ccgus / **fmdb**

A Cocoa / Objective-C wrapper around SQLite

513 commits | **10** branches | **16** releases | **57** contributors

Branch: **master** | **New pull request** | **Find file** | **Clone or download**

ccgus Fixed some build errors and warnings. Latest commit `458031a` Jun 13, 2017

Tests	Version 2.7.2	Jun 10, 2017
fmdb.xcodeproj	Add doc comments warnings	Jun 3, 2017
src	Fixed some build errors and warnings.	Jun 13, 2017
.gitignore	Merged in Chris Wright's date format additions to FMDatabase.	May 24, 2013
.travis.yml	Add a .travis.yml file to use Travis CI for automated testing.	Nov 24, 2013
CHANGES_AND_TODO_LIST.txt	Version 2.7.2	Jun 10, 2017
CONTRIBUTORS.txt	Update contrib stuff.	Apr 14, 2014
FMDB.podspec	Version 2.7.2	Jun 10, 2017
LICENSE.txt	Updated license.	Oct 20, 2014
README.markdown	Update documentation	Jun 10, 2017
fmdb.1	Initial import.	Jul 6, 2010

README.markdown

FMDB v2.7

pod v2.7.2 | Carthage compatible

This is an Objective-C wrapper around SQLite: http://sqlite.org/

The FMDB Mailing List:

http://groups.google.com/group/fmdb

Read the SQLite FAQ:

http://www.sqlite.org/faq.html

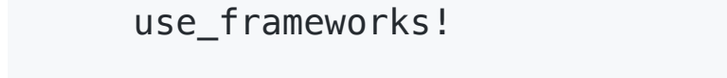
Since FMDB is built on top of SQLite, you're going to want to read this page top to bottom at least once. And while you're there, make sure to bookmark the SQLite Documentation page: http://www.sqlite.org/docs.html

Contributing

Do you have an awesome idea that deserves to be in FMDB? You might consider pinging ccgus first to make sure he hasn't already ruled it out for some reason. Otherwise pull requests are great, and make sure you stick to the local coding conventions. However, please be patient and if you haven't heard anything from ccgus for a week or more, you might want to send a note asking what's up.

Installing

CocoaPods



FMDB can be installed using CocoaPods.

If you haven't done so already, you might want to initialize the project, to have it produce a `Podfile` template for you:

```
$ pod init
```

Then, edit the `Podfile`, adding `FMDB`:

```
# Uncomment the next line to define a global platform for your project
# platform :ios, '9.0'

target 'MyApp' do
    # Comment the next line if you're not using Swift and don't want to use dynamic frameworks
    use_frameworks!

    # Pods for MyApp2

    pod 'FMDB'
    # pod 'FMDB/FTS'   # FMDB with FTS
    # pod 'FMDB/standalone'   # FMDB with latest SQLite amalgamation source
    # pod 'FMDB/standalone/FTS'   # FMDB with latest SQLite amalgamation source and FTS
    # pod 'FMDB/SQLCipher'   # FMDB with SQLCipher
end
```

Then install the pods:

```
$ pod install
```

Then open the `.xcworkspace` rather than the `.xcodeproj`.

For more information on Cocoapods visit https://cocoapods.org.

If using FMDB with SQLCipher you must use the FMDB/SQLCipher subspec. The FMDB/SQLCipher subspec declares SQLCipher as a dependency, allowing FMDB to be compiled with the `-DSQLITE_HAS_CODEC` flag.

Carthage

Once you make sure you have the latest version of Carthage, you can open up a command line terminal, navigate to your project's main directory, and then do the following commands:

```
$ echo ' github "ccgus/fmdb" ' > ./Cartfile
$ carthage update
```

You can then configure your project as outlined in Carthage's Getting Started (i.e. for iOS, adding the framework to the "Link Binary with Libraries" in your target and adding the `copy-frameworks` script; in macOS, adding the framework to the list of "Embedded Binaries").

FMDB Class Reference:

http://ccgus.github.io/fmdb/html/index.html

Automatic Reference Counting (ARC) or Manual Memory Management?

You can use either style in your Cocoa project. FMDB will figure out which you are using at compile time and do the right thing.

What's New in FMDB 2.7

FMDB 2.7 attempts to support a more natural interface. This represents a fairly significant change for Swift developers (audited for nullability; shifted to properties in external interfaces where possible rather than methods; etc.). For Objective-C developers, this should be a fairly seamless transition (unless you were using the ivars that were previously exposed in the public interface, which you shouldn't have been doing, anyway!).

Nullability and Swift Optionals

FMDB 2.7 is largely the same as prior versions, but has been audited for nullability. For Objective-C users, this simply means that if you perform a static analysis of your FMDB-based project, you may receive more meaningful warnings as you review your project, but there are likely to be few, if any, changes necessary in your code.

For Swift users, this nullability audit results in changes that are not entirely backward compatible with FMDB 2.6, but is a little more Swifty. Before FMDB was audited for nullability, Swift was forced to defensively assume that variables were optional, but the library now more accurately knows which properties and method parameters are optional, and which are not.

This means, though, that Swift code written for FMDB 2.7 may require changes. For example, consider the following Swift 3/Swift 4 code for FMDB 2.6:

```
guard let queue = FMDatabaseQueue(path: fileURL.path) else {
    print("Unable to create FMDatabaseQueue")
    return
}

queue.inTransaction { db, rollback in
    do {
        guard let db == db else {
            // handle error here
            return
        }

        try db.executeUpdate("INSERT INTO foo (bar) VALUES (?)", values: [1])
        try db.executeUpdate("INSERT INTO foo (bar) VALUES (?)", values: [2])
    } catch {
        rollback?.pointee = true
    }
}
```

Because FMDB 2.6 was not audited for nullability, Swift inferred that `db` and `rollback` were optionals. But, now, in FMDB 2.7, Swift now knows that, for example, neither `db` nor `rollback` above can be `nil`, so they are no longer optionals. Thus it becomes:

```
let queue = FMDatabaseQueue(url: fileURL)

queue.inTransaction { db, rollback in
    do {
        try db.executeUpdate("INSERT INTO foo (bar) VALUES (?)", values: [1])
        try db.executeUpdate("INSERT INTO foo (bar) VALUES (?)", values: [2])
    } catch {
        rollback.pointee = true
    }
}
```

Custom Functions

In the past, when writing custom functions, you would have to generally include your own `@autoreleasepool` block to avoid problems when writing functions that scanned through a large table. Now, FMDB will automatically wrap it in an autorelease pool, so you don't have to.

Also, in the past, when retrieving the values passed to the function, you had to drop down to the SQLite C API and include your own `sqlite3_value_XXX` calls. There are now `FMDatabase` methods, `valueInt`, `valueString`, etc., so you can stay within Swift and/or Objective-C, without needing to call the C functions yourself. Likewise, when specifying the return values, you no longer need to call `sqlite3_result_XXX` C API, but rather you can use `FMDatabase` methods, `resultInt`, `resultString`, etc. There is a new `enum` for `valueType` called `SqliteValueType`, which can be used for checking the type of parameter passed to the custom function.

Thus, you can do something like (as of Swift 3):

```
db.makeFunctionNamed("RemoveDiacritics", arguments: 1) { context, argc, argv in
    guard db.valueType(argv[0]) == .text || db.valueType(argv[0]) == .null else {
        db.resultError("Expected string parameter", context: context)
        return
    }

    if let string = db.valueString(argv[0])?.folding(options: .diacriticInsensitive, locale: nil) {
        db.resultString(string, context: context)
    } else {
        db.resultNull(context: context)
    }
}
```

And you can then use that function in your SQL (in this case, matching both "Jose" and "José"):

```
SELECT * FROM employees WHERE RemoveDiacritics(first_name) LIKE 'jose'
```

Note, the method `makeFunctionNamed:maximumArguments:withBlock:` has been renamed to `makeFunctionNamed:arguments:block:`, to more accurately reflect the functional intent of the second parameter.

API Changes

In addition to the `makeFunctionNamed` noted above, there are a few other API changes. Specifically,

- To become consistent with the rest of the API, the methods `objectForColumnName` and `UTF8StringForColumnName` have been renamed to `objectForColumn` and `UTF8StringForColumn`.
- Note, the `objectForColumn` (and the associted subscript operator) now returns `nil` if an invalid column name/index is passed to it. It used to return `NSNull`.
- To avoid confusion with `FMDatabaseQueue` method `inTransaction`, which performs transactions, the `FMDatabase` method to determine whether you are in a transaction or not, `inTransaction`, has been replaced with a read-only property, `isInTransaction`.
- Several functions have been converted to properties, namely, `databasePath`, `maxBusyRetryTimeInterval`, `shouldCacheStatements`, `sqliteHandle`, `hasOpenResultSets`, `lastInsertRowId`, `changes`, `goodConnection`, `columnCount`, `resultDictionary`, `applicationID`, `applicationIDString`, `userVersion`, `countOfCheckedInDatabases`, `countOfCheckedOutDatabases`, and `countOfOpenDatabases`. For Objective-C users, this has little material impact, but for Swift users, it results in a slightly more natural interface. Note: For Objective-C developers, previously versions of FMDB exposed many ivars (but we hope you weren't using them directly, anyway!), but the implmentation details for these are no longer exposed.

URL Methods

In keeping with Apple's shift from paths to URLs, there are now `NSURL` renditions of the various `init` methods, previously only accepting paths.

Usage

There are three main classes in FMDB:

1. `FMDatabase` - Represents a single SQLite database. Used for executing SQL statements.
2. `FMResultSet` - Represents the results of executing a query on an `FMDatabase`.
3. `FMDatabaseQueue` - If you're wanting to perform queries and updates on multiple threads, you'll want to use this class. It's described in the "Thread Safety" section below.

Database Creation

An `FMDatabase` is created with a path to a SQLite database file. This path can be one of these three:

1. A file system path. The file does not have to exist on disk. If it does not exist, it is created for you.
2. An empty string (`@""`). An empty database is created at a temporary location. This database is deleted with the `FMDatabase` connection is closed.
3. `NULL`. An in-memory database is created. This database will be destroyed with the `FMDatabase` connection is closed.

(For more information on temporary and in-memory databases, read the sqlite documentation on the subject: http://www.sqlite.org/inmemorydb.html)

```
NSString *path = [NSTemporaryDirectory() stringByAppendingPathComponent:@"tmp.db"];
FMDatabase *db = [FMDatabase databaseWithPath:path];
```

Opening

Before you can interact with the database, it must be opened. Opening fails if there are insufficient resources or permissions to open and/or create the database.

```
if (![db open]) {
    // [db release];   // uncomment this line in manual referencing code; in ARC, this is not necessary,
    db = nil;
    return;
}
```

Executing Updates

Any sort of SQL statement which is not a `SELECT` statement qualifies as an update. This includes `CREATE`, `UPDATE`, `INSERT`, `ALTER`, `COMMIT`, `BEGIN`, `DETACH`, `DELETE`, `DROP`, `END`, `EXPLAIN`, `VACUUM`, and `REPLACE` statements (plus many more). Basically, if your SQL statement does not begin with `SELECT`, it is an update statement.

Executing updates returns a single value, a `BOOL`. A return value of `YES` means the update was successfully executed, and a return value of `NO` means that some error was encountered. You may invoke the `-lastErrorMessage` and `-lastErrorCode` methods to retrieve more information.

Executing Queries

A `SELECT` statement is a query and is executed via one of the `-executeQuery...` methods.

Executing queries returns an `FMResultSet` object if successful, and `nil` upon failure. You should use the `-lastErrorMessage` and `-lastErrorCode` methods to determine why a query failed.

In order to iterate through the results of your query, you use a `while()` loop. You also need to "step" from one record to the other. With FMDB, the easiest way to do that is like this:

```
FMResultSet *s = [db executeQuery:@"SELECT * FROM myTable"];
while ([s next]) {
    //retrieve values for each record
}
```

You must always invoke `-[FMResultSet next]` before attempting to access the values returned in a query, even if you're only expecting one:

```
FMResultSet *s = [db executeQuery:@"SELECT COUNT(*) FROM myTable"];
if ([s next]) {
    int totalCount = [s intForColumnIndex:0];
}
```

`FMResultSet` has many methods to retrieve data in an appropriate format:

- `intForColumn:`
- `longForColumn:`
- `longLongIntForColumn:`
- `boolForColumn:`
- `doubleForColumn:`
- `stringForColumn:`
- `dateForColumn:`
- `dataForColumn:`
- `dataNoCopyForColumn:`
- `UTF8StringForColumn:`
- `objectForColumn:`

Each of these methods also has a `{type}ForColumnIndex:` variant that is used to retrieve the data based on the position of the column in the results, as opposed to the column's name.

Typically, there's no need to `-close` an `FMResultSet` yourself, since that happens when either the result set is deallocated, or the parent database is closed.

Closing

When you have finished executing queries and updates on the database, you should `-close` the `FMDatabase` connection so that SQLite will relinquish any resources it has acquired during the course of its operation.

```
[db close];
```

Transactions

`FMDatabase` can begin and commit a transaction by invoking one of the appropriate methods or executing a begin/end transaction statement.

Multiple Statements and Batch Stuff

You can use `FMDatabase`'s executeStatements:withResultBlock: to do multiple statements in a string:

```
NSString *sql = @"create table bulktest1 (id integer primary key autoincrement, x text);"
                "create table bulktest2 (id integer primary key autoincrement, y text);"
                "create table bulktest3 (id integer primary key autoincrement, z text);"
                "insert into bulktest1 (x) values ('XXX');"
                "insert into bulktest2 (y) values ('YYY');"
                "insert into bulktest3 (z) values ('ZZZ');";

success = [db executeStatements:sql];

sql = @"select count(*) as count from bulktest1;"
      "select count(*) as count from bulktest2;"
      "select count(*) as count from bulktest3;";

success = [self.db executeStatements:sql withResultBlock:^int(NSDictionary *dictionary) {
    NSInteger count = [dictionary[@"count"] integerValue];
    XCTAssertEqual(count, 1, @"expected one record for dictionary %@", dictionary);
    return 0;
```

```
}];
```

Data Sanitization

When providing a SQL statement to FMDB, you should not attempt to "sanitize" any values before insertion. Instead, you should use the standard SQLite binding syntax:

```
INSERT INTO myTable VALUES (?, ?, ?, ?)
```

The `?` character is recognized by SQLite as a placeholder for a value to be inserted. The execution methods all accept a variable number of arguments (or a representation of those arguments, such as an `NSArray`, `NSDictionary`, or a `va_list`), which are properly escaped for you.

And, to use that SQL with the `?` placeholders from Objective-C:

```
NSInteger identifier = 42;
NSString *name = @"Liam O'Flaherty (\"the famous Irish author\")";
NSDate *date = [NSDate date];
NSString *comment = nil;

BOOL success = [db executeUpdate:@"INSERT INTO authors (identifier, name, date, comment) VALUES (?, ?, ?
if (!success) {
    NSLog(@"error = %@", [db lastErrorMessage]);
}
```

> **Note:** Fundamental data types, like the `NSInteger` variable `identifier`, should be as a `NSNumber` objects, achieved by using the `@` syntax, shown above. Or you can use the `[NSNumber numberWithInt:identifier]` syntax, too.
>
> Likewise, SQL `NULL` values should be inserted as `[NSNull null]`. For example, in the case of `comment` which might be `nil` (and is in this example), you can use the `comment ?: [NSNull null]` syntax, which will insert the string if `comment` is not `nil`, but will insert `[NSNull null]` if it is `nil`.

In Swift, you would use `executeUpdate(values:)`, which not only is a concise Swift syntax, but also `throws` errors for proper error handling:

```
do {
    let identifier = 42
    let name = "Liam O'Flaherty (\"the famous Irish author\")"
    let date = Date()
    let comment: String? = nil

    try db.executeUpdate("INSERT INTO authors (identifier, name, date, comment) VALUES (?, ?, ?, ?)", va
} catch {
    print("error = \(error)")
}
```

> **Note:** In Swift, you don't have to wrap fundamental numeric types like you do in Objective-C. But if you are going to insert an optional string, you would probably use the `comment ?? NSNull()` syntax (i.e., if it is `nil`, use `NSNull`, otherwise use the string).

Alternatively, you may use named parameters syntax:

```
INSERT INTO authors (identifier, name, date, comment) VALUES (:identifier, :name, :date, :comment)
```

The parameters *must* start with a colon. SQLite itself supports other characters, but internally the dictionary keys are prefixed with a colon, do **not** include the colon in your dictionary keys.

```
NSDictionary *arguments = @{@"identifier": @(identifier), @"name": name, @"date": date, @"comment": comm
BOOL success = [db executeUpdate:@"INSERT INTO authors (identifier, name, date, comment) VALUES (:identi
```

```
if (!success) {
    NSLog(@"error = %@", [db lastErrorMessage]);
}
```

The key point is that one should not use `NSString` method `stringWithFormat` to manually insert values into the SQL statement, itself. Nor should one Swift string interpolation to insert values into the SQL. Use `?` placeholders for values to be inserted into the database (or used in `WHERE` clauses in `SELECT` statements).

Using FMDatabaseQueue and Thread Safety.

Using a single instance of `FMDatabase` from multiple threads at once is a bad idea. It has always been OK to make a `FMDatabase` object *per thread*. Just don't share a single instance across threads, and definitely not across multiple threads at the same time. Bad things will eventually happen and you'll eventually get something to crash, or maybe get an exception, or maybe meteorites will fall out of the sky and hit your Mac Pro. *This would suck*.

So don't instantiate a single `FMDatabase` object and use it across multiple threads.

Instead, use `FMDatabaseQueue`. Instantiate a single `FMDatabaseQueue` and use it across multiple threads. The `FMDatabaseQueue` object will synchronize and coordinate access across the multiple threads. Here's how to use it:

First, make your queue.

```
FMDatabaseQueue *queue = [FMDatabaseQueue databaseQueueWithPath:aPath];
```

Then use it like so:

```
[queue inDatabase:^(FMDatabase *db) {
    [db executeUpdate:@"INSERT INTO myTable VALUES (?)", @1];
    [db executeUpdate:@"INSERT INTO myTable VALUES (?)", @2];
    [db executeUpdate:@"INSERT INTO myTable VALUES (?)", @3];

    FMResultSet *rs = [db executeQuery:@"select * from foo"];
    while ([rs next]) {
        …
    }
}];
```

An easy way to wrap things up in a transaction can be done like this:

```
[queue inTransaction:^(FMDatabase *db, BOOL *rollback) {
    [db executeUpdate:@"INSERT INTO myTable VALUES (?)", @1];
    [db executeUpdate:@"INSERT INTO myTable VALUES (?)", @2];
    [db executeUpdate:@"INSERT INTO myTable VALUES (?)", @3];

    if (whoopsSomethingWrongHappened) {
        *rollback = YES;
        return;
    }

    // etc ...
}];
```

The Swift equivalent would be:

```
queue.inTransaction { db, rollback in
    do {
        try db.executeUpdate("INSERT INTO myTable VALUES (?)", values: [1])
        try db.executeUpdate("INSERT INTO myTable VALUES (?)", values: [2])
        try db.executeUpdate("INSERT INTO myTable VALUES (?)", values: [3])

        if whoopsSomethingWrongHappened {
            rollback.pointee = true
```

```
            return
        }

        // etc ...
    } catch {
        rollback.pointee = true
        print(error)
    }
}
```

(Note, as of Swift 3, use `pointee`. But in Swift 2.3, use `memory` rather than `pointee`.)

`FMDatabaseQueue` will run the blocks on a serialized queue (hence the name of the class). So if you call `FMDatabaseQueue`'s methods from multiple threads at the same time, they will be executed in the order they are received. This way queries and updates won't step on each other's toes, and every one is happy.

Note: The calls to `FMDatabaseQueue`'s methods are blocking. So even though you are passing along blocks, they will **not** be run on another thread.

Making custom sqlite functions, based on blocks.

You can do this! For an example, look for `-makeFunctionNamed:` in main.m

Swift

You can use FMDB in Swift projects too.

To do this, you must:

1. Copy the relevant `.m` and `.h` files from the FMDB `src` folder into your project.

You can copy all of them (which is easiest), or only the ones you need. Likely you will need `FMDatabase` and `FMResultSet` at a minimum. `FMDatabaseAdditions` provides some very useful convenience methods, so you will likely want that, too. If you are doing multithreaded access to a database, `FMDatabaseQueue` is quite useful, too. If you choose to not copy all of the files from the `src` directory, though, you may want to update `FMDB.h` to only reference the files that you included in your project.

Note, if you're copying all of the files from the `src` folder into to your project (which is recommended), you may want to drag the individual files into your project, not the folder, itself, because if you drag the folder, you won't be prompted to add the bridging header (see next point).

2. If prompted to create a "bridging header", you should do so. If not prompted and if you don't already have a bridging header, add one.

For more information on bridging headers, see Swift and Objective-C in the Same Project.

3. In your bridging header, add a line that says:

```
#import "FMDB.h"
```

4. Use the variations of `executeQuery` and `executeUpdate` with the `sql` and `values` parameters with `try` pattern, as shown below. These renditions of `executeQuery` and `executeUpdate` both `throw` errors in true Swift fashion.

If you do the above, you can then write Swift code that uses `FMDatabase`. For example, as of Swift 3:

```
let fileURL = try! FileManager.default
    .url(for: .documentDirectory, in: .userDomainMask, appropriateFor: nil, create: false)
    .appendingPathComponent("test.sqlite")

let database = FMDatabase(url: fileURL)

guard database.open() else {
    print("Unable to open database")
    return
```

```
}

do {
    try database.executeUpdate("create table test(x text, y text, z text)", values: nil)
    try database.executeUpdate("insert into test (x, y, z) values (?, ?, ?)", values: ["a", "b", "c"])
    try database.executeUpdate("insert into test (x, y, z) values (?, ?, ?)", values: ["e", "f", "g"])

    let rs = try database.executeQuery("select x, y, z from test", values: nil)
    while rs.next() {
        if let x = rs.string(forColumn: "x"), let y = rs.string(forColumn: "y"), let z = rs.string(forC
            print("x = \(x); y = \(y); z = \(z)")
        }
    }
} catch {
    print("failed: \(error.localizedDescription)")
}

database.close()
```

History

The history and changes are availbe on its GitHub page and are summarized in the "CHANGES_AND_TODO_LIST.txt" file.

Contributors

The contributors to FMDB are contained in the "Contributors.txt" file.

Additional projects using FMDB, which might be interesting to the discerning developer.

- FMDBMigrationManager, A SQLite schema migration management system for FMDB: https://github.com/layerhq/FMDBMigrationManager
- FCModel, An alternative to Core Data for people who like having direct SQL access: https://github.com/marcoarment/FCModel

Quick notes on FMDB's coding style

Spaces, not tabs. Square brackets, not dot notation. Look at what FMDB already does with curly brackets and such, and stick to that style.

Reporting bugs

Reduce your bug down to the smallest amount of code possible. You want to make it super easy for the developers to see and reproduce your bug. If it helps, pretend that the person who can fix your bug is active on shipping 3 major products, works on a handful of open source projects, has a newborn baby, and is generally very very busy.

And we've even added a template function to main.m (FMDBReportABugFunction) in the FMDB distribution to help you out:

- Open up fmdb project in Xcode.
- Open up main.m and modify the FMDBReportABugFunction to reproduce your bug.
  - Setup your table(s) in the code.
  - Make your query or update(s).
  - Add some assertions which demonstrate the bug.

Then you can bring it up on the FMDB mailing list by showing your nice and compact FMDBReportABugFunction, or you can report the bug via the github FMDB bug reporter.

Optional:

Figure out where the bug is, fix it, and send a patch in or bring that up on the mailing list. Make sure all the other tests run after your modifications.

Support

The support channels for FMDB are the mailing list (see above), filing a bug here, or maybe on Stack Overflow. So that is to say, support is provided by the community and on a voluntary basis.

FMDB development is overseen by Gus Mueller of Flying Meat. If FMDB been helpful to you, consider purchasing an app from FM or telling all your friends about it.

License

The license for FMDB is contained in the "License.txt" file.

If you happen to come across either Gus Mueller or Rob Ryan in a bar, you might consider purchasing a drink of their choosing if FMDB has been useful to you.

(The drink is for them of course, shame on you for trying to keep it.)