ORM y JPA

Autores: Simon Pickin
Florina Almenárez Mendoza
Natividad Martínez Madrid
Pablo Basanta Val

Dirección: Departamento de Ingeniería Telemática
Universidad Carlos III de Madrid
España

Versión: 1.0

Agradecimientos: Bill Burke, JBoss Group, Tal Cohen, IBM Haifa

Software de
Comunicaciones

© The Authors

1. Persistencia en Java:
Fallo en la adaptación objeto-relación y mapeo objeto-relación (ORM)

Software de Comunicaciones

© The Authors

La capa de persistencia

- JDBC no es adaptación suficiente de un sistema DBMS
  - Supone uso de RDBMS y SQL
  - Incluso entre dos RDBMS diferentes, el SQL puede cambiar
  - El desarrollador Java puede no conocer (o no querer conocer) SQL
  - Demasiado bajo nivel

- Sobre el uso de una capa de persistencia
  - Abstracción orientada a objeto de la base de datos
  - Parte de una capa de negocio
  - Entre la lógica de negocio y la capa de datos
  - ¿Qué forma para una capa de persistencia?

Software de Comunicaciones

© The Authors

El patrón Data Access Object (DAO)

- Patrón *Data Access Object* (DAO)
  - Objeto que encapsula el acceso a la base de datos
    - Separa la interfaz cliente de los mecanismos de acceso
    - Provee un API genérica
  - Mencionado en los “Core J2EE Patterns” de Sun
  - Similar al patrón Fowler’s *Table Data Gateway*
  - Utilizado a menudo con el patrón DTO

- Patrón *Data Transfer Object* (DTO)
  - Objeto que encapsula datos de la base de datos
  - similar al patrón *Value Object* de Sun (pero no al de Fowler’s)

© The Authors

Consideración sobre la capa DAO

- Capa DAO de una aplicación
  - Usualmente comprende muchos DAOs
  - Algunos de los DAOs pueden encapsular
    - Uniones de SQL
    - Actualizaciones sobre múltiples tablas
- Solución de la capa DAO
  - Demasiado pesada con algunos generadores de código para DAO
    - A no ser que la aplicación sea muy sencilla
  - Metadatos usados para generación de código podrán ser obtenidos de
    - Un descriptor definido por el desarrollador
    - Un *esquema* de la base de datos
    - ambos

© The Authors

Estrategias de implementación para DAO (1/3)

- Codificar cada clase DAO explícitamente
  - Más sencillo pero menos flexible
  - Cambios en la DBMS a menudo implican cambios de la implementación del DAO
- Usar una factoría DAO
  - Patrón del método de factoría (Gamma et al.)
  - Factorías *de clase única*
    - Crean instancias de una clase DAO simple
  - Factorías *multi-clase*
    - Crean instancias de múltiples clases DAO

© The Authors

Estrategias de implementación para DAO (2/3)

- Usar una factoría abstracta DAO
  - Patrón *factoría abstracta* (Gamma et al.)
  - Encapsula un conjunto de factorías DAO
    - Uno para cada DBMS o tipo de DBMS
  - Retorna un puntero a una factoría DAO
    - Cada factoría implementa la misma interfaz abstracta
  - Un cliente crea DAOs con la misma interfaz abstracta
    - Con cualesquiera DBMS o tipos de DBMS

© The Authors

Estrategias de implementación para DAO (3/3)

- Uso de clases genéricas para DAO
  - Manejan diferentes DBMS del mismo tipo
  - Los detalles dependientes de la fuente de datos son cargados a partir del fichero de configuración
    - Por ej. RDBMS: todo el SQL dependiente en el fichero de configuración
- Uso de una factoría DAO abstracta y clases DAO genéricas
  - Retorna un puntero a un factoría DAO
  - Un cliente utiliza una interfaz abstracta
    - Cubre diferentes tipos de DBMS
  - ... para crear DAOs genéricos
    - Cubre diferentes DBMSs de un tipo dado

© The Authors

El patrón *Active Record*

- Patrón *Active Record*
  - Puede ser visto como un patrón que combina DAO y DTO
  - Datos y comportamiento
  - “Un objeto que envuelve una columna de una tabla en una base de datos o vista, encapsula el acceso a la base de datos y añade lógica de domino sobre ese dato.” Fowler

Software de Comunicaciones

© The Authors

Active Record de *Ruby on Rails*

- Modelo del armazón MVC Ruby-on-Rails
- Patrón *Active Record* de Fowler
  - + herencia + asociaciones de objetos
    - El primero vía el patrón *Single Table Inheritance*
    - El último vía un conjunto de macros
- Principios CoC y DRY
- Impone
  - Algunos aspectos en el esquema de la base de datos
  - Algunas convenciones sobre el nombramiento
- Siguiendo las restricciones/convenciones RoR
  - Desarrollo muy rápido
- No siguiendo restricciones/convenciones RoR
  - Descarrila (*derails*) el desarrollo rápido

© The Authors

Desde el Active Record al ORM

- ORM puede ser visto como un *active record*, o similar, con generación de código máxima y soporte añadido para muchos de los siguientes mecanismos:
  - Transacciones
  - *Políticas caché*
  - Integridad relacional
  - Persistencia explicita
  - Mapeo de transacciones
    - Asociaciones a objeto (1:1, 1:N, N:1, M:N) ↔ restricciones de clave foránea
  - Mapeo flexible entre objetos y columnas de tabla:
    - Un objeto simple representa una fila
    - Múltiples objetos representan una fila
    - Objetos sencillo representando múltiples columnas (uniones de SQL)
  - Herencia de relaciones entre objetos persistentes
  - Carga perezosa

© The Authors

Fallo en la adaptación de impedancia de la relación OR. Visión general (1/2)

- Situación inicial
  - Lenguajes de programación, lenguajes de diseño:
    - usualmente OO
  - Sistemas gestores de bases de datos (DBMS)
    - Usualmente relacionales

  =>muchos, muchos desarrollos/aplicaciones usan ambos
- Base conceptual
  - Modelo de objeto:
    - Identificar, encapsulación del estado + comportamiento,
    - Herencia, polimorfismo
  - Modelo relacional
    - Relación, atributo, tupla,
    - valor relacional, variable relacional.

  => mapeo no-trivial: objeto relación $\leftrightarrow$ modelo relacional

Software de Comunicaciones

© The Authors

Fallo en la adaptación de impedancia de la relación OR. Visión general (2/2)

- Posibles soluciones
  1. Evitar el problema
     - Utilización de bases de datos XML
     - Uso de lenguajes OO + DBMS OO
     - Uso de lenguajes no OO + RDBMS
     - Uso de lenguajes OO que incorporan conceptos relacionales + RDBMS
  2. Mapeo manual
     - Codificar manualmente SQL usando herramientas relacionales por ej. JDBC, ADO.NET
  3. Utilizar una capa DAO
     - Aceptar las limitaciones mientras funcione para reducirlas
     - ¡Saber cuando parar de reducirlas!
  4. Usar un armazón ORM
     - Aceptar limitaciones mientras funcione para reducirlas
     - ¡Saber cuando parar de reducirlas!
  5. Mezclar y encajar
     - Por ejemplo, capa DAO parte de la cual usa ORM

© The Authors

Fallo en la adaptación impedancia de la relación OR. Detalles (1/9)

- Cuestiones de gestión
  - ¿Hasta qué punto debería el tipo de aplicación o la aplicación escogida constreñir el esquema de la base de datos?
    - El modelo relacional puede ser usado por múltiples aplicaciones
      - También hay que tener cuidado con el *caching* de la base de datos.
  - Se tienen que manejar dos modelos separados pero altamente acoplados
    - Equipos diferentes pueden ser responsables de cada modelo
    - Problemas con la evolución / refactorización
    - ¿Qué modelo va a ser considerado el principal?

Software de Comunicaciones

© The Authors

Fallo en la adaptación impedancia de la relación OR. Detalles (2/9)

- Asociaciones (también llamadas relaciones)
  - Asociaciones que pueden tener las siguientes multiplicidades
    - Nivel objeto: 1-1,1-N, N-1 y M-N; unidireccional y bidireccional
    - Nivel relacional : 1-1 y asociaciones N-1 unidireccionales sólo (claves foráneas no pueden apuntar a colecciones)
  - El modelo relacional no puede modelar
    - Asociaciones 1-N unidireccionales
    - Asociaciones N-1 bidireccionales
  - Modelo relacional estático incompleto
    - Asociaciones a nivel objeto de facto se mapean a claves + uniones
    - Para generar un modelo de dominio desde el modelo relacional
      - Se debe de añadir información
  - Asociaciones M-N se mapean a:
    - Una relación para cada clase (R1, R2) + una relación de unión Rx
    - Una asociación N-1 desde Rx a R1
    - Una asociación M-1 desde Rx a R2
  - Navegación de asociaciones M-N entre objetos persistentes
    - Implicaciones en el rendimiento

Software de Comunicaciones

© The Authors

Fallo en la adaptación impedancia de la relación OR. Detalles (3/9)

- Herencia
  1. Tabla-por-clase (conteniendo sólo campos específicos de esa clase)
     - Patrón *Class Table Inheritance* de Fowler.
     - Problemas de eficiencia: por ejemplo preguntar el valor de un atributo de una clase implica una unión sobre todas las relaciones que están mapeadas a clases derivadas.
  2. Tabla-por-clase-concreta (usualmente lo mismo que tabla por clase hoja)
     - Patrón *Concrete Table Inheritance* de Fowler.
     - Problemas de integridad, por ej. Unicidad de identificador definido en una clase abstracta no está forzado sencillamente a lo largo del conjunto de relaciones que están mapeadas a clases derivadas
     - No normalizado
  3. Tabla-por-jerarquía-de-herencia
     - Patrón *Single Table Inheritance* de Fowler
     - Poco probable que esté normalizado
     - A menudo necesita un campo discriminador para identificar la clase mapeada
     - Tabla enrarecida: muchos campos pueden ser nulos
     - Los problemas de eficiencia se reducen si la mayoría de los campos heredan de la clase base
  4. Entre las opciones 1 y 3: tabla-por-clase-familia

Software de Comunicaciones

© The Authors

Fallo en la adaptación impedancia de la relación OR. Detalles (4/9)

- Polimorfismo
  - Inclusión de polimorfismo en lenguajes OO:
    - Mecanismo que permite diferentes tipos para exponer diferente comportamiento a través de la misma interfaz
    - Uso de herencia
    - Uso más común: para sobrescribir un método de superclase en una subclase (enlazado tardío)
  - Contexto ORM
    - A veces puede emular la sobrescritura de método
      - Dependiendo del mapeo de herencia escogido

Software de Comunicaciones

© The Authors

Fallo en la adaptación impedancia de la relación OR. Detalles (5/9)

- Entidad
  - Modelo relacional
    - Dos entidades son idénticas si sus contenidos lo son
  - Modelo OO
    - Dos entidades son idénticas si su referencia lo es
      - Diferencia entre "==" y ".equals" en Java
  - La mayoría de RDMBS permite duplicar tuplas
    - A menudo considerado una práctica dudosa desde el punto de vista teórico
    - Las tuplas son indistinguibles en cualquier caso
  - Una misma entidad es consultada dos veces, cargada dentro de diferentes instancias de objeto
    - Puede entrañar problemas
  - ¿Solución?: Importar el mundo OO dentro del relacional
    - Añadir claves generadas por la base de datos y crear nuevos valores para cada instancia de objeto
    - Puede ser poco práctico
  - Interacción con acceso concurrente, *caching*, *clustering* ….

Software de Comunicaciones

© The Authors

Fallo en la adaptación impedancia de la relación OR. Detalles (6/9)

- Composición
  - Comprensión normal:
    - Borrar el compuesto → todos los componentes borrados
  - Modelo OO: No hay problema (recolección de basura)
  - Modelo relacional: Debería de ser explícito
    - La relación compuesta es 1-N: problemático
  - Posibilidades en el modelo relacional
    - Uso de disparadores para la base de datos
    - Mantener la coherencia a nivel de aplicación

© The Authors

Fallo en la adaptación impedancia de la relación OR. Detalles (7/9)

- Contención
  - Contenedores
    - Modelo OO: colecciones, listas, conjuntos
    - Modelo relacional: sólo relaciones
  - Dificultad a la hora de modelar contenedores OO en el modelo relacional
  - Ejemplo: listas con duplicados, soluciones:
    - Muchos DBMS permiten duplicados, incluso si son dudosos teóricamente, aunque no normalizados
    - Repartir sobre dos o más tablas para normalizar

Software de Comunicaciones

© The Authors

Fallo en la adaptación impedancia de la relación OR. Detalles (8/9)

- Encapsulación
  - Encapsulación OO
    - Modificadores de acceso: privado, protegido, público
    - Interfaces
  - Único equivalente relacional: vistas
  - Algunos problemas:
    - Uso de modificadores de acceso:
      - Pueden hacer difícil acceder a valores para hacer que persistan
      - Consecuencia: las consideraciones de acceso pueden influir el dominio del modelo
    - Uso de encapsulación OO: para controlar el acceso a datos
      - Pero con bases de datos compartidas entre aplicaciones

© The Authors

Fallo en la adaptación impedancia de la relación OR. Detalles (9/9)

- Mecanismo de recuperación de datos
  - Búsqueda-por-ejemplo, búsqueda-por-API
  - Búsqueda-por-lenguaje
    - (Uso de lenguajes de búsqueda de objetos)

- Navegación y carga de objetos
  - Mundo OO
    - El coste de la navegación es muy bajo
  - Mundo relacional
    - El coste de navegación (uniones, peticiones múltiples) es alto
    - Particularmente a lo largo de una red
  - Estrategias para reducir costes:
    - Carga perezosa / bajo-demanda de objetos enlazados a objeto recuperado
    - Carga agresiva de objetos enlazados a objeto recuperado
    - Carga perezosa / bajo-demanda de valores de atributo
      - Permite la optimización de búsquedas a nivel de comunicaciones con la base de datos

Software de Comunicaciones

© The Authors

Aproximaciones a ORM (1/3)

- Arriba-Abajo
  - Modelo relacional optimizado para el modelo de dominio
  - Puede involucrar la generación de un esquema de base de datos para un armazón ORM
  - Esquema orientado a una aplicación simple
    - No utilizable fácilmente por otras aplicaciones
    - Integridad usualmente forzada a nivel de aplicación
  - A menudo usa claves primarias generadas por la base de datos para todas las relaciones
    - Aunque más probablemente rompe la normalización
  - Común para prototipos / prueba de concepto

© The Authors

Aproximaciones a ORM (2/3)

- Abajo-Arriba
  - Modelo de dominio optimizado para el esquema relacional
  - Puede involucrar generación de código de aplicación desde el esquemático de la base de datos
  - A menudo usa mapeo 1-1 de tablas a clases
    - ¿Qué pasa con la herencia y la encapsulación?
  - Los objetos pueden perder sus asociaciones
    - Navegación vía el modelo relacional (ineficiente)
  - El modelo del domino producido no es muy entendible para no-desarrolladores
  - El modelo relacional no se desenvuelve sencillamente
    - La generación de código no puede generar una aplicación completa
  - ¿El problema de la adaptación de impedancia simplemente se mueve hacia arriba, hacia el UI?

© The Authors

Aproximaciones a ORM (3/3)

- Encontrarse en medio
  - Modelo de dominio optimizado para comunicación con no-desarrolladores
  - Esquema relacional optimizado para fiabilidad y eficiencia
  - Aproximación más común
    - Parcialmente causada por la larga vida de los esquemáticos de la base de datos

Software de Comunicaciones

© The Authors

Tecnologías de persistencia para Java (1/7)

- JDBC: 1997 (v1), 1998 (v2)
  - Basada en Microsoft's ODBC
    - Basada, a su vez, en X/Open's SQL CLI
  - Base: objetos para
    - Conexión
    - Ordenes
    - Conjuntos de resultados

- Toplink: 1995 (Smalltalk), 1997 (Java)
  - Desarrollado por una compañía creada en la universidad de Carleton
  - Contenía la mayor parte de las características de un ORM
  - TopLink Essentials: implementación reciente de JPA
  - (nótese que el JDK1 se liberó en 1996)

© The Authors

Tecnologías de persistencia para Java (2/7)

- EJB: 1998 (v1), 2001 (v2)
  - Especificación para componentes empresariales
  - Incluye beans de entidad para persistencia
    - Persistencia manejada por el bean (BMP)
    - Persistencia manejada por el contenedor (CMP)
- Diferencias entre los beans de entidad en EJB1 y EJB2
  - Mejor soporte para relaciones (1:1,1:N, M:N)
  - Avances en CMP
  - Interfaces locales
    - Para beans de entidad: facilita el patrón *Session Facade*

© The Authors

Tecnologías de persistencia para Java (3/7)

- Bean de entidad de tipo CMP
  - Mapeo objeto-relación (ORM)
  - No *usan Plain Old Java Objects* (POJOs); los EJBs deben
    - Implementar interfaces especiales
    - Clases extendidas especiales
  - Pesado y verboso
  - Clases persistentes automáticamente
  - Los objetos persistentes no pueden ejecutar fuera del contenedor
    - Dificultad al usar pruebas unitarias
  - El mapeo objeto-relación no está especificado completamente
    - Los detalles se dejan al implementador del contenedor
  - Define un lenguaje de búsqueda de objetos (EJBQL)

© The Authors

Tecnologías de persistencia para Java (4/7)

- Hibernate: 2002 (v1.0)
  - Producto de código abierto
  - Mapeo de objeto relacional
    - Algunas veces utilizado junto a una capa DAO o DAOs genéricos
  - Ligero
  - Utiliza POJOs
  - Los objetos persistentes pueden ejecutar fuera del contenedor de EJBs
  - Mapeo objeto-relación completamente especificado
  - Para hacer clases persistentes
    - Invoca explícitamente métodos sobre un gestor de persistencia
  - Define un lenguaje de consulta de objetos (HQL)
  - Utiliza JDBC
  - Anotaciones Hibernate + Manager Hibernate
    - Provee una implementación JPA encima de Hibernate Core

Software de Comunicaciones

© The Authors

Tecnologías de persistencia para Java (5/7)

- iBatis: 2003 (v1.0)
  - Producto de código libre (ahora Apache)
  - Ligero
  - Básicamente, una capa DAO
    - Con soporte para transacciones y herencia
  - Todo SQL externalizado a archivos XML (SQLMaps)
  - Los objetos almacenados e invocados son Java beans
  - Los POJOs pueden estar mapeados a parámetros de entrada de una orden SQL o a resultados de ejecutar una petición SQL
  - Usa JDBC

© The Authors

Tecnologías de persistencia para Java (6/7)

- Data Objects de Java: 2002 (v1.0)
  - Especificación para objetos persistentes
  - Inspiración de DBMS OO, en particular de la especificación ODMG
  - Mapeo objeto general-fuente de datos, no sólo ORM
  - Soporte a transacciones, *caching*, asociaciones, herencia
  - Utiliza POJOs
  - Para hacer que las clases persistan
    - Explícitamente invoca al método sobre el gestor de persistencia
  - Define un lenguaje de consultas (JDOQL)
  - Usualmente implementado a través de JDBC

Software de Comunicaciones

© The Authors

Tecnologías de persistencia para Java (7/7)

- JPA (2006)
  - Parte de la especificación EJB3 pero puede ser utilizada independientemente
  - Inspiración en Hibernate, TopLink, JDO,…
  - Especificación para el mapeo flexible de objeto-relación
  - Ligero
  - Usa POJOs
  - Los objetos persistentes pueden ejecutar fuera del contenedor de EJBs
  - Mapeo Objeto-Relación completamente especificado
  - Para hacer que las clases persistan
    - Se invoca explícitamente un gestor de entidad
  - Define un lenguaje de consulta de objetos (JPQL)
  - Alternativa al desarrollo de descriptores de despliegue: anotaciones Java

© The Authors

Bibliografía para la Sección 1

- *Persistence in the Enterprise: A Guide to Persistence Technologies*. Roland Barcia, Geoffrey Hambrick, Kyle Brown, Robert Peterson and Kulvir Singh Bhogal. IBM Press, 2008
  `http://my.safaribooksonline.com/9780768680591`
- *Patterns of Enterprise Application Architectures*. Martin Fowler. Addison-Wesley 2002. Summary available at:
  `http://martinfowler.com/eaaCatalog/`
- *J2EE Patterns*
  `http://java.sun.com/blueprints/patterns/`
- *The Object-Relational Impedance Mismatch*. Scott Ambler.
  `http://www.agiledata.org/essays/mappingObjects.html`
- *The Vietnam of Computer Science*. Ted Neward
  `http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx`
- *Object-Relational Mapping as a Persistence Strategy*. Douglas Minnaar.
  `http://www.eggheadcafe.com/tutorials/aspnet/e957c6de-8400-4748-b42d-027f7a228063/objectrelational-mapping.aspx`
- *Object Relational Mapping Strategies*. ObjectMatter.
  `http://www.objectmatter.com/vbsf/docs/maptool/ormapping.html`

Software de Comunicaciones

© The Authors

2. Persistencia Java: el API de Persistencia (JPA)

© The Authors

Entidades

- Entidad
  - Agrupación de estado tratado como unidad única
  - Puede ser persistida
  - Normalmente sólo creada, destruida y actualizada con una transacción
- Entidad JPA
  - POJO de grano fino con las siguientes restricciones
    - Público o protegido, constructor sin argumento
    - Los campos persistentes son privados, protegidos, o paquete privado
      - Los clientes acceden al estado vía métodos de acceso o métodos de negocio
    - Ni la clase ni sus métodos ni sus campos son `final`
  - Normalmente almacenados en un único lugar
  - Se puede asociar/desasociar a la capa de persistencia
    - Asociación: tiene una entidad persistente, única en el contexto de persistencia
    - Des-asociación: puede ser pasado por valor a otra JVM (si es serializable)
    - Por lo que también sirven como objetos de transferencia de datos

Software de Comunicaciones

© The Authors

Metadatos de entidad

- Las entidades JPA tienen asociados metadatos
  - Posibilitando una capa de persistencia para reconocer y manejar entidades
  - Especificada para bien usar anotaciones Java o un XML
    - Las anotaciones son más sencillas
  - “los parámetros de anotación” dan más información sobre las anotaciones
    - Los parámetros de anotación se llaman *elements*
    - Las expresiones del tipo `element=value` se llaman *element specifications*

- Metadatos por defecto
  - JPA define valores por defecto para muchas anotaciones
    - Particularmente *element specifications*
    - “configuración por excepción”
  - Facilidad-de-uso (especialmente el primero)
    - c.f. Principio CoC de Ruby on Rails

- Contenedor Java EE
  - Contenedor web + contenedor de EJBs + proveedor de persistencia

© The Authors

Gestionando entidades

- Las entidades son manejadas por un manejador de entidades
  - Cada gestor de entidades está asociado con un contexto persistente
  - Contexto persistente:
    - Conjunto de instancias manejadas por la entidad que existe en un lugar particular de almacenamiento
- API `EntityManager`
  - Crea y elimina entidades persistentes
  - Encuentra entidades a través de la clave primaria de la entidad
  - Permite que las peticiones puedan correr sobre entidades
    - Utiliza Java Persistence Query Language (JPQL)
  - Desasocia entidades o las reasocia a almacenamiento persistente
- Los gestores de entidad pueden ser
  - Gestionados por el contenedor
  - Gestionados por la aplicación

difieren en comportamiento transaccional y ciclo de gestión de vida

Software de Comunicaciones

© The Authors

Unidades de persistencia

- Una unidad de persistencia
  - Representa el dato contenido en un almacén de datos simple
  - Tiene asociado un **`EntityManagerFactory`**
    - Crea instancias del gestor de entidades con una configuración dada
- Una unidad de persistencia comprende
  - Una colección de entidades que son manejadas de forma conjunta
    - Esto es, un conjunto de clases de entidad
  - La información correspondiente a la configuración
    - Por ejemplo, proveedor de persistencia y código fuente
- Fichero **`persistence.xml`**:
  - Contiene la definición de una o más unidades persistentes
  - JPA dentro de un contenedor Java EE
  - Muchas más cosas para definir en el archivo **`persistence.xml`**

Software de Comunicaciones

© The Authors

Definiendo una entidad JPA

- Una entidad es una clase que:
  - Satisface las restricciones mencionadas anteriormente
  - Se anota con **`@Entity`**
  - Usa un mecanismo para denotar la clase primaria:
    - Un atributo anotado con **`@Id`**
    - Un método get (**`getXxx`**) anotado con **`@Id`**
  - Si la clave primaria está autogenerada
    - Se anota también con **`@GeneratedValue`** + elemento de la spec.
  - Si la clave es compuesta
    - Se anota con **`@EmbeddedId`** o **`@IdClass`** en vez de **`@Id`**

- Elementos por defecto definidos para la anotación **`@Entity`**:
  - Entidad mapeada a la relación del mismo nombre
  - Todos los atributos de entidad persistentes
  - Cada atributo de entidad mapeado a un atributo de relación del mismo nombre

  pueden ser sobrescritos (con especificaciones de elemento)

Software de Comunicaciones

© The Authors

La interfaz `Query`

- El método **`find()`** se usa para buscar una entidad simple por clave primaria
  - Búsquedas más complicadas requieren la interfaz **`Query`**
  - La misma interfaz usada para actualizaciones múltiples y borrados (nuevo en EJB3)

- Para obtener un objeto con la interfaz **`Query`**:
  - Dos métodos del gestor de entidad para peticiones dinámicas

```
public Query createQuery(String jpqlString)
public Query createNativeQuery(String sqlString)
```

  - Un método del gestor de entidad (3 variantes) para peticiones estáticas

```
public Query createNamedQuery(String sqlOrJpqlString)
```

- Métodos principales en la interfaz **`Query`**

```
public List getResultList()
public Object getSingleResult()
public int executeUpdate() //actualización de golpe o borrado
public Query setMaxResults(int maxResult)
public Query setFirstResult(int startPosition)
public Query setParameter(String name Object value)
Public Query setFlushMode(FlushModetype flushMode)
```

Software de Comunicaciones

© The Authors

El lenguaje de peticiones de Java

- Similar a SQL:
  - **SELECT FROM WHERE**, **GROUP BY**, **HAVING**, **ORDER BY**
  - **UPDATE SET WHERE**
  - **DELETE FROM WHERE**

  e incluso más a HQL (Hibernate Query Language)
- Se permiten sub-peticiones en cláusulas **WHERE** y **HAVING**
- Funciones de conjunto para ser usadas en peticiones de agrupamiento y agregación
  - **AVG**, **COUNT**, **MAX**, **MIN**, **SUM**
- Navegación a través de grafos de entidad
  - Usando expresiones de camino (notación .)
    - Permitido en clausulas **SELECT**, **SET**, **FROM**, **WHERE**
  - Aunque ilegal en expresiones de camino para navegar más allá de
    - Atributos de relación *collection-valued*
    - Atributos persistentes

Software de Comunicaciones

© The Authors

Ejemplo 1: CRUD simple (1/3)

```
// fuente: Pro EJB3, Java Persistence API. M. Keith, M. Schincariol
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;

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

    public int getId() {return id;}
    public void setId(int id) {this.id = id;}
    public String getName() {return name;}
    public void setName(String name) {this.name = name;}
    public long getSalary() {return salary;}
    public void setSalary(long salary) {this.salary = salary;}
}
```

© The Authors

Ejemplo 1: CRUD simple (2/3)

```
import javax.persistence.*;
import java.util.Collection;

public class EmployeeService {
    protected EntityManager em;

    public EmployeeService(EntityManager em) {
        this.em = em;
    }

    public Employee createEmployee(int id, String name, long salary) {
        Employee emp = new Employee(id);
        emp.setName(name);
        emp.setSalary(salary);
        em.persist(emp);
        return emp;
    }

    public Employee findEmployee(int id) {
        return em.find(Employee.class, id);
    }
```

Software de Comunicaciones

© The Authors

Ejemplo 1: CRUD simple (3/3)

```
    public RemoveEmployee(int id) {
        Employee.emp = findEmployee(id);
        if (emp != null){
            em.remove(emp);
        };
    }

    // La entidad se maneja; los cambios son persistentes
    public raiseEmployeeSalary (int id, long raise) {
        Employee.emp = findEmployee(id);
        if (emp != null){
            emp.setSalary(emp.getSalary()+ raise);
        };
        return emp;
    }

    public Collection<Employee> findAllEmployees() {
        Query query = em.createQuery("SELECT e FROM Employee e");
        return (Collection<Employee>) query.getResultList();
    }

}
```

Software de Comunicaciones

© The Authors

ORM: Entidades y Atributos simples

- Clase de entidad ↔ tabla:
  - Por defecto: nombre de tabla = nombre de clase sin calificar
  - Si se requiere otro nombre: úsese **`@Table(name="...")`**
  - Mapeo de una clase a múltiples tablas:
    - Usar anotaciones **`@SecondaryTables`** y **`@SecondaryTable`**
- Atributo de entidad cuyos valores son de un tipo único ↔ columna:
  - Notas relacionadas con los tipos simples:
    - Tipos primitivos, clases envoltorio de tipos primitivos, Strings, tipos temporales, tipos enumerados, objetos serializables,…
    - Se denotan explícitamente con **`@Basic`** pero no necesariamente a no ser que se añadan elementos, por ej. **`@Basic(fetch=FetchType.LAZY)`**
    - El conjunto de tipos permitidos para la clave primaria (denotada con **`@Id`** o **`@EmbeddedId`** / **`@IdClass`**) es un subconjunto de los tipos simples
  - Por defecto: el atributo es persistente y el nombre de la columna = nombre del atributo
    - No persistente: usar el modificador **`transient`** o la anotación **`@Transient`**
  - Si se requiere otro mapeo: hay que usar **`@Column(name="...")`**

Software de Comunicaciones

© The Authors

Ejemplo 2: Mapeo a tabla simple

```
// fuente: JBoss EJB3 tutorial, B. Burke
@Entity
@Table(name="AUCTION_ITEM")
public class Item {
  private long id;
  private String description;
  private String productName;
  private Set<Bid> bids = new HashSet();
  private User seller;

  @Id(generate=GeneratorType.AUTO)
  @Column(name="ITEM_ID")
  public long getId() {
      return id;
  }
  public void setId(long id) {
      this.id = id;
  }
  ….
}
```

```
create table AUCTION_ITEM
(
ITEM_ID Number,
DESC varchar(255),
ProductName varchar(255),
USER_ID Number
);
```

© The Authors

Ejemplo 2: Mapeo a tabla múltiple (1/2)

```
// fuente: JBoss EJB3 tutorial, B. Burke
@Entity
@Table(name="OWNER")
@SecondaryTable(name="ADDRESS",
pkJoinColumns={
 @PrimaryKeyJoinColumn(name="ADDR_ID",
   referencedColumnName="ADDR_ID"})
public class Owner {
    private long id;
    private String name;
    private String street;
    private String city;
    private String state;

   @Id(generate=GeneratorType.AUTO)
   @Column(name="OWNER_ID")
    public long getId() {
        return id;
    }
    public void setId(long id) {
        this.id = id;
    }
```

```
create table OWNER
(
OWNER_ID Number,
NAME varchar(255),
ADDR_ID Number,
);

create table ADDRESS
(
ADDR_ID Number,
STREET varchar(255),
CITY varchar(255),
STATE varchar(255)
);
```

Software de Comunicaciones

© The Authors

Ejemplo 2: Mapeo a tabla múltiple (2/2)

```
…

@Column(name="STREET",
      secondaryTable="ADDRESS")
 public String getStreet() {
      return street;
 }
 public void setStreet(String street) {
      this.street = street;
 }


 @Column(name="CITY",
      secondaryTable="ADDRESS")
 public String getCity() {
      return city;
 }
 protected void setCity(String city) {
      this.city  = city;
 }

 …
```

```
create table OWNER
(
OWNER_ID Number,
NAME varchar(255),
);

create table ADDRESS
(
ADDR_ID Number,
STREET varchar(255),
CITY varchar(255),
STATE varchar(255)
);
```

Software de Comunicaciones

© The Authors

Restricciones en la generación de esquemáticos

- Para especificar restricciones únicas
  - Úsese **`unique`**, elemento de **`@Column`** y **`@JoinColumn`**
  - Úsese **`uniqueConstraints`**, elemento de **`@Table`** y **`@SecondaryTable`**
- Para especificar una restricción no nula
  - Úsese **`nullable`**, elemento de **`@Column`** y **`@JoinColumn`**
- Para especificar restricciones de longitud de cadena
  - Úsese **`length`**, elemento de **`@Column`** (no hay tal elemento en **`@JoinColumn`**)
  - El valor por defecto es 255
- Para especificar restricciones de punto flotante
  - Úsese **`precision`** y **`scale`**, elementos de **`@Column`** (y no de **`@JoinColumn`**)
- Para especificar una cadena SQL en un DDL particular
  - Úsese **`columnDefinition`**, elemento de **`@Column`**, **`@JoinColumn`**, **`@PrimaryKeyJoinColumn`**, **`@DiscriminatorColumn`**

Software de Comunicaciones

© The Authors

ORM: Relaciones (1/2)

- Cada atributo cuyo valor es otra entidad
  - Puede ser anotada con una de las siguientes multiplicidades:
    - `@OneToOne`
    - `@OneToMany`
    - `@ManyToOne`
    - `@ManyToMany`
  - La clave foránea se denota con `@JoinColumn(name="...")`
- Cada asociación / relación se dice que tiene un lado perteneciente.
  - Una asociación bidireccional también se dice que tiene una lado inverso
  - En `OneToMany` y `ManyToOne`: el poseedor debe de tener muchos lados
  - En `OneToOne`: el poseedor debe de ser un lado que contiene una clave foránea
  - El lado inverso usa `mappedBy`, elemento de anotación de multiplicidades que identifica el atributo correspondiente sobre el lado perteneciente
- Las relaciones de composición se denotan con una especificación de elemento:
  - `cascade=REMOVE` sobre el lado inverso

Software de Comunicaciones

© The Authors

ORM: Relaciones (2/2)

- Se requiere una tabla de uniones para
  - Relaciones muchos-a-muchos
  - Relaciones unidireccionales uno-a-muchos
  - Ejemplo (muchos a muchos):

```
@Entity
public class Employee
@Id private ind id;
private String name;
@ManyToMany //En lado inverso hay @ManyToMany(mappedBy="projects")
@JoinTable(name="EMP_PROJ", // anotación sólo sobre lado poseedor
      joinColumns=@JoinColumn(name="EMP_ID"),
      inverseJoinColumns=@JoinColumn(name="PROJ_ID"))
private Collection<Project> projects;
```

- Columna de unión
  - Para el lado poseedor descrito en el elemento **`joinColumns`**
  - Para el lado inverso descrito en el elemento **`inverseJoinColumns`**
  - Se usan los nombres por defecto si no se especifica nada (también para la tabla de unión)

Software de Comunicaciones

© The Authors

ORM: Herencia

- La clase base de la jerarquía de herencia puede anotarse vía:
  - **`@Inheritance(InheritanceType=X)`**

  donde **`X`** es uno de los que sigue (por defecto es 1)
  1. **`SINGLE_TABLE`**: Patrón *Tabla de herencia simple*
  2. **`JOINED`**: Patrón de Herencia Clase-Tabla
  3. **`TABLE_PER_CLASS`**: Patrón de herencia de tabla concreta

- 2 y 3: la jerarquía puede contener clases que no se pueden persistir
  - “clases transitorias” (no entidades) o “superclases mapeadas”.

- **`SINGLE_TABLE`** y algunas veces herencia **`JOINED`**
  - La clase base tiene una anotación **`@DiscriminatorColumn`**
    - Los elementos especifican propiedades de la columna del discriminador, por ejemplo *tipo*
  - Cada entidad concreta usa una anotación **`@DiscriminatorValue`**

- Las peticiones sobre la clase de jerarquía son polimórficas

© The Authors

Sincronización con la base de datos

- La base de datos escribe de forma encolada hasta que **`EntityManager`** se sincroniza
  - Con llamadas a los métodos **`persist()`**, **`merge()`**, **`remove()`** de **`EntityManager`**
  - No coordinada con actualizaciones / destrucciones enviadas por la interfaz **`Query`**
- Escribe de forma alineada a la base de datos de acuerdo a una de las siguientes políticas:
  - Cuando una transacción se compromete
    `javax.persistence.FlushModeType= COMMIT`
  - Antes de que una petición concerniente a esos datos se ejecute
    `javax.persistence.FlushModeType = AUTO`
- Manejado explícito del alineamiento:
  - averiguar / cambiar la política de alineamiento
    `EntityManager.getFlushMode()`
    `EntityManager.setFlushMode(FlushModeType flushMode)`
    `Query.setFlushMode(FlushModeType flushMode)`
  - Para descargar cambios explícitamente a la base de datos
    `EntityManager.flush()` (c.f. también **`EntityManager.refresh()`**)

© The Authors

Cierre de la base de datos

- Política por defecto de cierre: control optimista de la concurrencia
  - En el compromiso de la transacción
    - Si los datos han cambiado con respeto a la versión en memoria: deshacer la transacción
    - Si no es así, seguir adelante con el compromiso
  - Basado en el uso de un campo de versión
    - Campo de entidad anotado con `@Version`
    - Se comprueba el número de versión para detectar cambios
- Otras posibilidades
  - Fijar el nivel de aislamiento globalmente (ver la sección de transacciones)
  - Al método `EntityManager.lock()`, pasarle una entidad como parámetro
    - Cerrojo de lectura: parámetro `LockModeType.READ`
      - Una actualización de transacción puede fallar si otra transacción tiene un cerrojo de lectura
    - Cerrojo de escritura: parámetro `LockModeType.WRITE`
      - Fuerza incrementos del campo de versión si la entidad es actualiza o no

Software de Comunicaciones

© The Authors

Métodos de *Call-Back* y Ciclo-de-Vida

- En los métodos llamados por el proveedor de persistencia
  - Una entidad se registra opcionalmente por métodos *call-back* (retrollamada) sobre eventos del ciclo de vida
    - Mediante anotaciones de la clase del bean
  - Restricciones sobre un método que puede ser del ciclo de la vida o retrollamada:
    - No tiene argumentos, retorna void, no lanza excepciones de aplicación
  - Pueden ser colocados en una clase escuchadora separada:
    - Declarada con la anotación **`@EntityListener`**

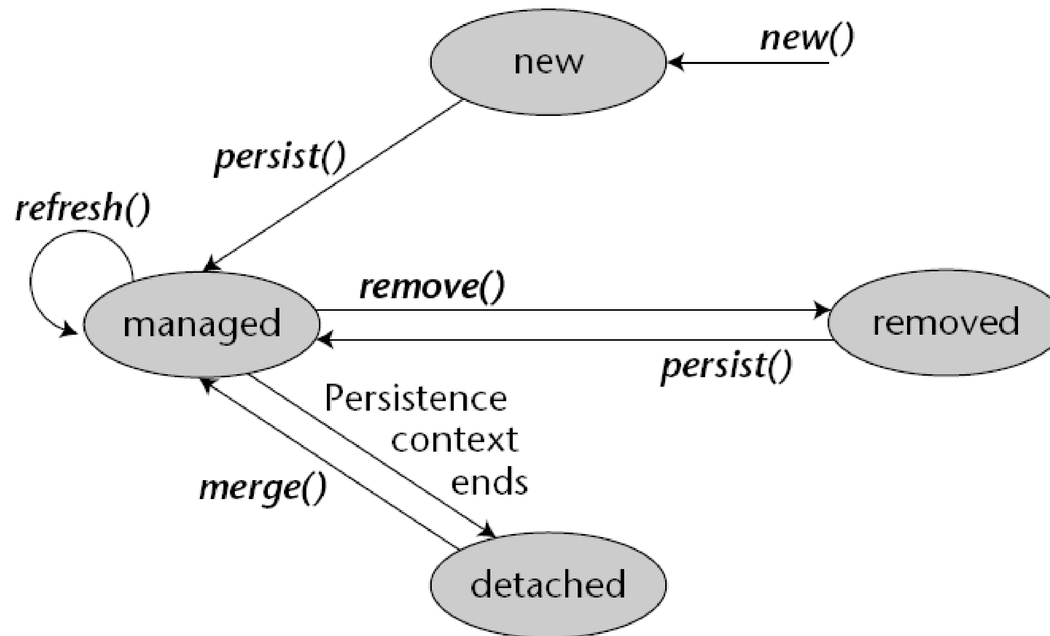
- Métodos posibles de retrollamada y ciclo de vida:
  - Método anotado con **`@PrePersist`** / **`@PostPersist`**
    - Llamado antes / después de ejecutar **`EntityManager.persist()`**
  - Método anotado con **`@PreRemove`** / **`@PostRemove`**
    - Llamado antes / después de ejecutar **`EntityManager.remove()`**
  - Método anotado con **`@PreUpdate`** / **`@PostUpdate`**
    - Llamado antes / después de la sincronización con la base de datos
  - Método anotado con **`@PostLoad`**
    - Llamado después de que una instancia se cargue vía EM **`find()`** o **`getReference()`**

Software de Comunicaciones

© The Authors

Ciclo de vida de la entidad

Fuente: Pro EJB3. Java Persistence API. Mike Keith et al



- Contexto de persistencia transaccional
  - El contexto de persistencia finaliza cuando finaliza la transacción
- Contexto de persistencia extendido
  - El contexto de persistencia finaliza cuando el llamante es destruido

© The Authors

Transacciones locales a recurso en JPA

- Las entidades pueden ser usadas dentro de transacciones locales a recurso
  - Transacciones bajo control completo de la aplicación
  - A través de la interfaz **`EntityTransaction`**
- La interfaz **`EntityTransaction`**
  - Es similar a la interfaz JTA **`UserTransaction`** (ver más tarde)
  - Operaciones: **`begin`**, **`commit`**, **`rollback`**, **`isActive`**, **`setRollbackOnly`**, **`getRollbackOnly`**,
  - Operaciones implementadas vía métodos de transacción JDBC **`Connection`**
  - Obtenidos vía **`EntityManager.getTransaction()`**
- No son usados normalmente cuando se está en un entorno Java EE
  - Se recomienda utilizar transacciones JTA (ver más tarde)

Software de Comunicaciones

© The Authors

Inversión de Control (IdC)

- Inversión de control (IdC) se refiere a una situación donde
  - Un sistema externo llama al código
  - En vez de que tu código llame al sistema externo
- Algunas veces se refieren a ese mecanismo como el principio Hollywood:
  - “no nos llames, nosotros te llamamos a ti".
- Ejemplo: Métodos de retrollamada de un EJB

© The Authors

Inyección de Dependencia (ID) (1/2)

- Para obtener un servicio dado, los componentes necesitan conocer:
  - Con qué otros componentes se necesitan comunicar
  - Dónde localizarlos
  - Cómo comunicarse con ellos
- Aproximación ingenua
  - Embeber la lógica de localización / instanciación de servicio en el código de su cliente
  - Problemas con la aproximación ingenua:
    - Los cambios en el acceso del servicio implican cambios en el código de muchos clientes
    - Difícil en casos de pruebas unitarias con componentes sin usar un servicio real
- Aproximación inyección de dependencia (ID)
  - Los clientes declaran sus dependencias en servicios
  - “código externo” inyecta el servicio
    - Esto es, asume la responsabilidad de localizar e instanciar los servicios y después provee la referencia del contenedor
  - “código externo”: a menudo un contenedor/armazón de IDs

© The Authors

Inyección de dependencia (ID) (2/2)

- Inyección de dependencia
  - Es una forma particular de IdC
  - Asegura que la configuración de servicios está separada del uso
  - Las dependencias están configuradas externamente en un lugar
    - Para clientes múltiples
- Externalizar la dependencia de acceso a servicio hace el código
  - Más reusable
  - Más fácil de probar: facilidad de inyección sobre un servicio fingido
  - Más legible
- Maneras de llevar a cabo la inyección de dependencia
  - Inyección del constructor
  - Inyección del método set
  - Inyección de la interfaz

Software de Comunicaciones

© The Authors

Ejemplo: Inyección de dependencia (1/3)

- Obteniendo un gestor de entidad gestionado para la aplicación (Java SE)
  - Usando la clase **Persistence** (sin inyección de dependencia)

```
// fuente: Pro EJB3, Java Persistence API. M. Keith, M. Schincariol
public class EmployeeClient {
   public static void main(String[] args) {
      EntityManagerFactory emf
       = Persistence.createEntityManagerFactory("EmployeeService");
      EntityManager em = emf.CreateEntityManager();

      Collection emps
      = em.CreateQuery("SELECT e FROM Employee e").getResultList();
      for(Iterator I = emps.iterator(); i.hasNext();){
         Employee e = (Employee) i.next();
         System.out.println(e.getId() + ", " + e.getName());
      }
   em.close();
   emf.close();
}
```

Software de Comunicaciones

© The Authors

Ejemplo: Inyección de dependencia (2/3)

- Obteniendo un gestor de entidad gestionado por la aplicación (Java EE)
  - Inyección de unidad de persistencia para obtener **EntityManagerFactory**
  - El contenedor Java EE realiza la inyección

```
// fuente: Pro EJB3, Java Persistence API. M. Keith, M. Schincariol
public class LoginServlet extends HttpServlet {
   @PersistenceUnit(unitName="EmployeeService")
   EntityManagerFactory emf;
   protected void doPost(HttpServletRequest request,
                         HttpServletResponse response){
      String userId = request.getParameter("user")
      // comprobar si es un usuario válido
      EntityManager em = emf.createEntityManager();
      try{
         User user = user.find(User.class, userId);
         if (user == null) {
            // retornar una página de error …
         }
      } finally {em.close();}
      // ...
   }
}
```

Software de Comunicaciones

© The Authors

Ejemplo: Inyección de dependencia (3/3)

- Obteniendo una gestor de entidad manejado por el contenedor (Java EE)
  - Inyección de contexto de persistencia para obtener un **EntityManager**
  - El contenedor Java EE hace la inyección
  - Explicación de la anotación **@stateless**: ver más adelante

```
// fuente: Pro EJB3, Java Persistence API. M. Keith, M. Schincariol
@Stateless
public class ProjectServiceBean implements ProjectService {
   @PersistenceContext(unitName="EmployeeService")
   EntityManager em;

   public void assignEmployeeToProject(int empId, int projectId) {
      Project project = em.find(Project.class, projectId);
      Employee employee = em.find(Employee.class, empId);
      project.getEmployees().add(employee);
      employee.getProjects().add(project);
   }
   // ...
}
```

Software de Comunicaciones

© The Authors

Bibliografía para la Sección 2

- *Pro EJB3. Java Persistence API*. Mike Keith and Merrick Schincariol. Apress, 2006
  `http://books.google.com/books?id=fVCuB_Xq3pAC`
- *The Java Persistence API*. Sun Microsystems
  `http://java.sun.com/javaee/technologies/persistence.jsp`
- *The Java EE 5 Tutorial. Part V: Persistence*. Sun Microsystems
  `http://java.sun.com/javaee/5/docs/tutorial/doc/bnbpy.html`
- *Inversion of Control Containers and the Dependency Injection pattern.* Martin Fowler, 2004
  `http://www.martinfowler.com/articles/injection.html`

© The Authors