



Java Persistence API

- Define Java persistence API
- Define the Primary Features of JPA
- Define and demonstrate O/R Mapping
- Define and build application using EntityManager & Persistence context
- Demonstrate entity annotations
- Define and Demonstrate Relation Mapping
- Define & Demonstrate JPQL (Java Persistence Query Language)

Previously we learnt about

JDBC

Data Access Objects (DAO) and Data Transfer Objects (DTO)

In JDBC, we "hard coded" SQL into our application

Then used Data Source/Connection Pooling

Then used DAO/DTO

But this just "hides" implementation from our business logic, you still implement DAO with JDBC

However,

We still have to understand a lot of implementation details (eg: connections, statements, resultsets etc)

What about relationships? Joins? Inheritance?

Object \leftrightarrow database impedance mismatch

J2EE tried to solve this with

"Entity Enterprise JavaBeans (EJB)"

Simpler alternatives included

Object Relational Mapping (ORM) tools:

e.g. Java Data Objects (JDO), Hibernate, iBatis, TopLink

Object Relational Mismatch

SQL Types and Java Types are different

- Databases also support SQL types differently
- Tend to define their own internal data types e.g. Oracle's NUMBER type
- Types must be mapped between Java and SQL/Database
- JDBC (Generic SQL) Types are defined in `java.sql.Types`
- java types are very rich; SQL types are more restrictive

How to map class to table? 1:1? 1:n?

How to map columns to class properties?

BLOB support? Streaming?

How to do Object Oriented design here?

What about inheritance? Abstraction? Re-use?

- **“Domain Model” pattern**
 - Focus on **business concepts**, not relational DB structure
 - Interconnected objects
 - Each object is a meaningful individual/concept
 - OO concepts: inheritance, object identity, etc.
 - Navigate data by walking the object graph, not the explicit relational model
- **Increased development speed & reduced code**
 - No “by-hand” mapping of JDBC ResultSets to POJOs
 - Less work synchronizing code with relational DB changes
 - Less JDBC boilerplate (repetitious CRUD)
 - Focus on business logic

- **Portability**

- **Mostly** DB independent (exception: some types of features, identifier generation, etc.)
- Query abstractions (OO APIs, OO structured languages, etc.)
- Vendor-specific SQL is auto generated

- **Performance**

- Granular control of “when”, “how”, “how much” data/relationships to load, based on the business logic
- Object and query caching
 - Concurrency & multiple-tenancy
 - Transactional
 - Scalable
 - Extendable (**many** types of extension points)

Java SE 5 added new constructs to Java language

- Generics
- Annotations
- Enumerations

Java EE 5 used these features to provide

- Ease of development
- "Dependency injection"
- Meaningful defaults, "code by exception"
- Simplified EJB
- New Java Persistence API (JPA) replaced Entity EJB

Java EE 5 still keeps JDBC

EJB 3 spec (JSR 220) split into 2:

- Session Beans, Message Beans

- Java Persistence API (JPA)

JPA jointly developed by TopLink, Hibernate, JDO, EJB vendors and individuals

JPA can also be used in Java SE 5 without a container!!!!

Java Persistence consists of three areas:

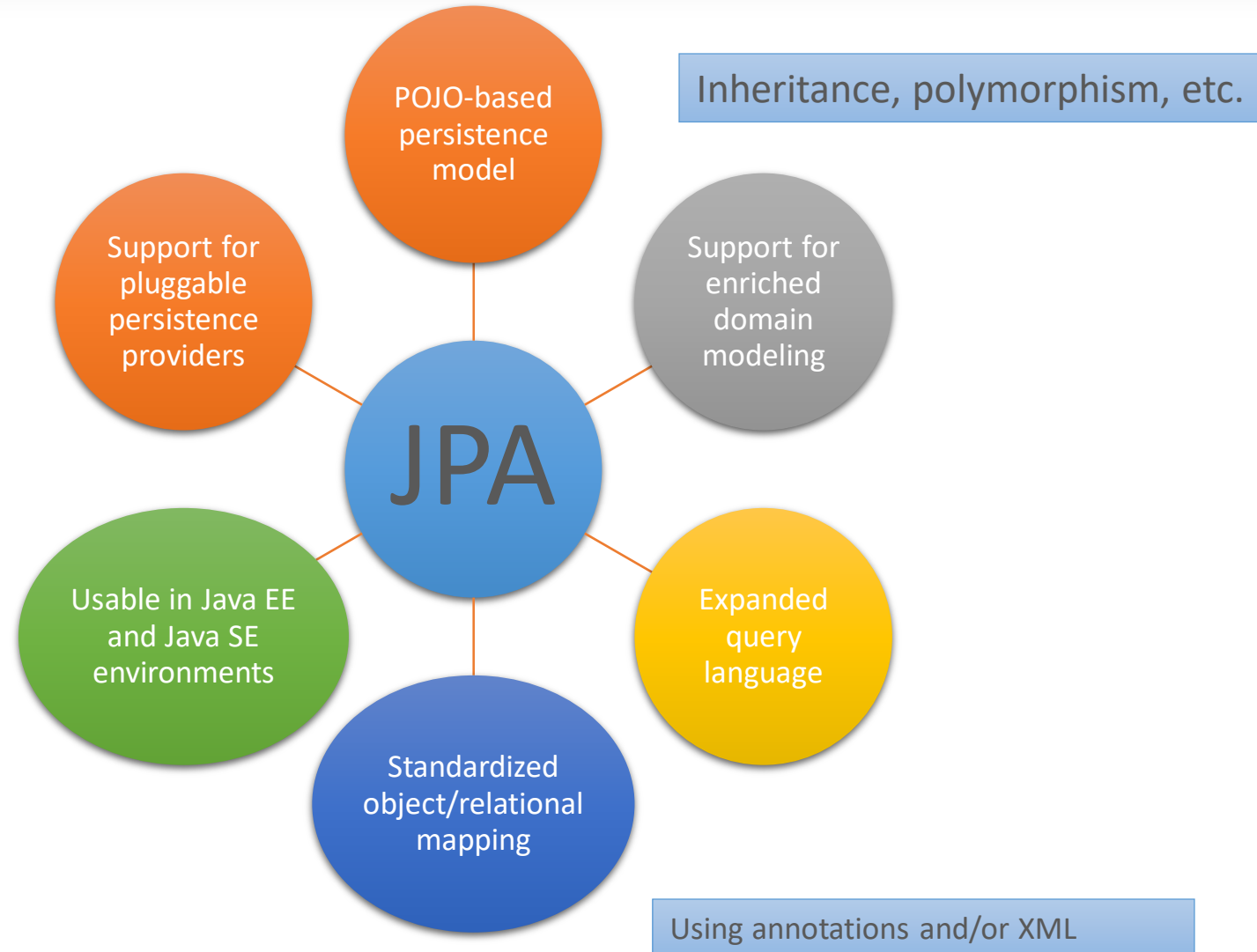
- The Java Persistence API
- The query language
- Object/relational mapping metadata

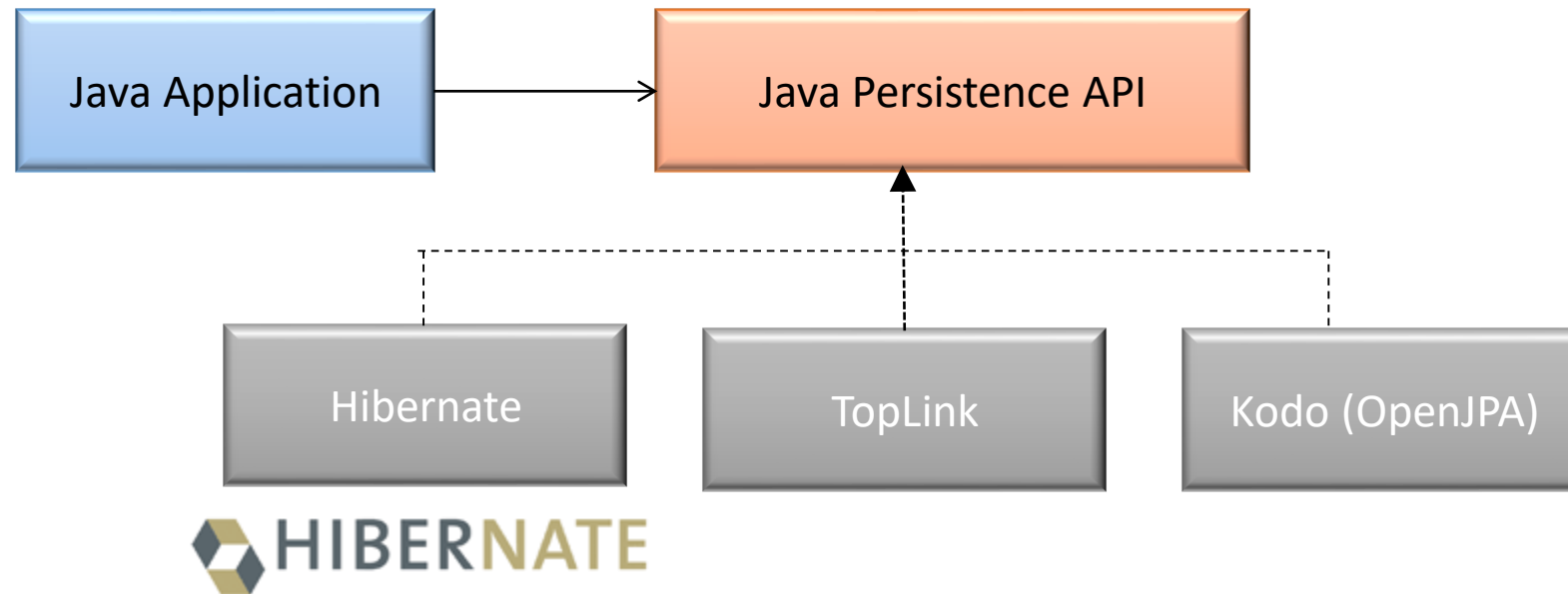
JPA implementation

Reference implementation: TopLink (GlassFish project)

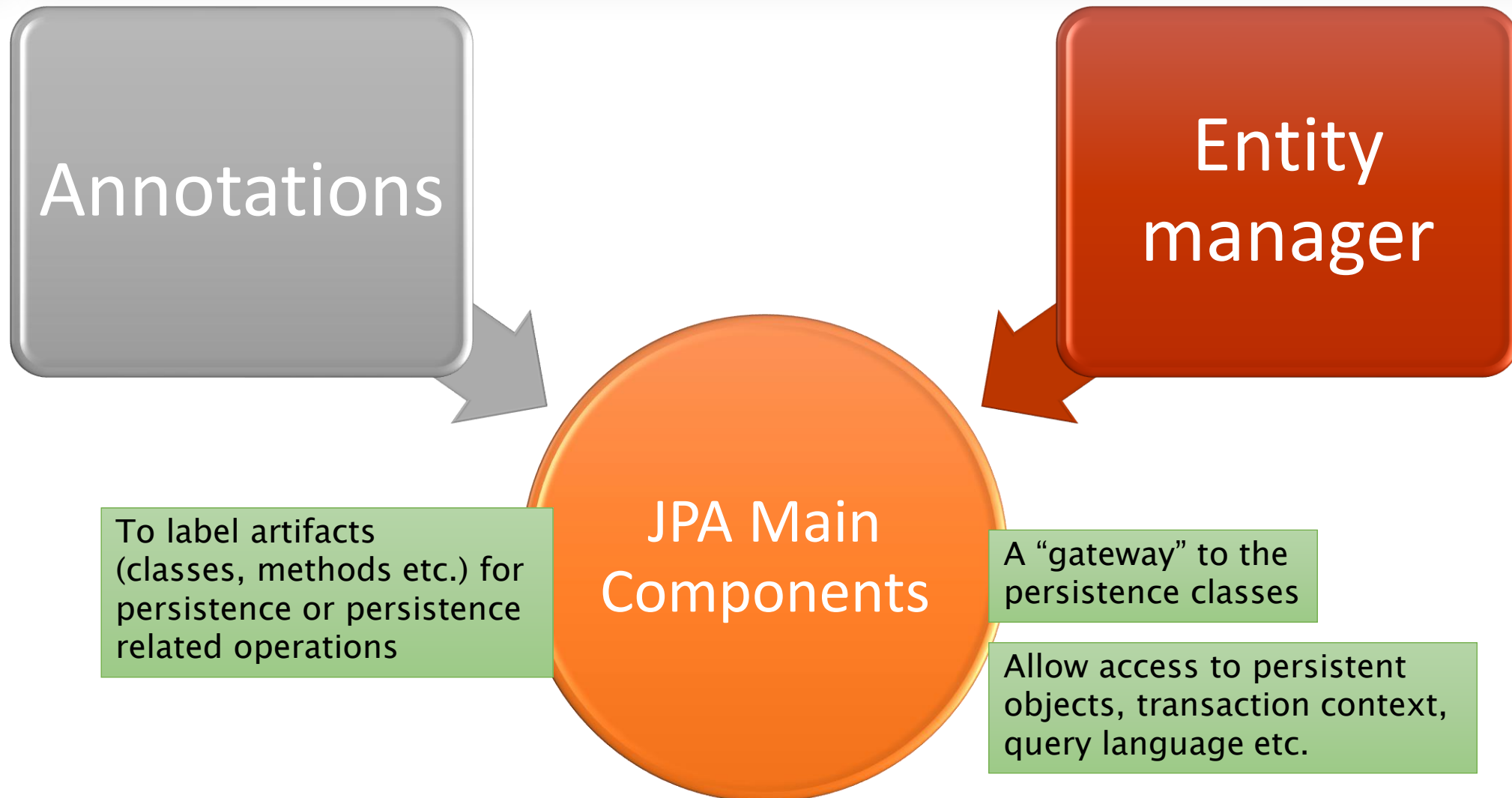
Most ORM vendors now have JPA interface

eg: Hibernate-JPA, EclipseLink (based on TopLink), OpenJPA (based on BEA Kodo)





Everyone can use their own favorite persistence technology



javax.persistence.*

- EntityManager
- EntityManagerFactory
- EntityTransaction
- Query

"Entity" – we use Plain Old Java Objects (POJO) instead.

- An entity is a plain old java object (POJO)
- The Class represents a table in a relational database.
- Instances correspond to rows

Requirements:

- Annotated with **@Entity**
- Contains a persistent **@Id** field
- No argument constructor (public or protected)
- Not marked final
- Class, method, or persistent field level
- Top level class
- Can't be inner class, interface, or enum
- Must implement Serializable to be remotely passed
- by value as a detached instance

The persistent state of an entity can be accessed:
through the entity's instance variables
through JavaBeans-style properties (getters/setters)

Supported types:

- primitive types, String, other serializable types, enumerated types
- other entities and/or collections of entities
- embeddable classes

All fields not annotated with `@Transient` or not marked as Java transient will be persisted to the data store!

- JPA annotations are defined in the javax.persistence

package: <http://java.sun.com/javaee/5/docs/api/javax/persistence/packagesummary.html>

- Annotations can be placed on fields or properties
- Field level access is preferred to prevent executing logic
- Property-level annotations are applied to "getter" method
- Can't mix style in inheritance hierarchy
- Must decide on field OR property

Entities must define an id field/fields corresponding

- the database primary key
- The id can either be simple or composite value

Strategies:

- @Id: single valued type - most common
- @IdClass: map multiple fields to table PK
- @EmbeddedId map PK class to table PK

Composite PK classes must:

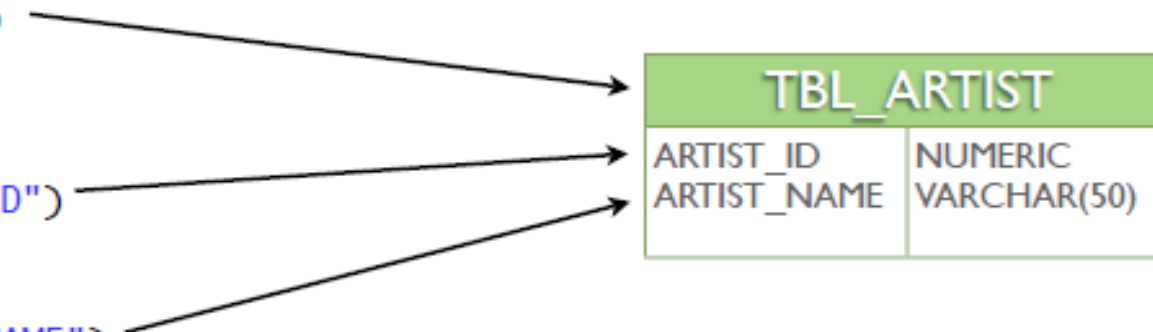
- implement Serializable
- override equals() and hashCode()

- ▶ Used to define *name* mappings between Java object and database table/columns
- ▶ @Table applied at the persistent class level
- ▶ @Column applied at the persistent field/property level

```
@Entity
@Table(name = "TBL_ARTIST")
public class Artist {

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

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



TBL_ARTIST	
ARTIST_ID	NUMERIC
ARTIST_NAME	VARCHAR(50)

In most of the cases, the defaults are sufficient

By default the table name corresponds to the unqualified name of the class

Customization:

```
@Entity  
@Table(name = "FULLTIME_EMPLOYEE")  
public class Employee{ ..... }
```

The defaults of columns can be customized using the **@Column** annotation

```
@Id @Column(name = "EMPLOYEE_ID", nullable = false)  
private String id;
```

```
@Column(name = "FULL_NAME" nullable = true, length = 100)  
private String name;
```

- ▶ 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

- ▶ Used to determine strategy for persisting Java enum values to database
- ▶ Values defined by EnumType:
 - ▶ EnumType.ORDINAL (default)
 - ▶ EnumType.STRING

```
@Entity
public class Album {
    ...
    @Enumerated(EnumType.STRING)
    private Rating rating;
    ...
}
```



ALBUM	
ALBUM_ID	NUMERIC
RATING	VARCHAR(10)

- ▶ Used to persist values to BLOB/CLOB fields
- ▶ Often used with @Basic to lazy load value

```
@Entity
public class Album {
    ...
    @Lob
    @Basic (fetch = FetchType.LAZY)
    @Column(name = "ALBUM_ART")
    private byte[] artwork;
    ...
}
```



ALBUM	
ALBUM_ID	NUMERIC
ALBUM_ART	BLOB

- ▶ 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
```

```
}
```


Identifiers can be generated in the database by specifying **@GeneratedValue** on the identifier

- Four pre-defined generation strategies:
- AUTO, IDENTITY, SEQUENCE, TABLE
- Generators may pre-exist or be generated
- Specifying strategy of AUTO indicates that the provider will choose a strategy

```
@Id @GeneratedValue(strategy=AUTO)  
private int id;
```

- ▶ Supports auto-generated primary key values
- ▶ Strategies defined by GenerationType enum:
 - ▶ GenerationType.AUTO (preferred)
 - ▶ GenerationType.IDENTITY
 - ▶ GenerationType.SEQUENCE
 - ▶ GenerationType.TABLE

```
@Id
```

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

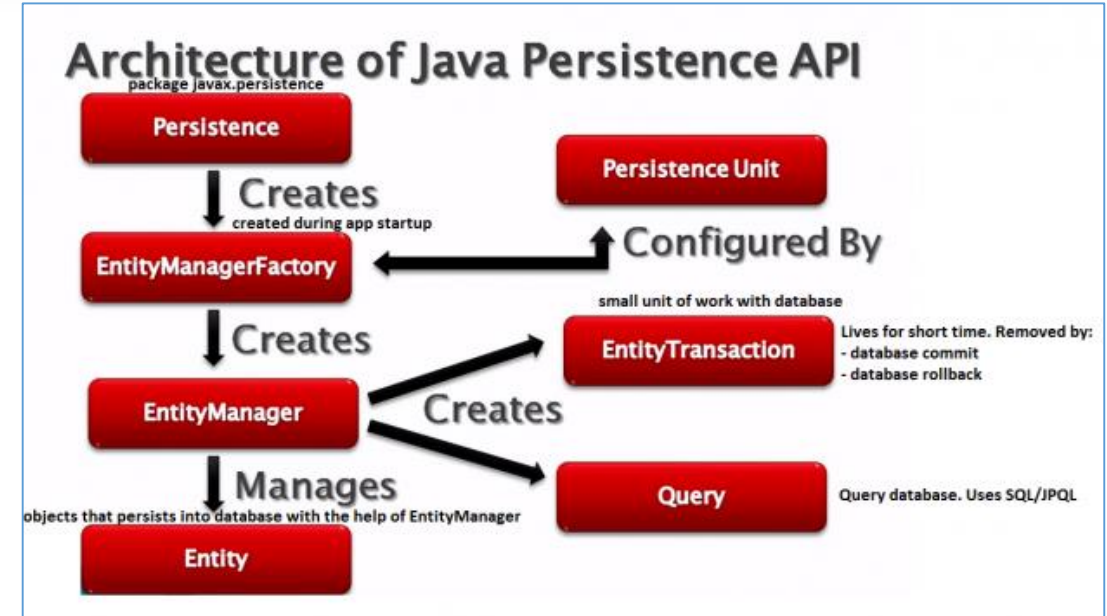
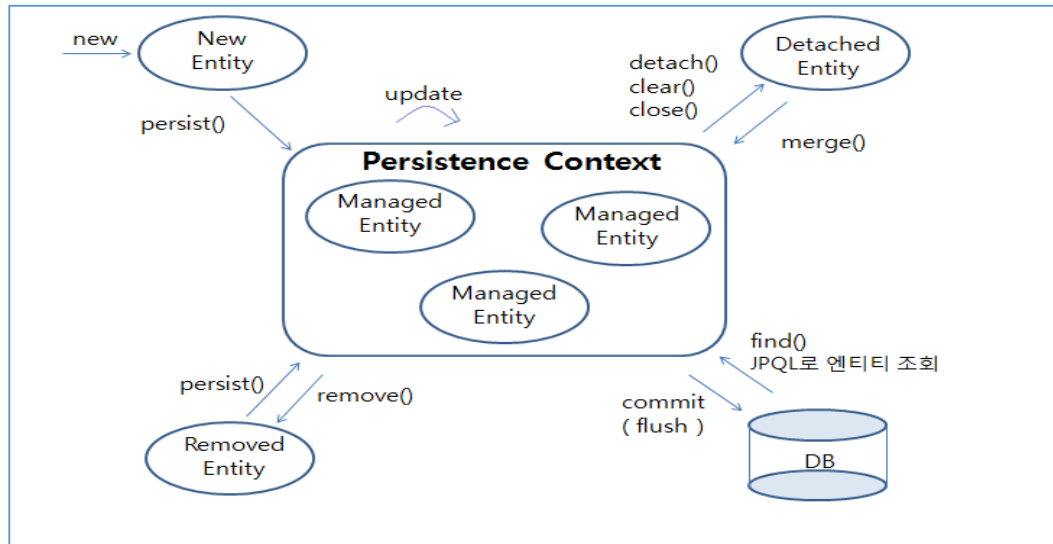
```
private Long id;
```

A **persistence unit** defines a set of all entity classes that are managed by **EntityManager** instances in an application

Each persistence unit can have different providers and database drivers

Persistence units are defined by the **persistence.xml** configuration file

```
<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="EmployeeService" transaction-type="RESOURCE_LOCAL">
    <class>com.soft.infg.model.Employee</class>
    <validation-mode>NONE</validation-mode>
  <properties>
    <property name="javax.persistence.jdbc.driver" value="oracle.jdbc.driver.OracleDriver"/>
    <property name="javax.persistence.jdbc.url" value="jdbc:oracle:thin:@127.0.0.1:1521:XE"/>
    <property name="javax.persistence.jdbc.user" value="JPA"/>
    <property name="javax.persistence.jdbc.password" value="jpa"/>
    <property name="hibernate.hbm2ddl.auto" value="create-drop" />
  </properties>
</persistence-unit>
</persistence>
```



Persistence Context

- Abstraction representing a set of **“managed”** entity instances
 - Entities keyed by their persistent identity
 - Only one entity with a given persistent identity may exist in the PC
 - Entities are added to the PC, but are not individually removable (**“detached”**)
- Controlled and managed by EntityManager
 - Contents of PC change as a result of operations on EntityManager API

An **EntityManager** instance is used to manage the state and life cycle of entities within a persistence context

Entities can be in one of the following states:

1. New
2. Managed
3. Detached
4. Removed

- EntityManager API
 - **persist()** - Insert the state of an entity into the db
 - **remove()** - Delete the entity state from the db
 - **refresh()** - Reload the entity state from the db
 - **merge()** - Synchronize the state of detached entity with the pc
 - **find()** - Execute a simple PK query
 - **createQuery()** - Create query instance using dynamic JP QL
 - **createNamedQuery()** - Create instance for a predefined query
 - **createNativeQuery()** - Create instance for an SQL query
 - **contains()** - Determine if entity is managed by pc
 - **flush()** - Force synchronization of pc to database

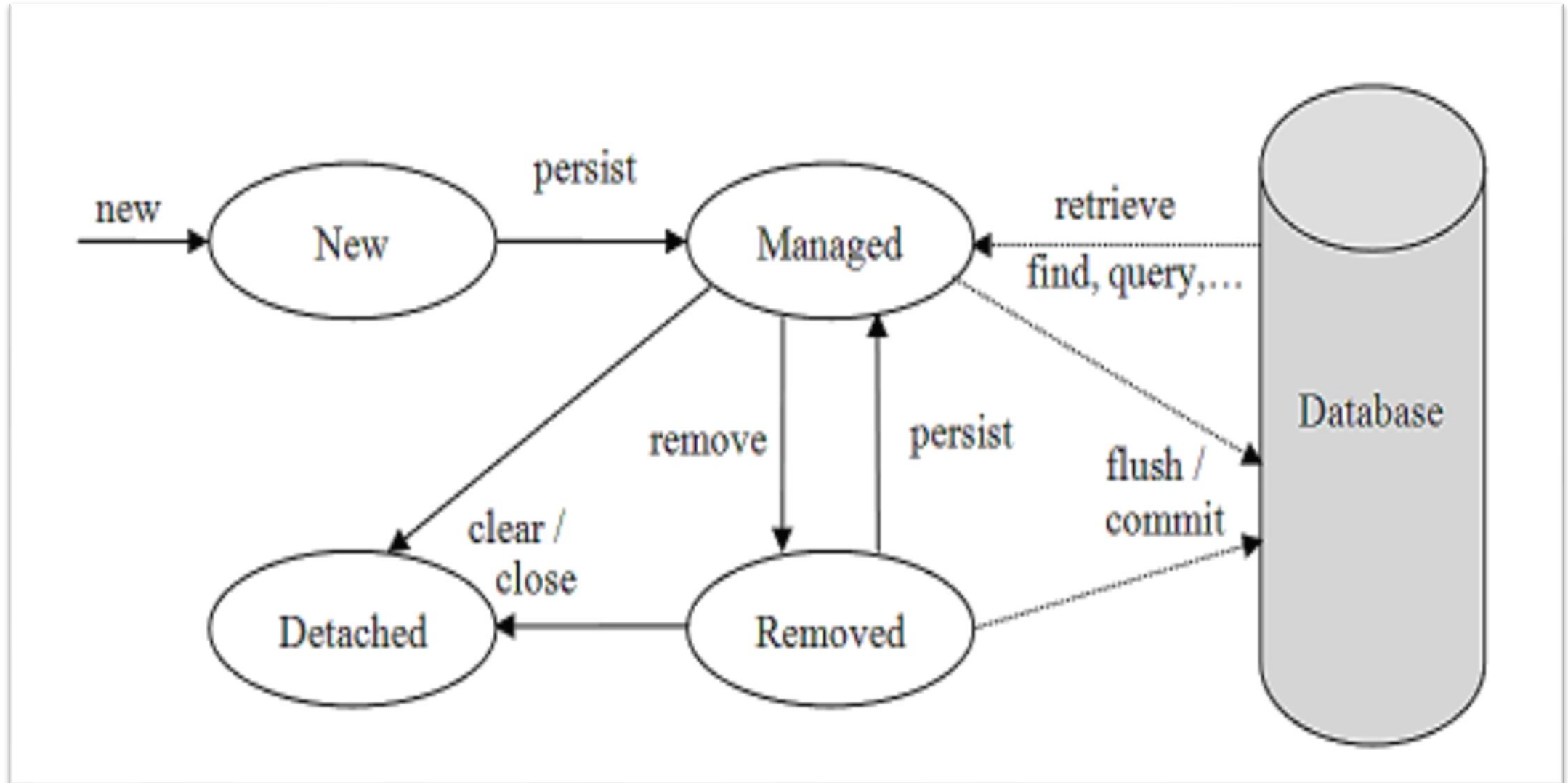
New – entity is instantiated but not associated with persistence context. Not linked to database.

Managed – associated with persistence context. Changes get synchronised with **database**

Detached – has an id, but not connected to database

Removed – associated with persistence context, but underlying row will be deleted.

The **state** of persistent entities is synchronized to the database when the transaction **commits**



The EntityManager API:

- **creates** and **removes** persistent entity instances
- **finds** entities by the entity's primary key
allows **queries** to be run on entities

There are two types of EntityManagers:

- **Application-Managed** EntityManagers
ie: run via Java SE
- **Container-Managed** EntityManagers
ie: run via Java EE Container eg: Tomcat

Java SE applications create EntityManager instances by using **directly Persistence** and **EntityManagerFactory**:

- **javax.persistence.Persistence**

- Root class for obtaining an EntityManager

- Locates provider service for a named persistence unit

- Invokes on the provider to obtain an EntityManagerFactory

- **javax.persistence.EntityManagerFactory**

- Creates EntityManagers for a named persistence unit or configuration

Applications must manage own transactions too..

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("EmployeeService");
EntityManager em = emf.createEntityManager();
EmployeeService service = new EmployeeService(em);
// create and persist an employee
em.getTransaction().begin();

// do operations here
em.close();
emf.close();
```

JPA transactions can be managed by:

- the users application

- a framework (such as Spring)

- a Java EE container

Transactions can be controller in two ways:

- **Java Transaction API (JTA)**

 - container-managed entity manager

- **EntityTransaction** API (`tx.begin()`, `tx.commit()`, etc)

 - application-managed entity manager



Thank You



www.infogain.com

Infogain Corporation, HQ

485 Alberto Way Los Gatos,
CA 95032 USA
Phone: 408-355-6000
Fax: 408-355-7000

Pune

7th Floor, Bhalerao Towers, CTS No.1669 -
1670, Behind Hotel Pride,
Shivaji Nagar, Pune - 411005
Phone : +91-20-66236700

Infogain Irvine

41 Corporate Park,
Suite 390 Irvine, CA 2606 USA
Phone: 949-223-5100
Fax: 949-223-5110

Infogain Austin

Stratum Executive Center Building D
11044 Research Boulevard Suite 200
Austin, Texas 78759

Noida

A-16, Sector 60, Noida Gautam Budh agar,
201301 (U.P.) India
Phone: +91-120-2445144
Fax: +91-120-2580406

Dubai

P O Box 500588 Office No.105,
Building No. 4, Dubai Outsource Zone,
Dubai, United Arab Emirates
Tel: +971-4-458-7336

