C# SQLite ORM

SQLite Helper: A Micro-ORM for SQLite Database

Code Artist

25 Aug 2024 MIT 9 min read

Rate me: 4.96/5 (10 votes)

23.7K 195 34 8

This project showcases how SQLiteHelper simplifies database interactions, which allows developers to concentrate more on the application's logic rather than the complexities of SQL query writing.

Download latest Repository Archive

Download local copy

SQLite Helper: A Micro-ORM for SQLite Database

nuget v1.13.2 downloads 1.7k

Introduction

SQLite Helper is a micro-Object-Relational Mapping (ORM) tool crafted to facilitate application development with SQLite databases. It is particularly well-suited for small to medium-scale applications, negating the necessity of authoring each SQL query from the ground up.

Conversely, Entity Framework (EF) is a comprehensive ORM offering a complete suite of functionalities. However, more features do not inherently equate to superiority. It is prudent to weigh the Pros and Cons of EF prior to its adoption.

While Entity Framework presents a robust ORM solution with an extensive feature set, SQLite Helper is tailored for simplicity and expediency, enabling streamlined interactions with SQLite databases through straightforward functions.

An article Micro ORM vs ORM written by Alex Shapovalov explained in details difference between Micro ORM vs ORM and how to choose among them.

Dependency

- NuGet: System.Data.SQLite.Core
- .NET Framework 4.8

Key Features of SQLite Helper

SQLite Helper comes with a set of features aimed at making your interaction with SQLite databases as smooth as possible:

1. **Manage Connection String:** `SQLiteHelper` requires only the database file path to establish a connection, streamlining the process significantly.
2. **Automatic Open and Close Connection:** `SQLiteHelper` employs a helper class to manage database connections, obviating the need for manual tracking of connection states and ensuring proper release post-write operations.
3. **Object Mapping to Database Class:** `SQLiteHelper` facilitates single-method read and write operations to the database by mapping objects directly to the database class, thereby simplifying data manipulation tasks.
4. **Handle Queries from Different Database Sources:** `SQLiteHelper` is adept at processing queries from various database sources, offering the requisite flexibility for managing multiple databases.
5. **Utility Functions:** `SQLiteHelper` includes utility methods such as `ClearTable`, `GetPrimaryKeys`, `GetTableSchema`, among others, all designed with safeguards against SQL injection—a common oversight for novices.

In conclusion, `SQLiteHelper` is an essential tool for developers working with SQLite databases. It simplifies database interactions, allowing developers to concentrate more on the application's logic rather than the complexities of SQL query writing. `SQLiteHelper` is effective in improving the development process for both small and medium-sized projects.

Anatomy of SQLiteHelper

- **SQLiteHelper** *(abstract)*: Primary class for the package. It encompasses all methods necessary for database read and write operations.
- **SQLiteDatabaseHandler** *(abstract)*: This is a subclass derived from `SQLiteHelper`. It inherits all features from the SQLiteHelper class and additionally has the ability to toggle between remote and local databases, as well as synchronize data from a remote source to a local cached copy.
- **SQLiteDataReaderEx** *(extension)*: Extension class for `SQLiteDataReader` which handle `null` check for get value method.
- **SQLAttribute**: Attribute base class for table mapping.

Using SQLiteHelper

As a helper class, most methods within `SQLiteHelper` are designated as protected, as it is not anticipated that users of derived classes will interact directly with the database layer. All operations related to the database should remain concealed from the user at both the API and application levels.

Create SQLite Database class

Create project specific database class inherits from `SQLiteHelper` class. `SetSQLPath` have an optional `readOnly` parameter when set to true will open the database file in read only mode.

```
public class MyDatabase : SQLiteHelper
{
    public MyDatabase(string databaseFilePath): base()
    {
        SetSQLPath(databaseFilePath);
    }
}
```

Handling Busy Connection

Proper handling of retry and timeout is crucial to ensure transaction completed successfully without impact user experience. Parameters `SQLStepRetries` and `SQLBusyTimeout` are 2 important parameters to define how many iteration and delay used to retry for busy (locked database) connection.

Basic Query Functions

In SQLite Helper, we offer a suite of query methods that internally handle database connections, eliminating the need to search for unclosed connections that could result in a locked database.

```
protected void ExecuteQuery(string query, Action<SQLiteDataReader> processQueryResults)
protected object ExecuteScalar(string query)
protected int ExecuteNonQuery(string query)
protected void ExecuteTransaction(Action performTransactions)
```

Example below show usage of `ExecuteQuery` method. Connection are closed and `SQLiteDataReader` object `r` is disposed at end of `ExecuteQuery` method.

```
ExecuteQuery(query, (r)=>
{
    while(r.Read())
    {
        //ToDo: Perform readback here...
    }
});
```

Read from Database Table

SQLite Helper offer 2 solutions to read data from Employee table into `Employee` class object as shown below:

```
Employee (Table)
  |- ID, INTEGER, Primary Key
  |- Name, TEXT
  |- Department, TEXT
  |- Salary, INTEGER
```

```
public class Employee
{
    public int ID {get; set;}
    public string Name {get; set;}
    public string Department {get; set;}
    public int Salary {get; set;}
}
```

1. The convention way, manually read data from database using `ExecuteQuery` method:

```
public Employee[] ReadEmployeeData()
{
    List<Employee> results = new List<Employee>();
    //Execute Query handle database connection
    ExecuteQuery("SELECT * FROM Employee", (r) =>
    {
        //(r) = Delegate call back function with SQLiteDataReader parameter r.
        //Disposal of r is taken care by ExecuteQuery method.
        int x;
        while(r.Read())
        {
            x = 0;
            Employee e = new Employee();
            e.ID = r.GetInt32(x++);
            e.Name = r.GetStringEx(x++);  //Extension method. Handle null value.
            e.Department = r.GetStringEx(x++);
            e.Salary = r.GetInt32Ex(x++);z
        }
    });
}
```

2. Implementation above can be further simplify using query with class - `ReadFromDatabase`

```
public Employee[] ReadEmployeeData()
{
    return ReadFromDatabase<Employee>().ToArray();
}
```

Write Data to Database

To update or write new data into database, you can utilize the `WriteToDatabase` method. Although it is possible perform the same action using `ExecuteNonQuery` method for simple table structure, `WriteToDatabase` method capable to handle more complex database structure which described in following section.

```
public void WriteEmployeeData(Employee[] newDatas)
{
    WriteToDatabase(newDatas);
}
```

Reading and Writing Complex Tables (ORM)

The `ReadFromDatabase` and `WriteToDatabase` methods make it easy to link objects in your code to tables in your database. They work well with tables that have relationships (child tables) and can handle working with more than one database using simple commands. Let's take a closer look at what they can do.

These methods follow the Fail Fast Principle, which means they quickly check if the structure of your objects matches the structure of your database tables when you first use them. This check is to make sure that all the columns match up. To avoid problems with older versions, your database tables can have extra columns that aren't in your objects, but not the other way around.

SQLite Write Option

The 'WriteOptions' properties specify by `SQLiteWriteOption` class, sets how SQLiteHelper behaves when reading and writing data with following options:

- `CreateTable`: Automatic create table in database if not exists when set to true. Do nothing if table already exists.
- `WriteMode`: Used by `WriteToDatabase` method to decide what to update. (Reserved for future implemntation, not available yet)

Table Name

Mapping a class to database table named *Employee*. Use `SQLName` attribute to overwrite default table name.

```
public class Employee { ... }

[SQLName(<span class="pl-s">"Employee")</span>]
public class Emp { ... }
```

Column Name

All public properties that have public getters and setters are regarded as SQL columns. The names of these properties are, by default, used as the names of the corresponding columns. The `SQLName` attribute can be used to overwrite the default column name or table name.

```
public class Employee
{
    //Database Column: Name = 'Name', Type = TEXT
    public string Name {get; set;}

    //Database Column: Name = 'Department', Type = TEXT
    [SQLName(<span class="pl-s">"Department")</span>]
    public string Dept {get; set;}

    //Database Column: Name = 'Salary', Type = INTEGER
    public int Salary {get; set;}

    //Database Column: Name = 'Cost', Type = NUMERIC
    public double Cost {get; set;}

    //Read only property is not a valid SQL Column
    public int Age {get;}
}
```

Data Type

The table below displays the default data type mappings between objects and the database. Ensuring matching data types is crucial for the accurate writing and reading of data. **NOTE**: SQLite may automatically convert values to the appropriate datatype. More details in SQLite documentation Type Affinity

Object Type	Database Type
string, Enum, DateTime	TEXT
int, long, bool	INTEGER
double, decimal, float	REAL

The `SQLDataType` attribute can be utilized to explicitly define the storage type of a value in the database. For instance, `Enum` and `DateTime` types can be stored as integers by applying the `SQLDataType` attribute, with Enum `Status` being stored as an integer and `DateTime` being stored as ticks (long).

```
public enum Status { ... }
public class MyTable
{
    [SQLDataType(DataType.INTEGER)]
    public Status CurrentStatus {get; set;}

    [SQLDataType(DataType.INTEGER)]
    public DateTime LastUpdate {get; set;}
}
```

Index Table

The example below demonstrates that `UserName` is stored as an index in the `NameID` column of the `Employee` table, while the actual string value is kept in a key-value pair table named `Name`. This method facilitates efficient data retrieval and management, particularly when the same name is used multiple times across different tables.

Table name parameter for `SQLIndexTable` is optional. If left blank, the property name `UserName` will be used as the table name. The values for the index table can be shared among multiple tables.

```
public class Employee
{
    [SQLIndexTable(<span class="pl-s">"Name")</span>]
    [SQLName(<span class="pl-s">"NameID")</span>]
    public string UserName {get; set;}
}
```

```
Employee (Table)
   |- NameID, INTEGER
   | ...

Name (Table)
   |- ID, INTEGER, Primary Key
   |- Name, TEXT, Unique
```

Primary Key

The primary key attribute is linked to the primary key in the database table. When the `WriteToDatabase` method is executed with an item whose ID is 0, it will create a new entry in the database table and assign it a unique ID. If the ID is not 0, it will update the existing row with the matching ID. **NOTE:** Primary key must be declared with `int` type.

```
public class Employee
{
    [PrimaryKey]
    public int ID {get; set;}
    ...
}
```

Parent and Child Tables

Let's examine the example provided: In database, `Department` (Table Name: Department) serves as a parent table, and `List<Employee>` (Table Name: Employees) functions as a child table with a one-to-many relationship, where each department can be associated with multiple employees. In other words, for every single entry in the `Department` table, there can be several corresponding entries in the

Employee table, each representing an individual employee belonging to that department, while each Employee is assigned to only one Department.

Child table must have a properties ID declared with ParentKey attribute which function as mapping between child and parent table. Value of DepartmentID in example below is assigned by SQLite Helper. PrimaryKey for class class Department is mandatory while it is optional for class Employee depends on need of the design.

A child table must have an ID property, decorated with ParentKey attribute, which serves as the link between child and parent table. In the example below, parent key value DepartmentID is assigned by SQLite Helper.

```
public class Department
{
    [PrimaryKey]
    public int ID { get; set; }
    public string Name { get; set; }
    public List<Employee> Employees { get; set; } = new List<Employee>();
    ...
}

public class Employee
{
    public string Name { get; set; }
    [ParentKey(typeof(Department))]
    public int DepartmentID { get; set; }
    ...
}
```

Equivalent database table are given as follow:

```
Department (Table)
  |- ID, INTEGER, Primary Key
  |- Name, TEXT

Employee (Table)
  |- Name, TEXT
  |- DepartmentID, INTEGER
```

Multiple Database Source

SQLite Helper also support multiple database source, allow data to be read and write from tables stored in different SQLite database files. Example below showing that Department table is stored in main database while Employee table is table stored in **Employee.db**. Switching between main and sub database are handled internally by read and write method. This SQLDatabase attribute can only be used with child table.

```
public class Department
{
    ...
    [SQLName(<span class="pl-s">"Employee")</span>]
    [SQLDatabase(<span class="pl-s">"Employee.db")</span>]
    public List<Employee> Employees { get; set; } = new List<Employee>();
}
```

Array Table

The array table functionality allows the storage of array properties from a sample table, `TableWithArray`, into separate SQLite tables. This process involves creating specific SQLite tables for each type of array property, enabling efficient storage and retrieval of array data. The following example demonstrates how to map the array properties into SQLite tables.

```
public class TableWithArray
{
    [PrimaryKey]
    public int ID { get; set; }

    ...

    public string[] ArrayData { get; set; }

    [SQLName(<span class="pl-s">"ArrayIntValue")</span>]
    public int[] ItemValue { get; set; }
}
```

`ArrayData`is a string array which mapped into ArrayTable with TEXT value.

```
ArrayData (Table)
  |- ID, INTEGER
  |- Value, TEXT
```

`ItemValue` is an integer array, mapped to the SQLite table `ArrayIntValue` using the SQLName attribute.

```
ArrayIntValue (Table)
  |- ID, INTEGER
  |- Value, INTEGER
```

Unique Constraint

The `SQLUnique` and `SQLUniqueMultiColumn` attributes are used to mark columns with a unique constraint. `SQLUnique` sets a unique constraint on a single column, while `SQLUniqueMultiColumn` sets unique constraints across multiple columns.

Example usage of these attributes as follow:

```
public class User
{
    [PrimaryKey]
    public int ID { get; set; }

    [SQLUnique]
    public string Email { get; set; }

    public string FirstName { get; set; }
    public string LastName { get; set; }

    [SQLUniqueMultiColumn]
    public string Username { get; set; }

    [SQLUniqueMultiColumn]
    public string PhoneNumber { get; set; }
}
```

SQL Table Structure

```
User (Table)
  |- ID, INTEGER, Primary Key
  |- Email, TEXT, Unique
  |- FirstName, TEXT
  |- LastName, TEXT
  |- Username, TEXT
  |- PhoneNumber, TEXT
     (Unique Username, PhoneNumber)
```

SQL Schema

```
CREATE TABLE User (
    Id INTEGER PRIMARY KEY AUTOINCREMENT,
    Email TEXT UNIQUE,
    FirstName TEXT,
    LastName TEXT,
    Username TEXT,
    PhoneNumber TEXT,
    UNIQUE (Username, PhoneNumber)
);
```

In this example:

The `Email` column is marked with the `SQLUnique` attribute, ensuring that each email address is unique. The `Username` and `PhoneNumber` columns are marked with the `SQLUniqueMultiColumn` attribute, ensuring that the combination of `Username` and `PhoneNumber` is unique across the table.

Suggestion and Feedback

We hope this document has provided you with clear and helpful information to use this tool. Your feedback is invaluable to us as it helps improve the quality of our work and clarity of our

documentation. Please share your suggestions, comments, or any difficulties you encountered while using this guide. Your input will assist us in enhancing our resources and supporting users like you more effectively. Thank you for your attention and contribution.
This article was originally posted at https://github.com/Code-Artist/CodeArtEng.SQLite

License

Written By
Code Artist
Technical Lead
Malaysia

Official Page: www.codearteng.com

Watch

Comments and Discussions

Add a Comment or Question ? Email Alerts Search Comments

Spacing Relaxed Layout Normal Per page 25 Update

First Prev Next

Some questions about the code snippets	**star__duster**	**21-Jul-24 20:59**
Re: Some questions about the code snippets	Code Artist	27-Jul-24 9:51
My vote of 5	**Ștefan-Mihai MOGA**	**25-Jun-24 0:07**
Re: My vote of 5	Code Artist	27-Jun-24 20:45
Why not .net core as dependency	**Jean Petrovic**	**18-Jun-24 20:36**
Re: Why not .net core as dependency	Code Artist	18-Jun-24 20:46
Re: Why not .net core as dependency	Raúl Duque	19-Jul-24 2:49

Re: Why not .net core as dependency   Code Artist   27-Jul-24 9:53

Last Visit: 21-Aug-24 19:37   Last Update: 26-Aug-24 16:42   Refresh   1

General News Suggestion Question Bug Answer Joke Praise Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.