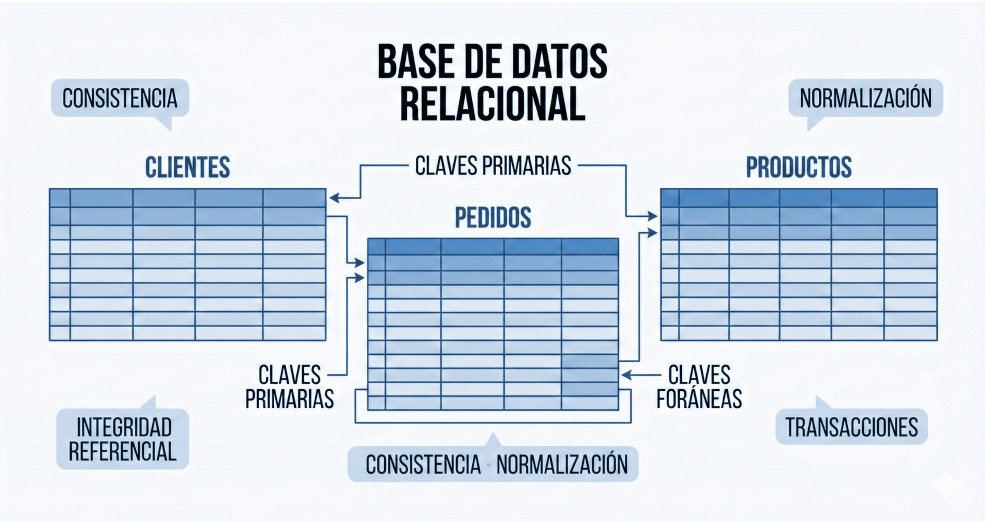
Unidad 24

Descargar estos apunte en pdf o html

Índice

Conceptos Básicos y Definiciones

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.

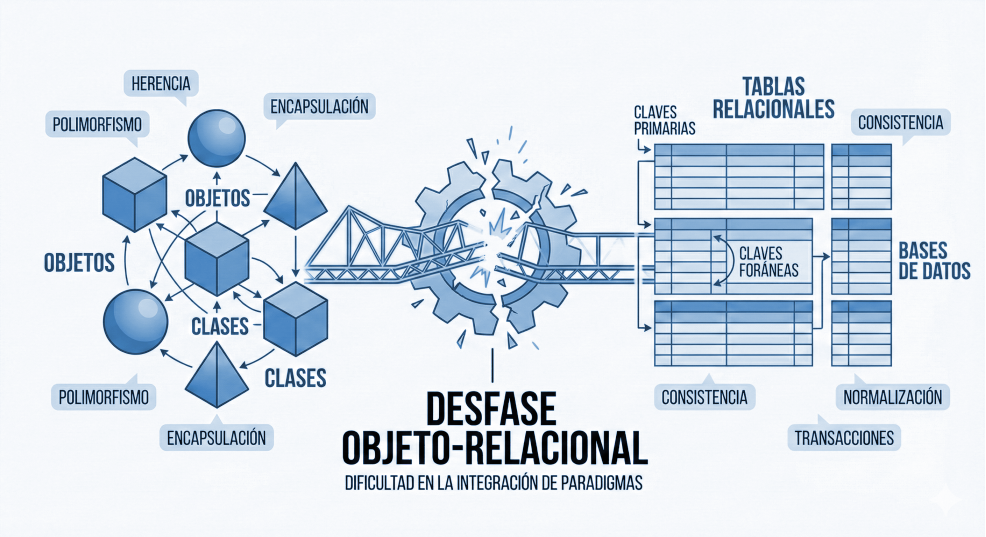
Cada tabla debe tener una clave primaria, asegurando la unicidad de sus registros. Adicionalmente, una tabla puede contener una o varias claves foráneas, que sirven para relacionarla con otras tablas.

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 y **declarativo SQL (Structured Query Language)**.

```
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.

Definición de la tabla `persona` en SQL

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

Definición de la clase `Persona`

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

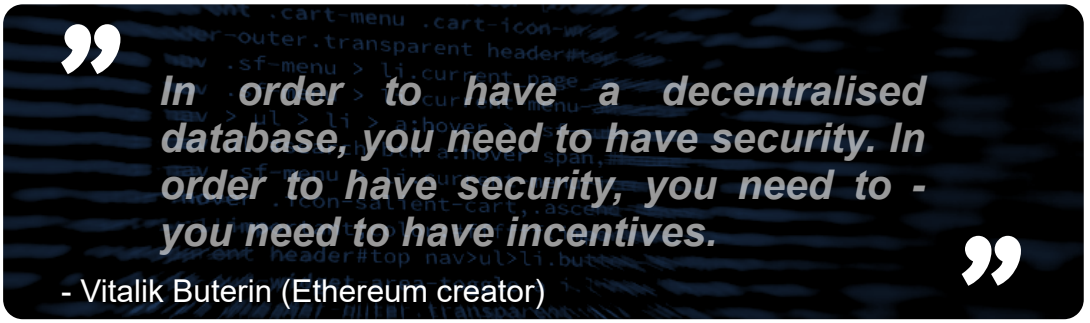
Manejo de la Integridad: Código vs. Base de Datos

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:

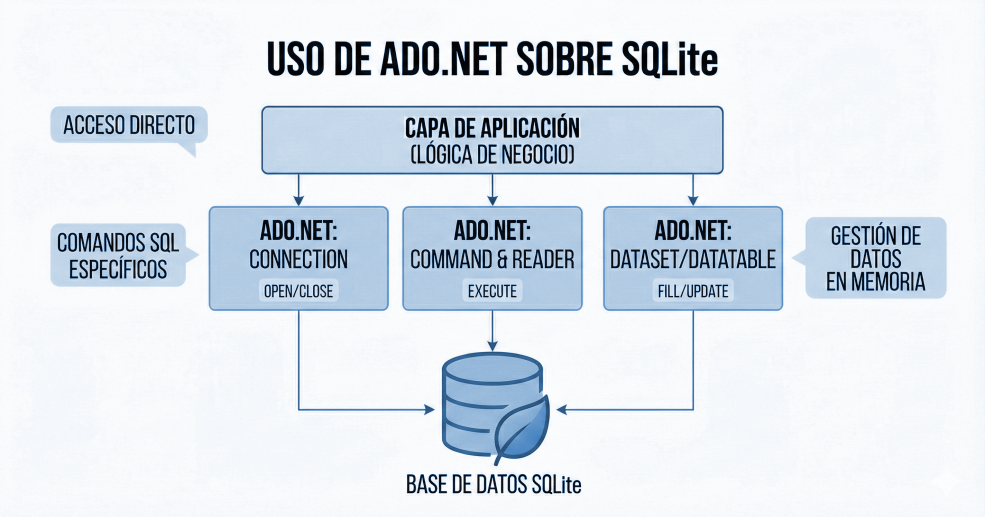
```
// "N" es el formato de 32 caracteres sin guiones
string uuid = Guid.NewGuid().ToString("N");

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

Base de datos Relacional SQLite

SQLite es un **SGBD 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 CSharp



Puesto que C# es una lenguaje de la plataforma **.NET de Microsoft** existe una **capa similar a JDBC en Java** llamada **ADO.NET** (**ActiveX Data Objects for .NET**). Esta existe desde principios de la década de 2000 y es la forma estándar de acceder a bases de datos relacionales desde aplicaciones .NET.

ADO.NET es, en esencia, el puente que permite a las aplicaciones .NET comunicarse con bases de datos. Esto lo hace a través de un conjunto de clases y librerías que te permiten conectar tu aplicación (ya sea web, escritorio o móvil) con fuentes de datos como SQL Server, Oracle o SQLite, para leer, escribir y modificar información.

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** del que hablaremos en otro tema. Para ello, se puede ejecutar el siguiente comando en un terminal **en la raíz del Workspace del proyecto**:

```
C:\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 (DDL)** 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`

> **Cuidado**
>
> 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** 💀. Por tanto, **nunca deberemos hacerlo** ❌.
>
> 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.

Ejemplo: Creación y Manipulación de una Tabla de Libros

Puedes descargar el código del ejemplo de **este enlace**.

Paso 1: Vamos a crear una base de datos con **SQLite** con una sola tabla de **libros** llamada `biblioteca_1t.db` usando C#. Para ello, vamos a definir en primer lugar un método estático `RutaEjecucion` que **nos devolverá la ruta del proyecto** independientemente de si estamos ejecutando desde VSCode o desde el terminal con `dotnet run`.

```
public static string RutaEjecucion() => Regex.Match(
    input: Directory.GetCurrentDirectory(),
    pattern: @"^(?<ruta>.*?)(?=\\bin)",
    options: RegexOptions.IgnoreCase
).Groups["ruta"].Value;
```

Para **tener localizada nuestra base de datos**, vamos a crear si no existe un directorio llamado `datos` en la raíz del proyecto y dentro de él crearemos la base de datos `biblioteca_1t.db`.

```
public static void Main()
{
    try
    {
        string rutaEjecucion = RutaEjecucion();
        string rutaDatos = Path.Combine(rutaEjecucion, "datos");
        if (!Directory.Exists(rutaDatos))
            Directory.CreateDirectory(rutaDatos);
        string cadenaConexion = $"Data Source={Path.Combine(rutaDatos, "biblioteca_1t.db")}";
        using SqliteConnection conexion = new(cadenaConexion);
        conexion.Open();
    }
    catch (SqliteException e)
    {
        Console.WriteLine(e.Message);
    }
}
```

Paso 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();
3   CrearTablaLibros(conexion);
    // ...
```

Paso 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);
    // ...
```

Paso 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.

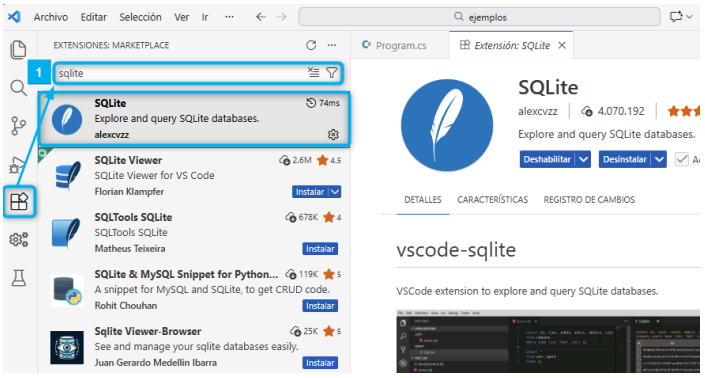
```
    // ...
    conexion.Open();
    CrearTablaLibros(conexion);
    AñadeLibros(conexion);
    VerLibros(conexion);
    // ...
```

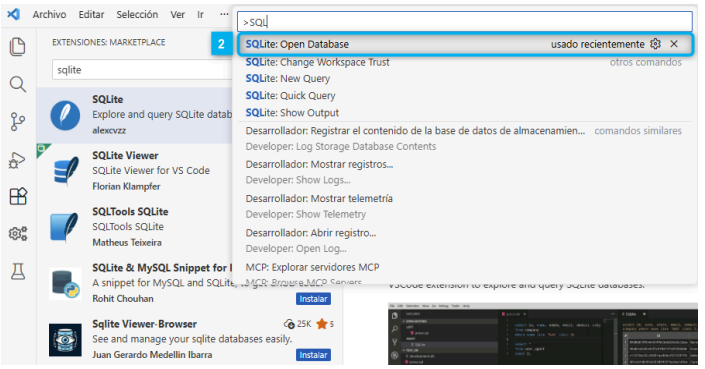
Nos debería mostrar una tabla similar a la siguiente:

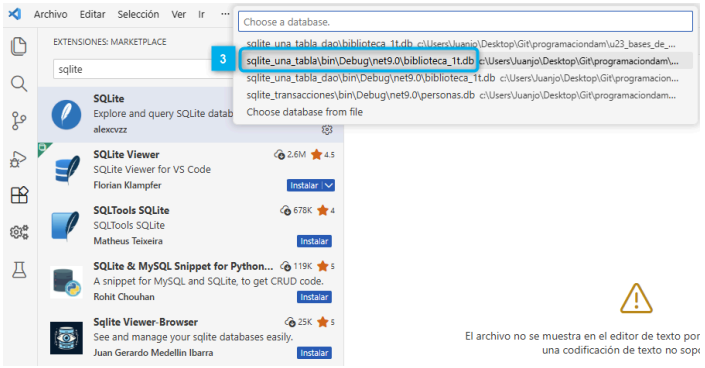
Id	Título	Autor
1	Macbeth	William Shakespeare
2	La Celestina (Tragicomedia de Calisto y Melibea)	Fernando de Rojas
3	El Lazarillo de Tormes	Anónimo
4	20.000 Leguas de Viaje Submarino	Julio Verne
5	Alicia en el País de las Maravillas	Lewis Carrol
6	Cien Años de Soledad	Gabriel García Márquez

Id	Título	Autor
7	La tempestad	William Shakespeare

Examinando Bases de Datos SQLite en VSCode







1️⃣ Vamos a extensiones y buscamos **sqlite**. Instalamos la extensión llamada **SQLite** de **alexcvzz** que es la más popular y suficiente para nuestros propósitos.

2️⃣ Vamos a la opción de menú `Ver -> Paleta de Comandos` (o `Ctrl+Shift+P`) y filtramos por `SQL...` y nos ofrecerá todos los comandos relacionados con SQLite. Seleccionamos `SQLite: Open Database`.

3️⃣ Nos mostrará todas las bases de datos `.db` que encuentre en el espacio de trabajo abierto ya sea proyecto o solución. Seleccionamos la que queramos examinar que se debería encontrar en la carpeta `datos\biblioteca_1t.db`. **Si aún no hemos generado ninguna, deberemos ejecutar alguna aplicación que cree una base de datos SQLite.**

4️⃣ Aparecerá el icono del explorador de BD de SQLite en la barra lateral izquierda. Hacemos clic en él para ver las tablas de la base de datos y al ejecutar una tabla nos mostrará sus datos en una nueva pestaña.

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 dos personas en la tabla `persona` y queremos que se inserten las dos o ninguna. Si la segunda inserción falla, queremos que se deshaga la otra.

Para el siguiente 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** y vamos a usar el mismo uuid para ambas inserciones.

```
// Hacemos una pequeñas modificaciones en la tabla persona.
public static void CrearTablaPersona(SqliteConnection conexion)
{
    string sql = """
    CREATE TABLE IF NOT EXISTS persona (
        id VARCHAR(255) PRIMARY KEY NOT NULL,
        nombre VARCHAR(255) NOT NULL,
        apellidos VARCHAR(255) NOT NULL,
        edad INTEGER
    )
    """;
    using SqliteCommand comando = new(sql, conexion);
    comando.ExecuteNonQuery();
}
```

Si nos fijamos hacemos los siguiente:

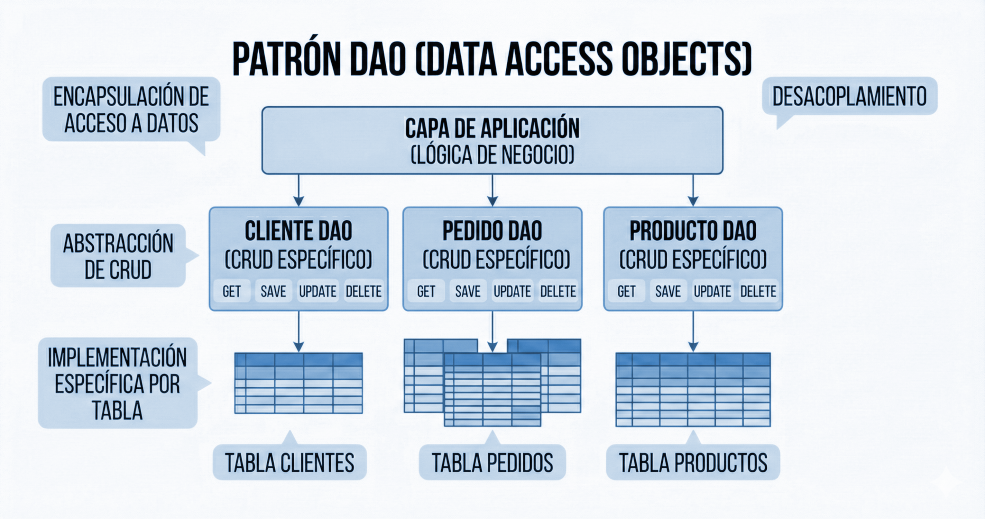
1. Definimos los dos comando `SqliteCommand` para insertar dos personas pero aún no los ejecutamos.
2. Obtengo la transacción con `transaction = conexion.BeginTransaction();`
3. Asocio la propiedad `Transaction` de cada comando a la transacción obtenida en el paso anterior.
4. Ejecuto los comandos con `ExecuteNonQuery()`.
    - Si durante la ejecución de los comandos **ocurre alguna excepción**, se ejecuta `transaction?.Rollback()` para deshacer todas las **operaciones asociadas a la transacción** desde el `BeginTransaction()`.
    - Si todo va bien, se ejecuta `transaction.Commit()` para dar por válidas todas los comandos y **reflejarse en la base de datos de manera permanente**.

```
public static void Main()
{
    string cadenaConexion = "Data Source=personas.db";
    using SqliteConnection conexion = new(cadenaConexion);
    conexion.Open();
    SqliteTransaction? transaction = null;
    try
    {
        CrearTablaPersona(conexion);

        string id = Guid.NewGuid().ToString("N");
        string sql = """
                    INSERT INTO persona (id, nombre, apellidos, edad)
                    VALUES (@id, @nombre, @apellidos, @edad);
                    """;
        using SqliteCommand insertaPersona1 = new(sql, conexion);
        insertaPersona1.Parameters.AddWithValue("@id", id);
        insertaPersona1.Parameters.AddWithValue("@nombre", "Juan");
        insertaPersona1.Parameters.AddWithValue("@apellidos", "García");
        insertaPersona1.Parameters.AddWithValue("@edad", 25);

        using SqliteCommand insertaPersona2 = new(sql, conexion);
        // id = Guid.NewGuid().ToString("N");
        insertaPersona2.Parameters.AddWithValue("@id", id);
        insertaPersona2.Parameters.AddWithValue("@nombre", "María");
        insertaPersona2.Parameters.AddWithValue("@apellidos", "Sánchez");
        insertaPersona2.Parameters.AddWithValue("@edad", 22);

        // Inicio una transacción en la conexión y signo la transacción a los comandos
30      transaction = conexion.BeginTransaction();
        insertaPersona1.Transaction = transaction;
        insertaPersona2.Transaction = transaction;
        // Ejecuto los comandos
        insertaPersona1.ExecuteNonQuery();
        insertaPersona2.ExecuteNonQuery();
        // Si todo va bien, confirmo la transacción
37      transaction.Commit();
    }
    catch (SqliteException e)
    {
        // Si ocurre un error, deshago todos los cambios realizados en la transacción hasta el error.
42      transaction?.Rollback();
        Console.WriteLine(e.Message);
    }
}
```

Patrones de Diseño con Bases de Datos

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.

DAO y CRUD



El patrón más básico 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. Puedes descargar el código del ejemplo de **este enlace**.

Modelando los datos en entidades

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)
);
```

Simplemente la podemos denominar `Libro` yen algunos casos se le puede llamar `LibroEntity`. Nosotros la vamos a llamar simplemente `Libro`.

```
public class Libro
{
    public int Id { get; }
    public string Titulo { get; set; }
    public string Autor { get; set; }
    public Libro(int id, string titulo, string autor)
    {
        Id = id;
        Titulo = titulo;
        Autor = autor;
    }
}
```

Definiendo un DAO

DAO (**Data Access Object**) es un patrón de diseño que se utiliza para abstraer la capa de acceso a datos de una aplicación. Este patrón permite separar la lógica de negocio de la lógica de acceso a datos, facilitando el mantenimiento y la escalabilidad del código.

Podemos decir que la lógica de negocio se encuentra en las entidades, en nuestro ejemplo la clase `Libro` y la lógica de acceso a datos en los DAOs. De tal manera que desde la vista (la consola) solo se interactuará con las entidades o clases de nuestro modelo y no con la base de datos directamente.

Primero vamos ha definir la clase `LibroDAO` que contendrá una propiedad de tipo `SqliteConnection` y un constructor que reciba la cadena de conexión y la abra. Además, implementará la interfaz `IDisposable` para cerrar la conexión y poder usar la cláusula `using` a la hora de crear un objeto de esta clase. Por tanto, todas las operaciones sobre la BD, se realizarán usando la misma conexión.

Si te fijas en el código siguiente, primero definimos la propiedad `Conexion` que es de **solo lectura y privada**.

Luego implementamos el interfaz `IDisposable` para cerrar la conexión cuando ya no se necesite. Para ello, implementamos el **patrón de disposición recomendado por Microsoft** que incluye un campo privado `_disposed` para evitar llamadas múltiples al método `Dispose`. En él, primero invalidamos el método `Dispose` público llamando al método protegido `Dispose(bool disposing)` (que es una invalidación de la clase `Object`) con el parámetro `true` y luego llamamos a `GC.SuppressFinalize(this)` para evitar que el recolector de basura llame al finalizador.

```
public class LibroDAO : IDisposable
{
    private SqliteConnection? Conexion { get; }
    private bool _disposed = false;

    public LibroDAO(string cadenaConexion)
    {
        Conexion = new SqliteConnection(cadenaConexion);
        Conexion.Open();
    }
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!_disposed)
        {
            if (disposing)
            {
                if (Conexion != null
                    && Conexion.State
                       != ConnectionState.Closed)
                {
                    Conexion.Close();
                }
            }
            _disposed = true;
        }
    }
}
```

Implementando operaciones CRUD en el DAO

CRUD es el acrónimo de **Create**, **Read**, **Update** y **Delete**, que son las operaciones básicas que se pueden realizar sobre una base de datos. Estas operaciones trabajarán sobre objetos de la capa de negocio, en nuestro caso con objetos de la clase `Libro`.

> **Nota**
>
> Inicialmente, nosotros vamos ha usar los nombres de los métodos en inglés, pero no tienen porque estar en inglés o incluso podemos utilizar sinónimos de los mismos. Por ejemplo, en lugar de `Create` podríamos utilizar `Insert`.

Create o Insert

```
public void Create(Libro libro)
{
    string insertarDatos = "INSERT INTO libro (titulo, autor) VALUES (@titulo, @autor)";
    using var comando = new SqliteCommand(insertarDatos, Conexion);
    comando.Parameters.AddWithValue("@titulo", libro.Titulo);
    comando.Parameters.AddWithValue("@autor", libro.Autor);
    comando.ExecuteNonQuery();
}
```

Si nos fijamos en el método `Create`, recibe un objeto de la clase `Libro` y lo inserta en la tabla `libro`. Encapsulando la lógica de acceso a datos. En nuestro caso la consulta **SQL** de inserción.

Read

`IEnumerable<Libro> Read()` sería el método más básico de lectura es el que nos devuelve todos los registros de la tabla. Una posible implementación sería devolviendo una **secuencia perezosa** de objetos `Libro` utilizando `yield return`. Es la opción más eficiente en cuanto a memoria y rendimiento **si la tabla tiene miles de registros**.

> **Nota**
>
> Podríamos pensar que `rs` que es de tipo `SqliteDataReader` contiene todos los registros cargados en memoria. Pero no es así, `SqliteDataReader` implementa una lectura por streaming con una caché interna que va leyendo los registros de la base de datos según se van necesitando.

```
public IEnumerable<Libro> Read()
{
    string verLibros = "SELECT * FROM libro";
    using var query = new SqliteCommand(verLibros, Conexion);
    using var rs = query.ExecuteReader();

    while (rs.Read())
    {
        yield return new
        (
            id: Convert.ToInt32(rs["id"]),
            titulo: rs["titulo"].ToString()!,
            autor: rs["autor"].ToString()!
        );
    }
}
```

También es interesante tener un método `Read` que nos devuelva un libro por su `id` o por su `titulo` para realizar búsquedas simples. En las siguientes propuestas `Libro` es un tipo **nullable** para poder devolver `null` en caso de que no se encuentre el libro.

```
public Libro? Read(int id)
{
    string verLibro = "SELECT * FROM libro WHERE id = @id";
    using var query = new SqliteCommand(verLibro, Conexion);
    query.Parameters.AddWithValue("@id", id);
    using var rs = query.ExecuteReader();

    return rs.Read()
    ? new Libro
    (
        id: Convert.ToInt32(rs["id"]),
        titulo: rs["titulo"].ToString()!,
        autor: rs["autor"].ToString()!
    )
    : null;
}
```

```
public Libro? Read(string titulo)
{
    string verLibro = "SELECT * FROM libro WHERE titulo = @titulo";
    using var query = new SqliteCommand(verLibro, Conexion);
    query.Parameters.AddWithValue("@titulo", titulo);
    using var rs = query.ExecuteReader();

    return rs.Read()
    ? new Libro
    (
        id: Convert.ToInt32(rs["id"]),
        titulo: rs["titulo"].ToString()!,
        autor: rs["autor"].ToString()!
    )
    : null;
}
```

Read de proyecciones y agregaciones usando DTO

DTO (**Data Transfer Object**) es un patrón de diseño que se utiliza para transferir datos entre subsistemas de un software. En el contexto de acceso a datos, un **DTO** es un objeto que se utiliza para transferir datos entre la capa de acceso a datos y la capa de presentación o lógica de negocio.

Por ejemplo, si queremos obtener el número de libros por autor, podemos crear un **DTO** llamado `LibrosPorAutorDTO` que contenga el nombre del autor y el número de libros que ha escrito.

En este caso, utilizaremos un `record` para definir el **DTO** sería la opción más adecuada.

> **Nota**
>
> Fíjate que aunque aquí también devolvemos un `IEnumerable<LibrosPorAutorDTO>`, en este caso no es una secuencia perezosa ya que primero cargamos todos los datos en una lista y luego la devolvemos. Esto es porque **hemos supuesto que consulta devuelve un número reducido de registros** (número de autores).

```
public record class LibrosPorAutorDTO(string Autor, int Titulos);

public IEnumerable<LibrosPorAutorDTO> ReadLibrosPorAutor()
{
    string verLibros = """
        SELECT autor, COUNT(*) AS cantidad
        FROM libro
        GROUP BY autor
        ORDER BY cantidad DESC
        """;
    using var query = new SqliteCommand(verLibros, Conexion);
    using var rs = query.ExecuteReader();

    List<LibrosPorAutorDTO> librosXAutor = [];
    while (rs.Read())
    {
        librosXAutor.Add(new LibrosPorAutorDTO(
            Autor : rs["autor"].ToString()!,
            Titulos : Convert.ToInt32(rs["cantidad"])
        ));
    }
    return librosXAutor;
}
```

Update

En la siguiente propuesta de implementación del método `Update`, recibe el libro a actualizar y definimos una precondición que es que el libro tenga un `id` existente en la tabla.

```
public void Update(Libro libro)
{
    if (Read(libro.Id) == null)
        throw new ArgumentException("El libro no tiene un ID válido");

    string actualizarDatos = "UPDATE libro SET titulo = @titulo, autor = @autor WHERE id = @id";
    using var comando = new SqliteCommand(actualizarDatos, Conexion);
    comando.Parameters.AddWithValue("@titulo", libro.Titulo);
    comando.Parameters.AddWithValue("@autor", libro.Autor);
    comando.Parameters.AddWithValue("@id", libro.Id);
    comando.ExecuteNonQuery();
}
```

Delete

En la siguiente propuesta de implementación del método `Delete`, recibe el `id` del libro a borrar y como en el caso anterior definimos una precondición que es que el libro tenga un `id` existente en la tabla.

```
public void Delete(int id)
{
    if (Read(id) == null)
        throw new ArgumentException($"El id {id} no existe en la base de datos");

    string borrarDatos = "DELETE FROM libro WHERE id = @id";
    using var comando = new SqliteCommand(borrarDatos, Conexion);
    comando.Parameters.AddWithValue("@id", id);
    comando.ExecuteNonQuery();
}
```

Count

Aunque no es parte del CRUD, es interesante tener un método que nos devuelva el número de registros de la tabla. Para ello, podemos utilizar la función de agregación `COUNT(*)` de SQL y posteriormente ejecutar el método `ExecuteScalar()` que nos devolverá **la primera columna de la primera fila del conjunto de resultados** en este caso el número de registros.

```
public int Count()
{
    string query = "SELECT COUNT(*) FROM libro";
    using var comando = new SqliteCommand(query, Conexion);
    return Convert.ToInt32(comando.ExecuteScalar());
}
```

Uso de un DAO

Recuerda que puedes bajarte el código completo del ejemplo en el **siguiente enlace**:

En el zip con el código del ejemplo, hemos incluido un archivo `libros.json` con una lista de libros para la **carga inicial de la base de datos que haremos solo si está vacía**. En el programa hemos supuesto que está situada en la carpeta `datos` junto a la base de datos.

Definiremos pues en `Program` el método `CargaJson` que leerá el archivo JSON y devolverá una lista de objetos `Libro`.

```
[
  {
    "Id": 1,
    "Titulo": "Macbeth",
    "Autor": "William Shakespeare"
  },
  ...
]
```

```
public static List<Libro> CargaJson(string ruta)
{
    string json = File.ReadAllText(ruta);
    List<Libro>? libros =
    JsonSerializer.Deserialize<List<Libro>>(json);

    return libros ?? [];
}
```

Fíjate que usamos `using` al definir el `LibroDAO` para asegurarnos de que la conexión se cierra correctamente.
Además, Seguimos manteniendo el método `CrearTablaLibros` porque las operaciones **DDL** no tienen sentido que las realice el **DAO**.

```
    public static void Main()
    {
        try
        {
            string rutaEjecucion = RutaEjecucion();
            string rutaDatos = Path.Combine(rutaEjecucion, "datos");
            if (!Directory.Exists(rutaDatos))
                Directory.CreateDirectory(rutaDatos);
            string cadenaConexion = $"Data Source={Path.Combine(rutaDatos, "biblioteca_1t.db")}";
            CrearTablaLibros(cadenaConexion);

12          using LibroDAO libroDAO = new(cadenaConexion);

            if (libroDAO.Count() == 0)
15              CargaJson(Path.Combine(rutaDatos, "libros.json")).ForEach(libro => libroDAO.Create(libro));
            MostrarLibros(libroDAO.Read());

            Console.WriteLine("Creando Hamlet...");
            libroDAO.Create(new Libro(id: 0, titulo: "Hamlet", autor: "Julio Verne"));
            MostrarLibros(libroDAO.Read());
```

```
            Console.WriteLine("Modificando Hamlet...");
            Libro hamlet = libroDAO.Read("Hamlet")!;
            hamlet.Autor = "William Shakespeare";
            libroDAO.Update(hamlet);
            MostrarLibros(libroDAO.Read());

            Console.WriteLine("Borrando Hamlet...");
            libroDAO.Delete(hamlet.Id);
            MostrarLibros(libroDAO.Read());

            Console.WriteLine("Insertando Sueño de una noche de verano...");
            libroDAO.Create(new Libro(id: 0, titulo: "Sueño de una noche de verano", autor: "William Shakespea
            MostrarLibros(libroDAO.Read());

            Console.WriteLine("Consultando libros por autor...");
            MostrarLibrosPorAutor(libroDAO.ReadLibrosPorAutor());
        }
        catch (SqliteException e)
        {
            Console.WriteLine(e.Message);
        }
    }
```