



Mindtree

Welcome to possible

Java Persistence API

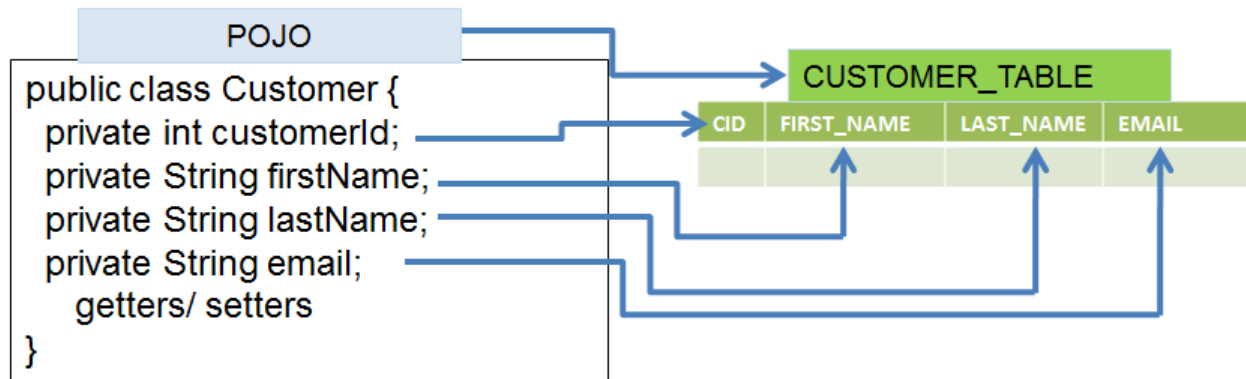
Mindtree

Objectives

- Understand ORM
- Understand Java Persistence API
- Understand Association mapping using JPA
- Understand how to fetch data using JPQL and Criteria API
- Understand how to handle concurrent access

Object Relational Mapping (ORM)

- ORM is a framework used to persist objects present in application to the tables in a relational database using metadata that describes the mapping between objects and the table.
- A Simple mapping will have a class mapped to a table and java properties are mapped to columns of a table



Object Relational Mapping (ORM)

- ORM solution should:
 - Facility for specifying mapping metadata
 - API for CRUD operations on objects
 - API for queries that refer to classes and properties rather than referring Table and columns.
 - Perform Dirty checking
 - Perform Lazy association fetching.
 - Support for Cache.

Object Relational Mapping (ORM)

- Why ORM?
 - Productivity
 - ORM eliminates much of the pushing and pulling code from database and helps you to concentrate on business problem.
 - Maintainability
 - ORM reduces lines of code. Less code means easier to refactor.
 - ORM insulates Java side to database side. One model is unaffected by small change in other model.
 - Vendor independence
 - ORM insulates application from underlying database and SQL Dialect, hence it is easier to develop cross-platform application using ORM.

Object Relational Mapping (ORM)

- Some list of frameworks which are available for Java developers?
 - iBATIS
 - iBATIS is not a ORM, it is a Data Mapper.
 - Good choice when a developer does not have control over SQL database schema.
 - Developer writes SQL for CRUD in XML instead of in a application.
 - After the development team moved from ASF to Google code it is named MyBatis.
 - TopLink
 - TopLink is an ORM developed for Smalltalk, BEA Systems added Java Version. TopLink is now acquired by Oracle Corporation.
 - Hibernate
 - Hibernate was started by Gavin King as an alternative to EJB2.x entity bean.
 - Jboss Inc (Red Hat) provide support to Hibernate.
 - Hibernate scores over other ORM frameworks in:
 - Performance
 - Providing extensibility
 - Excellent mapping API even when used with Native SQL.
 - Rich Key generation implementations.

Java Persistence API

- The Java Persistence API provides Java developers with an object/relational mapping facility for managing relational data in Java applications.
- Java Persistence consists of four areas:
 - The Java Persistence API
 - The query language
 - The Java Persistence Criteria API
 - Object/relational mapping metadata

JPA entities

- An entity is a lightweight persistence domain object. Typically, an entity represents a table in a relational database, and each entity instance corresponds to a row in that table.
- The primary programming artifact of an entity is the entity class.
- The persistent state of an entity is represented through either persistent fields or persistent properties.
- These fields or properties use object/relational mapping annotations to map the entities and entity relationships to the relational data in the underlying data store.

JPA entities

- Requirements for Entity Classes
 - The class must be annotated with the `javax.persistence.Entity` annotation.
 - The class must have a public or protected, no-argument constructor. The class may have other constructors.
 - The class must not be declared final. No methods or persistent instance variables must be declared final.
 - If an entity instance is passed by value as a detached object, such as through a session bean's remote business interface, the class must implement the `Serializable` interface.
 - Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.
 - Persistent instance variables must be declared private, protected, or package-private and can be accessed directly only by the entity class's methods. Clients must access the entity's state through accessor or business methods.

JPA Entities

@Entity

```
public class Person implements Serializable {  
    @Id  
    private Long id;
```

- @Entity is required
- Primary key (@Id) is required
- Must be persisted by EntityManager
- Serializable is recommended

JPA Entities

- Primary Key Generation

1. SEQUENCE indicates that a database sequence should be used to generate the identifier.

```
@Id
```

```
@GeneratedValue(strategy=GenerationType.SEQUENCE)
```

```
private long id;
```

- To use a specific named sequence object, whether it is generated by schema generation or already exists in the database, you must define a sequence generator using a @SequenceGenerator annotation.

```
@Id
```

```
@GeneratedValue(generator="InvSeq")
```

```
@SequenceGenerator(name="InvSeq",sequenceName="INV_SEQ",  
allocationSize=5)
```

```
private long id;
```

JPA Entities

- Primary Key Generation

2. Using Identity Columns

- When using a database that does not support sequences, but does support identity columns (such as SQL Server database),

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
private long id;
```

3. Using a Table

```
@Id
@GeneratedValue(generator="InvTab")
@TableGenerator(name="InvTab", table="ID_GEN",
    pkColumnName="ID_NAME", valueColumnName="ID_VAL",
    pkColumnValue="INV_GEN")
private long id;
```

Table
ID_GEN

ID_NAME	ID_VAL
INV_GEN	<last generated value >

JPA Entities

- Primary Key Generation

4. Using a Default Generation Strategy

Provider will select appropriate strategy

@Id

@GeneratedValue(strategy=GenerationType.AUTO)

private long id;

JPA Entities

- @Table
 - Specifies the primary table for the annotated entity. If no Table annotation is specified for an entity class, the default values apply
- ```
@Entity
@Table(name="ORDER_TABLE")
public class Order { ... }
```
- @Column
    - @Column annotation is used to fine-tune the relational database column for field
- ```
@Id  
@Column(name="ITEM_ID", insertable=false, updatable=false)  
private long id;
```

JPA Entities

- @Temporal

Used with `java.util.Date` or `java.util.Calendar` to determine how value is persisted

Values defined by `TemporalType`:

- ▶ `TemporalType.DATE` (`java.sql.Date`)
- ▶ `TemporalType.TIME` (`java.sql.Time`)
- ▶ `TemporalType.TIMESTAMP` (`java.sql.Timestamp`)

```
...  
@Temporal(value=TemporalType.DATE)  
@Column(name="BIO_DATE")  
private Date bioDate;  
...
```



TBL_ARTIST	
ARTIST_ID	NUMERIC
BIO_DATE	DATE

JPA Entities

- `@Enumerated`
 - Used to determine the strategy for persisting Java enumerated values to database.
 - Values defined by `EnumType`:
 - `EnumType.ORDINAL` (default)
 - `EnumType.STRING`
 - Example:

`@Enumerated(EnumType.STRING)`

`private Rating rating; // RATING will be VARCHAR type in DB`

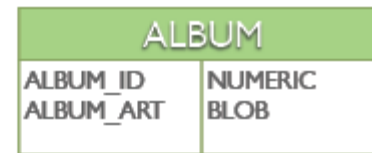
JPA Entities

- @Lob

Used to persist values to BLOB/CLOB fields

@Entity

```
public class Album {  
    @Lob  
    private byte[ ] artwork;  
}
```



JPA Entities

- @Transient

By default, JPA assumes all fields are persistent

Non-persistent fields should be marked as transient or annotated with @Transient

@Entity

```
public class Genre {
```

@Id

```
private Long id; ← persistent
```

```
private transient String value1; ← not persistent
```

@Transient

```
private String value2; ← not persistent
```

```
}
```

JPA Persistence Unit

- A persistence unit defines a set of all entity classes that are managed by EntityManager instances in an application. This set of entity classes represents the data contained within a single data store.
- Persistence units are defined by the persistence.xml configuration file.
- The persistence.xml file sample:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">
  <persistence-unit name="JPAService">
    ...
  </persistence-unit>
</persistence>
```

The persistence.xml sample listing

- Explained in the next slide.

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://
<persistence-unit name="JPAService" transaction-type="RESOURCE_LOCAL"> 1
    <provider>org.hibernate.ejb.HibernatePersistence</provider> 2
    <properties>
        <property name="hibernate.connection.driver_class" 3
            value="com.mysql.jdbc.Driver"/>
        <property name="hibernate.connection.password"
            value="Welcome123"/>
        <property name="hibernate.connection.url"
            value="jdbc:mysql://localhost/happytrip"/>
        <property name="hibernate.connection.username" value="root"/>
        <property name="hibernate.dialect"
            value="org.hibernate.dialect.MySQLDialect"/>
        <property name="hibernate.hbm2ddl.auto" value="update" />
        <property name="hibernate.archive.autodetection" value="class"/>
        <property name="hibernate.show_sql" value="true"/>
    </properties>
</persistence-unit>
</persistence>
```

The persistence.xml sample listing explained

1. A persistence unit is defined by a persistence-unit XML element. The required name attribute (“JPAService” in the example) identifies the persistence unit when instantiating an [EntityManagerFactory](#).
2. The provider element indicates which JPA implementation should be used. [Hibernate, TopLink, ObjectDB, OpenJPA, etc.]
3. JPA 2 defines standard properties for specifying database url, username and password, as demonstrated below:

```
<properties>
```

```
    <property name="javax.persistence.jdbc.url" value=""/>
```

```
    <property name="javax.persistence.jdbc.user" value=""/>
```

```
    <property name="javax.persistence.jdbc.password" value=""/>
```

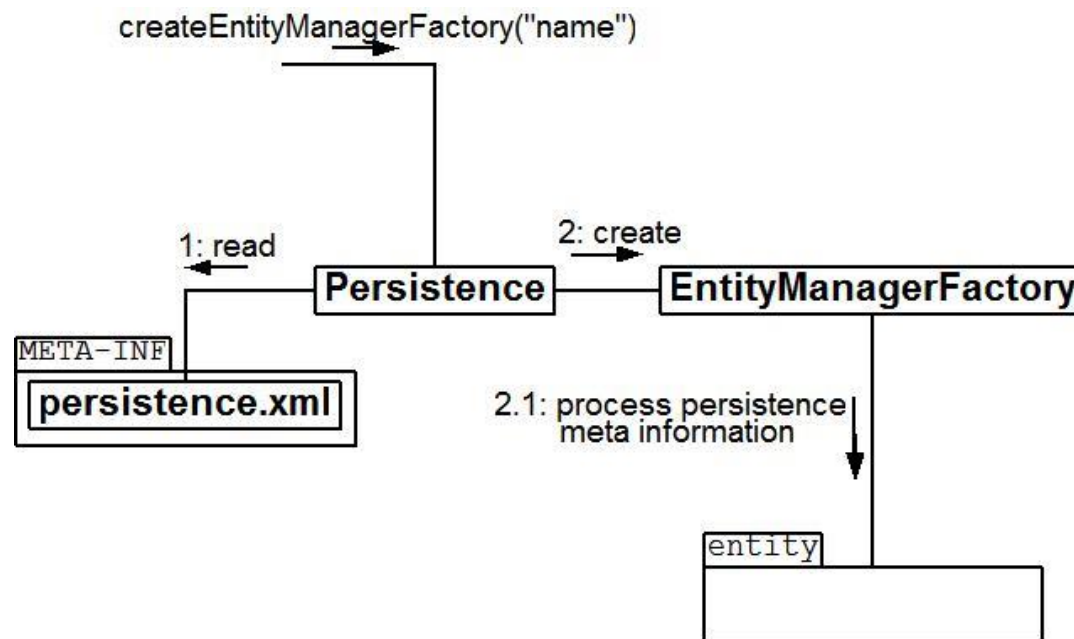
```
</properties>
```

javax.persistence.Persistence

- Entry point for using JPA.
- The method you'll use on this class is `createEntityManagerFactory("JPAService")` to retrieve an entity manager factory with the name "someName".
- This class requires a file called `persistence.xml` to be in the class path under a directory called `META-INF`

EntityManagerFactory

- An instance of this class provides a way to create entity managers.
- The Entity Manager Factory is the in-memory representation of a Persistence Unit
- An instance of this class provides a way to create entity managers.
- The Entity Manager Factory is the in-memory representation of a Persistence Unit.

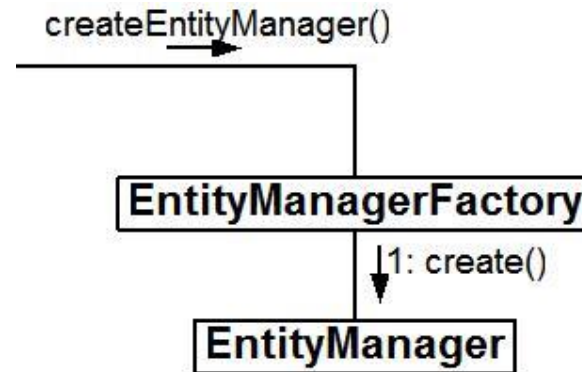


EntityManager in JPA

- **EntityManager** is a class that manages the persistent state(or lifecycle) of an entity.
- There are three main types of Entity Managers defined in JPA.
 - Container Managed Entity Managers
 - When a container of the application(be it a Java EE container or any other custom container like Spring) manages the lifecycle of the Entity Manager, the Entity Manager is said to be Container Managed.
 - Application Managed Entity Managers
 - An Entity Manager that is created not by the container, but actually by the application itself is an application scoped Entity Manager

EntityManager in JPA

- It provides methods for persisting, merging, removing, retrieving and querying objects. It is **not** thread safe so we need one per thread.
- The Entity Manager also serves as a first level cache.
- It maintains changes and then attempts to optimize changes to the database by batching them up when the transaction completes.



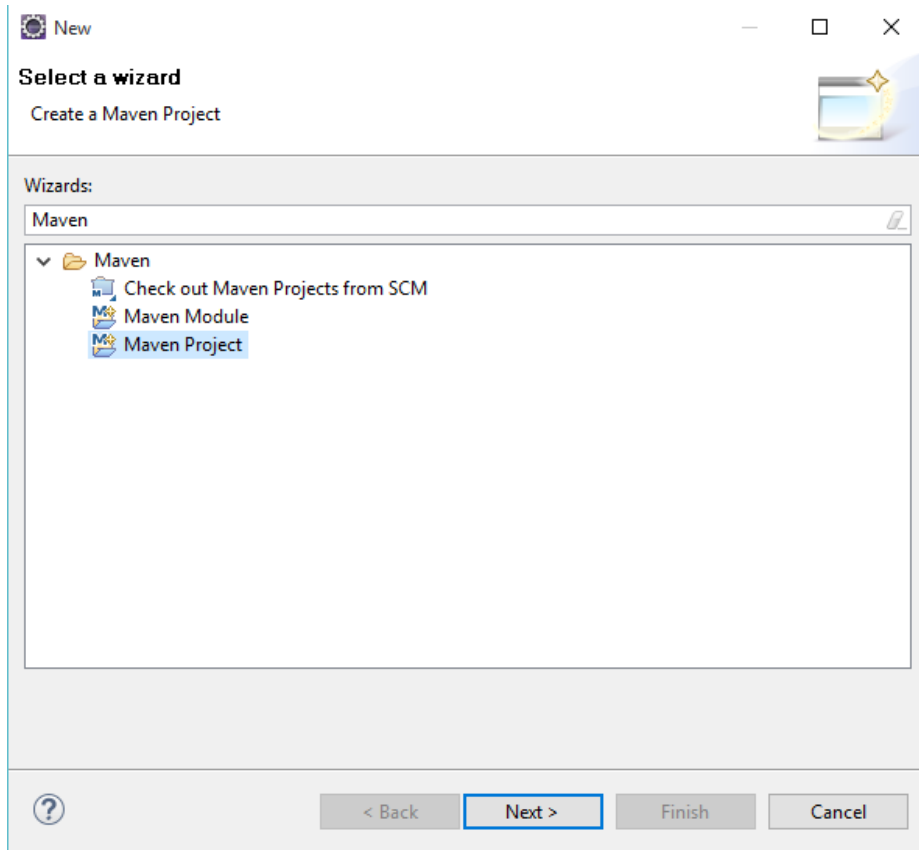
EntityManager in JPA

Controls life-cycle of entities

- `persist()` - insert an entity into the DB
- `remove()` - remove an entity from the DB
- `merge()` - synchronize the state of detached entities
- `refresh()` - reloads state from the database

Steps to create JPA application

1. Create a Maven Project



Steps to create JPA application

2. Select [Create a simple project (skip archetype selection)
3. Specify the below mentioned group id, artifact id and click finish

Artifact	
Group Id:	<input type="text" value="com.mindtree.kalinga"/>
Artifact Id:	<input type="text" value="jpa_basic"/>
Version:	<input type="text" value="0.0.1-SNAPSHOT"/>
Packaging:	<input type="text" value="jar"/>

Steps to create JPA application

- Add the following in pom.xml file

```
<properties>
    <hibernate.version>4.3.5.Final</hibernate.version>
</properties>
<dependencies>
    <!-- Hibernate -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>${hibernate.version}</version>
    </dependency>
    <!-- EntityManager for JPA -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>${hibernate.version}</version>
    </dependency>
    <!-- MySQL DB -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.34</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

Steps to create JPA application

- Create Product entity as shown.
- Generate default and parameterized constructors
- Generate getters/ setters
- Generate hashCode() and equals()

```
package com.mindtree.kalinga.entity;

import javax.persistence.Column;

/**
 *
 * @author Banu Prakash
 * @version 1.0
 * An entity class representing product entity
 */
@Entity
@Table(name="PRODUCTS")
public class Product {

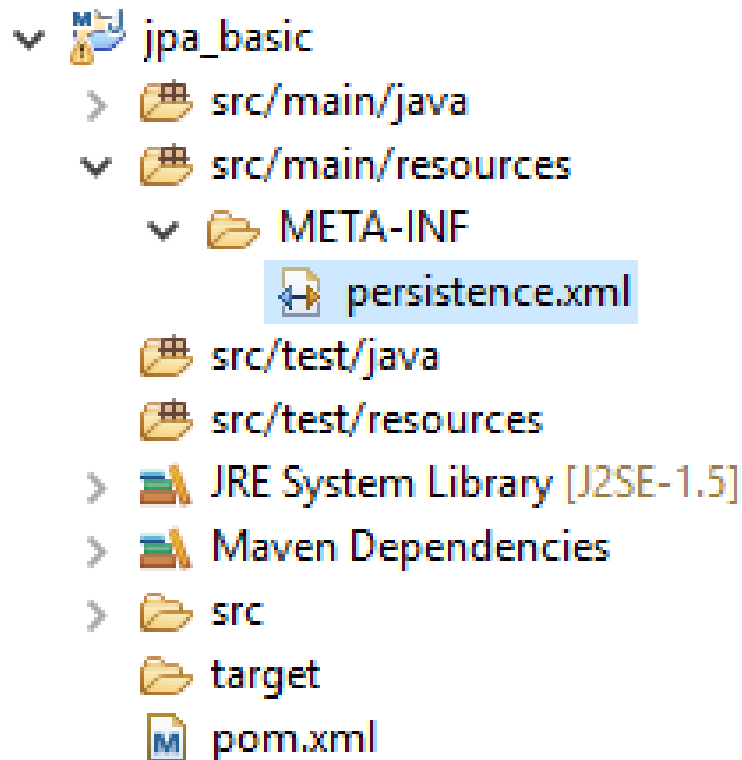
    @Id
    @Column(name="PROD_ID")
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int productId;

    @Column(name="PROD_NAME",length=50,nullable=false)
    private String name;

    @Column(name="AMOUNT",precision=2)
    private double price;
```

Steps to create JPA application

- Create META-INF/persistence.xml



Steps to create JPA application

- The persistence.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">

  <persistence-unit name="JPAService" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <properties>
      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
      <property name="hibernate.connection.url"
        value="jdbc:mysql://localhost:3306/sample_db?createDatabaseIfNotExist=true" />
      <property name="hibernate.connection.driver_class" value="com.mysql.jdbc.Driver" />
      <property name="hibernate.archive.autodetection" value="class" />
      <property name="hibernate.connection.username" value="root" />
      <property name="hibernate.connection.password" value="Welcome123" />
      <property name="hibernate.hbm2ddl.auto" value="update" />
      <property name="hibernate.format_sql" value="true" />
      <property name="hibernate.show_sql" value="true" />
    </properties>
  </persistence-unit>
</persistence>
```

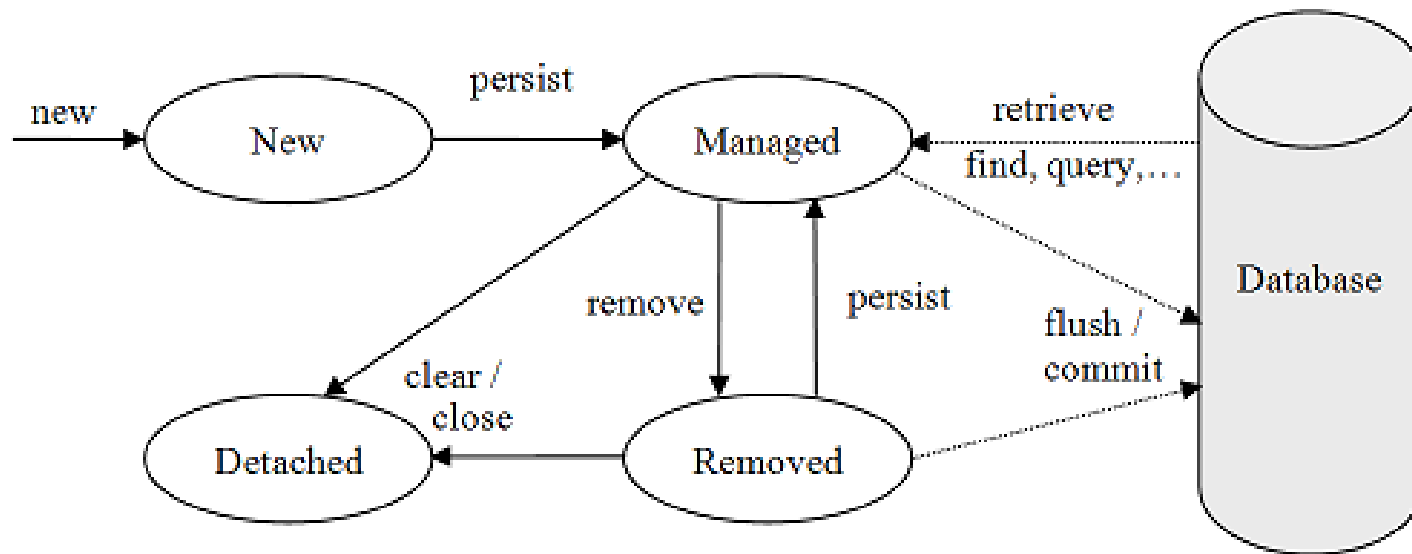

Steps to create JPA application

- Client code

```
public static void main(String[] args) {  
    Product p1 = new Product(0, "Reynolds Pen", 45.55);  
    Product p2 = new Product(0, "Moto G", 12999.00);  
  
    EntityManagerFactory emf = null;  
    EntityManager manager = null;  
  
    try {  
        emf = Persistence.createEntityManagerFactory("JPAService");  
        manager = emf.createEntityManager();  
        manager.getTransaction().begin();  
        manager.persist(p1);  
        manager.persist(p2);  
        manager.getTransaction().commit();  
    } catch (PersistenceException e) {  
        e.printStackTrace();  
    } finally {  
        manager.close();  
        emf.close();  
    }  
}
```

Entity Object Life Cycle

- Explanation in next slide



Entity Object Life Cycle

- When an entity object is initially created its state is New. In this state the object is not yet associated with an EntityManager and has no representation in the database.
- An entity object becomes Managed when it is persisted to the database via an EntityManager's persist method, which must be invoked within an active transaction. On transaction commit, the owning EntityManager stores the new entity object to the database.
- Entity objects retrieved from the database by an EntityManager are also in the Managed state.
- If a managed entity object is modified within an active transaction the change is detected by the owning EntityManager and the update is propagated to the database on transaction commit.
- A managed entity object can also be retrieved from the database and marked for deletion, by using the EntityManager's remove method within an active transaction. The entity object changes its state from Managed to Removed, and is physically deleted from the database during commit.
- Detached, represents entity objects that have been disconnected from the EntityManager.

Find by primary key

- `public <T> T find(Class<T> entityClass, Object primaryKey);`
 - Find by primary key.
 - Search for an entity of the specified class and primary key. If the entity instance is contained in the persistence context, it is returned from there.

```
EntityManagerFactory emf =  
    Persistence.createEntityManagerFactory("JPAService");  
  
EntityManager manager = emf.createEntityManager();  
  
Product prd = manager.find(Product.class, 1);
```

Deleting Entities

- To delete an object from the database, you need to obtain a managed object (usually by retrieval) and invoke the remove method within the context of an active transaction:

```
em.getTransaction().begin();  
em.remove(p); // delete entity  
em.getTransaction().commit();
```

- In the above code, p must be a managed entity object of the EntityManager em. The entity object is marked for deletion by the remove method and is physically deleted from the database when the transaction is committed.



Mindtree

Welcome to possible

JPA association mapping

Mindtree

Association Mapping

- JPA supports all standard relationships
 - One-To-One
 - One-To-Many
 - Many-To-One
 - Many-To-Many
- Supports unidirectional and bidirectional relationships

One-to-One Mapping

- Consider a relationship between Employee and Address.
- Employee has relationship with "Address" table.
- Employee has a Address and its unique and that address can't be assigned to other employees. This relationship is called "one-to-one".
- We can achieve one-to-one mapping by:
 - Sharing of Primary Key: by creating a primary key in each of the relations or tables, where the associated table will contain the primary key value of the associating table, which is called as foreign key.
 - Making Foreign key unique

@OneToOne using @JoinColumn


```
@Entity
public class Employee {
    @Id
    @GeneratedValue
    private int empid;
    private String name;

    @OneToOne
    @JoinColumn(name="ADD_ID")
    // @PrimaryKeyJoinColumn
    Address homeAddress;
}
```

```
@Entity
public class Address {
    @Id
    @GeneratedValue
    private int id;
    private String street;

    @OneToOne(mappedBy="homeAddress")
    Employee employee;
}
```

EMPID	NAME	ADD_ID
1	Ganesh	1



ID	STREET
1	M.G.Road

@OneToOne using @PrimaryKeyJoinColumn

```
@Entity
public class Employee {
    @Id
    @GeneratedValue
    private int empid;
    private String name;

    @OneToOne
    // @JoinColumn(name="ADD_ID")
    @PrimaryKeyJoinColumn ←
    Address homeAddress;
```

EMPID	NAME
1	Ganesh

```
@Entity
public class Address {
    @Id
    @GeneratedValue
    private int id;
    private String street;

    @OneToOne(mappedBy="homeAddress")
    Employee employee;
```

ID	STREET
1	M.G.Road

CASCADE types

- PERSIST: When the owning entity is persisted, all its related data is also persisted.
- MERGE: When a detached entity is merged back to an active persistence context, all its related data is also merged.
- REMOVE: When an entity is removed, all its related data is also removed.
- ALL: All the above applies.

@OneToMany


- @OneToMany defines the *one* side of a one-to-many relationship
- The *mappedBy* element of the annotation defines the object reference used by the *child* entity
- @OrderBy defines an collection ordering required when relationship is retrieved
- The child (many) side will be represented using an implementation of the `java.util.Collection` interface

One-to-Many unidirectional

```
@Entity
public class Trainer {
    @Id
    @GeneratedValue
    @Column
    private Integer id;
    @Column
    private String name;


    @OneToMany
    @JoinColumn(name = "trainer_id")
    private Set<Course> courses;
```

Table: TRAINER

ID 	NAME
1	Banu Prakash

```
@Entity
public class Course {
    @Id
    @GeneratedValue
    private Integer id;
    @Column
    private String name;
```

Table: COURSE

ID 	NAME	TRAINER_ID
2	Java	1
3	Hibernate	1
4	Spring	1

One-to-Many unidirectional

- Persisting without cascade setting

```
EntityManager em = emf.createEntityManager();
Trainer trainer = new Trainer();
trainer.setName("Banu Prakash");
em.persist(trainer);
em.getTransaction().begin();
Course c1 = new Course();
c1.setName("Java");
Course c2 = new Course();
c2.setName("Hibernate");
Course c3 = new Course();
c3.setName("Spring");
trainer.setCourses(new HashSet<Course>());
trainer.getCourses().add(c1);
trainer.getCourses().add(c2);
trainer.getCourses().add(c3);
em.persist(c1);
em.persist(c2);
em.persist(c3);
em.getTransaction().commit();
```

Table: TRAINER



ID 	NAME
1	Banu Prakash

Table: COURSE

ID 	NAME	TRAINER_ID
2	Java	1
3	Hibernate	1
4	Spring	1

One-to-Many unidirectional Without Join Column

- Without `@JoinColumn` a link table “TRAINER_COURSE” is created.

```
@Entity
public class Trainer {
    @Id
    @GeneratedValue
    @Column
    private Integer id;
    @Column
    private String name;

    @OneToMany
    // @JoinColumn(name = "trainer_id")
    private Set<Course> courses;
```

Table: TRAINER


ID 	NAME
1	Banu Prakash

Table: COURSE




ID 	NAME
2	Java
3	Hibernate
4	Spring

Table: TRAINER_COURSE

TRAINER_ID 	COURSES_ID 
1	4
1	2
1	3

One-to-many and Many-to-one [Bidirectional]

```
@Entity public class Customer {  
    @Id String name;
```

Table: CUSTOMER

NAME
Banu Prakash
Ajay

```
    @OneToMany(mappedBy = "customer", cascade = CascadeType.ALL)  
    Set<Order> orders = new HashSet<Order>();
```

```
@Entity  
@Table(name = "OrderTable")  
public class Order {  
    @Id  
    @GeneratedValue(strategy=GenerationType.SEQUENCE)  
    int orderId;
```

```
    @ManyToOne  
    @JoinColumn(name = "customer fk")  
    Customer customer;
```

Table: ORDERTABLE

ORDERID	AMOUNT	ORDERDATE	CUSTOMER_FK
10	1234	2008-02-11 11:58:13.0	Banu Prakash
11	43234	2008-02-11 11:58:13.0	Banu Prakash
12	81223	2008-02-11 11:58:13.0	Ajay
13	8234	2008-02-11 11:58:13.0	Ajay

@ManyToMany

TRACK	
TRACK_ID	NUMERIC

COMPOSER_TRACK	
TRACK_ID	NUMERIC
COMPOSER_ID	NUMERIC

COMPOSER	
COMPOSER_ID	NUMERIC

```
@Entity
public class Track {

    @Id
    @Column(name = "TRACK_ID")
    private Long id;

    @ManyToMany(mappedBy="compositions")
    private Set<Composer> composers
        = new HashSet<Composer>();
}
```

```
@Entity
public class Composer {

    @Id
    @Column(name = "COMPOSER_ID")
    private Long id;

    @ManyToMany
    @JoinTable(name="COMPOSER_TRACK",
        joinColumns = { @JoinColumn(name = "COMPOSER_ID") },
        inverseJoinColumns = { @JoinColumn(name = "TRACK_ID") }
    )
    private Set<Track> compositions;
}
```

Entity Inheritance Mapping Strategies

- The following mapping strategies are used to map the entity data to the underlying database:
 - A single table per class hierarchy
 - A table per concrete entity class
 - A “join” strategy, whereby fields or properties that are specific to a subclass are mapped to a different table than the fields or properties that are common to the parent class

The Single Table per Class Hierarchy Strategy

- With this strategy, which corresponds to the default `InheritanceType.SINGLE_TABLE`, all classes in the hierarchy are mapped to a single table in the database. This table has a discriminator column containing a value that identifies the subclass to which the instance represented by the row belongs.
- The “`@Inheritance`” indicates to the JPA which model of hierarchy persistence we will use.
- The “`@DiscriminatorColumn`” it is used to store a column name that will describe to which column that data line belongs to.

The Single Table per Class Hierarchy Strategy

- Example shown below creates a separate column “FROM_CLASS” which will have values “CAR_BEETLE” or “CAR_FERRARI” based on which entity is persisted

```
@Entity
@Table(name="CAR")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="FROM_CLASS", discriminatorType=DiscriminatorType.STRING)
public abstract class Car {
    ...
}
```

```
@Entity
@DiscriminatorValue(value="CAR_BEETLE")
public class Beetle extends Car {
    ...
}
```

```
@Entity
@DiscriminatorValue(value="CAR_FERRARI")
public class Ferrari extends Car {
    ....
}
```

TABLE_PER_CLASS - Table Per Concrete Class

- The TABLE_PER_CLASS strategy does not require columns to be made nullable, and results in a database schema that is relatively simple to understand. As a result it is also easy to inspect or modify manually.
- A downside is that polymorphically loading entities requires a UNION of all the mapped tables, which may impact performance. Finally, the duplication of column corresponding to superclass fields causes the database design to not be normalized. This makes it hard to perform aggregate (SQL) queries on the duplicated columns. As such this strategy is best suited to wide, but not deep, inheritance hierarchies in which the superclass fields are not ones you want to query on.

TABLE_PER_CLASS - Table Per Concrete Class

- Example

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Person {
```

```
@Entity
@Table(name="EMPLOYEE")
public class Employee extends Person {
```

FETCH types

Eager fetching	<p>Also know as eager loading, this is the default option, the entity-manager will attempt to retrieve all of the entity field data when the find method is invoked.</p> <p>When eagerly fetching the EntityManager will use a JOIN in the SELECT statement to retrieve the entity.</p>
Lazy fetching	<p>Lazy fetching supports more than one mechanism</p> <p>@Basic - specifying a column with this annotation means that the data will only be load the first time it is accessed</p> <p>When lazy fetching the EntityManager would use separate SELECT statements and thus is less efficient than eager loading, when using entity relationships.</p>

JP-QL: The Object Query Language

- The Java Persistence Query Language (JP-QL) has been heavily inspired by HQL, the native Hibernate Query Language.
- Both JP-QL and HQL are therefore very close to SQL, but portable and independent of the database schema.
- HQL is a strict superset of JP-QL and you use the same query API for both types of queries
- **Case Sensitivity**
 - Queries are case-insensitive, except for names of Java classes and properties. So SeLeCT is the same as sELEct is the same as SELECT but org.hibernate.eg.FOO is not org.hibernate.eg.Foo and foo.barSet is not foo.BARSET.

Java Persistence Query Language (JP-QL)

TVEHICLE table contents

VEHICLE_ID	REG_NO	DAILY_RENTAL	FUEL_TYPE	MILEAGE	MANUFACTURER	DESCRIPTION	CATEGORY_ID
1	KA-04-Z-1234	1200.99	Diesel	14	Toyota	Innova, 6 Seater	1
2	AP-08-XY-9822	800.88	Diesel	21	Tata	Indica, 4 Seater	1
3	JK-02-AA-120	450	Petrol	16	Maruthi	Omni Van , All in One	1
4	TN-09-EF-3411	3200	Diesel	5	Ashok Leyland	Luxury Bus	3
5	KA-08-AE-672	4000	Diesel	6	Tata	Heavy Duty Truck	2
6	KA-02-GH-788	900	Petrol	15	Hyundai	Hyndai Accent	1

com.mindtree.entity.Vehicle

```
-vehicleId: Integer  
+registrationNumber: String  
+name: String  
+manufacturer: String  
+description: String  
+mileagePerLiter: int  
+fuelType: String  
+dailyRentalAmount: double  
+category: Category
```

Description	SQL	JP-QL
Retrieve all vehicles	select * from TVEHICLE	Select v from Vehicle v
Retrieve all Petrol vehicles	select * from TVEHICLE where FUEL_TYPE = 'Petrol'	Select v from Vehicle v where v.fuelType = 'Petrol'
Retrieve selective columns (scalar values)	select REG_NO,MILEAGE from TVEHICLE	select registrationNumber, mileagePerLiter from Vehicle
Retrieve vehicles of a category	select * from TVEHILCE where CATEGORY_ID = 1	Select v from Vehicle v where v.category = 1

Java Persistence Query Language (JP-QL)

- JP-QL queries are represented with an instance of `javax.persistence.Query`.
- The Query interface offers methods for parameter binding, result set handling, and for the execution of the actual query.
- Query is obtained using the current EntityManager
- Query is usually executed by invoking a `getResultList()` method.
- SELECT Syntax :
 - `SELECT [<result>] [FROM <candidate-class(es)>]`
 - `[WHERE <filter>] [GROUP BY <grouping>] [HAVING <having>]`
 - `[ORDER BY <ordering>]`

Java Persistence Query Language (JP-QL)

- Example:

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

javax.persistence.Query query = em
    .createQuery("select c from Customer c");

List<Customer> customers = query.getResultList();

for (Customer customer : customers) {
    System.out.println(customer.getCustomerName() + ", "
        + customer.getCity());
}
```

Java Persistence Query Language (JP-QL)

- Named parameters are query parameters that are prefixed with a colon (:). Named parameters in a query are bound to an argument by the following method:
 - `javax.persistence.Query.setParameter(String name, Object value)`

```
String strQuery = "select c from Customer c "  
    + "where c.contactTitle =:title and c.city = :cty" ;  
javax.persistence.Query query = em  
    .createQuery(strQuery);  
query.setParameter("title", "owner");  
query.setParameter("cty", "Seattle");  
  
List<Customer> customers = query.getResultList();
```

Java Persistence Query Language (JP-QL)

- **Positional Parameters in Queries.**
 - You may use positional parameters instead of named parameters in queries.
 - Positional parameters are prefixed with a question mark (?) followed the numeric position of the parameter in the query.
 - The `Query.setParameter(integer position, Object value)` method is used to set the parameter values.

```
String strQuery = "select c from Customer c "  
    + "where c.contactTitle =?1 and c.city = ?2" ;  
javax.persistence.Query query = em  
    .createQuery(strQuery);  
query.setParameter(1, "owner");  
query.setParameter(2, "Seattle");
```

Java Persistence Query Language (JP-QL)

- Range of Results:
 - Query `setFirstResult(int startPosition)`
 - Set the position of the first result to retrieve.
 - Parameters: `startPosition` - the start position of the first result, numbered from 0
- Query `setMaxResults(int maxResult)`
 - Set the maximum number of results to retrieve.

```
EntityManager em = emf.createEntityManager();
```

```
String strQuery = "select c from Customer c ";  
javax.persistence.Query query = em.createQuery(strQuery);  
query.setFirstResult(0);  
query.setMaxResults(5);
```

Java Persistence Query Language (JP-QL)

- Named Queries
 - The `createNamedQuery` method is used to create static queries, or queries that are defined in metadata by using the `javax.persistence.NamedQuery` annotation.

```
@Entity
@Table(name="customers")
@NamedQueries({
    @NamedQuery(name="ParisCustomers",
        query = "select c from Customer c where c.city = 'Paris'"),
    @NamedQuery(name="FetchSalesManagers",
        query = "select c from Customer c where c.contactTitle = 'Sales Manager'")
})
public class Customer {
```

```
    javax.persistence.Query query = em.createNamedQuery("FetchSalesManagers");
    List<Customer> customers = query.getResultList();
```

JP-QL joins

- Joins illustrated using Product and Category

```
@Entity
@Table(name="Categories")
public class Category {
    @Id
    @Column(name = "CATEGORY_ID")
    private int categoryId;

    @Column(name = "CATEGORY_NAME")
    private String categoryName;
}
```

```
@Entity
@Table(name="products")
public class Product {
    @Id
    @Column(name = "PRODUCT_ID")
    private int productId;

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

    @Column(name = "UNIT_PRICE")
    private double unitPrice;

    @ManyToOne
    @JoinColumn(name = "CATEGORY_ID")
    private Category category;
}
```

CATEGORY_ID	CATEGORY_NAME
1	Beverages
2	Condiments
3	Confections
4	Dairy Products
5	Grains/Cereals
6	Meat/Poultry
7	Produce
8	Seafood

PRODUCT_ID	PRODUCT_NAME	UNIT_PRICE	CATEGORY_ID
1	Chai	18.0000	1
2	Chang	19.0000	1
3	Aniseed Syrup	10.0000	2
4	Chef Anton's Cajun Seasoning	22.0000	2
5	Chef Anton's Gumbo Mix	21.3500	2
6	Grandma's Boysenberry Spread	25.0000	2
7	Uncle Bob's Organic Dried Pears	30.0000	7
8	Northwoods Cranberry Sauce	40.0000	2
9	Mishi Kobe Niku	97.0000	6
10	Ikura	31.0000	8
11	Queso Cabrales	21.0000	4
12	Queso Manchego La Pastora	38.0000	4
13	Konbu	6.0000	8
14	Tofu	23.2500	7
15	Genen Shoyu	15.5000	2
16	Pavlova	17.4500	3

JP-QL joins

- Inner Join: Inner join retrieves result by combining column values of two tables based upon the join-predicate.

```
String strQuery = "select p,c from Product p inner join p.category c";  
javax.persistence.Query query = em.createQuery(strQuery);
```

```
List<Object[]> list = query.getResultList();
```

```
for(int i= 0 ; i < list.size(); i++) {  
    Object[] objects = list.get(i);  
    if(objects[0] != null) {  
        Product p = (Product)objects[0];  
        System.out.print(p.getProductName() + ", ");  
    }  
    if(objects[1] != null) {  
        Category c = (Category)objects[1];  
        System.out.println(c.getCategoryName());  
    }  
}
```

First entity will be
Product and second
entity will be
Category

JP-QL joins

- Right Outer join
 - A right outer join (or right join) closely resembles a left outer join, except with the treatment of the tables reversed. Every row from the "right" table will appear in the joined table at least once.

```
String strQuery = "select p,c from Product p right outer join p.category c";  
javax.persistence.Query query = em.createQuery(strQuery);
```

JP-QL Sub Queries

- Fetch all products whose UNIT_PRICE is above average UNIT_PRICE.

```
EntityManager em = emf.createEntityManager();
String strQuery = "select p from Product p "
    + " where p.unitPrice >= (select avg(unitPrice) from Product)";
javax.persistence.Query query = em.createQuery(strQuery);

List<Product> list = query.getResultList();
for(Product p : list) {
    System.out.println(p.getProductName() + "," + p.getUnitPrice());
}
```

- Fetch only seafood whose UNIT_PRICE is above average UNIT_PRICE

```
String strQuery = "select p from Product p "
    + " where p.unitPrice >= (select avg(unitPrice) from Product)"
    + " and p.category.categoryName = 'Seafood'";
```

Criteria API to Create Queries

- The Criteria API is used to define queries for entities and their persistent state by creating query-defining objects.
- Criteria queries are written using Java programming language APIs, are typesafe, and are portable.
- Criteria API work regardless of the underlying data store
- The Criteria API and JPQL are closely related and are designed to allow similar operations in their queries

Criteria API

- The basic semantics of a Criteria query consists of a
 - SELECT clause,
 - FROM clause,
 - and an optional WHERE clause

Criteria queries set these clauses by using Java programming language objects, so the query can be created in a typesafe manner.

Criteria API

- Creating a Criteria Query
 - The `javax.persistence.criteria.CriteriaBuilder` interface is used to construct
 - Criteria queries
 - Selections
 - Expressions
 - Predicates
 - Ordering
- To obtain an instance of the `CriteriaBuilder` interface, call the `getCriteriaBuilder` method on either an `EntityManager` or an `EntityManagerFactory` instance.

```
EntityManager em = ...;
```

```
CriteriaBuilder cb = em.getCriteriaBuilder();
```

Criteria API

- `javax.persistence.criteria.CriteriaQuery`
 - `CriteriaQuery` objects define a particular query that will navigate over one or more entities.
 - Obtain `CriteriaQuery` instances by calling one of the `CriteriaBuilder.createQuery` methods.
 - For creating typesafe queries, call the `CriteriaBuilder.createQuery` method as follows:

```
CriteriaQuery<Product> cq = cb.createQuery(Product.class);
```

Criteria API

- Query Roots
 - For a particular CriteriaQuery object, the root entity of the query, from which all navigation originates, is called the query root.
 - It is similar to the FROM clause in a JPQL query.
- Create the query root by calling the from method on the CriteriaQuery instance.
- The following code sets the query root to the Product entity:
- `CriteriaQuery< Product > cq = cb.createQuery(Product.class);`
- `Root< Product > pet = cq.from(Product.class);`

Criteria API

- Code Snippet:

```
CriteriaBuilder builder = emf.getCriteriaBuilder();
CriteriaQuery<Product> criteria = builder.createQuery(Product.class);
Root<Product> productRoot = criteria.from(Product.class);
criteria.select(productRoot);
List<Product> list = em.createQuery(criteria).getResultList();
```

Criteria API

- Restricting Criteria Query Results
 - The results of a query can be restricted on the CriteriaQuery object according to conditions set by calling the CriteriaQuery.where method. Calling the where method is analogous to setting the WHERE clause in a JPQL query.

Criteria API

- Restricting Criteria Query Results
 - **Using the Metamodel API to Model Entity Classes.**
 - The Metamodel API is used to create a metamodel of the managed entities in a particular persistence unit. For each entity class in a particular package, a metamodel class is created with a trailing underscore and with attributes that correspond to the persistent fields or properties of the entity class.

```
@StaticMetamodel(Product.class)
public class Product_ {
    public static volatile SingularAttribute<Product, Integer> productId;
    public static volatile SingularAttribute<Product, String> productName;
    public static volatile SingularAttribute<Product, Double> unitPrice;
    public static volatile SingularAttribute<Product, Category> category;
}
```

Criteria API

- Restricting Criteria Query Results
 - Populate the "where" method of the CriteriaQuery with "Predicate" objects.

```
EntityManager em = emf.createEntityManager();
CriteriaBuilder builder = emf.getCriteriaBuilder();
CriteriaQuery<Product> criteria = builder.createQuery(Product.class);
Root<Product> productRoot = criteria.from(Product.class);
criteria.select(productRoot);
Predicate predicate = builder.ge(productRoot.get(Product_.unitPrice), 100);
criteria.where(predicate);
List<Product> list = em.createQuery(criteria).getResultList();
```

Criteria API

- Conditional Methods in the CriteriaBuilder Interface

Conditional Method	Description
<code>equal</code>	Tests whether two expressions are equal
<code>notEqual</code>	Tests whether two expressions are not equal
<code>gt</code>	Tests whether the first numeric expression is greater than the second numeric expression
<code>ge</code>	Tests whether the first numeric expression is greater than or equal to the second numeric expression
<code>lt</code>	Tests whether the first numeric expression is less than the second numeric expression
<code>le</code>	Tests whether the first numeric expression is less than or equal to the second numeric expression
<code>between</code>	Tests whether the first expression is between the second and third expression in value
<code>like</code>	Tests whether the expression matches a given pattern

Criteria API

- Examples:

The following code uses the `CriteriaBuilder.gt` method:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);  
Root<Pet> pet = cq.from(Pet.class);  
Date someDate = new Date(...);  
cq.where(cb.gt(pet.get(Pet_.birthday), date));
```

The following code uses the `CriteriaBuilder.between` method:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);  
Root<Pet> pet = cq.from(Pet.class);  
Date firstDate = new Date(...);  
Date secondDate = new Date(...);  
cq.where(cb.between(pet.get(Pet_.birthday), firstDate, secondDate));
```

Criteria API

- Compound Predicate Methods in the CriteriaBuilder Interface
- Examples:

The following code shows the use of compound predicates in queries:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);  
Root<Pet> pet = cq.from(Pet.class);  
cq.where(cb.equal(pet.get(Pet_.name), "Fido")  
        .and(cb.equal(pet.get(Pet_.color), "brown")));
```

Native query

- You may also express queries in the native SQL dialect of your database. This is useful if you want to utilize database specific features

```
EntityManager em = emf.createEntityManager();
String strQuery = "select product_id,product_name from products";
javax.persistence.Query query = em.createNativeQuery(strQuery);
List<Object[]> list = query.getResultList();
System.out.println(list);
for(Object[] data : list) {
    System.out.println(data[0] + "," + data[1] );
}
```


JPA Exceptions

- JPA exceptions are represented by a hierarchy of unchecked exceptions

```
java.lang.RuntimeException
```

```
    javax.persistence.PersistenceException
```

```
        javax.persistence.EntityExistsException
```

```
        javax.persistence.EntityNotFoundException
```

```
        javax.persistence.LockTimeoutException
```

```
        javax.persistence.NonUniqueResultException
```

```
        javax.persistence.NoResultException
```

```
        javax.persistence.OptimisticLockException
```

```
        javax.persistence.PessimisticLockException
```

```
        javax.persistence.QueryTimeoutException
```

```
        javax.persistence.RollbackException
```

```
        javax.persistence.TransactionRequiredException
```

EntityExistsException

- Thrown by the persistence provider when `EntityManager.persist(Object)` is called and the entity already exists. The current transaction, if one is active, will be marked for rollback.
- The current transaction, if one is active, will be marked for rollback.

EntityNotFoundException

- Thrown by the persistence provider when an entity reference obtained by `EntityManager.getReference` is accessed but the entity does not exist.
- Thrown when `EntityManager.refresh` is called and the object no longer exists in the database.
- Thrown when `EntityManager.lock` is used with pessimistic locking is used and the entity no longer exists in the database.

LockTimeoutException

- Thrown by the persistence provider when an pessimistic locking conflict occurs that does not result in transaction rollback.
- This exception may be thrown as part of an API call, at, flush or at commit time.
- The current transaction, if one is active, will be not be marked for rollback.

NonUniqueResultException and NoResultException

- **NonUniqueResultException**

- Thrown by the persistence provider when `Query.getSingleResult()` or `TypedQuery.getSingleResult()` is executed on a query and there is more than one result from the query.

- **NoResultException**

- Thrown by the persistence provider when `Query.getSingleResult()` or `TypedQuery.getSingleResult()` is executed on a query and there is no result to return.
- These exception will not cause the current transaction, if one is active, to be marked for rollback.

LockException

- **OptimisticLockException**
- Thrown by the persistence provider when an optimistic locking conflict occurs. This exception may be thrown as part of an API call, a flush or at commit time.
- **PessimisticLockException**
- Thrown by the persistence provider when an pessimistic locking conflict occurs. This exception may be thrown as part of an API call, a flush or at commit time

JPA 2.0 Concurrency and locking

- **Optimistic Concurrency**
- In JPA for Optimistic locking you annotate an attribute with @Version as shown below:
- public class Employee {

 @ID int id;

 @Version int version;

The Version attribute will be incremented with a successful commit.

The Version attribute can be an int, short, long, or timestamp.

This results in SQL like the following:

“UPDATE Employee SET ..., version = version + 1

WHERE id = ? AND version = readVersion”

- The advantages of optimistic locking are that no database locks are held which can give better scalability. The disadvantages are that the user or application must refresh and retry failed updates.


JPA Entity Locking APIs

- **OPTIMISTIC (READ) LockMode**

- Version checking on the employee attribute would not throw an exception in this example since it was the dep Version attribute that was updated in transaction1.
- In this example the employee change should not commit if the department was changed after reading, so an OPTIMISTIC lock is used : `em.lock(dep, OPTIMISTIC)`.
- This will cause a version check on the dep Entity before committing transaction2 which will throw an `OptimisticLockException` because the dep version attribute is higher than when dep was read, causing the transaction to roll back.

<pre>tx1.begin(); dep = findDepartment(depId); //dep's original name is //"Eng" dep.setName("MarketEng"); tx1.commit();</pre>	<pre>tx2.begin(); emp= findEmp(eId); dep = emp.getDepartment(); em.lock(dep, OPTIMISTIC); if(dep.getName.equals("Eng") emp.raiseByTenPercent(); //Check dep version in db tx2.commit(); //e1 gets the raise he does //not deserve //Transaction rolls back</pre>
--	--

Time



Pessimistic Concurrency

- Pessimistic concurrency locks the database row when data is read, this is the equivalent of a (SELECT . . . FOR UPDATE [NOWAIT]) .
- Pessimistic locking ensures that transactions do not update the same entity at the same time, which can simplify application code, but it limits concurrent access to the data which can cause bad scalability and may cause deadlocks. Pessimistic locking is better for applications with a higher risk of contention among concurrent transactions.

```
//Read and lock:  
Account acct = em.find(Account.class,  
    acctId, PESSIMISTIC);  
// Decide to withdraw $100 (already locked)  
int balance = acct.getBalance();  
acct.setBalance(balance - 100);
```

Locks longer,
could cause
bottlenecks,
deadlock

```
//Read then lock:  
Account acct = em.find(Account.class, acctId);  
// Decide to withdraw $100 so lock it for update  
em.lock(acct, PESSIMISTIC);  
int balance = acct.getBalance();  
acct.setBalance(balance - 100);
```

Lock after read, risk
stale, could cause
**OptimisticLock
Exception**



Mindtree

Welcome to possible



Contact Person

Contact_ contact@mindtree.com

+Country code-Phone

www.mindtree.com