Formación JPA

<?xml version="1.0"?><DocumentFileOSQ xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"> <GraphicCharterDefinitionId>0</GraphicCharterDefinitionId> <TemplateBaseTypeId>0</TemplateBaseTypeId> <CompanyId>1</CompanyId> <ConfidentialId>0</ConfidentialId> <ConfidentialDescription /> <CountryId>20</CountryId> <PageSizeId>1</PageSizeId> <PageOrientationId>1</PageOrientationId> <PrePrintedStationary>false</PrePrintedStationary> <Project>GEMIS</Project> <Reference>20170124-170150-esgomez</Reference> <TemplateType>3</TemplateType> <CultureId>fr-FR</CultureId> <LanguageId>2</LanguageId> <Customer>Centro de Servicio Valencia</Customer> <DocumentDate>2018-11-26T11:51:34.0428357+01:00</DocumentDate> <Saved>true</Saved> <IsValid>true</IsValid> <FirstPageCover>false</FirstPageCover> <IsNew>false</IsNew> <CurrentVersion>1.00</CurrentVersion> <DocumentType>Solution Requirements Specification</DocumentType> <DocumentTypeId>-1</DocumentTypeId> <Entity /> <HasDistributionList>false</HasDistributionList> <HasForeword>false</HasForeword> <Recipient>Customer Stakeholders (including at least the IT PM and the Stakeholder and User manager)</Recipient> <Title>Spécifications des besoins de la solution - GEMIS</Title> <Status>2</Status> <StatusDescription>Projet</StatusDescription> <SetEdition>false</SetEdition> <SetVersion>false</SetVersion> <TemplateEditor>EN\_eMREQ-SRS</TemplateEditor> <TemplateVersionMajor>1.2</TemplateVersionMajor> <TemplateVersionMinor>4</TemplateVersionMinor> <TemplateYear>2016</TemplateYear> <TemplateState>1</TemplateState></DocumentFileOSQ>

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  | |  |
|  |  |  |  | |  |
|  |  | Centre de service Valence | | |  |
|  |  | GEMIS | | |  |
|  |  | Formación JPA - GEMIS | | |  |
|  |  | 1.0 Martes 20 Noviembre 2018 | | |  |
|  |  | Statut: Initiation | | |  |
|  |  |  | |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Version** | **Date** | | **Origine de la mise à jour** | **Rédigée par** | **Validée par** |
| 1.0 | 21/11/2018 | | Première version du document | Alba Bermejo Solís  Adrián Colmena Mateos  Emilio Guillem Simón |  |
|  | |
|  |  | |  |  |  |
|  |  | |  |  |  |

Contenidos

[1. Introducción 5](#_Toc530996481)

[1.1. Bases de Datos Relacionales 5](#_Toc530996482)

[1.2. Mapeo Objeto-Relacional 5](#_Toc530996483)

[1.2.1. Impedance mismatch 6](#_Toc530996484)

[1.2.2. Representación de Clases 6](#_Toc530996485)

[1.2.3. Relaciones 7](#_Toc530996486)

[1.2.4. Herencia 10](#_Toc530996487)

[1.3. Soporte para la Persistencia en Java 12](#_Toc530996488)

[1.3.1. Soluciones del ORM previas a JPA 12](#_Toc530996489)

[1.3.2. JDBC 12](#_Toc530996491)

[1.3.3. Enterprise JavaBeans 12](#_Toc530996492)

[1.3.4. Objetos de Datos de Java 13](#_Toc530996493)

[1.3.5. ¿Por qué otro Estándar? 13](#_Toc530996494)

[1.4. Java Persistence API (JPA) 13](#_Toc530996508)

[1.4.1. EJB 3.0 and JPA 1.0 14](#_Toc530996509)

[1.4.2. JPA 2.0 14](#_Toc530996510)

[1.4.3. JPA 2.1 14](#_Toc530996511)

[1.4.4. JPA 2.2 and EJB 3.2 14](#_Toc530996512)

[1.5. Resumen 15](#_Toc530996513)

[1.5.1. Persistencia de los POJOs 15](#_Toc530996514)

[1.5.2. No intrusivo 15](#_Toc530996515)

[1.5.3. Object queries 15](#_Toc530996516)

[1.5.4. Mobile Entities 15](#_Toc530996517)

[1.5.5. Configuración simple 15](#_Toc530996518)

[1.5.6. Integración y Tests. 16](#_Toc530996519)

[1.6. Conclusiones 16](#_Toc530996520)

[2. Introducción a Entidades y Persistencia 17](#_Toc530996521)

[2.1. Descripción general de una entidad 17](#_Toc530996522)

[2.2. Metadatos de las entidades 17](#_Toc530996523)

[2.2.1. Anotaciones 18](#_Toc530996524)

[2.2.2. XML 18](#_Toc530996525)

[2.2.3. Configuración por excepción (Default) 18](#_Toc530996526)

[2.3. Creando una entidad 19](#_Toc530996527)

[2.4. Entity Manager 20](#_Toc530996528)

[2.4.1. Obtener un Entity Manager 22](#_Toc530996529)

[2.4.2. El método *persist*. 22](#_Toc530996530)

[2.4.3. El método *find*. 23](#_Toc530996531)

[2.4.4. El método *remove*. 24](#_Toc530996532)

[2.4.5. Modificando una entidad 24](#_Toc530996533)

[2.4.6. Transacciones 25](#_Toc530996534)

[2.4.7. Queries 25](#_Toc530996535)

[2.5. Recopilación 27](#_Toc530996536)

[2.6. Uniendo todas las piezas. 30](#_Toc530996537)

[2.6.1. Persistence Unit 30](#_Toc530996538)

[2.7. Conclusiones 31](#_Toc530996539)

# Introducción

A pesar del éxito adquirido por la plataforma Java en el desarrollo en sistemas de bases de datos, durante un largo tiempo sufrió el mismo problema que muchos otros lenguajes de programación orientados a objetos. El intercambio de datos entre el sistema de la base de datos y el modelo de una aplicación Java era innecesariamente complicado. Los desarrolladores de Java se encontraban con la necesidad de escribir una gran cantidad de código para convertir las columnas y filas de la base de datos en objetos, o se hallaban sujetos a las especificaciones de los proveedores de frameworks que intentaban abstraer la complejidad de la base de datos. Afortunadamente, se introdujo una solución estándar en la plataforma para eliminar la brecha existente entre los modelos de dominio orientado a objetos y los sistemas de base de datos relacionales: la API de Persistencia de Java (JPA).

Este manual introduce la versión 2.2 de la API de Persistencia de Java dentro del contexto de Java EE 8 y explora todo lo que ofrece a los desarrolladores.

Uno de los puntos fuertes de JPA es la posibilidad de usarlo en cualquier tipo de aplicación (web, escritorio, etc).

Para entender cómo surge JPA, en este primer capítulo se dará un repaso a los problemas previos a su aparición, así como su historia

## Bases de Datos Relacionales

Las bases de datos relacionales continúan siendo las más versátiles y populares y son donde se almacena la gran mayoría de los datos corporativos del mundo. Son el punto de partida de cualquier aplicación empresarial y, a menudo, su vida útil supera a la de la propia aplicación.

Entender las bases de datos relacionales es fundamental para desarrollar aplicaciones empresariales exitosamente y suele ser una de las partes que más quebraderos de cabeza genera. Uno de los principales motivos del éxito de Java puede atribuirse a su utilización generalizada en la creación de sistemas de bases de datos corporativos. Desde páginas web para consumidores hasta puertas automatizadas, las aplicaciones Java son el corazón del desarrollo de aplicaciones de una empresa.

## Mapeo Objeto-Relacional

Puede parecer que el modelo de dominio (objetos) se parece lo suficiente al modelo relacional de la base de datos como para tener una manera sencilla de conseguir relacionarlos pero las diferencias son importantes y la relación no es directa.

La técnica que consiste en relacionar el modelo de objetos y el modelo relacional es conocida como Mapeo Objeto-Relacional, a veces referido como mapeo O-R o simplemente ORM. Se llama mapeo porque consiste en relacionar conceptos de un modelo con conceptos del otro, por ejemplo relacionar objeto con tabla, relación entre dos tablas con atributo de una clase, etc de manera a que el paso de uno a otro sea casi automático.

Tres ideas claves en un ORM son:

* El ORM debe abstraer de los detalles de la base de datos. Tendríamos que seguir programando en Java sin tener que estar pensando en los detalles de la base de datos como tablas, columnas, etc.
* *La* aplicación Java siempre necesita tener control sobre los objetos que persiste y estar al tanto de su ciclo de vida.
* Es mucho más común que una aplicación se adapte a un esquema de base de datos ya existente que crear uno nuevo desde cero. El soporte para esquemas heredados es uno de los casos más relevantes que pueden surgir, y es muy probable que tales bases de datos sobrevivan incluso más que nosotros mismos.

### Impedance mismatch

Los defensores de el mapeo objeto-relacional usualmente describen la diferencia entre el modelo objeto y el modelo relacional como la “impedance mismatch”. ya que el reto de mapear uno con el otro reside no en las similitudes entre ambos, sino en los conceptos de ambos modelos que no tienen un equivalente lógico en el otro.

En las siguientes secciones se presentan algunos ejemplos básicos de modelos de dominio orientado a objetos y una variedad de modelos relacionales para persistir los mismos datos. Como se verá, el reto en el mapeo objeto-relacional reside en la multitud de posibilidades existentes para un mismo caso. *Paragraph to be deleted if there are no entries in the table.*

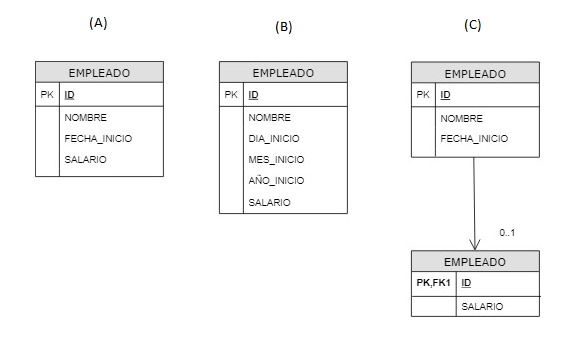
### Representación de Clases

Se empezará la discusión con una clase simple. La Figura 1 muestra la clase Empleado, la cual tiene cuatro atributos: id, nombre, fecha de inicio y salario.



**Figura 1: Esquema de la clase Empleado**

Ahora se tienen tres modelos relacionales distintos en la Figura 2. La representación ideal de esta clase en la base de datos corresponde al caso (A). Cada campo en la clase conecta directamente con una columna de la tabla. El id del empleado es la primary key. Con la excepción de alguna variación en los nombres de los atributos, este caso es un mapeo directo.



**Figura 2: Tres opciones de modelo relacional asociados a la clase Empleado.**

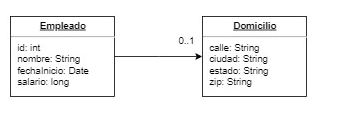
En el modelo (B), la fecha ha sido dividida en tres columnas distintas, día, mes y año, lo cual dificulta mucho el mapeo del atributo fecha al tener un formato distinto.

En este caso el modelo (C) sería el más apropiado para la representación relacional de la clase Empleado, ya que el salario constituye una información sensible que, normalmente, se trata de manera separada al empleado en cuestión. Como se puede observar, el campo salario se almacena en una tabla separada, lo que permite restringir el acceso.

Se ha visto que incluso el mapeo de una única clase puede ser complicado, pero se debe recordar siempre que las necesidades de la base de datos priman ante las de la aplicación.

### Relaciones

Los objetos rara vez existen de manera aislada, normalmente están asociados y dependen de otras clases de dominio. Por ejemplo, a la clase vista en el apartado anterior, Empleado, se le puede asociar una nueva clase, Domicilio, por lo que un mismo Empleado puede tener como mucho una instancia. En este caso se dice que Empleado y Domicilio tienen una relación one-to-one (uno a uno), que se representa en un diagrama UML (Unified Modeling Language) con 0..1. La Figura 3 ilustra esta relación.



**Figura 3: Esquema de la relación entre Empleado y Domicilio**

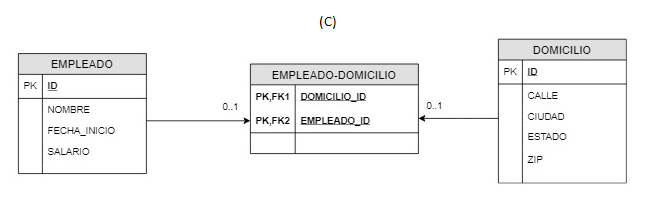
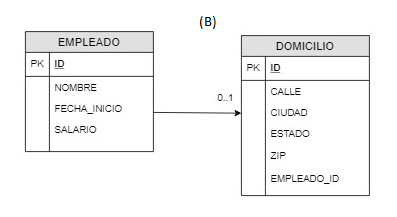
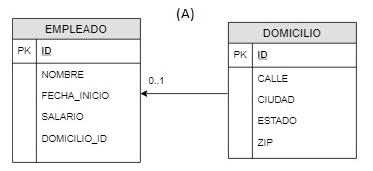
Al igual que en la sección anterior, en este caso también tenemos varias posibilidades para el esquema de base de datos, como se muestra en la Figura 4.

El elemento esencial en torno al cual se construye la relación es la primary key. Para relacionar ambas tablas es necesario que ambas tengan una primary key, por lo que se le añade un id a Domicilio que actuará como primary key y, por tanto, se deberá adaptar el mapeo de alguna manera.

En el esquema (A) se muestra el mapeo ideal para la relación, introduciendo en la tabla Empleado la primary key de Domicilio, que actúa como foreign key, de manera que se puede especificar el id del Domicilio al introducir un registro en la tabla Empleado.

En cuanto al esquema (B), aunque muy similar al (A), es mucho más complejo, ya que en el modelo de dominio una instancia de Domicilio no tiene una referencia a una instancia de Empleado. El mapeo objeto-relacional debe tener en cuenta este desajuste entre la clase de dominio y la tabla o se deberá añadir una referencia al empleado para cada domicilio.

Para complicar más las cosas, el esquema (C) introduce una tabla de unión entre Empleado y Domicilio en la que se almacenan las combinaciones (Empleado/Direccion) de foreign keys de cada tabla. Con este diseño en cada operación que se haga contra la base de datos que involucre a ambas tablas se debe de ir a buscar la información en esa tabla de unión. Se podría modificar el modelo de dominio incluyendo una clase de asociación Empleado-Domicilio pero no es la solución intuitiva para modelar la relación entre la clase Empleado y la clase dirección.

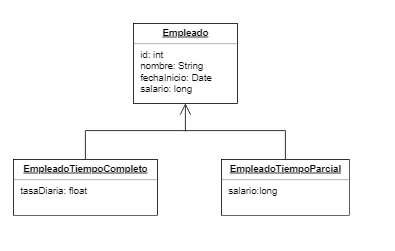


**Figura 4: Tres posibilidades para la representación en base de datos de la relación Empleado-Domicilio**

Las relaciones presentan un desafío en el mapeo objeto-relacional. En esta introducción se ha cubierto solo las relaciones one-to-one y aun así se han podido observar que existen maneras diferentes de diseñar una base de datos para almacenar la información de un modelo de dominio que pueden complicar más o menos el mapeo entre el modelo relacional (tablas, relaciones) y el modelo de dominio (clases).

### Herencia

La herencia es la manera natural de expresar relaciones generales entre clases similares y permite introducir el polimorfismo en la aplicación. Retomando el ejemplo visto anteriormente de la clase Empleado, se puede pensar en una empresa que necesite distinguir entre empleados a tiempo completo y a tiempo parcial, que difieren únicamente en el tipo de salario. Esto supone una buena oportunidad para ilustrar el concepto de herencia. En java, para trabajar con objetos de tipo empleado, se crearía una clase Empleado que almacenaría los atributos comunes y dos subclases (EmpleadoTiempoCompleto y EmpleadoTiempoParcial) que almacenarían los atributos específicos de cada una, es decir el salario o la tasa diaria, como se puede ver en la Figura 5. En estas subclases, además, se definirían las operaciones específicas de esa clase, como podría ser cambiar el salario,



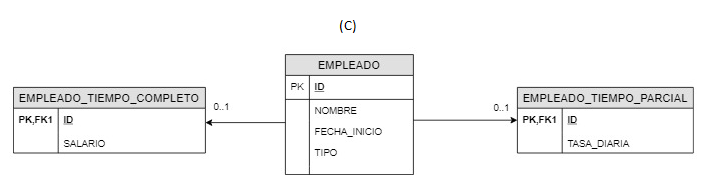
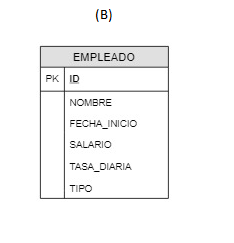
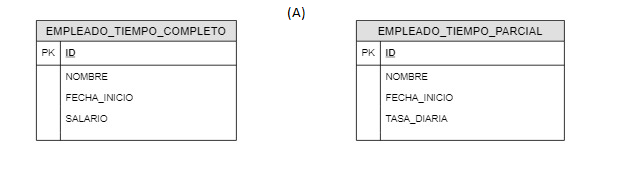
**Figura 5: Relación de herencia entre la clase Empleado y las subclases EmpleadoTiempoCompleto y EmpleadoTiempoParcial**

En la Figura 6 se muestran de nuevo tres posibles esquemas para la persistencia en base de datos.

Podría decirse que la solución más sencilla para el mapeo de una estructura de herencia en base de datos sería poner todos los datos necesarios para cada clase, incluyendo la clase madre Empleado, en tablas separadas, como ilustra el esquema (A). En este caso las tablas no tienen ninguna relación entre ellas, lo que supone que las consultas que impliquen a ambas serán mucho más complejas para el usuario.

Una solución más eficiente, aunque desnormalizada, se ilustra en el esquema (B), en el que todos los datos necesarios para cada clase se sitúan en una única tabla, lo que simplificaría enormemente las consultas. En este caso se introduce un nuevo parámetro “tipo” que indica si el empleado es a tiempo parcial o completo, lo cual implicaría cambiar el modelo de dominio para interpretar la información aportada por cada registro en la tabla.

El esquema (C) lleva esto un paso más allá y normaliza los datos en tablas separadas para cada tipo de empleado. Al contrario que en el esquema (A), estas tablas están relacionadas por una tabla EMP que reúne los datos comunes a ambas. Puede parecer algo excesivo introducir una tabla nueva por una única columna de datos extra, pero este método simplifica las consultas a realizar con ambas tablas y presenta los datos de manera lógica.



**Figura 6: Diferentes esquemas para el mapeo relacional de herencia entre clases**

La herencia se convierte rápidamente en un tema complejo en el mapeo objeto-relacional. No sólo existe un reto con el almacenamiento de los datos de clase, sino que las complejas relaciones de tablas también son difíciles de consultar de forma eficiente.

## Soporte para la Persistencia en Java

Desde el inicio de la plataforma Java, han existido interfaces de programación que permiten comunicar con la base de datos y abstraer detalles de la base de datos El API de persistencia de Java consta de cuatro áreas:

* Java Persistance API
* El Java Persistance Criteria API
* Query language (lenguaje de consulta)
* Object relational mapping metadata

### Soluciones del ORM previas a JPA

Las soluciones ORM han existido durante mucho tiempo, incluso más tiempo que el propio lenguaje Java. Productos como Oracle TopLink se inició en el mundo de Smalltalk antes de hacer el cambio a Java.

Los dos propietarios de APIs de persistencia más populares fueron TopLink en el espacio comercial e Hibernate en la comunidad de código abierto. Productos comerciales como TopLink estuvieron disponibles desde los primeros días de Java, fue más tarde cuando el mapeo de objeto-relacional como Hibernate se hicieron más populares y convirtió el mapeo de objeto relacional como la solución preferida para muchas aplicaciones. JPA está basado en estos productos.

### JDBC

La segunda versión de la plataforma Java, Java Development Kit(JDK)1.1, lanzada en 1997, marcó el inicio del primer gran soporte de persistencia en base de datos con JDBC. JDBC es una versión para Java de ODBC (Object Database Conectivity) que permite acceder a cualquier base de datos relacional desde cualquier idioma o plataforma. JDBC permite a programas de Java una interacción completa con la base de datospero depende en gran medida de SQL. Lo que permite es escribir consultas y declaraciones de manipulación de datos en lenguaje de base de datos, pero ejecutando y procesando un modelo de programación en Java, es decir, es sólo una interfaz para acceder de forma estándar a cualquier base de datos utilizando SQL pero no una solución ORM.

La desventaja, es que aunque SQL es un estándar, existen muchos dialectos para distintas bases de datos y el código desarrollado con JDBC + SQL puede no ser compatible de una base de datos a otra

### Enterprise JavaBeans

La primera versión de la plataforma Java 2 Enterprise Edition (J2EE) introdujo una nueva solución para la persistencia, llamados “beans” de entidad, que son parte de la familia de componentes de Enterprise JavaBean (EJB). EJB introdujo un enfoque basado en una interfaces y en clases generadas por el compilador que no eran utilizadas directamente en el código fuente. Estas clases generadas facilitaban la persistencia, la seguridad y la gestión de transacciones. La lógica de negocio se implementaba en una clase entidad que hereda de la interfaz que se configuraba con una combinación de Anotaciones y XML

EJB era una solución excesivamente compleja técnicamente y a la vez no resolvía problemas habituales por lo que no tuvo mucha aceptación y los ORM comerciales u open source eran utilizados en su lugar.

### Objetos de Datos de Java

Ante la frustración de no tener una API de persistencia estandarizada satisfactoria se inventaron los Java Data Objects(JDO) que fueron inspirados principalmente por los proveedores de base de datos orientadas a objetos (OODB) y nunca fueron adoptados por la comunidad de programación estándar.

JDO se llegó a convertir en una extensión del JDK pero nunca se convirtió en una parte integral de la plataforma Java. Por desgracia, los principales proveedores comerciales no compartían el mismo enfoque sobre cómo implementar la solución por lo que rara vez se llegó a usar.

Una vez que el movimiento de persistencia de EJB 3.0 se puso en marcha, los principales proveedores se inscribieron para formar parte de la nueva iniciativa de persistencia. Fue entonces cuando se anunció que JDO se reduciría a la especificación de modo de mantenimiento y JPA se basaría tanto en JDO como en los proveedores de persistencia y se convertiría en el único estándar del futuro.

### ¿Por qué otro Estándar?

Dicho estándar permitía al desarrollador poder tratar el objeto persistente como cualquier otro objeto Java y luego asignarlo a un almacén persistente y usar una API de persistencia para persistirlo. Debido a que los objetos eran objetos Java normales, este modelo de persistencia llegó a ser conocido como Plain Old Java Object (POJO).

La necesidad de hacer otro estándar, es permitir una implementación completa sin la necesidad de acoplar a la aplicación un producto en concreto.

Vincular un estándar a un proyecto de código abierto como Hibernate sería problemático para el estándar y probablemente aún peor para el proyecto Hibernate. Sin embargo, el uso de una tecnología estándar permite que la empresa pueda cambiar de proveedor si la elección inicial no llega a cumplir con los requisitos necesarios.

## Java Persistence API (JPA)

API de Persistencia Java es un framework ligero y basado en POJO.

Aunque el ORM es uno de los principales componentes de la API, también ofrece soluciones a otros problemas como la integración de la persistencia dentro de aplicaciones escalables. Los siguientes apartados dan una visión más amplia de la mayoría de aspectos de esta tecnología.

JPA no es un producto, sino sólo una especificación que no puede funcionar por sí sola y que requiere una base de datos para persistir.

### EJB 3.0 and JPA 1.0

Después de años de quejas sobre la complejidad para la construcción de aplicaciones con Java EE, «ease of development» fue el lema de salida para la plataforma Java EE 5. EJB 3.0 se encargó y encontró maneras de realizar Enterprise JavaBeans más sencillas y productivas.

Para conseguirlo se partió de cero, olvidando las versiones anteriores de EJB e intentando adoptar el modelo natural de entidades del lenguaje Java que ya usaban proveedores ORM como Hibernate y Toplink.

Los proveedores de soluciones ORM dieron un paso adelante y estandarizaron las mejores prácticas de sus productos para apuntarse al nuevo estándar EJB. Posteriormente se unieron a la iniciativa los proveedores de soluciones JDO.

El uso de anotaciones dio lugar a una nueva manera de usar la persistencia en aplicaciones que nunca antes se había visto.

La especificación resultante EJB 3.0, que vio la luz en 2006, terminó siendo dividida en tres partes. La primera contenía todo el contenido del modelo de componentes heredados de EJB y la segunda describía el nuevo modelo de componentes POJO simplificado. La tercera era JPA, una especificación independiente que describe el modelo de persistencia en los entornos Java SE y Java EE.

La Figura 1-8 muestra JPA en el entorno Java EE.

### JPA 2.0

La siguiente versión, JPA 2.0, nació en 2009 e incluyó una serie de nuevas funcionalidades que no estaban presentes en la primera versión, en concreto las que habían sido las más solicitadas por los usuarios. Esta nueva versión incluía capacidades de mapeo adicionales, formas flexibles de determinar la forma en que el proveedor accedía al estado de la entidad y extensiones a JPQL. Probablemente la nueva funcionalidad más significativa fue el Java Criteria API, una manera de crear consultas dinámicas. Esto permitió principalmente que los frameworks utilizaran JPA como un medio para construir código que accediera a los datos.

### JPA 2.1

Con JPA 2.1 en 2013 se extendió la funcionalidad hasta el punto de prácticamente poder desarrollar aplicaciones sólo con JPA y sin funcionalidades fuera del estándar. Tamibién incluía funcionalidades más exóticas como mapping converters, soporte a procedimientos almacenados o “unsynchronized persistence contexts for improved conversational operations”.

### JPA 2.2 and EJB 3.2

La versión de mantenimiento de JPA 2.2 fue publicada por Oracle en junio de 2017.

## Resumen

JPA es simple y elegante, potente y flexible. Su uso es natural y fácil de aprender. Cualquier API operativa podrá consistir de un pequeño número de clases.

### Persistencia de los POJOs

Quizás lo más importante de JPA es que los objetos son POJOs, lo que significa que no hay nada especial en ningún objeto que se haga persistente. De hecho, casi cualquier objeto de aplicación no final existente, con un constructor predeterminado, puede hacerse persistente sin siquiera cambiar una sola línea de código.

### No intrusivo

JPA existe como una capa separada de los objetos persistentes. Los objetos a persistir (Entity Beans) no necesitan implementar interfaces EJB.

### Object queries

JPA incluye QL (Query Language), un API basado en el lenguaje SQL que permite hacer consultas. De esta manera se pueden hacer consultas en la base de datos sin necesidad de conocer como están definidas físicamente las tablas y relaciones ya que los parámetros utilizados para construir las consultas son las clases y sus atributos Java. Estas consultas pueden devolver entidades (POJOS) pero también nuevos objetos, cálculos, etc

### Mobile Entities

Las aplicaciones web y cliente/servidor son las más populares y estas aplicaciones suelen ser distribuidas, es decir se ejecutan en varias máquinas a la vez. JPA permite que las entidades sean móviles en la red, y que los objetos puedan “moverse” de una máquina a otra.

Cuando un objeto sale de la capa de persistencia se lo denominan *detached*. Una característica clave del modelo de persistencia de JPA es la capacidad de cambiar entidades *detached* y luego volver a unirlas (*attach*) cuando regresen a la máquina de origen.

### Configuración simple

JPA tiene muchas opciones de configuración y todas ellas son configurables a través del uso de anotaciones, XML o una combinación de ambos. Las anotaciones Java (integradas en el código) permiten que el código sea fácil de usar y leer, y hacen posible que los principiantes pongan en marcha una aplicación rápida y fácilmente. Aun así, también se puede configurar JPA con ficheros XML como se ha hecho tradicionalmente en ORMs como Hibernate.

### Integración y Tests.

Normalmente las aplicaciones se ejecutan en Servidores de aplicaciones (contenedores), lo que complica hacer tests unitarios.

JPA está pensado para trabajar con contenedores, ya que es la forma habitual de desarrollo, pero a la vez permite que las aplicaciones desarrolladas con pocos cambios puedan ejecutarse relativamente fácil fuera del contenedor y también desarrollar tests unitarios que se ejecuten sin el contenedor.

## Conclusiones

En este capítulo se presenta una introducción a JPA. Para ello, primero se inicia con una explicación de la necesidad de conectar Java con Bases de datos, y los problemas a los que se enfrentaron los desarrolladores, ilustrado con una serie de ejemplos y posibilidades para cada uno de los casos.

A continuación, se introduce el concepto de ORM, así como el tratamiento de este, y la necesidad de un estándar que fuera a la vez versátil y completo. Las principales soluciones para la persistencia, tales como JDBC, EJB y JDO, son brevemente planteadas para introducir el contexto.

El capítulo se concluye con una breve revisión a la APP que incluye la historia de la misma y los proveedores que se unieron para crearla, introduciéndola en el marco de desarrollo de aplicaciones empresariales y, finalmente, describiendo algunas de las funcionalidades que ofrece la especificación.

# Introducción a Entidades y Persistencia

El principal objetivo de JPA es que fuera fácil de usar y entender y que permitiera a los desarrolladores desarrollar y usar entidades de una forma sencilla e intuitiva.

En este capítulo se verán las características básicas de las entidades y los requisitos que deben tener, definiendo que es una entidad y como operar sobre ella. Además, se presentarán los objetos EntityManager y Query para poder ejecutar consultas a base de datos. Finalmente, se reunirán todos los conceptos y métodos vistos en un ejemplo de aplicación.

## Descripción general de una entidad

En general, una entidad es una representación en Java de una tabla de la base de datos, y tiene una serie de características que la definen.

Se define a una entidad como objetos que tienen atributos y relaciones con otras entidades, así como la posibilidad de persistir dichos atributos y relaciones en una base de datos relacional.

En JPA cualquier objeto definido en la aplicación puede ser una entidad. A continuación, vamos a ver algunas de las características más importantes de una entidad:

1-Las entidades se persisten. Prácticamente cualquier objeto puede persistirse en una base de datos así que puede ser una entidad. Para marcarlo como entidad hay que declararla en la configuración de JPA de la aplicación en una *persistence unit*.

2- Las entidades tienen que tener un identificador de persistencia. Es un atributo o varios de la clase que sirven para identificar el registro o registros de la base de datos con los que se corresponde la entidad.

3-Las entidades tienen un estado. En cualquier momento tenemos que poder saber si ha sido persistida (guardado en base de datos) o no.

4- Las entidades pueden ser tan simples como una clase con un solo atributo o tan complejas como una que tiene miles, pero para que se puedan usar lo ideal es que sean ligeras como un objeto normal de Java.

## Metadatos de las entidades

Además del estado de persistencia de la entidad, es decir si ha sido si los datos han sido guardados o no, cada entidad de la aplicación tiene metadatos asociados que la describen. Estos metadatos permiten que la capa de persistencia reconozca, interprete y administre correctamente la entidad desde el momento en el que se almacena hasta el momento de su uso en tiempo de ejecución.

Los metadatos necesarios para la creación de las entidades suelen ser mínimos y se pueden especificar mediante anotaciones o ficheros XML.

### Anotaciones

Una anotación es una forma de añadir metadatos al código fuente que estén disponibles para la aplicación en tiempo de ejecución. JPA nos proporciona estas anotaciones, las cuales nos permiten definir entidades de una forma sencilla, al igual que nos permiten también definir el comportamiento de las diferentes propiedades de las clases donde se hayan.

Las anotaciones son la forma más simple de configurar JPA y tienen la ventaja de que se evita tener ficheros de configuración y que la entidad y sus metadatos están en el mismo fichero, es decir, en la clase Java, con lo que es muy rápido encontrar y consultar los metadatos de una entidad.

Con el uso de anotaciones, podemos convertir fácilmente una clase normal de Java en una entidad sin la necesidad de utilizar fichero de configuración XML. Todas las anotaciones que se van a ver a lo largo de este manual están definidas en el paquete *javax.persistence.*

La ventaja de usar anotaciones es que pueden ser usadas en cualquier clase, y permiten hacer el código de Java mucho más fácil de leer.

### XML

Para los puristas de XML, las anotaciones pueden ser cambiados por descriptores de XML, ya que estos han sido modelados en su mayor parte a partir de las anotaciones.

### Configuración por excepción (Default)

La configuración por excepción es otra forma de configurar JPA que hace que el motor de persistencia defina valores predeterminados, los que se aplican a la mayoría de las aplicaciones, y que el programador solo proporcione configuración adicional cuando dese sobrescribir algún valor. Estos valores predeterminados permiten que los metadatos sean más relevantes y concisos, ya que hay menos configuración y solo lo que se sale de lo habitual. Sin embargo, cuando una aplicación utiliza configuración por excepción, también puede también complicar el desarrollo ya que la configuración de JPA no es visible. Esto puede hacer que los usuarios no sean conscientes de la complejidad de la configuración de persistencia.

Los valores predeterminados de configuración, permiten que un desarrollador se inicie de manera fácil y rápida con JPA, y luego simplemente deberá añadir nuevas configuraciones/metadatos a medida que aumente la complejidad de su aplicación y haya entidades o relaciones que salgan de lo habitual.

## Creando una entidad

Las clases normales de Java son fácilmente transformables en entidades por medio de anotaciones, ya que cualquier clase con anotaciones y un constructor sin argumentos puede convertirse en una entidad.

Los requisitos mínimos que debe cumplir una entidad son los siguientes:

* Debe tener importado el paquete javax.persistence.Entity.
* Debe tener como mínimo un constructor *public* o *protected* sin argumentos.
* No debe tener ningún método o variables declaradas como final.
* Debe implementar la interfaz *Serializable.*
* Puede heredar tanto de clases que ya son entidades como de clases de no sean entidad,
* Los atributos de las clases que son declaradas como entidad deben ser privados, protegidos o de protegidas del paquete (es decir, solo pueden ser accedidas directamente por los métodos de la clase entidad).

Procedemos a ver la creación de una clase normal de Java: Empleado.class, ilustrado en el Ejemplo1.

***Ejemplo 1.*** Clase Empleado.

***@Entity***

public class Empleado {

//definición de las propiedades

***@Id***

private int id;

***@Column(name="nombre")***

private String nombre;

***@Column(name="salario")***

private long salario;

//definición de los constructores(uno con argumentos y otro sin)

public Empleado() {

}

public Empleado(int id) { this.id = id; }

//definicion de los métodos get y set de las propiedades

public int getId() { return id; }

public void setId(int id) { this.id = id; }

public String getNombre() { return nombre; }

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

public long getSalario() { return salario; }

public void setSalario (long salario) { this.salario = salario; }

}

La anotación *@Id*, que sirve para identificar la propiedad que es la *identidad* (equivalente a la clave primaria en la tabla de la base de datos)*,* se coloca en el campo o la propiedad asociada, lo que quiere decir que depende de las necesidades y gustos del creador de la entidad se puede anotar en la propiedad privada o en el método get de la propiedad.

Los campos de las entidades se persisten automáticamente en la base de datos una vez esté persistida la entidad.

También es posible asignarle un nombre en concreto a la entidad añadiendo a la anotación *@Entity* un campo *name*. Por ejemplo, *@Entity(name "Emp"),* lo cual permite identificar a la entidad en la base de datos por el nombre que se le ha asignado.

En cambio, si no se añade ningún nombre a la entidad, la tabla tomará el nombre por defecto de la entidad. En el ejemplo anterior, si no le ponemos el campo *name* a la anotación *@Entity,* el nombre por defecto de la entidad será "Empleado" y la tabla correspondiente a esta entidad en base de datos saldrá con el nombre "EMPLEADO".

De la misma manera sucede con los campos. Si no se especifica ningún nombre para las diferentes columnas de la tabla, las columnas cogerán el valor por defecto de los campos de la entidad. El nombre de los campos se puede cambiar añadiendo la anotación *@Column(name="nameColumn").* De esta manera, si ya tenemos un esquema en base de datos existente, podemos hacer que coincidan los nombres.

## Entity Manager

Para persistir una entidad en la base de datos con JPA, hacemos una llamada a su API. Esta API es implementada por el Entity Manager (EM) y encapsulada casi por completo en una única interfaz llamada *javax.persistence.EntityManager*. Hasta que un EM no crea, lee o escribe una entidad, esta no es más que un Objeto Java normal.

Cuando un EM es llamado por una entidad, este objeto pasará a estar gestionado por el propio EM. El grupo de entidades gestionadas dentro de un EM en un momento dado se llama el contexto de persistencia. No pueden existir simultáneamente dos instancias de Java con la misma clave primaria en el mismo contexto de persistencia. Por ejemplo, no pueden existir dos empleados que tengan un ID con valor 12. El EM se encarga de hacer este tipo de verificaciones.

Existen distintos EM para trabajar con una base de datos u otra. Cada uno de ellos forma parte de un *persistence provider* diferente.

Es el provider el que suministra el motor de implementación de soporte para todo JPA, desde el EntityManager hasta la implementación de clases query y la generación de SQL.

Todos los EM provienen de factories del tipo javax.persistence.EntityManagerFactory.

El motivo de crear un método Factory, en este caso EntityManagerFactory, es la ventaja de que el programador pueda desarrollar distintos EMs, pudiendo guiarse todos por una misma configuración definida por este EntityManagerFactory que se relacionará con directamente con la persistence unit, la cual establecerá los ajustes y las clases de entidad usadas por los EMs. Esto permite que el código pueda ser usado para cualquier persistence provider, es una capa intermedia que permite ordenar y estandarizar los distintos EMs que el programador cree.

Las persistence units se nombran para diferenciar una EntityManagerFactory de otra. Esto le da a la aplicación control sobre qué configuración o persistencia debe utilizarse para operar en una entidad en particular.

La Figura 1 muestra que por cada persistence unit hay una EntityManagerFactory y que muchos EMs pueden ser creados a partir de una EntityManagerFactory. No obstante, varios EMs pueden apuntar al mismo contexto de persistencia. Sólo hemos hablado de un EM y su contexto de persistencia, pero más adelante se verá que puede haber múltiples referencias a diferentes EMs, todos apuntando al mismo grupo de entidades administradas.

Persistence

EntityManagerFactory

EntityManager

Crea

1

\*

Crea

1

\*

Persistence Unit

Contexto de persistencia

Crea

1

\*

Controla a

1

\*

Configurada por

1

1

**Figura 1: Relación entre conceptos JPA**

|  |  |  |
| --- | --- | --- |
| **Objeto** | **Objeto del API** | **Descripción** |
| *Persistence* | *Persistence* | Clase de *bootstrap* que sirve para obtener un *EntityManagerFactory* |
| *EntityManagerFactory* | *EntityManagerFactory* | Objetos necesarios para obtener los *EntityManagers* |
| *Persistence Unit* | *--* | Configuración con nombre que declara las entidades y como almacenar los datos |
| *EntityManager* | *EntityManager* | API principal que sirve para realizar operaciones y queries en las entidades |
| Contexto de persistencia | -- | Conjunto de todas las instancias de la entidad administradas por un EM específico |

### Obtener un Entity Manager

Los EM siempre se obtienen de un *EntityManagerFactory,* la cual determina la configuración de sus EMs. El método estático *createEntityManagerFactory()* en la clase *Persistence* devuelve el *EntityManagerFactory* para la persitence unitindicada. El próximo ejemplo indica cómo se crearía un *EntityManagerFactory* para la unidad de persistencia llamada ServicioEmpleado:

EntityManagerFactory emf =

Persistence.createEntityManagerFactory("ServicioEmpleado ");

La persistence unit ServicioEmpleado determina que la configuración sea la de ese *EntityManagerFactory*, especificando sus atributos tales como los parámetros de conexión que los EMs usarán para conectarse a la base de datos. Ahora obtener un EM será tarea fácil:

EntityManager em = emf.createEntityManager();

El motivo de crear distintos EM, es el poder controlar qué y cuándo se persiste algo en base de datos, ya que como el texto explica, por muchos cambios que se realicen en el código, en base de datos sólo se reflejarán en la base de datos cuando se realice el *commit*.

### El método *persist*.

Persistir una entidad es la operación de tomar una entidad transitoria, o una que todavía no tiene ninguna representación persistente en la base de datos, y almacenar su estado para que pueda ser recuperada más tarde. Se usa el EM para persistir en una instancia de Empleado, como en este ejemplo:

Empleado emp = new Empleado(158);

em.persist(emp)

La primera línea en este código crea una instancia de Empleado que queremos persistir (solo fijando el ID, no el nombre o salario), la cual es sólo un objeto de Java. La segunda línea usa el EM para persistir la entidad, que si encuentra algún problema lanzará una *PersistenceException.*

El siguiente ejemplo muestra cómo crear un método simple que cree un nuevo empleado y lo persista en la base de datos.

***Ejemplo 2*. Método para crear un empleado.**

public Empleado crearEmpleado(int id, String nombre, long salario) {

Empleado emp = new Empleado(id);

emp.setNombre(nombre);

emp.setSalario(salario);

em.persist(emp);

return emp;

}

Este método utiliza el *EntityManager* em para persistir Empleado. No es necesario preocuparse por un error de ejecución ya que en caso de existir uno se lanzará una *PersistenceException*.

### El método *find*.

Cuando una entidad se encuentre en base de datos, será habitual querer recuperarla de la base de datos. Con esta línea podremos hacerlo:

Empleado emp = em.find(Empleado.class, 158);

Es necesario escribir la clase de la entidad que estemos buscando y el valor de su clave primaria. Una vez encontrado el empleado existirá en en contexto de persistencia actual asociada a ese EM. Al especificarse la clase se evita la necesidad posterior de un cast de objeto (lo que devuelve el *find* por defecto) a la clase de nuestra entidad.

Si para buscar un objeto utilizamos un ID incorrecto (un objeto que haya sido borrado o que nunca haya existido) devolverá null, por lo que antes de utilizar emp debe verificarse si el valor de este es null.

En el Ejemplo 3 se muestra un sencillo ejemplo del método find.

***Ejemplo 3***. Método para encontrar un Empleado

public Empleado encontrarEmpleado(int id) {

return em.find(Empleado.class, id);

}

### El método *remove*.

Si queremos ejecutar lo que sería en SQL un DELETE, en una o más tablas el método *remove* cobra sentido.

Para poder eliminar una entidad, esta tiene que estar en el contexto de persistencia. Es decir, antes de eliminarla, la aplicación debe haber recuperado a esa entidad a través del EM, como puede verse en las siguientes líneas de código:

Empleado emp = em.find(Empleado.class, 158);

em.remove(emp);

Primero, la entidad debe ser encontrada utilizando *find(),* que devolverá una instancia de Empleado, y luego eliminarla utilizando *remove().* La excepción *java.lang.IllegalArgumentException* aparecerá si el valor de *find()* es null. Esto puede solucionarse comprobando que existe el empleado antes de intentar borrarlo, como en el Ejemplo 4.

***Ejemplo 4.* Método para eliminar un empleado.**

public void removeEmpleado(int id) {

Empleado emp = em.find(Empleado.class, id);

if (emp != null) {

em.remove(emp);

}

}

En este ejemplo aseguramos que un empleado con una id existe, antes de intentar borrarlo.

### Modificando una entidad

Hay varias maneras de modificar una entidad, pero a continuación se muestra una de las maneras más simples y utilizadas. Para ello, debemos encontrar una entidad y una vez hallada, cambiar alguno de sus campos. En las próximas líneas se muestra un ejemplo en el que se modifica el salario del empleado 158.

Empleado emp = em.find(Empleado.class, 158);

emp.setSalario(emp.getSalario() + 1000);

En el ejemplo se muestra como modificar el sueldo de un empleado, de manera general, aunque este no se reflejará en la base datos hasta que hagamos *commit*, en la transacción.

***Ejemplo 5.* Método para modificar un Empleado.**

public Empleado subirSueldoEmpleado(int id, long raise) {

Empleado emp = em.find(Empleado.class, id);

if (emp != null) {

emp.setSalary(emp.getSalary() + raise);

}

return emp;

}

**El método devolverá el empleado ya modificado.**

### Transacciones

En los ejemplos anteriores no se han incluido las transacciones a pesar de que son necesarias si lo que se quiere es realizar un cambio persistente en la entidad. A excepción de *find(),* el cual puede ser llamado con o sin transacción debido a que no realiza ningún cambio en la base de datos, todos los demás métodos necesitan del uso de transacciones.

A la hora de realizar una transacción, se utiliza el API de transacción estándar de Java (JTA). La importancia de una transacción en el caso de no existir, en una operación que necesite de ella se lanzará una excepción o simplemente nunca se llegará a persistir el cambio en la base de datos.

En el entorno en el que se está trabajando en éste manual, Java SE, se necesita iniciar y cerrar una transacción con el servicio *javax.persistence.EntityTransaction* a la hora de realizar cualquier modificación de una entidad. Es decir, se debe iniciar la transacción, realizar el cambio pertinente y cerrar la transacción. La transacción se llama mediante el método *getTransaction()* del EM para obtener un *EntityTransaction* y luego llamar a *begin().* Del mismo modo que para cerrar una transacción se llama al método *commit()* sobre el objeto *EntityTransaction*. Un ejemplo ilustrativo de todo esto lo podemos ver en el Ejemplo 6 que se muestra a continuación.

***Ejemplo 6.*** Abriendo y cerrando una *EntityTransaction*

em.getTransaction().begin();

createEmpleado(158, "John Doe", 45000);

em.getTransaction().commit();

### Queries

Una query de JPA se diferencia de utilizar JDBC para enviar una query SQL, en que permite recuperar objetos de tipo entidad en vez de un objeto genérico de resultado (ResultSet) que haya que convertir luego a mano en entidades.

En JPA, una query es similar a una query de base de datos, pero en lugar de utilizar SQL (Structured Query Language) se utiliza un lenguaje llamado JP QL (Java Persistence Query Language).

Para crear una query en JPA tenemso que pasar por el EM y llamar a su método createQuery() pasándole parámetros para definir la query.

Las queries pueden ser o no tipadas, lo que significa que devuelven un tipo entidad conocido, como la entidad Empleado o no.

Hay dos tipos de queries, estáticas o dinámicas. Una query estática se define normalmente mediante anotaciones o con XML metadata, y debe incluir el query criteria y un nombre asignado por el usuario. Este tipo de queries son las llamadas “named queries”, y se buscan por su nombre en el momento de la ejecución del programa.

Por otro lado, una query dinámica se puede crear en el momento de la ejecución proporcionando el “query criteria” JP QL o un objeto de criterio. Pueden ser un poco más difíciles de ejecutar ya que el persistence provider no puede hacer ninguna preparación previa, pero las queries JP QL son muy simples de utilizar y pueden responder ante la lógica del programa o incluso ante la lógica de usuario.

En la versión de JPA 2.2 se introduce un nuevo método en las queries llamado getResultStream(), que devuelve como resultado de la consulta un stream de Java 8. Este nuevo método de crear queries aprovecha la nueva funcionalidad de expresiones lambda de Java 8 y permite explotar más fácilmente el resultado ya que se devuelve en forma de *stream*.

En el siguiente ejemplo se muestra cómo crear una query dinámica sencilla y ejecutarla para obtener todos los empleados de la base de datos.

***Ejemplo 7.*** Ejemplo de query dinámica.

* Ejemplo usando JPA 2.1:

TypedQuery<Empleado> query = em.createQuery("SELECT e FROM Empleado e",

Empleado.class);

List<Empleado> emps = query.getResultList();

* Ejemplo usando JPA 2.2

Stream<Empleado> Empleado = em.createQuery("SELECT a FROM Empleado e",

Empleado.class).getResultStream();

Se crea un objeto TypedQuery<Empleado> mediante la llamada createQuery en el EntityManager y pasando en la cadena JP QL que especifica el query criteria, así como la clase en la que la query debe estar parametrizada. La cadena JP QL se refiere a la entidad Empleado, no a la tabla Empleado de la base de datos, así que esta query selecciona todos los objetos Empleado sin filtrarlos.

Al igual que al usar JPA 2.2, el nuevo método getResultStream() devuelve una secuencia Java 8 del resultado de la query. Así que, en este caso, devolverá el stream del resultado de la query Empleado.

Estos ejemplos muestran lo sencillo que es de crear, ejecutar y procesar queries, pero no muestran lo poderosas que son.

## Recopilación

Una vez vistos todos los métodos anteriores, la idea es juntarlos y combinarlos en una única clase. La clase actúa como una clase de servicio, a la que se llamará ServicioEmpleado, y que permita realizar operaciones con los empleados. El código debería resultar ya bastante familiar con lo visto anteriormente.

El ejemplo 8 muestra la implementación de todos los métodos en la clase mencionada.

***Ejemplo 8.*** Clase de servicio para operar con la entidad Empleado.

import javax.persistence.\*;

import java.util.List;

public class ServicioEmpleado {

protected EntityManager em;

public ServicioEmpleado (EntityManager em) {

this.em = em;

}

public Empleado crearEmpleado (int id, String nombre, long salario) {

Empleado emp = new Empleado(id);

emp.setNombre(nombre);

emp.setSalario(salario);

em.persist(emp);

return emp;

}

public void borrarEmpleado(int id) {

Empleado emp = buscarEmpleado(id);

if (emp != null) {

em.remove(emp);

}

}

public Empleado aumentarSalario (int id, long aumento) {

Empleado emp = em.find(Empleado.class, id);

if (emp != null) {

emp.setSalario(emp.getSalario() + aumento);

}

return emp;

}

public Empleado buscarEmpleado (int id) {

return em.find(Empleado.class, id);

}

public List<Empleado> buscarTodosLosEmpleados () {

TypedQuery<Empleado> query = em.createQuery("SELECT e FROM Empleado e", Empleado.class);

return query.getResultList();

}

}

Esta sencilla clase puede ser utilizada para realizar las funciones básicas tales como crear, leer, modificar y borrar (CRUD) en la entidad Empleado. La clase exige crear un EntityManager para iniciar (método em.begin()) y llevar a cabo (método em.commit()) las transacciones especificadas. De esta manera, al establecer de manera independiente la lógica de transacción de la lógica de operación, el código está menos acoplado y es más reutilizable.

En el Ejemplo 9, se muestra un programa main muy sencillo que utiliza este servicio y lleva a cabo todas las acciones requeridas por el EntityManager y el transaction managment.

***Ejemplo 9.*** Método main que utiliza la clase ServicioEmpleado.

import javax.persistence.\*;

import java.util.List;

public class EmpleadoTest {

public static void main(String[] args) {

EntityManagerFactory emf =

Persistence.createEntityManagerFactory("ServicioEmpleado");

EntityManager em = emf.createEntityManager();

ServicioEmpleado servicio = new ServicioEmpleado(em);

// crear y persistir un empleado

em.getTransaction().begin();

Empleado emp = servicio.crearEmpleado(158, "John Doe", 45000);

em.getTransaction().commit();

System.out.println("Empleado persistido " + emp);

// buscar un Empleado determinado

emp = servicio.buscarEmpleado(158);

System.out.println("Encontrado " + emp);

// buscar todos los empleados

List<Empleado> emps = servicio. buscarTodosLosEmpleados ();

for (Empleado e : emps)

System.out.println("Empleados encontrados: " + e);

// modificar empleado

em.getTransaction().begin();

emp = servicio.aumentarSalario (158, 1000);

em.getTransaction().commit();

System.out.println("Modificado " + emp);

// borrar un empleado

em.getTransaction().begin();

servicio.borrarEmpleado(158);

em.getTransaction().commit();

System.out.println("Empleado 158 borrado");

// cerrar el EM y EMF

em.close();

emf.close();

}

}

Como se puede ver, al final del programa tanto el EntityManager como el EntityManagerFactory son cerrados mediante el comando close(), lo cual es muy importante, ya que nos asegura la correcta realización de las acciones y que todos los cambios han sido registrados.

## Uniendo todas las piezas.

Una vez conocidas todas las piezas clave de JPA, es hora de introducirlas todas para realizar una aplicación con Java SE, que será lo que se llevará a cabo en esta última parte del capítulo.

### Persistence Unit

La configuración que describe la *persistence unit* está definida en un archivo XML llamado *persistence.xml*. El fichero persistence.xml debe colocarse en una localización particular para que pueda ser encontrado por JPA, en la carpeta META-INF. Cada *persistence unit* tiene un nombre propio, de manera que cuando una aplicación quiere especificar la configuración, únicamente necesita hacer referencia al nombre de la *persistence unit* en la que se define dicha configuración. Un archivo *persistence.xml* puede contener una o varias configuraciones de *persistence unit*, pero cada una de ellas es distinta e independiente de las otras y no se encuentran relacionadas entre ellas. Esto podría ser útil en casos particulares, pero en general tendremos una única persistence unit.

Para este caso, únicamente son necesarias tres secciones, llamadas *transaction-type*, *class* y *properties*. El Ejemplo 10 muestra las partes más relevantes del archivo *persistence.xml* para este caso.

***Ejemplo 10.*** Elementos en el archivo *persistence.xml*.

<persistence>

<persistence-unit name="ServicioEmpleado "

transaction-type="RESOURCE\_LOCAL">

<class>examples.model.Empleado</class>

<properties>

<property name="javax.persistence.jdbc.driver"

value="org.apache.derby.jdbc.ClientDriver"/>

<property name="javax.persistence.jdbc.url"

value="jdbc:mysql://localhost:3306/EmpServDB;create=true"/>

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

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

</properties>

</persistence-unit>

</persistence>

El atributo *name* de la *persistence unit* indica su nombre y es la línea que se especifica cuando se crea el *EntityManagerFactory*, en este caso “ServicioEmpleado”. El atributo *transaction-type* sirve para configurar las transacciones y, en este caso, indica que la persistence unit usa el nivel de recursos del *EntityTransaction* en lugar de las transacciones de JTA. El elemento *class* enumera las clases que forman parte de la *persistence unit*. Cuando hay más de una entidad se pueden enumerar múltiples clases en esta lista. En Java EE, cuando usa un contenedor (*container)*, no suele ser necesario detallar las clases ya que el *container* buscará automáticamente las clases anotadas como *@Entity* como parte del proceso de despliegue de la aplicación, pero son necesarias para la ejecución simple en una aplicación de consola en Java SE. En ese ejemplo sólo existe una entidad Empleado.

La última sección consiste en una lista de propiedades, parámetros de configuración, que pueden ser estándar de JPA o específicas del distribuidor. En este caso son parámetros como el nombre de usuario y contraseña de la base de datos, datos que habitualmente se especifican en este archivo para que JPA sepa dónde debe realizar las acciones y la conexión se lleve a cabo con éxito.

## Conclusiones