Hibernate JPA Annotations - Contents:

Annotation	Package Detail/Import statement
@Entity	import javax.persistence.Entity;
@Table	import javax.persistence.Table;
@Column	import javax.persistence.Column;
@Id	import javax.persistence.Id;
@GeneratedValue	import javax.persistence.GeneratedValue;
@Version	import javax.persistence.Version;
@OrderBy	import javax.persistence.OrderBy;
@Transient	import javax.persistence.Transient;
@Lob	import javax.persistence.Lob;
Hibernate Association Mapping Annotations	
@OneToOne	import javax.persistence.OneToOne;
@ManyToOne	import javax.persistence.ManyToOne;
@OneToMany	import javax.persistence.OneToMany;
@ManyToMany	import javax.persistence.ManyToMany;
@PrimaryKeyJoinColumn	import javax.persistence.PrimaryKeyJoinColumn;
@JoinColumn	import javax.persistence.JoinColumn;
@JoinTable	import javax.persistence.JoinTable;
@MapsId	import javax.persistence.MapsId;
Hibernate Inheritance Mapping Annotations	
@Inheritance	import javax.persistence.Inheritance;
@DiscriminatorColumn	import javax.persistence.DiscriminatorColumn;
@DiscriminatorValue	import javax.persistence.DiscriminatorValue;

@Entity

Annotate all your entity beans with @Entity.

```
1@Entity
2public class Company implements Serializable {
3...
4}
```

@Table

Specify the database table this Entity maps to using the name attribute of @Table annotation. In the example below, the data will be stored in 'company' table in the database.

```
@Entity
@Table(name = "company")
public class Company implements Serializable {
...
}
```

@Column

Specify the column mapping using @Column annotation.

```
@Entity
@Table(name = "company")
public class Company implements Serializable {

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

...
}
```

@Id

Annotate the id column using @Id.

```
@Entity
@Table(name = "company")
public class Company implements Serializable {

   @Id
   @Column(name = "id")
   private int id;

...
}
```

@GeneratedValue

Let database generate (auto-increment) the id column.

```
@Entity
@Table(name = "company")
public class Company implements Serializable {

   @Id
   @Column(name = "id")
   @GeneratedValue
   private int id;

...
}
```

@Version

Control versioning or concurrency using @Version annotation.

```
@Entity
@Table(name = "company")
public class Company implements Serializable {

  @Version
  @Column(name = "version")
  private Date version;

...
}
```

@OrderBy

Sort your data using @OrderBy annotation. In example below, it will sort all contacts in a company by their firstname in ascending order.

```
@OrderBy("firstName asc")
private Set contacts;
```

@Transient

Annotate your transient properties with @Transient.

@Lob

Annotate large objects with @Lob.

Hibernate Association Mapping Annotations

Example App DB Schema

The database for this tutorial is designed to illustrate various association mapping concepts. In RDBMS implementations, entities are joined using the following ways:

- Shared Primary Key
- Foreign Key
- Association Table

In our example app,

- Tables company and companyDetail have shared values for primary key. It is a one-to-one assoication.
- Tables contact and contactDetail are linked through a foreign key. It is also a one to one association.

- Tables contact and company are linked through a foriegn key in many-to-one association with contact being the owner.
- Tables company and companyStatus are linked through a foreign key in many-to-one association with company being the owner.

@OneToOne

- Use @PrimaryKeyJoinColumn for associated entities sharing the same primary key.
- Use @JoinColumn & @OneToOne mappedBy attribute when foreign key is held by one of the entities.
- Use @JoinTable and mappedBy entities linked through an association table.
- Persist two entities with shared key using @MapsId

For entities Company and CompanyDetail sharing the same primary key, we can associate them using @OneToOne and @PrimaryKeyJoinColumn as shown in the example below.

Notice that the id property of CompanyDetail is NOT annotated with @GeneratedValue. It will be populated by id value of Company.

```
1
2
3  @Entity
4  @Table(name = "company")
5  public class Company implements Serializable {
6
7    @Id
8    @Column(name = "id")
9    @GeneratedValue
10   private int id;
11
12   @OneToOne(cascade = CascadeType.MERGE)
13   @PrimaryKeyJoinColumn
14   private CompanyDetail companyDetail;
15
16   ...
17 }
18
19 @Entity
20 @Table(name = "companyDetail")
21 public class CompanyDetail implements Serializable {
22
23   @Id
24   @Column(name = "id")
25   private int id;
26
27   ...
28 }
29
30
```

For entities Contact and ContactDetail linked through a foriegn key, we can use @OneToOne and @JoinColumn annotations. In example below, the id genereated for Contact will be mapped to 'contact_id' column of ContactDetail table. Please note the usage of @MapsId for the same.

```
1  @Entity
2  @Table(name = "contactDetail")
```

```
public class ContactDetail implements Serializable {

    @Id
    @Column(name = "id")
    @GeneratedValue
    private int id;

    @OneToOne
    @MapsId
    @JoinColumn(name = "contactId")
    private Contact contact;

    ...
}

@Entity
@Table(name = "contact")
public class Contact implements Serializable {

    @Id
    @Column(name = "ID")
    @GeneratedValue
    private Integer id;

    @OneToOne(mappedBy = "contact", cascade = CascadeType.ALL)
    private ContactDetail contactDetail;

    ....
}
```

Also note that the relationship between Company and CompanyDetail is uni-directional. On the other hand, the relationship between Contact and Contact Detail is bi-directional and that can be achieved using 'mappedBy' attribute.

The rationale to have one relationship as uni-directional and other as bi-directional in this tutorial is to illustrate both concepts and their usage. You can opt for uni-directional or bi-directional relationships to suit your needs.

@ManyToOne

- Use @JoinColumn when foreign key is held by one of the entities.
- Use @JoinTable for entities linked through an association table.

The two examples below illustrate many-to-one relationships. Contact to Company and Company to CompanyStatus. Many contacts can belong to a company. Similary many companies can share the same status (Lead, Prospect, Customer) - there will be many companies that are currently leads.

```
@Entity
@Table(name = "contact")
public class Contact implements Serializable {

```

```
    @ManyToOne
    @JoinColumn(name = "companyId")
    private Company company;

    ...

}

@Entity
@Table(name = "company")
public class Company implements Serializable {

    @ManyToOne
    @JoinColumn(name = "statusId")
    private CompanyStatus status;

    ...

}
```

@OneToMany

- Use mappedBy attribute for bi-directional associations with ManyToOne being the owner.
- OneToMany being the owner or unidirectional with foreign key - try to avoid such associations but can be achieved with @JoinColumn
- @JoinTable for Unidirectional with association table

Please see the many-to-one relationship between Contact and Company above. Company to Contact will be a one-to-many relationship. The owner of this relationship is Contact and hence we will use 'mappedBy' attribute in Company to make it bi-directional relationship.

```
@Entity
@Table(name = "company")
public class Company implements Serializable {

    @OneToMany(mappedBy = "company", fetch = FetchType.EAGER)
    @OrderBy("firstName asc")
    private Set contacts;

    ...

}
```

Company to CompanyStatus relationship as uni-directional.

@ManyToMany

- Use @JoinTable for entities linked through an association table.
- Use mappedBy attribute for bi-directional association.

@PrimaryKeyJoinColumn

@PrimaryKeyJoinColumn annotation is used for associated entities sharing the same primary key.

```
@Entity
@Table(name = "company")
public class Company implements Serializable {

  @Id
  @Column(name = "id")
  @GeneratedValue
  private int id;

  @OneToOne(cascade = CascadeType.MERGE)
  @PrimaryKeyJoinColumn
  private CompanyDetail companyDetail;

  ...
}
```

@JoinColumn

Use @JoinColumn annotation for one-to-one or many-to-one associations when foreign key is held by one of the entities. We can use @OneToOne or @ManyToOne mappedBy attribute for bi-directional relations.

```
@ManyToOne
@JoinColumn(name = "statusId")
private CompanyStatus status;
```

@JoinTable

Use @JoinTable and mappedBy for entities linked through an association table.

@MapsId

Persist two entities with shared key (when one entity holds a foreign key to the other) using @MapsId annotation.

```
@OneToOne
@MapsId
@JoinColumn(name = "contactId")
private Contact contact;
```

Hibernate Inheritance Mapping Annotations

To understand Inheritance Mapping annotations, Once you understand Inheritance mapping concepts, please review below for annotations to be used.

- table per class hierarchy - single table per Class Hierarchy Strategy: the <subclass> element in Hibernate

```
@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 { ... }
```

- table per class/subclass - joined subclass Strategy: the <joined-subclass> element in Hibernate

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

@Entity
@PrimaryKeyJoinColumn
public class Ferry extends Boat { ... }
```

- table per concrete class - table per Class Strategy: the <union-class> element in Hibernate

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

Note: This strategy does not support the IDENTITY generator strategy: the id has to be shared across several tables. Consequently, when using this strategy, you should not use AUTO nor IDENTITY.

Hibernate Mapping Annotations

I list all the Hibernate mapping annotations for your quick reference :

@AccessType

@Any

@AnyMetaDef

@AnyMetaDefs

@AttributeAccessor

@BatchSize

@Cache

@Cascade

@Check

@CollectionId

@CollectionType

@ColumnDefault

@Columns

@ColumnTransformer

@ColumnTransformers

@CreationTimestamp

@DiscriminatorFormula

@DiscriminatorOptions

@DynamicInsert

@DynamicUpdate

@Entity @Fetch

@FetchProfile

@FetchProfile.FetchOverride

@FetchProfiles

@Filter

@FilterDef

@FilterDefs

@FilterJoinTable

@FilterJoinTables

@Filters

@ForeignKey

@Formula

@Generated

@GeneratorType

@GenericGenerator

@GenericGenerators

@Immutable

@Index

@IndexColumn

@JoinColumnOrFormula

@JoinColumnsOrFormulas

@JoinFormula

@LazyCollection

@LazyGroup

@LazyToOne

@ListIndexBase

@Loader

@ManyToAny

@MapKeyType

@MetaValue

@NamedNativeQueries

@NamedQueries

@NamedQuery

@Nationalized

@NaturalId

@NaturalIdCache

@NotFound

@OnDelete

@OptimisticLock

@OptimisticLocking

@OrderBy

@ParamDef

@Parameter

@Parent

@Persister

@Polymorphism

@Proxy

@RowId

@SelectBeforeUpdate

@Sort

@SortComparator

@SortNatural

@Source

@SQLDelete

@SQLDeleteAll

@SqlFragmentAlias

@SQLInsert

@SQLUpdate

@Subselect

@Synchronize

@Table

@Tables

@Target

@Tuplizer

@Tuplizers

@Type

@TypeDef

@TypeDefs

@UpdateTimestamp

@ValueGenerationType

@Where

@WhereJoinTable

@AccessType

The @AccessType annotation is deprecated. You should use either the JPA @Access or the Hibernate native @AttributeAccessor annotation.

@Any

The @Any annotation is used to define the any-to-one association, which can point to one of several entity types.

@AnyMetaDef

The @AnyMetaDef annotation is used to provide metadata about an @Any or @ManyToAny mapping.

@AnyMetaDefs

The @AnyMetaDefs annotation is used to group multiple @AnyMetaDef annotations.

@AttributeAccessor

The @AttributeAccessor annotation is used to specify a custom PropertyAccessStrategy. This should only be used to name a custom PropertyAccessStrategy. For property/field access type, the JPA@Access annotation should be preferred.

@BatchSize

The @BatchSize annotation is used to specify the size for batch loading the entries of a lazy collection.

@Cache

The @Cache annotation is used to specify the CacheConcurrencyStrategy of a root entity or a collection.

@Cascade

The @Cascade annotation is used to apply the Hibernate specific CascadeType strategies (e.g. CascadeType.LOCK, CascadeType.SAVE_UPDATE, CascadeType.REPLICATE) on a given association.

For JPA cascading, I prefer using the javax.persistence.CascadeType instead. When combining both JPA and Hibernate CascadeType strategies, Hibernate will merge both sets of cascades.

@Check

The @Check annotation is used to specify an arbitrary SQL CHECK constraint which can be defined at the class level.

@CollectionId

The @CollectionId annotation is used to specify an identifier column for an idbag collection.

You might want to use the JPA@OrderColumn instead.

@CollectionType

The @CollectionType annotation is used to specify a custom collection type.

The collection can also name a @Type, which defines the Hibernate Type of the collection elements.

@ColumnDefault

The @ColumnDefault annotation is used to specify the DEFAULT DDL value to apply when using the automated schema generator.

The same behavior can be achieved using the definition attribute of the JPA @Column annotation.

@Columns

The @Columns annotation is used to group multiple JPA @Column annotations.

@ColumnTransformer

The @ColumnTransformer annotation is used to customize how a given column value is read from or write into the database.

@ColumnTransformers

The @ColumnTransformers annotation is used to group multiple @ColumnTransformer annotations.

@CreationTimestamp

The @CreationTimestamp annotation is used to specify that the currently annotated temporal type must be initialized with the current JVM timestamp value.

@DiscriminatorFormula

The @DiscriminatorFormula annotation is used to specify a Hibernate @Formula to resolve the inheritance discriminator value.

@DiscriminatorOptions

The @DiscriminatorOptions annotation is used to provide the force and insert Discriminator properties.

@DynamicInsert

The @DynamicInsert annotation is used to specify that the INSERT SQL statement should be generated whenever an entity is to be persisted.

By default, Hibernate uses a cached INSERT statement that sets all table columns. When the entity is annotated with the @DynamicInsert annotation, the PreparedStatement is going to include only the non-null columns.

@DynamicUpdate

The @DynamicUpdate annotation is used to specify that the UPDATE SQL statement should be generated whenever an entity is modified.

By default, Hibernate uses a cached UPDATE statement that sets all table columns. When the entity is annotated with the @DynamicUpdate annotation, the PreparedStatement is going to include only the columns whose values have been changed.

@Entity

The @Entity annotation is deprecated. Use the JPA @Entity annotation instead.

@Fetch

The @Fetch annotation is used to specify the Hibernate specific FetchMode (e.g. JOIN, SELECT, SUBSELECT) used for the currently annotated association:

@FetchProfile

The @FetchProfile annotation is used to specify a custom fetching profile, similar to a JPA Entity Graph.

@FetchProfile.FetchOverride

The @FetchProfile.FetchOverride annotation is used in conjunction with the @FetchProfile annotation, and it's used for overriding the fetching strategy of a particular entity association.

@FetchProfiles

The @FetchProfiles annotation is used to group multiple @FetchProfile annotations.

@Filter

The @Filter annotation is used to add filters to an entity or the target entity of a collection.

@FilterDef

The @FilterDef annotation is used to specify a @Filter definition (name, default condition and parameter types, if any).

@FilterDefs

The @FilterDefs annotation is used to group multiple @FilterDef annotations.

@FilterJoinTable

The @FilterJoinTable annotation is used to add @Filter capabilities to a join table collection.

@FilterJoinTables

The @FilterJoinTables annotation is used to group multiple @FilterJoinTable annotations.

@Filters

The @Filters annotation is used to group multiple @Filter annotations.

@ForeignKey

The @ForeignKey annotation is deprecated. Use the JPA 2.1 @ForeignKey annotation instead.

@Formula

The @Formula annotation is used to specify an SQL fragment that is executed in order to populate a given entity attribute.

@Generated

The @Generated annotation is used to specify that the currently annotated entity attribute is generated by the database.

@GeneratorType

The @GeneratorType annotation is used to provide a ValueGenerator and a GenerationTime for the currently annotated generated attribute.

@GenericGenerator

The @GenericGenerator annotation can be used to configure any Hibernate identifier generator.

@GenericGenerators

The @GenericGenerators annotation is used to group multiple @GenericGenerator annotations.

@Immutable

The@Immutable annotation is used to specify that the annotated entity, attribute, or collection is immutable.

@Index

The @Index annotation is deprecated. Use the JPA @Index annotation instead.

@IndexColumn

The @IndexColumn annotation is deprecated. Use the JPA @OrderColumn annotation instead.

@JoinColumnOrFormula

The @JoinColumnOrFormula annotation is used to specify that the entity association is resolved either through a FOREIGN KEY join (e.g. @JoinColumn) or using the result of a given SQL formula (e.g. @JoinFormula).

@JoinColumnsOrFormulas

The @JoinColumnsOrFormulas annotation is used to group multiple @JoinColumnOrFormula annotations.

@JoinFormula

The @JoinFormula annotation is used as a replacement for @JoinColumn when the association does not have a dedicated FOREIGN KEY column.

@LazyCollection

The @LazyCollection annotation is used to specify the lazy fetching behavior of a given collection.

The TRUE and FALSE values are deprecated since you should be using the JPA FetchType attribute of the @ElementCollection, @OneToMany, or @ManyToMany collection.

@LazyGroup

The @LazyGroup annotation is used to specify that an entity attribute should be fetched along with all the other attributes belonging to the same group.

To load entity attributes lazily, bytecode enhancement is needed. By default, all non-collection attributes are loaded in one group named "DEFAULT."

This annotation allows defining different groups of attributes to be initialized together when access one attribute in the group.

@LazyToOne

The @LazyToOne annotation is used to specify the laziness options, represented by LazyToOneOption, available for a @OneToOne or @ManyToOne association.

@ListIndexBase

The @ListIndexBase annotation is used to specify the start value for a list index, as stored in the database.

By default, List indexes are stored starting at zero. This is generally used in conjunction with @OrderColumn.

@Loader

The @Loader annotation is used to override the default SELECT query used for loading an entity loading.

@ManyToAny

The @ManyToAny annotation is used to specify a many-to-one association when the target type is dynamically resolved.

@MapKeyType

The @MapKeyType annotation is used to specify the map key type.

@MetaValue

The @MetaValue annotation is used by the @AnyMetaDef annotation to specify the association between a given discriminator value and an entity type.

@NamedNativeQueries

The @NamedNativeQuery annotation extends the JPA @NamedNativeQuery with Hibernate specific features.

@NamedQueries

The @NamedQueries annotation is used to group multiple @NamedQuery annotations.

@NamedQuery

The @NamedQuery annotation extends the JPA @NamedQuery with Hibernate specific features.

@Nationalized

The @Nationalized annotation is used to specify that the currently annotated attribute is a character type (e.g. String, Character, Clob) that is stored in a nationalized column type (NVARCHAR, NCHAR, NCLOB).

@NaturalId

The @NaturalId annotation is used to specify that the currently annotated attribute is part of the natural id of the entity.

@NaturalIdCache

The @NaturalIdCache annotation is used to specify that the natural id values associated with the annotated entity should be stored in the second-level cache.

@NotFound

The @NotFound annotation is used to specify the NotFoundAction strategy for when an element is not found in a given association.

@OnDelete

The @OnDelete annotation is used to specify the delete strategy employed by the currently annotated collection, array, or joined subclasses. This annotation is used by the automated schema generation tool to generate the appropriate FOREIGN KEY DDL cascade directive.

@OptimisticLock

The @OptimisticLock annotation is used to specify if the currently annotated attribute will trigger an entity version increment upon being modified.

@OptimisticLocking

The @OptimisticLocking annotation is used to specify the currently annotated an entity optimistic locking strategy.

@OrderBy

The @OrderBy annotation is used to specify a SQL ordering directive for sorting the currently annotated collection.

It differs from the JPA @OrderBy annotation because the JPA annotation expects a JPQL order-by fragment, not an SQL directive.

@ParamDef

The @ParamDef annotation is used in conjunction with @FilterDef so that the Hibernate Filter can be customized with runtime-provided parameter values.

@Parameter

The @Parameter annotation is a generic parameter (basically a key/value combination) used to parametrize other annotations, like @CollectionType, @GenericGenerator, @Type, and @TypeDef.

@Parent

The @Parent annotation is used to specify that the currently annotated embeddable attribute references back the owning entity.

@Persister

The @Persister annotation is used to specify a custom entity or collection persister.

For entities, the custom persister must implement the EntityPersister interface.

For collections, the custom persister must implement the CollectionPersister interface.

@Polymorphism

The @Polymorphism annotation is used to define the PolymorphismType Hibernate will apply to entity hierarchies.

@Proxy

The @Proxy annotation is used to specify a custom proxy implementation for the currently annotated entity.

@RowId

The @RowId annotation is used to specify the database column used as a ROWID pseudocolumn. For instance, Oracle defines the ROWID pseudocolumn as something that provides the address of every table row.

@SelectBeforeUpdate

The @SelectBeforeUpdate annotation is used to specify that the currently annotated entity state be selected from the database when determining whether to perform an update when the detached entity is reattached.

@Sort

The @Sort annotation is deprecated. Use the Hibernate specific @SortComparator or @SortNatural annotations instead.

@SortComparator

The @SortComparator annotation is used to specify a Comparator for sorting the Set/Map in-memory.

@SortNatural

The @SortNatural annotation is used to specify that the Set/Map should be sorted using natural sorting.

@Source

The @Source annotation is used in conjunction with a @Version timestamp entity attribute indicating the SourceType of the timestamp value.

@SQLDelete

The @SQLDelete annotation is used to specify a custom SQL DELETE statement for the currently annotated entity or collection.

@SQLDeleteAll

The @SQLDeleteAll annotation is used to specify a custom SQL DELETE statement when removing all elements of the currently annotated collection.

@SqlFragmentAlias

The @SqlFragmentAlias annotation is used to specify an alias for a Hibernate @Filter.

The alias (e.g. myAlias) can then be used in the @Filter condition clause using the {alias} (e.g. {myAlias}) placeholder.

@SQLInsert

The @SQLInsert annotation is used to specify a custom SQL INSERT statement for the currently annotated entity or collection.

@SQLUpdate

The @SQLUpdate annotation is used to specify a custom SQL UPDATE statement for the currently annotated entity or collection.

@Subselect

The @Subselect annotation is used to specify an immutable and read-only entity using a custom SQL SELECT statement.

@Synchronize

The @Synchronize annotation is usually used in conjunction with the @Subselect annotation to specify the list of database tables used by the @Subselect SQL query.

@Table

The @Table annotation is used to specify additional information to a JPA @Table annotation, like custom INSERT, UPDATE or DELETE statements or a specific FetchMode.

@Tables

The @Tables annotation is used to group multiple @Table annotations.

@Target

The @Target annotation is used to specify an explicit target implementation when the currently annotated association is using an interface type.

@Tuplizer

The @Tuplizer annotation is used to specify a custom tuplizer for the currently annotated entity or embeddable.

@Tuplizers

The @Tuplizers annotation is used to group multiple @Tuplizer annotations.

@Type

The @Type annotation is used to specify the Hibernate @Type used by the currently annotated basic attribute.

@TypeDef

The @TypeDef annotation is used to specify a @Type definition, which can later be reused for multiple basic attribute mappings.

@TypeDefs

The @TypeDefs annotation is used to group multiple @TypeDef annotations.

@UpdateTimestamp

The @UpdateTimestamp annotation is used to specify that the currently annotated timestamp attribute should be updated with the current JVM timestamp whenever the owning entity gets modified.

@ValueGenerationType

The @ValueGenerationType annotation is used to specify that the current annotation type should be used as a generator annotation type.

@Where

The @Where annotation is used to specify a custom SQL WHERE clause used when fetching an entity or a collection.

@WhereJoinTable

The @WhereJoinTable annotation is used to specify a custom SQL WHERE clause used when fetching a join collection table.