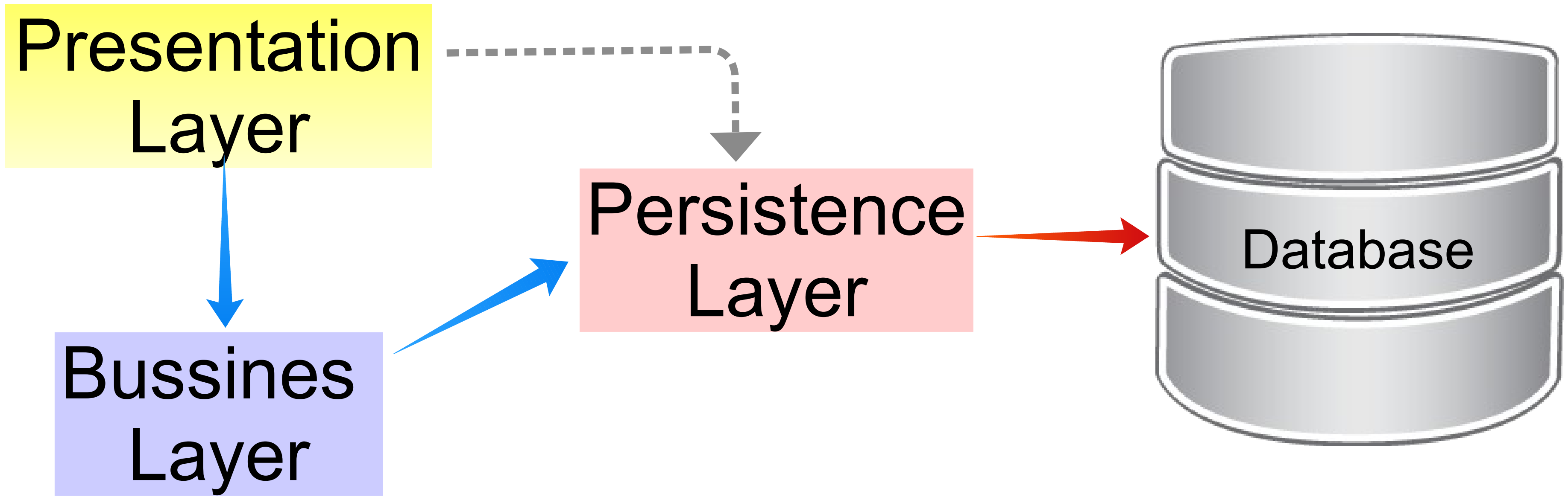
Java EE™

Java Technologies
Java Persistence API (JPA)

Persistence Layer

The Persistence Layer is used by an application in order to persist its state, that is to store and retrieve information using some sort of database management system.



Different Perspectives About Data

SQL Guy

- Relational Level

```
CREATE TABLE persons (
  id integer NOT NULL,
  name varchar(50) NOT NULL,
  salary float,
  PRIMARY KEY(id));
INSERT INTO persons (id, name) VALUES (1, ’John Doe’);
UPDATE persons SET salary=2000 WHERE id=1;
```

- Object Oriented Level

Programmer

```
public class Person {
  public String name;
  public float salary;
  public Person(String name) { ... }
}
  Person p = new Person("John Doe");
  somePersistenceLayer.save(p);
  p.setSalary(2000);
  somePersistenceLayer.update(p);
```

JDBC

an “SQL” API for Programmers

```
// Specify the driver
Class.forName("org.postgresql.Driver");

// Create a connection to the database
Connection con = DriverManager.getConnection(
  "jdbc:postgresql://localhost/demo", "dba", "sql");

// Create an SQL statement
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("select id, name from persons");

// Iterate through the ResultSet (SQL Cursor)
while (rs.next()) {
  int id = rs.getInt("id"));
  String nume = rs.getString("name"));
  System.out.println(id + ". " + name);
}
rs.close();      // Don't forget to close the ResultSet!
stmt.close();    // Don't forget to close the Statement!
con.close();     // Don't forget to close the Connection!!!
```

Transactions and Managers

- A *transaction* is a logical unit of work that is independently executed in the context of a resource (data) management system.
  - Atomic, Consistent, Isolated, Durable (ACID).
- A *transaction manager* is a part of an application that controls the coordination of transactions over one or more resources.
  - Responsible for creating transaction objects and managing their durability and atomicity.
  - Demarcation, Controlling the Transaction Context, Coordination, Recovery From Failure

JTA (Java Transaction API)

- Allows to demarcate transactions in a manner that is <u>independent of the transaction manager implementation</u>.
- A JTA transaction may span updates to multiple databases from different vendors, in a consistent and coordinated manner. It does not support nested transactions.
- To demarcate a JTA transaction, "someone" has to invoke the *begin*, *commit*, and *rollback* methods of either:
  - *javax.transaction.**UserTransaction***

    → explicitly management of transaction boundaries
  - *javax.transaction.**Transaction***

    → the contract between the transaction manager and the parties involved in a distributed transaction namely: resource manager, application, and application server

javax.transaction

- Each active transaction is represented by a transaction object, which implements the interface javax.transaction.Transaction. **This object keeps track of its own status, indicating if it is active, if it has been committed or rolled back, and so on**. The transaction also keeps track of the resources that were enlisted with it, such as JDBC connections. A transaction object lasts for the space of exactly one transaction—when the transaction begins, a new transaction object is created. After the transaction ends, the transaction object is discarded.
- A **transaction is usually associated with a thread**, which is how a transaction appears to be carried along throughout the duration of a request or other sequence of actions. Only one transaction can be associated with a thread at any one time. This leads to the notion of the current transaction, which is the transaction that is currently associated with a thread.
- **An application server can have many active transactions at once, each associated with a different thread running in the server**. A central service, called the Transaction Manager, is responsible for keeping track of all these transactions, and for remembering which transaction is associated with which thread. When a transaction is started, the Transaction Manager associates it with the appropriate thread. When a transaction ends, the Transaction Manager dissociates it from its thread.

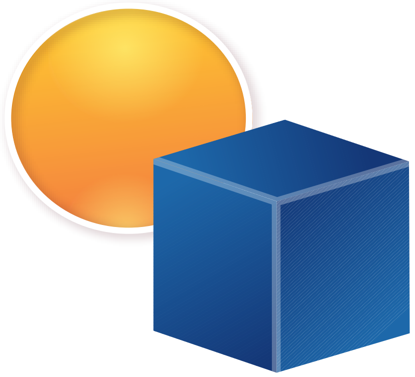
Object-Relational Mapping (ORM)

- **Accesing relational data using OO paradigm**
- **Objects ↔ Mapping Layer ↔ Relations**
- Advantages:
  - Simplified development using automated conversions between objects and tables. No more SQL in the Java code.
  - Less code compared to embedded SQL and stored procedures
  - Superior performance if object caching is used properly
  - Applications are easier to maintain
- Disadvantages:
  - the additional layer may slow execution *sometimes*
  - defining the mapping may be difficult *sometimes*

“Impedance Mismatch”

Graph of objects vs Relations (sets of tuples)

- **Granularity**
  - How many classes vs *How many tables*
- **Subtypes**
  - Inheritance vs *None*
- **Identity**
  - == or equals vs *Primary Keys*
- **Associations**
  - Unidirectional references vs *ForeignKeys*
- **Data Navigation**
  - One object to another vs *Queries*

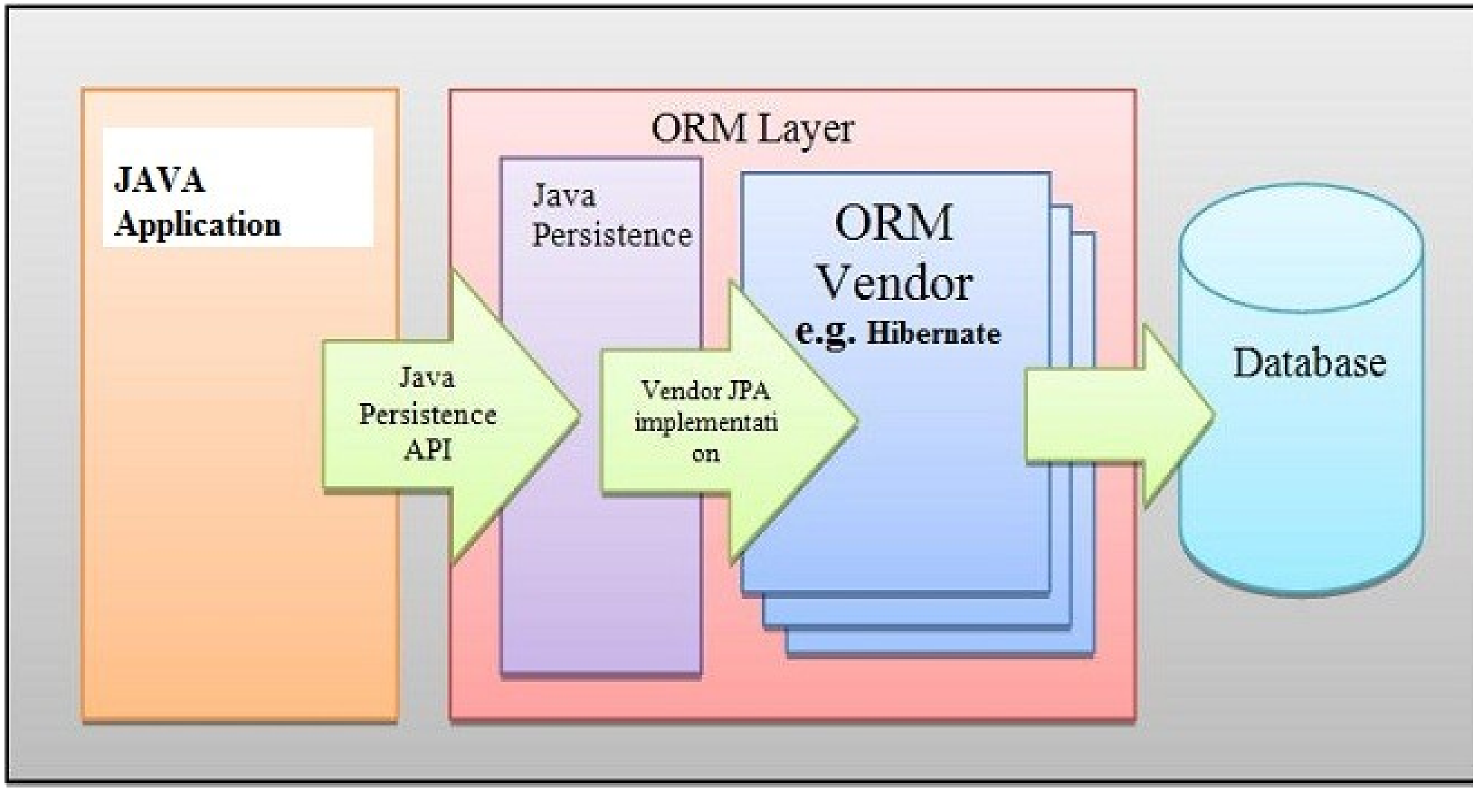


Java Persistence API

- Object/relational mapping **specifications** for managing relational data in Java applications
- Consists of:
  - The Java Persistence **API**
  - Java Persistence Query Language (**JPQL**)
  - The Java Persistence **Criteria API**
  - O/R mapping metadata (**Persistence Annotations**)
- Implemented by:
  - most of the Java ORM producers

What JPA-ORM Should I Use?

It doesn't (shouldn't) matter from the programmer perspective...



You don't like Hibernate <u>for some reason</u> anymore?
Simply <u>replace its libraries</u> with another implementation,
like EclipseLink or OpenJPA for instance.

Entities

- **Entity** = lightweight persistence domain object:
  - an entity class represents a table and
  - an entity instance corresponds to a row in that table.
- **Persistence annotations** are used to map the entities to the relational data.
- **Convention over configuration**

```
@Entity
@Table(name = "persons") //only configure the exceptions
public class Person implements Serializable {
  @Id
  private Integer id;

  private String name;

}
```

Persistence Annotations

```
@Entity
@Table(name = "PERSONS")
public class Person implements Serializable {
  @Id
  @SequenceGenerator(name = "sequence",
                     sequenceName = "persons_id_seq")
  @GeneratedValue(generator = "sequence")
  @Column(name = "PERSON_ID")
  private Integer id;

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

  @JoinColumn(name = "DEPT_ID")
  @ManyToOne
  private Departament departament;
}
```

Persistence Units

- A **persistence unit** defines the set of all entity classes that are managed by an application.
  - ➔ Defined at design time in **persistence.xml**
  - ➔ *javax.persistence.PersistenceUnit*
- This set of entity classes represents the data contained within a single data store.
  - ➔ An application may use multiple persistence units
- A **persistence context** defines a set of entity instances managed at runtime.
  - ➔ *javax.persistence.PersistenceContext*

persistence.xml (Local)

```
<persistence>
  <persistence-unit name="MyDesktopApplicationPU"
                    transaction-type="RESOURCE_LOCAL">

    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <class>myapp.entity.Student</class>
    <class>myapp.entity.Project</class>

    <properties>
      <property name="hibernate.dialect"
                value="org.hibernate.dialect.PostgreSQLDialect"/>
      <property name="hibernate.connection.driver_class"
                value="org.postgresql.Driver"/>
      <property name="hibernate.connection.url"
                value="jdbc:postgresql://localhost/sample"/>
    </properties>

  </persistence-unit>
</persistence>
```

persistence.xml (JTA)

```
<persistence>
  <persistence-unit name="MyWebApplicationPU"
                    transaction-type="JTA">

    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>

    <jta-data-source>jdbc/sample</jta-data-source>

    <exclude-unlisted-classes>false</exclude-unlisted-classes>

    <properties>
      <property name="eclipselink.logging.level" value="FINE"/>
    </properties>

  </persistence-unit>
</persistence>
```

We can use **jta-data-sources** in business method that are “transactional” - have the ability to declaratively control transaction boundaries. For example: EJBs or CDI managed beans.
We’ll study them soon!

persistence.xml (Non JTA)

```
<persistence>
  <persistence-unit name="MyWebApplicationPU"
                    transaction-type="RESOURCE_LOCAL">

    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>

    <non-jta-data-source>jdbc/sample</non-jta-data-source>

    <exclude-unlisted-classes>false</exclude-unlisted-classes>

  </persistence-unit>
</persistence>
```

We use **non-jta-data-sources** in business method that are not “transactional”. We have to *begin* and *commit* (or *rollback)* transactions ourselves.
We’ll use this approach to begin with.

Managing Entities

- Entities are managed by … the **EntityManager.**
- Each EntityManager instance is associated with a **persistence context**: *a set of managed entity instances that exist in a particular data store*.
- The EntityManager defines the methods used to interact with the persistence context:
  - `persist, remove, refresh, find, ...`
- An EntityManager $\leftarrow$ not expensive not thread safe
  is created by …
  an **EntityManagerFactory** $\leftarrow$ expensive thread safe

Creating an *EntityManager*

- **Container-Managed** Entity Managers ✓

```
@PersistenceContext
EntityManager em;
//
@PersistenceUnit
EntityManagerFactory emf;
```

The @PersistenceContext annotation can be used on any CDI bean, EJB, Servlet, Servlet Listener, Servlet Filter, or JSF ManagedBean.

- **Application-Managed** Entity Managers

```
EntityManagerFactory factory =
    Persistence.createEntityManagerFactory(
        "MyApplicationPU", properties);
EntityManager em = factory.createEntityManager();
...
em.close();
...
factory.close();
```

Container-Managed

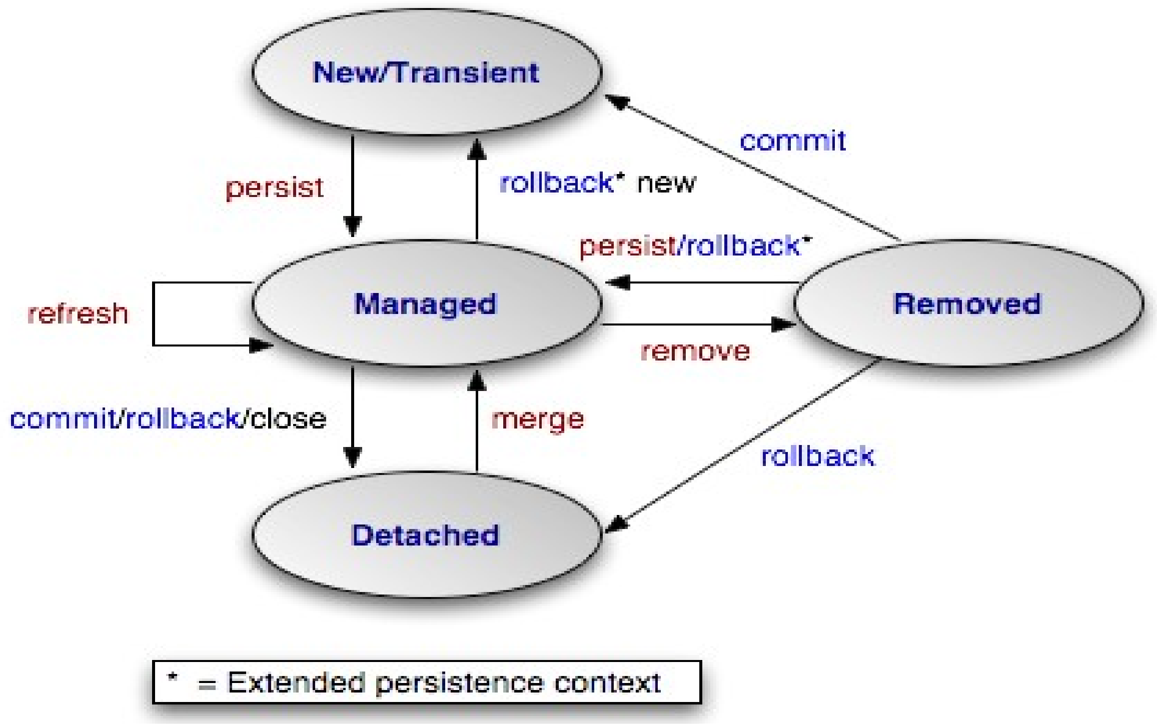
```
public class MyServlet extends HttpServlet {
    @PersistenceContext
    EntityManager em;

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response) {
        EntityTransaction tr = em.getTransaction();
        ...
    }
}
```

EntityManager-per-Request

Interface used to control transactions on resource-local entity managers.

The Lifecycle of an Entity



New entity instances have no persistent identity and are not yet associated with a persistence context.
Managed entity instances have a persistent identity and are associated with a persistence context.
Detached entity instances have a persistent identity and are not currently associated with a persistence context.
Removed entity instances have a persistent identity, are associated with a persistent context, and are scheduled for removal from the data store.

*EntityManager.**persist***

Make an instance managed and persistent.

```
// Get an EntityManager em1

// Start a transaction (we will get rid of this soon!)
em1.getTransaction().begin();

// Create an object
Person duke = new Person("Duke"); //--> duke: NEW
System.out.println("ID:" + duke.getId()); //-->null
System.out.println("Managed:" + em1.contains(duke)); //-->false

//Save the object in the database (SQL INSERT)
//----------------------------------------------
entityManager.persist(duke); //--> duke: MANAGED
//----------------------------------------------

System.out.println("ID:" + duke.getId()); --> 101
System.out.println("Managed:" + em1.contains(duke)); //-->true

// Commit the current transaction (get rid of it,soon)
em1.getTransaction().commit();
```

*EntityManager.**find***

Searches for an entity of the specified class and primary key.
If it is contained in the persistence context, it is returned from there.
Otherwise, it is added to the persistence context.

```
... (the sequence from the previous slide)
em1.getTransaction().commit(); //--> duke: DETACHED from em1

duke.setName("Mickey"); !
em1.close();

// Consider em2 another EntityManager
System.out.println("ID:" + duke.getId()); --> 101
System.out.println("Managed:" + em2.contains(duke)); //-->false

//------------------------------------------------------------
duke = em2.find(Person.class, dukeId); //duke: MANAGED in em2
//------------------------------------------------------------

System.out.println("Managed:" + em2.contains(duke)); //-->true
System.out.println(duke.getName());
```

?

Updating entities

The EntityManager offers no explicit method to save the changes of an entity instance back to the database. The changes are *tracked* by the EM

```
em.getTransaction().begin();

// Get an entity
Person duke = em.find(Person.class, dukeId); // --> MANAGED

//----------------------------------------
duke.setName("The Duke");      // --> SQL UPDATE
//duke is "dirty"
//----------------------------------------

em.getTransaction().commit(); // Duke gets his name changed
```

Automatic Dirty Checking

- An ORM doesn't update the database row of every single persistent object in memory at the end of the unit of work.
- An ORM must have a strategy for detecting which persistent objects have been modified by the application.
- An ORM should be able to detect exactly which properties have been modified so that it's possible to include only the columns that need updating in the SQL UPDATE statement.

*EntityManager.**remove***

Remove the entity instance.

```
em.getTransaction().begin();

// Get an entity
Person duke = em.find(Person.class, dukeId);

//-------------------------------------------------
em.remove(duke); //--> duke: REMOVED --> SQL DELETE
//-------------------------------------------------

em.getTransaction().commit();

System.out.println(duke.getId());
```

*EntityManager.**merge***

Merge the state of an entity into the current persistence context.

CLIENT

```
// Request an object
Record record =
   (Record) server.getObject(id);

...
// Modify the detached object
record.setSomeField("Foo");

...
// Send the object to the server
server.updateObject(record);
```

A **detached** instance is an object that has been persistent, but its context has been closed.

SERVER

```
//returns an object with a given id
public Record getObject(int id) {
  Record record =
    em.find(Record.class, id);
  //record --> MANAGED
  em.close();
  //record --> DETACHED
  return record;
}

//saves the object back to database
public Record updateObject(Object o) {
  em.getTransaction().begin();
  //-----------------------------
  Record merged =
     (Record) em.merge(o);
  merged.setOtherField("bar");
  //-----------------------------
  em.getTransaction().commit();
  em.close();
}
```

Mapping Associations

- **One-to-one**: Each entity instance is related to a single instance of another entity.
- **One-to-many**: An entity instance can be related to multiple instances of the other entities.
- **Many-to-one**: Multiple instances of an entity can be related to a single instance of the other entity.
- **Many-to-many**: The entity instances can be related to multiple instances of each other.

Direction in Entity Relationships

- A **unidirectional** relationship has only an owning side
- A **bidirectional** relationship has both an owning side and an inverse side.
  - The inverse side must refer to its owning side by using the **mappedBy element** of the @OneToOne, @OneToMany, or @ManyToMany annotation. The mappedBy element designates the property that is the owner of the relationship.
  - The many side of many-to-one bidirectional relationships must not define the mappedBy element. The many side is always the owning side of the relationship.
- The owning side of a relationship determines **how the Persistence runtime makes updates to the relationship in the database**.

OneToOne

- One-to-one association that maps a foreign key column (note that this FK column in the database should be constrained unique to simulate one-to-one multiplicity)
  Example: for each order an invoice is created

```
// In Invoice class
@JoinColumn(name = "ORDER_ID",
            referencedColumnName = "ID")
@OneToOne
private Order order;

// In Order class
@OneToOne(mappedBy = "order")
private Invoice invoice;
```

owning side

- The source and target share the same primary key values
- An association table is used to store the link between the 2 entities (a unique constraint has to be defined on each fk to ensure the one to one multiplicity).

Persisting Related Objects

```
@PersistenceContext(unitName="MyApplicationPU")
EntityManager em;
...
        Order order = new Order();
        order.setOrderDate(new java.util.Date());

        Invoice invoice = new Invoice();
        invoice.setOrder(order);
        invoice.setNumber(1);
        invoice.setValue(100);

        em.getTransaction().begin();
        em.persist(order);
        em.persist(invoice);
        em.getTransaction().commit();

        System.out.println(invoice.getOrder().getOrderDate());
        em.close();
        emf.close();
```

What if:

```
 em.persist(invoice);
 em.persist(order);
```

or even:

```
 em.persist(invoice);
```

~~em.persist(order);~~

OneToMany - ManyToOne

Example: each order contains a set of items.

```
// In Order class

@OneToMany( cascade = CascadeType.ALL,
            mappedBy = "order" )
public Set<OrderItem> items;



// In OrderItem class

@ManyToOne
@JoinColumn( name = "ORDER_ID",
             referencedColumnName = "ID"
             nullable = false )
public Order order;
```

owning side

ManyToMany

Example: each supplier offers a set of products, each product may be offered by different suppliers.

```
// In Product class
@JoinTable(name = "SUPPLIERS_PRODUCTS",
  joinColumns = {
     @JoinColumn(name = "PRODUCT_ID",
                 referencedColumnName = "ID")},
  inverseJoinColumns = {
     @JoinColumn(name = "SUPPLIER_ID",
                 referencedColumnName = "ID")})
@ManyToMany
private Set<Supplier> suppliers;
```

owning side

```
// In Supplier class
@ManyToMany(mappedBy = "suppliers")
private Set<Product> products;
```

Mapping Associative Tables

- *“A person may have different administrative functions in various departaments”.*
- ***Person*** class
  - List<Departament> is not good (no function...)
  - List<**PersonDepartament**> departaments;
- ***Departament*** class
  - List<Person> is not good (no function...)
  - List<**PersonDepartament**> persons;
- We need a class to describe the *associative* table:

  persons_departaments (person_id, departament_id, function_id)

The Associative Entity

```
@Entity
@Table(name = "persons_departaments")
public class PersonDepartament implements Serializable {

    @EmbeddedId
    private PersonDepartamentId id;

    @ManyToOne
    @MapsId("personId")
    @JoinColumn(name = "person_id")
    private Person person;

    @ManyToOne
    @MapsId("departamentId")
    @JoinColumn(name = "departament_id")
    private Departament departament;

    @ManyToOne
    @MapsId("functionId")
    @JoinColumn(name = "function_id")
    private Function function;
    …
}
```

Mapping the Composite PK

```
@Embeddable
public class PersonDepartamentId implements Serializable {

    @Column(name = "person_id")
    private Integer personId;

    @Column(name = "departament_id")
    private Integer departamentId;

    @Column(name = "function_id")
    private int functionId;

    …
}
```

@Embeddable specifies a class whose instances are stored as an intrinsic part of an owning entity and share the identity of the entity. Each of the persistent properties or fields of the embedded object is mapped to the database table for the entity.

Mapping Inheritance

- Entities support class inheritance, polymorphic associations, and polymorphic queries. Entity classes can extend non-entity classes, and non-entity classes can extend entity classes. Entity classes can be both abstract and concrete.
- **MappedSuperclass:** abstract not-entity parent class
- **Single Table:** one table per class hierarchy (default)
- **Joined Table:** each class is mapped to its table, common properties are not repeated (join required)
- **Table-Per-Class** – each class is mapped to its table, common properties are repeated (no join required)

Example – MappedSuperclass

Inheritance is defined at class level and not in the entity model.

```
@Entity
@MappedSuperclass
public class Person {
    @Id
    private int id;
    private String name;
    // ...
}

@Entity
@Table(name = "STUDENTS")
public class Student extends Person {
    private int yearOfStudy;
    // ...
}
@Entity
@Table(name = "LECTURERS")
public class Lecturer extends Person {
    private String position;
    // ...
}
```

Example - SingleTable

```
@Entity
@Table(name = "PERSONS")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="person_type",
  discriminatorType = DiscriminatorType.CHAR)
public class Person {
    @Id
    private int id;
    private String name;
    // ...
}

@Entity
@DiscriminatorValue("S")
public class Student extends Person {
    private int yearOfStudy;
    // ...
}
@Entity
@DiscriminatorValue("L")
public class Lecturer extends Person {
    private String position;
    // ...
}
```

Cascade Operations for Entities

- Entities that use relationships often have <u>dependencies on the existence of the other</u> entity in the relationship (*Order* ← *OrderItem*).
- If the parent entity is ... into the persistence context, the related entity will also be ....
  - CascadeType.ALL
  - CascadeType.DETACH
  - CascadeType.MERGE
  - CascadeType.PERSIST
  - CascadeType.REFRESH
  - CascadeType.REMOVE

Synchronizing Entity Data to the Database

- The state of persistent entities is synchronized to the database when the transaction with which the entity is associated **commits**. If a managed entity is in a bidirectional relationship with another managed entity, the data will be persisted, based on the owning side of the relationship.
- To force synchronization of the managed entity to the data store, invoke the **flush** method of the EntityManager instance. If the entity is related to another entity and the relationship annotation has the cascade element set to PERSIST or ALL, the related entity's data will be synchronized with the data store when flush is called.
- If the entity is removed, calling flush will remove the entity data from the data store.
- Use refresh to recreate the state of the instance from the database, using the specified properties, and overwriting changes made to the entity.

Lifecycle Callbacks

- *@PostLoad*
- *@PrePersist, @PostPersist,*
- *@PreUpdate, @PostUpdate*
- *@PreRemove, @PostRemove*

```
public class PersonLogger {
  @PostPersist
  public void logAddition(Object p) {
    getLog().debug ("Added:" + ((Person)p).getName());
  }
  @PreRemove
  public void logDeletion(Object p) {
    getLog().debug ("Terminated:"+((Person)p).getName());
    }
}
```

```
@Entity
@EntityListeners({ PersonLogger.class, ... })
public class Person {
   ...
}
```

JP Query Language (JPQL)

- **Queries for entities** and their persistent state.
- **Portable queries** that work regardless of the underlying data store.
- Uses an **SQL-like syntax**

```
QL_statement ::= select_clause from_clause
   [where_clause]
   [groupby_clause]
   [having_clause]
   [orderby_clause]
```

```
SELECT p FROM Person p WHERE p.name LIKE "%Duke%"
```

Creating Queries

- Dynamic Queries

```
public List<Person> findByName(String name) {
    Query query = entityManager.createQuery(
        "SELECT p FROM Person p WHERE p.name LIKE :personName")
        .setParameter("personName", name)
        .setMaxResults(10);
    return query.getResultList();
}
```

- Static Queries

```
@NamedQuery(name="findById",
    query="SELECT p FROM Person p WHERE p.id = :personId")
)
@Entity
@Table(name="persons")
class Person { ... }
```

- parsed and compiled “early” (at deployment)
- any errors will be detected quickly.

Using a static query:

```
Person p = em.createNamedQuery("findById")
    .setParameter("personId", 1)
    .getSingleResult();
```

The Generated SQL Statements

- persistence.xml

```
<property name="eclipselink.logging.level" value="FINE"/>
```

- **JPQL**: *select e from Person e order by e.name*

  SQL → `SELECT id, name FROM persons ORDER BY name`

- *select e from Person e where size(e.departaments) > 1*

```
→ SELECT t0.id, t0.name FROM persons t0 WHERE (
  (SELECT COUNT(t1.person_id) FROM persons_depts t1
   WHERE (t1.person_id = t0.id)) > 1)
```

Queries That Navigate to Related Entities

- An expression can traverse, or navigate, to related entities. JP-QL *navigates* to related entities, whereas SQL *joins* tables.
- Using JOIN

  -
    ```
    List<Person> persons = entityManager().createQuery(
     "select p from Person p join p.departaments as d "
        + "where d.name = :dept "
        + "order by p.name").
     setParameter("dept", someDeptName).
     getResultList();
    ```
- Using IN

  -
    ```
    select distinct p from Person, in (p.departaments) d
    ```

    *The **p variable** represents the Person entity, and the **d variable** represents the related Departament entity. The declaration for d references the previously declared p variable. The **p.departaments** expression navigates from a Person to its related Departament.*

Native Queries

Going back to good old SQL ...

- Simple SQL queries

```
BigInteger count = (BigInteger) entityManager.createNativeQuery(
  "select count(*) from persons where date_of_birth=:date").
  setParameter("date", birthDate).
  getSingleResult();
```

- SQL queries mapped to entites

```
String sqlQuery = "select * from persons where date_of_birth = ?";
Query q = entityManager.createNativeQuery(sqlQuery, Person.class);
q.setParameter( 1, birthDate);
List<Person> persList = q.getResultList();
```

Aggregate Functions

- count, min, max, avg, sum
- `select` **`count(distinct e.name)`** `from Person e`

-
  ```
  select max(a.number) from Invoice a
     where extract(year from a.date) = 2017;
  ```

-
  ```
  select order, sum(item.qty * item.price) as value
     from Order order
        join order.items item
        group by order
        having value > 100;
  ```

Result:
`List<Object[]>`

JPA Criteria API

- The Context: is the query below valid?

```
String jpql = "select p from Person where p.age > 20";
```

- Define **portable type-and-name safe** queries for entities
- **Pure object oriented** queries
- Define queries **dynamically** via construction of an *object-based CriteriaQuery* instance, rather than *string-based* approach used in JPQL
- Very good performance (translated to native SQL queries)

Example of Criteria Query

```
EntityManager em = ...
//CriteriaBuilder is the factory for CriteriaQuery.
CriteriaBuilder builder = em.getCriteriaBuilder();

//The generic type argument declares the type of result
//this CriteriaQuery will return upon execution
CriteriaQuery<Person> query = builder.createQuery(Person.class);
```

the return type

```
//Query expressions are the core units or nodes that are
//assembled in a tree to specify a CriteriaQuery
Root<Person> e = query.from(Person.class);
Predicate condition = builder.gt(e.get(Person_.age), 20);
query.where(condition);
```

a path in the query tree

```
TypedQuery<Person> q = em.createQuery(query);
List<Person> result = q.getResultList();
```

The equivalent JPQL query is
`SELECT e FROM Person e WHERE e.age > 20`

Using the Metamodel API

- A **metamodel class** describes the meta information of a persistent class in a **static** manner, generated at development time.
- The metamodel classes are usually generated automatically by the IDE.

```
import javax.persistence.metamodel.SingularAttribute;
@javax.persistence.metamodel.StaticMetamodel(Person.class)
public class Person_ {
  public static volatile SingularAttribute<Person,String> name;
  public static volatile SingularAttribute<Person,Integer> age;
}
```

This `Person` metamodel class is an alternative means of referring to meta information of `Person`. This alternative is similar to Java Reflection API, but with a major conceptual difference.

Typed / Tuple Criteria Queries

- Selecting an entity

```
CriteriaQuery<Person> criteria = builder.createQuery(Person.class);
Root<Person> personRoot = criteria.from(Person.class);
criteria.select(personRoot);
```

- Selecting a property or an expression

```
CriteriaQuery<Integer> criteria = builder.createQuery(Integer.class);
Root<Person> personRoot = criteria.from(Person.class);
criteria.select(personRoot.get(Person_.age )); //or
criteria.select(builder.max(personRoot.get(Person_.age )));
```

- Selecting multiple values or tuples

```
CriteriaQuery<Tuple> criteria = builder.createQuery(Tuple.class);
Path<Long> idPath = personRoot.get(Person_.id);
Path<Integer> agePath = personRoot.get(Person_.age);
criteria.multiselect(builder.array(idPath, agePath));
```

From: Root, Join, Fetch

Criteria queries are essentially an object graph, where each part of the graph represents an increasing (as we navigate down this graph) more atomic part of query.

```
CriteriaQuery<Person> personCriteria =
   builder.createQuery( Person.class );
Root<Person> personRoot = person.from( Person.class );

// Person.address is an embedded attribute
Join<Person,Address> personAddress =
   personRoot.join(Person_.address);

// Address.country is a ManyToOne
Fetch<Address,Country> addressCountry =
   personAddress.fetch( Address_.country );
...
```

Path Navigation. Where.

- The **Path.get** method is used to navigate to attributes of the entities of a query.
- The **where** method evaluates instances of the *Expression* interface to restrict the results
- *isNull, isNotNull, in*

```
cq.where(pet.get(Pet_.color).in("brown", "black"));
```

- *equal, notEqual, gt, ge, lt, le, between, like*

```
cq.where(builder.like(pet.get(Pet_.name), "*do"));
```

- *and, or, not*

```
cq.where(builder.equal(pet.get(Pet_.name), "Fido")
    .and(builder.equal(pet.get(Pet_.color), "brown")));
```

Dynamic Queries

- Queries that dependend on runtime values
- Example:

```
Predicate age18 = builder.gt(pers.get(Person_.age), 18);
Predicate hasNoJob =
 builder.isEmpty(pers.get(Person_.jobList));

Predicate filter = builder.conjunction();
if (isAgeFilterEnabled)
  filter = builder.and(filter, age18);
if (isJobFilterEnabled)
  filter = builder.and(filter, hasNoJob);
…
query.where(filter);
```

...or create an array of predicates

```
query.where(predicates);
```

JPQL or Criteria API?

- JPQL

  + easy to write

  - not type-safe (may poduce errors at runtime)

  - ugly for dynamic queries

- Criteria API

  + type-safe, object-oriented

  + nice for dynamic queries

  - verbose, not so easy to write

- *Alternatives (QueryDsl, ...)*

JPA Performance Tunning

- **Lazy Loading**
- **Caching**
- Pagination
- Join Fetching and Batch Fetching
- Read-Only Objects
- Sequence Pre-allocation
- Cascade Persist
- Batch Writing, etc.

Lightweight Objects “Only What You Need”

- The Context
  - `Company company = em.find(Company.class, companyId);`
  - `A Company has many Employees, Departaments, etc.`
- Objects should only initialize <u>what they need when needed</u> → wait until the feature of the object is needed and then initialize it.
- Substantially reduce the amount of time necessary to create an instance of the object.
- How is this implemented?
  Using an **interceptor** and a **sentinel value** for each property.

JPA Lazy Loading

- **Indirection** (also known as lazy reading, lazy loading, and just-in-time reading)

  → place holder for the referenced object

- All @Basic, @OneToMany, @ManyToOne, @OneToOne, and @ManyToMany annotations have an optional parameter called **fetch**.
  - **FetchType.LAZY**: the loading of that field may be delayed until it is accessed for the first time
  - **FetchType.EAGER**
- The default is to load property values eagerly and to load collections lazily.

Lazy Loading Implementation

https://www.eclipse.org/eclipselink/documentation/3.0/concepts/mappingintro002.htm

- **Build-time bytecode instrumentation**
  - The mechanism for instrumentation is modification of the byte-codes of methods.
  - The entity classes are instrumented after they have been compiled and before they are deployed.
- **Run-time bytecode instrumentation**
  - This requires installing a Java agent
- **Run-time proxies**
  - In this case the classes are not instrumented
  - The objects returned by the JPA provider are proxies to the actual entities (see Dynamic Proxy API, CGLIB, etc.)

The n+1 Query Problem

- Using too many SQL queries to retrieve the required entities from the database.
- Each entity has collections that relate to other entities – in some cases we need the data right away, in some not.

```
// In Order class
@OneToMany( mappedBy = "order", fetch = FetchType.LAZY)
public Set<OrderItem> items = new HashSet<OrderItem>();

//Somewhere else (How many SQL queries?)
List<Order> order = em.createQuery(
   "SELECT e FROM Order e").getResultList();

for (Order order : orders) {
   System.out.println("Order: " + order.getDate() + ": " +
   order.getItems().stream().map(item → item.getProduct())

     .collect(Collectors.joining(", ")));

}
```

JPA Entity Graphs

- The main goal of the JPA Entity Graph is to improve the runtime performance when loading the entity's related associations and basic fields.

```
@Entity
@Table(name = "orders")
@NamedEntityGraph(name = "graph.Order.items",
      attributeNodes = @NamedAttributeNode("items"))
public class Order { ... }
```

- Executing a query with a “hint”

```
EntityGraph graph = em.getEntityGraph("graph.Order.items");
em.createQuery("SELECT e FROM Order e")
   .setHint("javax.persistence.loadgraph", graph)
   .getResultList();
```

JPA Caching

Cache: storage for duplicating original data (usually in memory)

- **Object Caching**
  - cache the objects themselves including all of their structure and relationships
- **Data Caching**
  - cache only the database row data of objects
- **Query Caching**
  - cache the query results instead of objects, keyed on the query name and its parameters
- **Statement Caching**

Object Caching

- Caching the object's state for the duration of a transaction is normally called: *the first-level cache*
- A **second-level cache** is a local store of entity data managed (typically transparent to the application) by the persistence provider to improve application performance.

```
<persistence-unit name="examplePU" transaction-type="JTA">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
  <jta-data-source>jdbc/sample</jta-data-source>
  <shared-cache-mode>ALL</shared-cache-mode>
</persistence-unit>
```

- ALL, NONE, ENABLE_SELECTIVE, DISABLE_SELECTIVE, UNSPECIFIED
- To specify that an entity may be cached:

```
@Cacheable(true)
@Entity
public class Person{ ... }
```

Controlling the Second-Level Cache

```
EntityManager em = ...;
Cache cache = em.getEntityManagerFactory().getCache();

//Checking Whether an Entitys Data Is Cached
String personPK = ...;
if (cache.contains(Person.class, personPK)) {
  // the data is cached
} else {
  // the data is NOT cached
}

//Removing an Entity from the Cache
cache.evict(Person.class, personPK); //one instance
cache.evict(Person.class); //all insances

//Removing All Data from the Cache
cache.evictAll();
```

Stale Data

- **Stale data** appears when the cache version is getting out of sync with the original data.
- If there are other applications accessing the same database, the stale data can become a big issue!
- The **EntityManager.refresh()** operation is used to refresh an object's state from the database

Criteria Update/Delete

- Updating entities one by one is another common reason for performance issues in JPA
- Criteria API bulk update operations map directly to database update operations, bypassing any optimistic locking checks. The persistence context is not synchronized with the result of the bulk update.
- Example

```
CriteriaBuilder cb = this.em.getCriteriaBuilder();
CriteriaUpdate update = cb.createCriteriaUpdate(Author.class);

Root a = update.from(Author.class);
update.set(Author_.firstName,
           cb.concat(a.get(Author_.firstName), " - updated"));
update.where(cb.greaterThanOrEqualTo(a.get(Author_.id), 3));

// perform update
Query q = this.em.createQuery(update);
q.executeUpdate();
```