# Tema 13

Descargar estos apuntes en pdf o html

# Índice

# Bases de Datos Relacionales

Las bases de datos relacionales organizan la información basándose en el modelo relacional. Este enfoque estructura los datos en tablas interconectadas, donde las relaciones se definen mediante el uso de claves primarias y claves foráneas.

- **Clave Primaria:** Un atributo que distingue de manera única cada entrada dentro de una tabla.
- **Clave Foránea:** Un atributo en una tabla que hace referencia a la clave primaria de otra tabla, estableciendo un vínculo entre ellas.

Cada tabla debe tener una clave primaria, asegurando la unicidad de sus registros. Adicionalmente, una tabla puede contener una o varias claves foráneas, que sirven para relacionarla con otras tablas.

Los datos dentro de cada columna de una tabla se almacenan siguiendo un formato específico, que puede ser numérico, textual, de fecha, entre otros. Cada columna también tiene un nombre que la identifica. La naturaleza de los datos almacenados en una columna puede estar sujeta a ciertas restricciones, como la obligatoriedad de no ser nula o la exigencia de unicidad. Los tipos de datos disponibles varían según el Sistema de Gestión de Bases de Datos (SGBD) empleado.

En esencia, una tabla se puede entender como un conjunto de registros, donde cada registro está compuesto por una serie de campos.

La manipulación de estas bases de datos se realiza mediante el Lenguaje de Consulta Estructurado (SQL), que facilita la creación, modificación y consulta de las tablas.

```
CREATE TABLE tablaN (
    campo1 tipo1,
    campo2 tipo2,
    campo3 tipo3,
    ...
    campoI tipoI REFERENCES tablaM(campoJ),
    PRIMARY KEY (campo1)
);
```

# El Desfase Objeto-Relacional

Al desarrollar aplicaciones que interactúan con bases de datos, surge una divergencia: el modelo relacional difiere del modelo orientado a objetos. El modelo relacional se centra en tablas, mientras que el modelo orientado a objetos se basa en la noción de objetos. Esta diferencia impide la utilización directa de las tablas de una base de datos relacional en un programa orientado a objetos. Además, pueden surgir incompatibilidades en los tipos de datos. En resumen, existe una falta de compatibilidad directa entre ambos modelos, lo que requiere un esfuerzo adicional para integrarlos.

Esto implica la necesidad de gestionar relaciones y realizar la conversión de objetos a tablas (o entidades) y viceversa cada vez que se requiera la transición entre ambos modelos: de un objeto a una fila en una tabla, y de una fila de una tabla a un objeto.

Para mitigar este problema, se pueden emplear **mapeadores objeto-relacionales** (**ORM**), que permiten trabajar con objetos en lugar de tablas y filas. Estos mapeadores se encargan de traducir las consultas realizadas a la base de datos en sentencias SQL, y viceversa.

```
CREATE TABLE persona (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    nombre VARCHAR(50),
    apellidos VARCHAR(100),
    edad INTEGER
);
```

```
public class Persona {
    public int Id { get; set; }
    public string Nombre { get; set; }
    public string Apellidos { get; set; }
    public int Edad { get; set; }
}
```

# Manejo de la Integridad: Código vs. Base de Datos

La discrepancia entre el modelo relacional y el modelo orientado a objetos puede generar complicaciones en la gestión de la integridad de los datos. Este aspecto debe analizarse cuidadosamente, especialmente si la base de datos es compartida y las reglas de negocio de una aplicación específica no coinciden con las de otras aplicaciones que también utilizan la misma base de datos.

Por ejemplo, consideremos una tabla de personas y otra de direcciones (en una base de datos compartida). Si se elimina una persona, pero no se eliminan sus direcciones, se puede producir una inconsistencia en los datos (esto puede ser aceptable en ciertos escenarios, pero no en otros). Este problema se puede abordar de dos maneras:

- **Delegar la integridad a la base de datos:** En este enfoque, la base de datos se encarga de eliminar las direcciones asociadas a una persona eliminada. Para lograr esto, la tabla de direcciones incluirá una clave foránea que haga referencia a la tabla de personas, y se utilizarán mecanismos de propagación en cascada. De esta forma, la eliminación de una persona desencadenará automáticamente la eliminación de sus direcciones en la base de datos.
- **Gestionar la integridad mediante código:** En este caso, la aplicación es responsable de eliminar las direcciones de una persona que se elimina. Esto implica que la aplicación debe consultar las direcciones de la persona en cuestión y eliminarlas individualmente. Este método ofrece mayor flexibilidad, permitiendo definir casos específicos en los que se elimina una persona pero no sus direcciones, algo que no es posible con el enfoque anterior.

La elección entre estos enfoques depende del contexto específico de la aplicación, su ciclo de vida, el ciclo de vida de la base de datos, las aplicaciones que la utilicen y las reglas de negocio de cada una de ellas.

# Autonuméricos vs. UUID

Otro desafío al programar con bases de datos es la gestión de los identificadores de los registros. En el modelo relacional, estos identificadores, conocidos como claves primarias, se almacenan en una columna de la tabla y deben ser únicos.

**Nuevamente, existen dos enfoques principales**: que la base de datos genere los identificadores, o que la aplicación se encargue de esta tarea. Ambas opciones presentan ventajas y desventajas.

1. Si la base de datos genera los identificadores, se debe utilizar un tipo de dato que garantice valores únicos y no repetidos. Una opción común son los **valores autoincrementales**, que son enteros sin signo que se incrementan automáticamente con cada nuevo registro. Sin embargo, esto implica que el identificador de un nuevo registro no se conoce hasta que el registro se inserta en la base de datos. Esto puede ser problemático si se necesita el identificador para insertar registros relacionados, ya que sería necesario consultarlo después de la inserción inicial.
2. Si la aplicación genera los identificadores, también se requiere un tipo de dato que asegure la unicidad. Una alternativa es utilizar **UUID** (**Identificador Único Universal**), que genera identificadores únicos de forma aleatoria. Esto permite generar un identificador único para cada registro antes de insertarlo en la base de datos. Esto es una ventaja, pero también implica que la aplicación debe encargarse de almacenar y gestionar estos identificadores para su uso posterior, por ejemplo, al insertar registros relacionados que requieran el identificador como clave foránea.
   En C# se puede generar un UUID de la siguiente manera:

```
// "N" es el formato de 32 caracteres sin guiones
string uuid = Guid.NewGuid().ToString("N");

// "D" es el formato de 32 caracteres con guiones
string uuid = Guid.NewGuid().ToString("D");
```

# Base de datos Relacional SQLite

SQLite es un sistema de gestión de bases de datos relacional que se caracteriza por ser **ligero**, **rápido** y **autónomo**. No requiere un servidor dedicado, ya que se ejecuta directamente en el sistema operativo. Esto lo convierte en una excelente opción para aplicaciones que necesitan una base de datos local sin la complejidad de un servidor de bases de datos.

SQLite es compatible con la mayoría de los lenguajes de programación y sistemas operativos, lo que lo hace muy versátil. Además, es ampliamente utilizado en aplicaciones móviles, navegadores web y otros entornos donde se requiere una base de datos local.

# Acceso a Bases de Datos SQLite en C#

## Instalación

Para trabajar con SQLite en C#, se puede utilizar la biblioteca `System.Data.SQLite`, que proporciona una interfaz para interactuar con bases de datos SQLite desde aplicaciones .NET.

Para añadir esta biblioteca a un proyecto de C#, los más sencillo es usar **Nugget**. Para ello, se puede ejecutar el siguiente comando en un terminal **en la raíz del Workspace del proyecto**:

```
dotnet add package Microsoft.Data.Sqlite
```

## Conexión

Deberemos crear una única conexión a la base de datos, que se abrirá al principio de la aplicación y se cerrará al final.

```
string cadenaConexion = "Data Source=nombreBD.db";
using SqliteConnection conexion = new(cadenaConexion);

conexion.Open();
// Operaciones con la base de datos
conexion.Close();
```

Como `SqliteConnection` implementa el interfaz `IDisposable`, podemos usar la sentencia `using` para asegurarnos de que la conexión se cierra correctamente y por tanto ya no hará falta llamar al método `conexion.Close()`.

```
try
{
    string cadenaConexion = "Data Source=nombreBD.db";
    using SqliteConnection conexion = new(cadenaConexion);
    conexion.Open();
    // Operaciones con la base de datos
}
catch (SqliteException e)
{
    Console.WriteLine(e.Message);
}
```

## Action Queries utilizando el DDL de SQLite

Las **Data Definition Language (DML)** son sentencias SQL que modifican la estructura de la base de datos, como `CREATE`, `ALTER` o `DROP`.

Por ejemplo, para crear una la tala `Persona` con los campos `Id`, `Nombre`, `Apellidos` y `Edad`, se puede utilizar la siguiente sentencia:

```
string sql = """
CREATE TABLE IF NOT EXISTS persona (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    nombre TEXT NOT NULL,
    apellidos TEXT NOT NULL,
    edad INTEGER
)
""";
using SqliteCommand comando = new(sql, conexion);
comando.ExecuteNonQuery();
```

Fíjate que estamos utilizando la cláusula `using`

> **Nota**
>
> Cómo vemos en el ejemplo anterior, estamos incluyendo lenguaje SQL cómo cadena dentro de otro lenguaje como C#. Esto tiene varias desventajas:
>
> 1. Los errores de sintaxis en el SQL no se detectan hasta que se ejecute el método `ExecuteNonQuery()`.
> 2. La cadena SQL no tiene resaltado de sintaxis ni autocompletado.
> 3. Si interpolamos cadenas dentro de la sentencia SQL, podemos ser vulnerables a ataques de **SQL Injection**.
>
> Todas estas desventajas se pueden evitar utilizando **Query Builders** o **ORMs**.

## Action Queries utilizando el DML de SQLite

Las **Data Manipulation Language (DML)** son sentencias SQL que modifican los datos de la base de datos, como `INSERT`, `UPDATE` o `DELETE`.

```
string sql = """
INSERT INTO persona (
    nombre, apellidos, edad
) VALUES (
    'Juan', 'García', 25)";
""";

using SqliteCommand comando = new(sql, conexion);
comando.ExecuteNonQuery();
```

Si nos fijamos la forma de ejecutarlas es idéntica a las **DDL**.

También podremos **parametrizar las consultas** para evitar **SQL Injection**.

```
string sql = """
INSERT INTO persona (
    nombre, apellidos, edad
) VALUES (
    @nombre, @apellidos, @edad)";
""";

using SqliteCommand comando = new(sql, conexion);
comando.Parameters.AddWithValue("@nombre", "Juan");
comando.Parameters.AddWithValue("@apellidos", "García");
comando.Parameters.AddWithValue("@edad", 25);
comando.ExecuteNonQuery();
```

## Consultas utilizando el DQL de SQLite

Las **Data Query Language (DQL)** son sentencias SQL que recuperan datos de la base de datos, como `SELECT`.

```
string sql = "SELECT * FROM persona";
using SqliteCommand comando = new(sql, conexion);
using SqliteDataReader rs = comando.ExecuteReader();

while (rs.Read())
{
    Console.WriteLine($"""
    Id: {rs.GetInt32(0)}
    Nombre: {rs.GetString(1)}
    Apellidos: {rs.GetString(2)}
    Edad: {rs.GetInt32(3)}
    """);
}
```

Como vemos, la forma de ejecutarlas es muy similar a las *Action Queries* pero debemos utilizar el método `ExecuteReader()` en lugar de `ExecuteNonQuery()`. Este nos devolverá un `SqliteDataReader` que podremos recorrer para obtener cada una de las tuplas de datos devueltas por la consulta.

Además de los ***getters*** que devuelven el tipo concreto de cada columna pasando el índice de la misma empezando en 0. También podemos utilizar el nombre de la columna a través de un indizador.

```
while (rs.Read())
{
    Console.WriteLine($"""
    Id: {rs["id"]}
    Nombre: {rs["nombre"]}
    Apellidos: {rs["apellidos"]}
    Edad: {rs["edad"]}
    """);
}
```

En este caso devuelven un `object?` por lo que deberemos hacer un *casting* explícito al tipo concreto o usar `Convert.ToInt32(rs["id"])` para convertirlo.

Es importante también que **tengamos en cuenta si el campo es nullable o no** para tenerlo en cuenta en el casting a realizar.

## Transacciones

Las transacciones son un conjunto de operaciones que se ejecutan como una sola unidad. Si una de las operaciones falla, se deshacen todas las operaciones anteriores. Esto garantiza la integridad de los datos y evita inconsistencias.

Esto es especialmente importante a la hora de mantener la integridad referencial en las relaciones entre tablas. Imaginemos que queremos insertar una mascota y su dueño en la tabla `persona`. Primero insertaremos la persona y después la mascota. Si la inserción de la `mascota` en la tabla mascota falla, deberemos deshacer la inserción de la persona.

> **Nota**
>
> Para este ejemplo vamos a suponer que el **id** de persona ya no es autonumérico sino que es de tipo cadena y se genera en la aplicación mediante un **UUID**.

```
try
{
    string id = UUId.NewGuid().ToString("N");
    string sql = """
    INSERT INTO persona (
        id, nombre, apellidos, edad
    ) VALUES (
        @id, @nombre, @apellidos, @edad)";
    """;
    using SqliteCommand insertaPersona = new(sql, conexion);
    insertaPersona.Parameters.AddWithValue("@id", id);
    insertaPersona.Parameters.AddWithValue("@nombre", "Juan");
    insertaPersona.Parameters.AddWithValue("@apellidos", "García");
    insertaPersona.Parameters.AddWithValue("@edad", 25);

    string sqlMascota = """
    INSERT INTO mascota (
        nombre, especie, idPersona
    ) VALUES (
        @nombre, @especie, @idPersona)";
    """;
    using SqliteCommand insertaMascota = new(sqlMascota, conexion);
    insertaMascota.Parameters.AddWithValue("@nombre", "Rex");
    insertaMascota.Parameters.AddWithValue("@especie", "Perro");
    insertaMascota.Parameters.AddWithValue("@idPersona", id);

    conexion.BeginTransaction();
    insertaPersona.ExecuteNonQuery();
    insertaMascota.ExecuteNonQuery();
    conexion.Commit();
}
catch (SqliteException e)
{
    conexion.Rollback();
    Console.WriteLine(e.Message);
}
```

Si nos fijamos hacemos los siguiente:

1. `BeginTransaction()`
2. Hago los `INSERT`, `DELETE` o `UPDATE` que se se quedan en una caché.
3. Si durante la ejecución de las operaciones ocurre alguna excepción, se ejecuta `Rollback()` para deshacer todas las operaciones desde el `BeginTransaction()`.
4. `Commit()` para dar por válidas todas las operaciones en caché.

> 🎓 **Caso de estudio:**

**Descargar código del caso de estudio**

1. Vamos a crear una base de datos con **SQLite** con una sola tabla de **libros** llamada `biblioteca_1t.db` usando C#.

```
public static void Main()
{
    try
    {
        string cadenaConexion = "Data Source=biblioteca_1t.db";
        using SqliteConnection conexion = new(cadenaConexion);
        conexion.Open();

    }
    catch (SqliteException e)
    {
        Console.WriteLine(e.Message);
    }
}
```

2. A continuación, creamos un método estático `CrearTablaLibros` que recibe la conexión a la base de datos y crea la tabla `libro` borrándola si ya existe previamente.

```
public static void CrearTablaLibros(SqliteConnection conexion)
{
    string borrarTabla = "DROP TABLE IF EXISTS libro";
    using (SqliteCommand comando = new(borrarTabla, conexion))
    {
        comando.ExecuteNonQuery();
    }
    string crearTabla = """
        CREATE TABLE libro (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            titulo VARCHAR(60),
            autor VARCHAR(60)
        );
        """;
    using (SqliteCommand comando = new(crearTabla, conexion))
    {
        comando.ExecuteNonQuery();
    }
}
```

Fíjate que utilizamos la cláusula `using` para asegurarnos de que los recursos se liberan correctamente y les definimos un ámbito para que poder utilizar el mismo identificador de variable en ambos bloques.
Por último, llamamos al método `CrearTablaLibros` en el `Main` después de crear la conexión.

```
// ...
conexion.Open();
CrearTablaLibros(conexion);
// ...
```

3. Creamos ahora un método para insertar todos los libros.

```
public static void AñadeLibros(SqliteConnection conexion)
{
    string insertarDatos = """
        INSERT INTO libro (titulo, autor) VALUES
        ('Macbeth', 'William Shakespeare'),
        ('La Celestina (Tragicomedia de Calisto y Melibea)', 'Fernando de Rojas'),
        ('El Lazarillo de Tormes', 'Anónimo'),
        ('20.000 Leguas de Viaje Submarino', 'Julio Verne'),
        ('Alicia en el País de las Maravillas', 'Lewis Carrol'),
        ('Cien Años de Soledad', 'Gabriel García Márquez'),
        ('La tempestad', 'William Shakespeare');
        """;

    using SqliteCommand comando = new(insertarDatos, conexion);
    comando.ExecuteNonQuery();
}
```

y lo llamamos después de crear la tabla.

```
// ...
conexion.Open();
CrearTablaLibros(conexion);
AñadeLibros(conexion);
// ...
```

4. Por último, creamos un método para ver todos los libros realizando una consulta a la BD.

```
static void VerLibros(SqliteConnection conexion)
{
    string verLibros = "SELECT * FROM libro";
    using SqliteCommand query = new(verLibros, conexion);
    using SqliteDataReader rs = query.ExecuteReader();

    string separador = new(
        $"| {new('-', 3),-3} | {new('-', 55),-55} | {new('-', 25),-25} |\n"
    );

    StringBuilder salida = new StringBuilder(separador)
    .Append($"| {"Id",-3} | {"Título",-55} | {"Autor",-25} |\n")
    .Append(separador);
    while (rs.Read())
    {
        salida.Append(
            $"| {rs["id"],-3} | {rs["titulo"],-55} | {rs["autor"],-25} |\n"
        );
    }
    salida.Append(separador);
    Console.WriteLine(salida);
}
```

y lo llamamos después de añadir los libros.

```
// ...
conexion.Open();
CrearTablaLibros(conexion);
AñadeLibros(conexion);
VerLibros(conexion);
// ...
```

Nos debería mostrar una tabla similar a la siguiente:

| Id | Título | Autor |
|---|---|---|
| 1 | Macbeth | William Shakespeare |
| 2 | La Celestina (Tragicomedia de Calisto y Melibea) | Fernando de Rojas |
| 3 | El Lazarillo de Tormes | Anónimo |
| 4 | 20.000 Leguas de Viaje Submarino | Julio Verne |
| 5 | Alicia en el País de las Maravillas | Lewis Carrol |
| 6 | Cien Años de Soledad | Gabriel García Márquez |
| 7 | La tempestad | William Shakespeare |

# Patrones Básicos de Diseño

El trabajo con bases de datos en aplicaciones orientadas a objetos puede beneficiarse de la aplicación de patrones de diseño. Estos patrones proporcionan soluciones probadas y eficaces a problemas comunes en el desarrollo de software.

Entre los más básicos es utilizar el **patrón DAO** con **operaciones CRUD**, modelando las **tablas a través de clases**.

Vamos a verlo a través de la modificación del caso de estudio anterior.

## Modelando los datos

En primer lugar crearemos una clase llama `Libro` que modelará la tabla `libro` definida. A esta clase la denominaremos **entidad**.

```
CREATE TABLE libro (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    titulo VARCHAR(60),
    autor VARCHAR(60)
);
```

Aunque nosotros la vamos a llamar `Libro` en algunos casos se le puede llamar `LibroEntity` y definirse también usando un `record`.

```
public class Libro
{
    public int Id { get; set; }
    public string Titulo { get; set; }
    public string Autor { get; set; }
}
```

## Definiendo un DAO