Unidad 13

Mantenimiento de la persistencia de los datos

Programación

Índice

Introducción

Con el uso de sistema de ficheros nos permite gestionar una cantidad limitada de datos, cuando disponemos de una gran cantidad de datos los sistemas de ficheros no es suficiente y necesitamos una herramienta más potente, las bases de datos.

Las bases de datos permiten gestionar gran cantidad de datos ofreciendo mejores resultados y facilidad para su manejo. A su vez una base de datos se puede orientar a objetos como representación al sistema de clases en java.

Al finalizar esta unidad

- Conoceremos las base de datos orientadas a objetos
- Aprenderemos como conectarse a una base de datos
- Manejar el lenguaje JPQL
- Conocer cómo realizar consultas, inserciones, borrado, etc.

13.1. Base de datos orientada a objetos

Una base de datos es un sistema organizado de datos que permite almacenar o consultar información en cualquier instante independientemente del volumen de datos. Las base de datos más usadas son las base de datos relacionales que se ven en la asignatura de base de datos. Ahora se verá otro tipo de base de datos orientada a objetos (BDOO o OODB).

Las base de datos orientadas a objetos se diferencia de las demás por ser NoSQL, aplicar el paradigma orientado a objeto y representar la información en forma de objetos.

13.2. Características de las bases de datos orientadas a objetos

A continuación, se describen las principales características de las bases de datos orientadas a objetos:

- El acceso es más rápido, ya que es posible recuperar datos directamente sin realizar una búsqueda.
- Se usa los mismos tipos de datos que el lenguaje de programación.
- Se incluye algún tipo de lenguaje para realizar consultas.
- Trabajan conjuntamente con los lenguajes de programación orientados a objetos.
- No necesitan definir tablas o convertir objetos en registros usando SQL.
- Son transparentes a su representación y permiten trabajar directamente.

13.3. Instalación del gestor de bases de datos

Existen diferentes gestores de base de datos orientada a objetos, Object Database ++, ObjectStore, GemStone/S, DB4O, Wakanda, ObjectDB. En este capítulo trabajaremos con ObjectDB.

ObjectDB está diseñado para trabajar con JAVA. Tiene una parte de licencia gratuita y limitada para hacer uso de toda su funcionalidad. Para pequeñas aplicaciones y aprender a hacer uso de base de datos orientado a objetos es idoneo.

PARA AMPLIAR CONOCIMIENTO...

Para saber más de ObjectDB dirigíos al siguiente enlace.
https://www.objectdb.com/

Para el inicio de la instalación, nos dirigiremos a la página web de ObjectDB y nos descargaremos la última versión comprimida en un zip. La descomprimimos en una ubicación disponible.

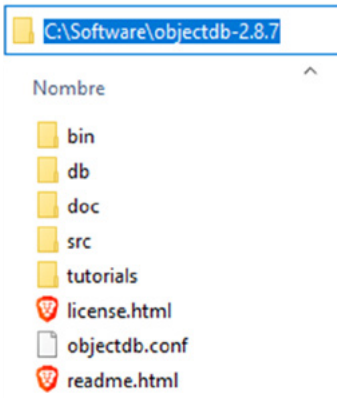

Imagen 1. Estructura de ficheros de instalación con ObjectDB

A continuación, se detalla los ficheros y directorios más importantes:

1. **objectdb.conf**

   Fichero de configuración de ObjectDB

2. **Carpeta bin**

   Programas esenciales para la funcionalidad del servidor, librerias, etc.

3. **Carpeta db**

   Contenedor de base de datos, por defecto ya contiene algunas base de datos, point.odb y world.odb.

13.4. Creación de bases de datos

Primero se tendrá que crear un proyecto Java donde se usará la base de datos. En el proyecto se añadira la librería objectdb.jar que se encuentra en la carpeta bin del directorio de instalación de ObjectDB. La librería dotará a nuestro proyecto toda la funcionalidad de ObjectDB.

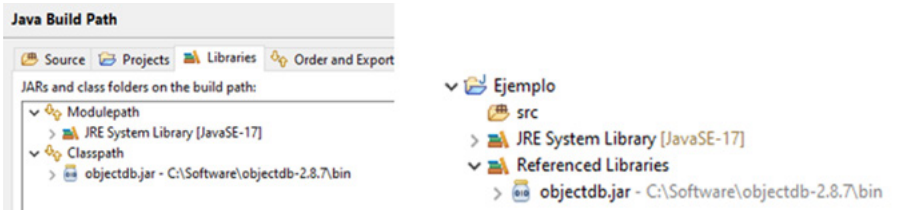

Imagen 2. Ejemplo de proyecto con la librería objectdb.jar

13.4.1. Establecimiento de la conexión

Para establecer la conexión primero hay que abrir la comunicación para ello necesitaremos crear dos objetos, el representante de la base de datos y el gestor de entidades.

1. El representante de la base de datos (**EntityManagerFactory**) abrira la base de datos que se le especifique por parametro. Se puede especificar la base de datos de la siguiente forma:
   - Usando una ruta absoluta donde se encuentra la base de datos.
   - Haciendo uso de la variable $objectdb. La variable toma de referencia el path donde se ubica la librería objectdb.jar
   - Indicando la URL del servidor

     (“objectdb://localhost/XXX.odb;user=usuario;password=contraseña”)

2. A partir del EntityManagerFactory se creará el gestor de entidades (EntityManager) y podrá realizar cualquier acción de creación, lectura, actualización y borrado sobre la base de datos.

```
String rutaBBDD = "$objectdb/db/contactos.odb";

// Abrir conexión
EntityManagerFactory emf = Persistence.createEntityManagerFactory(rutaBBDD);
EntityManager em = emf.createEntityManager();


// Cerrar conexión
em.close();
emf.close();
```

Imagen 3. Código para abrir y cerrar una conexión con la base de datos.

Para finalizar se tendrá que cerrar la conexión para ello primero cerraremos el EntityManager y despues el EntitymanagerFactory.

Si se indica una base de datos nueva, la primera vez que se ejecute la conexión creará la base de datos en la carpeta db de la instalación de ObjectDB.

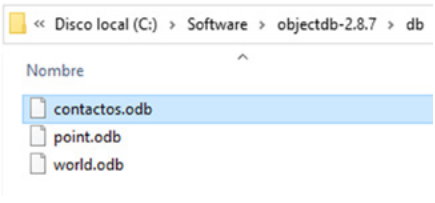

Imagen 4. Crear nueva base de datos

13.5. El lenguaje de definición de objetos

El lenguaje para definir objetos será JPA (Java Persistence API). JPA es una herramienta de persistencia que marca el estandar para manejar datos relacionales a objetos desde Java.

13.5.1. Tipos de datos

Los tipos de datos que se pueden definir en la base de datos son todos los tipos de datos que contempla el lenguaje de programación y que pueden aplicar la persistencia o serialización. Los tipos persistentes en JPA son:

- **Tipos primitivos:** int, float, byte, char, etc.
- **Tipos estructurados:** array y String. En array se debe cumplir que sus elementos pueden ser nulos o instancias de tipo persistente.
- **Tipos envoltorio:** Integer, Float, Double, etc.
- **Tipos Math:** BigDecimal, BigInteger.
- **Tipos fecha y hora.** Date, Time, Timestamp y Calendar. Se usará los tipos del paquete java.sql, si se usan los tipos de java.util habra que añadir la etiqueta @Temporal.
- **Tipos enumerados.** Se usará la etiqueta @Enumerated y especificando si su representación interna sera ordinal o por su nombre. @Enumerated(EnumType.ORDINAL) o @Enumerated(EnumType.STRING)
- **Tipos que implementan serializable.**
- **Cualquier colección o mapa.** Siempre que sus valores puedan ser nulos o instanacias de tipo persistente.
- **Cualquier clase definida por el usuario.** Haciendo uso de la etiqueta @Entity.

13.5.2. Clases

Las clases pueden ser almacenadas en una base de datos orientada a objetos siempre y cuando cumplan las siguientes condiciones:

- Tener un constructor sin argumentos.
- Los campos tienen que ser privados.
- Tener los métodos get y ser por cada campo.
- La clase y todos sus campos tienen que ser serializable.

La información que se guarda en la base de datos no es la estructura ni los métodos de la clase, si no, los valores que contiene los campos.

Para que la clase sea almacenada se tendrá que indicar la etiqueta @Entity o @Entity(name="Nombre") antes de la definición de la clase.

```
@Entity
public class Contacto implements Serializable {

    private static final long serialVersionUID = -1226598445347957691L;
```

Imagen 5. Ejemplo de clase usada como entidad de base de datos.

Si tenemos una clase que no queremos que se guarde directamente en la base de datos, si no que se incruste dentro de otra clase como campo, usaremos @Embeddable y @Embedded

```
@Embeddable
public class Direccion implements Serializable {

    private static final long serialVersionUID = -8894324235474321173L;

    private String direccion;
    private String codigoPostal;
    private String poblacion;
```

```
@Entity
public class Contacto implements Serializable {

    private static final long serialVersionUID = -1226598445347957691L;

    private String nombre;

    @Embedded
    private Direccion direccion;
```

Imagen 6. Ejemplo de clase incrustada como campo

13.5.3. Campos

Todos los campos de una clase se guardarán en la base de datos sin añadirles ninguna etiqueta, siempre y cuando sean del tipo descrito en listado del punto 13.5.1. A excepción de aquellos campos que se declaren como final, static o transient, o si lleva la etiqueta @Transient que no se guardarán en la base de datos.

```
@Entity
public class Campos implements Serializable {

    // Campos que se guardaran en base de datos
    private int campoEntero;
    private String campoCadena;
    private java.sql.Date campoFechaSQL;
    @Temporal(TemporalType.TIME)
    private java.util.Date campoDateUtil;
    @Enumerated(EnumType.STRING)
    private TipoEnumerado campoEnumerado;

    // Campos que no se guardaran en base de datos
    @Transient
    private String campoNoPersistente;
    private final int campoNoPersistente1 = 0;
    private static int campoNoPErsistente2;

}

enum TipoEnumerado {
    VALOR1, VALOR2, VALOR3
}
```

Imagen 7. Ejemplo de campos

Clave primaria

En cada clase se tiene que definir un campo como identificador único o clave primaria para poder referenciar a un objeto inequívocamente. La clave primaria se define añadiendo la etiqueta @Id en el campo concreto.

Una de las características de la clave primaria es que hay que definir como se establece sus valores, a continuación, se detallan varias formas:

- De **forma manual**, antes de almacenar en la base de datos, tendremos que añadir manualmente el valor de la clase primaria.
- Añadir la etiqueta **@GeneratedValue**. Genera un valor numerico automatico para cada base de datos. El valor es unico y no se reutiliza.
- Añadir la etiqueta **@GeneratedValue(strategy=GenerationType.AUTO)**. Igual que la anterior.
- Añadir la etiqueta **@Generated Value(strategy=GenerationType.IDENTITY)**. un generador de números único para cada jerarquía de tipos.
- Añadir la etiqueta **@Generated Value(strategy=GenerationType. SEQUENCE, generator= "seq")**. Genera una secuencia definida previamente usando la etiqueta SequenceGenerator(name="seq", initialValue=1, allocation-Size=100).

```
@Entity
public class Contacto implements Serializable {

    private static final long serialVersionUID = -1226598445347957691L;

    @Id
    @GeneratedValue
    private long id;

    private String nombre;
```

Imagen 8. Ejemplo de declarar y generar una clave primaria

Para claves primarias compuestas se tendra que usar una clase intermedia con los campos que forman la clave privaria y añadir la etiqueta @Embeddable. Luego se declara en la clase principal, como campo, la clase con las claves primarias con la etiqueta @EmbeddedId

```
@Embeddable
public class PK implements Serializable {

    private int clave1;
    private String clave2;

}
```

```
@Entity
public class Carrito implements Serializable {

    @EmbeddedId
    private PK identificador;

}
```

Imagen 9. Ejemplo de clave primaria compuesta

Métodos y propiedades

Una clase que se defina como entidad debe seguir las convenciones de los componentes JavaBeans.

Para los métodos JavaBeans define que cualquier campo de una clase debe ser privado y deben tener sus métodos get y set para poder consultar y establecer sus valores. La forma de definir los métodos get y set es usar el mismo nombre del campo con la primera letra mayuscula y por delante añadir get o set.

Si un campo no tiene bien definido sus métodos de get y set es posible que de errores al guardar los valores del campo en la base de datos.

13.5.4. Constructores

Cada clase que se indique como entidad a ser almacenada en una base de datos es necesario que tenga un constructor vacio y sin argumentos, con visibilidad public o protected y no debe sobreescribirse. Además, se podra añadir tantos constructores parametrizados como sea necesario.

13.5.5. Herencia

La herencia también se aplica a entidades. Como se indicó en apartados anteriores para que una clase sea persistente todos sus elementos deben ser persistentes, eso aplica a la relación de herencia. No hay etiquetas para definir la herencia, se usa @Entity o @Embeddable para cada clase.

Cuando se guarde la entidad en la base de datos, los campos de la clase y sus ancestros se guardarán para mantener la relación de herencia.

13.6. Mecanismos de consulta

Para realizar consultas a una base de datos JPA dispone de dos mecanismos Java Persistence Query Language (JPQL) y API Criteria.

- **JPQL** es un lenguaje parecido a SQL aportando facilidad y legibilidad a realizar cualquier consulta. Los inconvenientes de usar JPQL es que las consultas no son seguras y es necesario realizar conversiones de tipos de datos cuando se recuperan los datos.
- **API Criteria** usa el lenguaje de programación Java para realizar las consultas, son seguras, no requieren de conversion de tipo de datos y no es necesario aprender un lenguaje nuevo. El inconveniente de usar esta api es que no aporta facilidad ni legibilidad en las consultas.

En los siguientes puntos se basará en JPQL.

13.6.1. El lenguaje de consultas: sintaxis, expresiones, operadores

El lenguaje de consultas JPQL se basa en SQL, a continuación, se hace un breve repaso de las principales sentencias.

SELECT

Sentencia para realizar consultas en la base de datos. Sintaxis:

```
SELECT columna1, columna2, ...
FROM tabla
WHERE condicion;
```

INSERT

Sentencia para añadir datos en la base de datos. Sintaxis:

```
INSERT INTO tabla (columna1, columna2, columna3, ...)
VALUES (valor1, valor2, valor3, ...);
```

UPDATE

Sentencia para realizar actualizar datos en la base de datos. Sintaxis:

```
UPDATE tabla
SET columna1 = valor1, columna2 = valor2, ...
WHERE condicion;
```

DELETE

Sentencia para realizar borrar datos en la base de datos. Sintaxis:

```
DELETE FROM tabla
WHERE condición;
```

Para profundizar en las sentencias de SQL ir a la asignatura de base de datos.

13.6.2. Inserción, recuperación, modificación y borrado de información

En este apartado es necesario conocer como establecer la conexión a la base de datos y su cierre al finalizar las operaciones. Ver el apartado 13.4.1 Establecer conexión con la base de datos.

En los siguientes ejemplos que se muestran se usará una base de datos de contactos para almacer la información de un contacto (id, nombre, apellido1, apellido2, email y dirección).

```java
@Entity
public class Contacto implements Serializable {

    private static final long serialVersionUID = -1226598445347957691L;

    @Id @GeneratedValue private long id;
    private String nombre;
    @Embedded private Direccion direccion;
    private String apellido1;
    private String apellido2;
    private String email;

    public Contacto() { }

    public Contacto(String nombre, String apellido1, String apellido2, String email, Direccion direccion) {
        this.nombre = nombre;
        this.apellido1 = apellido1;
        this.apellido2 = apellido2;
        this.email = email;
        this.direccion = direccion;
    }

    public long getId() { return id; }

    public void setId(long id) { this.id = id; }

    public String getNombre() { return nombre; }

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

    public String getApellido1() { return apellido1; }

    public void setApellido1(String apellido1) { this.apellido1 = apellido1; }

    public String getApellido2() { return apellido2; }

    public void setApellido2(String apellido2) { this.apellido2 = apellido2; }

    public String getEmail() { return email; }

    public void setEmail(String email) { this.email = email; }

    @Override
    public String toString() {
        return String.format("(%d, %s, %s, %s, %s, %s)", this.id, this.nombre, this.apellido1, this.apellido2, this.email, this.direccion);
    }

}
```

Todos los Getters & Setters

Imagen 10. Entidad Contacto

```java
@Embeddable
public class Direccion implements Serializable {

    private static final long serialVersionUID = -8894324235474321173L;

    private String direccion;
    private String codigoPostal;
    private String poblacion;
    private String provincia;

    public Direccion() { }

    public Direccion(String direccion, String codigoPostal, String poblacion, String provincia) {
        this.direccion = direccion;
        this.codigoPostal = codigoPostal;
        this.poblacion = poblacion;
        this.provincia = provincia;
    }

    public String getDireccion() { return direccion; }

    public void setDireccion(String direccion) { this.direccion = direccion; }

    public String getCodigoPostal() { return codigoPostal; }

    public void setCodigoPostal(String codigoPostal) { this.codigoPostal = codigoPostal; }

    public String getPoblacion() { return poblacion; }

    public void setPoblacion(String poblacion) { this.poblacion = poblacion; }

    public String getProvincia() { return provincia; }

    public void setProvincia(String provincia) { this.provincia = provincia; }

    @Override
    public String toString() {
        return String.format("(%s, %s, %s, %s)", this.direccion, this.codigoPostal, this.poblacion, this.provincia);
    }

}
```

Imagen 11. Entidad Direccion

INSERCIÓN

La inserción o insert nos permite guardar elementos en la base de datos. Pasos para realizar una inserción en la base de datos:

1. Iniciar la transacción (begin)
2. Crear los objetos a guardar
3. Ejecutar el guardado (persist)
4. Finalizar la transacción (commit)

```java
String rutaBBDD = "$objectdb/db/contactos.odb";

// Abrir conexión
EntityManagerFactory emf = Persistence.createEntityManagerFactory(rutaBBDD);
EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

Direccion direccion = new Direccion("C/ Calle, 1","00822","Poblacion1","Provincia1");
Contacto contacto = new Contacto("Laura", "Carrillo", "Gonzalez", "laura@email.com",direccion);
Direccion direccion1 = new Direccion("Avd. avenida, 2","30123","Poblacion2","Provincia2");
Contacto contacto1 = new Contacto("Antonio","Baeza","Miró","antonio@email.com",direccion1);

em.persist(contacto);
em.persist(contacto1);

em.getTransaction().commit();

// Cerrar conexión
em.close();
emf.close();
```

Campos a guardar

Imagen 12. Inserción de entidades contactos con persist

CONSULTAS

Las operaciones de consultas o recuperación permiten obtener datos de la base de datos según las instrucciones que le indiquemos. Existen varias formas de consultar los datos, mediante un identificador, con una query estática o una query dinamica.

> Mediante un identificador

Cada objeto tiene un campo único, su clave primaria, conociendo el valor de la clave primaria se puede recuperar el objeto asociado a ese identificador. Para ello se usará el metodo .find(clase, identificador).

```
String rutaBBDD = "$objectdb/db/contactos.odb";

// Abrir conexión
EntityManagerFactory emf = Persistence.createEntityManagerFactory(rutaBBDD);
EntityManager em = emf.createEntityManager();

int identificador = 1;

Contacto contacto = em.find(Contacto.class, identificador);
System.out.println(contacto);

// Cerrar conexión
em.close();
emf.close();
```

Imagen 13. Ejemplo de consulta por identificador

> Con query estática.

Las query estáticas se define previamente en la cabecera de la clase. Permite tenerlas fuera de código usando etiquetas y se pueden usar en cualquier parte del programa que tenga visibilidad a la clase.

El procedimiento es declarar la consulta con la etiqueta @NamedQuery(name="nombre consulta", query="consulta"). Si hubieran más de una query las etiquetas se incluirian dentro de @NamedQueries. Una vez definidas desde código podemos llamar a la consulta por su nombre usando el método createNamedQuery("nombre de la consulta", clase del elemento).

```
@Entity
@NamedQueries({
    @NamedQuery(name="Contacto.consultaTodos", query="SELECT c FROM Contacto c"),
    @NamedQuery(name="Contacto.consultaId", query="select c FROM Contacto c WHERE c.id=:identificador")
})
public class Contacto implements Serializable {

    private static final long serialVersionUID = -1226598445347957691L;

    @Id
    @GeneratedValue
    private long id;

    private String nombre;

    @Embedded
    private Direccion direccion;

    private String apellido1;
    private String apellido2;
    private String email;
```

```
String rutaBBDD = "$objectdb/db/contactos.odb";

// Abrir conexión
EntityManagerFactory emf = Persistence.createEntityManagerFactory(rutaBBDD);
EntityManager em = emf.createEntityManager();

int identificador = 2;
TypedQuery<Contacto> consultaAlumnos= em.createNamedQuery("Contacto.consultaId", Contacto.class);
consultaAlumnos.setParameter("identificador", identificador);

List<Contacto> results = consultaAlumnos.getResultList();

for (Contacto c : results) {
    System.out.println(c);
}
```

Imagen 14. Ejemplo de consulta con query estática.

> Con query dinamica.

Una query dinamica es una consulta que se forma en tiempo de ejecución. Se define en el mismo código y permite flexibilidad dependiendo de los datos que pueda introducir el usuario.

El procedimiento a seguir será formar en un String la sentencia JPQL junto con los parametros de busqueda. Con el String de la consulta crear la query con .createQuery(consulta, clase).

```
String rutaBBDD = "$objectdb/db/contactos.odb";

// Abrir conexión
EntityManagerFactory emf = Persistence.createEntityManagerFactory(rutaBBDD);
EntityManager em = emf.createEntityManager();

int identificador = 7;
String consulta = "SELECT c FROM Contacto c WHERE id ="+identificador;

TypedQuery<Contacto> query = em.createQuery(consulta, Contacto.class);
List<Contacto> results = query.getResultList();
for (Contacto c : results) {
    System.out.println(c);
}

// Cerrar conexión
em.close();
emf.close();
```

Imagen 15. Ejemplo de consulta por query dinamica

MODIFICACIÓN

La modificación de elementos se produce cuando ya existe el elemento en la base de datos. Necesitamos conocer que campos se modificaran y sobre que elementos, para ello se usará su identificador unico, la clave primaria.

> Mediante un identificador

Primero hay que buscar el elemento a partir de su identificador. Con el elemento obtenido se procede la transacción y su actualización haciendo uso de los metodos set de la clase para finalizar se confirman los cambios con el commit().

```
String rutaBBDD = "$objectdb/db/contactos.odb";

// Abrir conexión
EntityManagerFactory emf = Persistence.createEntityManagerFactory(rutaBBDD);
EntityManager em = emf.createEntityManager();

int identificador = 7;

Contacto contacto = em.find(Contacto.class, identificador);
em.getTransaction().begin();
contacto.setNombre("Carla");
em.getTransaction().commit();

// Cerrar conexión
em.close();
emf.close();
```

Imagen 16. Ejemplo de actualización por identificador

> Con una query

Definimos la sentencia JPQL en un String y se contruye la query mediante createQuery(sentencia JPQL), finalmente se ejecuta la query con executeUpdate().

```
String rutaBBDD = "$objectdb/db/contactos.odb";

// Abrir conexión
EntityManagerFactory emf = Persistence.createEntityManagerFactory(rutaBBDD);
EntityManager em = emf.createEntityManager();

em.getTransaction().begin();
int identificador = 7;
String query = "UPDATE Contacto SET nombre = \"Susana\" WHERE id = "+identificador;
Query consulta = em.createQuery(query);
int resultado = consulta.executeUpdate();
em.getTransaction().commit();

// Cerrar conexión
em.close();
emf.close();
```

Imagen 17. Ejemplo de actualización por query

BORRADO

Permite borrar elementos de la base de datos, el procedimiento es similar al de modificación.

> Mediante un identificador

Primero hay que buscar el elemento a partir de su identificador. Con el elemento obtenido se procede la transacción y su borrado con .remove(elemento), finalmente se confirman los cambios con commit().

```
String rutaBBDD = "$objectdb/db/contactos.odb";

// Abrir conexión
EntityManagerFactory emf = Persistence.createEntityManagerFactory(rutaBBDD);
EntityManager em = emf.createEntityManager();

Contacto contacto = em.find(Contacto.class, 1);
em.getTransaction().begin();
em.remove(contacto);
em.getTransaction().commit();

// Cerrar conexión
em.close();
emf.close();
```

Imagen 18. Ejemplo de borrado de un elemento por identificador

> Con una query

Definimos la sentencia JPQL en un String y se contruye la query mediante createQuery(sentencia JPQL), finalmente se ejecuta la query con executeUpdate().

```
String rutaBBDD = "$objectdb/db/contactos.odb";

// Abrir conexión
EntityManagerFactory emf = Persistence.createEntityManagerFactory(rutaBBDD);
EntityManager em = emf.createEntityManager();

em.getTransaction().begin();
int identificador = 7;
String query = "DELETE FROM Contacto WHERE id = "+identificador;
Query consulta = em.createQuery(query);
int resultado = consulta.executeUpdate();
em.getTransaction().commit();

// Cerrar conexión
em.close();
emf.close();
```

Imagen 19. Ejemplo de borrado de un elemento por una query

13.7. Explorador de objetos

Para poder visualizar de forma gráfica la base de datos, ObjectDB incorpora un explorador de base de datos. El explorador se encuentra en la carpeta bin de la ruta de instalación.

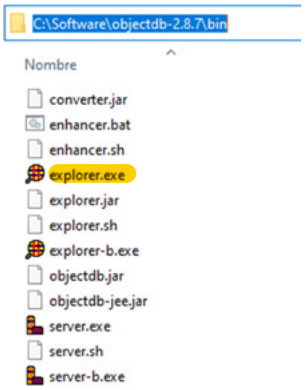

Imagen 20. Ubicación de explorer de ObjectDB

Con explorer es posible ver todas las entidades que contiene nuestra base de datos y visualizar los datos que hay guardados.

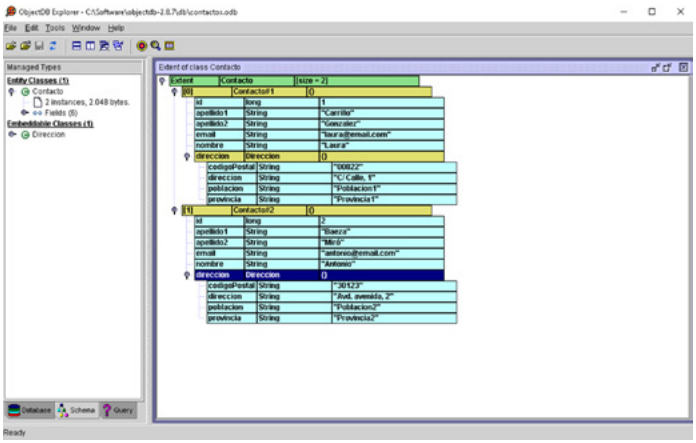

Imagen 21. Pantalla de Schema en explorer

En la pestaña de Query es posible lanzar consultas JPQL y ver su resultado.

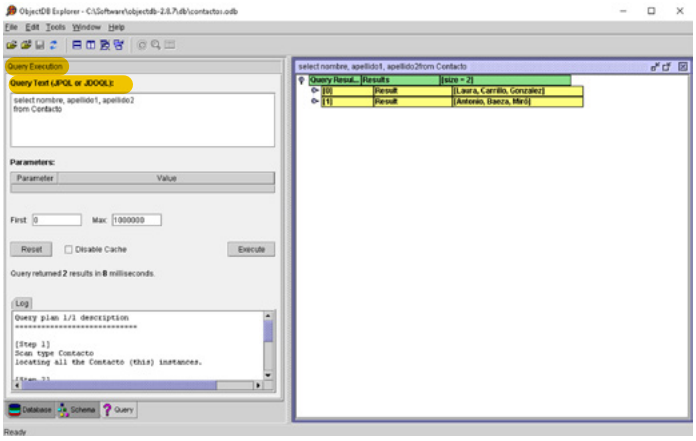

Imagen 22. Pantalla de Query en explorer

www.universae.com