PERSISTENCIA DE OBJETOS

Qué es la persistencia

Definición

La persistencia es uno de los aspectos fundamentales en cualquier aplicación que trabaja con datos. Representa el acto de persistir los datos en el tiempo.

Se utiliza normalmente para almacenar datos en una base de datos. En aplicaciones OO, persistencia significa persistir el estado del objeto en el tiempo.

Bases de datos relacionales

Las base de datos relacionales son la forma por excelencia de almacenar datos. Son extremadamente flexibles y robustas para la administración de datos. Los DBMS (DataBase Management Systems) poseen interfaces basadas en SQL para interactuar con los datos.

SQL tiene como categorías a DDL y DML:

- DDL significa Data Definition Language, y se utiliza para la creación de esquemas y tablas. Los comandos más conocidos son CREATE, ALTER y DROP.
- DML significa Data Manipulation Language, y se utiliza para la manipulación de datos. Los comandos más conocidos son SELECT, INSERT, UPDATE, DELETE.

Archivos planos

Los archivos planos son otra forma de persistencia. Si bien es posible guardar casi cualquier informacion, se hace relativamente difícil poder organizar y consultar de manera eficiente la información deseada.

En caso de objetos, se podrían persistir en archivos planos a través de serialización, y volver a utilizarlos a través de la deserealización.

Modelo Relacional vs. Modelo OO

Problemática

La problemática que se plantea es que los RDBMS (Relational DBMS) están basados en el modelo relacional, pero por su lado las aplicaciones orientadas a objetos están basadas en el paradigma de objetos.

Ambos modelos tienen sus propias características, y como consecuencia se genera la problemática de tratar datos de una tabla como un objeto y viceversa, es decir llevar datos de un objeto a una tabla.

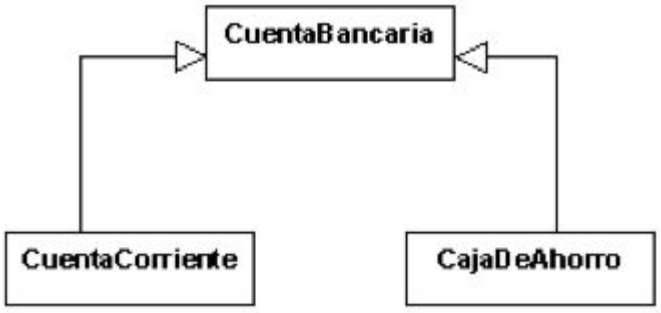
Se genera lo que se conoce como un “gap” entre ambos modelos, dificultando la integración y coordinación de ambos.

Una tabla, una clase

La forma más sencilla de relacionar ambos modelos es tratar a cada clase en el modelo de objetos, como una tabla en el modelo relacional. Si bien es la forma más sencilla de trabajar, se presentan varios problemas a tratar debido a las diferecias entre el paradigma de objetos y el relacional.

1) El problema de la herencia

Uno de los pilares del paradigma OO es la herencia. La herencia representa a la relación “es un”. Es muy común trabajar con diagramas de clases que poseen una gran jerarquía de clases a través de la herencia.



El modelo relacional está basado en la relación “tiene un”, pero no cuenta con la relación “es un”. La problemática se basa en que dentro del modelo relacional, no es posible armar una relación de herencia.

2) El problema de la identidad

Para realizar chequeos de identidad en registros, se utiliza la PK (Primary Key) como método de comparación.

Para realizar chequeos de identidad en objetos, se utilizan los métodos equals() y compareTo() según corresponda, como también el operador == dependiendo del caso.

Suponiendo la siguiente definición de la tabla usuarios:

La problemática surge cuando hay que realizar una actualización en cascada de la PK desde el modelo de objetos – el campo nombre - ya que el impacto es grande y genera procesamiento adicional que podría ser evitable.

Para evitar este tipo de problemática, la recomendación es utilizar una PK que nada tenga que ver con los datos, posiblemente un campo auto-numérico, y construir un atributo adicional ID en la clase correspondiente.

3) El problema de las asociaciones

En el paradigma de objetos, las asociaciones representan la forma de relacionar objetos.

Un objeto podría tener a otro objeto como asociado, por ejemplo un Auto tiene un Stereo, entonces es posible decir que la clase Auto tiene como atributo un objeto del tipo Stereo (obviamente, Stereo tiene sus propios atributos).

En el modelo relacional, dichas asociaciones se presentan a través de las claves foráneas.

Las asociaciones tienen dirección, es decir que cada clase que forma parte de la relación, tiene (o no) una referencia a la otra clase.

En una relación unidireccional, una de las dos clases tiene como referencia a la otra. Por ejemplo, el Auto tiene como referencia al Stereo, o podría ser al revés también, el Stereo tiene como referencia al Auto.

En una relación bidireccional, ambas clases se tienen como referencia. Por ejemplo, el Auto tiene como referencia al Stereo y el Stereo tiene como referencia al Auto.

Las claves foráneas no son en su concepción bidireccionales, lo que generaría inicialmente un desacople entre el modelo relacional y el modelo de objetos.

Un objeto podría tener a un conjunto (una colección) de objetos asociados, por ejemplo un Aula tiene Alumnos. La clase Aula tiene como atributo un objeto del tipo Set que contiene objetos del tipo Alumno (obviamente, cada Alumno tiene sus propios atributos).

Es normal contar con relaciones del tipo muchos-a-muchos, por ejemplo un Aula posee muchos alumnos, pero un Alumno puede tener asignada más de un Aula.

En el modelo de objetos, existen únicamente dos clases, pero en el modelo relacional surge una nueva tabla manejar la relación muchos-a-muchos.

4) El problema de la navegación

La relación entre clases está dada a través de una nueva clase, o colecciones. Por ejemplo, la clase Universidad tiene Facultades, la Facultad tiene Alumnos, el Alumno tiene Asignaturas, y así sucesivamente.

El problema está en que al realizar la consulta hay que determinar hasta qué nivel de información (es decir hasta qué objetos) traer como datos.

El problema de la navegación es uno de los problemas que más impacta sobre la performance de la aplicación.

Qué es ORM

Definición

ORM significa Object / Relational Mapping. Es el middleware que maneja la persistencia en la capa de acceso a datos.

Tiene como objetivo manejar la persistencia de objetos de una aplicación dentro de una base de datos, en forma automatizada y transparente. Utiliza metadatos para describir la relación entre el modelo de clases y el modelo de tablas.

Maneja los casos más comunes, que son los que aparecen en mayor cantidad. Los casos especiales deben ser manejados por el usuario, aunque éstos deberían ser los mínimos.

El uso de un ORM suele disminuir el tiempo de construcción de una aplicación en un 30%. Se estima que el trabajo de desarrollo de la capa de acceso a datos disminuye en un 85%.

Organización

Una solución del tipo ORM tiene como partes a:

1. Una API para realizar los ABMC en objetos de clases persistentes
2. Un lenguaje que permite realizar consultas referenciando a clases y atributos.
3. Una forma de especificar los mapeos entre meta-datos

Ventajas

Las ventajas que provee la utilización de un ORM son las siguientes:

- Independencia de la base de datos. Provee una abstracción de la base de datos utilizada, lo cual permite - en caso de no utilizar SQL propietario – una fácil migración.
- Productividad. Reduce significativamente el tiempo de desarrollo.

- Fácil mantenimiento. Agrega simplicidad al código, lo que hace que el mantenimiento sea simplificado.
- Menor trabajo. Disminuye de forma importante las cantidad de líneas de código de la aplicación, ya que la mayoría del código ya está pre-escrito.

Tecnologías ORM

User-defined DAOs

DAO significa Data Access Object. Es un patrón de diseño utilizado para realizar operaciones contra una base de datos, pero abstrayendo de la lógica de la misma. La lógica de acceso a datos queda encapsulada dentro de una DAO con lo cual no es visible para usuarios que utilizan el DAO.

El User-defined DAO es un DAO definido por el usuario. El desarrollador puede armar su propio modelo de acceso a datos con la construcción de diversos DAOs, formando así la DAL (Data Access Layer, o capa de acceso a datos).

Es una de las opciones más utilizadas para la persistencia de datos. En Java puntualmente, se trabaja con SQL y JDBC directamente. No se necesita ningún contenedor especial para utilizarlos. Son clases planas con funcionalidad.

Tiene como desventaja que se paga el precio de “reinventar la rueda”, debido a que hay que realizar el 100% de la codificación.

Jpa

Sus siglas significan Java Persistence API, es una API de persistencia desarrollada para la plataforma JAVA EE.

Todos los componentes de La API se encuentran definidos en el paquete javax.persistence.

JPA es una especificación, lo que significa que por sí sola no se puede implementar. Podemos utilizar annotations de JPA en nuestras clases, sin embargo sin una implementación nada sucederá. Hibernate es una implementación de JPA, cumple las especificaciones de JPA y además aporta sus propias annotations y funcionalidades.

Cuando se utilizan las annotations de JPA con Hibernate en realidad se está utilizando la implementación de Hibernate para las annotations de JPA. El beneficio es que podemos cambiar el ORM que estamos utilizando por cualquier otro que implemente JPA sin tener que modificar nuestro código.

Algunas implementaciones de JPA son:

- Hibernate
- ObjectDB
- TopLink
- CocoBase
- EclipseLink
- OpenJPA
- Kodo
- DataNucleus, antes conocido como JPOX
- Amber

Hibernate

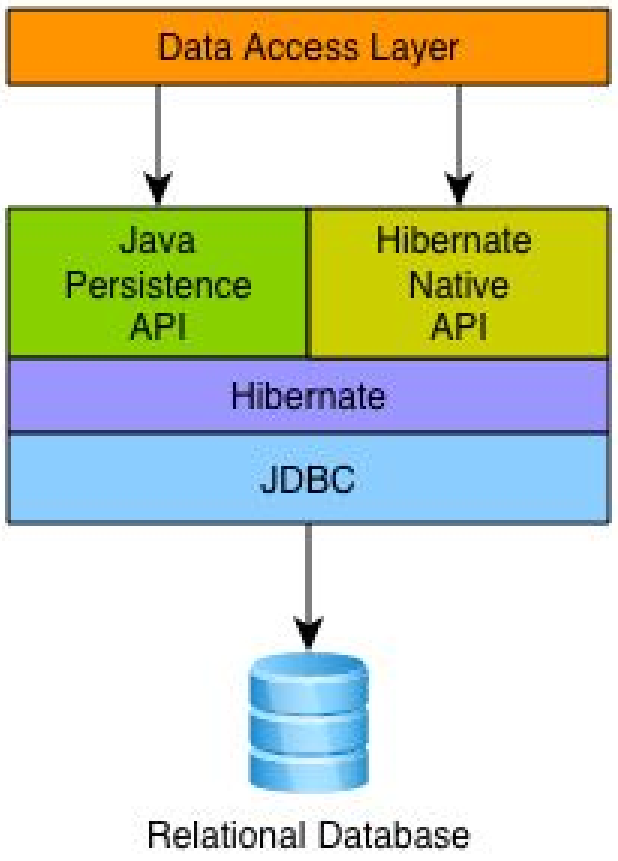
Es un framework – construido en JAVA - que se encarga de manejar la persistencia de datos. Es la implementación de un ORM.

Es un proyecto de código abierto no comercial (inicialmente), bajo los aspectos de la licencia pública GNU. En el año 2003 se une con JBoss.org y consigue su lado comercial: venta de soporte y capacitación.

Los fuentes/binarios están disponibles en http://www.hibernate.org

Tiene como objetivo ser una solución completa a la problemática de persistencia de datos con tecnología Java, dejando al desarrollador concentrarse fundamentalmente en los aspectos de negocio. Aumenta fuertemente la productividad y facilita la administración de la capa de acceso a datos.

Uno de los aspectos importantes de Hibernate es que está construido con clases java básicas, es decir que puedo agregar el framework a un proyecto propio. No se requiere ningún tipo de container (por ejemplo, un application server).



La necesidad de una DAL (Data Access Layer)

Arquitectura multicapa (n-tier Architecture)

Es una forma de organizar un sistema. Cada capa está compuesta - generalmente - por un conjunto de clases que cooperan con un objetivo en común. Cada capa le brinda servicios a la capa superior directa, no se realizan llamados entre capas “que no se ven”.

Una de las arquitecturas más utilizadas es la arquitectura 3 capas:

- PL (Presentation Layer)
- BL (Business Layer o Business Logic)
- DAL (Data Access Layer)

Se la denomina 3TA à 3-Tier Architecture, y provee una separación lógica (no necesariamente física) entre las 3 capas.

La PL, o Capa de Presentación, se encarga de administrar la lógica correspondiente a mostrar la información en pantalla. Generalmente viene acompañada del uso del patrón de diseño M-V-C.

La BL, o Capa de Negocios, se encarga de administrar la lógica de negocios, aquí es donde reside la implementación de las reglas de negocio (Business Rules).

La DAL, o Capa de Acceso a Datos, se encarga de manejar cualquier comunicación con la base de datos, ya sea para obtener información como para escribir.

Como ejemplo y basándonos en tecnología Java, la PL podría estar construida con JSP y Struts, la BL con EJB de tipo session, y la DAL con Hibernate.

Qué es DAL

DAL significa Data Access Layer, y representa la capa de Acceso a Datos. Se encarga de cualquier comunicación con la base de datos. Maneja todo tipo de operaciones: operaciones DML tales como los AMBC, como también la posibilidad de manejar DDL.

Generalmente, es un grupo de clases encargadas de realizar tanto la obtención de datos como su persistencia. Le brinda servicios a la capa de negocios.

Qué es POJO

Un POJO es un Plain Old Java Object, aunque también a veces se lo denomina Plain Ordinary Java Object. Es una clase que se encarga de manejar la persistencia.

Persigue las mismas características que los JavaBeans, es decir:

- Por lo general, es serializable, con lo cual implementa la interfaz (vacía) Serializable
- Posee atributos privados y métodos públicos de acceso
- Posee un constructor vacío, lo cual es obligatorio para trabajar con Hibernate

A continuación se presenta un ejemplo de un POJO:

[code]

```
public class Usuario implements Serializable {

        private String nombre;

        private String clave;



        public Usuario(){

        }



        // Setters y getters

                . . .
```

. . .

}

[/code]

Ventajas de una arquitectura Multi-Capa

Las ventajas de una arquitectura multi-capa son las siguientes:

- Provee una separación lógica entre las grandes tareas del sistema
- Permite que cada capa se concentre en una única tarea
- Aumenta la organización
- Aumenta la escalabilidad
- Facilita el mantenimiento

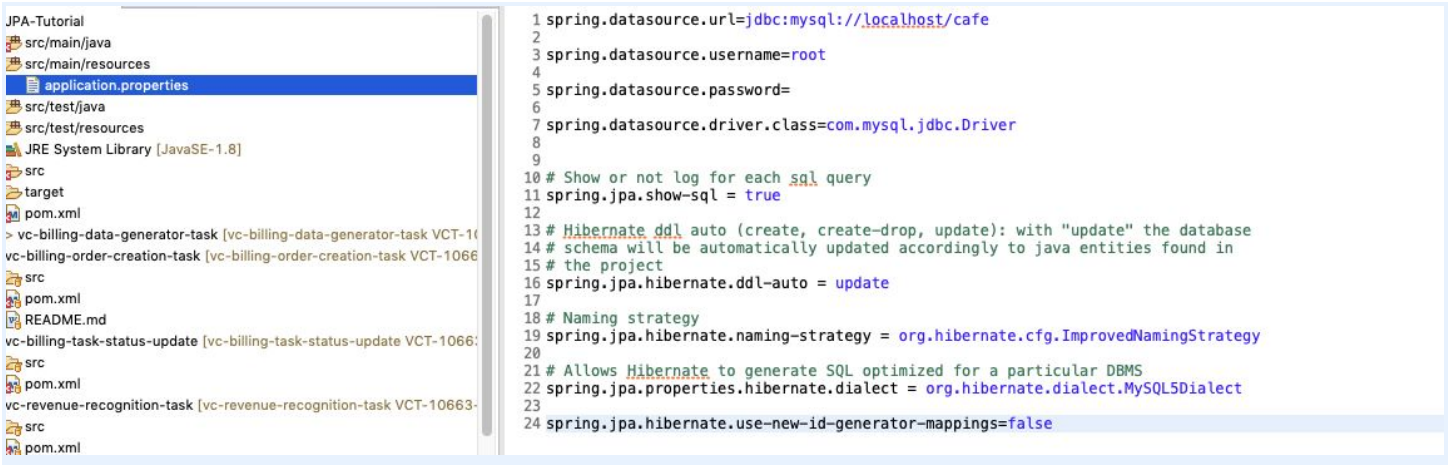
INTRODUCCIÓN A UN PROYECTO CON HIBERNATE

Configuración

Para configurar un proyecto con maven, es necesario agregarlo en el pom.xml de la siguiente manera:

```
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.6.RELEASE</version>
        <relativePath /> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>


        <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <!--version>5.1.47</version-->
        </dependency>
```

Y luego es necesario crear un archivo llamado application.properties que es el archivo utilizado por Hibernate para configurar su funcionamiento. Debe agregarse en la carpeta src/main/resources.



Propiedades de Hibernate

spring.datasource.driver.class representa el driver a utilizar. Corresponde al nombre de la clase que implementa el driver JDBC, incluido en el jar correspondiente. Dicho .jar, como ha sido especificado anteriormente, deberá formar parte del CLASSPATH de la aplicación.

spring.datasource.url representa la url de conexión a utilizar. Especifica el host, el puerto y la base de datos a utilizar. La url de conexión depende directamente del DBMS.

spring.datasource.username representa el usuario en la conexión a utilizar.

spring.datasource.password representa la contraseña en la conexión a utilizar.

spring.jpa.properties.hibernate.dialect representa el dialecto (el lenguaje) a utilizar. Es necesario determinarlo ya que la implementación de SQL es distinta en cada uno de los DMBS propietarios. El dialecto depende directamente del DBMS.

spring.jpa.show-sql especifica si mostramos como hibernate genera cada query.

spring.jpa.hibernate.ddl-auto especifica si queremos que hibernate modifique la base de datos, basandose en las entidades de java. Esto es muy útil en desarrollo pero no recomendable para Producción.

Las opciones son:

- validate: valida el schema, no cambia la base de datos.
- update: cambia la base de datos.
- create: crea el schema, destruyendo datos anteriores.
- create-drop: igual que el anterior pero además dropea el schema cuando se cierra la aplicación.

spring.jpa.hibernate.naming-strategy permite definer una estrategia para los nombres de las tablas.

spring.jpa.hibernate.use-new-id-generator-mappings permite definer una estrategia para los nombres de las claves primarias.

Hibernate/JPA Annotations

Qué son

Existen dos formas para realizar la transformación de información de un POJO a la base de datos y viceversa. Una es mediante archivos XML y otra es con Annotations, cada una tiene sus pros y sus contras, sin embargo las Annotations son la forma más nueva de realizar el mapeo de POJOs a la base de datos, en este curso utilizaremos la metodología de Annotations para la realización de todos los ejercicios.

Cuando se utiliza Hibernate Annotations toda la metadata está embebida en la misma clase Java que contiene al POJO, junto con el código Java, esto ayuda al desarrollador a relacionar la estructura de la tabla y del POJO al mismo tiempo que se escribe el código.

@Entity: A partir de esta annotation indicamos a Hibernate que nuestra clase es persistente. Se recomienda el uso de la annotation Entity de JPA, la de Hibernate suele traer dificultades. Una vez declarada persistente nuestra clase, todos sus atributos lo serán también por default.

@Table(name = "ITEM"): A través de esta annotation especificamos un nombre de tabla en caso de diferir al de nuestra clase. De no indicarlo, Hibernate tomará como nombre de la tabla asociada, el mismo que el de la clase. Esta es una convención, y constituye un ejemplo sencillo del concepto "convention over configuration".

@Id: A través de esta annotation indicamos que la property mapeada será la PK de nuestra tabla. En este caso se trata de una clave simple.

@GeneratedValue: Especifica la estrategia de generación de Ids. Hay varias opciones para generadores. En nuestro caso, al no acompañarla de parámetros adicionales, su valor será AUTO, que decide una estrategia de generación de Ids conveniente dependiendo del motor subyacente. (ej. Identity).

@Column(name = "ITEM_ID"): En este caso la annotation es opcional, si pretendemos que la columna de la tabla correspondiente se llame igual que la property. Aquí valen las mismas consideraciones que en el caso de @Table.

@Repository: es una anotación de Spring que especifica que es una clase que trabaja con entidades y es levantada por el container de Spring como un bean.