

---

## SQLitePCLRaw

SQLitePCLRaw is a Portable Class Library (PCL) for low-level (raw) access to SQLite. License: Apache License v2.

### Version 2.0

SQLitePCLRaw 2.0 is a major release. See the release notes for more information.

### Compatibility

As of version 2.0, SQLitePCLRaw requires NetStandard2.0:

- Xamarin.Android
- Xamarin.iOS
- UWP
- .NET Framework 4.6.1 or higher, preferably 4.7.2
- Linux with Mono
- MacOS with Mono
- .NET Core 3.1, .NET 5.x and up, etc
- NetStandard 2.0

### How the packaging works

The main assembly is SQLitePCLRaw.core. A portable library project would need to only take a dep on this one. All the other packages deal with initialization and the question of which instance of the native SQLite library is involved.

### Many different native SQLite libraries

In some cases, apps use a SQLite library which is externally provided. In other cases, an instance of the SQLite library is bundled with the app.

- On iOS, there is a SQLite library provided with the operating system, and apps are allowed to use it.

- 
- Android also has a SQLite library, and prior to Android N, apps were allowed to use it.
  - Recent versions of Windows 10 have a SQLite library.
  - In some cases, people want to use SQLCipher as their SQLite library.
  - Sometimes people want to compile and bundle their own custom SQLite library.

SQLitePCLRaw supports any of these cases.

## Providers

In this context, a “provider” is the piece of code which tells SQLitePCLRaw which instance of the native code to use.

More specifically, a “provider” is an implementation of the ISQLite3Provider interface. It is necessary to call SQLitePCL.raw.SetProvider() to initialize things.

The SQLitePCLRaw.core package contains no providers.

All the various providers are in packages with ids of the form SQLitePCLRaw.provider.\*.

## Provider names

There is a `dynamic` provider which does not use a hard-coded DllImport string. This one is used as often as possible.

The DllImport-based providers are named for the exact string which is used for DllImport (pinvoke).

For example:

```
1 [DllImport("foo")]
2 public static extern int whatever();
```

This pinvoke will look for a library called “foo”.

- On Windows, that means “foo.dll”.
- On Unix, “libfoo.so”
- On MacOS, “libfoo.dylib”

(The actual rules are more complicated than this.)

So, a provider where all the DllImport attributes were using “foo”, would have “foo” in its package id and in its class name.

---

## Included providers

SQLitePCLRaw includes the following providers:

- “dynamic” – Uses dynamic loading of the native library instead of DllImport attributes.
- “e\_sqlite3” – This is the name of all SQLite builds provided as part of this project.
- “e\_sqlcipher” – This is the name of the unofficial and unsupported SQLCipher builds which are provided as part of this project.
- “sqlite3” – This matches the name of the system-provided SQLite on iOS (which is fine), and Android (which is not allowed). And it matches the official name of builds provided at [sqlite.org](http://sqlite.org).
- “sqlcipher” – Intended to be used for official SQLCipher builds from Zetetic.
- “winsqlite3” – Matches the name of the library provided by recent builds of Windows 10.

## SQLitePCLRaw.lib

A provider is the bridge between the core assembly and the native code, but the provider does not contain the native code itself.

In some cases (like “winsqlite3”) this is because it does not need to. The provider is merely a bridge to a SQLite library instance which is known (or assumed) to be somewhere else.

But in cases where the app is going to be bundling the native code library, those bits need to make it into your build output somehow.

Packages with ids named “SQLitePCLRaw.lib.\*” contain native code. This project distributes two kinds of these packages:

- “e\_sqlite3” – These are builds of the SQLite library provided for the convenience of SQLitePCLRaw users. I try to keep them reasonably current with respect to SQLite itself ([www.sqlite.org](http://www.sqlite.org)). The build configuration is the same for every platform, and includes full-text-search. If you are building an app on multiple platforms and you want to use the same recent version of SQLite on each platform, e\_sqlite3 should be a good choice.
- “e\_sqlcipher” – These are unofficial and unsupported builds of the open source SQLCipher code.

The build scripts for both of the above are in the [ericsink/cb](https://github.com/ericsink/cb) repo.

---

## A trio of packages

So, using SQLitePCLRaw means you need to add two packages:

- SQLitePCLRaw.core
- SQLitePCLRaw.provider.whatever

And in many cases one of these as well:

- SQLitePCLRaw.lib.whatever

And in your platform-specific code, you need to call:

```
1 SQLitePCL.raw.SetProvider(new SQLitePCL.SQLite3Provider_whatever());
```

But the word “whatever” is different on each platform. For example, on Android, using e\_sqlite3, you need:

- SQLitePCLRaw.core
- SQLitePCLRaw.provider.e\_sqlite3.android
- SQLitePCLRaw.lib.e\_sqlite3.android

and you need to call:

```
1 SQLitePCL.raw.SetProvider(new SQLitePCL.SQLite3Provider_e_sqlite3());
```

## Bundles

To make things easier, SQLitePCLRaw includes “bundle” packages. These packages automatically bring in the right dependencies for each platform. They also provide a single Init() call that is the same for all platforms.

Think of a bundle as way of giving a “batteries included” experience.

So for example, SQLitePCLRaw.bundle\_e\_sqlite3 is a bundle that uses e\_sqlite3 in all cases. Just add this package, and call:

```
1 SQLitePCL.Batteries_V2.Init();
```

SQLitePCLRaw.bundle\_green is a bundle that uses e\_sqlite3 everywhere except iOS, where the system-provided SQLite is used.

The purpose of the bundles is to make things easier by taking away flexibility and control. You don’t have to use them.

---

## How do I build this?

### Requirements

- Install the `t4` cli tool with `dotnet tool install --global dotnet-t4`
- Clone the `cb` repository in the same directory as you cloned this `SQLitePCL.raw` repository
- Make sure that the *Mobile development with .NET* workload is installed

Then, from a Developer Command Prompt for Visual Studio 2017 or 2019:

```
1 cd build
2 dotnet run
```

### Can this library be used to write a mobile app?

Technically, yes, but that's not what you want to do. This is *not* the sort of SQLite library you would use to write an app. It is a very thin C# wrapper around the C API for SQLite. It's "raw".

Consequently, as much as possible, this library follows the stylistic conventions of SQLite, not those of the .NET/C# world.

For example, the C function for opening a SQLite file is `sqlite3\_open()`, so this API provides a method called `sqlite3\_open()`, not `SQLite3Open()`.

Similarly, the functions in this API return integer error codes rather than throwing .NET exceptions, because that's how the SQLite C API works.

As a library for app developers, this library is downright hostile. It feels like using C. Intentionally.

### So if this library is so unfriendly, why does it exist at all?

This library is designed to be the common portable layer upon which friendlier wrappers can be built. Before this existed, every C# SQLite library was writing their own P/Invoke and COM and marshaling and stuff. Building on this library instead allows folks to focus more on the upper layer and its goal of providing a pleasant, easy-to-use API for app developers.

### How does this compare to Microsoft.Data.Sqlite?

Microsoft.Data.Sqlite is an ADO.NET-style SQLite wrapper which is part of Entity Framework Core. It uses SQLitePCLRaw.

---

## How does this compare to sqlite-net?

sqlite-net is a very popular SQLite wrapper by Frank Krueger (@praeclarum). Unlike SQLitePCLRaw, it is designed to make writing apps easier. It even includes a lightweight ORM, and some basic support for LINQ.

The `sqlite-net-pcl` package uses SQLitePCLRaw:

<https://www.nuget.org/packages/sqlite-net-pcl/>

## How does this compare to System.Data.SQLite?

System.Data.SQLite is an ADO.NET-style SQLite wrapper developed by the core SQLite team. It is very full-featured, supporting LINQ and Entity Framework. And for obvious reasons, it does a fantastic job of the SQLite side of things. But it is not at all mobile-friendly.

## Why is this called SQLitePCLRaw?

SQLitePCL was a SQLite Portable Class Library released on Codeplex by MS Open Tech.

This library is a fork of that code. Sort of.

It is a fork in the 2007 sense of the word. I made significant use of the code. I preserved copyright notices.

However, this is not the the sort of fork which is created for the purpose of producing a pull request. The changes I've made are so extensive that I do not plan to submit a pull request unless one is requested. I plan to maintain this code going forward.

## What is SQLitePCL.Ugly?

Well, it's a bunch of extension methods, a layer that provides method call syntax. It also switches the error handling model from integer return codes to exception throwing.

For example, the `sqlite3_stmt` class represents a statement handle, but you still have to do things like this:

```
1 int rc;
2
3 sqlite3 db;
4 rc = raw.sqlite3_open(":memory:", out db);
5 if (rc != raw.SQLITE_OK)
6 {
```

---

```
7     error
8 }
9 sqlite3_stmt stmt;
10 rc = raw.sqlite3_prepare(db, "CREATE TABLE foo (x int)", out stmt);
11 if (rc != raw.SQLITE_OK)
12 {
13     error
14 }
15 rc = raw.sqlite3_step(stmt);
16 if (rc == raw.SQLITE_DONE)
17 {
18     whatever
19 }
20 else
21 {
22     error
23 }
24 raw.sqlite3_finalize(stmt);
```

The Ugly layer allows me to do things like this:

```
1 using (sqlite3 db = ugly.open(":memory:"))
2 {
3     sqlite3_stmt stmt = db.prepare("CREATE TABLE foo (x int)");
4     stmt.step();
5 }
```

This exception-throwing wrapper exists so that I can have something easier against which to write tests. It retains all the “lower-case and underscores” ugliness of the layer(s) below.

It does not do things “The C# Way”. As such, this is not a wrapper intended for public consumption.