EJB 3.0 Persistence API Quick Reference Guide

Obtaining an Entity Manager

Injection:

```
@PersistenceContext
private EntityManager em;
```

Programmatic bootstrapping:

```
EntityManagerFactory emf = Persistence.
  createEntityManagerFactory("accounting");
EntityManager em = emf.createEntityManager();
```

JNDI:

```
EntityManager em = (EntityManager)
  ctx.lookup("java:comp/env/accounting");
```

Primary Keys

Primary keys must be either a primitive, primitive wrapper, java.lang.String, java.util.Date, java.sql.Date, or a defined primary key class to map composite keys.

Basic primary key:

```
@Id long getId() { return id; }
```

Compound primary key:

```
@IdClass(com.example.PersonPK.class)
@Entity(access=AccessType.FIELD)
class Person {
  @Id String firstName;
  @Id String lastName;
}
```

Key Generation

Options for generation of keys by the container include AUTO, TABLE, SEQUENCE, IDENTITY, or NONE.

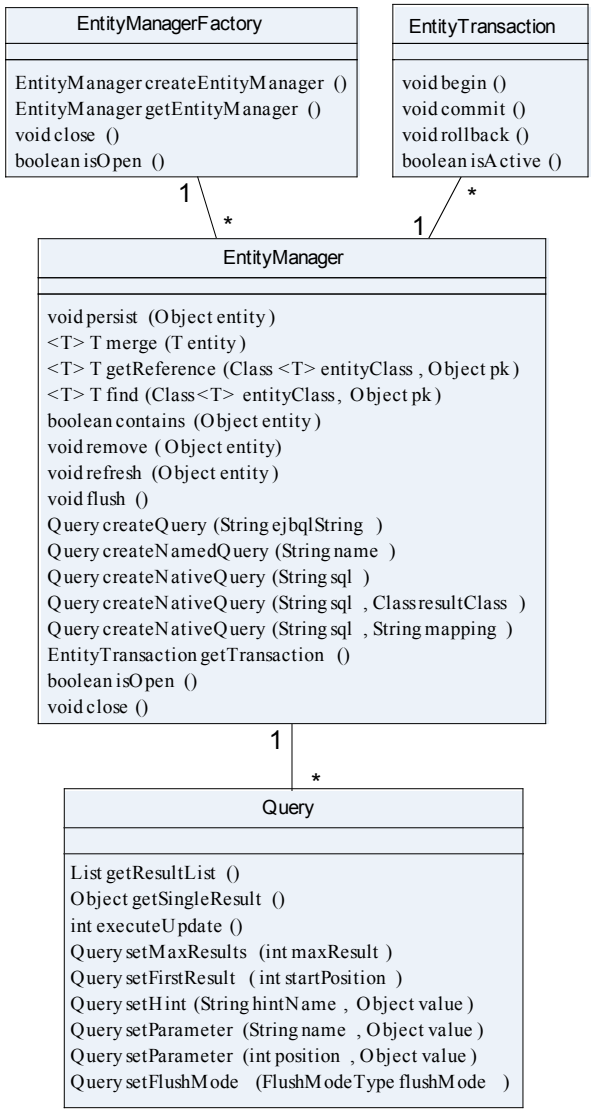
Implementation's preferred primary key generation

```
@Id(generate=GeneratorType.AUTO)
long getId() { return id; }
```

Database table primary key storage

```
@TableGenerator(name="KEYS", tableName="PKS",
  pkColumnValue="KEY")
@Id(generate=GeneratorType.TABLE,
  generator="KEYS")
long getId() { return id; }
```

javax.persistence Package Structure



Notes: some methods omitted for clarity; merge(T entity) will copy entity state to a new instance; find method returns null if no instance is found; instance returned by getReference will be lazily fetched

Identifying Entity Types

@Entity: Classes may be abstract or concrete. Both types are mapped as entities and may be used for queries.

@EmbeddableSuperclass: A class that provides mapping information, but is not an entity and thus not queryable nor persistent.

Table Mapping

Specify a single table for an entity:

Use the class fields for persistence

```
@Entity(access=AccessType.FIELD)
@Table(name="PEOPLE")
class Person { ... }
```

Specify multiple tables for a single entity:

Use the property accessor methods for persistence

```
@Entity
@Table(name="PEOPLE")
@SecondaryTable(name="PEOPLE_DETAILS"
  pkJoin=@PrimaryKeyJoinColumn(name="P_ID")
class Person { ... }
```

Map a field to a column:

```
@Column(name="CURRENT_AGE", nullable=false)
int getAge() { return age; }
```

Map a large object to a column in a secondary table:

```
@Lob
@Column(name="PICTURE",
  secondaryTable="PEOPLE_DETAILS")
JPEGImage getPhoto() {return photo; }
```

Transient field:

```
@Transient float getAgeInDecades() { ... }
```

Relationships

OneToOne:

```
@OneToOne(optional=false)
@JoinColumn(name="ADDRESS_ID",
  unique=true, nullable=false)
Address getAddress() { ... }
```

OneToMany / ManyToOne:

Map a Customer to collection of Orders

```
@OneToMany(cascade=CascadeType.ALL,
  mappedBy="customer")
Set<Order> getOrders() { ... }

@ManyToOne
@JoinColumn(name="CUST_ID", nullable=false)
Customer getCustomer() { ... }
```

ManyToMany:

Map a customer to a collection of phone numbers and a phone number to a collection of customers.

```
@ManyToMany
@JoinTable(table=@Table(name="CUST_PHONES")
Set getPhones() { return phones; }

@ManyToMany(mappedBy="phones")
Set<Customer> getCustomers() {
  return customers;
}
```


Content based upon EJB3.0 public draft http://solarmetric.com/resources/ejb-api-quickref.pdf

EJB 3.0 Persistence API Quick Reference Guide

Fetch Types

Fetch types govern when relationship data are loaded. The available options are LAZY and EAGER.

```
@OneToMany(fetch=FetchType.EAGER)
```
Children loaded when the parent is loaded

```
@OneToMany(fetch=FetchType.LAZY)
```
Children not initially loaded

In ManyToOne and OneToOne relationships EAGER is the default. For OneToMany and ManyToMany relationships LAZY is the default.

Cascading

Cascading can be used to propagate persistence operations to related entities. There are four available cascade types: PERSIST, MERGE, REMOVE, REFRESH. These can be used in combination or the value ALL can be used to signify all four.
If we define getChildren() as:

```
@OneToMany(cascade={CascadeType.PERSIST,
  CascadeType.REMOVE})
Set<Person> getChildren() {
  return children;
}
```

then persistence or removal of a person will also result in persistence or removal of all of the children. However, merge or refresh operations for the person will not cause the children to be merged or refreshed. If no cascading behavior is specified the default is no cascading at all.

For additional detail about EJB3 persistence see http://www.solarmetric.com/kodo4docs/

Listeners and Callbacks

Callbacks can be handled either by methods on the entity itself, or by a separate entity listener class. Listener classes must be stateless with a no-argument constructor. Kodo allows callback methods to access the EntityManager.

Available lifecycle events:

PrePersist	Before EM persist() method
PostPersist	After database insert
PreRemove	Before EM remove() method
PostRemove	After database delete
PreUpdate	Before database update
PostUpdate	After database update
PostLoad	After load from database

Callback method in entity class:

```
@PrePersist
void validateCreate() {
  if (getAge() > 130)
    throw new IllegalStateException();
}
```

Callback method in Listener class:

```
@Entity
@EntityListener(com.firm.Monitor.class)
class Person { ... }

class Monitor {
  @PostPersist
  public void newPersonAlert(Person p) {
    emailHR(p.getLastName(),p.getAddress());
```

Embeddable Classes

Embedded objects have no persistent identity of their own. They belong strictly to their owning entity.

Embeddable class:

```
@Embeddable
class DateRange {
  @Basic Date getStartDate();
  @Basic Date getEndDate();
  // ...
}
```

Use of an embeddable class:

Here we are also overriding the default column mapping.

```
@Embedded
@AttributeOverrides({
@AttributeOverride(name="startDate",
  column=@Column(name="EMP_START")),
@AttributeOverride(name="endDate",
  column=@Column(name="EMP_END"))
})
DateRange getEmploymentPeriod() { ... }
```

Inheritance and Polymorphism

There are three table mapping strategies for class hierarchies:

Table per hierarchy:

```
@Table(name="PEOPLE")
@Inheritance(discriminatorValue="General",
  strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="DISCRIM")
class Person { ... }

@Inheritance(discriminatorValue="Emp")
class Employee extends Person { ... }
```

Table per class:

```
@Table(name="PEOPLE")
@Inheritance(
  strategy=InheritanceType.TABLE_PER_CLASS)
class Person { ... }

@Table(name="EMPLOYEES")
class Employee extends Person { ... }
```

Main table joined with subclass tables:

```
@Table(name="PEOPLE")
@Inheritance(discriminatorValue="General",
  strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="DISCRIM")
class Person { ... }

@Table(name="EMPLOYEES")
@Inheritance(discriminatorValue="Emp")
@PrimaryKeyJoinColumn(name="P_ID")
class Employee extends Person { ... }
```

Only the single-table mapping is currently required by the specification. Use of discriminator is optional in the joined inheritance strategy.

Transactions

Transactions may be controlled either through JTA or directly using the resource-local EntityTransaction API.

JTA:

The EntityManager will participate in the current JTA transaction, if any.

Resource-local:

```
EntityTransaction tx = em.getTransaction();
tx.begin();
try {
  // ...
  tx.commit();
}
catch(MyException e) { tx.rollback(); }
```