



PROF. RENÊ XAVIER

**DESENVOLVIMENTO PARA IOS 11 COM
SWIFT 4**

ONDE ENCONTRAR O MATERIAL?

[HTTPS://GITHUB.COM/RENEFX/IOS-2018-01](https://github.com/renefx/ios-2018-01)

GITHUB
ALÉM DO TRADICIONAL BLACKBOARD DO IESB

O QUE VAMOS FAZER HOJE?

AGENDA

- ▶ UserDefaults
- ▶ Banco de dados Mobile



BANCOS DE DADOS

Aula passada vimos a camada **Model**.

Um elemento muito presente nessa camada é a persistência de dados.

A persistência serve para guardar objetos da camada Model que criamos.

Vamos ver dois tipos:

- UserDefaults
- Bancos de dados



USER DEFAULTS

USER DEFAULTS

O que é?

- ▶ O UserDefaults é uma forma de armazenamento simples
- ▶ É um arquivo .plist (Property List) com um acesso pronto feito pela Apple
- ▶ Funciona como um dicionário (chave-valor)
 - ▶ Você atribui um valor a uma chave (uma string específica)
 - ▶ Você busca a informação contida naquela chave
- ▶ Persiste esses dados ainda que o usuário "mate" o App

USER DEFAULTS

Key	Type	Value
▼ Root	Dictionary	(3 items)
lingua	String	English
aparenciaDonoLoja	Boolean	YES
corDoApp	String	azul

```
var dicionario = [  
    "lingua": "English",  
    "aparenciaDonoLoja": true,  
    "corDoApp": "azul"  
] as [String : Any]
```

```
let corApp = dicionario["corDoApp"]
```

```
dicionario["aparenciaDonoLoja"] = false
```

USER DEFAULTS

Para quê serve?

- ▶ Armazenar/obter pequenas quantidades de dados, nada muito rebuscado
- ▶ Você pode usar para indicar flags:
 - ▶ Qual versão do Banco de dados aquele celular está: **dbVersion** (string).
 - ▶ Outra flag pode ser: **executouTutorial** (true/false).
 - ▶ Outra flag: **executouLogin (true/false)** - Sessão do usuário
 - ▶ Outra flag: **ultimaSincronizacaoServer (date)**.
- ▶ Geralmente colocamos configurações iniciais do App que podem ser alteradas, um bom exemplo são as preferências do App (tema do App)

USER DEFAULTS

SIMPLES DE USAR

```
UserDefaults.standard.s
    M Void set(url: URL?, forKey: String)
    M Void set(value: Any?, forKey: String)
    M Void set(value: Bool, forKey: String)
    M Void set(value: Double, forKey: String)
    M Void set(value: Float, forKey: String)
    M Void set(value: Int, forKey: String)
```

```
UserDefaults.standard.set(true, forKey: "executouLogin")
UserDefaults.standard.set("dark", forKey: "temaDoApp")
```

USER DEFAULTS

E SE EU PRECISAR CRIAR UMA KEY QUE NÃO EXISTE?

```
UserDefault.standard.set(true, forKey: "executouLogin")
UserDefault.standard.set("dark", forKey: "temaDoApp")
```

VOCÊ JÁ CRIOU!!

SEMPRE QUE VOCÊ ATRIBUI UM VALOR A UMA KEY QUE NÃO EXISTE,
ESSA KEY JÁ É CRIADA

USER DEFAULTS

COMO EU RECUPERO DADOS DO USERDEFAULTS?

```
UserDefault.standard(forKey: String)
```

- [M] URL? url(forKey: String)
- [M] Bool bool(forKey: String)
- [M] Data? data(forKey: String)
- [M] [Any]? array(forKey: String)
- [M] Float float(forKey: String)
- [M] Double double(forKey: String)
- [M] Any? object(forKey: String)
- [M] String? string(forKey: String)
- [M] Int integer(forKey: String)
- [M] [String : Any]? dictionary(forKey: String)
- [M] [String]? stringArray(forKey: String)

USER DEFAULTS

COMO EU RECUPERO DADOS DO USERDEFAULTS?

```
let executouLogin = UserDefaults.standard.bool(forKey: "executouLogin")
let temaDoApp = UserDefaults.standard.string(forKey: "temaDoApp")
```

ATENÇÃO: PODE SER NIL!

USER DEFAULTS

EU VI QUE EXISTE UM
OBJECT...

COMO EU FAÇO PRA
SALVAR E RECUPERAR UM
OBJETO QUE EU CRIEI?

USER DEFAULTS

Com o Swift 4, ficou muito mais fácil fazer isso, sua Model só precisa implementar o protocolo **Codable**

Esse protocolo não requer nenhuma implementação de métodos

```
class Usuario: Codable {  
    var nome: String  
    var idade: Int  
  
    init(nome: String, idade: Int) {  
        self.nome = nome  
        self.idade = idade  
    }  
}
```

USER DEFAULTS

Como era antes:

```
class Usuario: NSObject, NSCoding {
    var nome: String
    var idade: Int

    init(nome: String, idade: Int) {
        self.nome = nome
        self.idade = idade
    }

    required convenience init(coder aDecoder: NSCoder) {
        let idade = aDecoder.decodeInteger(forKey: "idade")
        if let nome = aDecoder.decodeObject(forKey: "nome") as? String {
            self.init(nome: nome, idade: idade)
        } else {
            self.init(nome: "", idade: idade)
        }
    }

    func encode(with aCoder: NSCoder) {
        aCoder.encode(nome, forKey: "nome")
        aCoder.encode(idade, forKey: "idade")
    }
}
```

USER DEFAULTS

Você precisa fazer o encoding da sua classe usando o **JSONEncoder**

Então é possível usar o **UserDefaults** set

```
let user = Usuario(nome: "José", idade: 30)
do {
    let encodedData = try JSONEncoder().encode(user)
    UserDefaults.standard.set(encodedData, forKey: "executouLogin")
} catch let error as NSError {
    print(error)
}
```

USER DEFAULTS

É possível encontrar explicações que apresentam assim:

É contar com a sorte

```
let user = Usuario(nome: "José", idade: 30)
let encodedData = try! JSONEncoder().encode(user)
UserDefaults.standard.set(encodedData, forKey: "executouLogin")
```

USER DEFAULTS

Apara salvar um array, é só enviar um Array!

```
let user = Usuario(nome: "José", idade: 30)
let user1 = Usuario(nome: "Wesley", idade: 28)
let users = [user, user1]
do {
    let encodedData = try JSONEncoder().encode(users)
    UserDefaults.standard.set(encodedData, forKey: "executouLogin")
} catch let error as NSError {
    print(error)
}
```

USER DEFAULTS

Essa chave é o que "trava" o seu objeto no UserDefault

Para atualizar um objeto... não existe atualizar é só salvar um novo objeto com a mesma key

```
let user = Usuario(nome: "José", idade: 30)
let encodedData = try! JSONEncoder().encode(user)
UserDefaults.standard.set(encodedData, forKey: "executouLogin")
```



USER DEFAULTS

Para recuperar:

Recupera-se o objeto com o UserDefaults data

Com esse objeto, usamos o JSONDecoder passando a informação de como ele é

```
if let decoded = UserDefaults.standard.data(forKey: "executouLogin") {  
    do {  
        let usuariosRecuperados = try JSONDecoder().decode(Usuario.self, from: decoded)  
        print(usuariosRecuperados)  
    } catch let error as NSError {  
        print(error)  
    }  
}
```



USER DEFAULTS

Para recuperar um **Array** é semelhante, só é necessário mudar o tipo que esperamos de retorno:

De **Usuario** para **[Usuario]**

```
if let decoded = UserDefaults.standard.data(forKey: "executouLogin") {  
    do {  
        let usuariosRecuperados = try JSONDecoder().decode([Usuario].self, from: decoded)  
        print(usuariosRecuperados)  
    } catch let error as NSError {  
        print(error)  
    }  
}
```

USER DEFAULTS

COMO APAGAR UMA KEY DO USERDEFAULTS?

```
UserDefaults.standard.removeObject(forKey: "executouLogin")
```

COMO APAGAR TODAS AS KEYS DO USERDEFAULTS?

```
if let appDomain = Bundle.main.bundleIdentifier {  
    UserDefaults.standard.removePersistentDomain(forName: appDomain)  
}
```

USER DEFAULTS

O usuário tem como apagar isso?

- ▶ Para o caso de atualização do App, **TODOS** os valores do seu UserDefaults são mantidos como na última versão do App, com a alteração que o usuário fez.
- ▶ Para o caso do usuário desinstalar o seu App e instalar novamente, os dados do seu UserDefaults serão perdidos e iniciarão tudo novamente como **uma instalação completamente nova** do App.

USER DEFAULTS

Últimas considerações:

- ▶ É **muito** simples de usar
- ▶ Se sua aplicação não precisa de um BD, mas precisa de armazenar informações use UserDefaults.
- ▶ Se sua aplicação teria um BD com uma única Model que não tem muitas entradas use UserDefaults.
- ▶ Lembre-se: não é para ser usado como um Banco de dados.
- ▶ O UserDefaults não tem a mesma velocidade que um Banco de Dados.
Lembre-se: é um arquivo plist, vai gastar o mesmo tempo que o tempo de leitura de um arquivo.

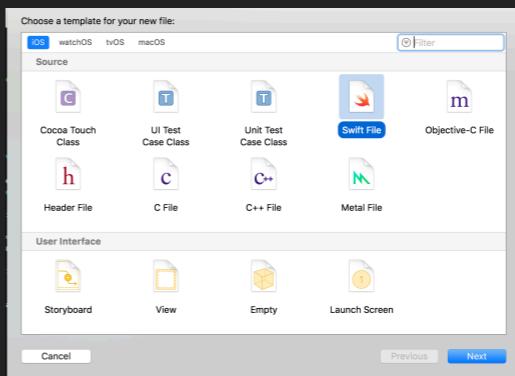


DICA

DICA

O usuário tem como apagar isso?

- ▶ O uso de muitas Strings pode ser problemático
- ▶ Um erro de digitação pode fazer errar a busca do objeto
- ▶ Que tal centralizar isso?
- ▶ Comece criando um novo Swift File
- ▶ Pode nomear Constants



DICA

- ▶ Nesse arquivos podemos concentrar nossas constantes que envolvem Strings
- ▶ Outros tipos também

```
struct UserDefaultsKeys {  
    static let usr_saved = "executouLogin"  
}
```

DICA

- ▶ Nesse arquivos podemos concentrar nossas constantes que envolvem Strings
- ▶ Outros tipos também
- ▶ Nas suas classes, você pode acessar assim:

```
if let decoded = UserDefaults.standard.data(forKey: UserDefaultsKeys.usr_saved) {  
    do {  
        let usuarioRecuperado = try JSONDecoder().decode(Usuario.self, from: decoded)  
        print(usuarioRecuperado)  
    } catch let error as NSError {  
        print(error)  
    }  
}
```

DICA

- ▶ É possível colocar outras structs no mesmo arquivo
- ▶ Tente agrupar em arquivos por temas se for o caso de ter muitas
- ▶ Tente colocar mensagens fixas do app (Erro, sem conexão...)
- ▶ Isso facilita encontrar e alterar caso haja algum erro

```
struct UserDefaultsKeys {
    static let usr_saved = "executouLogin"
}

struct Mensagens {
    static let noInternet = "O seu celular está sem internet"
    static let wrongLogin = "Usuário ou senha digitados errados"
}
```



BANCO DE DADOS

BANCOS DE DADOS

Pra quê usar?

- ▶ Salvar dados no dispositivo
- ▶ Nossos celulares (Android e iOS) possuem Bancos de Dados.
- ▶ Cada App controla seu BD e não tem acesso aos outros BD (segurança)
 - ▶ Esses dados serão acessíveis mesmo sem internet
 - ▶ Esses dados permanecerão acessíveis mesmo que o usuário "mate" o nosso App

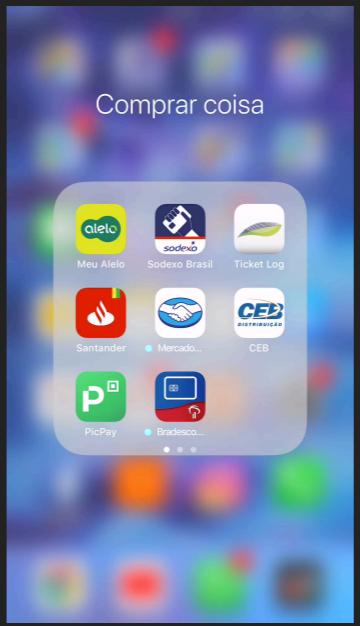
BANCOS DE DADOS

É obrigatório usar?

Não, até agora só fizemos Apps sem.

- ▶ Avalie se seu App pode ser usado sem internet
- ▶ Avalie ainda se não vale a pena só exibir uma tela estática quando estiver sem Internet

Só não pode ser como esse ➔



BANCOS DE DADOS

Existem três soluções para Banco de Dados pro iOS:

- ▶ SQLite
- ▶ Core Data
- ▶ Realm

Você pode escolher qualquer um, mas vou logo te avisando:

Não existe uma bala de prata!

Cada um tem seus prós e contras e para maioria dos App's a escolha não vai interferir em performance



SQLITE

- ▶ Solução open source
- ▶ É um banco de dados que você interage via SQL
- ▶ Tem certas limitações dependendo do que você está acostumado com SQL
(MySQL, PostgreSQL, Microsoft SQL Server, Oracle...)
- ▶ Ainda assim você consegue fazer tudo
- ▶ Tabelas, colunas, select
- ▶ Opera com os dados armazenados no disco em arquivos
- ▶ Dizem que é o mais usado

SQLITE

PRÓS

- ▶ Linguagem familiar
- ▶ Não precisa de configuração para adicionar ao projeto, já vem com
- ▶ Por já vir com o SQLite, isso também quer dizer que o seu uso não aumenta o tamanho do projeto
- ▶ Poder editar dados e drop tables sem ter que carregar na memória
- ▶ Seguro no caso de multithreads
- ▶ O seu banco de dados para o App Android e iOS pode ser exatamente o mesmo
- ▶ Fácil migração, pois você escreve a query

SQLITE

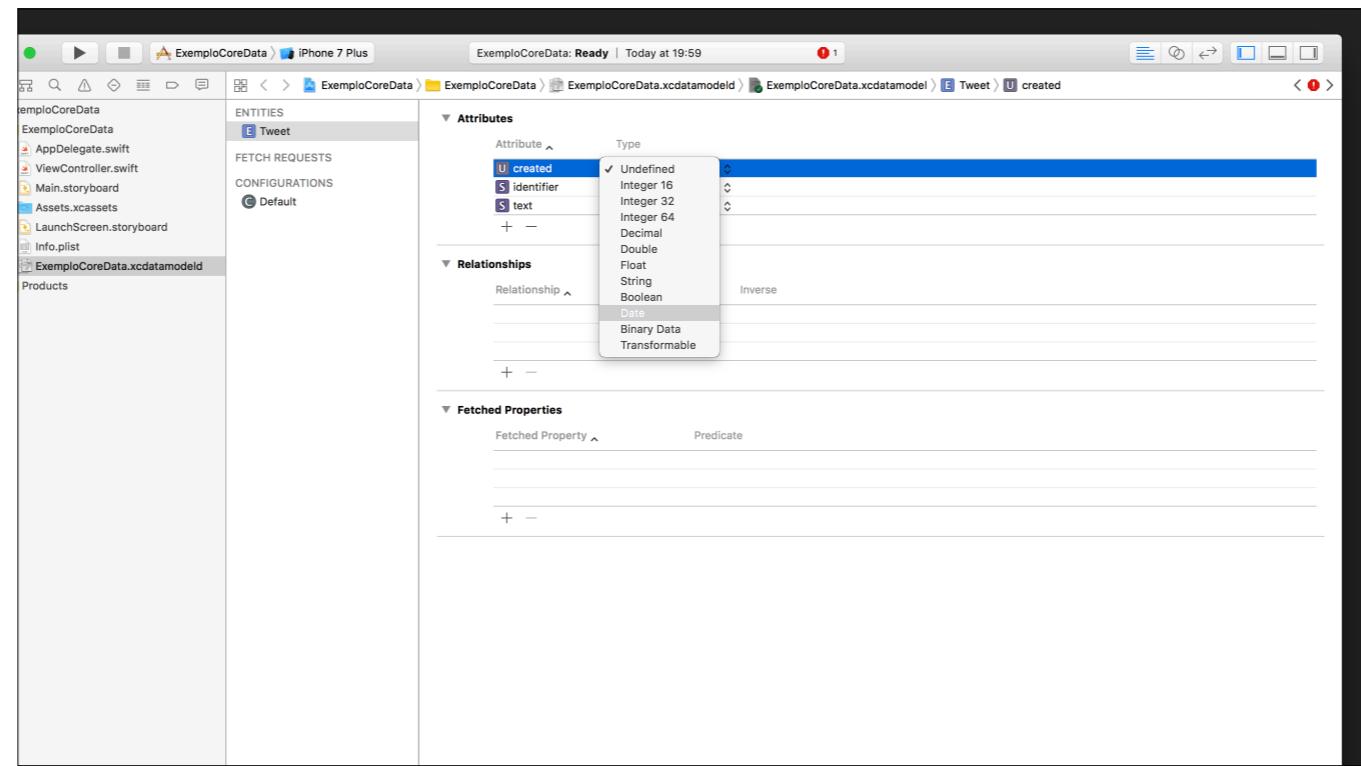
CONTRA

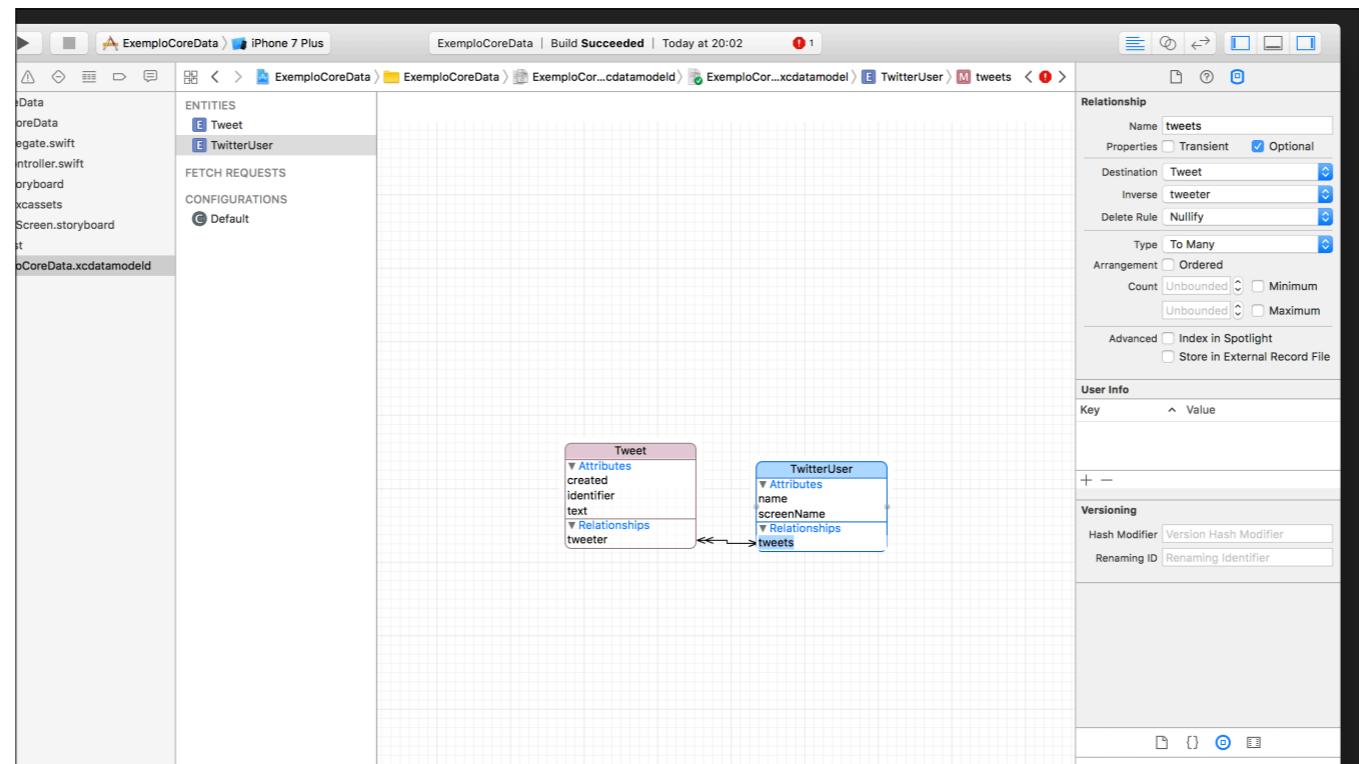
- É necessário escrever cada uma das suas querys perdendo um pouco de produtividade



CORE DATA

- ▶ Desenvolvido pela Apple
- ▶ Não é só um banco de dados, mas um framework.
 - ▶ Xcode tem uma ferramenta visual para montar o banco
 - ▶ Xcode gera as classes da sua Model
- ▶ Você resolve tudo em Swift
- ▶ Opera com os dados armazenados na memória prioritariamente
- ▶ Você trabalha com os objetos, não exatamente com querys
- ▶ Muito completo: Thread, Lazy load de objetos
- ▶ Parte da fundação é um SQLite, mas ele não trabalha somente dessa forma, tanto que os dados ficam na memória prioritariamente, não em arquivos





CORE DATA

PRÓS

- ▶ Suporte constante (Apple)
- ▶ Ferramenta de criação visual que já faz parte do Xcode
- ▶ Uma vez que você preenche o valor do seu objeto, automaticamente ele é salvo na memória
- ▶ Facilmente o mais robusto
- ▶ Simples inicialmente

CORE DATA

CONTRA

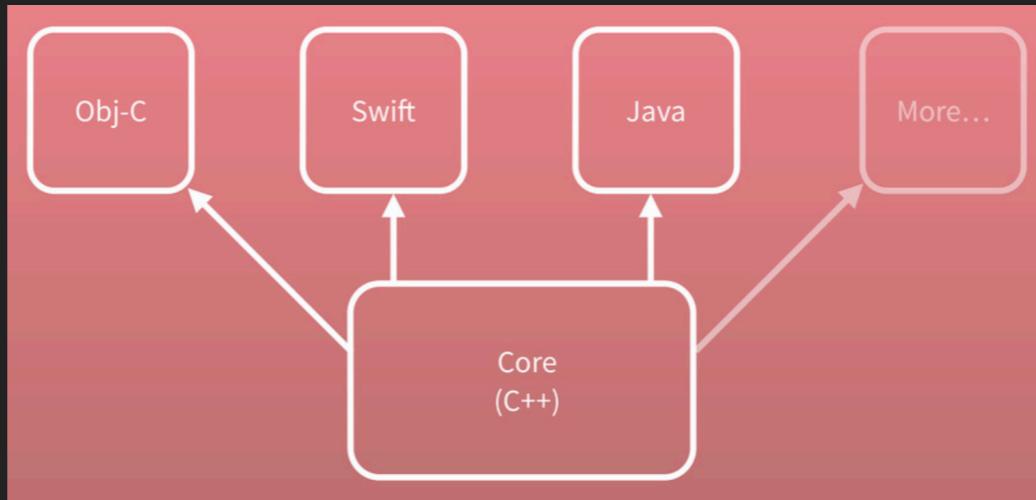
- ▶ Por ter que adicionar o framework, o tamanho do seu App aumenta
- ▶ Usa mais memória que o SQLite
- ▶ Não é preparado para automaticamente lidar com multithreads (thread-safe)
- ▶ Dados tem que ser carregados na memória, então um Drop irá carregar toda a tabela
- ▶ Seu banco é único para o iOS
- ▶ Difícil fazer uma migração que não seja a lightweight



REALM

- ▶ Solução privada, porém gratuita e open source
 - ▶ A solução paga envolve sincronização com a nuvem deles
- ▶ Foi desenvolvido após o SQLite e o Core Data
 - ▶ Desenvolvido em C++, diferente dos outros bancos até agora
- ▶ Feita para trabalhar com Swift, Obj-C, Java (Android), Kotlin ...
 - ▶ Não é um banco de dados relacional
 - ▶ Assim como o Core Data, trabalhamos com objetos, não com queries
- ▶ Apesar de ser o mais novo, tem ganho um grande peso

REALM



REALM

Trusted by Fortune 500 mainstays, innovative startups, and
#1-ranked app store successes

Realm is built into apps used by hundreds of millions of people every day.



REALM

PRÓS

- ▶ Ótima documentação e suporte (<https://realm.io/docs>)
- ▶ Uma vez que você preenche o valor do seu objeto, automaticamente ele é salvo na memória
- ▶ Objetos carregados de forma lazy - não precisa de paginação
- ▶ O mais simples de usar
- ▶ Pouco código adicional
- ▶ Fácil migração

REALM

CONTRA

- ▶ Por ter que adicionar o framework, o tamanho do seu App aumenta
- ▶ Não ordena por caracteres não latinos ('Latin Basic', 'Latin Supplement', 'Latin Extended A', 'Latin Extended B' (UTF-8 range 0-591))
- ▶ Não possui auto-increment
- ▶ Data e String tem tamanho máximo de 16MB