**PENDING:**

**JPA Collections – MAP – page 114**

**EJB 1.1:**  
- Entity Bean must be supported by EJB container  
- no Message Driven Bean  
- has CMP, but never describe how to configure relationship between CMP EntityEJBs, this depends on container tool provided by vendor  
- RMI-IIOP is suggested for EJB communication  
- Deployment Descriptor xml file is introduced  
**EJB 2.0:**  
- Message Driven Bean is introduced  
- With CMP, you can configure the relationship between CMP EntityEJBs, in ejb-jar.xml  
- EJB-QL for CMP EntityEJB query language  
- RMI-IIOP is a must for EJB communication

**JPA 2.0**

* POJO Persistence
* Nonintrusive (also mind the misconception Transparent Persistence)
* Object Queries (derived from EJBQL and modeled after SQL)
* Mobile Entities (Detaching and Attaching)
* Simple Configurations (xml / annotations)
* Not Container Managed (can be used outside Application Server)
* Configuration by exception.

Persistent Unit Entity ManagerFactory

Persistence Context Entity Manager

**Entity**

- annotated with @Entity

- Non final class , enums and interfaces cannot be defined as Entity

- Must have no-arg constructor - public/protected if with arg constructors are defined.

- Persistence Instance variables cannot be final

**Access Type:**

- Field Access (private,protected or default can be used)

\* all non transient fields are peristable ie not annotated with @Transient

- Property Access (public ,protected can be used)

- runtime Exceptions like PersistenceException while accessing the property methods will mark current transaction for roll-back.

Default Accesstype can be declared at the Entity level

@Access(value = AccessType.FIELD)

Property Types Allowed for Entity

- Java Primitives /wrappers , int[],byte[],Byte[],char[],Character[]

- java.lang.String

- java.math.BigDecimal, BigInteger

-java.util.Date,Calendar,

- java.sql.Date,Time,Timestamp

- user defined types that implement Serializable

- enums

- entity types , collection of entity types

- embeddable class, collection of basic and embeddable types

Entity Identity:

Primary Key

- All entities should have a primary key

- @Id

Composite Primary Key

- @EmbeddedId , @IdClass

- should implement Serializable

- should support both field access and property access

- should override equals and hashcode

- If the composite primary key class is represented as an id class, the names of primary key fields or properties in the

primary key class and those of the entity class to which the id class is mapped must correspond and their types must be the same.

Primary Derived Identity:

When an identifier(dependant entity) in one entity includes a foreign key to another entity (i.e parent entity).

there exists a one-one or many-one relationship from dependant to parent.

- a dependent entity might have multiple parent entities

- a dependent entity should have all its relationships to parents entities set before it can be persisted

-The Id attribute in the entity class and the corresponding attribute in the id class must

have the same name.

- If an Id attribute in the entity class is of basic type, the corresponding attribute in the id class must have the same type.

- Go through the example section 2.4.1.3 Examples of Derived Identitiesin JPA2 specification

@AttributeOverrides(

@AttributeOverride(name=”” , column=@column(name=””))

)

Annotations

@Entity(value="")

@Table( name="" schema="schema\_name")

or catalog="schema\_name"

@Access(value=AccessType.FIELD)

@Column

name

nullable

insertable

updateable

@Basic(fetch=FetchType.LAZY)

FetchType.EAGER

optional= true/false

@Lob

This handles all Lobs like CLOB, BLOB

@Enumerated(value=EnumType.ORDINAL) or @Enumerated(EnumType.ORDINAL)

EnumType.ORDINAL

EnumType.STRING

@GeneratedValue(generator="" strategy="")

GenerationType.AUTO , IDENTITY,SEQUENCE

GenerationType.TABLE

@TableGenerator(name="" , table="", pkColumnName="' , valueColumnName="" , initialValue= , allocationSize= , schema="", catalog="")

can be defined in any class file and its available for the entire persistence unit.

Eventhough its is available for entire persistence unit , it still needs to defined for each class.

@SequenceGenerator(name="" , sequenceName="" , initialValue= , schema="", catalog="")

Table RelationShip Annotations

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

@ManyToOne

@JoinColumn(name="" , referencedColumnName ="")

name --> should refer to actual fk column name in the table

will denote the foreign key column in the relationship owner table.

@OneToOne ()

mappedBy=""placed on the inverse side of the relationship.

cascade={CascadeType.ALL, PERSIST,MERGE,DETACH,REFRESH,REMOVE,}

targetEntity=Sample.class

fetch=FetchType

@OneToMany

@ManyToMany

@JoinTable(name="',

joinColumns=@JoinColumn(name=""),inverseJoinColumns=@JoinColumn(name="")

)

## Unidirectional and Bidirectional

The main differenece is that bidirectional relationship provides navigational access in both directions, so that you can access the other side without explicit queries. Also it allows you to apply cascading options to both directions.

Note that navigational access is not always good, especially for "one-to-very-many" and "many-to-very-many" relationships. Imagine a Group that contains thousands of Users:

* How would you access them? With so many Users, you usually need to apply some filtering and/or pagination, so that you need to execute a query anyway (unless you use [collection filtering](http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/filters.html), which looks like a hack for me). Some developers may tend to apply filtering in memory in such cases, which is obviously not good for performance. Note that having such a relationship can encourage this kind of developers to use it without considering performance implications.
* How would you add new Users to the Group? Fortunately, Hibernate looks at the owning side of relationship when persisting it, so you can only set User.group. However, if you want to keep objects in memory consistent, you also need to add User to Group.users. But it would make Hibernate to fetch all elements of Group.users from the database!

So, I can't agree with the recommendation from the [Best Practices](http://docs.jboss.org/hibernate/core/3.3/reference/en/html/best-practices.html). You need to design bidirectional relationships carefully, considering use cases (do you need navigational access in both directions?) and possible performance implications.

@ElementCollection (targetClass=Sample.class)

JPA 2.0 defines an ElementCollection mapping. It is meant to handle several non-standard relationship mappings. An ElementCollection can be used to define a one-to-many relationship to an Embeddable object, or a Basic value (such as a collection of Strings). An ElementCollection can also be used in combination with a Map to define relationships where the key can be any type of object, and the value is an Embeddable object or a Basic value.

- @CollectionTable(name="", @JoinColumns=)

More Reference : http://en.wikibooks.org/wiki/Java\_Persistence/ElementCollection#ElementCollection

@OrderBy(" attributeName ASC") -

ASC is default

DESC for decending order.

changing the order in memory will not make changes in Db

the ordering is performed at the oracle query using "order by"

@OrderColumn(name="")

default value would be "attributename"\_ORDER .

Can be annotated in owner or the inverse side.

Persistence unit – points to the Database.

Persistence Contexts – a set managed entity instances. And are restricted to a single Persistence Unit.

Managed Entity – Entity in persistence context , hence can be managed by EntManag.

Note that all Container specific annotations like @PersistenceContext etc will work only in a EE container.

Though we could make some work in Spring Container.

<context:annotation-config />

<tx:annotation-driven/>

<bean id="entityManagerFactory"

class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">

<property name="staticMethod">

<value>javax.persistence.Persistence.createEntityManagerFactory</value>

</property>

<property name="arguments">

<list>

<value>persistenceUnit</value>

</list>

</property>

</bean>

<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">

<property name="entityManagerFactory" ref="entityManagerFactory"/>

</bean>

## Entity Manager Types

1. Container Managed
   1. Transaction-scoped

EntityManager – lookup for persistence context in Transaction – reuses it else creates new one – performs CRUD – releases persistence context –

End of Transaction – Persistence context destroyed.

* 1. Extended

1. Application Managed

Create EMF 🡪 EM 🡪 CRUD 🡪EM.close 🡪 EMF.close.

Transaction Management