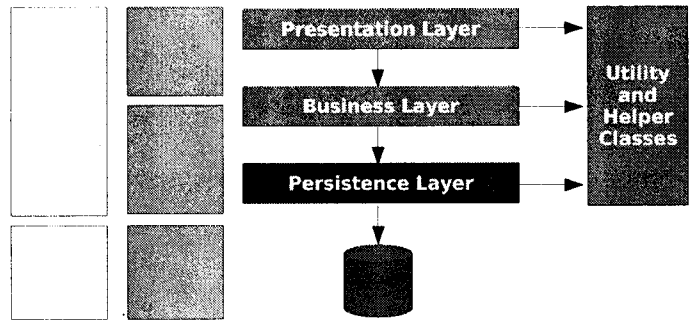
- **Application Design**
  - **Layers**
  - **Use Case Examples**
  - **DAOs**
  - **DTOs**
  - Conversations with Hibernate
  - Audit Logging and Hibernate Events
  - Data Validation
  - Open-Session-In-View Pattern

Application Design

- Hibernate can be used everywhere
  - Environments
    - Standalone Java applications (even rich clients)
    - Web containers with JSP and Servlets (Tomcat, Jetty, etc.)
    - Web-applications with any framework (Struts, Tapestry, Spring, etc.)
    - In any application server with EJB (JBoss, Web Logic, Websphere, etc.)
    - In enterprise applications with EJB session beans
    - In lightweight containers with POJO models (Spring, Picocontainer, etc.)
  - Hibernate just needs a datasource and you just need a Hibernate SessionFactory to load and save objects.

Application Design – Looking at Layers

- Layered applications
  - Layers group concerns
    - The presentation layer deals with user interaction
    - The business layer implements and executes business logic
    - The persistence layer manages persistent data
    - The datastore layer holds persistent state
  - Two and three tier architecture



Application Design – Simple Use Case

- A simple use case
  - When a user places a new bid for an item, we
    - Check the amount of the bid, must be greater than any existing bid
    - Check that the auction has not yet ended
    - Create a new Bid instance for the selected Item
    - Inform the user if any of the checks failed
  - Use case executes in a single HTTP request/response cycle
    - Maps to a single transaction
  - We can
    - use a servlet in a two-tier system first
    - expand with an EJB facade to three
  - We write quick and dirty code now... and clean up later.

Application Design – Use Case Implementation

```
public void execute() {
  Long itemId = ...       // Get all required values from request
  try {
    EntityManager em = EntityManagerUtil.createEntityManager();
    em.getTransaction().begin();
    try {

       // Load requested Item
      Item item = (Item) em.find(Item.class, itemId);


      // Check auction still valid
      if ( item.getEndDate().before( new Date() ) ) {
        ...   // Forward to error page
      }


      // Check amount of Bid
      Query q = em.createQuery("select max(bid.amount)" +
              " from Bid bid where bid.item.id = :item");
     q.setParameter("item", item.getId());
     BigDecimal currentMaxBidAmount =(BigDecimal)q.uniqueResult();
     if (currentMaxBidAmount.compareTo(bidAmount) > 0)
        ...   // Forward to error page
```

Example

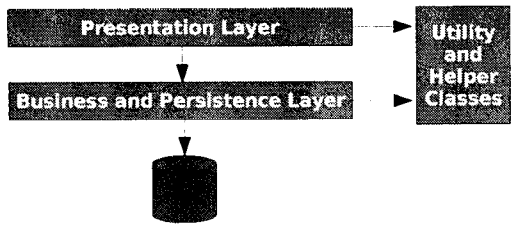
Application Design – Use Case Implementation

```
            // Add new Bid to Item
            User bidder = (User) em.getReference(User.class, userId);
            Bid newBid = new Bid(bidAmount, item, bidder);
            item.addBid(newBid);
            ...     // Place new Bid in scope for next page
            em.getTransaction().commit();
            ...     // Forward to success page
    } catch (RuntimeException relevantException) {
      if (em.getTransaction().isActive()) {
        try {
         em.getTransaction().rollback();
        } catch (PersistenceException rollBackException) {
          logger.warn("Rollback failed", rollBackException);
        }
      throw relevantException;
      }
    } finally {
      em.close();
    }
}
```

Example

- *NOTE: We are mixing controller logic, business logic, and data access in this code - not good at all. Let's start to clean it up...*

Application Design – Layer



- Introducing a Business Layer
  - Responsible for business logic
  - Provides transaction handling
  - Is testable without GUI

Application Design – Layer

```
public void execute(){
Long itemId = null; // get the value from user input
BigDecimal amount = null; // user input
Long bidderId = null; // user input

final AuctionService auctionService =
  ServiceFactory.createService(AuctionService.class);
try {
  auctionService.placeBid(itemId, amount, bidderId);
  forwardTo("success");
} catch (AuctionService.BidFailedException e) {
  if(e.getReason() == Reason.AMOUNT_TO_LOW)
    forwardTo("amountToLow");
  else if(e.getReason() == Reason.AUCTION_ENDED)
    forwardTo("auctionEnded");
}
}
```

- Presentation Layer
  - Extracts the user input
  - Calls the business logic
  - Takes care of the navigation

Application Design – Layer

```
public class AuctionServiceImpl implements
  AuctionService {
private EntityManager em;
  @Inject
  public void setEm(EntityManager em) {
    this.em = em;
  }
  @Override
  @Transactional
  public void placeBid(Long itemId, BigDecimal amount,
    Long bidderId) throws BidFailedException {
    // Load requested Item
    Item item = (Item) em.find(Item.class, itemId);
    // Check auction still valid
    if (item.getEndDate().before(new Date())) {
      throw new
        BidFailedException(Reason.AUCTION_ENDED);
  } ...
```

- Business Layer
  - Business Logic
  - Declarative Transaction Handling
  - Hibernate Code
- Cross cutting concerns
  - Transaction Handling
  - Exception handling
  - Moved to framework
    - Dependency Injection
    - EJB3

Application Design – Layer

```
... // continued
  // Check amount of Bid
  Query q = em.createQuery("select max(bid.amount)" +
  " from Bid bid where bid.item.id = :item");
  q.setParameter("item", item.getId());
  BigDecimal currentMaxBidAmount = (BigDecimal)
   q.getSingleResult();
  if (currentMaxBidAmount.compareTo(amount) > 0)
    throw new
      BidFailedException(Reason.AMOUNT_TO_LOW);
  User bidder = em.getReference(User.class, bidderId);
  Bid newBid = new Bid(amount, item, bidder);
  item.addBid(newBid);
  }
}
```

- Business Layer
  - Throws exceptions

Application Design – Domain Objects

- Persistent Domain Object
  - Enrich domain model with business logic
  - We separate control logic from business logic and move business logic into the domain model as business methods

```
public class Item {
    ...
  public Bid placeBid(BigDecimal amount, User bidder) throws BidFailedException {
    // Check auction still valid
    if (getEndDate().before(new Date())) {
         throw new BidFailedException(BidFailedException.Reason.AUCTION_ENDED);
    }

    if (getMaxBid().compareTo(amount) > 0)
      throw new BidFailedException(BidFailedException.Reason.AMOUNT_TO_LOW);

    final Bid newBid = new Bid(amount, this, bidder);
    addBid(newBid);
    return newBid;
  }
}
```

Example

Application Design – Domain Objects

- Action with a persistent domain object
  - Now we can move most business logic out of `execute()`:

```
public class AuctionServiceImpl implements AuctionService
{
  @Override
  @Transactional
  public void placeBid(Long itemId, BigDecimal amount,
    Long bidderId) throws BidFailedException {

  // Load requested Item
  Item item = (Item) em.find(Item.class, itemId);

  // Check amount of Bid
  Query q = em.createQuery("select max(bid.amount)" +
    " from Bid bid where bid.item.id = :item");
  q.setParameter("item", item.getId());
  BigDecimal currentMaxBidAmount = (BigDecimal)
     q.getSingleResult();

  User bidder = em.getReference(User.class, bidderId);
  Bid newBid = item.placeBid(amount, bidder);

}
```

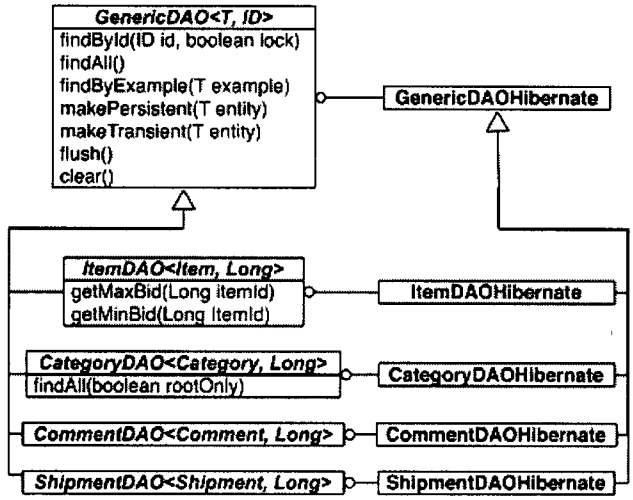
This query is much more performant than iterating over all bids in the Item class, but still pollutes our control logic...

Application Design – DAO

- Creating a Data Access Object (DAO)
  - A DAO encapsulates persistence operations
    - Actually already provided by JPA/Hibernate
  - Motivation is to provide higher level data access methods
  - Common design questions
    - *What granularity should my DAO classes have? One per entity?*
    - *What granularity should my DAO methods have?*
    - *What granularity should my DAO method parameters have?*
    - *Should I use DAO at all? Do I need persistence layer portability?*

Application Design – DAO

- Generic implementation of a DAO



Application Design – DAO

- Generic super-interface
  - Provides basic state management operations

```
public interface GenericDAO<T, ID extends Serializable> {
  T findById(ID id, boolean lock);
  List<T> findAll();
  List<T> findByExample(T exampleInstance,String... excludeProperty);
  T makePersistent(T entity);
  void makeTransient(T entity);
  void flush();
  void clear();
}
```

  - Clients can (should they?) control the persistence context with `flush()` and `clear()`

Application Design – DAO

- Implementation of the generic super-interface:

```
public abstract class GenericHibernateDAO<T, ID extends Serializable>
implements GenericDAO<T, ID> {
  private Class<T> persistentClass;
  private EntityManager em;
  public GenericHibernateDAO() {
    this.persistentClass = (Class<T>) ((ParameterizedType)getClass()
      .getGenericSuperclass()).getActualTypeArguments()[0];
  }
  @Inject
  public void setEntityManager(EntityManager em) {
    this.em = em;
  }
  public Class<T> getPersistentClass() {
    return persistentClass;
  }
  ...
```

Application Design – DAO

```
...
public T findById(ID id, boolean lock) {
  T entity;
  if (lock) {
    entity = em.find(getEntityBeanType(), id);
    em.lock(entity, javax.persistence.LockModeType.WRITE);
  } else {
    entity = em.find(getEntityBeanType(), id);
  }
  return entity;
}
@SuppressWarnings("unchecked")
public List<T> findAll() {
  return em.createQuery("from " + getEntityBeanType().getName())
    .getResultList();
}
public void persistent(T entity) {
  em.persist(entity);
}
public T merge(T entity) {
  return em.merge(entity);
}
public void remove(T entity) {
  em.remove(entity);
}
...
```

Application Design – DAO

```
...

public void flush() {
  em.flush();
}

public void clear() {
  em.clear();
}

@SuppressWarnings("unchecked")
Public List<T> findByCriteria(org.hibernate.criterion.Criterion...
criterion) {
  // Using Hibernate, more difficult with EntityManager and EJB-QL
  org.hibernate.Session session = (Session) em.getDelegate();
  org.hibernate.Criteria crit =
session.createCriteria(getEntityBeanType());
  for (org.hibernate.criterion.Criterion c : criterion) {
    crit.add(c);
  }
  return crit.list();
}
```

- *If you expose `findByCriteria()` on the `public` interface, clients have more flexibility but are bound to Hibernate Criteria...*

Application Design – DAO

- Implementation of the entity DAO
  - Interface adds non-CRUD operations
  - Implementation extends the generic superclass

```
public interface ItemDAO extends GenericDAO<Item, Long> {
  Bid getMaxBid(Long itemId);
  Bid getMinBid(Long itemId);
}
public class ItemDAOHibernate extends GenericHibernateDAO<Item, Long>
  implements ItemDAO {
  public Bid getMaxBid(Item item) {
    List<?> result = em
        .createNamedQuery("Item-getMaxBid")
        .setParameter("itemId", item.getId())
        .getResultList();
    return (result.size() > 0) ? (Bid) result.get(0) : null;
  }
  public Bid getMinBid(Item item) {
    List<?> result = em
      .createNamedQuery("Item-getMinBid")
      .setParameter("itemId", item.getId())
      .getResultList();
    return (result.size() > 0) ? (Bid) result.get(0) : null;
  }
}
```

Application Design – DAO

- Business Service with a DAO
  - Our business service doesn't have any persistence logic and little business code

```
@Override
@Transactional
public void placeBid(Long itemId, BigDecimal amount, Long bidderId)
throws BidFailedException {

        // Load requested Item
        Item item = itemDao.findById(itemId);

        // Check amount of Bid
        BigDecimal currentMaxBidAmount = itemDao.findMaxBid(item);

        User bidder = userDao.findById(bidderId);
        Bid newBid = item.placeBid(amount, bidder);

}
```

Rest of logic code

Application Design – EJB

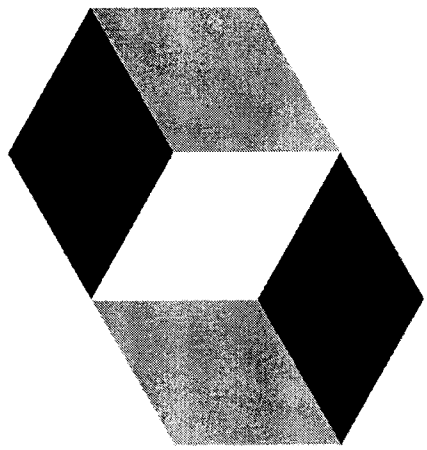
- Considering an EJB architecture
  - Transaction by demarcation and exception handling
  - Local access to Session Beans
    - Lazy loading is possible
      - See JBoss Seam framework
  - Remote access to Session Beans
    - Separate presentation and business tier
      - Inter-process communication has to be minimized (facade or command)
      - Keep database transactions as short as possible
      - A single user request should be a single request to the EJB tier
    - We can't use lazy loading while rendering the view!
    - Do we need Data Transfer Objects (DTOs)?

Application Design – DTO

- Data Transfer Objects
  - DTOs provide value objects for simple data transfer
    - Exists because fine-grained remote access is bad
    - EJB entity beans are not serializable, can't transport data
  - We have three reasons why we may consider DTO:
    - Externalization of the data
    - Separation of tiers (clear interfaces for several developers)
    - Easy assembly of required data (DTO assembler fetches data)
  - We have three reasons why we should not use DTO:
    - They only have state, no behavior ("not object-oriented")
    - Small changes to any business entity trigger many changes in DTOs
    - Lots of parallel class hierarchies

JBoss by Red Hat

Roadmap



- **Application Design**
  - Layers
  - Use Case Examples
  - DAOs
  - DTOs
  - **Conversations with Hibernate**
  - Audit Logging and Hibernate Events
  - Data Validation
  - Open-Session-In-View Pattern

Application Design – Conversation

- Conversations with Hibernate
  - The use case:
    - A new `Item` is created in draft state
    - The `User` may set the `Item` to pending state for approval
    - The `User` or any administrator can still edit the `Item`
    - A system administrator approves the auction, setting an active state
  - It is essential that an administrator sees the latest version of an `Item` before activating an auction!

Application Design – Conversation

- Analyzing the Conversation
  - Our Conversation spans two request/response cycles
    - First, the administrator selects a pending item for activation.
    - Second, the administrator activates the auction.
  - First, add an `approve()` business method to `Item`

```
public void approve(User administrator) throws BusinessException {
    if ( !administrator.isAdministrator() )
        throw new PermissionException("User not an administrator.");
    if ( !state == ItemState.PENDING )
        throw new IllegalStateException("Item is still in DRAFT.");
    state = ItemState.ACTIVE;
    approvedBy = administrator;
    approvalDatetime = new Date();
}
```

  - Now we write two routines:
    - first must load and display an `Item`
    - second checks for concurrent updates and approves the `Item`...

Application Design – Conversation

- The Hard Way
  - We discard all persistent instances between each request...
  - Load data in first request:

```
public Item viewItem(Long itemId) {
    return ItemDAO.getItemById(itemId);
}
```

  - Manual version check (session bean) in second request

We had to store the `itemID` and `itemVersion` somewhere!

```
public approveAuction(Long itemId, int itemVersion,
                                        Long adminId) {
    Item item = new ItemDAO().getItemById(itemId);
    User administrator = new UserDAO().getUserById(adminId);
    if ( !( itemVersion==item.getVersion() ) )
          throw new StaleItemException();
    item.approve(administrator);
}
```

JBoss by Red Hat

Application Design – Conversation

- Detached Objects
  - We use persistent instances in several requests and keep then between requests, e.g. in `HttpSession`...
  - We reattach the detached instance in the second request

```
public approveAuction(Item item, User admin) {
    new ItemDAO().saveOrUpdate(item);
    item.approve(admin);
}
```

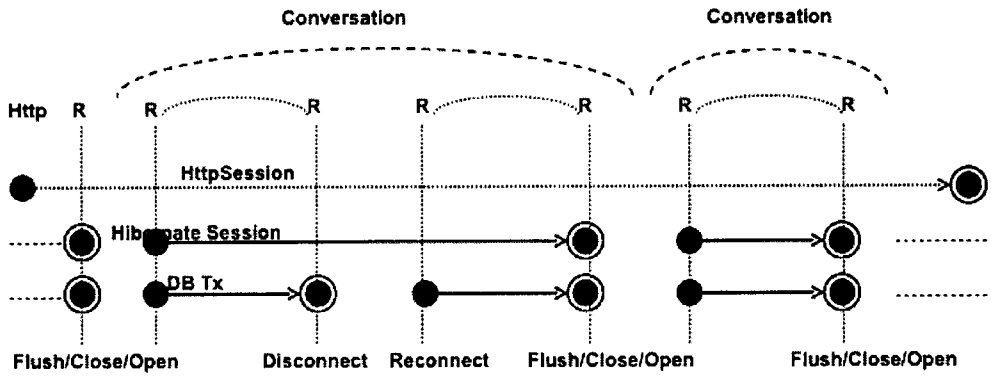
  - This looks much better, Hibernate will do version checking at flush.

Application Design – Conversation

- Extended Session
  - A single Hibernate `Session` spans the whole Conversation, it is disconnected from the JDBC connection and stored between requests... see the picture on next slide.
  - Where do we keep the Hibernate `Session` between requests?
    - Web applications use the `HttpSession`
    - EJBs use stateful session beans (e.g. a `UserSession` SSB)
    - Conversation scope → JBoss Seam
  - How do we automatically disconnect the Hibernate `Session`?
    - Web applications use an enhanced servlet `HibernateFilter`
    - EJB 3.0 has built-in extended EntityManager
  - How do we flush/close/open the Hibernate `Session`?
    - Demarcate conversation boundaries in your controllers

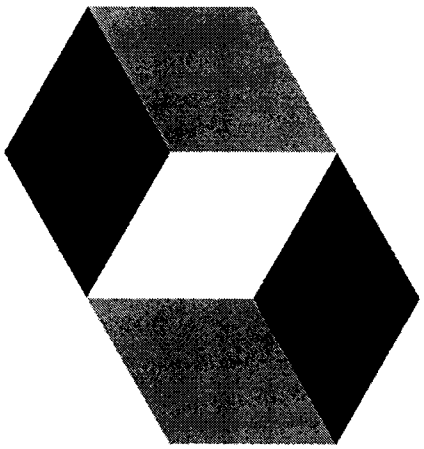
Application Design – Conversation

- Chained conversations
  - If you design with conversation, make them chained
  - Use one Hibernate `Session` per Conversation, not longer!



JBoss by Red Hat

Roadmap



- **Application Design**
  - Layers
  - Use Case Examples
  - DAOs
  - DTOs
  - Conversations with Hibernate
  - **Audit Logging and Hibernate Events**
  - Data Validation
  - Open-Session-In-View Pattern

Application Design – Audit Log

- Is a table that contains changes made to other tables
  - For example, when and by whom was an `Item` updated?
  - Usually includes the `item`, `user`, the `time` and `date`, and the type of event.
  - Audit logging may be implemented with a trigger and procedure in the database, or in the application.

```
Session session = factory.openSession();
Transaction tx = session.beginTransaction();
Serializable newId  = session.save(newItem);
AuditLog.logEvent( "Create", newId, Item.class,
        user.getId(), session.connection() );
tx.commit();
session.close();
```

  - *Note: Mixing audit logging and business logic is a bad practice, but we'd like an automatic call of `logEvent()` by Hibernate. So, how could we solve this problem??*

Hibernate and JPA Events

- Event based architecture
- Events are fired before and after data is loaded, updated or persisted
- Use cases
  - Setting last change date or user
  - Calculating values
  - Trigger like operations
- Hibernate extension
  - Event hooks for virtual any operation
  - All the methods of the Session interface correlate to an event. You have a `LoadEvent`, a `FlushEvent`, etc
  - See the XML configuration-file DTD or `org.hibernate.event` package for all defined event types
  - Alternative: Hibernate Interceptors

Java Persistence Events

```
@Entity
@EntityListeners(MessageAudit.class)
public class Message { … }

/* The event listener */
public class MessageAudit {
  final Logger logger =
   LoggerFactory.getLogger(MessageAudit.class);

  @PrePersist
  void onPersist(Message message){
    logger.debug("Persisting {}", message);
    message.setLastChangeUser(
      UserService.getCurrentUser());
  }

  @PreUpdate
  void onUpdate(Message message){
    logger.debug("Updating {}", message);
    message.setLastChangeUser(
      UserService.getCurrentUser());
  }
}
```

- Event types
  - @PrePersist, @PostPersist
  - @PreRemove, @PostRemove
  - @PreUpdate, @PostUpdate
  - @PostLoad
- Message class can be its own event listener
- Default event listener
  - Configured in XML mappings

Hibernate Event System

- Events can be used instead of or with Interceptors
- All Hibernate core operations produce corresponding events
  - `Post/Pre/LoadEvent`
  - `Post/Pre/UpdateEvent`
  - `Post/Pre/DeleteEvent`
  - etc...
  - `InitializeCollectionEvent`
  - `DirtyCheckEvent`
  - `FlushEntityEvent`

- *Note: Two options to implement a custom listeners*
  - Implement the complete functionality of corresponding interface
  - Extend the Hibernate core listeners.

Hibernate Events – Implement Security Checks

- A custom listener for LoadEvent

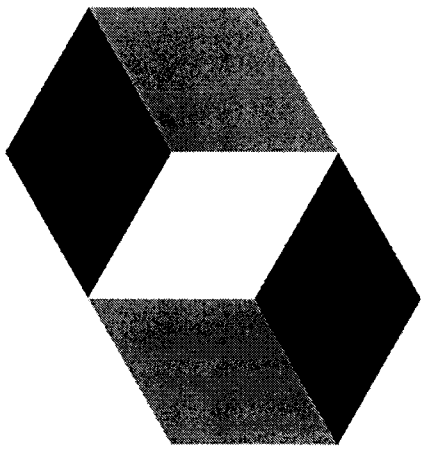
```
public class AccessListener extends DefaultLoadEventListener
{
    public Object onLoad(LoadEvent event,
                         LoadEventListener.LoadType loadType)
                  throws HibernateException {
        if ( !MySecurity.isAuthorized(
              event.getEntityName(), event.getEntityId() ) )
            throw MySecurityException("Unauthorized access");
        return super.onLoad(event, loadType);
    }
}
```

  - Registering the listener in Hibernate configuration

```
<session-factory>
    ...
    <listener type="load" class="AccessListener"/>
```

JBoss by Red Hat

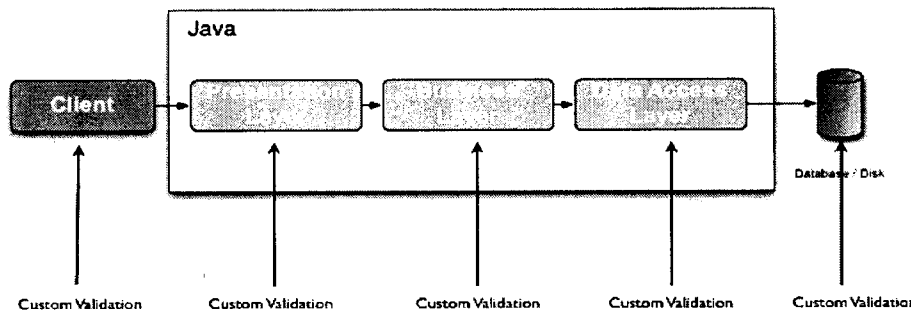
Roadmap



- **Application Design**
  - Layers
  - Use Case Examples
  - DAOs
  - DTOs
  - Conversations with Hibernate
  - Audit Logging and Hibernate Events
  - **Data Validation**
  - Open-Session-In-View Pattern

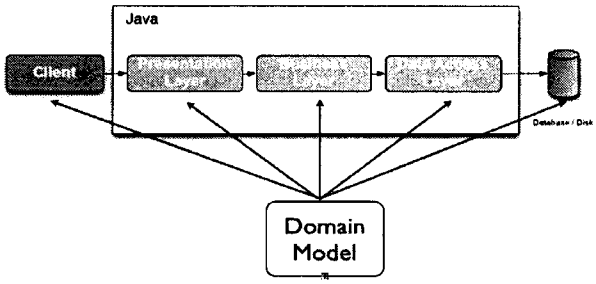
Hibernate Validator Framework - Overview

- Validating objects against different criteria is a requirement in basically every software development project.
  - Examples could include:
    - a given object might never be allowed to be `null`.
    - a number could be required to be always in the interval [`1 ... 100`].
    - a String might be expected to match some regular expression or to represent a valid e-mail address, credit card number, license plate etc.



Hibernate Validation Framework - Overview

- *Hibernate Validator* is the reference implementation for *JSR 303: Bean Validation*
  - JSR 303 defines a metadata model and API for JavaBean validation
  - Default metadata source is annotations
    - Ability to override and extend the meta-data through the use of XML validation descriptors
  - API is not tied to a specific application tier or programming model
    - Hibernate and Java Persistence bootstraps validation if the validation JAR is available
    - Validations can be executed programatically
    - JSF 2 provides native integration for the web tier



Validation – Beginnings with Constraints

- Constraints in Bean Validation are expressed via Java annotations
- A differentiation between three different type of constraint annotations must be managed
  - Field
  - Property
  - Class-level

  - *Note: Not all constraints can be placed on all of these levels.*
    - *Class-level constraints allows cross field validation*
    - *None of the default constraints defined by Bean Validation can be placed at class level*
    - *The* `java.lang.annotation.Target` *annotation in the constraint annotation itself determines on which elements a constraint can be placed*
    - *Consider creating custom constraints*

Validation Constraints

```
@Entity
@ValidUser
public class User {
 @Id
 @GeneratedValue
 private Integer id;

 @Length(min = 6)
 private String name;

 @Length(min = 6)
 @ValidPassword
 private String password;

 @Email
 private String email;

...
```

- Sample
  - Mixture of predefined and custom constraints
  - Name and password have longer than 6 characters
  - Password follows the rules defined in the custom *ValidPassword* validator
  - Email is a valid
  - User object is checked by rules in *ValidUser* validator

Validation Constraints

```
final EntityManager em =
   emf.createEntityManager();
try {
  em.getTransaction().begin();
  em.persist(new User("Sebastian"));
  em.getTransaction().commit();
}
catch (ConstraintViolationException e)
{
  for (ConstraintViolation v :
       e.getConstraintViolations()) {
    System.out.println(v);
  }
} finally {
  em.close();
}

/*
Please note, we have ignored proper
exception handling and did not try to
role back the transaction.
*/
```

- Java Persistence and Hibernate bootstrap validation by default
- Validation throws *ConstraintViolationException*
  - Try catch required to react to validation
- Exception provides detailed information about the validation failures

Validation Constraints

- Predefined constraints
  - `@Length`
  - `@Range`
  - `@Email`
  - `@NotEmpty`
  - `@NotBlanc`
  - `@ScriptAssert`
    - Allows developers to execute scripts to validate data
    - Java Scripting API JSR 223
    - Java 6 includes JavaScript
    - Other: Ruby, Groovy, Python
- Custom constraints
  - Allows to provide custom validators
  - Allows to combine multiple validators

Custom Constraints

```
/* Annotation */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy=ValidUserValidator.class)

public @interface ValidUser {
 String message() default
       "{com.jboss.training.validuser}";
 Class<?>[] groups() default {};
 Class<? extends Payload>[] payload() default {};
}

/* Validator implementation */
public class ValidUserValidator implements
    ConstraintValidator<ValidUser,User> {
  @Override
  public void initialize(ValidUser
    constraintAnnotation) {}

  @Override
  public boolean isValid(User user,
    ConstraintValidatorContext context) {
  return !user.getName().equals("Sebastian");
  }
}
```

- Very simple to write
- Components
  - Annotation like @*ValidUser*
  - A validator class
  - The error message
    - Add the error message to a file named *ValidationMessages.properties*

Constraint Inheritance

- When validating an object that implements an interface or extends another class, all constraint annotations on the implemented interface and parent class apply in the same manner as the constraints specified on the validated object itself.
- Example:

```
public class RentalCar extends Car {
  private String rentalStation;
    public RentalCar(String manufacturer, String rentalStation) {
            super(manufacturer);
            this.rentalStation = rentalStation;
    }

    @NotNull
    public String getRentalStation() {
        return rentalStation;
    }

    public void setRentalStation(String rentalStation {
                          this.rentalStation = rentalStation;
    }
}
```

Validation - Walking the Entire Object Graph

- The `Bean Validation` API will allow validation upon an entire Object Graph:
  - Annotate a field or property representing a reference to another object with `@Valid`.
  - If the parent object is validated, all referenced objects annotated with `@Valid` will be validated as well (applied to their children also).

Object Graph Validation - Example

```
public class Person {
 @NotNull
 private String name;
 public Person(String name) {
   super();
   this.name = name;
 }
 public String getName() {
   return name;
 }
 public void setName(String name){
   this.name = name;
 }
}
```

```
public class Car {
 @NotNull
 @Valid
 private Person driver;

 public Car(Person driver) {
     this.driver = driver;
 }

 //getters and setters ...
}
```

Programmatic validation

- Introducing the Interface
  - The `Validator` interface is the main entry point to Bean Validation
    - Obtain a `Validator` Instance – via `Validation` class & `ValidatorFactory`
      - static `Validation.buildDefaultValidatorFactory( )`
    - Invoke the appropriate validator APIs
      - Validate Example:

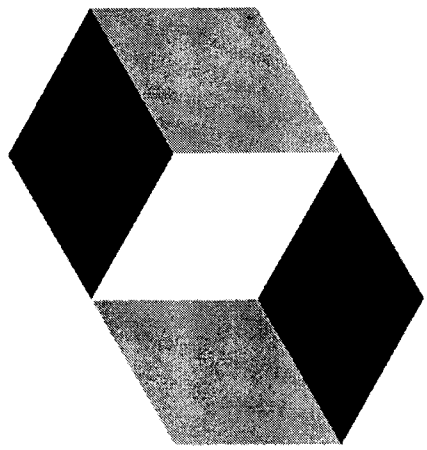
```
ValidatorFactory factory =
Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();

Car car = new Car(null);

Set<ConstraintViolation<Car>> constraintViolations =
validator.validate(car);

assertEquals(1, constraintViolations.size());
assertEquals("may not be null",
constraintViolations.iterator().next().getMessage());
```

Custom Validator Integration with Others

- Hibernate Validator
  - multi-layered data validation
  - But constraints are expressed in a single place (the annotated domain model)
  - Database schema-level validation
  - ORM integration
    - Hibernate
    - JPA
  - Presentation layer validation
- Extendible
  - Custom validations
  - Custom validation factories
- Flexible
  - Group validation

JBoss by Red Hat

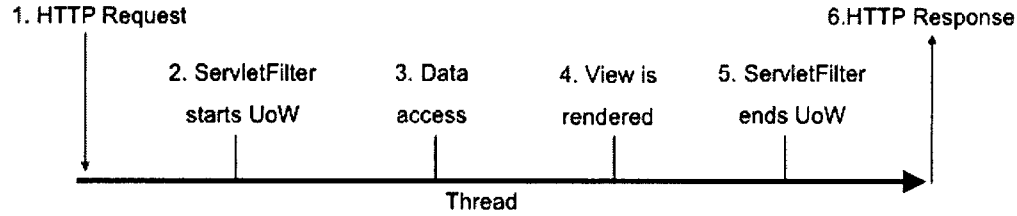
Roadmap



- **Application Design**
  - Layers
  - Use Case Examples
  - DAOs
  - DTOs
  - Conversations with Hibernate
  - Audit Logging and Hibernate Events
  - Data Validation
  - **Open-Session-In-View Pattern**

Understanding *Open-Session-In-View* Pattern

- Guidelines:
  - Why do we need a `ThreadLocal Session`?
    - The `Session` is closed when we forward to the (JSP) view
    - Un-initialized properties/collections throw an exception on access
  - What is `ThreadLocal Session`?
    - A `ThreadLocal` variable with thread scope holds the `Session` instance
    - Automatically provided by Hibernate with `getCurrentSession()`
- What is Open Session in View (OSIV)?
    - An interceptor opens and closes the unit of work automatically



Re-writing Our Use Case Codebase

- *Our `execute()` method can be cleaned up:*

```
public void execute() {
    // Get values from request
    try {
        Session session =
                 HUtil.getSessionFactory().getCurrentSession();
        // Load requested Item
        // Check auction still valid
        // Check amount of Bid
        // Add new Bid to Item
        // Place new Bid in scope for next page
        // Forward to success page
    }
    catch (Exception ex) {
        // Throw application specific exception
    }
}
```

Example

- *Note: An interceptor will start and end the unit of work - the `Session` and the `Transaction`.*

Implementing the *Interceptor* for OSIV

- Servlet filters are interceptors for HTTP request/response cycles:

```
public class HibernateFilter implements Filter {
    public void doFilter(ServletRequest request,
                         ServletResponse response,
                         FilterChain chain) {
        SessionFactory sf = HUtil.getSessionFactory();
        try {
          sf.getCurrentSession().beginTransaction();
          chain.doFilter(request, response);
          sf.getCurrentSession().getTransaction().commit();
        } catch (RuntimeException ex) {
          ... // handle exception
        }
    }
}
```

Example

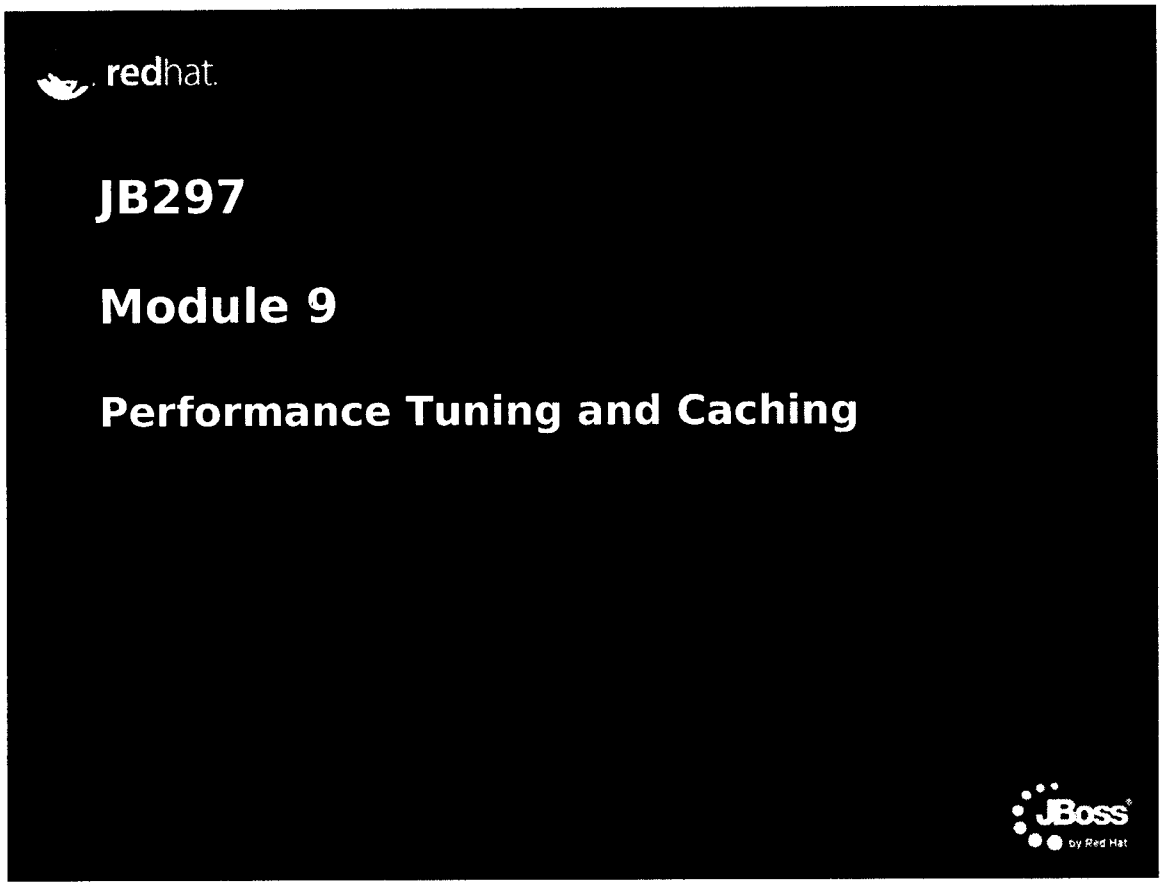
- *Note: Usually you want more sophisticated exception handling here, for `StaleObjectStateException` and versioning conflicts. You can write another filter "on top" that handles exceptions.*

Application Design

- Problems with Open Session In View:
  - Any change made to objects in a `Session`
    - Is flushed to the database at irregular intervals (e.g. before a query)
    - Is inside a transaction, committed when our view has been rendered
  - The buffer of the servlet engine may have already flushed parts of our page to the client browser.
    - The browser already got a `HTTP 200 OK`
    - While we still render and execute database statements (e.g. lazy fetching), an error occurs with one of our SQL statements
  - Solutions
    - Increase servlet buffer size (unsafe)
    - Flush the `Session` before forwarding to the view
    - Use a web framework with a separate output stream for rendering

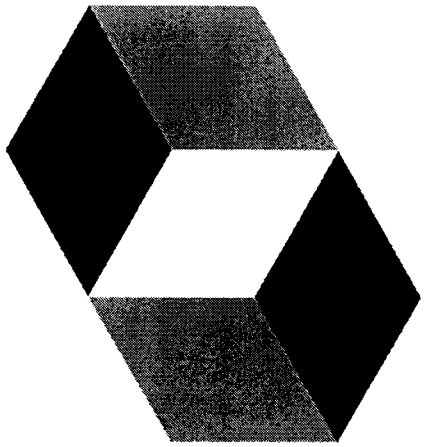
Summary

- In this lesson, you learned about:
  - Application design, the layers in an application and you looked at an example use case.
  - Various design approaches such as Data-Access-Objects, and Data-Transfer-Objects.
  - Conversation state approaches with Hibernate.
  - Implementing an Audit Log with Java Persistence Events
  - The Hibernate Event System and how to implement security checks.
  - Using and extending the validation framework
  - Open Session in View pattern



JBoss by Red Hat

Roadmap



Bulk Operations

Caching

Efficiently Querying Data

Lab 7

Hibernate Bulk Operations Approach

- Bulk operations:
  - HQL (and EJB 3.0) supports `UPDATE` and `DELETE` statements

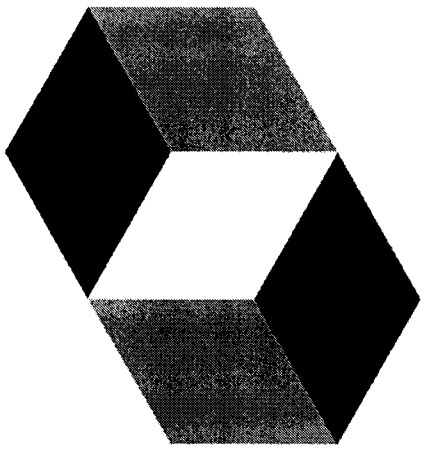
```
int updates = em.createQuery
  ("update User set loginAllowed = false").executeUpdate();
int deletes = em.createQuery("delete from Item").executeUpdate();
```

  - This works for all inheritance hierarchies (uses temporary tables internally) -
    - *Note: The `Session` is bypassed and not synchronized*
  - Recommended: execute bulk operations first thing in a new `Session`

```
int inserts = em.createQuery("insert into Customer (id, name)
          select u.id, u.login from User u").executeUpdate();
```

  - Bulk `INSERT` with `SELECT`

Roadmap



Bulk Operations

Caching

Efficiently Querying Data

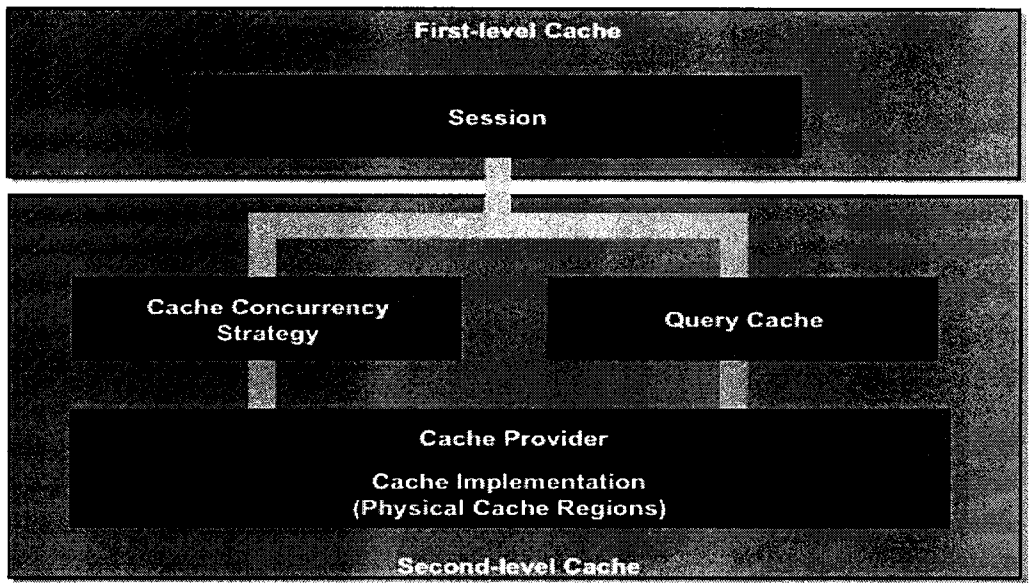
Lab 7

Caching Overview - Revisited

- Guidelines
  - The cache is used whenever
    - The application performs a lookup by identifier with `find()` or `getReference()`
    - Hibernate resolves an association lazily
    - Queries may also be cached
  - The cache type can be
    - Transaction scope cache (a single unit of work, i.e. `Session`)
    - Process scope cache (shared by many `Sessions`)
    - Cluster scope cache (shared by many processes)
  - The Hibernate caching system has several layers, i.e. a cache miss at the transaction level might be followed by a lookup at the process- or cluster level and only then produce a database hit.

The Hibernate Cache Architecture

- The Hibernate has a two-level cache architecture:



Persistence Context – First-Level Cache

- Persistence Context = first-level cache
  - Always enabled, can't be turned off (mandatory for transaction integrity).
  - *persist(), merge(), find()*, etc. all add objects to the cache.
  - *"I get an OutOfMemoryException when I load 500,000 objects?"*
    - Solutions
      - First, don't transfer the complete data into memory!
      - Don't use ORM, use a Stored Procedure or direct SQL
      - Use Hibernate3/EJB3 bulk operations
      - Use `scroll()` to load query results incrementally
      - `evict()` each object immediately from the `Session`
      - Or `clear()` the Session after each batch
    - In other words: you have to manage the first-level cache!

Managing Cache Interaction

- Controlling interaction:
  - Can be set in the persistence.xml, the EntityManager and per query or retrieval
  - Defines how the Session interacts with the second-level cache and query cache
  - javax.persistence.cache.retrieveMode
    - CacheRetrieveMode.USE – uses the cache (default)
    - CacheRetrieveMode.BYPASS – ignores the cache
  - javax.persistence.cache.storeMode
    - CacheStoreMode.USE – update cache on reading and writing from the database
    - CacheStoreMode.BYPASS – only invalidation of existing cache entries
    - CacheStoreMode.REFRESH – update cache even if entries do already exist

Managing Cache Interaction

```
final EntityManager em = emf.createEntityManager();
em.setProperty("javax.persistence.cache.storeMode",
    CacheRetrieveMode.BYPASS);
try {
    em.getTransaction().begin();
    for(int i = 0; i< 10;i++){
        Sender sender = new Sender("John");
        em.persist(sender);
        if(i % 20 == 0){
            em.flush();
            em.clear();
        }
    }
    em.getTransaction().commit();
} finally {
    em.close();
}
```

Second-Level Cache – A First Look

- The second-level cache
  - Process or cluster scope caching
    - Makes data visible in concurrent transactions.
    - This can cause problems if the ORM instance has non-exclusive access to the data.
    - It is expensive to ensure consistency of cache and database, and respect transaction isolation.
  - Non-exclusive data access
    - In clustered applications (use a cluster cache).
    - With shared legacy data (define cache expiry and transaction isolation).
  - Hibernate will not know when shared data has been updated, but you may implement a trigger notification system (difficult).

Ideal Cache Candidates

- Cache candidate guidelines
  - Especially good classes for caching represent
    - Data that is shared between many users
    - Data that changes rarely
    - Non-critical data (i.e. content-management data)
    - Data that is local to the application and not shared
  - Potentially bad classes for caching are
    - Data that is owned by a particular user
    - Data that is updated often
    - Financial data
    - Data that is shared with legacy applications
  - Reference data is an excellent candidate for caching
    - A small number of instances (< 1000)
    - Instances are rarely (or never) updated

Concurrency Strategy

- Hibernate cache concurrency strategies:
  - `transactional`
    - Available in managed environments (appserver), full isolation up to repeatable read. Use this strategy for read-mostly data where it is critical to prevent stale data in concurrent transactions, in the rare case of an update
  - `read-write`
    - Maintains read-committed isolation, only in non-cluster environments, uses a timestamp mechanism, use it for read-mostly data
  - `nonstrict-read-write`
    - Makes no guarantee of consistency, uses an expiry timeout
  - `read-only`
    - Useable for data that never changes, use it for reference data

Hibernate Providers - Caching

- Hibernate built-in cache providers
  - EHCache
    - Default, simple process cache for a single VM, supports query caching
  - OpenSymphony OSCache
    - Single process cache, rich set of expiration options and query caching
  - JBossCache
    - Fully transactional replicated cache, supports query caching.
    - Supports invalidation and synchronizing of clustered cache
  - Infinispan
    - Fully transactional replicated cache, supports query caching.
    - Supports invalidation and synchronizing of clustered cache
    - Maintains performance enhancements and maintenance APIs
  - *Note: Not every Cache Provider can use every Concurrency Strategy*

Cache Compatibility Matrix

Cache Provider	Read-Only	Non-strict Read-Write	Read-Write	Transactional
Infinispan	X			X
JBossCache	X			X
EHCache	X	X	X	
OSCache	X	X	X	

Cache Configuration for an Entity Class

- Cache configuration
  - Needs to be enabled in the *persistence.xml*
  - Defines the cache provider
  - Cache configuration is provider specific

```
<property name="hibernate.cache.provider_class"
  value="org.hibernate.cache.EhCacheProvider"/>
<property name="hibernate.cache.use_second_level_cache"
  value="true"/>
```

Cache Configuration for an Entity Class

- Cache configuration for `Category`
  - Different data requires different cache policies, so the second-level cache is granular
  - The `Category` is a good candidate for a read-write cache
    - Small number of instances
    - Updated rarely and shared by many units of work
    - A read committed isolation level is good enough
  - Caching all instances of `Category`, in mapping metadata

```
@Entity
@Cacheable
@org.hibernate.annotations.Cache(usage =
   CacheConcurrencyStrategy.READ_WRITE)
public class Category {
...
}
```

Caching Entities

- Caching associated entities
  - Hibernate extension
  - Cache the IDs of associated entities of `Category`:

```
@ManyToMany(mappedBy="categories")
@Cache(usage=CacheConcurrencyStrategy.READ_WRITE)
private Set<Item> items = new HashSet<Item>();
```

Annotations

  - *Note: The cache will now hold the identifier values of all associated instances, not the item entities.*

Caching Entity Associations Example

- Caching *Bids*
  - A *Bid* is placed on an item but is not mutable
    - We can use read-only caching

```
@Entity
@Immutable
@Cache(usage=CacheConcurrencyStrategy.READ_ONLY)
public class Bid {
    . . .
}
```

Managing the Second-Level Cache

- Guidelines
  - We can evict an object (or all objects of a class) from the cache

```
final Cache cache = emf.getCache();

// check if an object is in the cache
cache.contains(Sender.class, sender.getId());
// evict an object from the cache
cache.evict(Sender.class, sender.getId());
// evict all Sender.class objects from the cache
cache.evict(Sender.class);
// clear the cache
cache.evictAll();
```

  - Configuration of physical options (memory, timeouts, etc.) is done using the mechanisms of the cache provider. The binding between Hibernate and the cache provider are cache regions.

Understanding Cache Regions

- Guidelines
  - A cache can have multiple cache regions with different settings
  - Default regions
    - Hibernate tries to use default region names per cached class
    - Samples
      - Region of persistent class `Category`: `org.hibernate.auction.Category`
      - Region of cached association/collection items: `org.hibernate.auction.Category.items`
  - `EHCache`'s configuration file is `ehcache.xml`

```
<ehcache>
    ...
    <cache name="org.hibernate.auction.model.Category"
        maxElementsInMemory="500" eternal="true"
        timeToIdleSeconds="0" timeToLiveSeconds="0"
        overflowToDisk="false" />
</ehcache>
```

Hibernate – The Query Cache Mechanism

- Highlights:
  - The query cache is not enabled by default
    - Enable in persistence.xml

      ```
      <property name="hibernate.cache.use_query_cache"
      value="true"/>
      ```

    - Configure the default query and the update timestamps cache region
      - `org.hibernate.cache.StandardQueryCache`
      - `org.hibernate.cache.UpdateTimestampsCache`
  - Enable caching of a particular query

    ```
    Query query = em.createQuery("from
    Sender").setHint("org.hibernate.cacheable", true);
    ```

  - Optimization and cache management
    - Select a named cache region

      ```
      setHint("org.hibernate.cacheRegion", "myRegion");
      ```

    - Make sure the entities returned by the query are themselves cacheable.

Hibernate's Monitoring Abilities

- Monitoring
  - First, enable statistics generation for the `SessionFactory`
    - `hibernate.generate_statistics = true`
  - You have two options to access statistics:
    - Access the SessionFactory directly with `getStatistics()`
    - Register a `Statistics` MBean with the JMX server
  - *Let's have a look at some numbers...*

Class Statistics Provide Access...

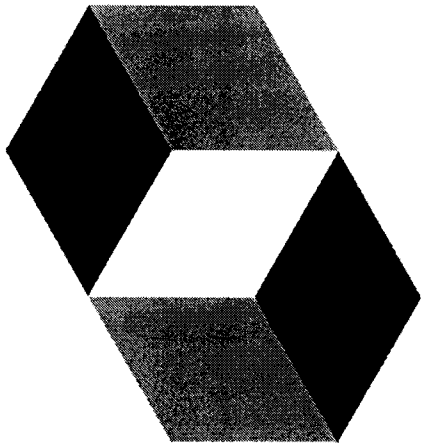
- Example Generated Output:

```
Statistics statistics =
SessionFactoryUtil.getInstance().getStatistics();
statistics.logSummary();
```

```
 INFO StatisticsImpl:437 - Logging statistics....
 INFO StatisticsImpl:438 - start time: 1112267039761
 INFO StatisticsImpl:439 - sessions opened: 3
 INFO StatisticsImpl:440 - sessions closed: 3
 INFO StatisticsImpl:441 - transactions: 6
 INFO StatisticsImpl:442 - successful transactions: 3
 INFO StatisticsImpl:443 - flushes: 3
 INFO StatisticsImpl:444 - connections obtained: 3
 INFO StatisticsImpl:445 - second level cache puts: 5
 INFO StatisticsImpl:446 - second level cache hits: 5
 INFO StatisticsImpl:448 - entities loaded: 6
 INFO StatisticsImpl:450 - entities inserted: 5
...
```

JBoss by Red Hat

Roadmap



Bulk Operations

Caching

Efficiently Querying Data

Lab 7

Batch Processing

- Batch inserts:
  - You have to *flush()* and *clear()* regularly

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
  for(int i = 0; i< 100000;i++){
    User user = new User("");
    em.persist(user);
    if(i % 20 == 0){
          em.flush();
          em.clear();
    }
  }
em.getTransaction().commit();
```

  - Don't forget to set a JDBC batch size appropriately
  - It's performing better if you disable the second-level cache

Batch Processing - Updates

- Batch updates
- Hibernate extension
  - Use database cursors to avoid memory exhaustion.

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Session session = em.unwrap(Session.class);
final ScrollableResults customers = session
    .createQuery("from Customer").setCacheMode(CacheMode.IGNORE)
    .scroll();

int count=0;
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    if ( ++count % 20 == 0 ) {
        session.flush();
        session.clear();
    }
}
em.getTransaction().commit();
em.close();
```

- *Note: You should disable the caches*

Caching and Tuning – Strategic Steps

- Step 1 - SQL / DB
  - Is a join operation faster than two selects?
  - Are all indexes used properly?
  - Get help from your DBA!
- Step 2 - Note the metrics for each use case
  - Used query type (HQL, Criteria, navigational access)
  - Number of SQL statements and complexity (joins, etc.)
  - Cache hit/miss ratio
- Step 3 - Optimize each use case
  - Tune metadata fetching strategy
  - Tune dynamic fetching (HQL and Criteria) and use reporting queries
  - Tune the second-level cache and query cache
- Apply step 2 and 3 to each use case, then repeat all to avoid side-effects of previous optimizations.
- Note: Never tune cache without benchmarks!

Retrieving Objects Efficiently

- Object retrieval options
  - We can get objects out of the database using
    - Retrieval by identifier (done that already)
    - HQL, a full object-oriented query language (SQL extension)
    - Criteria & Example, a type-safe way to build complex queries
    - Native SQL, with Hibernate result set mapping
    - Implicit, by navigating the object graph
  - All query options can use the Hibernate fetching strategies for best performance.
  - Our goal is to reduce the number and complexity (joins, table scans, etc.) of produced SQL queries.

Retrieving objects efficiently

- Each association and/or collection needs a fetching strategy
- Fetching strategies:
  - Select fetching
    - Second SELECT is executed to retrieve associated entity or collection
    - By default this will happen lazily, i.e. on first access.
  - Subselect fetching
    - Second SELECT is executed to retrieve all associated collections, re-executing the first query in a subselect
  - Batch fetching
    - Second SELECT is executed to retrieve all associated entities or collections, specifying a batch of key values
  - Join fetching
    - Hibernate retrieves the associated instance or collection in the same SELECT, using an OUTER JOIN.

Retrieving objects efficiently

- The n+1 selects problem
  - Let's use a criteria to load all items (1 query)

```
List<Item> items = session.createCriteria(Item.class).list();
```

  - The bids collection of each item has to be initialized (n queries)

```
for (Item item : items) {
  for (Bid bid : item.getBids()) {
    if (bid.getAmount() > 123.0) {
      // do something
    }
  }
}
```

Retrieving objects efficiently

- Enabling join fetching in mapping metadata
  - This affects all object loading using
    - Retrieval by identifier
    - Criteria queries
    - Implicit graph navigation

```
@OneToMany @JoinColumn
@Fetch(FetchMode.JOIN)
private Set<Bid> bids = new HashSet<Bid>();
```

  - Note: Join fetching should be set on only one collection, parallel OUTER JOINs create a Cartesian product, slower than select fetching

Retrieving objects efficiently

- Batch fetching
  - If one lazy collection or single-valued proxy has to be fetched, Hibernate loads up to 5 of them.
  - Batch Size should be reasonable

```
@OneToMany @JoinColumn
@BatchSize(size=5)
private Set<Bid> bids = new HashSet<Bid>();
```

  - Batch SQL query for collections (n/5+1) or proxies (n/9+1):

```
SELECT b.id, b.amount, b.item_id FROM bid b
    WHERE (b.item_id = 1) or (b.item_id = 2) or ...
SELECT c.id, c.name FROM category c
    WHERE c.id IN (1,2,3,4,5,6,7,8,9) ...
```

Retrieving objects efficiently

- Subselect fetching
  - If one lazy collection or single-valued proxy has to be fetched, Hibernate loads all of them, re-running the query that retrieved the original owners in a subselect.

```
@OneToMany @JoinColumn
@Fetch(FetchMode.SUBSELECT)
private Set<Bid> bids = new HashSet<Bid>();
```

  - Batch SQL query

```
SELECT b.id, b.amount, b.item_id FROM bid b
    WHERE b.item_id in (SELECT item_id FROM .... )
```

Retrieving objects efficiently

- Extra lazy collections
  - Hibernate will no longer initialize the (usually very large) collection when `size()` and `contains()` are called.

```
@OneToMany @JoinColumn
@LazyCollection(LazyCollectionOption.EXTRA)
private Set<Bid> bids = new HashSet<Bid>();
```

Retrieving objects efficiently

- Solving the n+1 selects problem
  - Best solution - keep everything lazy and use runtime fetching options.
    - Using Criteria

```
List results = session.createCriteria(Item.class)
                .setFetchMode("bids", FetchMode.JOIN).list();
```

    - Using HQL

```
List results = session.createQuery("select i from Item i left
                      join fetch i.bids").list();
```

  - Distinct results are possible

```
session.createCriteria(Item.class)
  .setFetchMode("bids",  FetchMode.JOIN)
  .setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY).list();
```

Lab - 7

Hibernate Queries & Fetch Strategy, Performance

- Details and instructions for the lab can be found in the lab guide.
- The instructor will answer any questions and will determine the stop time.

JBoss by Red Hat

Summary

- In this lesson, you learned about:
  - First-Level Cache vs. Second-Level Cache
  - Cache Provider Concurrency Strategy
  - Cache Monitoring & Measurement – Statistics APIs
  - Efficiently querying and loading data

JB297 - Hibernate
Copyright © 2011 – Red Hat Inc.

Lab 7: Cache and Performance Tuning

Requirements

- JB297-Hibernate Lab Archives
- Pre-Configured JBoss Developer Studio IDE
- Time: 30 Minutes

Goal

This lab demonstrates the use of entity relations and inheritance

Description

- In this lab you need to complete mappings and code. There are a number of tasks and each tasks can be validated running the Junit tests.

- The tasks are marked with TODO markers. Search each of the source files listed below for Tasks

  Example Task:
  /* TODO: A Task Title
  * Step 1: Code Something Here, As Instructed - Tags
  */

- You can find a detailed description below.
- Import JB297's **lab-7** project as an existing project, copied to the workspace
- The working/full source code is available in the lab-7/solution/ directory. View this directory for help, or backup the lab-7/src/ directory, rename the solution folder to src.

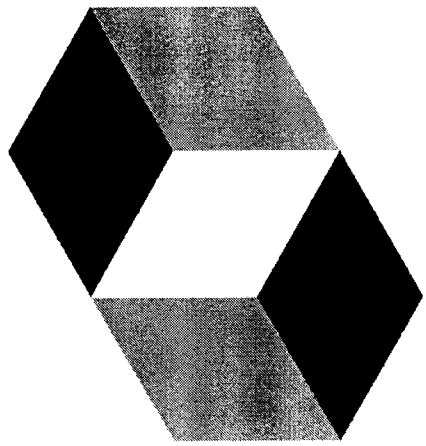
Project Instructions

1. We are printing a users name, sold items and the categories of the items. Currently this causes a lot of queries.
   i. Run the test *displayUserRepeatedly* and check in the console log how many queries are issued.
   ii. Try to reduce the queries using the second level cache. You need to add cache annotations to classes and two collections.
   iii. Run the test *displayUserRepeatedly* again to validate your changes.

2. We want to display all users, their sold items and the item's categories. This time your job is to optimize the mapping to load the data more efficiently.
   i. Run the test *displayUser* and check in the console log how many queries are issued.
   ii. Try to reduce the number of queries. You don't have to change the **code** in *displayUser* but the **mapping**.
   iii. Run the test *displayUser* again to validate your changes.

3. We are reusing the last use case, again displaying users, sold items and categories.
   i. Try to load all data with a single query.
   ii. Run the test *singleQuery* to validate your changes.

Roadmap



- **Hibernate Testing Framework**
  - **Framework within a Framework**
  - **Introducing TestNG**
  - **Testing the Persistence Layer**
  - **Introducing DBUnit**
    - **Writing a Test Class**
    - **Asserting DB Tables**
    - **Executing Integration Tests**

Hibernate Testing Framework...

- Software Testing Categories
  - Acceptance Testing
    - Not necessarily automated
    - Usually code by customer for project requirements verification
    - Can possibly include/use any metrics
  - Performance Testing
    - Stress / Load testing "exercise" the system with real world concurrency, etc
  - Unit Testing
    - Continuously & automatically tests the components' *functionality*

...more, next section

...Hibernate Testing Framework

- Unit Testing - Granularity
  - Logic Unit Testing
    - Independently tests components (*business logic*) for correct outcome
  - Integration Unit Testing
    - Tests *interaction* between components – services – and any subsystems
  - Functional Unit Testing
    - Exercise an *entire* use case – workflow – user interface for *correctness*

Introducing *TestNG* (New Generation)

- Testing framework inspired from JUnit and Nunit introduces some new functionalities that make it more powerful and easier to use:
  - JDK 5 Annotations (JDK 1.4 is also supported with JavaDoc annotations)
  - Flexible test configuration
  - Support for data-driven testing (with `@DataProvider`)
  - Support for parameters
  - Allows distribution of tests on slave machines
  - Powerful execution model (no more TestSuite)
  - Supported by a variety of tools and plug-ins (Eclipse, IDEA, Maven, etc...)
  - Embeds BeanShell for further flexibility
  - Default JDK functions for runtime and logging (no dependencies)
  - Dependent methods for application server testing

  - *Note: TestNG is designed to cover all categories of tests: unit, functional, end-to-end, integration, etc.*

TestNG - Examples

- Logical Unit Test via TestNG:

```
import org.testng.annotations.*;
public class AuctionLogictest {
  @Test(groups = { "logic" })
  public void aFastTest() {
  System.out.println("Fast test");

  /* A User & Item are required... /
  User usr = new User(. . .);
  Item auction = new Item(. . .);

  /*Place a Bid /
  BigDecimal bidAmt = new BigDecimal( "100.00");
  auction.placeBid(user, bidAmt, new BigDecimal(0), new BigDecimal(0));

  /*Place another higher bid*/
  BigDecimal bidAmt = new BigDecimal( "110.00");
  auction.placeBid(user, bidAmt, new BigDecimal(0), new BigDecimal(0));

  /* Assert State */
  assert item.getBids().size() == 2;
 }
}
```

Expecting Failures in a Test

- Expecting Failure:

```
public class AuctionLogic {
    @Test(groups =  "logic");
   @ExpectedExceptions(BusinessException.class)
      public void initialPriceConsidered( ) {

        /* A user considered... /
        User usrs = new Users(. . .);

        /* Create an Item instance with a high price... /
        Item auction = new Item( . . ., new BigDecimal("200.00");

        /* Place a Bid that id too low... */
        BigDecimal bidAmount = new BigDecimal("100.00");
        auction.placeBid(usr, bidAmount, new BigDecimal( 0),
                                          new BigDecimal( 0 ));
        }
}
```

Creating a Test Suite

- *Note: TestNG can detect all tests in a package (scanning for annotations)*
- Create a *testng.xml* test suite:

```
<DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name = "CaveatEmptor" verbose = "2">

<test name = "BusinessLogic">
<run> <include name = "logic.*" /> </run>
<packages> <package name = "auction.test" /> </packages>

<!-- Or simply the class...
<classes> <class name = "auction.test.AuctionLogic" /> </classes>
 - - >

</test>
</suite>
```

Running the TestNG Tests

- Either use one of the IDE plugins or add this to your Ant build process:

```
<taskdef resource = "testngtasks" classpathref =
                                                "project.classpath" />

<target name = "unittest.logic" depends = "compile, copymetafiles",
descripton = "Run logic unit tests with TestNG">

  <delete dir = "${basedir}/test-output" />
  <mkdir dir =  "${basedir}/test-output" />

  <testng outputDir = "${basedir} /test-output" classpathref =
                                                "project.classpath" >
    <xmlfileset dir = "${basedir}" > <include name =
                              "test-logic.xml" /> </xmlfileset>
  </testng>
</target>
```

Testing the Persistence Layer

- Testing Mappings
  - All columns & tables that are mapped match the properties & classes
- Testing Object State Transitions
  - Test if an object transitions correctly from transient – to persistent – to detached state (including *cascading*)
- Testing Queries
  - Any *non-trivial* HQL, *Criteria* , & possibly SQL query should be tested for correctness of the returned dataset
- *Note: These are all integration unit tests that involve the DBMS. Which one to use?*

Preparing the Infrastructure

- A DBMS is installed and ready...
  - An integration test sequence:
    - Reset the database to a well-known state (often empty)
    - Create any database for the unit test via imports – DBUnit is an ideal tool
    - Create objects – execute data access operations
    - Assert state after testing procedure (the actual test)

    more to come...

Introducing DBUnit -

- DBUnit provides support for the following:
  - Definition of base data within `DBUnit` *`Datasets`*
  - Execution of operations with datasets – e.g. `INSERT` or `DELETE` all base data
  - Clean-up of data

Note: For convenience – write a test superclass that executes DBUnit operations

DBUnit Test SuperClass

```
public abstract class HibernateIntegrationTest {
private ReplacementDataSet dataSet;

@Configuration(beforeTestClass = true)
public void prepare() throws Exception {
  dataSet = new ReplacementDataSet(
    new FlatXmlDataSet( getDataSetLocation() ));
  dataSet.addReplacementObject("[NULL]", null);
}

@Configuration(beforeTestMethod = true)
public void beforeTestMethod() throws Exception {
  executeOperations(getBeforeTestMethodStack());
}

@Configuration(afterTestMethod = true)
public void afterTestMethod() throws Exception {
  executeOperations(getAfterTestMethodStack());
}

private void executeOperations(DatabaseOperation[] operations)
throws Exception {
  IDatabaseConnection con = new DatabaseConnection( getConnection() );
  for (DatabaseOperation op : operations) op.execute(con, dataSet);
}
...
```

JBoss by Red Hat

DBUnit Test Superclass II

- Subclasses can/must override the following methods:

```
protected Connection getConnection() throws Exception {
  Connection con =
    ((SessionFactoryImpl)sessionFactory).getSettings()
    .getConnectionProvider().getConnection();

  // Disable foreign key constraint checking
  // This really depends on the DBMS product... here for HSQL DB
  con.prepareStatement("set referential_integrity FALSE").execute();
  return con;
}

protected DatabaseOperation[] getBeforeTestMethodStack() {
  return new DatabaseOperation[]{};
}

protected DatabaseOperation[] getAfterTestMethodStack() {
  return new DatabaseOperation[]{};
}

protected abstract String getDataSetLocation();
```

Preparing the Dataset

- An Empty row for a Table (`ITEM`) is useful, DBUnit can clean it up

```
<dataset>

        <USERS USER_ID ="1"
                        OBJ_VERSION ="0"
                        FIRSTNAME ="Root"
                        LASTNAME ="Toor"
                        USERNAME ="root"
                        PASSWORD ="secret"
                        EMAIL ="root@toor.tld"
                        RANK ="0"
                        IS_ADMIN ="true"
                        CREATED ="2009-11-23 23:45:00"
                        HOME_STREET ="[NULL]"
                        HOME_ZIPCODE ="[NULL]"
                        HOME_CITY ="[NULL]"
                        DEFAULT_BILLINGDETAILS_ID ="[NULL]"
                />
        <ITEM/>
</dataset>
```

Writing a Test Class

- A Test Class groups tests that rely on a particular dataset

```
public class PersistentStateTransitions extends
HibernateIntegrationTest {

        protected String getDataSetLocation() {

        return "ce/modules/auction/test/basedata.xml";

        }

        protected DatabaseOperation[] getBeforeTestMethodStack() {
                return new DatabaseOperation[]
{ DatabaseOperation.CLEAN_INSERT };
        }
...
}
```

- *Note: Override `getBeforeTestMethodStack()` and `getAfterTestMethodStack()` to customize DBUnit operations for this class*

Writing a Test Procedure

```
@Test(groups = "integration.hibernate")
public void storeAndLoadItem() {

// Start a unit of work
sessionFactory.getCurrentSession().beginTransaction();

... // Prepare the DAOs

// Prepare a user object
User user = userDAO.findById(1l, false);

// Make a new auction item persistent
Calendar startDate = GregorianCalendar.getInstance();
Calendar endDate = GregorianCalendar.getInstance();
endDate.add(Calendar.DAY_OF_YEAR, 3);
Item newItem = new Item( "Testitem", "Test Description", user, new
BigDecimal(123), new BigDecimal(333), startDate.getTime(),
endDate.getTime() );
itemDAO.makePersistent(newItem);

// End the unit of work
sessionFactory.getCurrentSession()
.getTransaction().commit();
...
```

Asserting Database State

- Use plain JDBC or Hibernate's *StatelessSession* Interface for direct database access

```
// Direct SQL query for database state in auto-commit mode
StatelessSession s = sessionFactory.openStatelessSession();

Object[] result = (Object[]) s.createSQLQuery(
         "select INITIAL_PRICE ip," + " SELLER_ID sid from ITEM")
         .addScalar("ip", Hibernate.BIG_DECIMAL).addScalar("sid",
         Hibernate.LONG).uniqueResult();

s.close();

// Assert correctness of state
assert result[0].getClass() == BigDecimal.class;
assert result[0].equals( newItem.getInitialPrice().getValue() );
assert result[1].equals( 11 );
```

Running the Integration Tests

- Note: A suite can be configured with test group names and wildcards.

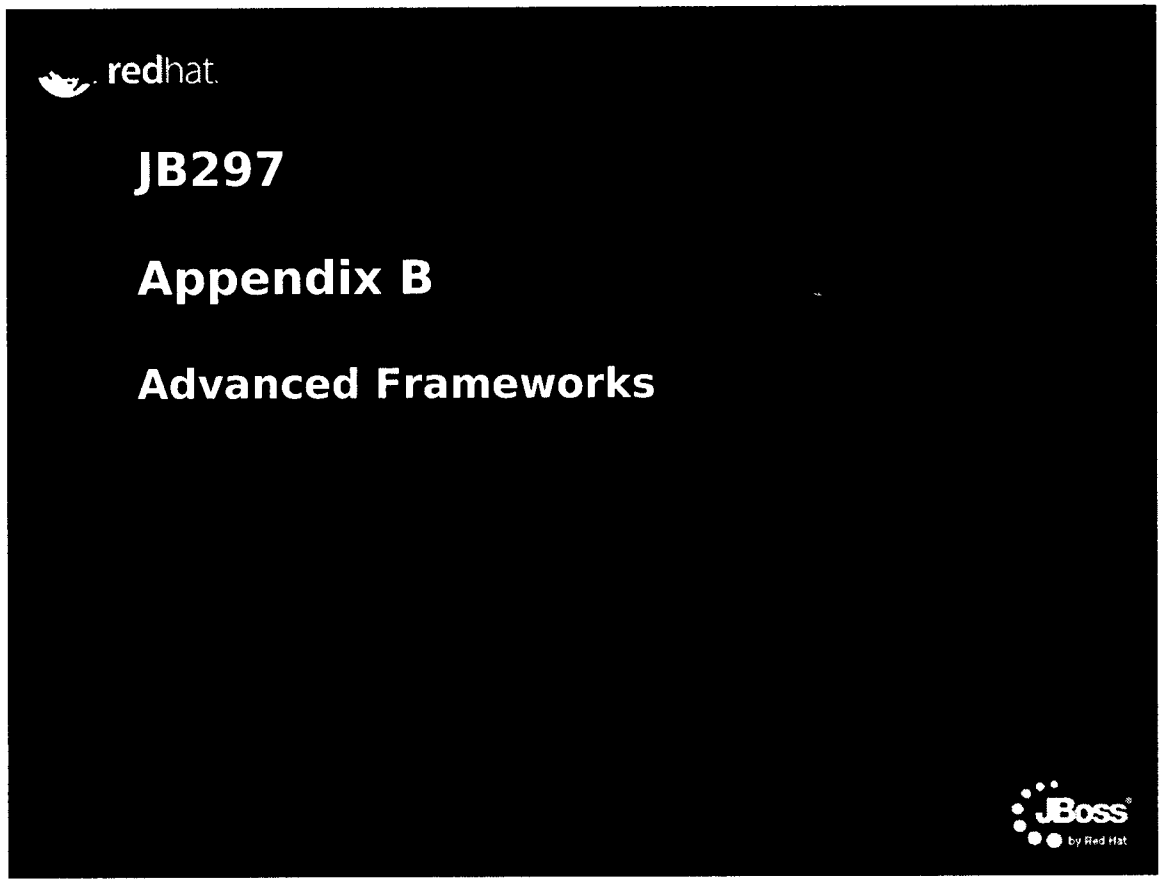
```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite name="CaveatEmptor" verbose="2">
<test name="PersistenceLayer">

<groups>
        <run><include name="integration.hibernate.*"/></run>
</groups>

<packages>
        <package name="auction.test.dbunit"/>
</packages>

</test>
</suite>
```

A Note about Performance Testing

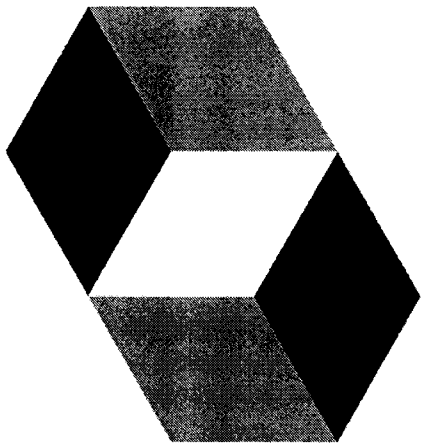
- Test scalability with:
  - real-world data sets, use a test data generator if you don't have (enough) test data.
  - real use cases, pick the most prevalent and important cases.
  - concurrency and read/write access of multiple users.
- Don't test with microbenchmarks - Hibernate is slower than JDBC if you only load and store 50,000 objects.

Summary

- In this lesson, you learned about:
  - The Hibernate Testing Framework, which includes TestNG and DBUnit
  - The Hibernate Advanced Frameworks, which include the Hibernate Search Appliance, Hibernate Validator Framework and Hibernate Shards.



Roadmap



- **Hibernate Advanced Frameworks**
  - **Hibernate Search**
  - Hibernate Envers – Data Historization

Hibernate Search Overview

- Googling The Persistent Domain Model
  - SQL Search vs. Full-text Search
  - Object model/Full-text Search Mismatched
  - Hibernate Search Architecture
  - Configuration / Mapping
  - Working towards Java Persistence (JPA 2.0) Integration

SQL Search Limitations...

- Query by word
  - "`*hibernate*`" a/k/a % inside SQL
- Query by Approximation (or Synonym)
  - "`hybernat`"
- Proximity Search
  - "Java" close to "Persistence"
- Relevance – Result Scoring
- Multiple - "column" Search

Full Text Search

- Search Information
  - By Word
  - Inverted Indices (word frequency, position)
- In RDBMS Engines
- Portability (proprietary add-on on top of SQL)
- Flexibility
- Standalone Engine
- Apache Lucene - http://lucene.apache.org

Mismatches within the Domain Model

- Structural Mismatch
- Full text index are text only
- No reference between documents
- Synchronization mismatch
- Keeping index and database up to date
- Retrieval Mismatch
- The *index* does not store objects
- Certainly not managed objects

JBoss by Red Hat

Hibernate Search Illustration – Google the Apps

- Guidelines
- Belongs to the Hibernate Platform
- Follows LGPL
- Uses Apache Lucene "under the hood"
- Within the "Top 10 Downloaded" products at Apache
- Very powerful – rich API feature set exposes low-level API/fine-granularity
- Easy to use incorrectly
- Hibernate Search – Solves the Domain Model Paradigm Mismatch

Hibernate Search Architecture...

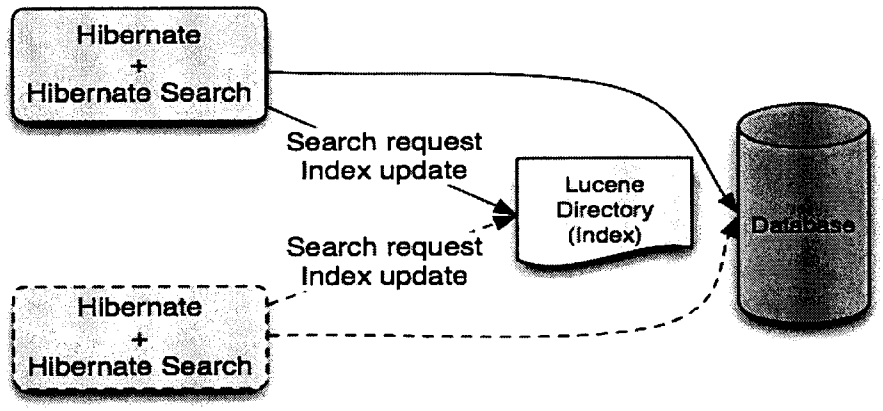
- Architecture Highlights
  - Indexing triggered by Java Persistence API Event System
  - Utilized `PERSIST` / `UPDATE` / `DELETE` Functionality
  - Converts the object structure into "Index" structure
  - Provides Operation Batching "per transaction"

- Adheres to ACID principles
  - Provides enhanced Lucene performance
  - Provides a "pluggable" scope

- Back end
  - Synchronous/Asynchronous Modes
  - Lucene – Java Messaging Service (JMS)

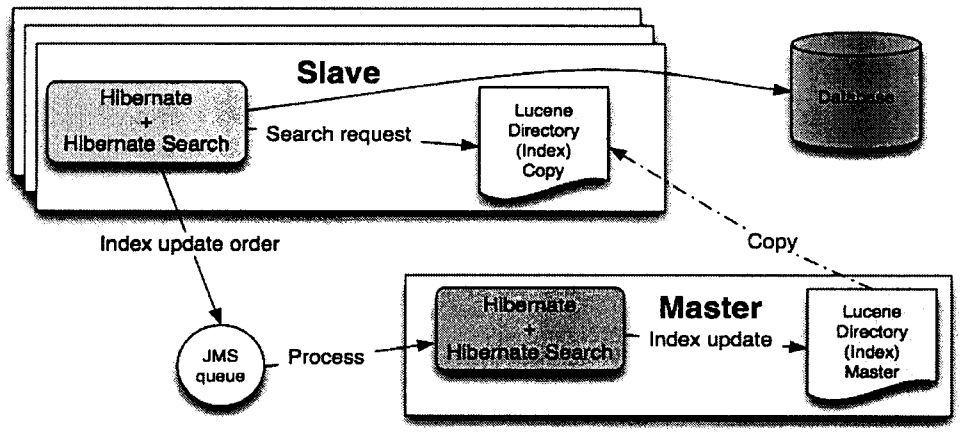
...Hibernate Search Architecture...

- Hibernate Search Backend – Lucene Directory
  - Standalone or Symmetric Cluster
  - Immediate index change visibility
  - Can impact front-end runtime



...Hibernate Search Architecture

- Search Backend Architecture – (cont)
  - JMS Technology (Cluster)
    - Search Processed Locally
    - Changes sent to a "master node"(JMS Technology)
    - Asynchronous Indexing (Delay)
    - No front-end extra cost (LOC)



Search – Configuration and Mapping

- Guidelines
  - Configuration
    - Event Listener wiring (transparent in Hibernate Annotations)
    - Backend Configuration

- Annotation-based
  - `@Indexed`
  - `@Field(store, index)`
  - `@IndexedEmbedded`
  - `@FieldBridge`

Hibernate Search Mapping - Example

- Entity Mapping Example:

```
@Entity @Indexed(index="indexes/essays")
public class Essay {
…

@Id @DocumentId
public Long getId() { return id; }

@Field(name="Abstract", index=Index.TOKENIZED,
store=Store.YES)
public String getSummary() { return summary; }

@Lob @Field(index=Index.TOKENIZED)
public String getText() { return text; }

@ManyToOne @IndexedEmbedded
public Author getAuthor() { return author; }
```

Hibernate Search – Understanding the Query

Search Query Guidelines

- Retrieve object – not documents
  - No boilerplate conversion code
- Objects from the Persistence Context (managed)
  - Same semantics as JPQL or Criteria API Query
- Uses `org.hibernate.Query` APIs
  - Common API for all your queries
  - Pagination support
  - List/Scroll/Iterate Support
- Query upon correlated objects
  - Provides "JOIN"-like query API

Search Query - Example

- Search Query Example

```
. . .
org.apache.lucene.search.Query luceneQuery;
String queryString = "summary:Festina Or brand:Seiko"
luceneQuery = parser.parse( queryString );

org.hibernate.Query fullTextQuery =
fullTextSession.createFullTextQuery( luceneQuery );

fullTextQuery.setMaxResult(200);

List result = fullTextQuery.list();
//return a list of managed objects

queryString = "title:hybernat~" //Approximate search
queryString = "\"Hibernate JBoss\"~10" //Proximity search
queryString = "author.address.city:Atlanta"
//correlated search
 . . .
```

Java Persistence API Integration - Overview

- Existing Integration Points
  - Java Persistence API entity listener (index update)
  - Transaction synchronization in JTA (index update)
  - JMS API (Clustering)

- Problems
  - Transaction Synchronization within JDBC
  - Initialization Lifecycle
    - No one exists currently within JPA
  - Standard Lazy state detection
    - To avoid load triggered by Search engine
  - Standard Lazy Object Initialization
    - To force "lazy loading" load within the Search engine

Hassle-Free Full-text Search

- Transparent Index Synchronization
- Automatic structural conversion via mapping
- No *paradigm-shift* when retrieving data
- High-availability / Clustering capability "out-of-the-box"

Hassle-Free Full-text Search

- For More Resource Information
  - Hibernate Search
    - http://search.hibernate.org
  - JBoss Seam Framework
    - http://www.jboss.com/products/seam
  - Apache Lucene
    - http://lucene.apache.org
  - Lucene In Action (book)
  - Java Persistence with Hibernate API (book)

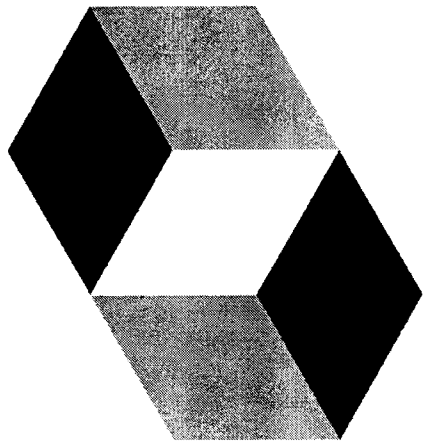
Hibernate Search Resources

For More Information

- Hibernate Search
  - http://search.hibernate.org
- JBoss Seam
  - http://www.jboss.com/products/seam
- Apache Lucene
  - http://lucene.apache.org
- Lucene In Action
  - Java Persistence API with Hibernate

JBoss by Red Hat

Roadmap



- **Hibernate Advanced Frameworks**
  - Hibernate Search
  - **Hibernate Envers – Data Historization**

Hibernate Envers

- Stores old revisions of entities and allows to query them
  - Allows to answer: What did the article entity with id 123 look like 3 months ago?

- Auditing of entities
  - Old revisions of entities are stored in separate tables
  - Simple integration with Hibernate and Java Persistence
  - Querying of historical data
    - When did an entity change?
    - Which revision was valid at a specific date?
    - Which revisions are available for a given entity?

- Integration requires only two steps
  - Configuration of event listener
  - Add annotations to entities to specific which entity is audited

Hibernate Envers

- Add event listener to persistence.xml

```
<property name="hibernate.ejb.event.post-insert"
value="org.hibernate.ejb.event.EJB3PostInsertEventListener,org.hibernate.envers.event.AuditEventListener"/>
<property name="hibernate.ejb.event.post-update"
value="org.hibernate.ejb.event.EJB3PostUpdateEventListener,org.hibernate.envers.event.AuditEventListener"/>
<property name="hibernate.ejb.event.post-delete"
value="org.hibernate.ejb.event.EJB3PostDeleteEventListener,org.hibernate.envers.event.AuditEventListener"/>
<property name="hibernate.ejb.event.pre-collection-update"
value="org.hibernate.envers.event.AuditEventListener"/>
<property name="hibernate.ejb.event.pre-collection-remove"
value="org.hibernate.envers.event.AuditEventListener"/>
<property name="hibernate.ejb.event.post-collection-recreate"
value="org.hibernate.envers.event.AuditEventListener"/>
```

Hibernate Envers

```
@Entity
@Audited
public class Sender {
  @Id @GeneratedValue
  private Long id;
  private String name;
...
}
```

- Configuring audited entities
  - Mark entities with *@Audited*
  - Relations can be audited as well
  - There are limitations
    - Read the documentation

Hibernate Envers

- Querying audited data
  - You can query revision details

```
final AuditReader auditReader = AuditReaderFactory.get(emf.createEntityManager());
final List<Object[]> resultList = auditReader.createQuery()
    .forRevisionsOfEntity(Message.class, false, true).getResultList();
for (Object o[] : resultList) {
  Message m = o[0];
  DefaultRevisionEntity entity = (DefaultRevisionEntity) o[1];
  final RevisionType typeOfModification = (RevisionType) o[2];
  if (typeOfModification == RevisionType.ADD)
    System.out.println(String.format("Message added in revision %s was added at %s. Content: %s",
      new Object[]{entity.getId(), entity.getRevisionDate(), m}));
  else if (typeOfModification == RevisionType.MOD)
    System.out.println(String.format("Message modified in revision %s at %s. Content: %s",
      new Object[]{entity.getId(), entity.getRevisionDate(), m}));
  else
    System.out.println(String.format("Message deleted in revision %s at %s. ",
new Object[]{entity.getId(), entity.getRevisionDate()}));
}
```

Hibernate Envers

```
List<Message> messages =
  auditReader.createQuery()
  .forRevisionsOfEntity(Message.class, true, true)
  .add(AuditEntity.revisionNumber()
  .between(159,162)).getResultList();


Message m = (Message) auditReader.createQuery()
   .forEntitiesAtRevision(Message.class, 159)
   .add(AuditEntity.id().eq(157L))
   .getSingleResult();


final Calendar tenMinutesAgo = Calendar.getInstance();

tenMinutesAgo.add(Calendar.MINUTE, -10);

Message m = (Message) auditReader.createQuery()
  .forRevisionsOfEntity(Message.class, true, true)

  .add(AuditEntity.id().eq(157L)).add(

     AuditEntity.revisionProperty("timestamp")
     .lt(tenMinutesAgo.getTime().getTime()))

  .addOrder(AuditEntity.revisionNumber().desc())
  .setMaxResults(1).getSingleResult();
```

- Querying audited data
  - Query looks like Hibernate Criteria query
  - Entity and revision information can be selected
  - Sample queries
    - Messages between revision 159 and 162
    - Message with id 157 at revision 159
    - Message with id 157, ten minutes ago

Hibernate Envers

- Customization
  - Revision entity
    - Default entity has a revision number and a timestamp
    - You might add additional fields like last change user
  - RevisionListener
    - Is executed when a revision is created
    - Allows to modify the revision entity
      - For example: set the currently logged in user

Summary

- In this lesson, you learned about:
  - Hibernate Search integration with Lucene
  - Hibernate Envers Auditing

redhat.

(11) 3529-6000
http:// www.br.redhat.com/training