CUADRO RESUMEN DE LA LECTURA 2 (ORM)

SIMÓN NARANJO - 1255229
UNIVERSIDAD DEL VALLE
INGENIERÍA DE SISTEMAS DE INFORMACIÓN
ACTUALIZACIÓN EN COMPETENCIAS LABORALES
ING. DIANA LORENA VELANDIA
TULUÁ
2016

ASPECTOS MÁS IMPORTANTES	DETALLES
ARQUITECTURA POR CAPAS	Usar un ORM nos permite trabajar con los datos usando un paradigma orientado a objetos, el cuál se usa para crear aplicaciones de múltiples capas.
PERSISTENCIA DE OBJETOS	Por lo general un objeto es creado, usado y eliminado, pero usando un ORM podremos guardar el estado de los objetos en el disco, lo cual nos permite reconstruir dichos objetos con su mismo estado.
MANEJO DE BASES DE DATOS RELACIONALES	Inicialmente para acceder a los datos de una RDB se utilizaba una JDBC API pero tenia inconvenientes como lo eran tener que escribir mucho código SQL y realizar a mano conexiones y desconexiones. Usando una herramienta que nos permita trabajar con ORM nos estamos ahorrando todo ese trabajo y tiempo.
METADATOS	Un ORM utilizada el código que se ha generado automáticamente a partir de metadatos para acceder a las tablas, por ejemplo tipos de datos y llaves.
PROBLEMA DE GRANULARIDAD	En un paradigma orientado a objetos podemos definir clases e incrustarlas dentro de otras libremente sin limites, en cambio usando una base de datos SQL solo podremos tener una granularidad de nivel dos(Tabla y Columna), un ORM entonces debe hacer uso de JOIN para permitir más niveles en una clase.
PROBLEMA DE LOS SUBTIPOS	Una de las ventajas de usar un paradigma orientado a objetos es poder definir herencias y polimorfismos, pero desafortunadamente, por lo general las bases de datos SQL no soportan esas características y las que lo hacen no tienen una sintaxis estándar. Un ORM puede permitir trabajar con herencias y polimorfismos pero para ello requiere que las tablas tengan una estructura especial.
PROBLEMA DE IDENTIDAD	En Java para establecer la similitud entre dos objetos usamos el "==" o "equals" pero en una fila de base de datos la identidad está dada por una llave primaria y no hay una forma "estándar" de hacer esa comparación de similitud.
PROBLEMA DE LAS ASOCIACIONES	En un paradigma OO las asociaciones se representan usando referencia a objetos, pero en cambio las asociaciones en RDB se representan a través de llaves foráneas. Un ORM entonces debe mapear esas referencias de llaves foráneas en objetos.

PROBLEMA DE NAVEGACIÓN DE DATOS	En un paradigma orientado a objetos se accede a datos entre objetos usando getters en cambio en una base de datos relacional se necesita hacer uso de JOINS entre tablas, en Java este problema se trata con JPQL.
ENTIDAD	Una entidad es una clase que representa una tabla y cada instancia de una entidad representa una fila.
ENTIDADES INCRUSTADAS	Algunas entidades pueden usar a otras para representar su estado, aquellas entidades de nivel dos no tienen una persistencia propia y en cambio solo existen dentro del estado de la entidad primaria.
CONTEXTO DE PERSISTENCIA	Es una especie de "caché" que será sincronizada con la DB, contiene una serie de entidades de persistencia, cuando la operación termina todos los objetos persistentes son separados del contexto y por lo tanto ya no serán gestionados.
CONTROLADOR DE ENTIDADES	Es una API usada para crear y eliminar instancias de entidades de persistencia, así como también para buscar llaves primarias y buscar datos de las instancias de las entidades.
UNIDAD DE PERSISTENCIA	Define las entidades que van a ser gestionadas por un controlador de entidades, así como también define todas las clases que serán relacionadas por la aplicación. Una unidad de persistencia se define en el archivo persistence.xml
ESTADO DE LAS INSTANCIAS DE UNA ENTIDAD	Un ORM no solo evita tener que especificar cada aspecto del manejo de una base de datos, sino que también nos permite hacer un manejo del ciclo de vida de cada instancia a través de estados.
ESTADO "NEW"	Este estado se da cuando instanciamos una entidad pero no la hemos relacionado ni con un ID, ni con una DB, ni con un contexto. Se dice que no tiene identidad de persistencia.
ESTADO "MANAGED"	Una entidad pasa a este estado cuando ya cuenta con una identidad de persistencia, lo que da a entender que ya tiene un ID único(llave primaria), tiene representación en la DB y está asociada a un contexto de persistencia, al estar en este contexto está a la espera de un commit para ser sincronizada con la DB, después de ser sincronizada su estado cambia a "DETACHED".
ESTADO "DETACHED"	Este estado se da cuando una entidad no está asociada a un contexto pero si tiene identidad de persistencia, lo cual le permite seguir existiendo y ser modificada por la aplicación. Cuando se desee sincronizar una entidad en este estado con la DB es necesario usar el método "merge()" para que su estado cambie a "MANAGED".
ESTADO "REMOVED"	Una instancia en este estado es similar a una instancia del estado "MANAGED", la diferencia está en que al darse el commit y hacer la sincronización con la DB los datos serán eliminados y no actualizados.
RELACIÓN ENTRE ENTIDADES	Las relaciones entre entidades pueden ser de 8 tipos, dependiendo de las necesidades al momento de almacenar los datos. En las DBs estas relaciones pueden ser de los siguientes tipos: UnoAUno, UnoAMuchos, MuchosAUno, MuchosAMuchos y tienen una dirección la cual pude ser UniDireccional o BiDireccional.

JAVA PERSISTENCE QUERY LANGUAGE (JPQL)	Este lenguaje define consultas sobre entidades, su persistencia y su estado. Se diferencia de SQL en que este trabaja sobre nombres de objetos(modelos) y no con tablas y columnas, para ello usa el esquema de persistencia de las entidades. Por otra parte su sintaxis es similar a la de SQL.
HERENCIA DE ENTIDADES	Las bases de datos por lo general no soportan herencia, ni polimorfismo, pero aplicando algunas estrategias al momento de mapear una clase es posible emular dichos comportamientos, además se requieren unas tablas con unas condiciones "especiales". Actualmente hay tres estrategias siendo la primera, la estrategia por defecto: Una sola tabla por jerarquía de clases, otra tabla con los campos de las subclases, una tabla por cada clase/entidad.
UNA SOLA TABLA POR JERARQUÍA DE CLASES (DEFAULT)	En esta estrategia todas las clases van en la misma tabla, y debe existir una columna en la cual se indique si la fila pertenece a la clase padre o a alguna de las subclases.
OTRA TABLA CON LOS CAMPOS DE LAS SUBCLASES	En esta estrategia, la clase padre va en una tabla y todas las subclases van en otra, en la cual se deben almacenar los campos pertenecientes solo a las subclases.
UNA TABLA POR CADA CLASE	Cada clase se mapea en una tabla diferente y todos sus campos son mapeados en ella.

REFERENCIAS	1. Bauer, C. & King, G. 2007. Java Persistence with Hibernate. Manning Publication Co.
	2. King G., Bauer C., Andersen M. R., Bernard, E. & Ebersole S. 2009. Hibernate Reference Documentation. Red Hat Middleware. (http://docs.jboss.org/hibernate/core/3.3/reference/en/html/)
	3. Sriganesh, R., P. 2006. Mastering Enterprise JavaBeans 3.o. Wiley Publishing, Inc.
	4. Sun Microsystems. 2009. JSR 317: Java Persistence API, Version 2.0. Final Release.