Hibernate Basics

"Code with Passion!"

Topics

- Why use Object/Relational Mapping(ORM)?
- What is and Why Hibernate?
- Hibernate architecture
- Domain classes
- Instance states
- Persistence life-cycle operations
- Hibernate framework classes
- Hibernate configuration
- Object to table mapping
- DAO pattern

Topics covered in other presentations

- Hibernate Criteria API
- Hibernate HQL
- Hibernate Mapping
- Hibernate Fetch modes, N+1 select problem
- Hibernate caching
- Hibernate transaction & locking

Why Use ORM?

Two different ways of Configuring Hibernate

- Programmatic configuration
- XML configuration file
  - Specify a full configuration in a file named *hibernate.cfg.xml*
  - By default, is expected to be in the root of your classpath

Why Object/Relational Mapping (ORM)?

- ORM handles Object-Relational impedance mismatch
  - Relational database is table driven (with rows and columns) because it is designed for fast query operation of table-driven data
  - We, Java developers, want to work with classes/objects, not rows and columns, however
  - ORM handles the mapping between the two
- If you are not using ORM, you will work directly with JDBC layer, in which you will have to be responsible for this mapping
- ORM solutions are mature

What is & Why Hibernate?

What is Hibernate?

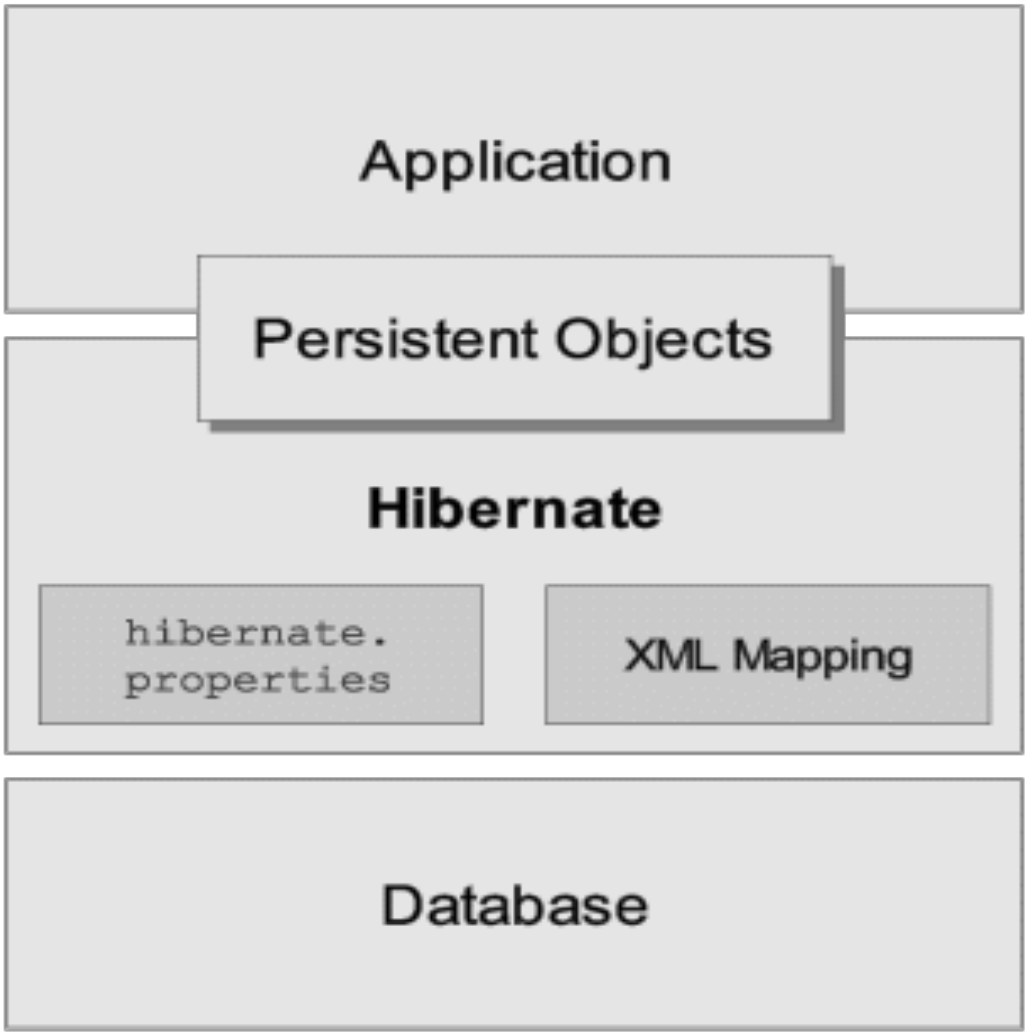
- It is one of the most popular ORM frameworks
  - Let you work without being constrained by table-driven relational database model – handles Object-Relational impedance mismatch
  - No need to work with low-level JDBC layer
- Enables transparent POJO persistence
  - Lets you build persistent objects following common OO programming concepts: Inheritance, polymorphism

Why use Hibernate as a ORM Solution?

- Performance
  - High performance object caching
  - Configurable materialization strategies
- Sophisticated query facilities
  - Criteria API
  - Query By Example (QBE)
  - Hibernate Query Language (HQL)
  - Native SQL
- Proven in the market place

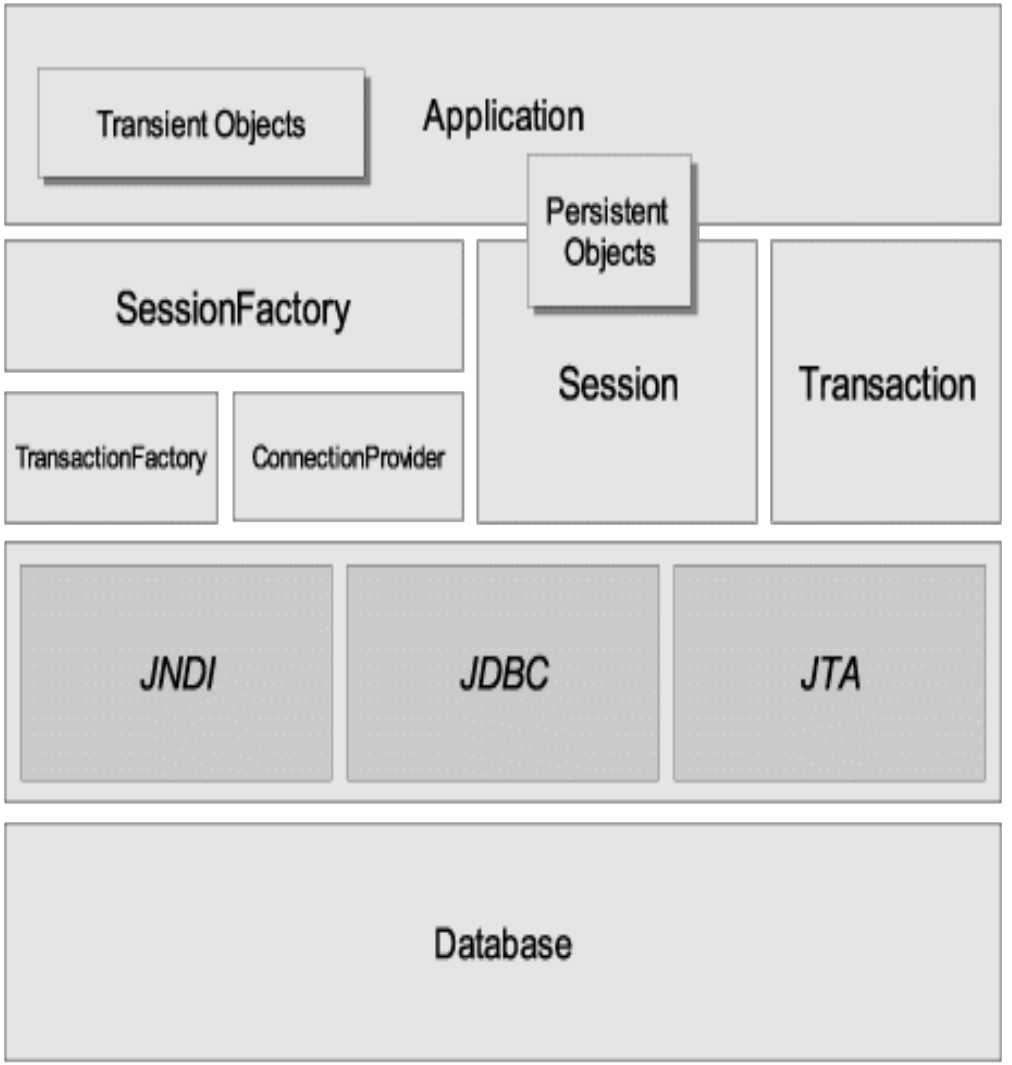
Hibernate Architecture

Hibernate Architecture



Hibernate Architecture

- The architecture abstracts the application away from the underlying JDBC/JTA APIs and lets Hibernate take care of the plumbing details.



Domain Classes

Domain Classes

- Domain classes are classes in an application that implement the entities of the business domain (e.g. Customer and Order in an E-commerce application)

- Hibernate works best if these classes follow some simple rules, also known as the Plain Old Java Object (POJO) programming model.

Steps to write a Domain Class

- Step 1: Implement a no-argument constructor
  - All persistent classes must have a default constructor so that Hibernate can instantiate them
- Step 2: Provide an identifier property
  - This property maps to the primary key column of a database table.
  - The property can be called anything, and its type can be any primitive type, any primitive "wrapper" type, *java.lang.String* or *java.util.Date*
  - Composite key is possible
- Step 3: Declare getter/setter methods for persistent fields

Instance States (of Domain Object)

Instance States

- An instance of a domain class (domain object) may be in one of three different states, which are defined with respect to a persistence context
  - transient (does not belong to a persistence context)
  - persistent (belongs to a persistence context)
  - detached (used to belong to a persistence context)
- The persistence context is represented by Hibernate Session object
  - In JPA, the persistence context is represented by *EntityManager*, which plays same role of Session in Hibernate

“transient” state

- The instance is not, and has never been associated with any session (persistence context)
- It has no persistent identity (primary key value) called “Object Identifier” yet
- It has no corresponding row in the database
- ex) When POJO instance is created outside of a session meaning before it is persisted, it is in “transient” state
- Changes made to transient objects do not get reflected to the database table - They need to be persisted before the change get reflected to the database table (when committed or flushed)

“persistent” state

- The instance is currently associated with a single session (persistence context)
- It has a persistent identity (primary key value) called Object Identifier and likely to have a corresponding row in the database (if it has been committed before or read from the table)
- Changes made to persistent objects (objects in “persistent” state) are reflected to the database tables when they are committed or flushed
- ex) When a transient object gets persisted through save(..) method

Transient state and Persistent state

```
Person person = new Person();  // transient state
person.setName("Sang Shin");
session.save(person);              // persistent state

// You can get an object identifier via getIdentifiler() method
Object identifier = session.getIdentifier(person);
```

“detached” state

- The instance was once associated with a persistence context, but that context was closed, or the instance was serialized to another process
- It still has a persistent identity (Object identifier) and, perhaps, a corresponding row in the database
- Used when POJO object instance needs to be sent over to another program for manipulation without having persistent context
- Changes made to detached objects do not get reflected to the database table - They need to be “merged” before the change get reflected to the database table when they are committed or flushed

Difference between “transient” and “detached” Object Instances

- “transient” instance does not have Object identifier while “detached” instance has Object identifier

State Transitions

- Transient instances may be made persistent by calling *save()*, *persist()* or *saveOrUpdate()*
- Persistent instances may be made transient by calling *delete()*
- Any instance returned by a *get()* or *load()* method is persistent
- Detached instances may be made persistent by calling *update()*, *saveOrUpdate()*, *lock()* or *replicate()*
- The state of a detached instance may also be made persistent as a new persistent instance by calling *merge()*
- Persistent objects in a session becomes detached objects after the session it belongs to closes

Methods of "Session" Interface

Types of Methods in “Session” Interface

- Life cycle operations
- Transaction and Locking
- Managing resources
- JDBC Connection

Lifecycle Operations

Lifecycle Operations

- *Session* interface provides methods for lifecycle operations
  - Saving objects
  - Loading objects
  - Getting objects
  - Refreshing objects
  - Updating objects
  - Deleting objects
  - Replicating objects
- Result of lifecycle operations affect the instance state

Saving Objects

- An object remains to be in “transient” state until it is saved
- The object state changed “persistent” state
  - Once in “persistent state”, an object is managed by a particular session (Sometimes it is said the object belongs to a particular persistence context)

Java methods for saving objects

- From Session interface

```
// Persist the given transient instance.
// Hibernate assigns a new Object identifier
// and returns it as a return value.
public Serializable save(Object object)
```

Example: Saving Objects

```
Person person = new Person();  // object is in transient state now
person.setName("Sang Shin");
session.save(person);          // object is in persistent state

// You can get an object identifier via getIdentifiler() method
Object identifier = session.getIdentifier(person);
```

Loading Objects

- Used for loading objects from the database
- Each *load(..)* method requires object's primary key as an identifier
  - The identifier must be *Serializable* – any primitive identifier must be converted to object
- Each *load(..)* method also requires which domain class or entity name to use to find the object with the id
- The returned object, which is returned as *Object* type, needs to be type-casted to a domain class
- The returned object is in “persistent” state

Java methods for loading objects

- From Session interface

```
// Return the persistent instance of the given entity
// class with the given identifier, assuming that the
// instance exists.
public Object load(Class theClass, Serializable id)
```

Getting Objects

- Works like load() method

load() vs. get()

- Only use the *load()* method if you are sure that the object exists
  - *load()* method will throw an exception if the unique id is not found in the database
- If you are not sure that the object exists, then use one of the *get()* methods
  - *get()* method will return null if the unique id is not found in the database

Java methods for getting objects

- From Session interface

```
// Return the persistent instance of the given entity
// class with the given identifier, or null if there is no
// such persistent instance.
public Object get(Class theClass, Serializable id)
```

Example: Getting Objects

```
Person person = (Person) session.get(Person.class, id);
if (person == null){
    System.out.println("Person is not found for id " + id);
}
```

Refreshing Objects

- Used to refresh objects from their database representations in cases where there is a possibility of persistent object is not in sync. with the database representation

- Scenarios you might want to do this
  - Your Hibernate application is not the only application working with this data
  - Your application executes some SQL directly against the database (not using Hibernate)
  - Your database uses triggers to populate properties on the object

Java methods for Refreshing objects

- From Session interface

// Re-read the state of the given instance from the

// underlying database.

public void refresh(Object object)

Updating Objects

- Hibernate (Session) automatically manages any changes made to the persistent objects
  - The objects must be in “persistent” state not “transient” state in order the changes to be managed
- If a property changes on a “persistent” object, Hibernate session will perform the change in the database when a transaction is committed (possibly by queuing the changes first)
- You can force Hibernate to commit all changes using *flush()* method
- You can also determine if the session is dirty through *isDirty()* method

Merging Objects

- Copy the state of the given object onto the persistent object with the same identifier
- If there is no persistent instance currently associated with the session, it will be loaded
- Return the persistent instance. If the given instance is in “transient” state, it creates a copy of it and then returns it a newly persistent instance.
  - The given instance does not become associated with the session.

Example: Merging Objects

// p1 is “detached” object and remains to be in “detached” state.

// p1_merged is “persistent” object

p1_merged = (Person) session.merge(p1);

Lab:

Exercise 1: Session Life- cycle operations
3514_hibernate_basics.zip

Hibernate Framework Classes

Hibernate Framework Classes

- *org.hibernate.SessionFactory*
- *org.hibernate.Session*
- *org.hibernate.Transaction*

(We will cover Hibernate annotation in another session)

org.hibernate.Session

- A single-threaded, short-lived object representing a conversation between the application and the persistent store
- A session represents a persistence context
  - A persistence context is a container of “persistent” object instances
  - A “persistent” object is always associated with a session (in other terms, belongs to a particular persistent context) and always has an Object Identifier
- Handles life-cycle operations - create, read and delete operations - of persistent objects
- Factory for *Transaction*

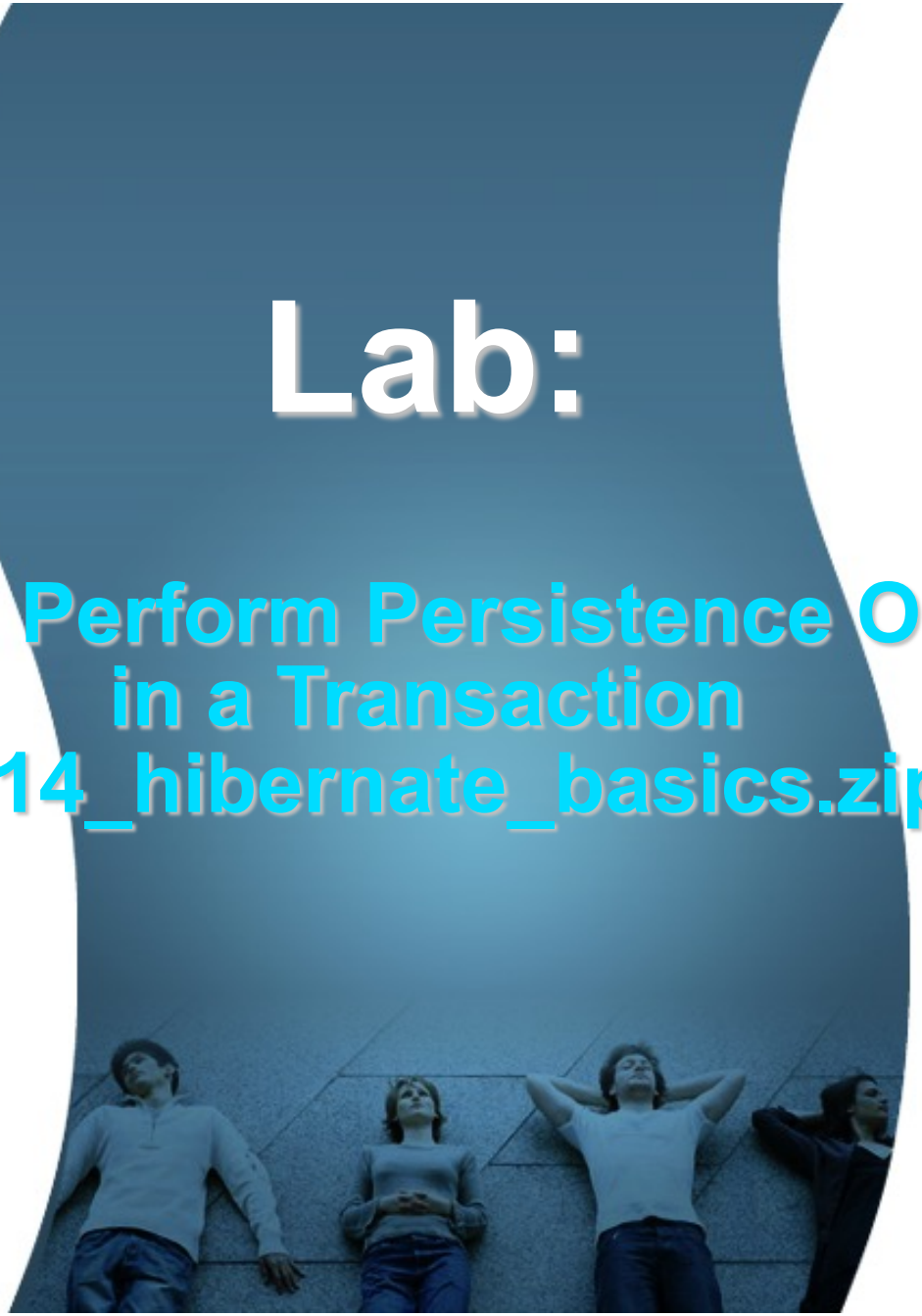
org.hibernate.Transaction

- A single-threaded, short-lived object used by the application to specify atomic unit of work
- Abstracts application from underlying JDBC, JTA or CORBA transaction.
- However, transaction demarcation, either using the underlying API or Transaction, is never optional! In other words, any operations with the database is always under a transaction.

(We will cover Transaction in detail in Hibernate Transaction)

Lab:

Exercise 2: Perform Persistence Operations in a Transaction
3514_hibernate_basics.zip



Hibernate Configuration

Hibernate Configuration File

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
   "-//Hibernate/Hibernate Configuration DTD//EN"
   "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

   <!-- a SessionFactory instance listed as /jndi/name -->
   <session-factory
      name="java:hibernate/SessionFactory">

      <!-- properties -->
      <property
         name="connection.datasource">java:/comp/env/jdbc/MyDB</property>
      <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
      <property name="show_sql">false</property>
      <property name="transaction.factory_class">
         org.hibernate.transaction.JTATransactionFactory
      </property>
      <property name="jta.UserTransaction">
         java:comp/UserTransaction
      </property>
```

Hibernate Configuration File

```
    <!-- mapping files -->
    <mapping resource="org/hibernate/auction/Item.hbm.xml"/>
    <mapping resource="org/hibernate/auction/Bid.hbm.xml"/>

    <!-- cache settings -->
    <class-cache class="org.hibernate.auction.Item" usage="read-write"/>
    <class-cache class="org.hibernate.auction.Bid" usage="read-only"/>
    <collection-cache collection="org.hibernate.auction.Item.bids"
                      usage="read-write"/>

  </session-factory>

</hibernate-configuration>
```

Object to Table Mapping

Object to Table Mapping

- Object to table mappings can be specified in two ways
  - Through annotations in source code (preferred)
  - Through Hibernate mapping file (XML file) (deprecated)

Object to Table Mapping: Annotation

```
// Domain class is annotated, no hibernate mapping file required
@Entity
@Table(name = ”people")
public class Person implements Serializable{

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name = "id")
    private int id;

    @Column(name = ”NAME")
    private String name;

    protected Person() {
    }
    ...
```

Mapping Classes to Tables

Mapping classes to tables

- The *class* element

```
<class name="myPackage.Answer"
        table="answer"
        dynamic-update="false"
        dynamic-insert="false">
...
</class>
```

Possible attributes of <class> element

```
<class
        name="ClassName"                          (1)
        table="tableName"                         (2)
        discriminator-value="discriminator_value"     (3)
        mutable="true|false"                      (4)
        schema="owner"                            (5)
        catalog="catalog"                         (6)
        proxy="ProxyInterface"                    (7)
        dynamic-update="true|false"               (8)
        dynamic-insert="true|false"               (9)
        select-before-update="true|false"         (10)
        polymorphism="implicit|explicit"          (11)
        where="arbitrary sql where condition"     (12)
        persister="PersisterClass"                (13)
        batch-size="N"                            (14)
        optimistic-lock="none|version|dirty|all"  (15)
        lazy="true|false"                         (16)
        entity-name="EntityName"                  (17)
        check="arbitrary sql check condition"     (18)
        rowid="rowid"                             (19)
        subselect="SQL expression"                (20)
```

Attributes of <class> element

- *name*: The fully qualified Java class name of the persistent class (or interface)
- *table* (optional - defaults to the unqualified class name): The name of its database table
- *discriminator-value* (optional - defaults to the class name): A value that distinguishes individual subclasses, used for polymorphic behavior

Attributes of <class>

- *dynamic-update* (optional, defaults to false): Specifies that UPDATE SQL should be generated at runtime and contain only those columns whose values have changed.
- *dynamic-insert* (optional, defaults to false): Specifies that INSERT SQL should be generated at runtime and contain only the columns whose values are not null.
- *optimistic-lock* (optional, defaults to version): Determines the optimistic locking strategy.

Data Access Object (DAO) Pattern

What is DAO pattern?

- DAO pattern
  - Separation of data access (persistence) logic from business logic
  - Enables easier replacement of database without affecting business logic
- DAO implementation strategies
  - Strategy 1: Runtime pluggable through factory class
  - Strategy 2: Domain DAO interface and implementation

Example: PersonDaoInterface

```
public interface PersonDaoInterface {
        public void create(Person p) throws SomeException;
        public void delete(Person p) throws SomeException;
        public Person find(Long id) throws SomeException;
        public List findAll() throws SomeException;
        public void update(Person p) throws SomeException;
        public WhateverType
                whateverPersonRelatedMethod(Person p)
                                        throws SomeException;
}
```

Lab:

Exercise 4: DAO Pattern
3514_hibernate_basics.zip

Code with Passion!