

## CICLO 2

[FORMACIÓN POR CICLOS]


# Introducción a **HIBERNATE**



Ingeni@  
Soluciones TIC



UNIVERSIDAD  
DE ANTIOQUIA  
Facultad de Ingeniería

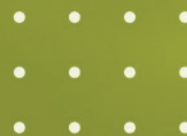


La persistencia usando bases de datos es el principal mecanismo usado en el desarrollo de aplicaciones para lograr que la información asociada a las mismas se guarde en el mediano y largo plazo. Las bases de datos, en la mayoría de los casos, se almacenan en servidores específicos a un sistema de gestión de bases de datos (SGBD). Por su parte, Java ofrece la API *Java Database Connectivity* (JDBC) para acceder a dichas bases de datos. Como sabemos, JDBC constituye un puente que permite comunicar las aplicaciones escritas en Java con bases de datos en general, y con MySQL en particular.

Sin embargo, el contar con herramientas para esta tarea, no necesariamente significa que podamos hacer una comunicación fácil y eficaz entre una aplicación escrita en Java y una base de datos. Un programa escrito en Java, en la mayoría de los casos, estará orientado a objetos; mientras que una base de datos, al menos en SGBDs como MySQL, Oracle o MS SQL Server, se enmarcará en el paradigma relacional. El primero, tiene como componentes fundamentales las clases, los objetos y sus conceptos relacionados; mientras que el segundo, tiene como componentes fundamentales las relaciones (tablas), los registros, los campos y sus conceptos relacionados. Estas diferencias llevan a incongruencias entre ambos paradigmas, que llevan a la necesidad de un gran esfuerzo a la hora de escribir código para comunicarse directamente con una base de datos.

Es en este contexto donde surgen las herramientas de Mapeo Objeto-Relacional u *Object/Relational Mapping* (ORM). ORM es una técnica de mapeo entre el modelado objetual propio de lenguajes de programación orientados a objetos y el modelado relacional propio de bases de datos relacionales. Lo anterior, facilita la comunicación entre una aplicación escrita en Java y una base de datos relacional, más allá de simplemente usar JDBC.

Hibernate es un *framework* para *Object/Relational Mapping* (ORM) para el lenguaje Java, siendo tal vez la opción más popular con ese propósito. Hibernate se encarga del mapeo entre clases de Java y tablas de SQL, así como del mapeo entre tipos de datos de Java y tipos de datos de SQL. Dado lo anterior, este *framework* busca una interacción más simple y natural entre una aplicación escrita en Java y una base de datos relacional, evitando la escritura manual de decenas de líneas de código, tanto de JDBC como de SQL.



Otra ventaja que ofrecen los *frameworks* para ORM en general, e Hibernate en particular, es la posibilidad de cambiar de SGBD en medio del desarrollo de una aplicación de software, sin que esto implique cambios sustanciales en el código de la aplicación o en su lógica de negocio. Un cambio de SGBD (por ejemplo, de MySQL a Oracle, o viceversa) en medio del proceso de desarrollo puede llevar a uno de los dos siguientes escenarios:


- Si no se está usando una herramienta de ORM, requiere el cambio de las sentencias SQL al lenguaje propio del SGBD y, por lo tanto, cambios sustanciales a la capa de persistencia de la aplicación. A veces, lo anterior implicará también cambios en la lógica de negocio.
- Si sí se está usando una herramienta de ORM, requiere cambios en la configuración en la misma, y cambios menores en la capa de persistencia de la aplicación.

Lo anterior es posible debido a que la mayor parte de los *frameworks* para ORM cuentan con un lenguaje propio similar a SQL (en el caso de Hibernate, HQL), que permite hacer consultas a una base de datos siempre de la misma manera, independientemente de su SGBD. La arquitectura de una aplicación escrita en Java, usando un ORM como Hibernate, se puede apreciar a continuación:



El primer elemento de la figura representa el código en sí escrito por el desarrollador. Este incluye la interfaz gráfica, la lógica de negocio y la capa de persistencia, como sea que la hayamos diseñado. El segundo elemento, representa es la API del ORM (clases y funciones) que usamos para interactuar con la base de datos. El tercer elemento, es el motor que incluye Hibernate o cualquier otro ORM, encargado de hacer el mapeo entre el modelo objetual de la aplicación, y el modelo relacional requerido por la base de datos. Internamente, Hibernate se comunica con la base





de datos usando JDBC, el cual está representado por el cuarto elemento de la figura. Finalmente, el quinto elemento representa la base de datos en sí, alojada en un servidor correspondiente a un SGBD específico (p.e., MySQL, Oracle o MS SQL Server).

A continuación, vamos a explorar un ejemplo sencillo de cómo podemos configurar y escribir una aplicación en Java que se comuniquen con una base de datos MySQL usando Hibernate como *framework* ORM<sup>1</sup>.

## Configuración General del Proyecto

Como con otros *frameworks*, con Hibernate es necesario realizar algunas configuraciones en el proyecto de desarrollo, de tal manera que este funcione correctamente y que el *framework* cuente con la información que requiere. Lo primero a hacer, es incluir las dependencias necesarias en el proyecto. Es decir, las librerías y paquetes necesarios para que este funcione. En nuestro caso, se requiere el *driver* del SGBD que vayamos a utilizar, y la dependencia correspondiente a Hibernate. Como en el caso de JDBC, ambas dependencias se pueden incluir de dos formas:

- Una forma es descargar e importar el *driver* de MySQL y la dependencia de Hibernate dentro del proyecto en el IDE. Esta es la primera elección cuando no se trata de un proyecto con *Maven* o *Gradle*. Lo más conveniente es descargar el archivo .JAR correspondiente al *driver* e importarlo al proyecto dentro del IDE en la sección de librerías o de dependencias. En el caso de MySQL, el *driver* lo podemos descargar aquí: <https://dev.mysql.com/downloads/connector/j/>. En el caso de Hibernate, lo podemos descargar aquí: <https://hibernate.org/orm/releases/5.6/>
- La forma recomendada, si estamos trabajando en un proyecto con *Maven*, es agregar la dependencia

---

<sup>1</sup>El ejemplo completo se puede encontrar aquí: [https://github.com/leonjaramillo/udea\\_ruta2\\_ciclo2](https://github.com/leonjaramillo/udea_ruta2_ciclo2)





de MySQL en el archivo *pom.xml* del proyecto<sup>2</sup>. Para MySQL, la dependencia se puede encontrar aquí: <https://mvnrepository.com/artifact/mysql/mysql-connector-java/8.0.26>. En el caso de Hibernate, la podemos encontrar aquí: <https://mvnrepository.com/artifact/org.hibernate/hibernate-core/5.6.8.Final>.

En nuestro ejemplo, vamos a usar Maven para la gestión de dependencias. Tal como lo mencionamos arriba, las incluiremos en el archivo *pom.xml* así:

```
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.26</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.6.8.Final</version>
</dependency>
```

## Configuración de Hibernate en el Proyecto

Una vez hemos configurado las dependencias necesarias para el proyecto es necesario, en el caso de Hibernate, construir el archivo de configuración del proyecto para el *framework*. Se trata del archivo *hibernate.cfg.xml*. Este se debe guardar en la carpeta *src/main/resources* del proyecto. Si no existe, debemos crearla.

El archivo *hibernate.cfg.xml* es el archivo de configuración estándar de Hibernate. Se escribe en lenguaje XML y cuenta con todos los parámetros necesarios para el

---

<sup>2</sup>El archivo *pom.xml* respectivo, se puede encontrar aquí: [https://github.com/leonjaramillo/udea\\_ruta2\\_ciclo2/blob/main/pom.xml](https://github.com/leonjaramillo/udea_ruta2_ciclo2/blob/main/pom.xml)

funcionamiento del *framework*. Ante lo complejo que este puede llegar a ser, la recomendación es tomar como base un archivo ofrecido en la documentación y modificarlo de acuerdo a nuestras necesidades. A continuación, tenemos un archivo de ejemplo para el proyecto de ejemplo del presente documento<sup>3</sup>.

```
<?xml version='1.0' encoding='utf-8'?>
<!--
  ~ Hibernate, Relational Persistence for Idiomatic Java
  ~
  ~ License: GNU Lesser General Public License (LGPL), version
  2.1 or later.
  ~ See the lgpl.txt file in the root directory or <http://www.
  gnu.org/licenses/lgpl-2.1.html>.
  -->
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configura-
    tion-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <!-- Database connection settings -->
        <property name="connection.driver_class">com.mysql.
jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://local-
host:3306/concesionario</property>
        <property name="connection.username">root</property>
        <property name="connection.password">rootroot</proper-
ty>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.
MySQL5Dialect</property>
```

<sup>3</sup>El archivo de configuración referido, se puede encontrar aquí: [https://github.com/leonjaramillo/udea\\_ruta2\\_ciclo2/blob/main/main/resources/hibernate.cfg.xml](https://github.com/leonjaramillo/udea_ruta2_ciclo2/blob/main/main/resources/hibernate.cfg.xml)

```

    <!-- Disable the second-level cache -->
    <property name="cache.provider_class">org.hibernate.
cache.internal.NoCacheProvider</property>

    <!-- Echo all executed SQL to stdout -->
    <property name="show_sql">true</property>

    <!-- Drop and re-create the database schema on startup
-->
    <property name="hbm2ddl.auto">create</property>

    <!-- Names the annotated entity class -->
    <mapping class="co.edu.udea.udea_ruta2_ciclo2.hiber-
nate.Vendedor"/>

    </session-factory>

</hibernate-configuration>

```

En el archivo anterior se definen un conjunto de propiedades a tener en cuenta para la conexión entre nuestra aplicación y la base de datos. Cabe señalar que para que dicha conexión sea exitosa, el servidor debe haber sido instalado previamente y estar ejecutándose. Además, se debe haber creado la base de datos o esquema donde se va a guardar la información. Específicamente, se define lo siguiente:

- El *driver*, es decir, la clase que se usará para la conexión con la base de datos. En este caso es `com.mysql.jdbc.Driver`. Esta es la misma clase que se usa para las conexiones con JDBC y varía de acuerdo con el SGBD que hay detrás (cada SGBD tiene su propio *driver*).
- La URL de la base de datos a la cual se va a acceder. Esta URL incluye el protocolo de conexión (JDBC con MySQL), la dirección del servidor y el puerto por medio del cual se accede (*localhost* al tratarse del equipo local y 3306 siendo el puerto por defecto de MySQL), y finalmente el nombre de la base de datos o del esquema donde se encuentran las tablas. La cadena completa es la siguiente: `jdbc:mysql://localhost:3306/concesionario`.
- El nombre de usuario con el que se accede al servidor (en este caso, `root`).
- La contraseña con la que se accede al servidor (en este



caso, rootroot).

- Hibernate puede conectarse con diferentes tipos de bases de datos y SGBD. Cada uno de ellos cuenta con un dialecto diferente de SQL. Ya que, si bien SQL es un lenguaje estándar, tiene diferencias sutiles entre cada SGBD. En el presente archivo se define, entonces el dialecto a usar (en nuestro caso, el correspondiente a MySQL, es decir `org.hibernate.dialect.MySQL5Dialect`).

- Una propiedad que establece que se impriman en la consola las acciones realizadas por Hibernate, entre otras cosas. Para este caso sí se están mostrando las acciones, y por eso su valor es `true`. En caso contrario, se debería de poner en `false` (la propiedad es `show_sql`).

- Una propiedad (`hbm2ddl.auto`) con la cual especificamos qué sucede con la base de datos o esquema dentro del servidor al momento de correr el proyecto. Al asignarle el valor `create`, estamos especificando que el contenido de la base de datos se borra y crea dentro del servidor MySQL cada vez que corremos el proyecto. También se pueden asignar otros valores. Por ejemplo, si le asignáramos `update`, dicho contenido no se volvería a crear al correr el proyecto de manera sucesiva, sino que se trabajaría sobre las tablas existentes, y de ser el caso, se actualizarían.

- Finalmente, se especifican la o las clases que se mapearán para su persistencia en la base de datos, para el caso del presente proyecto sólo se mapeará una, y será `co.edu.udea.udea_ruta2_ciclo2.hibernate.Vendedor`.

## La Entidad o Clase a Mapear

A la hora de construir una aplicación en Java es necesario gestionar, es decir, crear, modificar y almacenar, la información que usa la misma. Por ejemplo, en una aplicación de inventario, es necesario guardar información sobre productos, existencias, movimientos, entre otras cosas.

Para lograr lo anterior se usan clases, y a la postre objetos, que almacenen dicha información en sus atributos. Una clase tipo entidad representa un registro de información dentro de una tabla en una base de datos. Por lo general, cuenta con atributos, cada uno de los cuales representa un





campo dentro de dicha tabla (a menos que se indique lo contrario). Los atributos mencionados suelen ser privados, con sus respectivos *getters* y *setters*. Hibernate permite luego guardar dicha información en una base de datos, así como recuperarla de ella y manipularla. Con el anterior fin, se agregan las anotaciones necesarias a la clase que queramos mapear como tabla en una base de datos.

A continuación, podemos observar la clase correspondiente a los vendedores dentro de una hipotética aplicación de ventas. Las anotaciones se pueden distinguir porque tienen una arroba (@) al principio, y pueden o no tener parámetros. Notemos que se importan tal como se hace con las clases<sup>4</sup>.

```
import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="vendedores")
public class Vendedor implements Serializable {
    @Id
    private int documento;
    @Column(nullable=false)
    private String nombres;
    @Column(nullable=false)
    private String apellidos;
    private String ciudad;

    public Vendedor() {
    }

    public Vendedor(int documento, String nombres, String apellidos, String ciudad) {
        this.documento = documento;
        this.nombres = nombres;
        this.apellidos = apellidos;
        this.ciudad = ciudad;
    }
}
```

---

<sup>4</sup>La clase referida se puede encontrar aquí: [https://github.com/leonjaramillo/udea\\_ruta2\\_ciclo2/tree/main/main/java/co/edu/udea/udea\\_ruta2\\_ciclo2/hibernate](https://github.com/leonjaramillo/udea_ruta2_ciclo2/tree/main/main/java/co/edu/udea/udea_ruta2_ciclo2/hibernate)

```

public int getDocumento() {
    return documento;
}

public void setDocumento(int documento) {
    this.documento = documento;
}

public String getNombres() {
    return nombres;
}

public void setNombres(String nombres) {
    this.nombres = nombres;
}

public String getApellidos() {
    return apellidos;
}

public void setApellidos(String apellidos) {
}

public String getCiudad() {
    return ciudad;
}

public void setCiudad(String ciudad) {
    this.ciudad = ciudad;
}

@Override
public String toString() {
    return "Vendedor{" + "documento=" + documento + ", nom-
bres=" + nombres + ", apellidos=" + apellidos + ", ciudad=" +
ciudad + `}`;
}
}

```

Si observamos detalladamente, podremos notar lo siguiente:

- Para luego poder imprimir más fácilmente el contenido de un objeto de dicha clase, agregamos un método `toString()`.
- Al declararla, especificamos que la clase implementa la interfaz `Serializable`. Esta es una recomendación para las clases a persistir usando Hibernate.
- La anotación `@Entity` se aplica a la clase. Con esto,



indicamos que dicha clase se mapeará como una tabla en una base de datos relacional. Esto también implica que los atributos de la clase se mapearán como campos de la tabla respectiva (de nuevo, a menos que se indique lo contrario para un atributo o atributos específicos).

- La anotación `@Table` es opcional, y en este caso cuenta con el parámetro `name`. Este parámetro indica el nombre que tendrá la tabla en la base de datos. Si este parámetro no se especifica, la tabla tomará el nombre exacto de la clase.
- La anotación `@Id` se aplica al atributo `documento`, e indica que dicho atributo corresponderá al campo de la tabla del mismo nombre, y que será su clave primaria.
- La anotación `@Column` permite indicar diferentes propiedades sobre los atributos a los cuales se aplica. En el caso del código anterior, indica que los campos correspondientes a los atributos `nombres` y `apellidos` no podrán ser nulos.

Además de las anotaciones anteriores, existen muchas más que permiten especificar diferentes propiedades para las tablas y campos correspondientes a una clase determinada. Con estas anotaciones también es posible especificar relaciones entre dos tablas mediante claves foráneas. Todo lo anterior se puede explorar en la documentación de Hibernate y de la *Java Persistence API* (JPA).

## Interacción con la Base de Datos

Una vez se ha configurado el proyecto y se han creado y anotado las clases que se mapearán en la base de datos mediante tablas, podemos pasar a interactuar con la base de datos cuando y como sea necesario<sup>5</sup>. Lo anterior, se puede llevar a cabo siguiendo ciertos procedimientos, patrones de diseño y buenas prácticas, las cuales están fuera del alcance de la presente lectura. Sin embargo, en el siguiente ejemplo se ilustra dicha interacción. La finalidad del código mostrado a continuación es conectarse a la base de datos donde se guardarán los datos de los vendedores de un concesionario hipotético. Una vez se hace esto, se crean, listan, editan y eliminan registros. Todo esto es posible de verificar en una base de datos o esquema con el nombre definido en el archivo de configuración, a su vez alojado en un servidor con los datos también ahí referidos.

---

<sup>5</sup>La clase referida se puede encontrar aquí: [https://github.com/leonjaramillo/udea\\_ruta2\\_ciclo2/tree/main/main/java/co/edu/udea/udea\\_ruta2\\_ciclo2/hibernate](https://github.com/leonjaramillo/udea_ruta2_ciclo2/tree/main/main/java/co/edu/udea/udea_ruta2_ciclo2/hibernate)

```
import java.util.List;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.query.Query;

public class EjemploConcesionario {

    public static void main(String[] args) {
        StandardServiceRegistry registry = new StandardServiceRegistryBuilder()
            .configure()
            .build();
        SessionFactory factory = new MetadataSources(registry).buildMetadata().buildSessionFactory();

        crearRegistros(factory);
        listarRegistros(factory);
        editarRegistros(factory);
        listarRegistros(factory);
        borrarRegistros(factory);
        listarRegistros(factory);
    }

    public static void crearRegistros(SessionFactory factory) {
        try (Session session = factory.openSession()) {
            Transaction transaccion = session.beginTransaction();

            Vendedor v1 = new Vendedor(100, "Juan", "Gómez", "Medellín");
            Vendedor v2 = new Vendedor(200, "José", "Rodríguez", "Bogotá");
            Vendedor v3 = new Vendedor(300, "Luis", "Giraldo", "Cartagena");
            session.save(v1);
            session.save(v2);
            session.save(v3);
            transaccion.commit();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    public static void listarRegistros(SessionFactory factory) {
        try (Session session = factory.openSession()) {
            Transaction transaccion = session.beginTransaction();
```



```

tion();

        Query consulta = sesion.createQuery("from Vende-
dor");


        List<Vendedor> vendedores = consulta.list();
        for (Vendedor v : vendedores) {
            System.out.println(v);
        }
        transaccion.commit();
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}

public static void editarRegistros(SessionFactory factory)
{
    try (Session sesion = factory.openSession()) {
        Transaction transaccion = sesion.beginTransaction();
        Vendedor v = sesion.get(Vendedor.class, 100);
        v.setApellidos("Gómez Ramírez");
        v.setCiudad("Medellín, Antioquia");
        sesion.update(v);
        transaccion.commit();
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}

public static void borrarRegistros(SessionFactory factory)
{
    try (Session sesion = factory.openSession()) {
        Transaction transaccion = sesion.beginTransaction();
        Vendedor v = sesion.get(Vendedor.class, 300);
        sesion.delete(v);
        transaccion.commit();
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
}

```

En el código anterior, se realizan múltiples acciones. En primer lugar, en el método *main*, creamos y construimos un registro de servicios. Esto se hace al declarar, construir y configurar el objeto de la clase *StandardServiceRegistry*. Este *registro de servicios* permite gestionar servicios (desde el punto de vista del software) en la aplicación de una manera ordenada. Dentro de esa línea de código, al llamar el método *configure()*, se integran a la aplicación las propiedades definidas previamente en el archivo *hibernate.cfg.xml*.



Luego, declaramos y construimos una *session factory*, pasándole como argumento el registro de servicios creado en el paso anterior. Esto, se hace en la línea donde se declara e inicializa el objeto de tipo `SessionFactory`. Una *session factory* es, como su nombre en inglés lo indica, una fábrica de sesiones, y nos permitirá crearlas luego. Por su parte, una sesión es un objeto que necesitamos crear cada vez que vamos a interactuar con la base de datos.

Después, todavía en el método *main*, invocamos los métodos `crearRegistros()`, `listarRegistros()`, `editarRegistros()` y `borrarRegistros()`. A todos ellos se les pasa como parámetro la *session factory* creada previamente, ya que en todos los casos se necesita para crear las sesiones, realizando las tareas que se intuyen en el nombre de cada método.

En el método `crearRegistros()`, todo se enmarca en una sentencia `try-with-resources`. En primer lugar, se crea una sesión (que es un objeto de la clase `Session`) usando la *session factory* creada anteriormente. Una sesión representa una interacción entre nuestra aplicación y la base de datos, y es donde se genera la conexión con la misma, de manera similar a JDBC.

- Luego, se crea e inicia una transacción, que se implementa mediante un objeto de la clase `Transaction`. Una transacción representa un conjunto de tareas que se realizan como una sola, es decir, o se ejecutan todas de manera exitosa, o no se ejecuta ninguna. En este caso, la transacción comprende las siguientes instrucciones dentro del método hasta que se invoca el método `commit()`.
- A continuación, se crean tres objetos de la clase `Vendedor` y se inicializan. Recordemos que esta clase tendrá una tabla equivalente en la base de datos llamada `vendedores`.
- Luego, se invoca el método `save()` del objeto `sesion`, pasando como argumento cada uno de los objetos de la clase `Vendedor`. Lo que se hace en este punto es guardar los datos de cada objeto en la base de datos.

En el método `listarRegistros()`, de nuevo, todo se enmarca en una sentencia `try-with-resources`. En primer lugar, se crean la sesión y transacción respectivas.

- Luego, se crea una consulta, que es un objeto de la



clase `Query`. Si nos fijamos bien, notaremos que a la consulta se le pasa como parámetro la cadena de caracteres `"from Vendedor"`. Esta está escrita en el lenguaje *Hibernate Query Language* (HQL) y es el equivalente de SQL dentro de Hibernate. Este lenguaje nos permite hacer, de la misma manera, consultas de todo tipo a todos los SGBD soportados por el *framework*. Específicamente en este caso, `"from Vendedor"` es el equivalente a una consulta `SELECT` en lenguaje SQL. Esta consulta, "traerá" de la base de datos todos los registros de la tabla referida y los guardará en una lista usando el método `list()`.

- Después, iteramos sobre la lista a la cual se asignaron los resultados de la consulta, mostrándolos, a su vez, en pantalla.

En el método `editarRegistros()`, también, todo se enmarca en una sentencia `try-with-resources`.

- Primero, se crean la sesión y la transacción respectivas. Luego, mediante el método `get()` del objeto sesión "traemos" de la base de datos el vendedor con número de documento `100`. Es así como el objeto previamente creado (`v`) de la clase `Vendedor` tendrá en sus atributos los valores que se encuentren en la base de datos en los campos del registro consultado.
- Luego, modificamos en el objeto `v` los atributos que necesitemos cambiar. Y finalmente, se guardan de vuelta en la base de datos los nuevos valores mediante el método `update()`.

En el método `borrarRegistros()`, todo se enmarca en una sentencia `try-with-resources`.

- Primero, se crean la sesión y la transacción respectivas. Luego, mediante el método `get()` del objeto sesión "traemos" de la base de datos el vendedor con número de documento `300`. Es así como el objeto previamente creado (`v`) de la clase `Vendedor` tendrá en sus atributos los valores que se encuentren en la base de datos en los campos del registro consultado, representando dicho registro en nuestra aplicación.
- Luego, pasamos el objeto `v` como parámetro al método `delete()` de la sesión. Esto hará que mediante Hibernate se elimine el registro correspondiente a dicho vendedor en la base de datos.

Como pudimos observar, la mayor parte de las interacciones de nuestras aplicaciones con una base de datos usando Hibernate siguen un patrón de pasos similar.





## Más Información

Encontrar en internet documentación de abundante y de calidad acerca de Hibernate puede ser desafiante, dada su cantidad, y dadas las versiones existentes del *framework*. Además, la mayor parte de esta se encuentra en inglés. Una regla básica que aplicar es que la documentación se refiera a las versiones 5 de Hibernate en adelante. Un ejemplo adecuado sería la documentación oficial del lenguaje (en inglés): <https://hibernate.org/orm/documentation/5.6/>.

