Hibernate Annotations

Hibernate 2.x, mapping metadata is most of the time declared in XML text files.

What is Annotation

- XDoclet, utilizing Javadoc source code annotations and a preprocessor at compile time. The same kind of annotation support is now available in the standard JDK, although more powerful and better supported by tools.
- IntelliJ IDEA, and Eclipse for example, support autocompletion and syntax highlighting of JDK 5.0 annotations.
- Annotations are compiled into the bytecode and read at runtime (in Hibernate's case on startup) using reflection, so no external XML files are needed.

Setting up an annotations project

- set up your classpath (after you have created a new project in your favorite IDE)
- Copy all Hibernate3 core and required 3rd party library files (see lib/README.txt in Hibernate).
- Copy hibernate-annotations.jar and lib/ejb3-persistence.jar from the Hibernate Annotations distribution to your classpath as well.Hibernate Lucene Integration, add the lucene jar file.
- We also recommend a small wrapper class to startup Hibernate in a static initializer block, known as HibernateUtil. You might have seen this class in various forms in other areas of the Hibernate documentation. For Annotation support you have to enhance this helper class as follows

Singleton class

```
public class HibernateUtil {
    private static final SessionFactory sessionFactory;
    static {
        try {
                sessionFactory = new AnnotationConfiguration().buildSessionFactory();
            } catch (Throwable ex) {
                / / Log exception!
                throw new ExceptionInInitializerError(ex);
            }
       }
    public static Session getSession() throws HibernateException {
            return sessionFactory.openSession();
    }
}
```

Hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.DerbyDialect</property>
    <property name="hibernate.connection.driver_class">org.apache.derby.jdbc.ClientDriver</property>
    <property name="hibernate.connection.url">jdbc:derby://localhost:1527/sample</property>
    <property name="hibernate.connection.username">app</property>
    <property name="hibernate.connection.password">app</property>
    <property name="hibernate.hbm2ddl.auto">update</property>
    <mapping class="com.demo.Laptop"/>
    <mapping class="com.demo.Person"/>
  </session-factory>
</hibernate-configuration>
```

Java Configuration
(No more hibernate.cfg.xml)

```
sessionFactory = new AnnotationConfiguration()
    .addPackage("test.animals") //the fully qualified package name
    .addAnnotatedClass(Flight.class)
    .addAnnotatedClass(Sky.class)
    .addAnnotatedClass(Person.class)
    .addAnnotatedClass(Dog.class)
    .buildSessionFactory();
```

Key Notes

- You can use your favorite configuration method for other properties (**hibernate.properties**, **hibernate.cfg.xml**, programmatic APIs, etc).
- You can even mix annotated persistent classes and classic **hbm.cfg.xml** declarations with the same SessionFactory.
- You can however not declare a class several times (whether annotated or through hbm.xml).
- You cannot mix configuration strategies (hbm vs annotations) in a mapped entity hierarchy either.
- HBM files are then prioritized over annotated metadata on a class to class basis. You can change the priority using **hibernate.mapping.precedence** property.
- The default is hbm, class, changing it to class, hbm will prioritize the annotated classes over hbm files when a conflict occurs.

Entity Beans

- The mappings are defined through JDK 5.0 annotations (an XML descriptor syntax for overriding will be defined in the EJB3 specification, but it's not finalized so far).
- Annotations can be split in two categories, the logical mapping annotations (allowing you to describe the object model, the class associations, etc.) and the physical mapping annotations (describing the physical schema, tables, columns, indexes, etc).
- EJB3 annotations are in the **javax.persistence.*** package. Most JDK 5 compliant IDE (like Eclipse, IntelliJ IDEA and Netbeans) can autocomplete annotation interfaces and attributes.

Sample Entity Class

```
@Entity
@Table(name="tbl_sky", uniqueConstraints = { @UniqueConstraint( columnNames = {"month", "day"})})
public class Flight implements Serializable {
    @Id
    private Long id;
    @Version
    @Column(name="version", updatable = false, name = "flight_name", nullable = false, length=50)
    private Integer versionNo;    // Persistent Property
   @Transient
   private transient int counter; //transient property
   @Basic(fetch = FetchType.LAZY)
   @Lob
   private String detailedComment;
   @Temporal(TemporalType.TIME)
   private java.util.Date departureTime;
   @Enumerated(STRING)
   private Starred Note;
    ……..
}
```

Sample Entity Class

- **@Entity** declares the class as an entity bean (i.e. a persistent POJO class), **@Id** declares the identifier property of this entity bean.
- The other mapping declarations are implicit. This configuration by exception concept is central to the new EJB3 specification and a major improvement.
- The class Flight is mapped to the Flight table, using the column id as its primary key column.
- Hibernate will guess the access type from the position of **@Id** or **@EmbeddedId**

A Simple Entity Class

- The version column may be a numeric (the recommended solution) or a timestamp as per the EJB3 spec.
- Hibernate support any kind of type provided that you define and implement the appropriate UserVersionType.
- entity manager will use it to detect conflicting updates (preventing lost updates you might otherwise see with the last-commit wins strategy)
- Every non static non transient property (field or method) of an entity bean is considered persistent, unless you annotate it as **@Transient**. The **@Basic** annotation allows you to declare the fetching strategy for a property

Sample Entity Class

- unique constraints to the table using the @**Unique**Constraint annotation in conjunction with @**Table** (for a unique constraint bound to a single column, refer to @**Column**)
- A unique constraint is applied to the tuple month, day. Note that the columnNames array refers to the logical column names.

@Column

```
@Column(
        name="columnName"; (1)
        boolean unique() default false; (2)
        boolean nullable() default true; (3)
        boolean insertable() default true; (4)
        boolean updatable() default true; (5)
        String columnDefinition() default ""; (6)
        String table() default ""; (7)
        int length() default 255; (8)
        int precision() default 0; // decimal precision (9)
        int scale() default 0; // decimal scale
)
```

Embedded objects

Component classes have to be annotated at the class level with the **@Embeddable** annotation. It is possible to override the column mapping of an embedded object for a particular entity using the **@Embedded** and **@AttributeOverride** annotation in the associated property

@Embedded

```
@AttributeOverrides( {
    @AttributeOverride(name="city", column = @Column(name="fld_city") )
    @AttributeOverride(name="nationality.iso2", column = @Column(name="nat_Iso2") ),
    @AttributeOverride(name="nationality.name", column = @Column(name="nat_CountryName") )
//nationality columns in homeAddress are overridden
} )
Address homeAddress;
```

Embedded objects

A embeddable object inherit the access type of its owning entity (note that you can override that using the Hibernate specific @**AccessType** annotations

```
@Embeddable
public class Address implements Serializable {
        private String city;
        private Country nationality; //no overriding here
         private String iso2;
        @Column(name="countryName")
        private String name;
}
```

Non-annotated property defaults

If a property is not annotated, the following rules apply:

- If the property is of a single type, it is mapped as **@Basic**
- Otherwise, if the type of the property is annotated as **@Embeddable**, it is mapped as **@Embedded.**
- Otherwise, if the type of the property is Serializable, it is mapped as **@Basic** in a column holding the object

in its serialized version

- Otherwise, if the type of the property is **java.sql.Clob** or **java.sql.Blob**, it is mapped as **@Lob** with the appropriate LobType

@GeneratedValue

The **@Id** annotation lets you define which property is the identifier of your entity bean. This property can be set by the application itself or be generated by Hibernate (preferred). You can define the identifier generation strategy thanks to the **@GeneratedValue** annotation:

- **AUTO** - either identity column, sequence or table depending on the underlying DB
- **TABLE** - table holding the id
- **IDENTITY** - identity column
- **SEQUENCE** - sequence

```
@Id
@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="SEQ_STORE")
private Long id;
```

@GeneratedValue

The scope of a generator can be the application or the class. Class-defined generators are not visible outside the class and can override application level generators. Application level generators are defined at package level

```
@javax.persistence.TableGenerator( name="EMP_GEN",
table="GENERATOR_TABLE",
        pkColumnName = "key",  valueColumnName = "hi", pkColumnValue="EMP",
        allocationSize=20)
@javax.persistence.SequenceGenerator( name="SEQ_GEN",
        sequenceName="my_sequence")
package org.hibernate.test.metadata;
```

@SequenceGenerator

```
@Entity
@javax.persistence.SequenceGenerator( name="SEQ_STORE",
                                      sequenceName="my_sequence")
public class Store implements Serializable {
private Long id;
@Id
@GeneratedValue(strategy=GenerationType.SEQUENCE,
                generator="SEQ_STORE")
public Long getId() { return id; }
}
```

Composite Primary Key

- annotate the component property as @**Id** and make the component class @**Embeddable** annotate the component property as @**EmbeddedId** annotate the class as @**IdClass** and annotate each property of the entity involved in the primary key with @**Id**

```
@Entity
@Id
Class(FootballerPk.class)
public class Footballer {
//part of the id key
   @Id
private String firstname;
}
```

Composite Primary Key

```
@Embeddable
public class FootballerPk implements Serializable {
//same name and type as in Footballer
public String getFirstname() {
return firstname;
}
```

Composite Identifier

```
@Entity
@AssociationOverride( name="id.channel", joinColumns = @JoinColumn(name="chan_id") )
public class TvMagazin {
    @EmbeddedId public TvMagazinPk id;
    @Temporal(TemporalType.TIME) Date time;
}
@Embeddable
public class TvMagazinPk implements Serializable {
    @ManyToOne public Channel channel;
    public String name;
    @ManyToOne public Presenter presenter;
}
```

Mapping inheritance

Hibernate supports the three types of inheritance:

- Table per Class Strategy: the **<union-class>** element in Hibernate
- Single Table per Class Hierarchy Strategy: the **<subclass>** element in Hibernate
- Joined Subclass Strategy: the **<joined-subclass>** element in Hibernate

Table per class

Hibernate work around most of them implementing this strategy using SQL UNION queries. It is commonly used for the top level of an inheritance hierarchy:

```
@Entity
@Inheritance(strategy =
        InheritanceType.TABLE_PER_CLASS)
public class Flight implements Serializable {}
```

Single table per class hierarchy

All properties of all super- and subclasses are mapped into the same table, instances are distinguished by a special discriminator column

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="planetype",
discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("Plane")
public class Plane { ... }
@Entity @DiscriminatorValue("A320")
public class A320 extends Plane { ... }
```

Joined subclasses

The **@PrimaryKeyJoinColumn** and **@PrimaryKeyJoinColumns** annotations define the primary key(s) of the joined subclass table

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Boat implements Serializable { ... }
@Entity
public class Ferry extends Boat { ... }
@Entity
@PrimaryKeyJoinColumn(name="BOAT_ID")
public class AmericaCupClass extends Boat { ... }
```

Inherit properties from superclasses

This is sometimes useful to share common properties through a technical or a business superclass without including it as a regular mapped entity (ie no specific table for this entity) **@MappedSuperclass.**

```
@MappedSuperclass
public class BaseEntity {
    @Basic
    @Temporal(TemporalType.TIMESTAMP)
    public Date getLastUpdate() { ... }
    public String getLastUpdater() { ... }
}
@Entity
public class Order extends BaseEntity {
     @Id
     public Integer getId() { ... }
}
```