Hibernate.

1.Requisitos antes de comenzar

Requisitos para poder utilizar Hibernate

- Instalar SGBD. Nosotros instalaremos MySQL.
- Necesitaremos usar un cliente para acceder a la BBDD, podemos elegir entre las siguientes:
    - MySql Workbench
    - PHPMyAdmin
- Instalar driver JDBC (Descargar- necesitamos el bin.jar o por maven)
- Instalar liberías Hibernate (Descargar o por maven)
    - Mínimo Java 8

1.1. Instalación MySQL

En **Windows**, podemos seguir la siguiente guía. Al instalar MySQL, nos da la opción de instalar la versión de “Developer” que ya instala el Mysql Workbench.

En **Ubuntu**, podéis seguir las siguientes páginas de guía:

- Instalar Mysql
- Instalar Mysql Workbench

1.2 Para los drivers JDBC y las librerías de Hibernate.

Las enlazamos desde Maven, añadiendo las dependencias en el pom.xml

2. Preparamos la BBDD sobre la que trabajar.

2.1. Preparando esquema.

Crear un espacio en la base de datos donde podremos crear tablas e insertar datos. En MySQL, esto se llama *esquema (schema)*.

Para crear un nuevo esquema desde MySQL Workbench tan solo tenemos que pulsar con el botón derecho del ratón sobre el panel *Navigator*, y seleccionar la opción *Create Schema*. A continuación, indicamos el nombre (por ejemplo, `AD_hibernate`), y pulsamos sobre el botón Apply.

Nosotros haremos uso de este nuevo esquema con un usuario que hemos creado durante la instalación de MySQL que se llama `ADHibernate` y cuya contraseña es `abc123`.

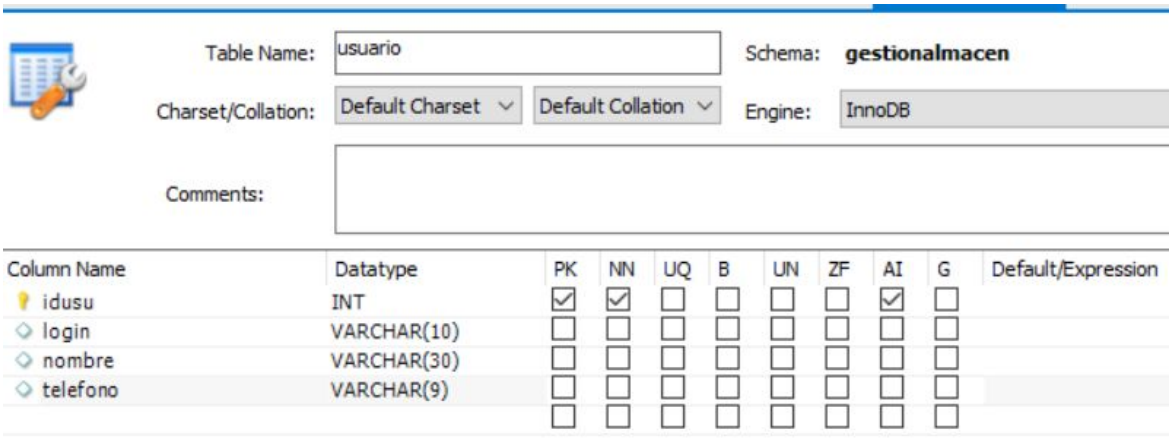
```
CREATE USER 'ADHibernate'@'localhost' IDENTIFIED BY 'abc123.';

GRANT ALL PRIVILEGES ON * . * TO 'ADHibernate'@'localhost';
```

2.2. Creando tablas

Crear la tabla usuario, con los siguientes campos (idusu, login, nombre, telefono).

Pondremos el campo idusu como:

- Primary Key,
- Not null
- Auto_Increment, al ser auto_increment, al añadir campos en esta tabla, no le daremos el idusu, este lo genera automáticamente el gestor de BBDD.




```
CREATE TABLE `gestionalmacen`.`usuario` (
  `idusu` INT NOT NULL AUTO_INCREMENT,
  `login` VARCHAR(10) NULL,
  `nombre` VARCHAR(30) NULL,
  `telefono` VARCHAR(9) NULL,
  PRIMARY KEY (`idusu`));
```

3. Configuración genérica del proyecto

3.1 Creación y configuración inicial

Vamos a comenzar creando nuestro proyecto Maven con un JDK igual o superior a 1.8

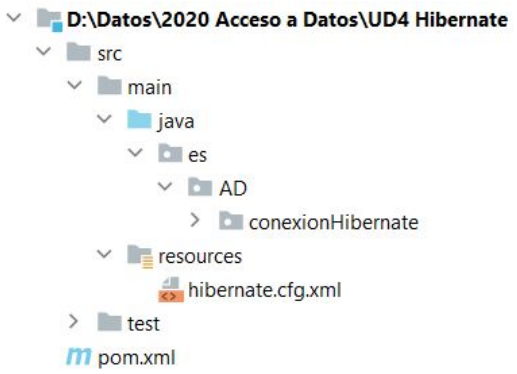
3.2 Configuración del fichero pom.xml

Ahora vamos a añadir las dependencias necesarias al fichero `pom.xml`, serán las dependencias con hibernate y con JDBC de mysql. Quedarán de la siguiente forma:

```
<dependencies>
   <!-- Otras dependencias que podamos necesitar ... -->
   <dependency>

      <!-- Hibernate ... -->

       <groupId>org.hibernate</groupId>
       <artifactId>hibernate-core</artifactId>
       <version>5.4.27.Final</version>
   </dependency>

   <dependency>

      <!-- MYSQL... -->

       <groupId>mysql</groupId>
       <artifactId>mysql-connector-java</artifactId>
       <version>8.0.18</version>
   </dependency>
</dependencies>
```

3.3. Creación del fichero de configuración de Hibernate (hibernate.cfg.xml).

1. Para ello, crearemos una carpeta de recursos o ya la tendrá creada el proyecto. La carpeta de recursos estará en la ruta : `/src/main/resources`. Para ello, según el IDE empleado, se podrá hacer pulsando sobre el proyecto, botón derecho y *New* > *Source Folder*.



2. Añadir fichero de recursos cuyo nombre será **hibernate.cfg.xml** y su contenido será el que configure la conexión a la BBDD que usarán las librerías de Hibernate. Copiar el siguiente código en el fichero:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN" "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
```

```
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="connection.url">jdbc:mysql://localhost:3306/AD_Hibernate</property>
    <property name="connection.username">root</property>
    <property name="connection.password">abc123.</property>
    <property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>
    <property name="hibernate.show_sql">true</property>
  </session-factory>
</hibernate-configuration>
```

3. Personalizar el driver, la url, username y password, en función de nuestro gestor de BBDD y la BBDD y usuario que empleemos para nuestro proyecto.

3.4 Definición de nuestras clases modelo.

1. Crearemos una clase, que se corresponda con la estructura de la tabla. Con una propiedad por cada campo de la tabla con el que vayamos a trabajar.

```
public class Usuario implements Serializable {
	private int idusu;
	private String login;
	private String nombre;
	private String telefono;
}
```

2. Esta clase implementará `Serializable`
3. Definiremos los siguientes métodos:
   - getter and setter de todos los campos
   - constructor vacío
   - constructor con todos los campos, menos el Id (al ser autogenerado no se lo vamos a dar)
   - toString con todos los campos.

4. Añadimos las anotaciones:
   - `@Entity(name="nombreTabla")` indica que esta clase es una *entidad* que deberá ser gestionada por el motor de persistencia de Hibernate.
   - `@Id` indica que, de todos los atributos, ese será tratado como clave primaria.
   - `@Column(name="nombreColumna")` indica que ese atributo es una columna de la tabla resultante.

   @Entity(name="nombreTabla") y @Column(name="nombreCampoenBBDD") tienen que tener los mismos nombre que la tabla y los campos de la BBDD.

5. Al añadir las anotaciones, tenemos que añadir: `import javax.persistence.*;`

La clase finalmente quedaría como:

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Usuario implements Serializable {
  @Column
  @Id
  private int idusu;
  @Column
```

```
    private String login;
    @Column
    private String nombre;
    @Column
    private String telefono;


    /*
     GETTER Y SETTER de todas las propiedades
     */
    public int getIdusu() {
        return idusu;
    }

    public void setIdusu(int idusu) {
        this.idusu = idusu;
    }

    public String getLogin() {
        return login;
    }

    public void setLogin(String login) {
        this.login = login;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getTelefono() {
        return telefono;
    }

    public void setTelefono(String telefono) {
        this.telefono = telefono;
    }

    /*
    CONSTRUCTORES  (sin el id)
     */
    public Usuario() {
    }

    public Usuario(String login, String nombre, String telefono) {
        this.login = login;
        this.nombre = nombre;
        this.telefono = telefono;
    }

    /*
    toString  para obtener los datos en formato string
     */
    @Override
    public String toString() {
        return "Usuario{" +
                "idusu=" + idusu +
                ", login='" + login + '\'' +
                ", nombre='" + nombre + '\'' +
                ", telefono='" + telefono + '\'' +
                '}';
    }
}
```

4. Desarrollando nuestro proyecto

4.1 Cargar la configuración de hibernate (SessionFactory)

Implementar el código necesario para cargar la configuración del fichero `hibernate.cfg.xml`.

Tiene dos pasos:

- Lee el archivo de configuración.

```
SessionFactory misf= new Configuration().configure().buildSessionFactory();
```

- Crea objetos de tipo Session sobre los que se trabajará para conseguir los objetivos del proyecto.

```
Session misession = misf.openSession();
```

Estos trabajarán sobre los paquetes org.hibernate.

Al finalizar se deben de cerrar tanto la session como la SessionFactory:

```
misession.close();
misf.close();
```

4.2 Operaciones CRUD básicas.

Para realizar cualquier operación sobre la BBDD tiene que estar entre una transacción:

```
session.beginTransaction();
//operación sobre la BBDD...
session.getTransaction().commit() o session.getTransaction().rollback();
```

Las operaciones de CRUD son:

- Create: crea objetos
- Read: lee los objetos
- Update: modifica los objetos
- Delete: Borra los objetos

Añadir un registro (Session.save(objeto))

En nuestro caso, para añadir un usuario:

```
// Crear un objeto y asignarle valores
Usuario usu = new Usuario("login1","nombre1","111111111");

session.beginTransaction();
session.save(usu);
session.getTransaction().commit();
```

Leer un registro (Session.get (clase,clave))

Para poder realizar este get, necesitamos que la clase Usuario implemente Serializable

```
Usuario usuario=(Usuario)session.get(Usuario.class,1);
```

Actualizar un registro (Session.update(objeto))

Se actualizará el objeto que se envía como parámetro, buscando por la clave y actualizando el resto de valores

```
session.beginTransaction();
session.update(usu);
session.getTransaction().commit();
```

Borrar un registro (Session.delete(objeto))

```
session.beginTransaction();
session.delete(usu);
session.getTransaction().commit();
```

4.3 Consultas (createQuery)

Todas estas tareas realizadas anteriormente, puede necesitarse aplicar ciertos filtros, y añadirles condiciones, para eso ejecutaremos las consultas.

Leer registros

```
// Recupera los usuarios con nombre que empieza por nombre
session.beginTransaction();

List<Usuario> listaUsuarios =
session.createQuery("from Usuario where nombre like 'nombre%'").getResultList();

session.getTransaction().commit();

for (Usuario unUsuario:listaUsuarios){
   System.out.println(unUsuario.toString());
}
```

Eliminar un registro

Eliminar los usuarios con idusu = 3

```
session.beginTransaction();
session.createQuery("delete Usuario where idusu=3").executeUpdate();
session.getTransaction().commit();
```

Actualizar un registro

Actualiza los usuarios con ‘nombre3’ a ‘nombrecompleto3’

```
session.beginTransaction();
session.createQuery("update Usuario set nombre='nombrecompleto3' where nombre =
'nombre3'").executeUpdate();
session.getTransaction().commit();
```

4.6 HQL

La sentencia que va dentro de createQuery no es lenguaje SQL, sino HQL.

- Para indicar que empiece por ‘n*’ en HQL sería ‘n%’
- Los nombres de tablas y campos, no son los de la tabla, sino los de los objetos que van a persistir, por tanto son case sensitive.
- Los nombres en este ejemplo de la clase Usuario y sus propiedades podrían no coincidir con los de la tabla, indicando en la definición la correspondencia.

  Por ejemplo si en vez de idusu, le llamásemos en nuestra clase Usuario como id

  ```
  @Column(name=”idusu”)
    private int id;
  ```

  la consulta de borrado, debería de quedar como:

  ```
  session.createQuery("delete Usuario where id=3").executeUpdate();
  ```

5. Actualizaciones en cascada en varias tablas

5.1 Definiciones de clases y configuración para que esto funcione

Vamos a crear una tabla ahora llamada usuarioacceso, de manera, que registro todos los accesos al sistema de un usuario. Es decir, esta tabla tendrá relación n:1 con la de usuario.

```
CREATE TABLE `usuarioacceso` (
  `idAcceso` int NOT NULL AUTO_INCREMENT,
  `idUsuario` int DEFAULT NULL,
  `fechaAcceso` date DEFAULT NULL,
  PRIMARY KEY (`idAcceso`)
)
```

Para esto, los pasos a seguir serán:

- Creamos una clase 'UsuarioAcceso' con los campos de la tablas, getter, setter, constructores y método toString.
- **Clase Usuario:** Tenemos que modificar la definición de la clase "Usuario", para asociarlo a la tabla "UsuarioAcceso". Para esto:
    - En la clase "Usuario" añadimos una propiedad a UsuarioAcceso con sus getters y setters. Usuario está del lado de uno en la relación 1:n.

```
@Entity
public class Usuario implements Serializable {
    @Column
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int idusu;
    @Column
    private String login;
    @Column
    private String nombre;
    @Column
    private String telefono;

    @OneToMany(mappedBy="usuario",cascade=CascadeType.ALL)
    private Set<UsuarioAcceso> usuarioAcceso;
```

El id tendremos que indicar que es autogenerado, para que lo tenga disponible para añadir en cascada en la tabla UsuarioAcceso.

```
@GeneratedValue(strategy=GenerationType.IDENTITY)
```

En el lado 1, hay que indicarle mappedBy con el nombre de la tabla que referencia

```
@OneToMany(mappedBy="usuario",cascade=CascadeType.ALL)
    private Set<UsuarioAcceso> usuarioAcceso;
```

    - Lo añadimos al constructor, y una función para ir añadiendo los Accesos:

```
public Usuario(String login, String nombre, String telefono) {
   this.login = login;
   this.nombre = nombre;
   this.telefono = telefono;
   this.usuarioAcceso = new HashSet<>();
}

public void addUsuarioAcceso(UsuarioAcceso acceso){
   this.usuarioAcceso.add(acceso);
}
```

- **Clase UsuarioAcceso**: está del lado muchos, por lo tanto hay que añadir la referencia a la clase Usuario de la siguiente manera:

```
@ManyToOne
@JoinColumn(name="idUsuario")
private Usuario usuario;
```

@ManyToOne: Porque estamos del lado muchos
@JoinColumn: Indica el nombre de la columna por la que se va a enlazar en esta tabla.

- **SessionFactory** añadimos las dos clases

```
SessionFactory sf = new Configuration().configure("hibernate.cfg.xml")
                    .addAnnotatedClass(Usuario.class)
                    .addAnnotatedClass(UsuarioAcceso.class)
                    .buildSessionFactory();
```

5.2. Funcionamiento

El comportamiento esperado ahora es que:

- Cuando grabamos un usuario, al que enlazamos un acceso, se grabará este acceso de usuario. Concretando a nuestro ejemplo, tenemos un session.save(usu) que grabará en la tabla “usuario” y en la tabla “usuarioacceso”.
- Cuando recuperamos un usuario, recupera también los accesos del usuario
- Cuando borramos un usuario, también se borra su acceso asociado.

El código queda como se muestra a continuación.

```
//Creamos un usuario
Usuario usu = new Usuario("usu5","nombre5","123456789");

//Creamos un accesos para el usuario
UsuarioAcceso acceso = new UsuarioAcceso( Date.valueOf(LocalDate.of(2021, 2, 13)),usu);
usu.addUsuarioAcceso(acceso);

//Guardamos el usuario y se guarda también los accesos del objeto usu
session.beginTransaction();
session.save(usu);
session.getTransaction().commit();


// Recuperamos un usuario
```

```
Usuario usuario = (Usuario) session.get(Usuario.class,36);
System.out.println(usuario.toString());
System.out.println(usuario.getUsuarioAcceso().toString());

// Borramos el usuario
session.beginTransaction();
session.delete(usuario);
session.getTransaction().commit();
```