

5.2 Capa de negocio: EJB 3.0 y Entidades JPA

Tecnologías de JEE 5 para dar soporte a los componentes que forman la **capa de negocio**

1. Soporte de persistencia: mediante entidades JPA (*Java Persistence API*)

- Objetos Java cuyo estado (valores de atributos) se almacena en BDs permanentes
- Representan los *objetos de dominio persistentes* manejados por la aplicación JEE
 - Encapsulan los datos manejados por la aplicación cuyo almacenamiento gestiona la capa de datos
- JPA ofrece facilidades de *mapeo Objeto-Relacional* que ocultan el manejo de bases de datos SQL
- No son exclusivos de JEE ni de la capa de negocio ⇒ puede usarse JPA en aplicaciones JSE ("de escritorio")

2. Implementación de la lógica de negocio: mediante componentes EJB (*Enterprise JavaBeans*)

- Componentes gestionados por el contenedor de EJB que implementan la lógica de la aplicación
 - Gestionan el tratamiento y flujo de datos en la parte del servidor
 - Ofrecen a sus clientes (capa de presentación web, aplicaciones JSE *stand-alone*, otros EJBs, servicios Web) una *interfaz de negocio* con las operaciones que estos pueden ejecutar para acceder y manejar esos datos
- Son objetos remotos accesibles mediante RMI/IIOP (RMI sobre Corba) cuya creación y gestión (acceso concurrente, seguridad, transacciones, ...) es responsabilidad del contenedor EJB
 - EJBs abstraen al programador de esos detalles de bajo nivel típicos de aplicaciones distribuidas

Objetivo de las especificaciones JPA y EJB3: simplificar el desarrollo

- Uso de **POJOs** (*plain old java objects*): objetos Java simples (sin restricciones especiales)
- Uso de **anotaciones** para especificar metadatos y "configurar" los objetos de negocio manejados por la aplicación
- **Inyección de dependencias** para facilitar la creación de las instancias de objetos
 - Separa la definición de una clase de la instanciación de los objetos de los que depende
 - Programador se despreocupa de incluir código en sus componentes para añadir a referencias a otros objetos que puedan necesitar ⇒ lo hará el contenedor en base a la información de las anotaciones
 - Elimina la necesidad de ficheros de configuración (descriptores de despliegue: *deployment descriptors*)

5.2.1 Enterprise JavaBeans, versión 3.0

Componentes del lado del servidor que encapsulan la lógica de la aplicación y que son gestionados por el contenedor de EJBs

- Cada Enterprise JavaBean encapsula parte de la lógica de negocio de la aplicación
- Paquete `javax.ejb.*`: interfaces, anotaciones, excepciones, ...

Ventajas que aportan

- Permiten la distribución de la lógica entre distintas máquinas manteniendo la transparencia
 - Uso de directorios JNDI para nombrar y acceder a los componentes y recursos
- Manejo de transacciones para controlar accesos concurrentes asegurando la integridad de los datos
 - Controladas por el contenedor de EJB
- Simplifican el desarrollo de aplicaciones que atiendan una gran variedad de clientes (clientes web o "de escritorio" (*stand-alone*), cliente locales o remotos, etc)
- Diseño orientado a componentes → reusabilidad y abstracción
 - El desarrollador se concentra en la lógica de negocio y utiliza los servicios proporcionados por el contenedor

Servicios básicos a los que da soporte el contenedor EJB

- Persistencia mediante JPA (*Java Persistence API*)
- Invocación remota de métodos mediante RMI/IIOP
- Procesamiento de transacciones y control de la concurrencia
 - Cada invocación de un método de un EJB forma parte de una transacción
- Procesamiento de eventos (*API Java Message Service [JMS]*)
- Servicios de directorio: JNDI (*Java Naming and Directory Interface*)
- Seguridad: autenticación, control de acceso y privilegios y criptografía (JAAS [*Java Authentication and Authorization Service*] y JCE [*Java Cryptography Extension*])
- Publicación de los métodos de negocio de los EJBs como servicios Web (mediante JAX-WS)

(a) Tipos de EJB

Message-Driven EJBs (EJBs dirigidos por mensajes)

- Permiten el procesamiento de mensajes (operaciones) de forma asíncrona (recepción y tratamiento de eventos JMS)
- Actúan como *listeners* (escuchadores) de eventos JMS (*Java Message Service*)
 - Implementan el interfaz `MessageListener`
- Se "suscriben" a una cola (*queue*) quedando a la espera y se activan cuando se recibe un mensaje dirigido a dicha cola
 - Esos mensajes JMS pueden ser enviados por cualquier componente de una aplicación Java EE (clientes, componentes Web, otros EJBs)
 - Los "clientes" de un *Message-Driven EJB* no invocan directamente sus métodos, simplemente envían mensajes JMS

Session EJBs (EJBs de sesión)

- Representan *procesos de negocio* (funcionalidades de la aplicación) ⇒ implementan un *interfaz de negocio* (*business interface*)
- Gestionan la interacción con los clientes y encapsulan el flujo y manipulación de la información en el servidor
 - Sirven a los clientes como una "fachada" de los servicios proporcionados por otros componentes disponibles en el servidor (patrón de diseño *Facade*)
- Ofrecen operaciones síncronas (petición-respuesta)
 - Procesos de negocio ejecutados en respuesta a una solicitud del cliente
- Contenedor de EJBs crea e inicializa sus instancias (inyectándoles las referencias necesarias) y las asigna a los clientes a medida que estos los van requiriendo (*pool* de EJBs)
- En un instante de tiempo dado, sólo un cliente tiene acceso a los métodos de la instancia del EJB (control de concurrencia)
 - Contenedor de EJBs garantiza que cada método del EJB se ejecuta dentro de una *transacción atómica*

(b) EJBs de sesión

Tipos de EJB de sesión: en función la interacción con el cliente

- **Stateful EJB:** tienen un *estado conversacional* dependiente del cliente
 - Cada llamada está condicionada por las anteriores (importa el orden)
 - Los valores de sus atributos se mantienen entre las distintas invocaciones de un mismo cliente
 - Representan el estado de la interacción con un cliente específico
 - Cada instancia de un *stateful* EJB "atiende" invocaciones de un único cliente
 - En función del *estado conversacional* el resultado de las operaciones realizadas podrá variar
 - Estado conversacional no es persistente (no se mantiene cuando el cliente "libera" el EJB)
 - Valores de atributos no premanentes: finalizada sesión el estado desaparece
 - Por eficiencia el contenedor de EJBs puede decidir detener temporalmente la ejecución un *stateful* EJB, almacenando temporalmente su estado en disco para recuperarlo al volver a la ejecución
 - Ejemplo: objeto "*carrito de la compra*" en el cual se van añadiendo productos
- **Stateless EJB:** no almacenan datos de los clientes que los invocan
 - Cada llamada es independiente de las demás
 - Para el cliente todos los *stateless* EJB son idénticos
 - Estado de un *stateless* EJB no contiene información específica de un cliente
 - ◇ no se garantiza que los valores de sus atributos se mantengan entre llamadas
 - *Stateless* EJBs realizan tareas genéricas e idénticas para todos los clientes
 - Pueden implementar Servicios Web (anotaciones @WebService y @WebMethod)
 - Ejemplo: objeto "*gestor de artículos*": altas, bajas, modificaciones de artículos

Interfaces en EJB de sesión

- *Interfaz de negocio* de un EJB define el tipo de acceso permitido a sus clientes (Define la visión del EJB que tiene el cliente)
- Un EJB de de sesión puede tener varios interfaces de negocio con distintos tipos de acceso

Cientes remotos: pueden ejecutarse en máquinas virtuales distintas a la del EJB que invocan

- Pueden ser: componentes web (servlet, JSP), clientes remotos u otros EJBs
- Para el cliente remoto la localización del EJB es transparente

Cientes locales: deben ejecutarse en la misma máquina virtual que el EJB que invocan

- Pueden ser: componentes web (servlet, JSP) u otros EJBs
- Para el cliente local la localización del EJB no es transparente

(c) Definición de EJBs de sesión

■ (1) Definición del interfaz de negocio

- Interfaces de negocio → interfaces Java decorados con las anotaciones `@Remote` o `@Local` para indicar las restricciones de acceso
 - El interfaz y todos los métodos definidos deben de ser `public`
 - Mismas restricciones para los tipos de parámetros y valores de retorno que Java RMI
- Si no se indica con una anotación en el interfaz o en el EJB se asume todos los interfaces son locales por defecto
- Ejemplos:

`@Remote`

```
public interface CarritoCompra {  
    public void inicializarCarrito(Cliente c);  
    public void anadirArticulo(Articulo a, int cant);  
    public void eliminarArticulo(Articulo a);  
    public void vaciarCarrito();  
};
```

`@Local`

```
public interface GestorClientes {  
    public void alta(Cliente c);  
    public void baja(Cliente c);  
    public void actualizar(Cliente c);  
    public cliente buscarDNI(String DNI);  
};
```

- **Nota:** Llamadas remotas mucho más costosas que locales
 - En general se tiende a que pasar parámetros "grandes" ⇒ reducir núm. de llamadas pasando mayor cantidad de información

■ (2) Implementación de EJBs de sesión

- Clases de implementación marcadas con las anotaciones `@Stateless` ó `@Stateful` según del tipo de *bean*
- Se indican los interfaces de negocio implementados (`implements`)

`@Stateless`

```
public class GestorClientesBean implements GestorClientes { ... }
```

`@Stateful`

```
public class CarritoCompraBean implements CarritoCompra { ... }
```

- Opcionalmente se puede asociar un nombre JDNI al EJB para su acceso (parámetro `mappedName="..."` de ambas anotaciones)

`@Stateless(mappedName="gestor_clientes") ...`

`@Stateful(mappedName="carrito_compra") ...`

Nota: si no se indica nada el contendor EJB asignará uno

- Opcionalmente se puede explicitar el nombre y el tipo de acceso de los interfaces de negocio soportados

`@Stateless @Local(GestorClientes.class) ...`

`@Stateful @Remote(CarritoCompra.class) ...`

- Se pueden marcar métodos con anotaciones especiales para indicar que sean invocados por el contenedor EJBs en momentos concretos del ciclo de vida del *bean*
 - `@PostConstruct`: una vez que el contenedor instancia el EJB y le inyecta dependencias (antes de recibir su primera invocación)
 - `@PreDestroy`: justo antes de eliminar definitivamente el EJB
 - `@PrePassivate`: (sólo *stateful*) justo antes almacenar el estado en disco temporalmente
 - `@PostPassivate` (sólo *stateful*) justo de recuperar el estado almacenado en disco temporalmente
- Método anotado con `@Remove` puede ser invocado por clientes para eliminar una instancia del EJB del contenedor
- **(3) Empaquetado y despliegue**
 - **Empaquetado de EJBs:** Creación de fichero JAR conteniendo:
 - *Bytecode* (ficheros `.class`) de

{

clases de implementación de EJBs
interfaces de negocio
otras clases auxiliares necesarias
 - ◇ Ficheros `.class` estructurados en paquetes (subdirectorios) según corresponda
 - Directorio `META-INF` conteniendo los ficheros de configuración exigidos por el contenedor o que pudieran ser necesarios
 - ◇ descriptor de despliegue [`ejb-jar.xml`]
 - ◇ descriptor de persistencia [`persistence.xml`]
 - ◇ `MANIFEST.MF`
 - ◇ otros ...
 - Librerías adicionales (ficheros JAR)
 - **Despliegue de EJBs:** El fichero JAR resultante se puede desplegar directamente en el contenedor JEE o incluirlo como un módulo EJB de una Aplicación JEE (paquetes *EAR*)
 - Al ser desplegado un EJB en un contenedor se le asocia un nombre según lo indicado en sus anotaciones o en su descriptor de despliegue
 - Ese nombre se registra en el servidor de nombres del contenedor EJB, accesible por JNDI.
 - Una vez desplegado los clientes pueden acceder al EJB usando ese nombre e invocar sus métodos

(c) Acceso e invocación de EJBs de sesión

- Para utilizar métodos de un EJB desplegado es necesario que el cliente cuente con una referencia al mismo (*stub*), que será la que gestione las invocaciones sobre RMI/IIOP
- **Opción 1:** Inyección de dependencias por parte del contenedor JEE
 - Uso de la anotación `@EJB` acompañando al atributo donde se mantendrá la referencia al EJB
 - El tipo del atributo/referencia será el nombre del interfaz de negocio que se desea invocar
 - El contenedor usará el nombre por defecto del EJB (o el que se especifique en el parámetro `@EJB(mappedName="...")`) para consultar al servidor de nombres JNDI, inyectando en ese atributo la referencia encontrada
 - Inyección de referencias con `@EJB` sólo es posible dentro de un entorno de ejecución JEE
 - *Contenedor de servlets*: es posible inyectar EJBs para invocarlos desde servlets y páginas JSP
 - *Contenedor de EJBs*: es posible inyectar EJBs para invocarlos desde otros EJBs
 - *Contenedor de clientes JEE*: clientes Java *stand-alone* ejecutándose en el contexto de un contenedor de clientes JEE pueden obtener e invocar EJBs
 - ◇ Es necesario disponer de las API de JEE 5 para inyectar estas dependencias
 - ◇ Ejemplo: `appclient` en *GlassFish/Sun Application Server*
 - ◇ **Importante:** Los contenedores de clientes JEE sólo pueden inyectar EJBs sobre atributos de tipo `static` de la clase principal donde se incluya el método `main()`
 - Ejemplo:

```
public class ClaseLlamadora {
    @EJB(mappedName="carrito_compra") // mappedName es opcional
    private CarritoCompra carrito;

    ...

    public void metodoLlamador(String args[]) {
        ...
        carrito.inicializar(...);
        carrito.anadir(...);
        ...
    }
}
```

- **Opción 2:** Consulta JNDI empleando el nombre asignado al EJB
 - Siempre es posible consultar el servicio JNDI presente en todos los servidores aplicaciones JEE
 - Se usa el API de JNDI → paquete `javax.naming.*`
 - Interfaz *InitialContext*, métodos *lookup(...)*
 - Se obtiene una referencia a un EJB a partir del nombre asignado (en la anotación, en el descriptor de despliegue o del nombre por defecto asignado por el contenedor)
 - Este era el modo en que se hacía en las versiones de EJB 2.x y anteriores
 - El nombre por defecto con el que se registran los EJBs no está estandarizado y cada servidor de aplicaciones organiza el directorio de nombres de forma distinta
 - ◇ GlassFish usa el nombre de la clase completamente cualificado (lista de paquetes)
 - Consultas JNDI desde clientes *stand-alone*
 - Es necesario configurar las *properties* de la máquina virtual cliente para indicar la dirección y el puerto de escucha del servicio JNDI del contenedor donde está desplegado el EJB (dependiente de cada contenedor)
- Nota:** con GlassFish las propiedades a especificar son:
- ◇ `org.omg.CORBA.ORBInitialHost` → nombre servidor aplicaciones JEE
 - ◇ `org.omg.CORBA.ORBInitialPort` → 3700
- Pueden indicarse como parámetros de la JVM al arrancar (`-Dparametro=valor`) o con `System.setProperty("parametro", valor)`;
- Con esa información será posible obtener un *InitialContext* donde buscar (*lookup()*) el nombre del EJB deseado

- **Ejemplo:**

```
import javax.naming.InitialContext;

public class EjemploCliente {
    public static void main(String args[]) {
        ...
        InitialContext ic = new InitialContext();
        CarritoCompra carrito = (CarritoCompra) ic.lookup("carrito_compra");
        carrito.inicializar(...);
        carrito.anadir(...);
        ...
    }
}
```


5.2.2 Persistencia: Entidades JPA

JPA (*Java Persistence API*) especifica un marco para definir e implementar mapeos Objeto-Relacional

- Mapeo Objeto-Relacional (*Object/Relational Mapping - ORM*): permite manejar datos en BD relacionales mediante objetos
 - Simplifica el desarrollo de aplicaciones basadas en BD relacionales (todo son objetos)
 - Ofrece persistencia de objetos de forma transparente
 - Automatiza y unifica las tareas de mantenimiento de datos relacionales
 - Evita que el código sea dependiente de características específicas de un gestor de BD concreto
- JPA unifica la forma en que funcionan las distintas soluciones que proveen el mapeo ORM
 - JPA define la forma en que se describe ese mapeo ORM
 - JPA define el interfaz con el proveedor de persistencia que será quien realmente realice la transformación e interactúe con el gestor de la BD
- Elementos definidos por JPA:
 - Especificación del mapeo O/R mediante anotaciones o ficheros XML
 - Juego de anotaciones para especificar el mapeo entre tablas y *entidades JPA* (objetos Java persistentes)
 - Tabla → clase de *entidades JPA* (*entities*)
 - Tupla → instancia de una entidad JPA
 - Columna → atributo de una instancia
 - Especificación el interfaz *EntityManager* con las operaciones de persistencia sobre las entidades JPA
 - Define el ciclo de vida de las entidades
 - Gestiona los detalles de la interacción con el motor de BD (localización, usuarios, adaptadores JDBC, ...)
 - Especificación del lenguaje de consulta JPQL (*Java Persistence Query Language*)
 - Similar a SQL: maneja entidades en lugar de tablas
- Reemplaza los *EntityBeans*: mapeo O/R de EJB 2.x
- JPA no es exclusivo de Java EE, puede usarse en aplicaciones Java independientes

(a) Especificiación de Entidades JPA

Cada tabla de la BD estará representada en Java por una *Entidad*

- Una entidad es una clase Java marcada con la anotación `@Entity`
 - Cada instancia de esa clase Entidad mapeará una tupla de esa tabla
 - Entidades deben ser JavaBeans $\left\{ \begin{array}{l} 1 \text{ constructor vacío} \\ \text{accesos con } get() \text{ y } set() \end{array} \right.$
 - Todos los atributos de la Entidad, salvo los marcados `@Transient`, son persistentes y se corresponderán con columnas de la tabla mapeada
 - Cada instancia de una Entidad es única y debe estar identificada por uno o más de sus atributos
 - Atributos persistentes permitidos:
 - No se permiten atributos estáticos o de tipo *final*
 - Tipos primitivos: *int*, *float*, etc
 - Objetos para tipos primitivos: *java.lang.Integer*, *java.lang.Float*, etc
 - *java.lang.String*, *java.math.BigInteger*, *java.math.BigDecimal*
 - *java.util.Date*, *java.util.Calendar*, *java.sql.Date*, *java.sql.Timestamp*
 - Otras entidades (relaciones entre entidades)
 - Clases embebibles: marcadas con `@Embeddable`
 - Colección y conjuntos de entidades: *java.util.Collections*, *java.util.Sets*
- Anotación `@Table`: especifica la tabla relacionada con la entidad (si no coinciden sus nombres)
 - `name`: nombre de la tabla (por defecto coincide con el de la entidad)
 - `catalog`: nombre del catálogo
 - `schema`: nombre del esquema
 - `uniqueConstraints`: restricciones de unicidad
- Anotaciones para el **mapeo de atributos**

Pueden asociarse $\left\{ \begin{array}{l} \text{a la declaración del atributo en la Entidad} \\ \text{al método } get() \text{ de ese atributo} \end{array} \right.$

 - `@Column`: especifica una columna de la tabla a mapear sobre un atributo de la entidad
 - Es opcional: si no se especifica lo contrario se asume que todos los atributos de la entidad mapean una columna de la BD con su mismo nombre y tipo
 - ◇ *name* (nombre de la columna)
 - ◇ *unique* (atributo con valor único)
 - ◇ *nullable* (permite nulos)
 - ◇ *insertable*, *updatable*
 - ◇ *length*, *precision*, *scale*, ...
 - `@Transient`: especifica un atributo no persistente (no se mapeará a la BD)

- **@Id**: especifica que se trata de un atributo identificador
 - JPA exige que toda *Entity* tenga un atributo identificador
- **@GeneratedValue**: asociado a atributos clave primaria, indica como se debe generar
 - *strategy* estrategia para la generación de clave;
 - ◇ *GenerationType.AUTO* (valor por defecto, contenedor decide la estrategia en función de la BD)
 - ◇ *GenerationType.IDENTITY* (utiliza un contador autoincremental gestionado por la BD)
 - ◇ *GenerationType.SEQUENCE* (utiliza una secuencia)
 - ◇ *GenerationType.TABLE* (utiliza una tabla de identificadores)
 - *generator*: forma en la que se genera la clave
- **@SequenceGenerator** (opc.) especifica el generador de claves primarias
- **@TableGenerator** (opc.) especifica una tabla de claves primarias
- **@EmbeddedId**: indica una clave primaria con múltiples campos, almacenados en una clase *Embebida* (marcada con anotación **@Embebedable**)
- **@IdClass**: especifica una clave primaria compuesta, mapeada a varios atributos de la entidad.
- **@Temporal**: especifica el mapeo de campos de BD que almacenan fechas
- **@Lob**: especifica el mapeo de campos de BD que almacenan datos binarios (BLOB)
- **@Basic**: permite especificar detalles adicionales del mapeo de tipos básicos
- **@JoinColumn**: especifica un atributo que actúa como *clave foránea* de otra entidad
 - *name* nombre de la columna que actúa como *clave foránea* en la tabla
 - *referenced* nombre de la columna referencia en la tabla relacionada (opc)
 - *table* nombre de la tabla relacionada que contiene la columna (opc)
 - *unique, nullable, insertable, updatable, ...* (opc)
- **@JoinColumns** permite agrupar agrupar varias **@JoinColumn**

■ Especificación de **relaciones entre Entidades** (interpretadas desde el punto de vista de la Entidad definida)

Parámetros comunes:

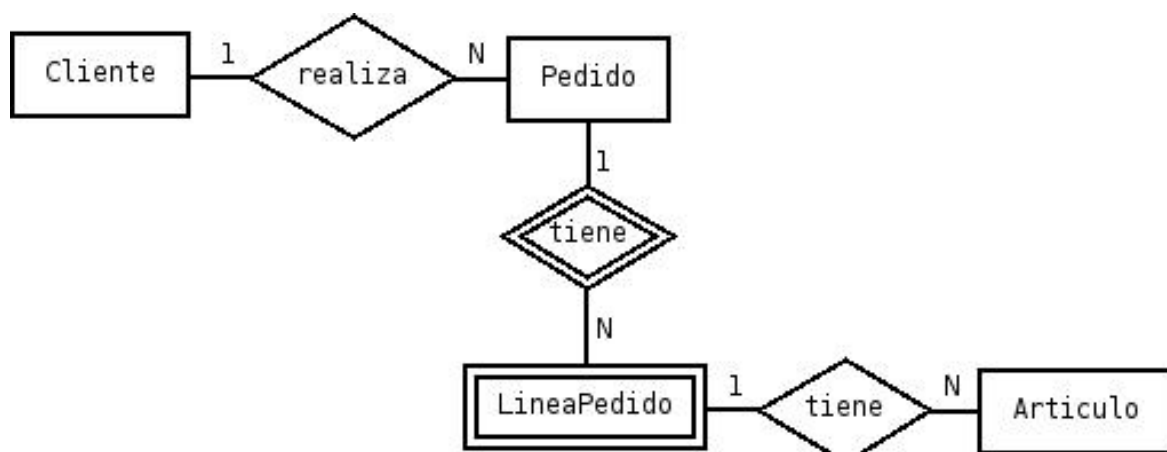
- *cascade*: indica las operaciones (ALL, PERSIST, MERGE, REMOVE, REFRESH) que se aplicarán en cascada sobre las Entidades relacionadas
- *fetch*: forma en que se cargan los datos
 - *FetchType.LAZY* carga las entidades relacionadas cuando se accede al atributo (carga perezosa)
 - *FetchType.EAGER* carga completa de todas las entidades relacionadas
- *optional* relación opcional
- *mappedBy* campo de la Entidad relacionada que posee la relación (únicamente se especifica en uno de los lados de la relación [el que no tiene clave foránea])

Tipos de relaciones:

- @OneToOne (relación 1:1) indica un atributo que está asociado con una tupla de otra Entidad
 - Puede requerir una anotación @JoinColumn (marca el atributo como clave foránea)
 - Puede requerir especificar el parámetro *mappedBy*
- @OneToMany (relación 1:N) indica un atributo que está asociado con varias tuplas de otra Entidad (lado N)
 - Aplicada sobre atributos de tipo *Set<Entidad>* o *Collection<Entidad>*
 - Requiere especificar el parámetro *mappedBy* (clave foránea en el extremo N)
- @ManyToOne (relación N:1) indica un atributo (lado N) asociado con una tupla de otra Entidad (lado 1)
 - Requiere una anotación @JoinColumn para marcar el atributo como clave foránea
- @ManyToMany (relación N:M) indica un atributo que tiene una relación N:M con tuplas de otra Entidad
 - Aplicada sobre atributos de tipo *Set<Entidad>* o *Collection<Entidad>*
 - Requiere especificar la tabla que implementa la relación con sus claves foráneas con la anotación @JoinTable
- @JoinTable mapeo de una relación *ManyToMany* o una *OneToMany* que haga uso de una tabla
 - *name*: nombre de la tabla join que soporta la relación
 - *catalog*, *schema*
 - *joinColumns*: columnas con las claves foráneas de la primera tabla de la relación
 - *inverseJoinColumns*: columnas con las claves foráneas de la segunda tabla de la relación
 - *uniqueConstraints*
- @MapKey especifica el mapeo sobre colecciones de tipo *java.util.Map* (tablas Hash)
- @OrderBy especifica el orden de las colecciones mapeadas

■ Otras especificiones de mapeos

- Mapeo de herencia entre tablas/Entidades
 - @Inheritance define el tipo de herencia
 - ◇ *strategy*: SINGLE_TABLE, JOINED, TABLED_PER_CLASS
 - @DiscriminatorColumn
 - @DiscriminatorValue
 - @MappedSuperclass



Ejemplos Mapeo O/R

```
@Entity @Table(name = "ARTICULO")
public class Articulo implements Serializable {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "NUM_ARTICULO") private Integer numArticulo;
    @Column(name = "DESCRIPCION") private String descripcion;
    @Column(name = "PRECIO") private float precio;
    ...
}

@Entity @Table(name = "PEDIDO")
public class Pedido implements Serializable {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "NUM_PEDIDO")
    private Integer numPedido;
    @Column(name = "FECHA") @Temporal(TemporalType.DATE)
    private Date fecha;
    @JoinColumn(name = "NUM_CLIENTE", referencedColumnName = "NUM_CLIENTE")
    @ManyToOne
    private Cliente cliente;
    @OneToMany(mappedBy="pedido", cascade=CascadeType.ALL, fetch=FetchType.EAGER)
    private Collection<LineaPedido> lineasPedido;
    ...
}

@Entity @Table(name = "CLIENTE")
public class Cliente implements Serializable {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "NUM_CLIENTE") private Integer numCliente;
    @Column(name = "NOMBRE") private String nombre;
    @Column(name = "NIF") private String nif;
    @Column(name = "DIRECCION") private String direccion;
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "cliente")
    private Collection<Pedido> pedidos;
    ...
}

@Entity @Table(name = "LINEA_PEDIDO")
public class LineaPedido implements Serializable {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "NUM_LINEA") private Integer numLinea;
    @Column(name = "CANTIDAD") private float cantidad;
    @Column(name = "PRECIO") private float precio;
    @JoinColumn(name = "NUM_ARTICULO", referencedColumnName = "NUM_ARTICULO")
    @ManyToOne
    private Articulo articulo;
    @JoinColumn(name = "NUM_PEDIDO", referencedColumnName = "NUM_PEDIDO")
    @ManyToOne
    private Pedido pedido;
    ...
}
```

(b) Interfaz *EntityManager* (EM)

EntityManager (EM) ofrece los métodos con los que se interactúa con el entorno de persistencia

- Control de transacciones
- Gestión del ciclo de vida de las Entidades
- Creación de consultas (*Query*)
- Otros: gestión de cachés, gestión identidad de las Entidades

Cada *EntityManager* estará asociado a una *Persistence Unit*

- Definen los contextos de persistencia de la aplicación
- Especifican el proveedor de persistencia a emplear (*toplink*, *hibernate*, *JDO*, etc) y el *DataSource* con información de la BD a utilizar
- Descritos en el fichero *persistence.xml*

- Ubicado en directorios $\left\{ \begin{array}{l} \text{META-INF/ en paquetes JAR de módulos EJBs} \\ \text{WEB-INF/classes/META-INF en paquetes WAR de módulos Web} \end{array} \right.$

```
<persistence>
  <persistence-unit name="EjemploPedidosPU" transaction-type="JTA">
    <jta-data-source>jdbc/EjemploPedidos</jta-data-source>
    <properties>
      <property name="toplink.ddl-generation" value="create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

- Elementos

- *persistence-unit* name: nombre del contexto de persistencia
- *jta-data-source*, *non-jta-data-source*: nombre JNDI del *DataSource*
 - ◇ El *DataSource* de la correspondiente BD deberá haber sido definido previamente

```
<resources>
  <jdbc-resource enabled="true" jndi-name="jdbc/EjemploPedidos"
    object-type="user" pool-name="mysqlPool"/>
  <jdbc-connection-pool ... name="mysqlPool" ...
    ... res-type="javax.sql.DataSource" ... >
    <property name="serverName" value="localhost"/>
    <property name="portNumber" value="3306"/>
    <property name="databaseName" value="scs_pedidos"/>
    <property name="User" value="scs"/>
    <property name="Password" value="scs"/>
    <property name="URL" value="jdbc:mysql://localhost:3306/scs_pedidos"/>
    <property name="driverClass" value="com.mysql.jdbc.Driver"/>
  </jdbc-connection-pool>
</resources>
```

- *properties*: configuración específica del proveedor de persistencia JPA

Obtención de referencias al EM

- Inyección de referencia al *EntityManager* mediante la anotación `@PersistenceContext`

```
@PersistenceContext
EntityManager em;
```

Se debe indicar el nombre de la *PersistenceUnit* a utilizar si hay más de una definida

- Mediante la factoría *EntityManagerFactory* indicando el nombre del contexto de persistencia configurado en `persistence.xml`

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

Se puede inyectar un *EntityManagerFactory*

```
@PersistenceUnit
EntityManagerFactory emf;
```

Métodos del interfaz EM

Todas las operaciones del EM deben realizarse dentro de una transacción

- En EJBs o componentes Web (servlets, JSPs) las transacciones son gestionadas por el Servidor de Aplicaciones
- Desde el propio EM se puede acceder al gestor de transacciones para iniciar y finalizar transacciones (*getTransaction().begin()* y *getTransaction().commit()*)
 - Método empleado en aplicaciones de escritorio JSE

Ciclo de vida de las Entidades

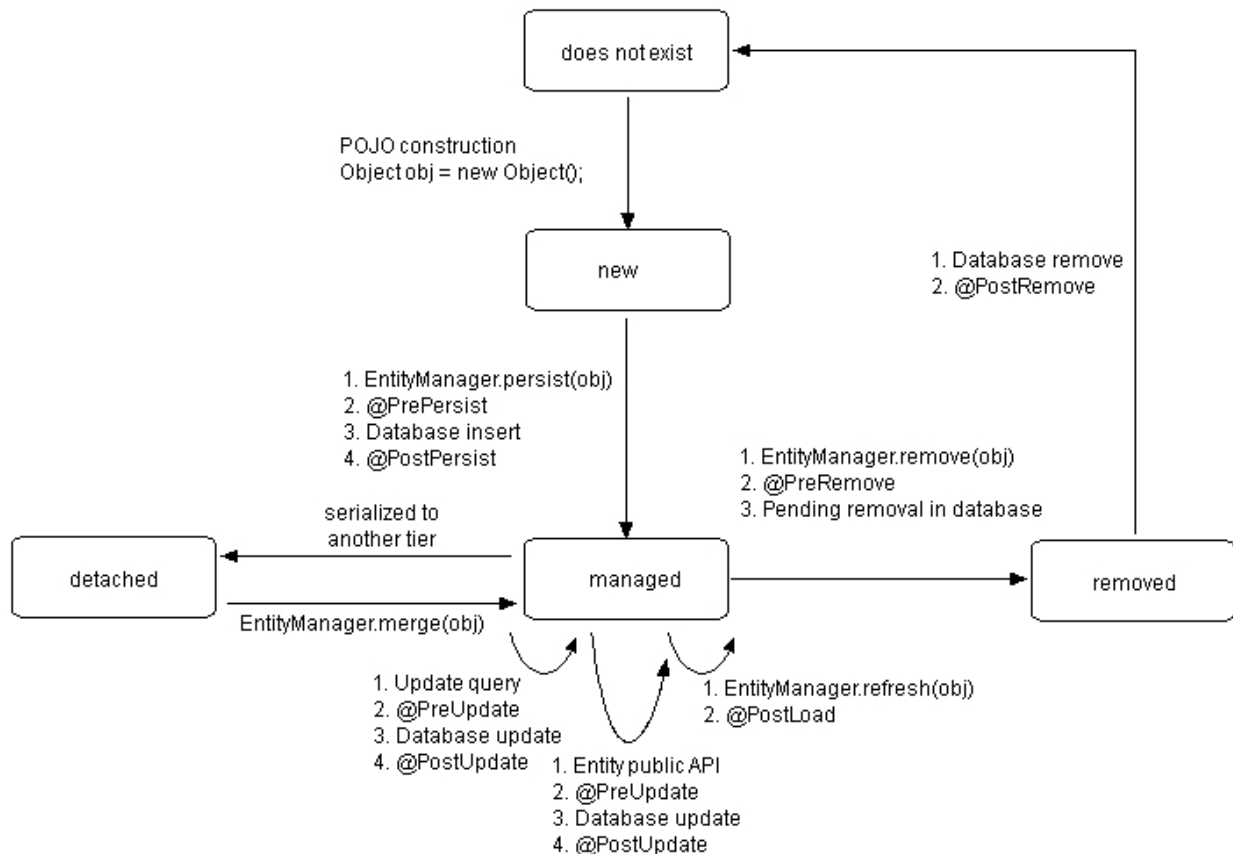
Las modificaciones sobre *instancias gestionadas* por el EM realizadas dentro de una transacción se reflejarán automáticamente en la BD

- Instancias gestionadas por el EM pertenecen a su *contexto de persistencia*
- Escritura efectiva en BD cuando se complete satisfactoriamente la transacción

Estados del ciclo de vida de las Entidades:

- *new*: instancia de una Entidad recién creada
 - Sólo reside en memoria, el EM no tiene constancia de su existencia
- *managed*: instancia de una Entidad gestionada por el EM
 - Forma parte del contexto de persistencia manejado por el EM
 - Los cambios que se produzcan sobre ella se harán persistentes en la BD (al finalizar la transacción actual o al llamar a *flush()*)

- *detached*: instancia de una Entidad desligada del contexto de persistencia
 - Está fuera del contexto de persistencia
 - Suelen ser instancias de Entidades pasadas como parámetro serializado entre distintas capas de la aplicación (llamadas remotas)
 - Es necesario llamar a *merge()* para tener conocimiento de sus modificaciones
- *removed*: instancia *managed* sobre la que se ha llamado al método *remove()*



Gestión del ciclo de vida de las Entidades (métodos del EM)

`T find(Class<T>e, Object id)` : busca una instancia de una Entidad usando su clave primaria

- Devuelve una instancia con todos sus atributos rellenos (excepto los que soporten relaciones marcadas con *FetchType.LAZY*)

`persist(Object e)` : almacena una nueva instancia de una entidad en la base de datos

- Pasa a formar parte del conjunto de instancias gestionadas por el EM

```
Articulo a = new Articulo();
a.setDescripcion("CPU AMD");
a.setPrecio(120);
```

```
em.persist(a);
...
```


merge(T e) : actualiza las modificaciones sobre instancia de una entidad, devolviendo una nueva instancia de esa entidad (instancia gestionada por el EM)

- Permite incorporar modificaciones realizadas sobre instancias que salieron del contexto de persistencia del EM
- Típicamente serán modificaciones realizadas sobre copias serializadas pasadas en invocaciones remotas a otros contextos

SERVIDOR	CLIENTE
<pre>Articulo buscar(int id) { ... Articulo a = em.find(Articulo.class, id); return(a); }</pre>	<pre>... Articulo a = servidor.buscar(105); a.setPrecio(150); a.setDescripcion(a.getDescripcion() + "(en oferta)"); servidor.modificar(a); ...</pre>
<pre>void modificar(Articulo a) { ... em.merge(a); ... }</pre>	

remove(Object e) : elimina una instancia de una entidad

```
Cliente c = em.find(Cliente.class, 104);  
em.remove(c);
```

refresh(Object e) : actualiza los valores de una instancia con su contenido de la BD

- Refresca todos los atributos con carga de tipo *FetchType.EAGER*
- Desecha los valores de campos con carga de tipo *FetchType.LAZY*

flush() : sincroniza las instancias gestionadas por el EM actualizando la BD

(c) Lenguaje de Consulta JPQL

JPQL: *Java Persistence Query Language*

- Ofrece una sintaxis similar a la de SQL para realizar consultas sobre entidades JPA
- Clausula FROM maneja nombres de Entidades (no tablas)
 - Hace uso de las relaciones mapeadas 1:N o N:M para definir los JOINS entre tablas relacionadas
 - Sintaxis: IN ó JOIN
 - No necesario con relaciones 1:1 o N:1
- Nombres de campos usan notación Java: *Entidad.atributo*
 - Los elementos manejados en WHERE o devueltos por SELECT son siempre objetos
- Se permiten consultas de tipo SELECT, UPDATE y DELETE

Consultas (objetos de tipo *Query*) son creadas por el EM

`Query createQuery(String q)` : crea una consulta JPQL

`Query createNamedQuery(String q)` : crea una consulta nombrada, vinculada a una Entidad y definida con la anotación `@NamedQuery`

`Query createNativeQuery(String q)` : crea una consulta SQL

Ejecución de consultas

`Object getSingleResult()` : devuelve un objeto con el resultado único de la consulta

- Devuelve un array (`Object[]`) cuando SELECT especifica varios valores

`List getResultList()` : devuelve una colección *List* con los resultados de la consulta

- Se puede especificar el rango de valores a devolver (útil para paginar resultados)
- `Query setFirstResult(int)`: inicio del rango
- `Query setMaxResults(int)`: tamaño del rango

`int executeUpdate()` : ejecución de Querys de tipo UPDATE o DELETE

Consultas parametrizadas

- Notación: $\left\{ \begin{array}{l} \text{:nombre (parámetros con nombre)} \\ \text{?3 (parámetros posicionales)} \end{array} \right.$
- Métodos de la clase *Query*
`Query setParameter(String s, Object o)` : asocia valor a un parámetro nombrado
`Query setParameter(int pos, Object o)` : asocia valor a un parámetro posicional

Ejemplos:

- Lista de artículos cuya descripción empiece por CPU, "paginados" de 10 en 10

```
Query q1 = em.createQuery("SELECT a FROM Artículo a "+  
                           "WHERE a.descripcion LIKE \"CPU%\" ");
```

```
List<Artículo> lista;  
int i = 0;  
do {  
    q1.setFirstResult(i);  
    q1.setMaxResult(10);  
    lista = q1.getResultList();  
    i = i + 10;  
    ...  
} while (lista != null);
```

- Lista de pares (Pedido, importe total pedido) entre 2 fechas

```
Query q2 = em.createQuery("SELECT p, SUM(l.cantidad * l.articulo.precio)"+  
                           "FROM Pedido p JOIN p.lineas"+  
                           "WHERE p.fecha BETWEEN :fechaIni AND :fechaFin"+  
                           "GROUP BY p.numPedido);  
q2.setParameter("fechaIni", new Date(.....));  
q2.setParameter("fechaFin", new Date(.....));  
Iterator pares = q2.getResultList().iterator();  
for(Object[] par: pares) {  
    pedido = (Pedido) par[0];  
    total = (Float) par[1];  
    ...  
}
```