Procesos de la Ingeniería Software
Tema 4

Soporte Java para construcción de aplicaciones empresariales

3. Capa de persistencia en Jakarta EE: Jakarta Persistence (antigua JPA)

Bibliografía

Básica

- Oracle and affiliates (2021): The Jakarta EE Tutorial (Release 9.1)
  - Part VIII: Persistence (Capítulos 40 – 42)

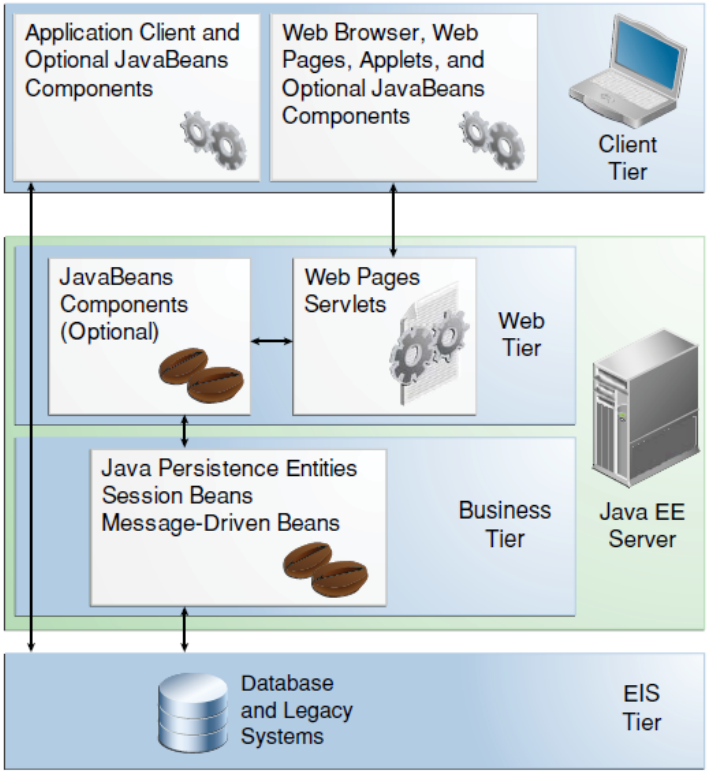
Complementaria

- Antonio Goncalvez (2013): Beginning Java EE 7, Apress
  - Capítulo 4 - 6

- Eclipse Foundation: Jakarta Persistence
  - https://jakarta.ee/specifications/persistence/3.0/jakarta-persistence-spec-3.0.html

Capa de persistencia en aplicaciones empresariales

- La capa de negocio de una aplicación empresarial gestiona y usa los datos almacenados en la BBDD
- La capa de persistencia constituye el enlace entre la capa de negocio y la BBDD
- En sistemas OO, la capa de persistencia se encarga del **Mapeado Objeto-Relacional** (ORM)
  - Mapeo de objetos software a relaciones/tuplas del modelo relacional
- Formas de implementar la capa de persistencia:
  - Con interfaces de bajo nivel: JDBC
  - A través de frameworks ORM: Ibatis, Hibernate, EclipseLink, etc.



Capa de persistencia en Jakarta EE - Jakarta Persistence

- **Jakarta Persistence** es la especificación Jakarta EE para la implementación de capas de persistencia
  - Es independiente del framework ORM y del gestor de BBDD
    - Simplifica el ORM, pero quien realmente realiza el mapeo es el proveedor de persistencia subyacente
  - No es exclusiva de Jakarta EE, se puede usar también en aplicaciones Java SE
  - Versión actual: 3.0
  - Paquete `jakarta.persistence`
  - Evolución de **Java Persistence API** (**JPA**) de Java EE

- Los elementos principales de Jakarta Persistence son:
  - **Entidades** (Entity) con sus correspondientes metadatos (anotaciones o descriptores XML)
  - Gestor de entidades (**EntityManager**)
  - Lenguaje de consulta **JPQL** (Jakarta Persistence Query Language)

- Implementaciones:
  - Implementación de referencia: EclipseLink 3.0.0
  - Otras implementaciones: Hibernate 5.5.0 y 6.0.0

Elementos de JPA - Entidad (Entity)

- Una **entidad** es un objeto de una aplicación cuyo estado se hace persistente
  - El estado lo forman los atributos o las propiedades (pareja get/set) del objeto

- Una **clase entidad** representa una tabla en una BBDD relacional
  - Cada entidad (instancia de la clase) representa una fila de dicha tabla
  - Jakarta Persistence sincroniza entidades con sus correspondientes tablas en la BBDD
    - Gracias a metadatos definidos en forma de anotaciones o descriptores xml
  - Cuando la clase entidad y la tabla están sincronizadas, el contenedor se encarga de mapear automáticamente las invocaciones en cada objeto entidad (creación, modificación de atributos, etc.) a las operaciones en la BBDD

Modelo Relacional (BBDD)	JPA
Relación (Tabla)	Clase entidad
Atributo (Columna)	Atributo/Propiedad de la clase
Tupla (Fila)	Instancia de la clase entidad

Entidad - Características principales

- Una clase entidad es un POJO:
  - Anotado como **@Entity**
  - Con, al menos, un constructor público sin parámetros
    - Puede tener más constructores de utilidad
  - Con una clave primaria (primary key)
    - Habitualmente, un atributo/propiedad anotado como **@Id**
    - Existe la posibilidad de definir claves compuestas (no lo vemos)
  - Con patrón get/set para aquellos atributos/propiedades que se quieran hacer persistentes

- Además:
  - Ni la clase ni ninguno de los atributos persistentes pueden ser final ni static
  - Si la entidad se va a usar como parámetro de una interfaz remota, debe implementar Serializable

```
import jakarta.persistence.Entity;

@Entity
public class Usuario implements Serializable {

   @Id
   private String nombre;

   private String email;

   public void Usuario() {
   }

   public void Usuario(String n) {
      nombre = n;
   }

   public String getNombre () {
      return nombre;
   }

   public void setNombre (String nombre) {
      this.nombre = nombre;
   }

   public String getEmail() {
      return email;
   }

   public void setEmail(String email) {
       this.email = email;
   }
}
```

Entidad – Reglas básicas de mapeado

- Reglas de mapeado por defecto:
  - Clase => Tabla de igual nombre
  - Atributo anotado como @Id => Primary Key
  - Atributo/propiedad simple => Columna de la tabla
    - Con mismo nombre, nullable = true y tipo equivalente (e.g. String se mapea a varchar(255))
  - Atributo/propiedad múltiple => Columna de la tabla
    - Reglas de mapeado más complejas (ver T10 – T16)


```java
import jakarta.persistence.Entity;

@Entity
public class Usuario implements Serializable {

   @Id
   private String nombre;

   private String email;

   public void Usuario() {
   }

   public void Usuario(String n) {
      nombre = n;
   }

   public String getNombre () {
      return nombre;
   }

   public void setNombre (String nombre) {
      this.nombre = nombre;
   }

   public String getEmail() {
      return email;
   }

   public void setEmail(String email) {
       this.email = email;
   }
}
```

Entidad - Anotaciones @Id y @Transient

- Toda clase entidad debe tener al menos un atributo que sirva como identificador único de cada instancia de la entidad
  - Anotado como **@Id**
  - PrimaryKey en la BBDD
  - La asignación de la anotación @Id define el criterio usado para establecer en qué consiste el estado persistente de la entidad
    - Asignado a atributo => El estado lo forman los valores de los atributos
    - Asignado a propiedad (get/set) => El estado lo forman los valores de las propiedades
  - Se pueden definir PrimaryKey compuestas
- El estado persistente de la entidad lo forman los valores de todos sus atributos/propiedades excepto aquellos anotados con **@Transient**

```
import jakarta.persistence.Entity;

@Entity
public class Usuario implements Serializable {

   @Id
   private String nombre;

   private String email;

   public void Usuario() {
   }

   public void Usuario(String n) {
      nombre = n;
   }

   public String getNombre () {
      return nombre;
   }

   public void setNombre (String nombre) {
      this.nombre = nombre;
   }

   public String getEmail() {
      return email;
   }

   public void setEmail(String email) {
       this.email = email;
   }
}
```

Entidades – Resumen de anotaciones

Ámbito	Anotación (params)	Uso	Objetivo
Clase	@Entity	Obligatorio	Identifica las clases entidad (las que se van a hacer persistentes)
	@Table (name, …)	Opcional	Configura propiedades de la tabla a la que se mapea la entidad, e. g. el nombre de la tabla. Valor por defecto: Nombre de la clase
Atributo/ Propiedad	@Id	Obligatorio	Identifica el atributo que funciona como primaryKey del objeto
	@GeneratedValue (value)	Opcional	Define el modo en que se genera la primaryKey (sólo para atributos @Id). Valor por defecto: GenerationType.AUTO (el proveedor de persistencia decide la estrategia)
	@Transient	Opcional	Indica que el atributo/propiedad no se hace persistente
	@Column (name, nullable, length, …)	Opcional	Configura cómo se mapea el atributo/propiedad a la columna correspondiente en la BBDD Valores por defecto: Mismo nombre/tipo que el atributo, nullable = true y length = 255 (Strings)
	@Temporal (value)	Opcional	Define el modo en que se mapean atributos de tipo Date/Calendar. Posibles valores: TemporalType.TIME, TemporalType.DATE, TemporalType.TIMESTAMP
	@Enumerated (value)	Opcional	Configura el modo en que se mapea un valor enumerado. Posibles Valores: EnumType.STRING / EnumType.ORDINAL (Mejor usar STRING por mantenibilidad)
	@JoinColumn (name, …)	Opcional	Indica que un atributo actúa como clave foránea de otra entidad name es el nombre de la columna de la clave foránea en la otra entidad
	@OneToOne (fetch, cascade, …)	Opcional	Indica que el atributo está asociado con una tupla de otra Entidad
	@OneToMany (fetch, cascade, ...)	Opcional	Indica que el atributo está asociado con varias tuplas de otra Entidad
	@ManyToOne (fetch, cascade, ...)	Opcional	Indica que el atributo (lado N) está asociado con una tupla de otra Entidad
	@ManyToMany (fetch, cascade, ...)	Opcional	Indica que el atributo tiene una relación N:M con tuplas de otra Entidad

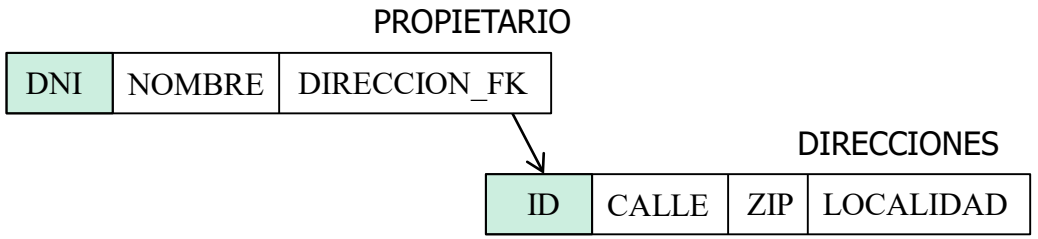
Relaciones entre entidades - OneToOne unidireccional

```java
@Entity
public class Propietario implements Serializable {

    @Id
    private String DNI;

    private String nombre;

    @OneToOne
    @JoinColumn(name="direccion_fk")
    private Direccion direccion;
    // name = "DIRECCION_ID" por defecto
    // name = <Nombre atributo>_<PrimarykeyClaseRef>

    public void Propietario() {  }
    …
}
```

```java
@Entity
@Table(name="Direcciones")
public class Direccion implements Serializable {

    @Id
    @GeneratedValue
    private Long id;

    private String calle;

    @Column(name="zip")
    private String codigoPostal;

    private String localidad;

    public void Direccion() {  }
    …
}
```

@OneToOne, @Column, etc. no son necesarias si no se modifica ningún valor por defecto

@JoinColumn en relaciones @OneToOne y @ManyToOne se mapea a una columna "FK" en la tabla de la clase en la que aparece la anotación

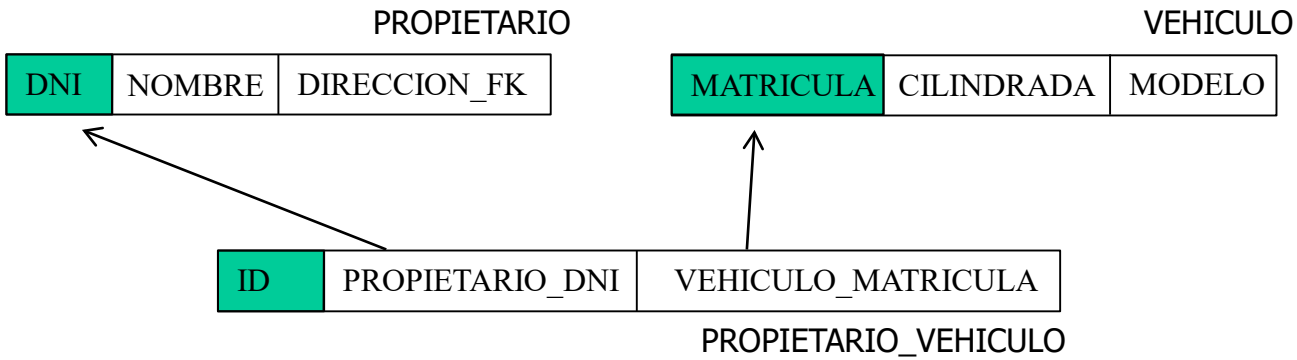


Relaciones entre entidades - OneToMany unidireccional con JoinTable (1)

```java
@Entity
public class Propietario  implements Serializable {

    @Id
    private String DNI;
    private String nombre;
    @OneToOne
    @JoinColumn(name="direccion_fk")
    private Direccion direccion;

    private List<Vehiculo> vehiculos;

    public void Propietario() {    }
    …
}
```

```java
@Entity
public class Vehiculo implements Serializable {

    @Id
    private String matricula;
    private int cilindrada;
    private String modelo;

    public void Vehiculo() {    }
    …
}
```

El mapeado por defecto de una asociación OneToMany es a través de una JoinTable



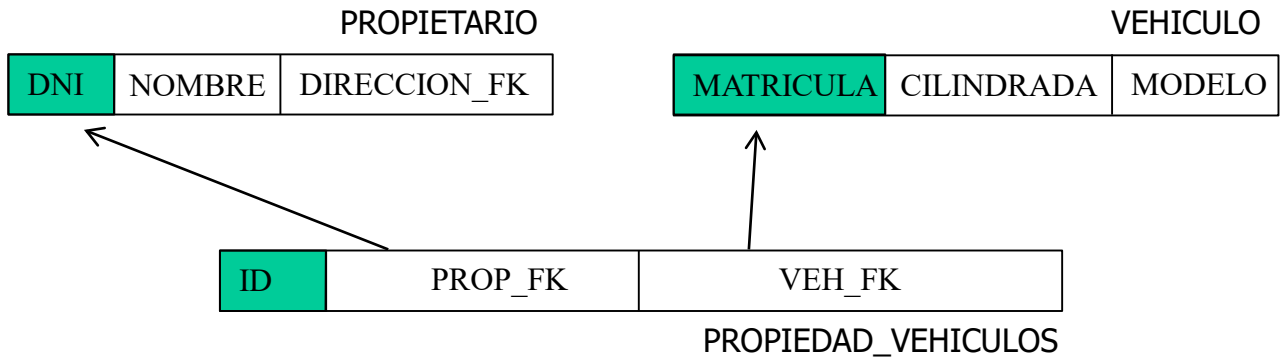
Relaciones entre entidades - OneToMany unidireccional con JoinTable (2)

```java
@Entity
public class Propietario  implements Serializable {

   @Id
   private String DNI;
   private String nombre;
   @OneToOne
   @JoinColumn(name="direccion_fk")
   private Direccion direccion;

   @OneToMany
   @JoinTable(name="Propiedad_Vehiculos",
              joinColumns=@JoinColumn(name="Prop_FK"),
              inverseJoinColumns=@JoinColumn(name="Veh_FK"))
   private List<Vehiculo> vehiculos;

   public void Propietario() {   }
   …
}
```

```java
@Entity
public class Vehiculo implements Serializable {

   @Id
   private String matricula;
   private int cilindrada;
   private String modelo;

   public void Vehiculo() {  }
   …
}
```

Con @JoinTable podemos modificar la JoinTable generada

Se asigna en el lado "owner"

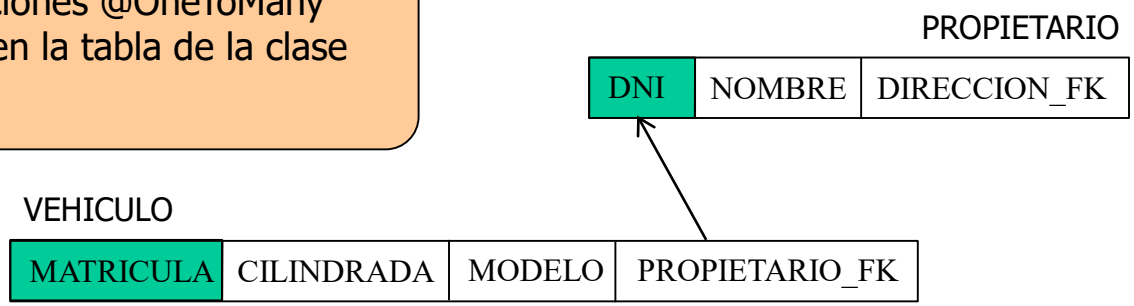


Relaciones entre entidades - OneToMany unidireccional con JoinColumn

```java
@Entity
public class Propietario  implements Serializable {

   @Id
   private String DNI;
   private String nombre;
   @OneToOne
   @JoinColumn(name="direccion_fk")
   private Direccion direccion;

   @OneToMany
   @JoinColumn(name="propietario_fk")
   private List<Vehiculo> vehiculos;

   public void Propietario() {

   }
   …
}
```

```java
@Entity
public class Vehiculo implements Serializable {

   @Id
   private String matricula;
   private int cilindrada;
   private String modelo;

   public void Vehiculo() {

   }
   …
}
```

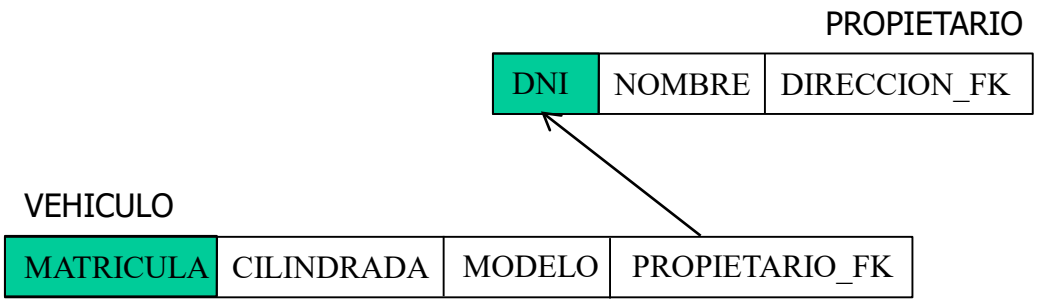
@JoinColumn en relaciones @OneToMany se mapea a columna en la tabla de la clase referenciada



Relaciones entre entidades - Bidireccionales

```java
@Entity
public class Propietario implements Serializable {

   @Id
   private String DNI;
   private String nombre;
   @OneToOne
   @JoinColumn(name="direccion_fk")
   private Direccion direccion;

   // Lado inverso de la relación
   @OneToMany(mappedBy="propietario")
   private List<Vehiculo> vehiculos;

   public void Propietario() {  }
   …
}
```

```java
@Entity
public class Vehiculo implements Serializable {

   @Id
   private String matricula;
   private int cilindrada;
   private String modelo;

   // Lado poseedor (owner) de la relación
   @ManyToOne
   @JoinColumn(name="propietario_fk")
   private Propietario propietario;

   public void Vehiculo() {  }
   . . .

}
```



El poseedor (owner) de la relación es quién define cómo se realiza el mapping (con @JoinColumn o @JoinTable).

El atributo mappedBy se asigna en el otro extremo de la relación (mappedBy indica quién posee la relación)

Quién es el owner queda a decisión del diseñador, excepto en el caso de relaciones OneToMany/ManyToOne, donde el owner debe ser el lado Many

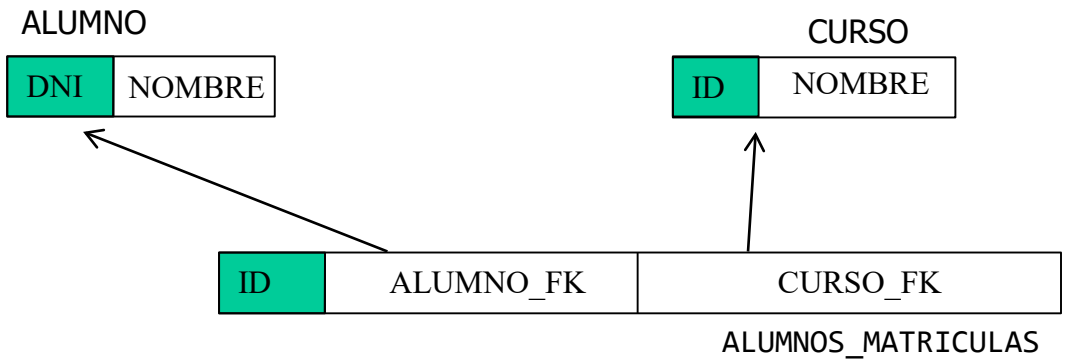
En caso de JoinColumn, la columna con la foreign key se genera en la tabla del owner

Relaciones entre entidades - ManyToMany

```java
@Entity
public class Curso implements Serializable {

    @Id
    private Long ID;
    private String nombre;

    @ManyToMany(mappedBy="cursos")
    private List<Alumno> alumnos;

    public void Curso() {

    }
    …
}
```

```java
@Entity
public class Alumno implements Serializable {

    @Id
    private String dni;
    private String nombre;

    @ManyToMany
    @JoinTable(name="Alumnos_Matriculas",
               joinColumns=@JoinColumn(name="Alumno_FK"),
               inverseJoinColumns=@JoinColumn(name="Curso_FK"))
    private List<Curso> cursos;

    public void Vehiculo() {

    }
    …
}
```

Las relaciones @ManyToMany se mapean siempre a JoinTable



Relaciones entre entidades - Carga de entidades relacionadas

- El atributo **fetch** de las anotaciones @OneToOne, @OneToMany, etc. define el modo en que se cargan en memoria las entidades referenciadas

- Posibles valores:
  - **FETCH.EAGER**: Carga inmediata
    - Puede sobrecargar el sistema
    - Un único acceso a la base de datos
  - **FETCH.LAZY**: Carga retardada
    - No sobrecarga el sistema
    - Muchas peticiones a la base de datos
    - Para que se carguen los valores, hay que invocar el correspondiente get

- Valores por defecto:
  - @OneToOne => EAGER
  - @ManyToOne => EAGER
  - @OneToMany => LAZY
  - @ManyToMany => LAZY

Relaciones entre entidades - Propagación de operaciones

- El atributo **cascade** de las anotaciones @OneToOne, @OneToMany, etc. define el modo en que las operaciones invocadas en la entidad principal se propagan a sus entidades relacionadas
  - El valor del atributo define cuáles de las operaciones se realizan en cascada
  - Posibles valores: ALL, PERSIST, MERGE, REMOVE, REFRESH
  - Valor por defecto: Ninguna operación

```
@Entity
public class Propietario {

   @OneToOne
   private Direccion direccion;

   . . .
}
```

```
Propietario p= new Propietario("72111111", "Pepe");
Direccion d = new Direccion("Avda. de los Castros",
                            "39006", "Santander");
p.setDireccion(d);

em.persist(d);
em.persist(p);
```

```
@Entity
public class Propietario {

   @OneToOne(cascade = Cascade_Type.PERSIST)
   private Direccion direccion;

   . . .
}
```

```
Propietario p= new Propietario("72111111", "Pepe");
Direccion d = new Direccion("Avda. de los Castros",
                            "39006", "Santander");
p.setDireccion(d);

em.persist(p);
```

Relaciones entre entidades - Entidades Embeddable

- Una entidad **Embeddable** es aquella que no se persiste por sí, sino que está contenida en otra entidad
  - Mismas reglas que una Entity, pero anotadas como `@Embeddable`
  - Se mapean a la misma tabla que la entidad contenedora
  - Relación de contención estricta
    - Si desaparece el contenedor, desaparece el contenido

```
@Entity
public class Propietario implements Serializable {

   @Id
   private String DNI;

   private String nombre;

   @Embedded
   private Direccion direccion;

   . . .

}
```

```
@Embeddable
public class Direccion implements Serializable {

   private String calle;

   @Column(name="zip")
   private String codigoPostal;

   private String localidad;

   . . .
}
```

PROPIETARIO

DNI	NOMBRE	CALLE	ZIP	LOCALIDAD

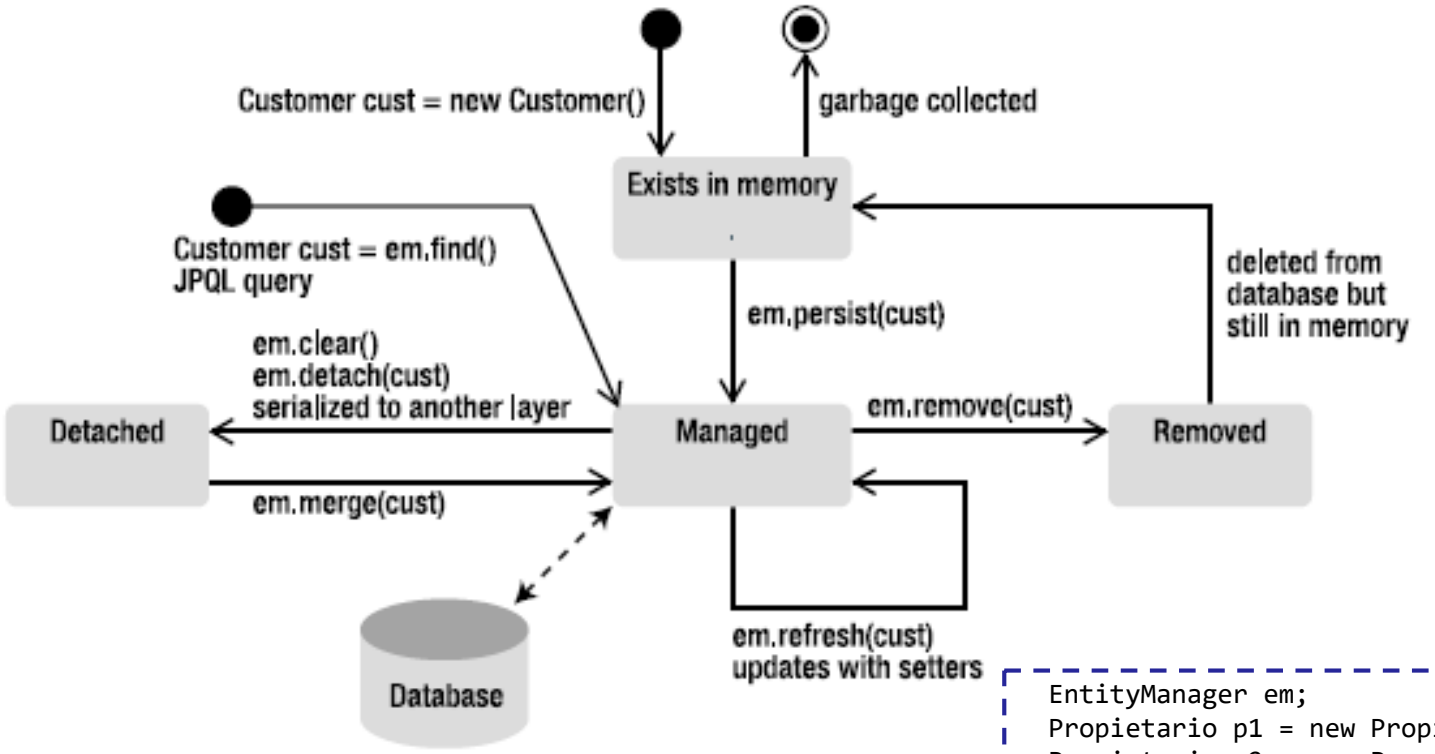
Gestor de entidades (EntityManager)

- **EntityManager:** gestor de entidades o gestor de persistencia en Jakarta Persistence
  - Proporciona la API para gestionar el ORM
    - Operaciones **CRUD** y de **búsqueda** selectiva de entidades (a través del lenguaje JPQL)
    - Las operaciones se definen al nivel de abstracción de OO
      - Haciendo transparente las invocaciones SQL o JDBC
  - Gestiona el ciclo de vida de las entidades
    - Una entidad es un objeto en memoria (no persistente) hasta el momento en que es asociado al EntityManager
      - **Las entidades SÍ se construyen con new** (a diferencia de los Enterprise beans)

A partir de aquí el objeto nuevo es un objeto entidad, se dice que es un Managed Object

```
@Stateless
public class PropietariosDAO {

   private EntityManager em;

   public void anhadePropietario(Propietario nuevo) {

       // Hacemos el propietario persistente
       em.persist(nuevo);
   }
}
```

EntityManager - Ciclo de vida de entidades




```
EntityManager em;
Propietario p1 = new Propietario("11111111X", "Pepe");
Propietario p2 = new Propietario("22222222X", "Juan");
em.persist(p1);
em.persist(p2);
p1.setNombre("Juan");
em.detach(p1);
p1.setNombre("Manolo");
P2.setNombre("Andrés");
System.out.println(p1.getNombre());
System.out.println(p2.getNombre());
System.out.println(em.find(Propietario.class, 1).getNombre);
```

EntityManager - Métodos principales

```
public interface EntityManager {

   /* Hace persistente (crea la tupla en la BBDD) y
    managed la entidad que se pasa como parámetro */
   public void persist(Object entity);

   /* Elimina la entidad del contexto de persistencia
    (deja de estar managed) y de la BBDD */
   public void remove(Object entity);

   /* Actualiza la BBDD con el estado de la entidad
    que se pasa como parámetro. Si no existe la crea
    (como persist). En ambos casos retorna la entidad
    managed (la que se pasa como parámetro no lo está)*/
   public <T> T merge(T entity);

   /* Actualiza el estado de la entidad que se pasa como
    parámetro con los valores que se encuentren en la
    BBDD. Método contrario a merge */
   public void refresh(Object entity);

   /* Desliga la entidad del contexto de persistencia
    (pasa de managed a detached) */
   public void detach(Object entity);

   /* Comprueba si la entidad forma parte del
    contexto de persistencia (es managed) */
   public boolean contains (Object entity);

   /* Sincroniza el contexto de persistencia
    con la BBDD (modifica la BBDD con los
    valores actuales de todas las entities
    managed)*/
   public void flush();

   /* Búsqueda de entidades por clave primaria.
    La entidad que se retorna está managed.
    Si no la encuentra retorna null */
   public <T> T find(Class<T> entityClass,
          Object primaryKey);

   . . .

}
```

EntityManager - Unidad y Contexto de persistencia

- Un EntityManager mapea una **unidad de persistencia** (Persistence Unit) a una BBDD
- Unidad de persistencia: Conjunto de clases entidad mapeadas a una BBDD
  - Tiene un nombre que la identifica unívocamente
  - La unidad de persistencia debe estar desplegada antes de poder hacer uso de ella (instrucciones para el despliegue en la siguiente transparencia)
- En ejecución, el EntityManager gestiona un **contexto de persistencia** (Persistence Context)
  - Gestionar = Mantener sincronización con la BBDD
- Contexto de persistencia: Conjunto de instancias de clases entidad, pertenecientes a una unidad de persistencia dada
  - Formado por objetos managed
  - Si el contexto de persistencia se cierra, las instancias dejan de ser managed, es decir, dejan de estar sincronizadas con la BBDD
  - Al final de cada transacción, todos las instancias entidad son actualizadas a la base de datos (flush implícito)

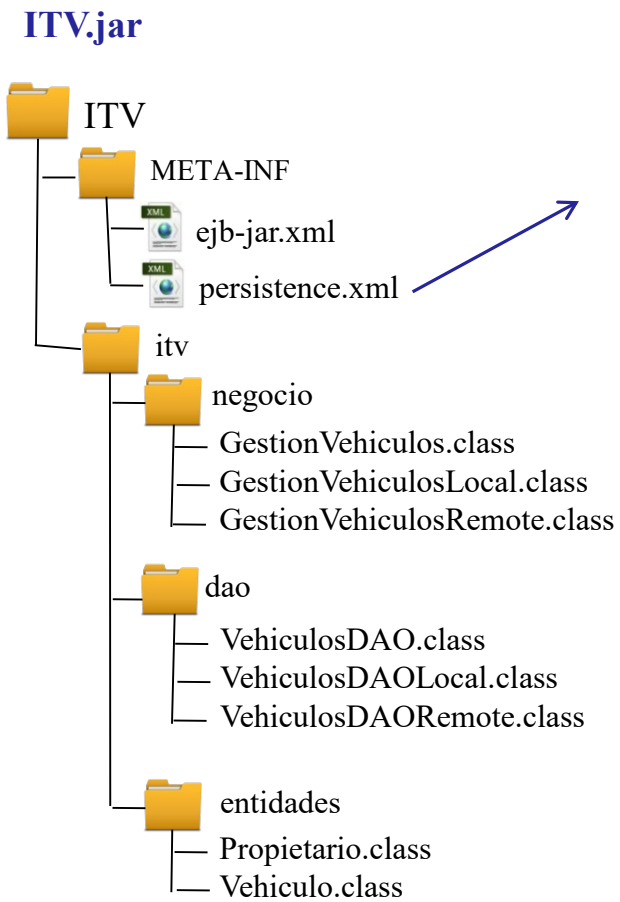
Entity Manager - Despliegue de unidades de persistencia

- Para su despliegue, cada **unidad de persistencia** (o varias a la vez) se empaqueta dentro de un archivo .war o .jar

- Una unidad de persistencia contiene:
  - Las clases entidad que comprende
  - Un descriptor denominado **persistence.xml**
    - Almacenado en el directorio META-INF de la raíz de la unidad de persistencia, que puede ser:
      - Archivo JAR, módulo JAR dentro de un EAR, directorio WEB-INF/classes de un WAR

- El descriptor persistence.xml define:
  - El nombre de la unidad de persistencia
  - El tipo de gestión de transacciones que soporta
    - JTA cuando estamos en Java EE (gestionadas por el contenedor)
    - RESOURCE_LOCAL cuando estamos en Java SE (hay que gestionarlas a mano)
  - Propiedades de conexión con la base de datos
    - Usando JTA, basta con indicar el nombre de la "data-source", que debe estar definida en el correspondiente servidor
  - Clases entidad que forman la unidad. Dos opciones de declaración:
    - Basado en descriptor: Las clases entidad se indican explícitamente en el descriptor
    - Basado en anotaciones: Todas las clases incluidas en el .jar y anotadas con @Entity

Entity Manager – Descriptor persistence.xml




```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
                      https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
             version="3.0"

    <!- Nombre de la unidad de persistencia -->
    <persistence-unit name="vehiculosPU" transaction-type="JTA">

        <!-- JDBC Resource a través del que se obtienen los
          datos de acceso a la BBDD->
        <jta-data-source>jdbc/vehiculos</jta-data-source>

        <!-- Sólo si las clases no están anotadas -->
        <class>itv.entidades.Propietario</class>
        <class>itv.entidades.Vehiculo</class>

        <!- Asignación de propiedades propias del proveedor de persistencia -->
        <properties>
            <!-- Standard and Provider-specific config may go here -->
            <property name="myprovider.property.name" value="someValue"/>
        </properties>
    </persistence-unit>

</persistence>
```

En la "Guía de instalación del software" hay un ejemplo de persistence.xml para Glassfish

EntityManager - Acceso desde componentes Jakarta EE

- En componentes Jakarta, el acceso al EntityManager se consigue a través de inyección de dependencias
  - Indicando el contexto de persistencia que se quiere manejar (identificado por el nombre de la unidad de persistencia)
    - `@PersistenceContext(unitName="XXXXXXX")`
  - Las transacciones y el ciclo de vida del EntityManager son gestionadas por el contenedor

Inyección del EntityManager

```
@Stateless
public class VehiculosDAO implements VehiculosDAOLocal, VehiculosDAORemote {

   @PersistenceContext(unitName="vehiculosPU");
   private EntityManager em;

   public boolean creaVehiculo(Vehiculo v) {
      // Hacemos persistente el vehículo
     try {
         em.persist(v);
         return true;

     } (catch EntityExistsException e) {
         // Ya existe un vehículo en la BBDD con la misma matrícula (PK)
         return false;
     }
   }
}
```

El contenedor maneja la transacción (flush() al acabar el método)

EntityManager - Acceso desde Java SE

- En aplicaciones Java SE, el acceso al EntityManager se consigue a través de una clase Factory
  - Hay que controlar las transacciones y la propia vida del Entity Manager de manera explícita
  - El fichero persistence.xml debe estar accesible en el classpath

```
public static void main(String[] args) {
   // Creación de la entidad
   Vehiculo v = new Vehiculo("7777XGH", "Toyota");
   Propietario p = new Propietario("Pepe");

   // Obtención del Entity Manager
   EntityManagerFactory emf = Persistence.createEntityManagerFactory("vehiculosPU");
   EntityManager em = emf.createEntityManager();

   // La entidad se hace persistente
   // Aquí hay que tratar explícitamente la transacción
   EntityTransaction tx = em.getTransaction();
   tx.begin();
   em.persist(v);
   em.persist(p);
   v.setPropietario(p);
   tx.commit();

   // Liberación de recursos
   em.close();
   emf.close();
}
```

La transacción (comienzo y fin) se maneja de manera explícita

Hay que cerrar el EntityManager de manera explícita

Implementación de entidades con JPA

- Dos estrategias para implementar las entidades en una aplicación Java EE:
  - Codificar las clases entidad, anotarlas y generar automáticamente la BBDD
  - Implementar la BBDD y generar automáticamente las clases entidad

- Los proveedores de persistencia avanzados suelen ofrecer ambas capacidades

- Lo veremos en prácticas en el caso de usar EclipseLink

Elementos de JPA - Jakarta Persistence Query Language (JPQL)

- **JPQL** es un lenguaje definido en Jakarta para realizar consultas en la BBDD
  - Independiente del servidor de BBDD subyacente
  - Sintaxis similar a SQL pero orientada a objetos (centrada en entidades)
    - La clausula FROM se refiere a entidades
    - Los parámetros de búsqueda corresponden a atributos de las entidades
  - Soporta sentencias SELECT, UPDATE y REMOVE

- Ejemplos de sintaxis
  - Búsqueda de todas las instancias de una entidad
    - `SELECT v FROM Vehiculo v`
    - `SELECT p FROM PROPIETARIO p ORDER BY p.edad DESC`
  - Búsqueda de todas las instancias de una entidad restringiendo algún atributo
    - `SELECT v FROM VEHICULO v WHERE v.modelo = 'Toyota Auris'`
    - `SELECT p FROM PROPIETARIO p WHERE p.edad > 35`
    - `SELECT p FROM PROPIETARIO p WHERE p.direccion.localidad = 'Santander'`
    - `SELECT p FROM PROPIETARIO p WHERE p.direccion.localidad = 'Santander' AND p.edad > 35`
  - Búsqueda del atributo(s) de las instancias de una determinada entidad restringiendo un atributo
    - `SELECT v.matricula FROM VEHICULO v WHERE v.modelo = 'Toyota Auris'`

JPQL - Consultas a través de Clase Query

- Las consultas JPQL se realizan a través de objetos de la clase **Query**

- Tres tipos de Query:
  - Dynamic Query: Consulta JPQL que se genera en tiempo de ejecución
  - Named Query: Consulta JPQL estática y no modificable
  - Native Query: Consultas nativas SQL

- La clase EntityManager proporciona métodos para la creación de objetos Query

```
public interface EntityManager {

    // Crea una instancia de una consulta JPQL dinámica
    public Query createQuery(String jpqlString);

    // Crea una instancia de una consulta JPQL con nombre
    public Query createNamedQuery(String jpqlString);

    // Crea una instancia de una consulta SQL nativa
    public Query createNativeQuery(String sqlString);

    ...
}
```

JPQL - Ejecución de consultas dinámicas

- La interfaz **Query** proporciona métodos para la ejecución de consultas

```
public interface Query {

    // Ejecuta una sentencia SELECT y retorna la lista de objetos resultante
    public List<Object> getResultList();

    // Ejecuta una sentencia SELECT y retorna el objeto resultado
    // Lanza NoResultException si no hay resultado
    // Lanza NonUniqueResultException si hay más de un resultado
    public Object getSingleResult();

    // Ejecuta una sentencia UPDATE/REMOVE y retorna el número de entidades afectadas
    public int executeUpdate();

    ...
}
```

```
public class VehiculosDAO () {

   @PersistenceContext("vehiculosPU")
   private EntityManager em;

   public List<Vehiculo> vehiculos() {
       Query q = em.createQuery("SELECT v FROM Vehiculo v");
       return vehiculos = q.getResultList();
   }
}
```

JPQL - Consultas parametrizadas

- JPQL permite definir consultas parametrizadas
  - Parámetros posicionales

```
SELECT v FROM VEHICULO v WHERE v.modelo = ?1
```

  - Parámetros con nombre

```
SELECT v FROM VEHICULO v WHERE v.modelo = :modelo
```

- La interfaz Query proporciona métodos para la ejecución de consultas parametrizadas

```
// Asocia el valor o al parámetro de índice position
public void setParameter(int position, Object o);
// Asocia el valor o al parámetro de nombre name
public void setParameter(String name, Object o);
```

```
public class VehiculosDAO () {

   @PersistenceContext("vehiculosPU")
   private EntityManager em;

   public Vehiculo vehiculoPorMatricula(String matricula) {
       Query q = em.createQuery("SELECT v FROM Vehiculo v WHERE v.matricula = :mat");
       q.setParameter("mat", matricula);
       try {
           return q.getSingleResult();
       } (catch NoResultException e) {
           return null;
       }
}
```

JPQL - Consultas estáticas (Named Queries)

- Consultas estáticas y no modificables
- Más eficientes que las consultas dinámicas
- Definición: A través de la anotación `@NamedQuery` en las entidades
  - Los nombres deben ser únicos a nivel de unidad de persistencia
- Ejecución: Método `createNamedQuery()`

```
@Entity
@NamedQueries( {
    @NamedQuery(name="vehiculos", query="SELECT v FROM Vehiculo v"),
    @NamedQuery(name="vehiculosPorMatricula" query="SELECT v FROM VEHICULO v WHERE v.matricula = :mat'")
})

public class Vehiculo {
   ...
}
```

```
public class VehiculosDAO {

   @PersistenceContext("vehiculosPU")
   private EntityManager em;

   public Vehiculo vehiculoPorMatricula(String matricula) {
       Query q = em.createNamedQuery("vehiculosPorMatricula");
       q.setParameter("mat", matricula);
       try {
          return q.getSingleResult();
       } (catch NoResultException e) {
           return null;
       }
   }
}
```