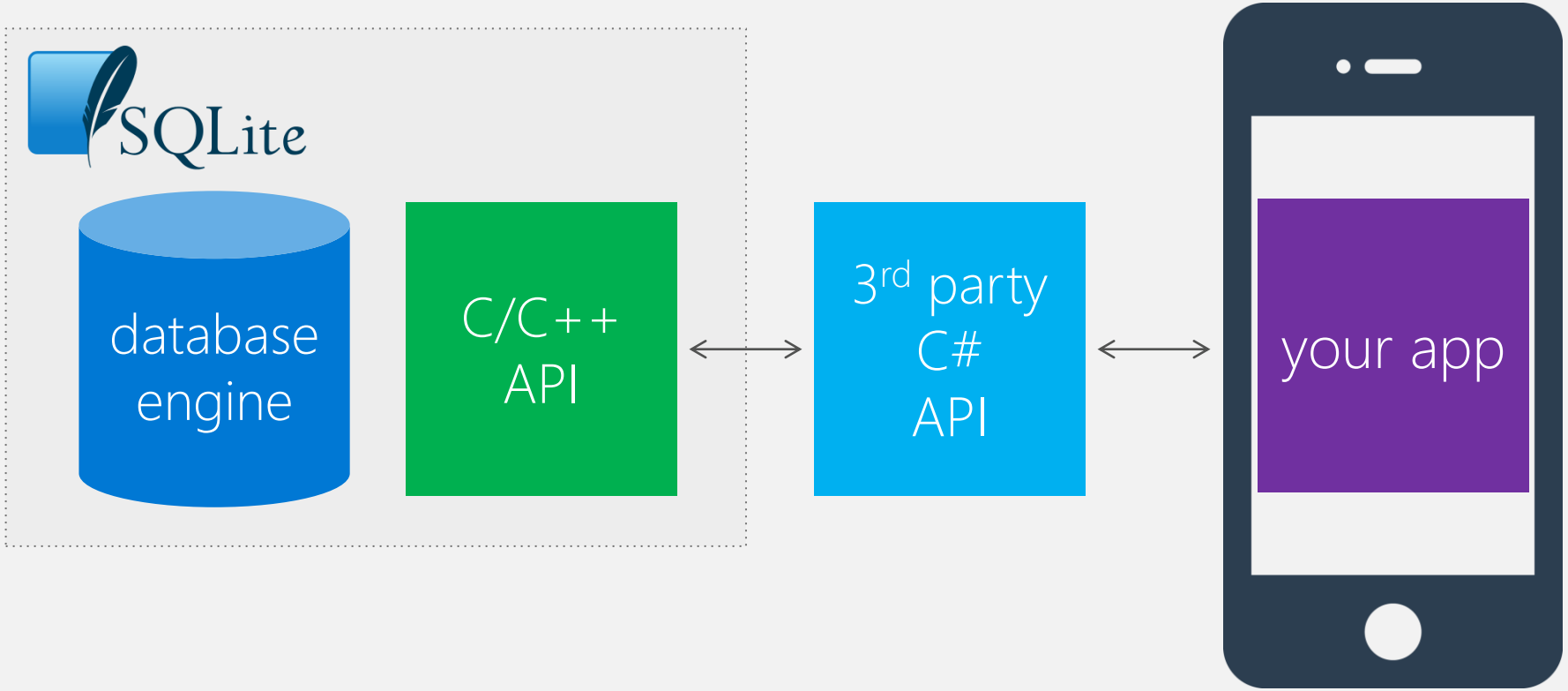
80. Usar SQLite en aplicaciones .NET MAUI

SQLite.NET

El motor de base de datos de SQLite permite que las aplicaciones .NET MAUI carguen y guarden datos en una base de datos local usando código compartido.

SQLite

El motor SQLite expone la API de C/C++ a la que luego accede .NET a través de un wrapper



Instalación del paquete NuGet de SQLite

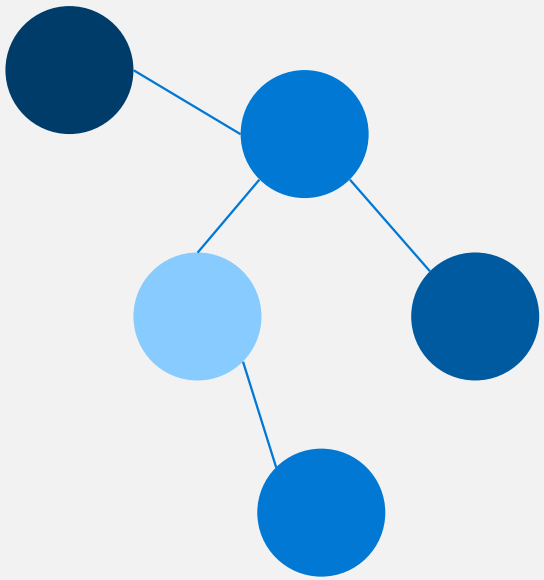
Use el administrador de paquetes NuGet para buscar el paquete **sqlite-net-pcl** y agregar la versión más reciente al proyecto .NET MAUI.

Hay varios paquetes NuGet con nombres similares. El paquete correcto tiene los siguientes atributos:

Id.: sqlite-net-pcl
Autores: SQLite-net
Propietarios: praeclarum
Vínculo de NuGet:sqlite-net-pcl

SQLite.NET

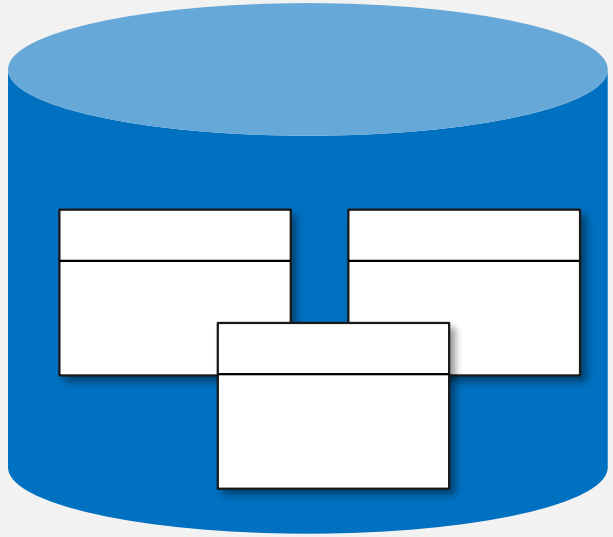
SQLite.NET proporciona un mecanismo para asignar clases a tablas



Objetos en memoria

SQLite.NET

Este tipo de mapeo se conoce como mapeo relacional de objetos u ORM.



Tablas en SQLite

Configuración de SQLite

Los datos de configuración, como el nombre de archivo y la ruta de acceso de la base de datos, se pueden almacenar como constantes en la aplicación.

```
public static class Constants
{
    public const string DatabaseFilename = "TodoSQLite.db3";

    public const SQLite.SQLiteOpenFlags Flags =
        // Abre la base de datos en modo lectura y escritura
        SQLite.SQLiteOpenFlags.ReadWrite |
        // Crea la base de datos si no existe
        SQLite.SQLiteOpenFlags.Create |
        // Habilita el acceso a la base de datos desde multiples hilos
        SQLite.SQLiteOpenFlags.SharedCache;

    public static string DatabasePath =>
        Path.Combine(FileSystem.AppDataDirectory, DatabaseFilename);
}
```

Configuración de SQLite

En el ejemplo anterior, **SQLiteOpenFlag** que se usan para inicializar la conexión de base de datos. La enumeración SQLiteOpenFlag admite estos valores:

- **Create**: la conexión creará automáticamente el archivo de base de datos si no existe.
- **FullMutex**: la conexión se abre en modo de subproceso serializado.
- **NoMutex**: la conexión se abre en modo multiproceso.
- **PrivateCache**: la conexión no participará en la memoria caché compartida, incluso si está habilitada.
- **ReadWrite**: la conexión puede leer y escribir datos.
- **SharedCache**: la conexión participará en la memoria caché compartida, si está habilitada.
- **ProtectionComplete**: el archivo está cifrado y es inaccesible mientras el dispositivo está bloqueado.
- **ProtectionCompleteUnlessOpen**: el archivo se cifra hasta que se abre, pero es accesible incluso si el usuario bloquea el dispositivo.
- **ProtectionCompleteUntilFirstUserAuthentication**: el archivo se cifra hasta después de que el usuario haya arrancado y desbloqueado el dispositivo.
- **ProtectionNone**: el archivo de base de datos no está cifrado.

Acceso a la base de datos

Es habitual al trabajar con SQLite el crear una clase que abstrae la capa de acceso a datos del resto de la aplicación. Esta clase centraliza la lógica de consulta y simplifica la administración de la inicialización de la base de datos, lo que facilita la refactorización o expansión de las operaciones de datos a medida que crece la aplicación. En nuestra aplicación ToDo, vamos a definir una clase TodoItemDatabase para este propósito.

Acceso a la base de datos

TodoItemDatabase usa la inicialización diferida asincrónica para retrasar la inicialización de la base de datos hasta que se accede por primera vez, con un método simple **Init** al que llama cada método de la clase.

```
public class TodoItemDatabase
{
    SQLiteAsyncConnection Database;

    public TodoItemDatabase()
    {
    }

    async Task Init()
    {
        if (Database is not null)
            return;

        Database = new SQLiteAsyncConnection(Constants.DatabasePath, Constants.Flags);
        var result = await Database.CreateTableAsync<TodoItem>();
    }
    ...
}
```

Manipulación de datos

La clase TodoItemDatabase incluye métodos para los cuatro tipos de manipulación de datos: **crear, leer, editar y eliminar**. La biblioteca SQLite.NET proporciona un mapa relacional de objetos simple (ORM) que permite almacenar y recuperar objetos sin escribir instrucciones SQL.

```
public class TodoItemDatabase
{
    ...
    public async Task<List<TodoItem>> GetItemsAsync()
    {
        await Init();
        return await Database.Table<TodoItem>().ToListAsync();
    }

    public async Task<List<TodoItem>> GetItemsNotDoneAsync()
    {
        await Init();
        return await Database.Table<TodoItem>().Where(t => t.Done).ToListAsync();

        // SQL queries are also possible
        //return await Database.QueryAsync<TodoItem>("SELECT * FROM [TodoItem] WHERE [Done] = 0");
    }
```

Manipulación de datos

```
public class TodoItemDatabase
{
    ...

    public async Task<TodoItem> GetItemAsync(int id)
    {
        await Init();
        return await Database.Table<TodoItem>().Where(i => i.ID == id).FirstOrDefaultAsync();
    }

    public async Task<int> SaveItemAsync(TodoItem item)
    {
        await Init();
        if (item.ID != 0)
            return await Database.UpdateAsync(item);
        else
            return await Database.InsertAsync(item);
    }

    public async Task<int> DeleteItemAsync(TodoItem item)
    {
        await Init();
        return await Database.DeleteAsync(item);
    }
}
```

Acceso a los datos

La clase TodoItemDatabase se puede registrar como un singleton que se puede usar en toda la aplicación si usa la inserción de dependencias. Por ejemplo, puede registrar las páginas y la clase de acceso de base de datos como servicios en el IServiceCollection objeto, en MauiProgram.cs, con los métodos **AddSingleton** y **AddTransient** :

```
builder.Services.AddSingleton<TodoListPage>();
builder.Services.AddTransient<TodoItemPage>();

builder.Services.AddSingleton<TodoItemDatabase>();
```

Acceso a los datos

```
TodoItemDatabase database;

public TodoItemPage(TodoItemDatabase todoItemDatabase)
{
    InitializeComponent();
    database = todoItemDatabase;
}

async void OnSaveClicked(object sender, EventArgs e)
{
    if (string.IsNullOrWhiteSpace(Item.Name))
    {
        await DisplayAlert("Name Required", "Please enter a name for the todo item.", "OK");
        return;
    }

    await database.SaveItemAsync(Item);
    await Shell.Current.GoToAsync("..");
}
```

Configuración avanzada

De forma predeterminada, SQLite usa un diario de reversión tradicional. Una copia del contenido de la base de datos sin cambios se escribe en un archivo de reversión independiente y, a continuación, los cambios se escriben directamente en el archivo de base de datos. Commit se produce cuando se elimina el diario de reversión.

Write-Ahead Registro (WAL) escribe primero los cambios en un archivo WAL independiente. En el modo WAL, un COMMIT es un registro especial, anexado al archivo WAL, que permite que se produzcan varias transacciones en un único archivo WAL. Un archivo WAL se combina de nuevo en el archivo de base de datos en una operación especial denominada punto de control.

WAL puede ser más rápido para las bases de datos locales porque los lectores y escritores no se bloquean entre sí, lo que permite que las operaciones de lectura y escritura sean simultáneas. Sin embargo, el modo WAL no permite cambios en el tamaño de página, agrega asociaciones de archivos adicionales a la base de datos y agrega la operación de punto de comprobación adicional.

```
await Database.EnableWriteAheadLoggingAsync();
```