## Trabalhando com banco de dados SQLite e a biblioteca FMDB

Lidar com bancos de dados e manipular dados em geral é uma parte importante e crucial de qualquer aplicativo.

O Swift possui uma biblioteca que também pode ser usada para manipular banco de dados – SwiftDB. Ambas as bibliotecas servem ao mesmo objetivo: permitir que seja possível gerenciar bancos de dados SQLite e os dados do aplicativo com eficiência. No entanto, eles não são semelhantes em tudo ao modo como são usados. O SwiftyDB oferece uma API de programação de alto nível, ocultando todos os detalhes de SQL e outras operações avançadas, enquanto o FMDB fornece uma maneira melhor de lidar com dados detalhados por ser apenas uma API de baixo nível.

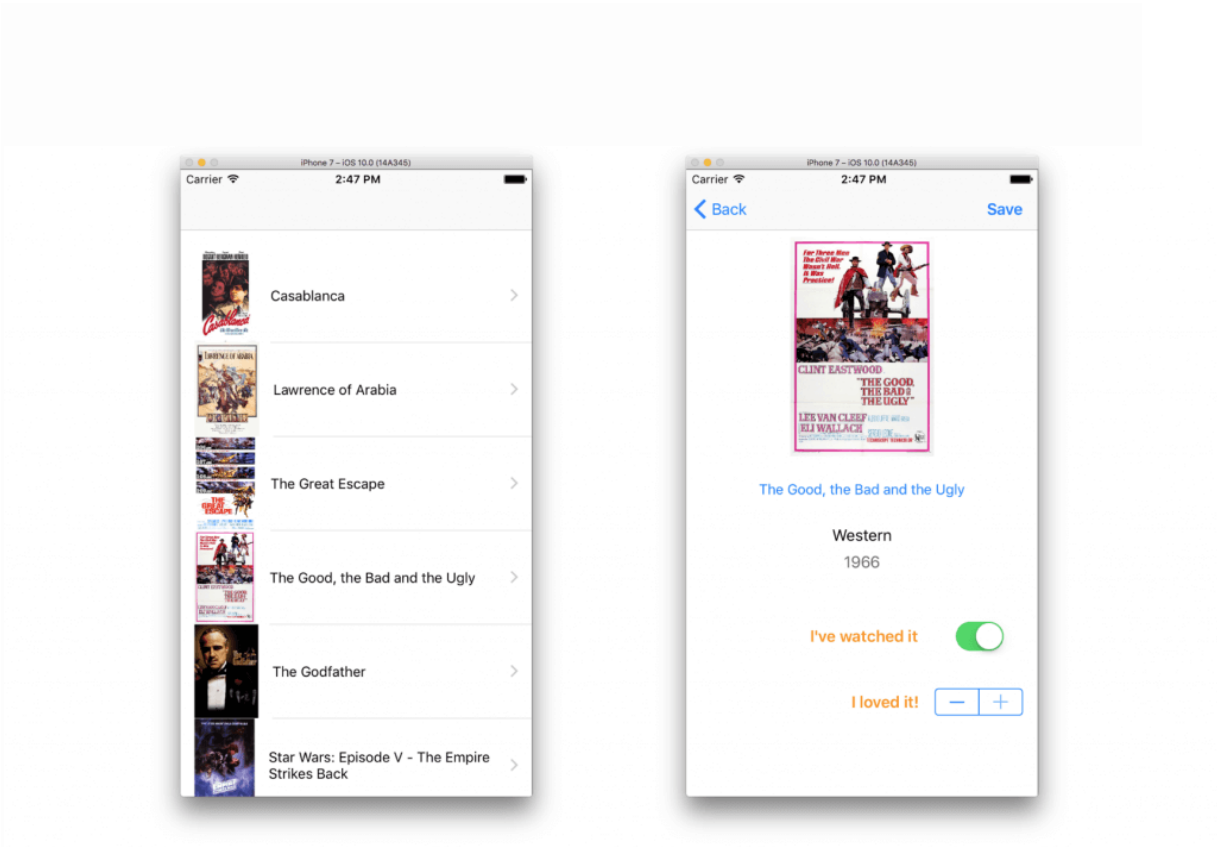
Ele ainda “esconde” os detalhes de conexão e comunicação no banco de dados SQLite. Mas, em geral, um pode ser melhor que o outro em casos específicos, e isso sempre depende da natureza e do propósito de cada aplicativo. Portanto, ambos são ótimas ferramentas que podem se adequar perfeitamente às nossas necessidades.

Com o foco na biblioteca FMDB- que, em linhas gerais, serve como um wrapper (envolucro) SQLite - o que significa que ela fornece os recursos do SQLite em um nível mais alto, para que não seja preciso lidar com as estruturas de conexão, assim como com a escrita e leitura de dados de e para o banco de dados. Essa é a melhor opção quando se deseja usar conhecimentos de SQL e escrever as próprias consultas SQL, mas sem, necessariamente, escrever o próprio gerenciador de SQLite.

FMDB funciona tanto com o Objective-C quanto com o Swift, e como é muito rápido integrá-lo em um projeto, a produtividade não tem custo nesse caso.
O exercício se inicia criando um novo banco de dados programaticamente e assim será possível ver todas as operações usuais que podem ser aplicadas aos dados: insert, update, delete e select.

## Visão geral do aplicativo de demonstração

O projeto inicial (fornecido pelo instrutor) vai exibir uma lista de filmes e seus detalhes podem ser apresentados em um novo View Controller. Junto com os detalhes, poderemos marcar um filme como *assistido* e dar notas(variando de 0 a 3).




Os dados para os filmes serão armazenados em um banco de dados SQLite, que será gerenciado usando a biblioteca FMDB. As iniciais do filme (duas primeiras) serão inseridas no banco de dados a partir de um *tab separated file (.tsv)*. O objetivo é focar nas estruturas do banco de dados, principalmente.

No projeto inicial, estão a disposição:

- a implementação do aplicativo padrão, bem como o arquivo .tsv original;

O aplicativo é baseado em navegação, com dois view controllers:

- ✓ O primeiro chamado *MoviesViewController* e contém uma tableview onde serão exibidos o título e uma imagem para cada filme (existem 20 filmes no total). Apenas para os registros, as imagens do filme não são armazenadas localmente; em vez disso, eles são buscados de forma assíncrona, quando a lista está sendo exibida.

- ✓ Ao tocar em uma célula de filme, o segundo controlador de exibição chamado *MovieDetailsViewController* será apresentado. Os detalhes a seguir para cada filme serão exibidos lá:

- Imagem (image)
- Título (title) - Este será um botão que, ao ser acessado a página do filme no site do IMDB, será aberto no Safari
- Categoria (category)
- Ano (Year)

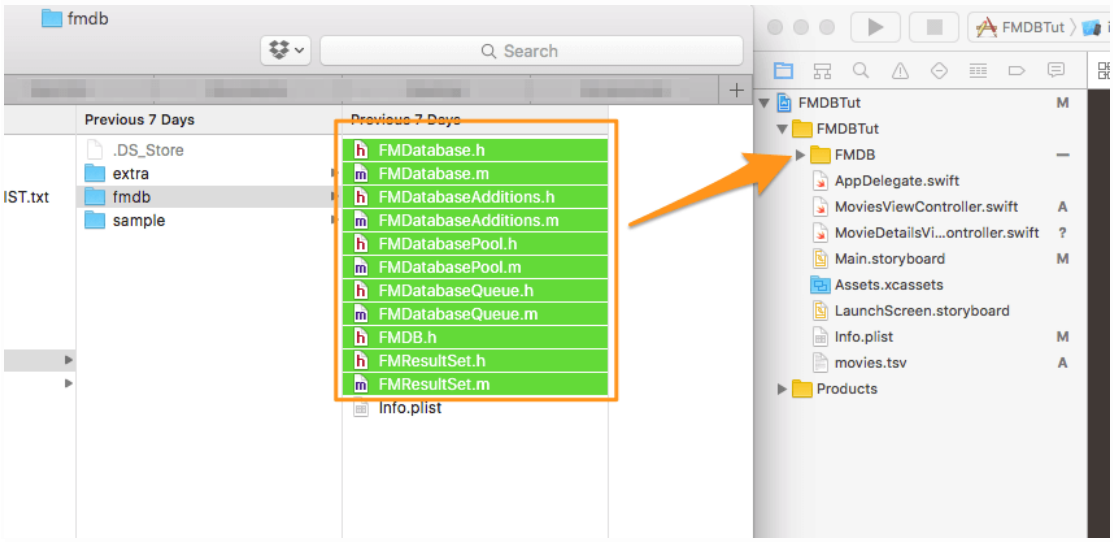
Além disso, também teremos um *switch* para indicar se um filme específico foi assistido ou não e um *stepper control* para aumentar ou diminuir o número de notas que gostaríamos de dar a cada filme. Os detalhes do filme atualizados serão armazenados no banco de dados.

Além disso, no arquivo *MoviesViewController.swift* também é possível encontrar uma estrutura chamada **MovieInfo**. Suas propriedades correspondem aos campos da tabela que serão mantidas no banco de dados e um objeto da estrutura *MovieInfo* representará um filme programaticamente.

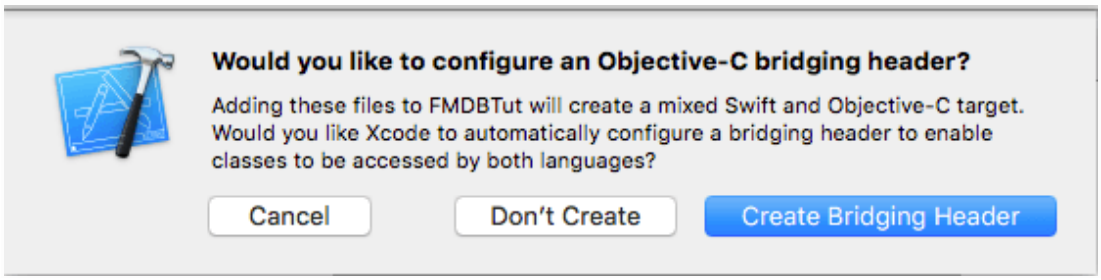
## Integrando o FMDB em seu projeto Swift

A maneira normal e usual de integrar a biblioteca do FMDB em seu projeto é instalando-a através do CocoaPods. No entanto, e especialmente para projetos Swift, é muito mais rápido baixar o repositório como um arquivo *zip* e adicionar arquivos específicos ao seu projeto. Você será solicitado a adicionar um arquivo de *bridging header*, também, porque a biblioteca FMDB está escrita em Objective-C e o arquivo de *bridging* é necessário para permitir que as duas linguagens trabalhem juntas. (Arquivo .zip fornecido pelo instrutor).

Depois de abrir o zip e descompactar seu conteúdo, navegue até o **diretório fmdb-master / src / fmdb** (no Finder). Os arquivos encontrados lá são aqueles que você precisa adicionar ao projeto inicial. Crie um primeiro novo *Group* no Project Navigator para esses arquivos; assim, é possivel mantê-los separados dos arquivos restantes do projeto. Selecione-os (há também um arquivo *.plist*, este é arquivo é desnecessário para o proejto) e, em seguida, arraste e solte no navegador do Project Navigator no Xcode.




Depois de adicionar os arquivos ao projeto, o Xcode solicitará que você crie um arquivo de *bridging header*.



Para que não seja necessário implementar esse arquivo manualmente, clique no botão *Create Briding Header* (em destaque) o XCode criará o arquivo de maneira automatica. Mais um arquivo será adicionado ao projeto, chamado *FMDBTut-Bridging-Header.h* . Abra e escreva a seguinte linha:

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

Agora, as classes do FMDB estarão disponíveis em todo o projeto Swift, e já é possível começar a usá-las.

## Criando o banco de dados

Trabalhar com o banco de dados quase sempre envolve as mesmas etapas gerais de ação:

- ✓ estabelecer uma conexão
- ✓ carregar ou modificar os dados armazenados
- ✓ fechar a conexão.

Isso é algo para se fazer em qualquer classe no projeto, pois as classes do FMDB estão disponíveis sempre que aplicativo necessita delas.

A sugestão é criar uma classe que faça o seguinte:

1. Manipule a comunicação com o banco de dados por meio da API do FMDB - Não será necessário escrever mais de uma vez um código que verifique se o arquivo de banco de dados realmente existe ou se o banco de dados está aberto ou não.

2. Implementar métodos relacionados ao banco de dados - operar os dados criando métodos personalizados específicos, dependendo de necessidades do aplicativo, e chamaremos esses métodos de outras classes apenas para fazer uso dos dados.

Então será criado um tipo de API de banco de dados de alto nível com base no FMDB, mas totalmente relacionado aos propósitos do nosso aplicativo. Para dar uma maior flexibilidade à forma como esta classe irá funcionar, vamos torná-la um *singleton* e poderemos usá-la sem criar novas instâncias (novos objetos) dela quando precisarmos usá-la.

Inicie criando uma nova classe para o gerenciador de banco de dados (no Xcode, vá para **o File menu > New > File… -> Cocoa Touch Class**). Quando você for perguntado pelo Xcode para dar um nome, defina como **DBManager** e certifique-se de torná-lo uma subclasse da classe **NSObject** .

Abra a classe *DBManager* e adicione a seguinte linha para torná-la um singleton:

```
static let shared: DBManager = DBManager()
```

A partir de agora, é necessário apenas que escrever a seguinte linha de código `DBManager.shared.Do_Something()` . Não há necessidade de inicializar novas instâncias da classe (mas é possível fazê-lo).

Agora é preciso declarar três propriedades importantes para o aplicativo:

1. O nome do arquivo de banco de dados - não é necessário tê-lo como uma propriedade, mas é recomendado para fins reutilizáveis.

2. O caminho para o arquivo de banco de dados.

3. Um objeto **FMDatabase** (da biblioteca FMDB) que estará acessando e operando no banco de dados real.

   São estas:

```
let databaseFileName = "database.sqlite"

var pathToDatabase: String!

var database: FMDatabase!
```

Não podemos esquecer do método `init()` que deve existir em nossa classe:

```
override init() {
        super.init()

        let documentsDirectory =
(NSSearchPathForDirectoriesInDomains(.documentDirectory,
.userDomainMask, true)[0] as NSString) as String
        pathToDatabase =
documentsDirectory.appending("/\(databaseFileName)")

 }
```

O método `init()` não está vazio; o método `init()` é o melhor lugar para especificar o caminho para o diretório de documentos do aplicativo e compor o caminho para o arquivo de banco de dados.

Agora é possível criar o banco de dados em um novo método personalizado que se chamará `createDatabase()`. Esse método retornará um valor `Bool` indicando se o banco de dados foi criado com sucesso ou não. A finalidade do valor de retorno será ainda mais importante na medida que a implementação do banco de dados avançar. A criação do banco de dados e a inserção inicial dos dados são duas ações que acontecerão apenas uma vez, na primeira vez em que o aplicativo for executado.

Vamos ver agora como o arquivo de banco de dados será criado:

```
    func createDatabase() -> Bool {
        var created = false

        if !FileManager.default.fileExists(atPath:
pathToDatabase) {
            database = FMDatabase(path: pathToDatabase!)


        }

        return created
    }
```

Duas coisas importantes e necessárias para entender:

1. Procede a criação do banco de dados e o que vem a seguir é apenas se o arquivo de banco de dados não existir. Isso é importante, porque não necessitamos criar o arquivo de banco de dados novamente e destruir o banco de dados original.

2. A linha: `database = FMDatabase(path: pathToDatabase!)` está criando o arquivo de banco de dados especificado pelo argumento do *initialiser*, se apenas o arquivo não for encontrado (essa é a necessidade). Nenhuma conexão está sendo estabelecida nesse ponto. É possível saber depois dessa linha é possivel usar o `database` property para ter acesso ao banco de dados.

Não é necessário verificar a flag `created`, neste momento. Ainda é preciso definir o seu valor adequado, posteriormente.

De volta ao nosso novo método, continuamos certificando-nos de que o banco de dados foi criado e abrindo-o:

```
func createDatabase() -> Bool {
        var created = false

        if !FileManager.default.fileExists(atPath:
pathToDatabase) {
            database = FMDatabase(path: pathToDatabase!)
if database != nil {
                // Open the database.
                if database.open() {

                }
                else {
                    print("Could not open the database.")
                }
            }

        }

        return created
    }
```

A `database.open()` é uma linha chave, acima, já que estamos autorizados a agir nos dados do banco de dados somente depois que ele for aberto. Posteriormente, o banco de dados será fechado(a conexão com ele, na verdade) de maneira semelhante à acima.

Neste passo, vamos criar uma *tabela* no banco de dados. Os campos dessa tabela (que vamos nomear *movies*) são os mesmos para as propriedades da estrutura *MovieInfo* , então se você simplesmente abrir o arquivo *MoviesViewController.swift* no Xcode, será possível ve-las.

```
let createMoviesTableQuery = "create table movies
(\(field_MovieID) integer primary key autoincrement not null,
\(field_MovieTitle) text not null, \(field_MovieCategory) text
not null, \(field_MovieYear) integer not null, \(field_MovieURL)
text, \(field_MovieCoverURL) text not null,
\(field_MovieWatched) bool not null default 0,
\(field_MovieLikes) integer not null)"
```

A estrutura, a seguir, executará a consulta acima e criará a nova tabela em nosso banco de dados:

```
try database.executeUpdate(createMoviesTableQuery, values: nil)
created = true
```

O método `executeUpdate(...)` é usado para todas as consultas que podem modificar o banco de dados. O segundo argumento usa um array de valores que, provavelmente, passaremos junto com a consulta, mas, por enquanto, não precisamos usá-la.

O código acima poderá disparar um erro no Xcode, porque esse método pode *executar* uma exceção se ocorrer algum erro. Esse fato nos faz mudar a última linha para isso:

```
do {
       try database.executeUpdate(createMoviesTableQuery,
values: nil)created = true
                    }
                    catch {
                        print("Could not create table.")
                        print(error.localizedDescription)

                    }
```

Observe que a flag `created` se torna `true` se apenas a tabela for criada com sucesso, e você pode ver isso no bloco `do` da declaração.

Logo a seguir temos o método `createDatabase()`. Observe a declaração `catch` onde fechamos o banco de dados, não importa o que tenha acontecido antes:

```
func createDatabase() -> Bool {
        var created = false

        if !FileManager.default.fileExists(atPath:
pathToDatabase) {
            database = FMDatabase(path: pathToDatabase!)

            if database != nil {
                // Open the database.
                if database.open() {
                    let createMoviesTableQuery = "create table
movies (\(field_MovieID) integer primary key autoincrement not
null, \(field_MovieTitle) text not null, \(field_MovieCategory)
text not null, \(field_MovieYear) integer not null,
\(field_MovieURL) text, \(field_MovieCoverURL) text not null,
\(field_MovieWatched) bool not null default 0,
\(field_MovieLikes) integer not null)"

                    do {
                        try
database.executeUpdate(createMoviesTableQuery, values: nil)
                        created = true
```

```
                    }
                    catch {
                        print("Could not create table.")
                        print(error.localizedDescription)
                    }

                    database.close()
                }
                else {
                    print("Could not open the database.")
                }
            }
        }

        return created

    }
```

Na classe *DBManager* , insira o método abaixo:

```
  func openDatabase() -> Bool {
        if database == nil {
            if FileManager.default.fileExists(atPath:
pathToDatabase) {
                database = FMDatabase(path: pathToDatabase)
            }
        }

        if database != nil {
            if database.open() {
                return true
            }
        }

        return false
    }
```

A princípio, o método verifica se o objeto do banco de dados já foi inicializado ou não, e isso ocorre no caso de ainda ser nulo. Em seguida, ele tenta abrir o banco de dados. O valor de retorno desse método é um `Bool` . Quando é *true*, o banco de dados foi aberto com sucesso, caso contrário, o arquivo de banco de dados não existe ou outro erro ocorreu e o banco de dados não pôde ser aberto. Mas, geralmente, se o método retornar *true*, então configuramos um manipulador para nosso banco de dados pronto para ser usado (o `database` *object*) e, mais

importante, implementando esse método, não precisamos escrever as linhas acima toda vez que precisamos para abrir o banco de dados. Sinta-se à vontade para estender a implementação acima e adicionar mais condições, verificações ou mensagens de erro.

Na parte anterior, compusemos uma consulta SQL que cria a tabela de *movies* :

```
let createMoviesTableQuery = "create table movies
(\(field_MovieID) integer primary key autoincrement not null,
\(field_MovieTitle) text not null, \(field_MovieCategory) text
not null, \(field_MovieYear) integer not null, \(field_MovieURL)
text, \(field_MovieCoverURL) text not null,
\(field_MovieWatched) bool not null default 0,
\(field_MovieLikes) integer not null)"
```

Essa consulta de acordo, mas existem “potenciais problemas” em todas as consultas subsequentes. O “problema” está nos nomes dos campos e no fato de termos que escrever os literais do nome em todas as consultas que forem criadas. Se continuarmos com esse procedimento - pode-se digitar incorretamente o nome de um ou mais campos e isso resultará em erros. Há uma boa maneira de se livrar desse risco: atribuir os nomes de campo (de todas as tabelas, neste caso, apenas uma tabela) em constant properties.
Navegue até a classe DBManager e adicione o seguinte código:

```
    let field_MovieID = "movieID"
    let field_MovieTitle = "title"
    let field_MovieCategory = "category"
    let field_MovieYear = "year"
    let field_MovieURL = "movieURL"
    let field_MovieCoverURL = "coverURL"
    let field_MovieWatched = "watched"

    let field_MovieLikes = "likes"
```

Foi adicionado o prefixo "field" para facilitar a localização do campo que você deseja ao digitar no Xcode. Se você começar escrevendo “field”, o Xcode irá sugerir automaticamente todas as propriedades que contenham esse termo, e é fácil encontrar o nome do field no qual você está interessado. A segunda parte de cada nome é na verdade uma pequena descrição sobre cada campo. Você poderia avançar ainda mais e incluir o nome da tabela em cada propriedade também:

```
let field_MovieID = "movieID"
```

Para este exercício isso não é necessário; existe apenas uma tabela, mas faz uma grande diferença se o aplicativo possuir várias tabelas.

Ao atribuir os nomes dos *field* em *constants*, não há necessidade de digitar nenhum nome de *field* novamente, já que estaremos usando as *constants*, garantindo que não existam erros de digitação. Se atualizarmos nossa consulta, veja como ficará no final:

```
let createMoviesTableQuery = "create table movies
(\(field_MovieID) integer primary key autoincrement not null,
\(field_MovieTitle) text not null, \(field_MovieCategory) text
not null, \(field_MovieYear) integer not null, \(field_MovieURL)
text, \(field_MovieCoverURL) text not null,
\(field_MovieWatched) bool not null default 0,
\(field_MovieLikes) integer not null)"
```

## Inserindo Registros

Nesta passo, serão inseridos alguns dados iniciais no banco de dados, e a fonte para esses dados será o arquivo chamado *movies.tsv* que já existe no projeto inicial (apenas *localize* -o no Project Navigator). Este arquivo contém dados para 20 filmes, e os registros do filme são separados pelos caracteres “ \ *r* \ *n* ” (sem as aspas). Um caractere de *tabulação* (“\ t”) separa os dados de um único filme, e esse formato fará com que nosso trabalho de *análise seja* realmente fácil. A ordem dos dados é a seguinte:

- Movie title
- Category
- Year

- Movie URL

- Movie cover URL

Para o restante dos campos que existem na tabela, apenas inseriremos alguns valores padrão.

Na classe *DBManager* , vamos implementar um novo método que fará todo o trabalho para nós. Vamos começar fazendo uso do método implementado na parte anterior, então abrimos o banco de dados em apenas uma linha:

```
func insertMovieData() {
        if openDatabase() {

        }
    }
```

A lógica é a seguinte:

1. Primeiro, vamos localizar o arquivo “movies.tsv” e carregaremos seu conteúdo em um objeto *String* .

2. Em seguida, separaremos os dados de filmes quebrando a string com base na substring */ r / n* , e criaremos uma matriz de strings ( `[String]` ). Cada posição manterá a string com os dados de um único filme.
3. Em seguida, e usando um loop, vamos percorrer todos os filmes e buscá-los um por um, e dividiremos cada sequência de filme de maneira semelhante à anterior, mas dessa vez com base no caractere de tabulação (“\ t”). Isso resultará em um novo array, em que cada posição conterá uma parte diferente dos dados de cada filme. Será muito simples usar os dados e compor as consultas de inserção desejadas.

Começando pelo primeiro ponto, vamos pegar o caminho do arquivo “movies.tsv” e vamos carregar seu conteúdo em um objeto de string:

```
 if let pathToMoviesFile = Bundle.main.path(forResource:
"movies", ofType: "tsv") {
                do {
                    let moviesFileContents = try
String(contentsOfFile: pathToMoviesFile)

    }
    catch {
        print(error.localizedDescription)
          }
}
```

Criar uma string com o conteúdo do arquivo pode lançar uma exceção, portanto, `do-catch` é necessário usar a instrução. Agora vamos ao segundo ponto e vamos quebrar o conteúdo da string em um array de strings baseado nos caracteres “\ r \ n”:

```
let moviesData = moviesFileContents.components(separatedBy:
"\r\n")
```

Atingindo o terceiro ponto agora, vamos fazer um `for` loop e dividir os dados de cada filme em arrays. Note que antes do loop vamos inicializar outro valor de string (chamado `query`) que usaremos para compor os comandos insert em alguns segundos.

```
                    var query = ""
                     for movie in moviesData {
                         let movieParts =
movie.components(separatedBy: "\t")

                        if movieParts.count == 5 {
                            let movieTitle = movieParts[0]
                            let movieCategory = movieParts[1]
                            let movieYear = movieParts[2]
                            let movieURL = movieParts[3]
                            let movieCoverURL = movieParts[4]
    }

}
```

Dentro do corpo da declaração `if`, acima, estaremos compondo nossas consultas de inserção. Como está descrito no próximo snippet, cada consulta termina com um símbolo de *ponto - e - vírgula (;)* por um motivo simples: Queremos executar várias consultas de uma só vez, e o SQLite conseguirá diferenciá-las com base no (*;*) . Observe mais duas coisas:
para os nomes de campo, estão sendo usados os valores constantes que criados anteriormente.
preste atenção aos símbolos de aspas simples “ *' ” para valores de string dentro da consulta. É possível ocorrer problemas se você omitir qualquer (‘) necessária*.

```
query += "insert into movies (\(field_MovieID),
\(field_MovieTitle), \(field_MovieCategory), \(field_MovieYear),
\(field_MovieURL), \(field_MovieCoverURL),
\(field_MovieWatched), \(field_MovieLikes)) values (null,
'\(movieTitle)', '\(movieCategory)', \(movieYear),
'\(movieURL)', '\(movieCoverURL)', 0, 0);"
```

Nos dois últimos campos, especificamos alguns valores. Posteriormente, executaremos as consultas de atualização para alterá-las.

No momento em que o `for` loop terminar, a `query` string conterá todas as consultas inseridas que queremos executar (20 consultas no total aqui). Executar múltiplas instruções de uma só vez é fácil com o FMDB, já que tudo o que temos a fazer é usar o `executeStatements(_:)` método através do `database` object:

```
if !database.executeStatements(query) {
                        print("Failed to insert initial data
into the database.")
                        print(database.lastError(),
database.lastErrorMessage())

                    }
```

O `lastError()` e `lastErrorMessage()` mostrado acima será realmente útil caso a operação de inserção falhe. Esses dois métodos reportarão o problema encontrado e provavelmente onde exatamente o erro está, para que você possa consertá-lo facilmente. Esse trecho de código, claro, tem que ser escrito após o fechamento do loop.

Mesmo que isso possa não parecer importante, não esqueça de fechar a conexão com o banco de dados, então complete o código adicionando o

comando `database.close()`. Aqui está o código de `insertMovieData()`, depois de ter completado sua implementação:

```
func insertMovieData() {
        if openDatabase() {
            if let pathToMoviesFile =
Bundle.main.path(forResource: "movies", ofType: "tsv") {
                do {
                    let moviesFileContents = try
String(contentsOfFile: pathToMoviesFile)

                    let moviesData =
moviesFileContents.components(separatedBy: "\r\n")

                    var query = ""
                    for movie in moviesData {
                        let movieParts =
movie.components(separatedBy: "\t")

                        if movieParts.count == 5 {
                            let movieTitle = movieParts[0]
                            let movieCategory = movieParts[1]
                            let movieYear = movieParts[2]
                            let movieURL = movieParts[3]
                            let movieCoverURL = movieParts[4]

                            query += "insert into movies
(\(field_MovieID), \(field_MovieTitle), \(field_MovieCategory),
\(field_MovieYear), \(field_MovieURL), \(field_MovieCoverURL),
\(field_MovieWatched), \(field_MovieLikes)) values (null,
'\(movieTitle)', '\(movieCategory)', \(movieYear),
'\(movieURL)', '\(movieCoverURL)', 0, 0);"
                        }
                    }

                    if !database.executeStatements(query) {
                        print("Failed to insert initial data
into the database.")
                        print(database.lastError(),
database.lastErrorMessage())
                    }
                }
                catch {
                    print(error.localizedDescription)
                }
            }

            database.close()
        }
```

```
    }
```

Resta uma última coisa a fazer; devemos chamar nossos novos métodos que criam o banco de dados e inserem os dados iniciais no banco de dados. Abra o arquivo *AppDelegate.swift* e *localize* o `applicationDidBecomeActive(_:)` Delegate method . Adicione as próximas duas linhas:

```
func applicationDidBecomeActive(_ application: UIApplication) {
        // Restart any tasks that were paused (or not yet
started) while the application was inactive. If the application
was previously in the background, optionally refresh the user
interface.

        if DBManager.shared.createDatabase() {
            DBManager.shared.insertMovieData()
        }

    }
```

## Carregando dados

Na classe *MoviesViewController,* há uma tableview com a implementação básica já feita, no entanto, está "aguardando" que a implementação seja concluída para que seja possível listar os filmes que carregaremos do banco de dados. A fonte de dados para essa tableview é um array chamado `movies` e é uma coleção de objetos *MovieInfo* . A struct *MovieInfo* , que também é encontrada no arquivo *MoviesViewController.swift* , consiste na representação programática da tabela de *movies* no banco de dados, e um objeto descreve um único filme. Com isso em mente, o que queremos nesse passo é carregar os filmes

existentes do banco de dados e atribuir os detalhes em objetos *MovieInfo*, que usaremos para preencher os dados na tableview.

Retornando à classe *DBManager* mais uma vez, o objetivo mais importante aqui é ver como as consultas *SELECT* são executadas no FMDB; vamos gerenciar isso carregando os dados dos filmes dentro do corpo de um novo método customizado:

```
func loadMovies() -> [MovieInfo]! {

}
```

O valor de retorno é uma coleção de objetos *MovieInfo* , exatamente o que precisamos na classe *MoviesViewController* . Começaremos a implementar esse método declarando um array local para armazenar os resultados que serão carregados do banco de dados e abrindo o banco de dados:

```
unc loadMovies() -> [MovieInfo]! {
        var movies: [MovieInfo]!


        if openDatabase() {


}


return movies


}
```

Nossa próxima etapa é criar a consulta SQL que informa ao banco de dados quais dados carregar:

```
let query = "select * from movies order by \(field_MovieYear)
asc"
```

Essa consulta é executada conforme mostrado a seguir:

```
  do {
                print(database)
                let results = try database.executeQuery(query,
values: nil)

}
            catch {
                print(error.localizedDescription)

            }
```

O método `executeQuery(...)` do objeto *FMDatabase* obtém dois parâmetros: A string de consulta e uma matriz de valores que devem ser passados junto com a consulta. Se não houver valores, a configuração nula está correta. O método retorna um *objeto FMResultSet (é uma classe FMDB)* que contém todos os dados recuperados e, em alguns instantes, veremos como *acessamos* os dados retornados.

Com a consulta acima, estamos apenas pedindo ao FMDB para buscar todos os filmes pedidos em ordem crescente com base no ano de lançamento. Esta é apenas uma consulta simples dada como exemplo, mas consultas mais avançadas podem ser criadas de acordo com suas necessidades também. Vamos ver outro, um pouco mais complicado, em que carregamos os filmes de uma categoria específica apenas, ordenados pelo ano novamente, mas em ordem decrescente:

```
let query = "select * from movies where \(field_MovieID)=?"
order by (field_MovieYear) desc”
```

É possível observar que o nome da categoria para a cláusula *where* não é especificado na própria consulta. Em vez disso, definimos um marcador na consulta e forneceremos o valor real, como mostrado abaixo (estamos dizendo ao FMDB para carregar somente os filmes que pertencem à categoria *Crime* ):

```
let results = "try database.executeQuery(query,values[“Crime”])
```

Outro exemplo, onde todos os dados de filmes são carregados para uma categoria específica e ano de lançamento maior que o ano que especificaremos, ordenados por seus valores de ID em ordem decrescente:

```
let query = "select * from movies where
\(field_MovieCategory)=?" and \(field_MovieYear)=? order by
(field_MovieID) desc”
```

O código acima espera que dois valores sejam fornecidos junto com a consulta:

```
let results = "try database.executeQuery(query,values[“Crime”,

1990])
```

Agora, façamos uso dos dados retornados. No snippet a seguir, estamos usando um `while` loop para percorrer todos os registros retornados. Para cada um, estamos inicializando um novo objeto *MovieInfo* que *anexamos* ao `movies` array e, eventualmente, criamos a coleção de dados que será exibida na tableview.

```
while results.next() {
                    let movie = MovieInfo(movieID:
Int(results.int(forColumn: field_MovieID)),
                                          title:
results.string(forColumn: field_MovieTitle),
                                          category:
results.string(forColumn: field_MovieCategory),
                                          year:
Int(results.int(forColumn: field_MovieYear)),
                                          movieURL:
results.string(forColumn: field_MovieURL),
                                          coverURL:
results.string(forColumn: field_MovieCoverURL),
                                          watched:
results.bool(forColumn: field_MovieWatched),
                                          likes:
Int(results.int(forColumn: field_MovieLikes))
                    )

                    if movies == nil {
                        movies = [MovieInfo]()
```

```
                    }

                    movies.append(movie)
                }
```

Há um requisito importante e obrigatório no código acima, que sempre se aplica, independentemente de se estar esperando que dados múltiplos ou únicos sejam buscados: O método `results.next()` deve ser sempre chamado. Quando vários registros são usados com a instrução `while` ; para resultados de registro único, você pode usá-lo com uma declaração `if` :

```
If results.next() {
}
```

Outro detalhe para se ter em mente: Cada `movie` object é inicializado usando o inicialiser padrão para a estrutura *MovieInfo* . Isso é possível de acontecer no nosso exercicio, porque estamos pedindo que todos os campos sejam retornados para cada registro que será recuperado em nossa consulta ( `select * from movies ...` ). Se, no entanto, você decidir que deseja obter um subconjunto dos campos (por exemplo, `select (field_MovieTitle), (field_MovieCoverURL) from movies where ...` ), o inicialiser acima não funcionará e o aplicativo simplesmente falhará. E isso acontece porque qualquer método `results.XXX(forColumn:)` que tente buscar dados para campos não carregados encontrará *nil* em vez de valores reais. Então, observe isso e sempre tenha com calreza quais campos você pediu para serem carregados do banco de dados quando você lida com os resultados.

Vamos ver agora o método que criamos aqui em estrutura final:

```
func loadMovies() -> [MovieInfo]! {
        var movies: [MovieInfo]!

        if openDatabase() {
```

```
            let query = "select * from movies order by
\(field_MovieYear) asc"

            do {
                print(database)
                let results = try database.executeQuery(query,
values: nil)

                while results.next() {
                    let movie = MovieInfo(movieID:
Int(results.int(forColumn: field_MovieID)),
                                          title:
results.string(forColumn: field_MovieTitle),
                                          category:
results.string(forColumn: field_MovieCategory),
                                          year:
Int(results.int(forColumn: field_MovieYear)),
                                          movieURL:
results.string(forColumn: field_MovieURL),
                                          coverURL:
results.string(forColumn: field_MovieCoverURL),
                                          watched:
results.bool(forColumn: field_MovieWatched),
                                          likes:
Int(results.int(forColumn: field_MovieLikes))
                    )

                    if movies == nil {
                        movies = [MovieInfo]()
                    }

                    movies.append(movie)
                }
            }
            catch {
                print(error.localizedDescription)
            }

            database.close()
        }

        return movies

    }
```

Vamos fazer uso disso, então conseguimos preencher os dados dos filmes na tableview. Abra o arquivo *MoviesViewController.swift* e implemente o método `viewWillAppear(_:)`. Adicione as duas linhas seguintes que carregarão os dados dos filmes usando o método acima, e isso acionará um recarregamento na visualização de tabela:

```
override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)

        movies = DBManager.shared.loadMovies()
        tblMovies.reloadData()

    }
```

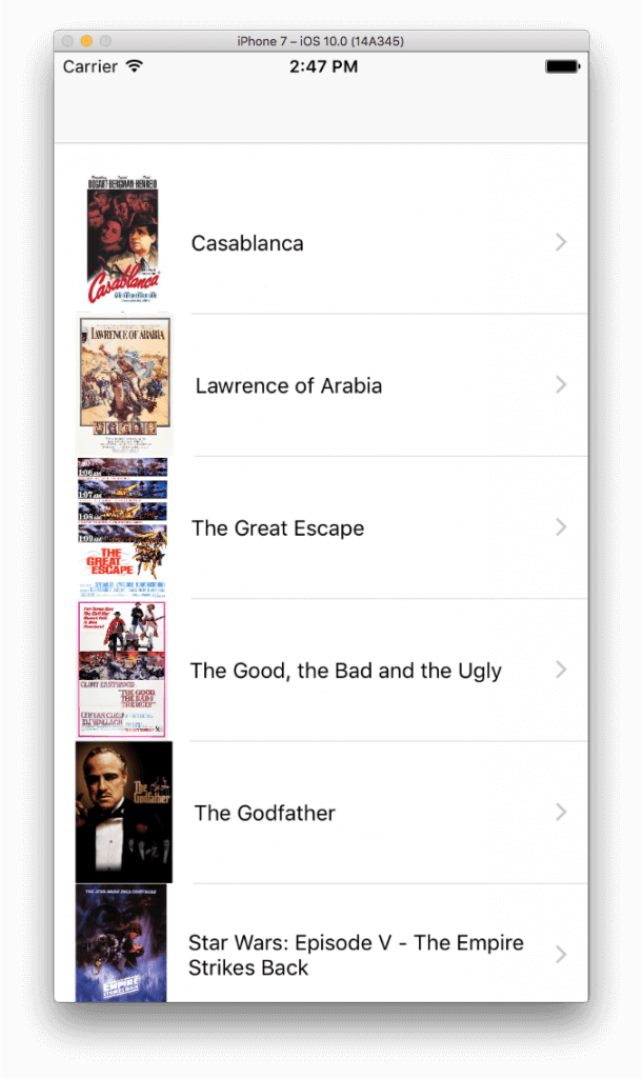
Mas ainda assim, temos que especificar o conteúdo de cada célula no método `tableView(_:, cellForRowAt indexPath:)`:

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier:
"cell", for: indexPath)

        let currentMovie = movies[indexPath.row]

        cell.textLabel?.text = currentMovie.title
        cell.imageView?.contentMode =
UIViewContentMode.scaleAspectFit

        (URLSession(configuration:
URLSessionConfiguration.default)).dataTask(with: URL(string:
currentMovie.coverURL)!, completionHandler: { (imageData,
response, error) in
            if let data = imageData {
                DispatchQueue.main.async {
                    cell.imageView?.image = UIImage(data: data)
                    cell.layoutSubviews()
                }
            }
        }).resume()

        return cell
    }
```

Cada imagem de filme é baixada de forma assíncrona e é exibida na célula quando seus dados se tornam disponíveis. O bloco *URLSession*, escrito em várias linhas, ficaria assim:

```
let sessionConfiguration = URLSessionConfiguration.default
let session = URLSession(configuration:
URLSessionConfiguration.default)
let task = session.dataTask(with: URL(string:
currentMovie.coverURL)!) { (imageData, response, error) in
    if let data = imageData {
        DispatchQueue.main.async {
            cell.imageView?.image = UIImage(data: data)
            cell.layoutSubviews()
        }
    }
}
task.resume()
```

De qualquer forma, agora você pode executar o aplicativo pela primeira vez. No primeiro lançamento, o banco de dados será criado e os dados iniciais serão inseridos nele. Em seguida, os dados serão carregados e os filmes serão exibidos na tableview, como mostrado na próxima captura de tela:




## Atualizando

Precisamos que nosso aplicativo exiba os detalhes do filme quando tocamos em uma célula na tableview, o que significa que queremos apresentar o *MovieDetailsViewController* e preenchê-lo com os detalhes do filme selecionado. Mesmo que a abordagem mais fácil seja simplesmente passar o objeto *MovieInfo*selecionado para o *MovieDetailsViewController* , implementaremos um caminho diferente. Será passado o ID do filme e depois carregaremos o filme do banco de dados.

Começaremos atualizando o método que prepara a sequência que apresentará o *MovieDetailsViewController* , de modo que continuamos no arquivo *MoviesViewController.swift* . Há uma implementação inicial, portanto, apenas atualize da seguinte maneira (adicione as duas linhas na declaração `if` interna ):

```
override func prepare(for segue: UIStoryboardSegue, sender:
Any?) {
    if let identifier = segue.identifier {
        if identifier == "idSegueMovieDetails" {
            let movieDetailsViewController = segue.destination
as! MovieDetailsViewController
            movieDetailsViewController.movieID =
movies[selectedMovieIndex].movieID
        }
    }

}
```

A propriedade `selectedMovieIndex` recebe seu valor no seguinte método tableview que já está implementado no projeto inicial:

```
func tableView(_ tableView: UITableView, didSelectRowAt
indexPath: IndexPath) {
    selectedMovieIndex = indexPath.row
    performSegue(withIdentifier: "idSegueMovieDetails", sender:
nil)
}
```

Além disso, há uma propriedade nomeada `movieID` no *MovieDetailsViewController* , portanto, o código acima funcionará bem.

Agora que passamos o ID do filme selecionado para o próximo view controller, precisamos escrever um novo método que carregará os dados do filme especificado por esse ID. Não haverá material relacionado ao banco de dados

que não tenhamos visto antes dentro desse método. No entanto, haverá uma diferença: normalmente você esperaria que esse método retornasse um objeto *MovieInfo* . Em vez de um valor de retorno, usaremos um *completion handler (manipulador de conclusão)* para passar os dados buscados de volta para a classe *MovieDetailsViewController* ; para mostrar como é possivel usar *completion handler* (manipuladores de conclusão) em vez de retornar valores ao buscar dados do banco de dados.

Vamos ao arquivo *DBManager.swift* e vamos ver a linha de cabeçalho do nosso novo método:

```
func loadMovie(withID ID: Int, completionHandler: (_ movieInfo:
MovieInfo?) -> Void) {

}
```

Há dois parâmetros aqui: O primeiro é o ID do filme que queremos carregar. O segundo é o manipulador de conclusão, que por sua vez tem um parâmetro, o filme carregado como um objeto *MovieInfo* .

Em relação à sua implementação completa deste passo:

```
func loadMovie(withID ID: Int, completionHandler: (_ movieInfo:
MovieInfo?) -> Void) {
    var movieInfo: MovieInfo!

    if openDatabase() {
        let query = "select * from movies where
\(field_MovieID)=?"

        do {
            let results = try database.executeQuery(query,
values: [ID])

            if results.next() {
                movieInfo = MovieInfo(movieID:
Int(results.int(forColumn: field_MovieID)),
                                      title:
results.string(forColumn: field_MovieTitle),
```

```
                                        category:
results.string(forColumn: field_MovieCategory),
                                        year:
Int(results.int(forColumn: field_MovieYear)),
                                        movieURL:
results.string(forColumn: field_MovieURL),
                                        coverURL:
results.string(forColumn: field_MovieCoverURL),
                                        watched:
results.bool(forColumn: field_MovieWatched),
                                        likes:
Int(results.int(forColumn: field_MovieLikes))
                )

            }
            else {
                print(database.lastError())
            }
        }
        catch {
            print(error.localizedDescription)
        }

        database.close()
    }

    completionHandler(movieInfo)
}
```

No final do método, chamamos o manipulador de conclusão passando o `movieInfo` objeto, não importa se ele foi inicializado com os valores do filme, ou é nulo porque algo deu errado.

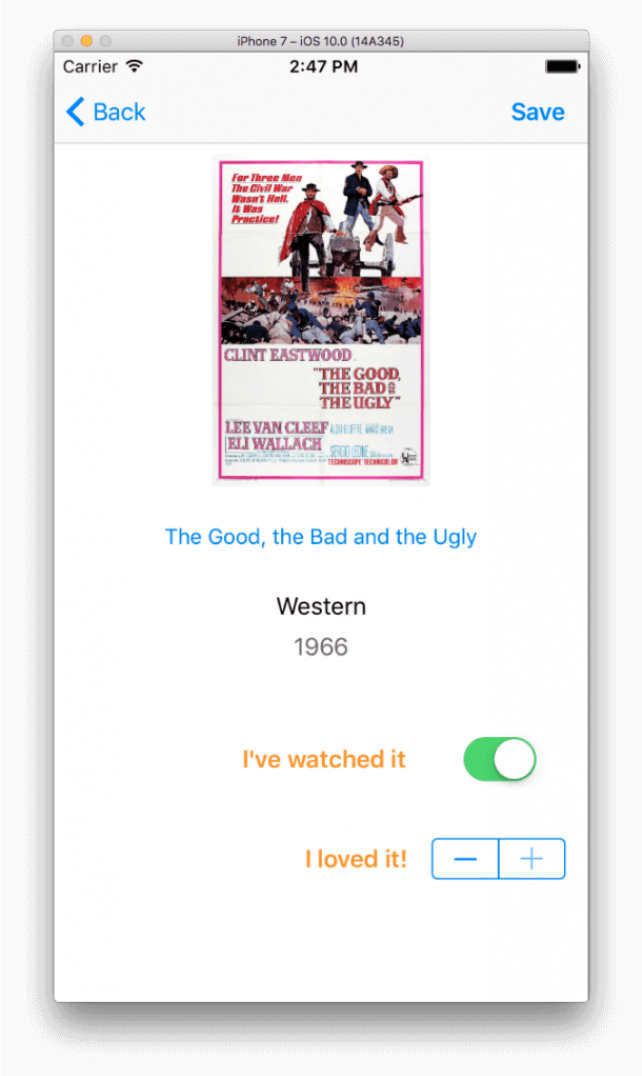
No *MovieDetailsViewController.swift* agora, vamos direto ao `viewWillAppear(_:)` método e vamos chamar o acima:

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

    if let id = movieID {
        DBManager.shared.loadMovie(withID: id,
completionHandler: { (movie) in
            DispatchQueue.main.async {
                if movie != nil {
                    self.movieInfo = movie
                    self.setValuesToViews()
```

```
                }
            }
        })
    }

}
```

Duas coisas para observar:

- ✓ Primeiro, o `movie` object no ompletion handler é atribuído à propriedade `movieInfo` (já declarada) , portanto, podemos usar os valores buscados em toda a classe.
- ✓ Em segundo lugar, usamos o thread principal ( `DispatchQueue.main` ) porque o `setValuesToViews()` método atualizará a interface do usuário e isso é algo que sempre deve acontecer no thread principal. Se o resultado acima for bem-sucedido e conseguirmos buscar um filme corretamente, seus detalhes serão preenchidos para as visualizações apropriadas. Isso é algo que você pode tentar mesmo agora se você executar o aplicativo e selecionar um filme:




Mas isso não é suficiente. Queremos ser capazes de atualizar o banco de dados e os dados para o filme específico, e acompanhar o estado assistido (se tivermos assistido ao filme), bem como avaliá-lo de acordo com o quanto gostamos dele. É fácil conseguir isso, pois só precisamos escrever um novo método na classe *DBManager* que executará a atualização. Então, de volta ao arquivo *DBManager.swift* , vamos adicionar o próximo:

```
func updateMovie(withID ID: Int, watched: Bool, likes: Int) {
    if openDatabase() {
        let query = "update movies set \(field_MovieWatched)=?,
\(field_MovieLikes)=? where \(field_MovieID)=?"

        do {
            try database.executeUpdate(query, values: [watched,
likes, ID])
        }
        catch {
            print(error.localizedDescription)
        }

        database.close()
    }
```

```
}
```

Esse método aceita três parâmetros:

o ID do filme que queremos atualizar, um valor *Bool* indicando se o filme foi assistido ou não e o número de curtidas que damos ao filme. Criar a consulta é fácil e de acordo com o que discutimos anteriormente. A parte interessante aqui é o método `executeUpdate(...)` já visto quando criamos o banco de dados. Esse método é aquele necessário  usar para executar qualquer tipo de alteração no banco de dados ou, caso contrário, você o utiliza quando não executa instruções *Select*. O segundo parâmetro desse método é novamente um array de *Qualquer* objeto que possa passar junto com a consulta que será executada. Opcionalmente, é possivel retornar um valor *Bool* para indicar se a atualização foi bem-sucedida ou não.

Vamos voltar agora para o arquivo *MovieDetailsViewController.swift* , já que é hora de usar o método acima. Encontre o método `saveChanges(_:)` IBAction e adicione o seguinte conteúdo:

```
@IBAction func saveChanges(_ sender: AnyObject) {
    DBManager.shared.updateMovie(withID: movieInfo.movieID,
watched: movieInfo.watched, likes: movieInfo.likes)

    _ = self.navigationController?.popViewController(animated:
true)
}
```

Com o acima adicionado, o aplicativo irá atualizar o filme com o estado assistido e os gostos toda vez que tocar no botão Save, e então ele retornará ao *MoviesViewController* .

## Excluir registros

Até agora, vimos como criar um banco de dados programaticamente, como executar instruções em lote, como carregar dados e como atualizar. Há uma última coisa que resta a ser vista, e é como excluir os registros existentes. Manteremos as coisas simples e permitiremos a exclusão de um filme deslizando uma célula para o lado esquerdo, de modo que o botão vermelho de exclusão seja exibido.

Vamos fazer uma visita pela última vez na classe *DBManager* . Nossa tarefa é implementar um novo método que realizará a exclusão do registro correspondente ao filme que selecionamos para excluir. Mais uma vez, você verá que o método `executeUpdate(...)` da classe *FMDatabase* será usado para executar a consulta que criaremos. Implemente o código abaixo:

```
func deleteMovie(withID ID: Int) -> Bool {
    var deleted = false

    if openDatabase() {
        let query = "delete from movies where
\(field_MovieID)=?"

        do {
            try database.executeUpdate(query, values: [ID])
            deleted = true
        }
        catch {
            print(error.localizedDescription)
        }

        database.close()
    }

    return deleted
}
```

Não há nada de novo aqui que mereça ser discutido, exceto pelo fato de que o método retorna um valor Bool para indicar se a exclusão foi bem-sucedida ou não. Precisamos dessas informações, porque precisamos atualizar a fonte de dados da tableview (o array `movies`) e da tableview conforme você veremos a seguir.

Agora vamos ao *MoviesViewController* e vamos implementar o seguinte método tableview:

```
func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
    if editingStyle == .delete {

    }

}
```

O código acima ativará o botão vermelho Delete e o disponibilizará para nós quando passarmos da direita para a esquerda. Completaremos a instrução `if` chamando o metodo `deleteMovie(_:)` e, se for bem-sucedida, removeremos o objeto *MovieInfo* correspondente do array `movies`. Por fim, vamos recarregar a tableview para fazer desaparecer a respectiva célula do filme:

```
func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
    if editingStyle == .delete {
        if DBManager.shared.deleteMovie(withID:
movies[indexPath.row].movieID) {
            movies.remove(at: indexPath.row)
            tblMovies.reloadData()
        }
    }
}
```

Agora você pode executar o aplicativo novamente e tentar excluir um filme. O banco de dados será atualizado com a exclusão do filme selecionado e esse filme não estará disponível sempre que você executar o aplicativo a partir de agora.