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>2018-11-21T15:22:35.1106374+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 | | |  |
|  |  | Version 1.0 Martes 20 Noviembre 2018 | | |  |
|  |  | Statut: Initiation | | |  |
|  |  |  | |  |  |

Historique

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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 4](#_Toc530577084)

[1.1. Bases de Datos Relacionales 4](#_Toc530577085)

[1.2. Mapeo Objeto-Relacional 5](#_Toc530577086)

[1.2.1. Impedance mismatch 5](#_Toc530577088)

[1.2.2. Representación de Clases 5](#_Toc530577089)

[1.2.3. Relaciones 7](#_Toc530577090)

[1.2.4. Herencia 8](#_Toc530577091)

[1.3. Soporte para la Persistencia en Java 10](#_Toc530577092)

[1.3.1. Soluciones del ORM previas a JPA 10](#_Toc530577093)

[1.3.2. JDBC 11](#_Toc530577094)

[1.3.3. Enterprise JavaBeans 11](#_Toc530577095)

[1.3.4. Objetos de Datos de Java 11](#_Toc530577096)

[1.3.5. ¿Por qué otro Estándar? 12](#_Toc530577097)

[1.4. Java Persistence API (JPA) 12](#_Toc530577152)

[1.4.1. EJB 3.0 and JPA 1.0 12](#_Toc530577153)

[1.4.2. JPA 2.0 13](#_Toc530577154)

[1.4.3. JPA 2.1 13](#_Toc530577155)

[1.4.4. JPA 2.2 and EJB 3.2 14](#_Toc530577156)

[1.5. Resumen 15](#_Toc530577157)

[1.5.1. Persistencia de los POJOs 15](#_Toc530577158)

[1.5.2. No intrusivo 15](#_Toc530577159)

[1.5.3. Object queries 15](#_Toc530577160)

[1.5.4. Mobile Entities 15](#_Toc530577161)

[1.5.5. Configuración simple 15](#_Toc530577162)

[1.5.6. Integración y Tests. 16](#_Toc530577163)

[1.6. Conclusiones 16](#_Toc530577164)

# 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. La [Figura 1-1](#Figura1) muestra un ejemplo de base de datos relacional.

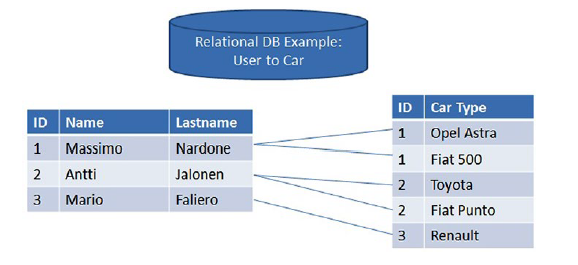


Figura 1-1: Esquema de la base de datos relacional entre usuario y coche.

## 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 1-2 muestra la clase Empleado, la cual tiene cuatro atributos: id, nombre, fecha de inicio y salario.

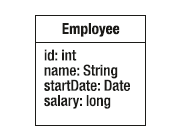


Figura 1-2: Esquema de la clase Empleado

Ahora se tienen tres modelos relacionales distintos en la Figura 1-3. 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.

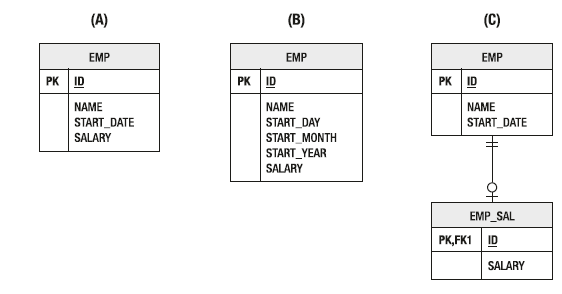


Figura 1-3: Tres opciones de modelo relacional asociados a la clase Empleado.

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 1-4 ilustra esta relación.

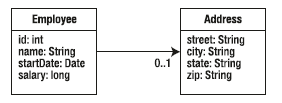


Figura 1-4: Esquema de la relación entre Empleado y Domicilio

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 1-5.

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.

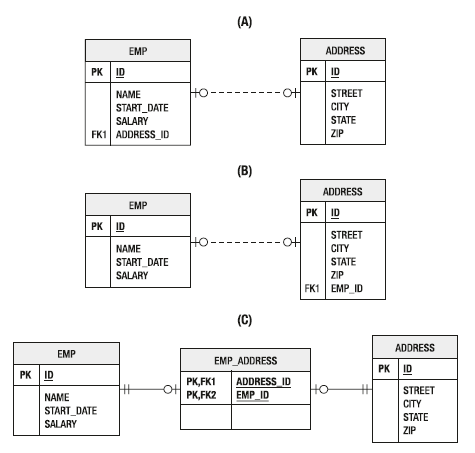


Figura 1-5: Tres esquemas posibles para la representación en base de datos de la relación Empleado-Domicilio.

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 1-6. En estas subclases además, se definirían las operaciones específicas de esa clase, como podría ser cambiar el salario,

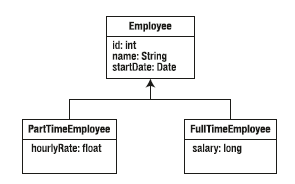


Figura 1-6: Relación de herencia entre la clase Empleado y las subclases EmpleadoTiempoCompleto y EmpleadoTiempoParcial.

. En la Figura 1-7 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.

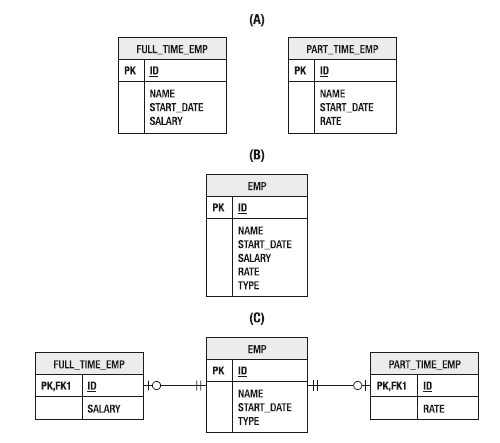


Figura 1-7: Diferentes esquemas para el mapeo relacional de herencia entre clases.

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 o 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?

Ya existía una manera de persistir entidades pero debían adaptarse a la manera de trabajar de cada uno de los diferentes proveedores, tanto comerciales como de código abierto. Los desarrolladores no tenían que preocuparse que tecnología persistía sus entidades.

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).

No sólo Hibernate y TopLink encontraron soluciones para la persistencia en Java. Pero seguía surgiendo una falta de estandarización y una necesidad constante de actualización del EJB. La solución era clara, ofrecer un estándar para el ORM de Hibernate. Pero ni siquiera un producto con tanto éxito como Hibernate podía adaptarse sólo a una especificación aunque pudiera implementarla. El objetivo debía ser que todos los proveedores pudieran utilizar el ORM para sus diferentes interfaces estándar y sus lenguajes.

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.



**Figura 1-8: Esquema de JPA en 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. Aún 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