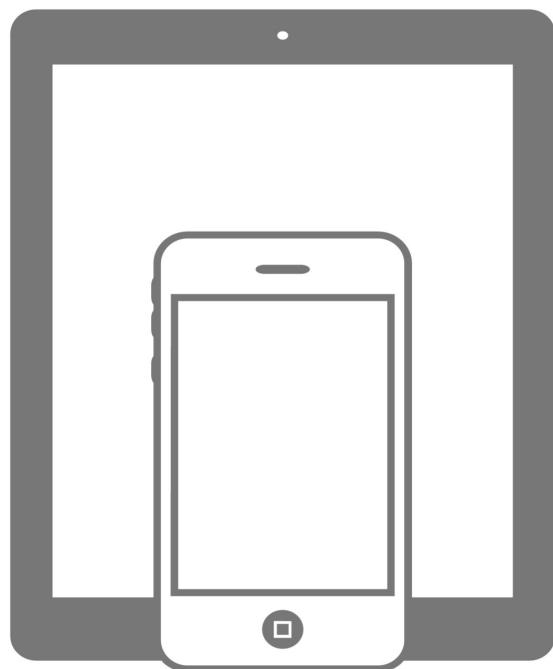


Desenvolvimento móvel com iOS

Curso IP-67



Sumário

1 Introdução	1
1.1 iPhone, iPad e o iOS	1
1.2 Boas práticas de design no iOS	1
2 Introdução nas linguagens utilizadas no desenvolvimento nativo para plataformas Apple	3
2.1 Estrutura de arquivos do Objective-C e controle de acesso	4
2.2 Estrutura de arquivos do Swift	4
2.3 Declaração de classes	5
2.4 Declaração de métodos	5
2.5 Invocando métodos	6
2.6 Interpolação e Concatenação de textos	7
2.7 Blocks e Closures	10
2.8 Alguns detalhes técnicos do Swift	15
3 Sua primeira aplicação	20
3.1 Conhecendo o Xcode: o ambiente de desenvolvimento da Apple	20
3.2 Criando um novo projeto	21
3.3 Exercício - Criando novo projeto no Xcode	23
3.4 Primeira Aplicação para iOS	25
3.5 Exercício - O simulador do Xcode	29
3.6 Exercício - Adicionando comportamentos, NSLog e o console do Xcode	33
4 Aplicação de Contatos	36
4.1 Motivação: Aplicação inspirada pelo aplicativo nativo Contacts.	36
4.2 Obtendo dados do usuário usando um formulário	36
4.3 Completando a tela do formulário	38
4.4 Exercício - Criando a tela do formulário	39
4.5 Criando um contato a partir dos dados do formulário	40
4.6 Exercício - Dados do formulário	42
5 Criando código de fácil manutenção: suas próprias classes	45

Sumário	Caelum
5.1 Classes e Objetos: compreendendo melhor os arquivos .h e .m	46
5.2 Arquivo Bridging-Header	46
5.3 Exercício - Criação da classe Contato	47
5.4 Instanciando objetos	50
5.5 Propriedades e o gerenciamento de memória	51
5.6 Instanciando e usando objetos	56
5.7 Exercício - Criando um Contato a partir de um formulário	58
6 Armazenando registros na memória: Array	60
6.1 Armazenando objetos em instâncias de Array	60
6.2 Armazenando contatos na memória	61
6.3 Customizando a inicialização de objetos: o método init	61
6.4 Boas práticas de Orientação a Objetos: dividindo responsabilidades	63
6.5 Exercício - Armazenando contatos com NSMutableArray	65
7 Um componente para listar registros: UITableViewController	68
7.1 View Controllers são apenas classes	69
7.2 Exercício - Um UITableViewController como tela principal da aplicação	73
8 Navegando entre telas de forma intuitiva	76
8.1 UINavigationController: o componente padrão para navegar entre telas	76
8.2 Configurando um título para uma tela exibida em um UINavigationController	80
8.3 Adicionando botões em um navigation bar	81
8.4 Transições simples entre telas	83
8.5 Exercício - Exibindo o formulário a partir da listagem	86
8.6 Boas práticas de Orientação a Objetos: dividindo responsabilidades	88
8.7 Exercício - Usando um UIBarButtonItem para armazenar um contato	93
9 Apresentando dados da mesma forma que as aplicações nativas: UITableView	98
9.1 UITableViewController: um view controller para listar registros	98
9.2 Exercício - Listando contatos	106
10 Usando o modo de edição nativo do UITableView	110
10.1 Precisa editar uma view? Use o padrão do iOS!	110
10.2 A fonte de dados de uma tabela: o UITableViewDataSource	111
10.3 Excluindo um registro da tabela	112
10.4 Exercício - Removendo um contato	114
11 Selecionando registros em um UITableView: editando os dados de um contato	116
11.1 Selecionando um registro da tabela	116
11.2 Exercício - recuperando registro selecionado em um UITableView	116

11.3 Exibindo um formulário já preenchido	117
11.4 Exercícios - passando um contato para o formulário	119
11.5 Escolhendo a ação de um botão no formulário	121
11.6 Exercícios - Botão específico para a ação de editar	123
12 Entendendo delegation e implementando o padrão com protocolos	125
12.1 Delegate: um padrão usado por todo o iOS	125
12.2 Exercício - criando um delegate	129
12.3 Destacando uma linha na tabela via seleção	131
12.4 Exercício - selecionando uma linha na tabela programaticamente	133
12.5 Removendo a seleção da linha	134
12.6 Exercício - deselecionando assíncronamente	135
13 Usando reconhecimento de gestos. Integração com telefone, mapas e mais	136
13.1 O evento long press: para sua aplicação se comportar como as nativas	136
13.2 Exercício - Gerenciando o evento de longPress e utilizando o UIActionSheet	140
13.3 Utilizando URIs para integração com outros aplicativos	142
13.4 Exercício - Integração com outros aplicativos através de URI's	145
14 Tirando proveito do hardware: interagindo com a câmera	147
14.1 Usando a biblioteca de imagens nativa do dispositivo	148
14.2 Exercício - selecionando uma imagem da biblioteca	152
14.3 Armazenando objetos do tipo imagem	155
14.4 Exercício - Armazenando a imagem do contato	156
14.5 UIAlertController - ActionSheet: uma forma nativa de selecionar uma ação	157
14.6 Acessando a câmera do dispositivo	158
14.7 Exercício (opcional): Escolhendo entre câmera ou biblioteca	160
15.1 Criando uma tela com o elemento MKMapView	162
15.2 Exercício - Criando outra view e adicionando o Mapa	163
15.3 Mostrando várias telas com o elemento UITabBarController	163
15.4 Exercício - Criando um UITabBarController e exibindo mais de uma tela	164
15.5 Alterando os ícones de uma UITabBarController	166
15.6 Carregando uma imagem do projeto para a aplicação	167
15.7 Exercício: Adicionando botões na UITabBar	168
15.8 Mostrando a localização atual do usuário	170
15.9 Simulando uma localização com o XCode	172
15.10 Exercício: Localizando o usuário e arquivo GPX	174
15 Buscando a localização geográfica do usuário	177
16.1 Gerenciando dependências com CocoaPods	178

16.2 Instalando TPKeyboardAvoiding	181
16.3 Melhorando nosso formulário com TPKeyboardAvoidingScrollView	182
16.4 Exercício: Utilizando CocoaPods e TPKeyboardAvoiding	183
16.5 Buscando as coordenadas do usuário a partir do seu endereço	185
16.6 Exercício: Fazendo geocode do endereço do usuário	189
16.7 Melhorando a usabilidade do usuário com UIActivityIndicatorView	191
16.8 Exercício: Adicionando um spinner ao efetuar o geocode	194
16.9 Visualizando os contatos no mapa	196
16.10 Qual o melhor momento para adicionar os contatos no mapa?	198
16.11 Exercícios: Mostrando os contatos no mapa	200
16.12 Customizando e adicionando comportamento ao pino	202
16.13 Exercício: Customizando a aparência do mapa	207
16 Trabalhando com banco de dados utilizando o Core Data	210
17.1 Banco de dados, o SQLite	210
17.2 Mapeamento Objeto Relacional	210
17.3 Utilizando o Data Modeler	211
17.4 Exercício - Criando o modelo de contato, utilizando o Data Modeler	212
17.5 Entendendo como obter o contexto do Core Data	213
17.6 Exercício - isolando os métodos de CoreData	216
17.7 Inserindo contatos no banco de dados	218
17.8 Trabalhando corretamente com a classe NSManagedObject	219
17.9 Exercício - Contatos como NSManagedObject	222
17.10 Armazenando configurações com UserDefaults	223
17.11 Exercícios - Carregando informações para o banco apenas uma vez com UserDefaults	224
17.12 Buscando dados com NSFetchedRequest	226
17.13 Exercícios - Listando contatos com NSFetchedRequest	227
17.14 Exercícios - Migrando a aplicação para o Core Data	227
17 Consumindo WebServices em nossa aplicação	229
18.1 O Serviço	229
18.2 Exercício: Efetuando cadastro	229
18.3 Preparando nosso app para consumir o serviço	231
18.4 Exercício: Criando e acessando a tela de temperatura	236
18.5 Consumindo o WebService	239
18.6 Exercício: Implementando integração	248
18 Apêndice A: Disponibilizando seu aplicativo na Apple App Store	251
19.1 App Store: a plataforma oficial de distribuição de aplicações da Apple	251

19.2 Burocracias e dificuldades para testar sua aplicação em um dispositivo	252
19.3 Sobre os guidelines da Apple para desenvolvimento de aplicações iOS	254

Versão: 20.6.23

INTRODUÇÃO

A invenção do telefone normalmente se dá a *Alexander Graham Bell*. Porém, em *15 de junho de 2002*, foi aceito pelo congresso dos EUA que o aparelho foi inventado por volta de 1860 pelo italiano **Antonio Meucci**, que o chamou de "telégrafo falante". A invenção se deu pois ele precisava falar com a sua esposa doente no andar de cima, sem precisar ficar gritando pela casa. Já o telefone celular nasceu em 1947 pelo laboratório Bell, nos EUA.

1.1 IPHONE, IPAD E O IOS

O iPad e o iPhone dispensam qualquer apresentação. Em 2007, no lançamento do iPhone, seu sistema operacional era conhecido por **iPhoneOS**. Apesar de haver uma relação forte com o Mac OS X e a Apple até falar isso no início, eles são sistemas distintos.

Com o sucesso do iPhone, a Apple se concentrou em seu desenvolvimento, renomeando-o para **iOS**, que agora roda no iPhone, iPod Touch, iPad e AppleTV. Apesar de ser um sistema operacional, a Apple não o licencia para rodar em hardwares não-Apple, diferentemente do que os concorrentes Android e Windows Mobile fazem.

1.2 BOAS PRÁTICAS DE DESIGN NO IOS

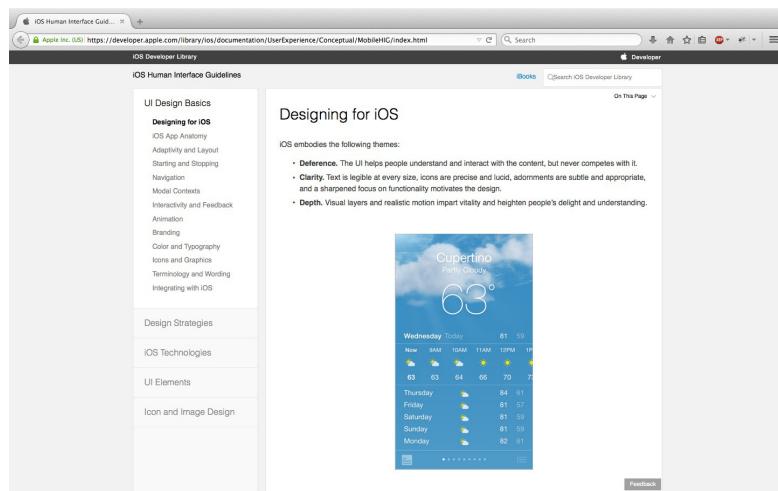


Figura 1.1: iOS HIGs

O iOS Human Interface Guidelines (HIG) é um importante guia, escrito pela própria Apple, com

dicas e truques para você não cair nos erros mais comuns ao desenhar sua aplicação. Entre alguns pontos principais, podemos destacar:

O menor componente de tela, para manter um certo conforto para o usuário, deve ser de 22 x 22 pixels. É importante conhecer bem a resolução das telas dos diversos dispositivos que rodam iOS:

- iPhone 6 Plus - 1920 x 1080 pixels
- iPhone 6 - 1334 x 750 pixels
- iPhone 5 - 1136 x 640 pixels
- iPhone 4 - 960 x 640 pixels
- iPads - 2048 x 1536 pixels

Estratégias de desenvolvimento:

- Enumere as funcionalidades que seu usuário pode querer
- Determine quem são seus usuários
- Seja objetivo em sua aplicação, sem rodeios para executar as ações
- Continue inovando na sua app para mantê-la sempre em foco

Os ícones da sua aplicação serão a primeira lembrança do usuário, ao procurar a app na lista do dispositivo.

Ícones do dispositivo:

- iPhone 6 Plus - 180 x 180
- iPhone 6, 5 e 4s - 120 x 120
- iPad - 152 x 152

Ícone da App Store:

- iPhone, iPod touch e iPad - 1024 x 1024

O guia completo pode ser encontrado no endereço abaixo: <http://bit.ly/Nl53cm>

INTRODUÇÃO NAS LINGUAGENS UTILIZADAS NO DESENVOLVIMENTO NATIVO PARA PLATAFORMAS APPLE

Para desenvolver aplicações nativas para os sistemas operacionais OS X e iOS, até pouco tempo atás era necessário utilizar a linguagem Objective-C . A partir de 2014 com o lançamento da linguagem Swift podemos optar por utilizar Objective-C , Swift ou **ambas** na mesma aplicação. Tendo essa possibilidade de escolher, é comum um desenvolvedores mobile se depararem com códigos escrito tanto em uma linguagem quanto na outra.

Atualmente a tendência é que os projetos iniciem utilizando Swift , porém o legado ainda é em Objective-C e, pensando nisso, é de suma importância que o desenvolvedor domine uma ou as duas linguagens. Caso ele não dominie as duas, é importante ao menos conhecer a outra linguagem (ou ter uma noção).

OBJECTIVE-C

Objective-C é uma linguagem de programação orientada a objetos criada na década de 80 e que adiciona transmissão de mensagens, inspirada pelo Smalltalk, por meio de uma fina camada escrita sobre o ambiente de execução da linguagem C.

Hoje em dia, é utilizada principalmente no Mac OS X e GNUstep, dois ambientes baseados no padrão OpenStep.

Vale notar que o Objective-C é realmente compatível com toda especificação do ANSI-C, sendo perfeitamente possível escrever um código puramente C e utilizar as bibliotecas já existentes. Há também diferenças em relação à orientação a objetos que estamos acostumados. No Objetive-C dizemos que os objetos trocam mensagens, enquanto em C# e Java nós invocamos métodos. Apesar disso, é frequente usarmos o termo *invocar um método* para Objective-C.

Mais informações sobre a história do OSX e da linguagem:
<http://osxbook.com/book/bonus/ancient/whatismacosx/history.html>

SWIFT

Swift é uma linguagem de programação multiparadigma criada pela Apple para desenvolvimento de programas para iOS e OS X. Apresentada na Apple's 2014 Worldwide Developers Conference, Swift pegou ideias de linguagens de programação como: Objective-C, Rust, Haskell, Ruby, Python, C#, CLU, entre outras.

A partir de 2015 Swift tornou-se um projeto open source e seu código fonte pode ser encontrado em <https://github.com/apple/swift>

Para mais informações sobre a linguagem: <https://swift.org/>

SWIFT OU OBJECTIVE-C

O foco do nosso curso é estudar o desenvolvimento de aplicações para iOS utilizando **Swift**. Como **Objective-C** ainda é uma realidade faremos uma pequena parte do projeto em **Objective-C** para sabermos integrar as duas linguagens. E assim, se cairmos em um projeto onde a linguagem seja **Objective-C**, não ficarmos perdidos e podermos fazer novas implementações já em **Swift**.

COMPARANDO A SINTAXE DAS DUAS LINGUAGENS

2.1 ESTRUTURA DE ARQUIVOS DO OBJECTIVE-C E CONTROLE DE ACESSO

Uma classe em *Objective-C* possui dois arquivos: um com a extensão ".h" e o outro com a extensão ".m".

Os arquivos com extensão ".h" são conhecidos como *header files*. É nesses arquivos que ficam declaradas as propriedades e comportamentos que queremos usar em nossa aplicação. Em orientação a objetos, essa é a forma de descrever dados e comportamentos que os objetos gerados a partir dessa classe terão. O que for declarado neste arquivo tem acesso liberado às demais classes.

Os arquivos com extensão ".m" possuem as implementações dos métodos declarados no *header file*. Aqui, dizemos ao programa como as propriedades devem ser utilizadas, implementando os comportamentos. Além disso, se declararmos o método somente no arquivo ".m", as demais classes não conseguirão acessá-lo. Podemos ter propriedades e métodos privados declarados neste arquivo.

2.2 ESTRUTURA DE ARQUIVOS DO SWIFT

Em Swift temos um unico arquivo com a extensão ".swift" onde declararemos nossa classe. Dentro dele definiremos modificadores de acessos para dizer o que é liberado ou não para as demais classes de nossa aplicação. Algo semelhando ao que é feito em Java , C# , Python entre outras linguagens.

2.3 DECLARAÇÃO DE CLASSES

Objective-C

Para declarar uma classe em Objective-C devemos declarar a interface da nossa classe em um arquivo ".h". No nosso caso "MyClass.h"

- Note que `interface` pode parecer com Java ou C# , mas o significado é bem diferente. Interface em Objective-C é utilizado para declarar uma classe:

```
#import <Foundation/Foundation.h>

@interface MyClass: NSObject

@end
```

Agora que já declaramos a nossa interface no arquivo ".h", devemos declarar nossa implementação em um arquivo ".m", no nosso caso "MyClass.m":

```
#import "MyClass.h"

@implementation MyClass

@end
```

Swift

Para declarar a mesma classe em Swift precisamos de um arquivo ".swift" no nosso caso "MyClass.swift":

```
import Foundation

class MyClass: NSObject {
```

2.4 DECLARAÇÃO DE MÉTODOS

Objective-C

Para declarar um método em Objective-C devemos declarar a assinatura do método no arquivo ".h". Os métodos em Objective-C tendem a ser bem verbosos:

```
#import <Foundation/Foundation.h>
```

```

@interface MyClass: NSObject

-(void)mostrarNoConsole:(NSString*) texto;

@end

```

- - define o modificador de acesso do método. - quer dizer que o método é de instância (ou seja preciso de uma instância do objeto para executá-lo). E + quer dizer que o método é da classe (executo o método a partir da classe)
- (void) é o tipo de retorno
- mostrarNoConsole é o nome do método.
- A partir do : são os parâmetros/argumentos do método que seguem o seguinte padrão: nomeDoMetodo:(Tipo) variável textoParaOProximoArgumento:(Tipo) variável; . No nosso caso NSString é o tipo que o Objective-C usa para representar texto.

Agora que já definimos a assinatura do método vamos declarar a implementação do mesmo no arquivo ".m":

```

#import "MyClass.h"

@implementation MyClass

-(void)mostrarNoConsole:(NSString*) texto{
    NSLog(texto);
}

@end

```

- Utilizamos a função NSLog para imprimir textos no console.

Swift

Para declarar um método em Swift basta adicioná-lo dentro da classe no arquivo ".swift":

```

import Foundation

class MyClass: NSObject {

    func mostrarNoConsole(texto:String){
        print(texto)
    }
}

```

- Swift utiliza o tipo String para representar textos, e a função print para imprimir textos no console.

2.5 INVOCANDO MÉTODOS

Objetive-C

Para utilizar nossa classe e invocar seu método `mostrarNoConsole`, teremos uma declaração da seguinte forma:

```
MyClass* myClass = [MyClass new];
[myClass mostrarNoConsole:@"Hello World"];
```

- A declaração `@"Hello World"` é uma forma de definir uma string literal em Objective-C

Swift

Para obtermos o mesmo resultado em Swift teremos a seguinte declaração:

```
let myClass = MyClass()
myClass.mostrarNoConsole("Hello World")
```

- A declaração `"Hello World"` é uma forma de definir uma string literal em Swift

2.6 INTERPOLAÇÃO E CONCATENAÇÃO DE TEXTOS

Objective-C

Para podermos interpolar textos em Objective-C podemos utilizar um método `stringWithFormat` da classe `NSString`:

```
NSString *nome = @"Fulano de Tal";
NSString *mensagem =
    [NSString stringWithFormat:@"Meu nome é %@", nome];
```

Na String que usamos como primeiro parâmetro para o método `stringWithFormat` apareceu um `%@`. Essa é a forma de especificar qual trecho da String (primeiro parâmetro) será substituído pelo próximo parâmetro passado para o método. E se houvesse mais de uma String que gostaríamos de substituir, por exemplo o nome e também o sobrenome? Sem problemas, basta criar dois "espaços para substituição" usando o `%@`, e passar os valores como parâmetro, veja:

```
NSString *nomeCompleto =
    [NSString stringWithFormat:@"Sou o %@ %@", @"Fulano", @"de Tal"];
```

A classe `NSString` também oferece uma maneira de se formar uma string a partir de objetos de outros tipos que não são string. Um exemplo de uso seria interpolar uma String e um número para formar uma nova String. Vamos criar uma variável com um nome e uma outra com a idade de uma pessoa:

```
NSString *nome = @"Steve Jobs";
int idade = 61;
```

Como prometido, vamos usar o método `stringWithFormat`: para interpolar a String e também o número. Dessa vez além do `%@` que marca uma posição a ser substituída para uma string, também vamos usar o `%d` que permite criar espaço para um número:

```
NSString *nome = @"Steve Jobs";
int idade = 61;

NSString nomeIdade =
    [NSString stringWithFormat:@"O nome é %@ e a idade: %d", nome, idade];
```

Como o método sabe que a variável `nome` deverá entrar no lugar do `@` e a variável `idade` no lugar do `%d`? Os parâmetros são substituídos na ordem em que aparecem na invocação do método! A função `NSLog` também suporta esses marcadores para substituição, poderíamos escrever o seguinte código para testar isso:

```
NSString *nome = @"Steve Jobs";
int idade = 61;

NSLog(@"O nome é %@ e a idade: %d", nome, idade);
```

Você deve ter notado que, na String utilizada como base no último exemplo, foram utilizados **símbolos** diferentes para indicar o espaço que deverá ser preenchido por uma **String** e o espaço preenchido por um **número**. Isso porque esses dois valores têm significados diferentes dentro da linguagem Objective-C. Existem vários tipos nativos da linguagem e vários outros que podemos adicionar conforme escrevemos nossos programas.

Além do `%@` e do `%d` há diversos outros marcadores que podem ser utilizados. Veja mais informações abaixo:

CARACTERES ESPECIAIS PARA A FORMATAÇÃO DE STRINGS

Outros **wildcards** que podem ser usados no NSLog ou no método `stringWithFormat` são:

- %@ Object
- %d, %i signed int
- %u unsigned int
- %f float/double
- %x, %X hexadecimal int
- %o octal int
- %zu size_t
- %p pointer
- %e float/double (in scientific notation)
- %g float/double (as %f or %e, depending on value)
- %s C string (bytes)
- %S C string (unichar)
- %.*s Pascal string
- %c character
- %C unichar
- %lld long long
- %llu unsigned long long
- %Lf long double

Para concatenar strings em Objective-C usamos o método `stringByAppendingString` da classe `NSString`:

```
NSString *bruce = @"Bruce";
NSString *batman = [bruce stringByAppendingString:@" Wayne"];
```

Como as Strings no Objective-C são imutáveis, se quisermos ter a String atualizada, precisamos atribuir o valor para uma nova variável.

Podemos ainda utilizar uma versão mutável da String para não precisarmos criar esta String extra na memória utilizando a classe `NSMutableString`:

```
NSMutableString *spiderman = [NSMutableString stringWithFormat:@"Peter"];
[spiderman appendString:@" Parker"];
```

- Por causa destas características, é muito mais comum vermos interpolação em códigos escritos

em Objective-C do que a concatenação.

Swift

Para interpolar Strings utilizando `Swift` temos a seguinte declaração:

```
let nome: String = let nome = "Fulano de Tal";
let mensagem: String = "Meu nome é \(nome)"
```

Se quisermos concatenar ao invés de interpolar podemos fazer:

```
let nome: String = let nome = "Fulano de Tal";
let mensagem: String = "Meu nome é " + nome
```

Uma coisa legal de utilizar *interpolação* é que podemos passar qualquer objeto para interpolar com uma String.

Para interpolar outros tipos de objetos que não são Strings temos a seguinte declaração:

```
let nome: String = "Steve Jobs";
let idade: Int = 61;

let mensagem: String = "O nome é \(nome) e a idade \(idade)"
```

2.7 BLOCKS E CLOSURES

Tanto em `Swift` quanto em `Objective-C` temos uma funcionalidade de poder criar trechos de códigos e passa-los como argumentos em nossos métodos como um `callback` ou até mesmo referenciar esse trecho de código a uma variável para chamar futuramente. Essa funcionalidade geralmente são chamadas de `funções anônimas`, `code Block`, `closure`, `expressão lambda` entre outros.

Esse funcionalidade é chamada de *blocks* em `Objective-C`, e de *closures* em `Swift`.

Essa técnica é muito útil quando precisamos delegar alguma responsabilidade para quem está chamando nosso método.

Vamos agora verificar como usar essas funcionalidades nas duas linguagens.

Objective-C

A sintaxe geral para declaração de um *block* é:

```
tipoDeRetorno (^nomeDoBlock)(parametros...);
```

Podemos comparar essa sintaxe que declara um *block* com a assinatura de um método; Onde temos o tipo de retorno, nome e parametros.

A primeira vista a sintaxe de um *block* pode parecer bem estranha e complicada, mas quando

começamos a utiliza-la, vemos que ela é bem simples.

Vamos à um exemplo, vamos declarar um *block* que não retorna nada e somente imprime o que ele receber por parâmetro no console.

```
//declarando um blocks sem implementação sem retorno e que recebe um NSString como parâmetro.
void (^printConsole)(NSString *);

//criando uma implementação para o block
printConsole = ^(NSString *console){
    NSLog(@"%@", console)
};

//usando o block
printConsole(@"Sistema conectado");
```

Essa mesma declaração poderia ser simplificada da seguinte maneira:

```
//declarando um blocks com implementação.
void (^printConsole)(NSString *) = ^(NSString *console){
    NSLog(@"%@", console)
};

//usando o block
printConsole(@"Sistema conectado");
```

Vamos agora à um exemplo de um *block* que retorna um valor:

```
double (^calculo)(double, double, double);

//Delta
calculo = ^(double a, double b, double c){
    return (b*b) - 4*a*c;
}

double resultado = calculo(0,2,4);

NSLog(@"resultado: %f", resultado);
```

No momento conseguimos declarar *blocks* com um unico nome, mas e se eu quiser ter uma assinatura genérica e reutilizar essa assinatura em mais de uma implementação?

Para isso vamos recorer a utilização de uma funcionalide do C chamada *typedef*. Com ela conseguimos criar um tipo customizado (um apelido para um tipo já existente).

```
typedef double (^calculoCom3Variaveis)(double, double, double);
```

Com essa declaração criamos um tipo com o nome do nosso *block*, e conseguimos criar váriaveis com esse novo tipo, cada uma com sua devida implementação.

```
typedef double (^CalculoCom3Variaveis)(double, double, double);

CalculoCom3Variaveis delta = ^(double a, double b, double c){
    return (b*b) - 4 * a * c;
}
```

```

CalculoCom3Variaveis bhaskaraPositivo = ^(double a, double b, double c){

    double resultadoDelta = delta(a, b, c);
    double raizDelta = sqrt( resultadoDelta );

    return ( -b + raizDelta ) / ( 2 * a);
}

CalculoCom3Variaveis bhaskaraNegativo = ^(double a, double b, double c){

    double resultadoDelta = delta(a, b, c);
    double raizDelta = sqrt( resultadoDelta );

    return ( -b - raizDelta ) / ( 2 * a);
}

double resultadoPositivo = bhaskaraPositivo(1, 8, -9);
double resultadoNegativo = bhaskaraNegativo(1, 8, -9);

NSLog(@"%@",[resultadoPositivo, resultadoNegativo"]);

```

Uma das forma mais utilizadas de *blocks* é como argumento de um método

```

@interface MyClass : NSObject

-(void)fazAlgumaCoisaCom: (void(^)(double a, double b))block;

@end

@implementation MyClass

-(void)fazAlgumaCoisaCom:(void (^)(double a, double b))block{
    NSLog(@"Faz alguma coisa antes");

    block(1, 2);

    NSLog(@"Faz alguma coisa depois");
}

@end

```

Dessa forma alguém que utilizar `MyClass` e invocar o método `fazAlgumaCoisaCom` tem que passar um *block* que será chamado dentro da execução do método (callback).

```

MyClass* myClass = [MyClass new];

[myClass fazAlgumaCoisaCom:^(double a, double b) {

    NSLog(@"Parametros recebidos: {a=%f, b=%f}", a, b);

}];

```

Ao executarmos esse código teremos o seguinte resultado no console:

```

Faz alguma coisa antes
Parametros recebidos: {a=1.000000, b=2.000000}
Faz alguma coisa depois

```

Dessa forma conseguimos delegar a execução para quem estiver usando nossa classe.

Para saber mais Acesse o link: <http://goshdarnblocksyntax.com/>

Swift

Em Swift temos algo análogo ao *block* de Objective-C , essa funcionalidade chama-se *closure*.

A sintaxe geral de declaração de *closure* é:

```
{ (parametros) -> tipoDeRetorno in  
    //corpo da função  
};
```

Vamos ver alguns exemplos de utilização

```
//Declarando a variável que será um closure e que deve receber uma String e não retornar nada  
var printConsole: (String) -> Void  
  
//Defindo uma implementação para o closure  
printConsole = { (console) in  
    print("INFO - [ \(console) ]")  
}  
  
printConsole("Sistema Conectado")
```

A mesma declaração poderia ser simplificada da seguinte forma:

```
//Declarando a variável que será um closure e que deve receber uma String e não retornar nada e sua  
//devida implementação  
var printConsole = { (console: String) -> Void in  
    print("INFO - [ \(console) ]")  
}  
  
printConsole("Sistema Conectado")
```

Da mesma forma que conseguimos criar um apelido para um *block* também conseguimos apelidar um *closure*. Para isso usamos *typealias* Apelido = *closure*:

```
var printConsole = { (console: String) -> Void in  
    print("INFO - [ \(console) ]")  
}  
  
typealias CalculoCom3Variaveis = (Double, Double, Double) -> Double  
  
let delta: CalculoCom3Variaveis = { (a,b,c) in return (b*b) - 4 * a * c }  
let bhaskaraPositivo: CalculoCom3Variaveis = { (a,b,c) in  
    let resultadoDelta = delta(a,b,c)
```

```

let raizDelta = sqrt(resultadoDelta)

return (-b + raizDelta) / (2 * a)
}

let bhaskaraNegativo: CalculoCom3Variaveis = { (a,b,c) in

let resultadoDelta = delta(a,b,c)
let raizDelta = sqrt(resultadoDelta)

return (-b - raizDelta) / (2 * a)
}

let resultadoPositivo = bhaskaraPositivo(1, 8, -9)
let resultadoNegativo = bhaskaraNegativo(1, 8, -9)

printConsole("Resultado: \(resultadoPositivo)", \(resultadoNegativo) " ")

```

Assim como em Objective-C também podemos receber uma função anônima como parâmetro de um método/função.

```

class MyClass {
    func fazAlgumaCoisaCom(closure: (Double, Double) -> () ){
        print("Faz alguma coisa antes")

        closure(1,2)

        print("Faz alguma coisa depois")
    }
}

let myClass = MyClass()

myClass.fazAlgumaCoisaCom( { (a, b) in
    print("Parametros recebidos: {a=\(a), b=\(b)}")
})

```

Além disso no Swift temos algumas funcionalidades quando trabalhamos com *closures* para deixar nosso código mais expressivo

Trailing Closures Syntax

Quando temos um *closure* como último argumento de uma função, podemos passar ele fora dos parênteses dos argumentos da função:

```

func foreach(array: [AnyObject], closure: (AnyObject) -> () ){
    for anObject in array {
        closure(anObject)
    }
}

foreach(["Bruce", "Petter", "Clark", "Barry"]) { (nome) in
    print(nome)
}

```

```
foreach([1, 2, 3, 3]) { (num) in
    print(num)
}
```

Shorthand argument names

Podemos nos referenciar aos parâmetros que recebemos nos *closures* através da sua posição: \$0 para o primeiro parâmetro, \$1 para o segundo, \$2 para o terceiro e assim por diante. Para isso devemos omitir a primeira parte do *closure* (justamente os parâmetros).

```
func foreach(array: [String], closure: (String) -> () ){
    for anString in array {
        closure(anString)
    }
}

foreach(["Bruce", "Petter", "Clark", "Barry"]) { print($0) }
```

Para saber mais Acesse os links:
https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Closures.html <http://goshdarnclosuresyntax.com/>

2.8 ALGUNS DETALHES TÉCNICOS DO SWIFT

As duas linguagens são muito poderosas, e conseguimos fazer muitas coisas com elas. Agora que vimos as semelhanças entre elas vamos entrar em mais detalhe na linguagem que será usada durante o curso.

Swift é uma linguagem totalmente nova e está em constante evolução, vamos abordar alguns conceitos que devemos saber antes de começar nossos desenvolvimentos.

ponto-e-virgula (;

Na grande maioria das linguagens usamos o ponto-e-virgula para indicar que terminamos uma instrução e iremos iniciar a próxima.

Em Swift o uso do ponto-e-virgula para indicar que terminamos uma declaração é opcional.

let ou var

Em swift quando precisamos de um valor mutável utilizamos a declaração var , já quando precisamos de um valor que não mude (constante) utilizamos a declaração let . Isso é útil pois conseguimos restringir nas nossas classes que somente os atributos necessário seja alterado.

Inferência de Tipos

Em swift é opcional declarar o tipo de uma variável/constante. Caso não seja explícito o tipo dessa variável, será assumido o tipo do conteúdo dessa variável.

```
let nome = "Bruce"  
é o mesmo que  
let nome:String = "Bruce"
```

A partir do momento que definimos um conteúdo para uma variável/constate o tipo desse conteúdo será o utilizado durante todo o ciclo de vida da mesma.

Generics

Generics resolve o problema de termos sobrecarga de métodos para atender a mesma funcionalidade para tipos diferentes. Por exemplo nossa função `foreach`, da forma que fizemos só conseguimos iterar sobre um array de `String`.

```
class Iterator {  
  
func foreach(array: [String], closure: (String) -> () ){  
    for anString in array {  
        closure(anString)  
    }  
}  
}
```

E se tivessemos a necessidade de iterar sobre um array de `Double` ou de `Int`? Bom uma solução poderia ser fazer um overload (sobrecarga) do método:

```
class Iterator {  
  
func foreach(array: [String], closure: (String) -> () ){  
    for anString in array {  
        closure(anString)  
    }  
}  
  
func foreach(array: [Double], closure: (Double) -> () ){  
    for anDouble in array {  
        closure(anDouble)  
    }  
}  
  
func foreach(array: [Int], closure: (Int) -> () ){  
    for anInt in array {  
        closure(anInt)  
    }  
}  
}
```

Dessa forma conseguimos iterar sobre `String`, `Double` e `Int`, mas percebam que temos 3 vezes o

mesmo método, e com isso temos um problema com a manutenção do código. Caso altere alguma coisa na forma como iteramos sobre objetos, teremos que propagar essa alteração método a método.

Uma outra alternativa seria utilizar Polimorfismo. Ao invés de iterarmos sobre um tipo específico podemos iterar sobre um tipo mais alto na hierarquia `AnyObject`.

```
class Iterator {  
  
    func foreach(array: [AnyObject], closure: (AnyObject) -> () ){  
        for anObject in array {  
            closure(anObject)  
        }  
    }  
}
```

O problema dessa implementação é que não podemos invocar um método de uma implementação (a menos que efetuemos um cast). Ou seja só poderei invocar métodos de `AnyObject`, caso eu precise invocar um método específico terei que fazer um cast para o tipo específico e depois invocar o método.

Pensando nessa situação, não queremos especificar um tipo em tempo de compilação, mas queremos que nosso método use o tipo correto que foi passado para essa classe. Ou seja no momento de compilação aceite qualquer tipo, e em execução use o tipo correto que foi passado para a classe.

E é justamente nesse ponto que os `Generics` nos ajudam, podemos escrever a mesma classe para que ela seja usada por qualquer tipo utilizando `generics`.

```
class Iterator<T> {  
  
    func foreach(array: [T], closure: (T) -> () ){  
        for anObject in array {  
            closure(anObject)  
        }  
    }  
}
```

Com essa declaração, estamos dizendo que nossa classe vai trabalhar com um tipo genérico `T`. E quem for usar essa classe especifique qual é o valor para esse tipo `T`.

```
let iteratorString = Iterator<String>()  
  
iteratorString.foreach(["Bruce", "Petter", "Clark", "Barry"]) { print($0) }  
  
let iteratorInt = Iterator<Int>()  
  
iteratorInt.foreach([10, 20, 30, 40, 50]) { print($0) }
```

Optional

Vamos pegar como exemplo a seguinte declaração:

```
let stringNumero = "123"
```

Agora que converter essa `String` para um `Int`, para isso a classe `Int` tem um construtor que

recebe uma `String` e tenta converter para `Int`. Nesse caso nosso código ficaria assim:

```
let stringNumero = "123"  
let numero = Int(stringNumero)
```

Até aqui sem problemas, temos uma `String` com o valor "123" e criamos um número inteiro a partir dessa `String`.

Mas o que aconteceria, se nossa `String` tivesse como valor "A" ao invés de "123"?

Teríamos um erro, lançado pela classe `Int` pois foi passado para ela um valor que ela não consegue converter para inteiro.

É justamente para evitar esse tipo de erro que temos o `optional`. Como o próprio nome já diz um `optional` é um valor que pode não estar presente quando formos usa-lo. Ou seja opcional.

Para dizer que um valor é opcional utiilzamos `?` após a declaração.

```
func devolveValorOpcional(String numero) -> Int? {  
    // converte para inteiro  
}
```

Podemos também ter variáveis com valores opcionais:

```
var optionalNumero?  
ou  
var optionalNumero:Int?
```

No nosso exemplo o construtor que recebe uma `String` e retorna um `Int`, não retorna um valor do tipo `Int` ele retorna um valor do tipo `Int?`. Como podemos passar para ele um valor que ele não consiga converter para interio ele retorna um valor opcional. Dessa forma fica a critério de quem estiver usando esse `optional` verificar se o valor está ou não presente.

Para saber se o valor está ou não presente podemos verificar se o `optional` está nulo:

```
let stringNumero:String = "123"  
let numero:Int? = Int(stringNumero)  
  
if numero != nil {  
    print(numero)  
}else{  
    print("não tem numero")  
}
```

Dessa forma caso exista um valor para a variável número ele será impresso. Só tem um problema, nossa variável número não armazena um valor inteiro ela armazena um valor opcional inteiro. E o resultado no console seria assim `Optional(123)`.

Ou seja realmente existe um valor dentro desse opcional. Esse opcional está embrulhando nosso valor, o que preciso agora é desembrulhar esse valor.

Para desembrulhar o valor de um `optional` utilizamos `!` após a declaração.

```
let stringNumero:String = "123"
let numero:Int? = Int(stringNumero)

if numero != nil {
    print(numero!)
}else{
    print("não tem numero")
}
```

Utilizando `!` estamos forçando o valor que está contido no `optional` a ser desembrulado.

Caso não estivesssemos fazendo a checagem se o valor está presente ou não e nossa `String` não fosse um número. Ou seja se nosso código estivesse assim:

```
let stringNumero:String = "A"
let numero:Int? = Int(stringNumero)

print(numero!)
```

Seria lançado uma exceção `fatal error: unexpectedly found nil while unwrapping an Optional value`. Pois usamos `!` para forçar a ser desembrulado um valor nulo.

Uma outra forma de desembrular um `optional` é utilizando uma declaração `if let`

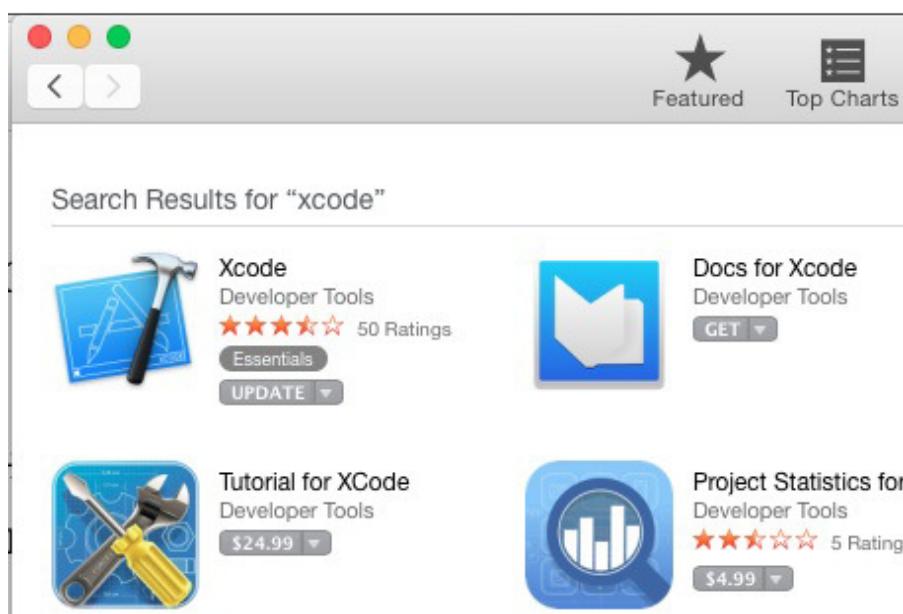
```
let stringNumero:String = "123"
let numero:Int? = Int(stringNumero)

if let numeroConcreto = numero {
    print(numeroConcreto)
}else{
    print("não tem numero")
}
```

SUA PRIMEIRA APLICAÇÃO

3.1 CONHECENDO O XCODE: O AMBIENTE DE DESENVOLVIMENTO DA APPLE

O Xcode é a ferramenta da Apple para o desenvolvimento mobile e desktop com uma série de utilidades, além de compilador e diversos extras. Podemos fazer seu download diretamente na App Store do Mac, gratuitamente:



O Xcode é uma IDE (*Integrated Development Environment*) bastante poderosa. Ao instalá-lo também ganhamos toda a documentação para **iOS** e **Mac OS X**, **SDKs** (*Software Development Kits*), além de ferramentas para testes, profiling e publicação na App Store. O Xcode permite acesso fácil a todas essas aplicações dentro da própria ferramenta.

DIVERSAS FERRAMENTAS DISPONÍVEIS

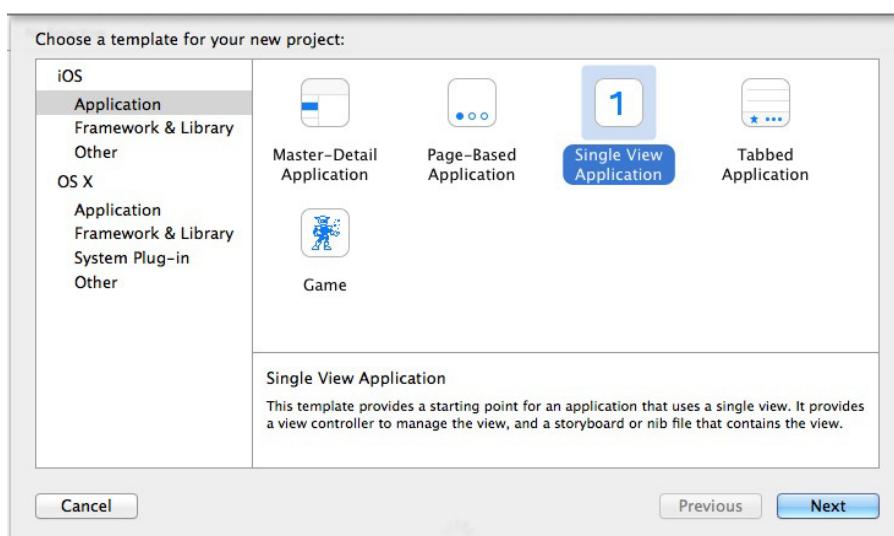
Não se preocupe em entender todas as ferramentas presentes ao instalar o Xcode, vamos conversar mais sobre elas no decorrer do curso.

3.2 CRIANDO UM NOVO PROJETO

Ao iniciar o Xcode, um diálogo será exibido para que você escolha entre a edição de um projeto já existente ou a criação de um novo. Vamos criar um novo projeto: selecione a opção **Create a new Xcode project**.



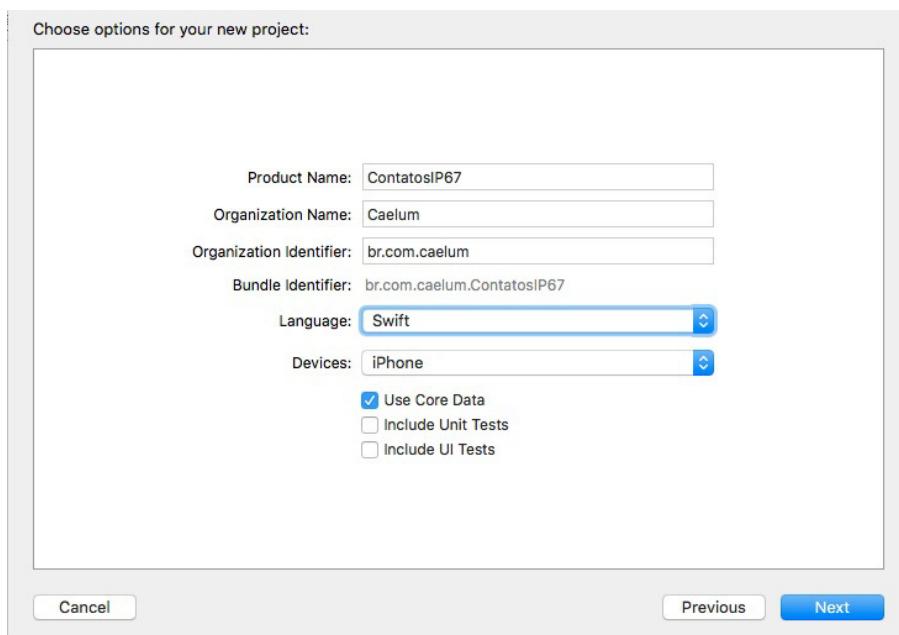
Será exibida uma tela com diversas configurações de projeto. Cada uma delas gera algum código específico para facilitar o início de um projeto, já com aparência e comportamento básico pré-estabelecido. O Xcode chama esse conjunto de código inicial de uma aplicação de *template*. Para nossa aplicação, vamos selecionar o template *Single View Application*.



ALTERANDO AS CONFIGURAÇÕES

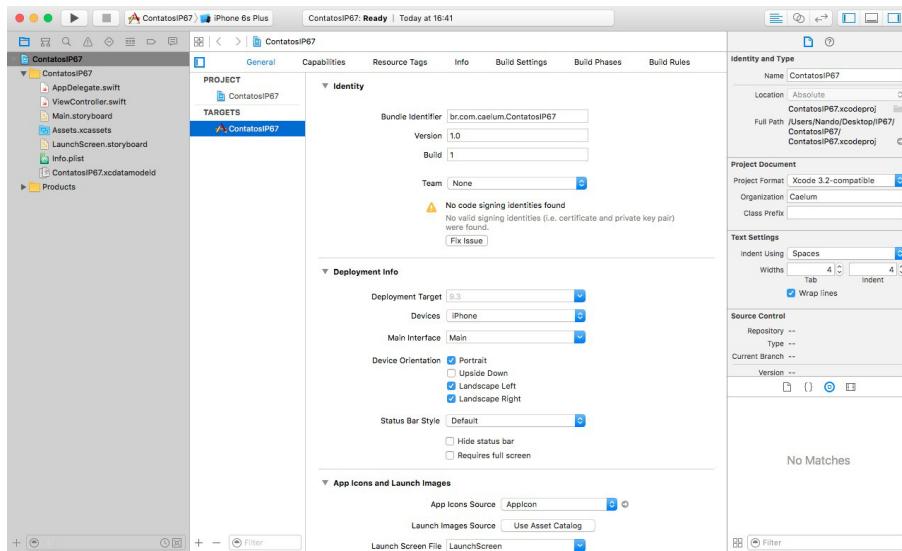
Não se preocupe em escolher o template correto logo de início, você poderá alterar as configurações que foram usadas para criar o seu projeto posteriormente. Durante o curso você vai compreender mais sobre cada um dos templates fornecidos pelo Xcode.

O Xcode vai mostrar uma nova janela para que você faça algumas configurações adicionais. É nessa etapa que escolheremos um nome para o projeto. Vamos criar um projeto que será baseado na lista de contatos nativa do iPhone, então para o `Product Name` colocamos `ContatosIP67`. Agora, podemos configurar o *Organization Name* e o *Organization Identifier*, esses nomes serão utilizado quando formos distribuir a aplicação, por enquanto, vamos usar `Caelum` e `br.com.caelum` respectivamente. Em `Language` vamos selecionar `Swift` e em `Devices`, `iPhone`. Marcaremos também o checkbox de `Core Data` que veremos mais tarde. No final, essa tela deverá estar como a seguinte:



Na próxima tela, selecione a pasta de destino para salvar o projeto. Neste diretório, o Xcode vai criar alguns arquivos, que serão vistos com mais detalhes mais para frente.

Após a criação do projeto, vamos visualizar a tela principal do Xcode, que deverá ser parecida com o exemplo:



Do lado esquerdo, o Xcode apresentará uma estrutura de diretórios e arquivos que representa o seu projeto. Veja que já foram criados vários arquivos, essa estrutura variará conforme o template selecionado no início da configuração. A ideia dos templates é fornecer um esqueleto inicial.

Nessa tela, é possível editar diversas configurações relativas ao projeto. O Xcode fornece uma interface visual para isso.

CONFIGURAÇÕES AVANÇADAS DE UM PROJETO

Não se preocupe se parecer que estamos passando rapidamente por muitas configurações. Iremos abordar diversas delas no decorrer do curso.

Está pronto para criar sua primeira aplicação?

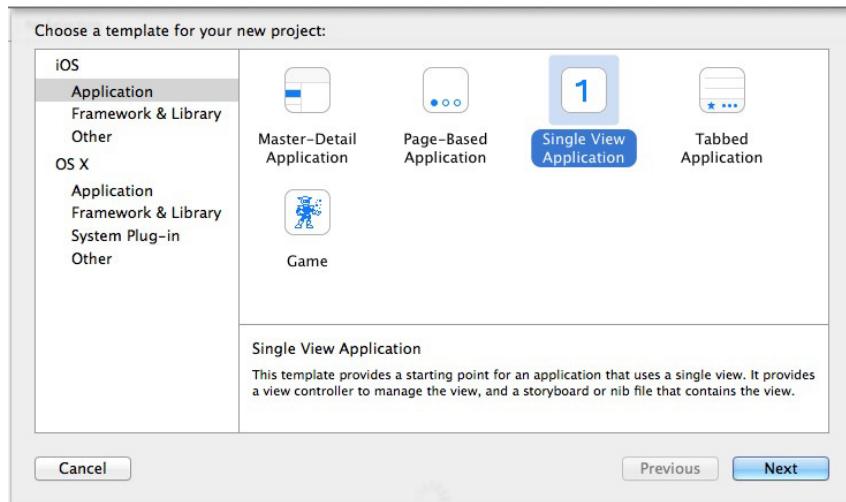
3.3 EXERCÍCIO - CRIANDO NOVO PROJETO NO XCODE

1. Primeiramente, é preciso criar o projeto no Xcode:

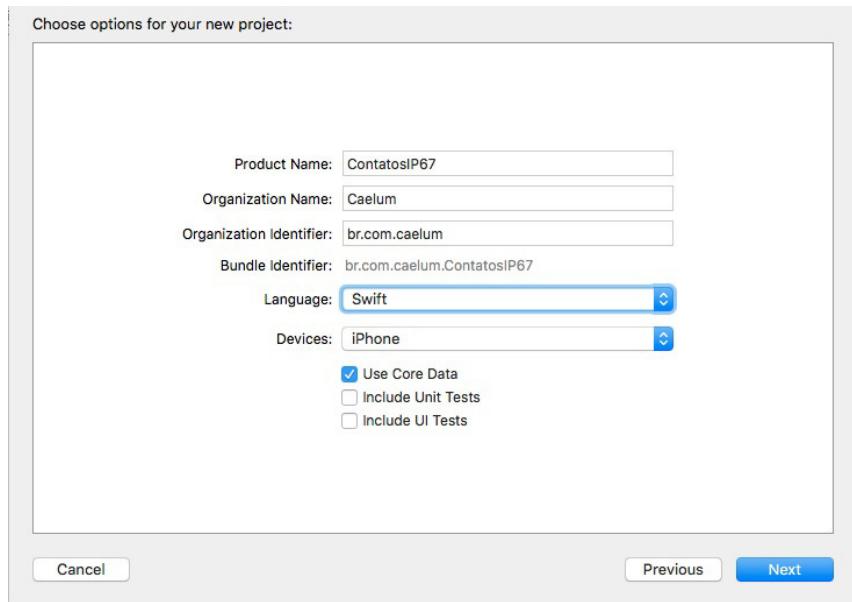
- Execute o Xcode. Há um ícone para esse aplicativo no Dock, ou dentro da pasta *Applications*
- Na janela que se abrir, selecione a opção *Create a new Xcode project*



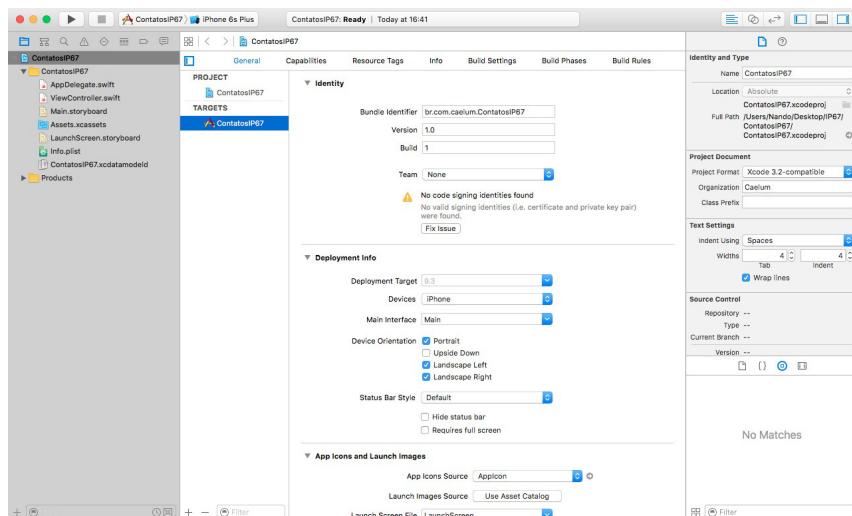
- Selecione o template *Single View Application* e clique em *Next*



- A seguir, será preciso configurar sua aplicação:
 - na opção *Product Name*, digite **ContatosIP67**
 - em *Organization Name*, digite **Caelum**
 - em *Organization Identifier*, digite **br.com.caelum**
 - em *Language*, selecione a opção **Swift**
 - em *Devices*, selecione a opção **iPhone**
 - marque também o checkbox da opção **Core Data**
 - clique em *Next*



- No próximo diálogo, escolha o diretório para gravar seu projeto e clique em *Create*. A seguir, o Xcode exibirá a tela inicial para seu projeto



Agora, você está pronto para desenvolver sua primeira aplicação.

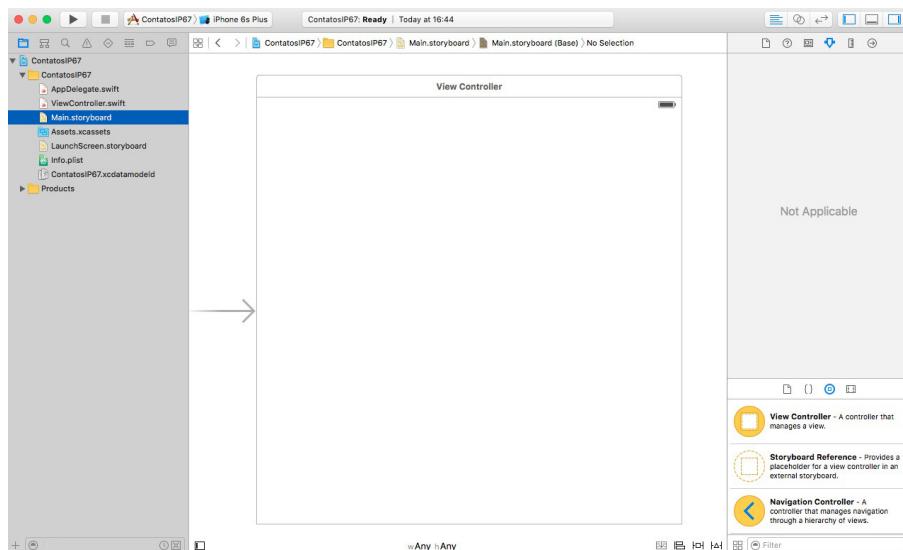
3.4 PRIMEIRA APLICAÇÃO PARA IOS

Vamos colocar as mãos na massa e criar o nosso primeiro aplicativo. No Xcode, selecione (único clique) o arquivo `Main.storyboard` exibido na lateral esquerda. Ele estará dentro do diretório que tem o nome do seu projeto, no nosso caso `ContatosIP67`. Esse arquivo contém uma representação dos componentes gráficos que utilizaremos. Uma forma bastante simples de editar essa representação é utilizando o **Interface Builder**, que é uma ferramenta integrada ao Xcode que nos permite arrastar

componentes visuais para formar uma interface.

Essa é uma das principais características do Xcode, ele nos permite grande produtividade para a criação de aplicações. Tudo que "desenharmos" no **Interface Builder** será armazenado no arquivo **Main.storyboard**.

A exibição inicial do **Interface Builder** para a edição do arquivo **Main.storyboard** deverá ser assim:



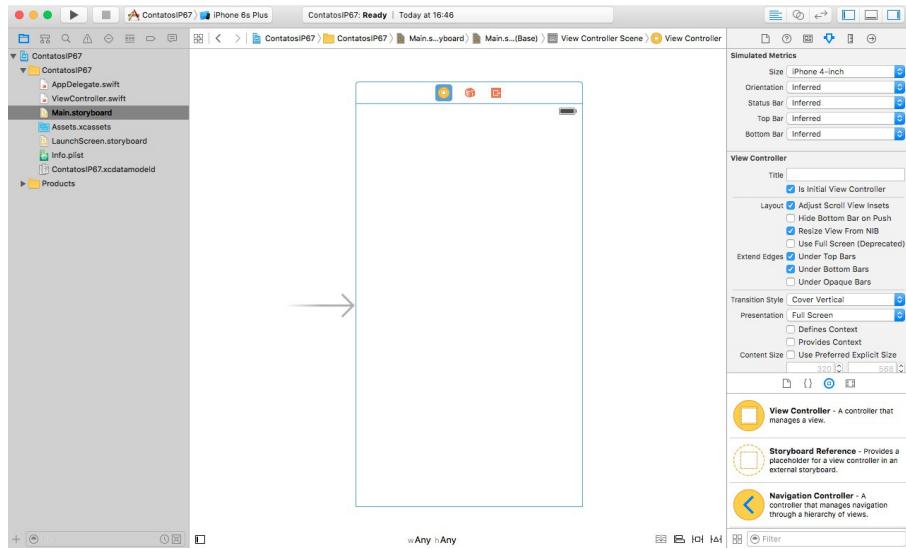
CRIANDO COMPONENTES VISUAIS FORA DO INTERFACE BUILDER

O **Interface Builder** não é a única forma de criar um componente visual. Podemos fazer isso usando apenas código, o que pode ser necessário algumas vezes, como veremos mais adiante. Porém o **Interface Builder** é uma ferramenta que nos dá muita produtividade e não há motivo para evitá-lo.

A tela inicial que vemos no **Interface Builder** é a representação da aparência do aplicativo no iPhone. Portanto, é nela que posicionaremos os componentes visuais que darão vida ao nosso primeiro aplicativo. Ao lado direito do Xcode existe uma lista com os componentes visuais nativos do iOS. Perceba que, além do nome e descrição de cada um desses componentes, também é exibida uma pequena imagem, uma espécie de preview. Para utilizar um deles basta clicar e arrastá-lo para o **Interface Builder**, soltando-o sobre a tela do aparelho.

Antes de iniciarmos, vamos alterar a visualização do tipo de iPhone que queremos utilizar no **Storyboard**. Para isso, selecionamos a tela e vamos até o menu do lado esquerdo onde clicamos em **Attributes Inspector** (quarto ícone da esquerda para a direita). Em **Simulated Metrics** vamos

selecionar o tamanho iPhone 4-inch. A tela deve ficar da seguinte forma:

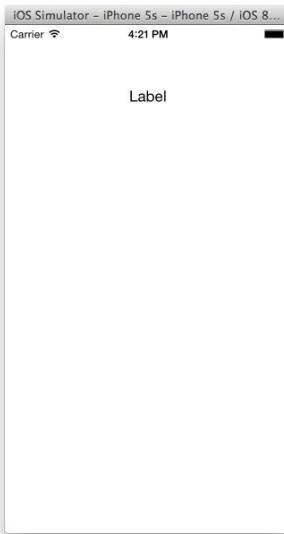


Para iniciar, vamos utilizar um *Label*, que é um componente que faz um texto qualquer ser exibido na tela. Clique sobre o componente na lista do lado esquerdo e arraste-o. Durante o processo, um ícone com o símbolo + vai aparecer abaixo do nome do componente. Quando o cursor estiver sobre a tela, solte o componente. Você pode reposicioná-lo a vontade. Repare que, enquanto você posiciona o *Label* na tela, algumas linhas azuis com uma espécie de efeito magnético aparecem. Essa é a forma do Xcode lhe dizer a melhor posição para o seu componente segundo as diretrizes da própria Apple para a elaboração de interfaces.

GUIDELINES PARA A CRIAÇÃO DE INTERFACE

No capítulo de introdução há mais detalhes sobre as boas práticas de elaboração de interfaces para aplicativos iOS definidas pela própria Apple.

Vamos executar o aplicativo. No Xcode, no canto superior esquerdo, existe um botão *Run*. Ao clicar no botão o Xcode vai compilar o projeto e carregar o **Simulador do iOS**. Essa ferramenta do Xcode nos permitirá realizar alguns testes na aplicação sem que para isso seja necessária a sua instalação em um dispositivo (como o iPhone ou o iPod Touch).



Agora precisamos adicionar alguma funcionalidade em nossa aplicação. Primeiramente, vamos mudar o texto que aparece no *Label*: dê um duplo clique sobre ele na interface e assim entrará em modo edição. Agora vamos digitar um texto. Digite Olá .

Vamos adicionar um campo de texto para permitir a entrada de dados do usuário. Arraste o componente *Text Field* para a tela, como foi feito com o *Label*. Depois de adicionar o componente, podemos posicioná-lo e redimensioná-lo como ficar melhor (basta colocar o cursor sobre a extremidade da caixa de texto, clicar e arrastar).

Rode novamente a aplicação (basta clicar no botão *Run*) para verificar as últimas alterações. Ao clicar no campo de texto, o teclado nativo do iOS aparecerá na tela, para que o usuário possa digitar alguma informação.

TECLADO NATIVO NO SIMULADOR

Pode ser que o teclado do iOS não apareça ao clicar no campo de texto do simulador. Caso isto aconteça, você pode clicar no menu `Hardware` , `Keyboard` , `Toggle Software Keyboard` ou ainda, utilizar o atalho `Command+K` para acessar esta opção.

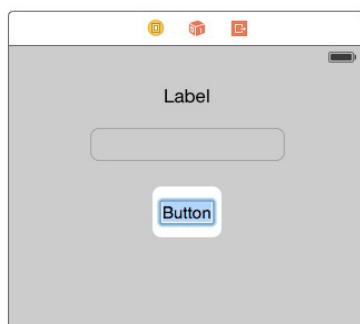


3.5 EXERCÍCIO - O SIMULADOR DO XCODE

Para que você possa executar o simulador do iPhone e se acostumar com a ferramenta usada para testar a construção do aplicativo, crie alguns componentes visuais no **Interface Builder**.

1. Selecione o arquivo `Main.storyboard`. Com o **Interface Builder** e adicione ao layout um *Label*, um *Text Field* e um *Button*. A lista de componentes visuais fica no canto inferior direito, bastando clicar no pequeno cubo.

Altere o texto do botão para algo como "Adicionar". Para isso, após arrastar o botão para o layout, clique duas vezes sobre ele para visualizar o modo de edição do texto.



Ao final, o layout deverá ficar semelhante à imagem abaixo:



2. Rode a aplicação, e o simulador do *iPhone* abrirá carregando a tela que fizemos. O que acontece ao clicarmos no botão?

SHORTCUTS

Podemos usar alguns atalhos para facilitar na hora de desenvolver. Abaixo segue alguns atalhos úteis para o desenvolvimento:

Files:

- ⌘N - Novo Projeto
- ⌘N - Novo arquivo
- ⌘O - Abrir um arquivo
- ⌘W - Fechar janela atual
- ⌘S - Salvar arquivo atual
- ⌘C - Commit
- ⌘X - Baixar atualizações do repositório (pull)

Views

- ⌘1 - Project View
- ⌘↔ - Assistant
- ⌘Y - Debug Area

Product

- ⌘R - Run
- ⌘T - Test
- ⌘B - Build
- ⌘K - Clean
- F7 - Step into
- F6 - Step over

Para visualizar a lista completa de atalhos acesse:

https://developer.apple.com/library/mac/documentation/IDEs/Conceptual/xcode_help-command_shortcuts/MenuCommands/MenuCommands014.html

Ao clicarmos no botão, nada acontece. Precisamos agora colocar comportamentos em nossa aplicação. Para isto, no próximo capítulo vamos ver como a linguagem *Swift* trabalha.

ADICIONANDO COMPORTAMENTOS EM NOSSA APLICAÇÃO

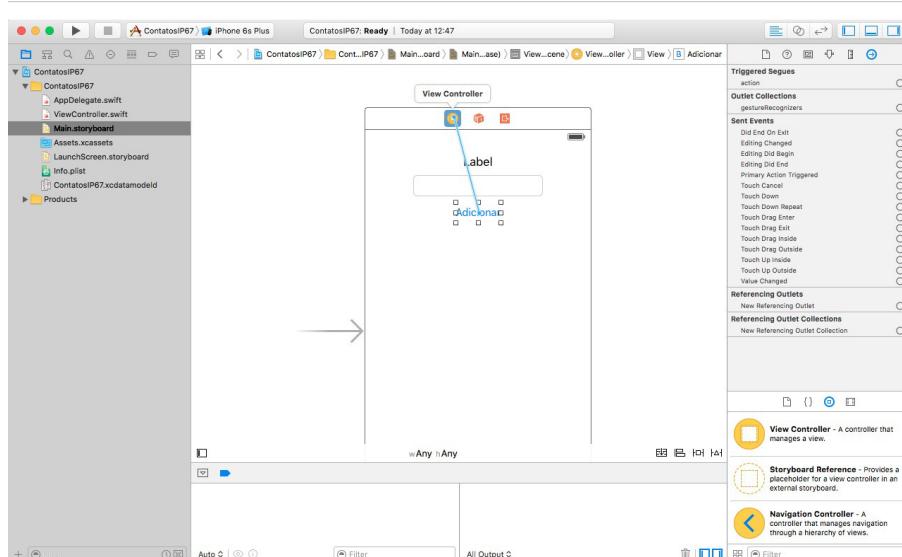
Vamos fazer com que, ao clicarmos no botão que adicionamos anteriormente, uma mensagem seja impressa no console do XCode. Precisamos declarar um método que fará isto. Vamos editar o arquivo `ViewController.swift` e declarar o método que pegará os dados deste nosso formulário:

```
class ViewController: UIViewController {  
    ...  
  
    func pegaDadosDoFormulario(){  
    }  
  
}
```

Criamos o método mas como fazemos para que o botão saiba que ele precisa chamar este método no momento em que for clicado? Podemos fazer com que o botão avise o momento do clique para o método. O próprio Interface Builder consegue fazer isso, só temos que tornar o método visível para ele! Fazemos isto anotando o método com `@IBAction`.

```
class ViewController: UIViewController {  
    ...  
  
    @IBAction func pegaDadosDoFormulario(){  
    }  
}
```

Repare que apareceu uma "bolinha" vazia ao lado do nome do método. Isto quer dizer que este método agora é uma "ação" visível ao Interface Builder. Vamos voltar então ao arquivo `Main.storyboard` e dizer para o botão qual a ação que deve ser executada no clique do mesmo. Fazemos isto pressionando *control* e, sem soltar a tecla, temos que clicar sobre o botão e logo em seguida, mantendo o clique pressionado, devemos arrastar o cursor até o ícone do *View Controller*.



Um diálogo chamado *Sent Events* vai aparecer listando todas as ações declaradas no

`ViewController.swift`. Selecione a ação chamada `pegaDadosDoFormulario`. Dessa forma o botão e a ação que queremos executar já estarão conectados.



Ao rodarmos a aplicação podemos clicar no botão, que agora o método `pegaDadosDoFormulario` será executado. Mas como podemos ter certeza se não aparece nada? Vamos colocar uma mensagem no console para verificarmos se o método realmente foi executado.

3.6 EXERCÍCIO - ADICIONANDO COMPORTAMENTOS, NSLOG E O CONSOLE DO XCODE

Vamos implementar o método e acrescentar uma mensagem para quando o botão for clicado.

1. Adicione no arquivo `ViewController.swift`, o método que deverá ser chamado ao clicarmos no botão utilizando a anotação `@IBAction`:

```
import UIKit

class ViewController: UIViewController {

    ...

    @IBAction func pegaDadosDoFormulario(){
    }
}
```

2. Abra novamente o arquivo `Main.storyboard` dentro do **Interface Builder** e faça a conexão entre os componentes visuais e o nosso código. Você deve conectar a ação do botão ao método que declaramos e anotamos com `@IBAction` dentro do `ViewController.swift`
3. No projeto, certifique-se de que esteja visualizando o console do Xcode.

- Selecione as três opções de visualização no canto superior direito do Xcode:



Figura 3.18: Xcode perspective

- Certifique-se também de que o console esteja visível em sua tela:



Figura 3.19: Console

- Precisamos implementar as funcionalidades da nossa aplicação. Adicione uma chamada à função `print` ao método `pegaDadosDoFormulario` para exibir a mensagem de que o botão foi clicado.

Feitas as modificações, seu arquivo deverá ter a seguinte aparência:

```
import UIKit

class ViewController: UIViewController {

    ...

    @IBAction func pegaDadosDoFormulario(){
        print("Botão foi clicado.")
    }

}
```

- Rode a aplicação clicando no botão *Run* e faça os testes para validar se os comportamentos definidos estão funcionando.

- Use os controles no canto superior esquerdo do Xcode



Figura 3.20: Executando a aplicação

- O resultado final será semelhante ao seguinte:

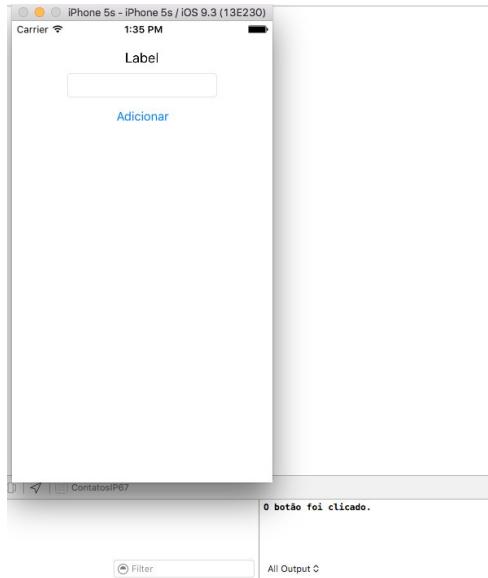


Figura 3.21: Resultado no console

APLICAÇÃO DE CONTATOS

4.1 MOTIVAÇÃO: APLICAÇÃO INSPIRADA PELO APPLICATIVO NATIVO CONTACTS.

Vamos criar uma lista de contatos inspirada no aplicativo nativo do iPhone chamado Contacts. Durante o processo, vamos conhecer outras funcionalidades que os dispositivos baseados em **iOS** nos oferecem de maneira fácil.

4.2 OBTENDO DADOS DO USUÁRIO USANDO UM FORMULÁRIO

Uma das principais funcionalidades da nossa aplicação será a possibilidade de cadastrar novos contatos. Criaremos um formulário para que o usuário da aplicação possa realizar esses cadastros.

Vamos querer guardar vários dados para um novo contato, então devemos deixar nossa tela pronta para receber estas informações. Para isto, alteraremos o arquivo `Main.storyboard` acrescentando *Labels* e *Text Fields* para cada uma das informações que queremos armazenar. Como nossa tela representará o formulário para a entrada de dados de um contato, vamos também alterar o nome de nosso `ViewController` para `FormularioContatoViewController`. Para isso devemos efetuar os 3 passos abaixo:

1. Renomear o arquivo `.swift`

Selecione o arquivo `ViewController.swift`, com o arquivo selecionado clique sobre o nome do arquivo novamente para entrar em modo de edição.

Feito isso altere o nome do arquivo para `FormularioContatoViewController.swift`:

```

// ViewController.swift
// ContatosIP67
//
// Created by Fernando on 30/06/16.
// Copyright © 2016 Caelum. All rights reserved.
//

import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

    @IBAction func pegaDadosDoFormulario(){
        print("O botão foi clicado.")
    }

}

```

2. Renomear a classe

Ainda com o arquivo selecionado vamos alterar o nome da nossa classe.

Altere a declaração `class ViewController: UIViewController ...` para

```
class FormularioContatoViewController: UIViewController ...
```

```

// ViewController.swift
// ContatosIP67
//
// Created by Fernando on 30/06/16.
// Copyright © 2016 Caelum. All rights reserved.
//

import UIKit

class FormularioContatoViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

    @IBAction func pegaDadosDoFormulario(){
        print("O botão foi clicado.")
    }

}

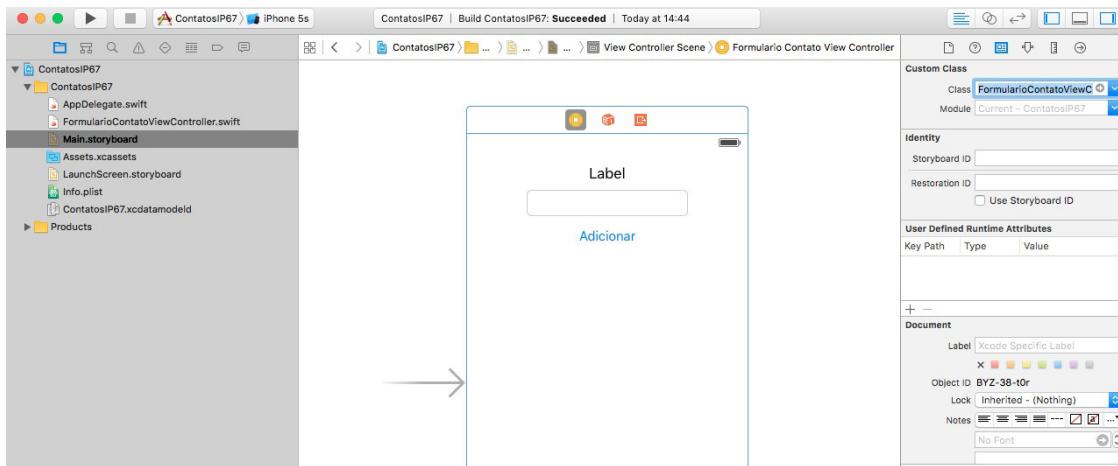
```

3. Renomear classe no `Main.storyboard`

Mesmo após renomenando o arquivo e o nome da classe nosso formulário continua associado à classe `ViewController`.

Selecione o arquivo `Main.storyboard` e nele selecione o formulario, feito isso vamos clicar no

icone `ViewController` (o mesmo que utilizamos para associar nosso método `pegaDadosDoFormulario`). E então vamos selecionar a aba *identity inspector* e então alterar o campo `class` dentro de *Custom Class* para `FormularioContatoViewController`



4.3 COMPLETANDO A TELA DO FORMULARIO

Cada contato cadastrado poderá ter as seguintes informações: **nome**, **telefone**, **endereço** e **site**. Para obter todos esses dados será preciso criar diversos componentes do tipo `UITextField`. E, para que o usuário saiba qual campo é referente a que informação, também criaremos vários `UILabel` para identificar os campos de texto.

Quando estiver criando os elementos visuais que irão compor a tela de formulário, lembre-se de obedecer as regras do **HIG (Human Interface Guidelines)**: os critérios de aprovação de um aplicativo na *AppStore* são rigorosos e, se você não as seguir, o aplicativo não será publicado.

Veja como ficará a tela com o formulário já criado:



wAny hAny

4.4 EXERCÍCIO - CRIANDO A TELA DO FORMULÁRIO

1. Renomeie o arquivo, a classe e o storyboard de `ViewController` para `FormularioContatoViewController`.
2. No arquivo `Main.storyboard` crie os campos e as labels para o usuário digitar o **nome**, **telefone**, **endereço** e **site** de um contato.

Veja o resultado final esperado após a configuração dessa tela:



wAny hAny

Figura 4.5: Layout para o formulário

4.5 CRIANDO UM CONTATO A PARTIR DOS DADOS DO FORMULÁRIO

Excelente, temos o layout para o usuário passar as informações de um contato, mas ao clicarmos no botão, ainda temos a mensagem de que o botão foi clicado. Queremos agora é que apareçam os dados digitados para depois armazenar estes dados de alguma forma mais estruturada.

Quando o usuário clicar nesse botão, vamos capturar as informações digitadas e mostrar no log do console, para que seja possível descobrir qual foi o valor digitado para o nome, telefone, e assim por diante.

Da mesma forma que colocamos a anotação `@IBAction` para especificar ao *Interface Builder* que o método estava disponível para ser vinculado ao botão, devemos especificar agora que os valores que queremos guardar também estão disponíveis para serem utilizados em nosso código. Fazemos isto utilizando a anotação `@IBOutlet` ao declararmos as propriedades no arquivo `FormularioContatoViewController.swift`. A anotação `@IBOutlet` fará com que os objetos criados na tela fiquem disponíveis para manipulação no nosso *view controller*. Vamos declarar os `IBOutlet`s e

fazer as devidas ligações.

```
import UIKit

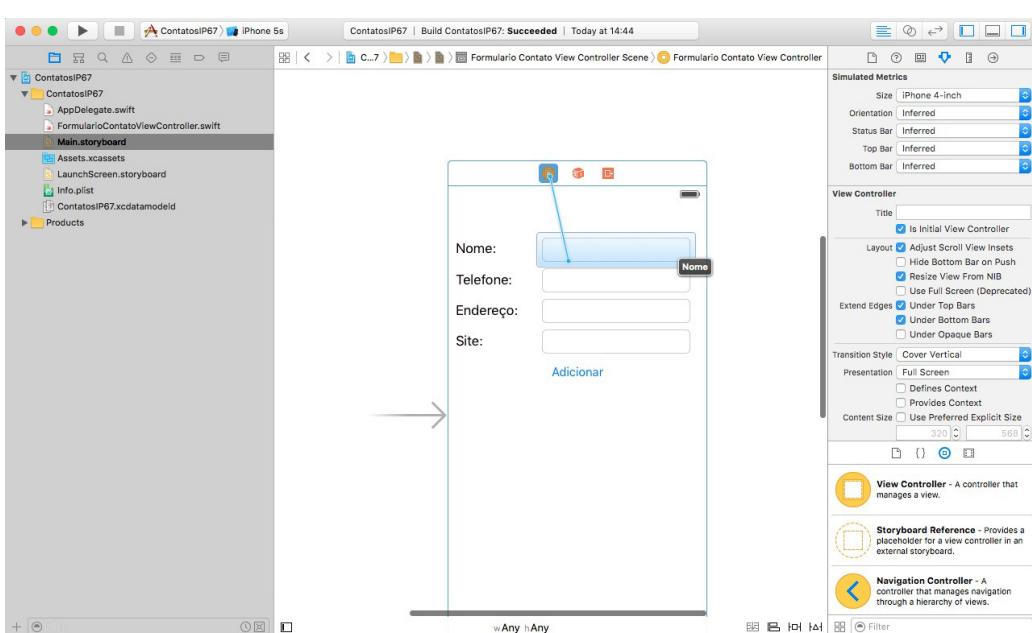
class FormularioContatoViewController: UIViewController {

    @IBOutlet var nome: UITextField!
    @IBOutlet var telefone: UITextField!
    @IBOutlet var endereco: UITextField!
    @IBOutlet var siteText: UITextField!

    @IBAction func pegaDadosDoFormulario(){
        print("O botão foi clicado.")
    }

}
```

Será preciso associar os `UITextField`s da interface com o código da classe. Vamos novamente editar o arquivo `Main.storyboard`. Desta vez é o código que quer pegar as informações dos elementos então, com a tecla **CTRL** pressionada, vamos clicar sobre o símbolo amarelo que representa o `_view controller` e, sem soltar a tecla, vamos arrastar o mouse até um dos `UITextFields` na interface. Será exibido um menu listando os `IBOutlets` disponíveis para ligação. Selecione o correspondente ao campo de texto desejado e repita o procedimento para todos os campos.



Com as ligações feitas, podemos imprimir os dados digitados pelo usuário no método `pegaDadosDoFormulario`. Para isto, precisamos pegar o texto de dentro do `UITextField` correspondente. Isto pode ser efetuado invocando o método `text` para o campo desejado. No arquivo `FormularioContatoViewController.swift` fazemos:

```
import UIKit

class FormularioContatoViewController: UIViewController {
```

```

@IBOutlet var nome: UITextField!

...
@IBAction func pegaDadosDoFormulario(){
    let nome = self.nome.text!
}
}

```

Repare a palavra `self` antes do nome da propriedade. Sempre que quisermos pegar uma propriedade de nosso próprio objeto, ou ainda invocar um método para o mesmo, utilizamos o `self` para informar ao compilador de quem estamos buscando aquela informação. Devemos fazer isto para os demais campos:

```

import UIKit

class FormularioContatoViewController: UIViewController {

    @IBOutlet var nome: UITextField!
    @IBOutlet var telefone: UITextField!
    @IBOutlet var endereco: UITextField!
    @IBOutlet var site: UITextField!
    ...

    @IBAction func pegaDadosDoFormulario(){
        let nome = self.nome.text!
        let telefone = self.telefone.text!
        let endereco = self.endereco.text!
        let site = self.site.text!

        print("Nome: \(nome), Telefone: \(telefone), Endereço: \(endereco), Site: \(site)")
    }
}

```

4.6 EXERCÍCIO - DADOS DO FORMULÁRIO

1. No arquivo `FormularioContatoViewController.h`, adicione a declaração das propriedades.

```

@IBOutlet var nome: UITextField!
@IBOutlet var telefone: UITextField!
@IBOutlet var endereco: UITextField!
@IBOutlet var site: UITextField!

```

2. Para fazer a conexão entre os elementos da tela e o código que declaramos edite o arquivo `Main.storyboard` no **Interface Builder**, pressione a tecla **CTRL** e clique sobre o ícone amarelo logo acima da tela que representa o *view controller*. Arraste o cursor do mouse até um `UITextField` na interface.
 - Solte o mouse sobre o *text field* desejado.

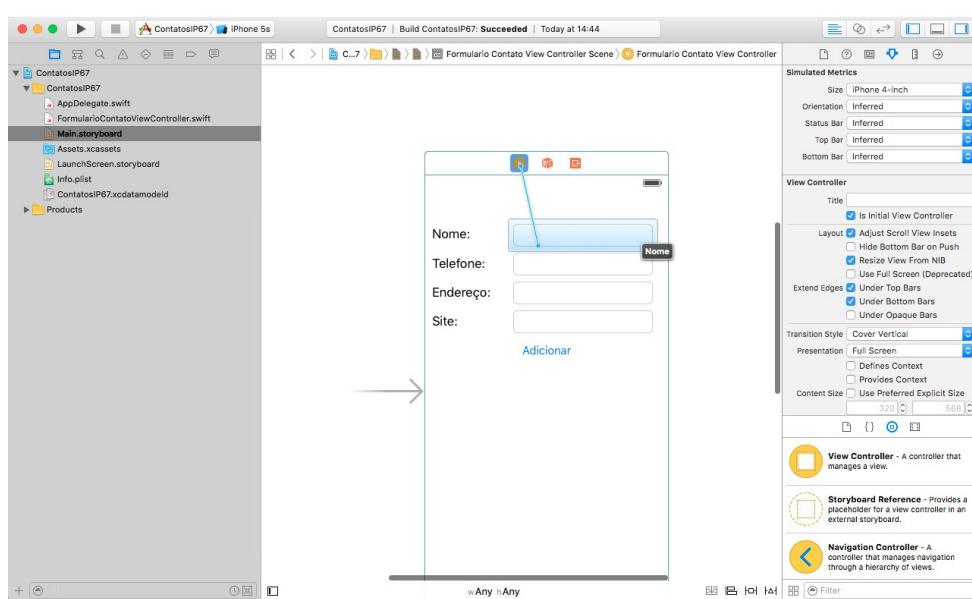


Figura 4.7: Ligando um Outlet com seu respectivo UITextField

- No menu, escolha o `Outlet` correspondente.



Figura 4.8: Selecionando o Outlet referente ao UITextField

Repeta esse processo para todos os campos de texto do formulário.

3. No arquivo `FormularioContatoViewController.swift`, vamos alterar o método `pegaDadosDoFormulario` para obter os dados dos campos e imprimir no console:

```
import UIKit

class FormularioContatoViewController: UIViewController {

    @IBOutlet var nome: UITextField!
    @IBOutlet var telefone: UITextField!
    @IBOutlet var endereco: UITextField!
    @IBOutlet var site: UITextField!

    ...

    @IBAction func pegaDadosDoFormulario(){

```

```

let nome = self.nome.text!
let telefone = self.telefone.text!
let endereco = self.endereco.text!
let site = self.site.text!

print("Nome: \(nome), Telefone: \(telefone), Endereço: \(endereco), Site: \(site)")

}

```

Rode a aplicação, preencha o formulário e veja se, ao clicar no botão, os dados do contato serão exibidos no *console* conforme na imagem abaixo:

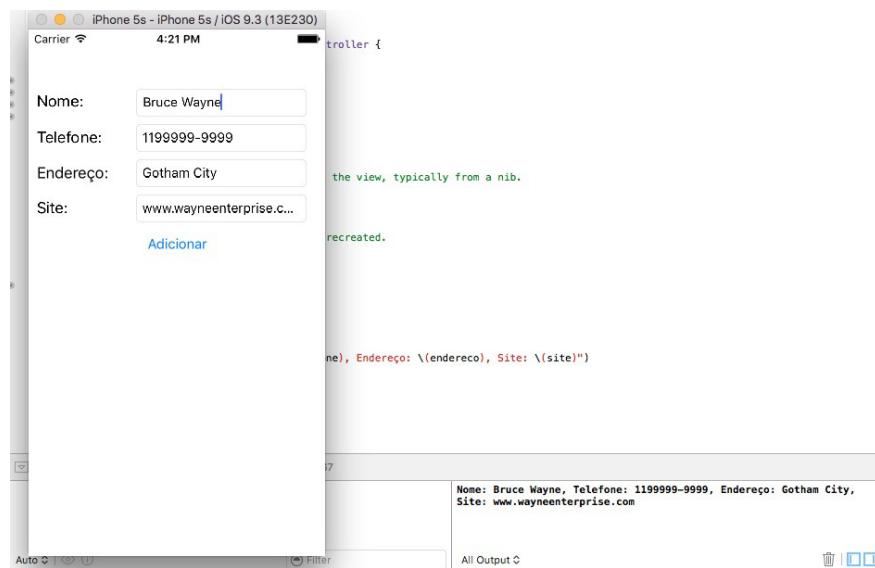


Figura 4.9: Mostrando os dados

CRIANDO CÓDIGO DE FÁCIL MANUTENÇÃO: SUAS PRÓPRIAS CLASSES

Nesse momento, já é possível capturar os dados de um contato digitado pelo usuário no formulário de cadastro. Poderíamos usar essa informação para criar uma lista de contatos, mas onde poderíamos armazenar estes dados?

Além disso, imagine que, na listagem de contatos, vamos exibir o nome do contato seguido pelo telefone envolvido pelos caracteres "[" e "]". O contato **João da Silva** com o telefone **11 12345-6789** tem que ser exibido como **João da Silva [11 12345-6789]**. Não há nenhum problema, certo? Basta que, no código que vai listar os contatos em algum lugar, façamos a interpolação do nome com o telefone da forma desejada:

```
let nomeComTelefone: String = "\$(nome) [\$(telefone)]"
```

Veja que se essa mesma forma de apresentar o nome seguido do telefone se repetir em outros lugares do sistema, além da listagem, teremos que replicar esse código em outras partes. Caso haja alguma mudança na formatação dessa informação (nome e telefone), teremos que alterar o mesmo código em vários lugares diferentes, causando problemas. Imagine que, por algum motivo, seja preciso mudar os caracteres "[" e "]" que envolvem o telefone para os sinais "<" e ">" como em: **João da Silva <11 12345-6789=>**.

Como o código responsável pela concatenação do nome e do telefone está espalhado em vários pontos da aplicação, teríamos que lembrar de todos os lugares onde isso ocorre e realizar a alteração. Nada impossível, porém nada prático também. E se pudéssemos obter uma *String* com a representação de nome e telefone que desejamos por meio de uma chamada de método? Algo como no exemplo a seguir:

```
let nomeComTelefone: String = contato.nomeComTelefone()
```

Poderíamos usar esse método `nomeComTelefone` sempre que fosse preciso exibir os dados concatenados de uma forma específica. Sabe o que mais ganhamos com isso além de diminuir linhas em nosso código? Imagine o mesmo caso onde é preciso alterar a lógica de concatenação dessas informações, só seria preciso uma única alteração. Se toda a aplicação usasse sempre o método `nomeComTelefone` quando precisasse desses dados, então bastaria alterar esse único método e a aplicação já estaria usando a forma correta de exibir os dados.

Podemos isolar toda a lógica que representa um contato em uma classe que irá descrever o que é realmente um contato. Podemos, inclusive, definir métodos que permitam alterar dados de um contato, ou recuperar suas informações após executar alguma lógica, como seria o caso do nome concatenado com o telefone. Está pronto para criar uma classe em Objective-C? Então vamos lá!

5.1 CLASSES E OBJETOS: COMPREENDENDO MELHOR OS ARQUIVOS .H E .M

Vamos criar uma nova classe para descrever um `Contato`. Para isso, será preciso criar um arquivo `.h` e um arquivo `.m`. O Xcode automatiza a geração dos dois arquivos com um mesmo nome, vamos chamá-los de `Contato`.

Use o atalho `Command + n` e selecione `Cocoa Touch Class` para ativar o assistente de criação de arquivos e crie a nova classe `Contato`. Siga as instruções do assistente e lembre-se de configurar a classe `Contato` como `Subclass` de `NSObject` e selecionar a linguagem `Objective-C`.

5.2 ARQUIVO BRIDGING-HEADER

Como nosso aplicativo está sendo feito em `Swift` e criamos uma classe em `Objective-C` precisamos de uma forma de integrar as duas linguagens, para que uma possa conversar com a outra.

Quando temos uma aplicação em `Swift` e criamos uma classe em `Objective-C` é sugerido a criação de um arquivo `Bridging Header`.

Esse arquivo nada mais é do que um arquivo de cabeçalho com um padrão de nomenclatura `NomeDoApp-Bridging-Header.h`. Nele iremos importar os arquivos de declaração das classes (".`h`") em `Objective-C` que queremos utilizar nas classes em `Swift`.

O arquivo de declarações: `Contato.h`

Vamos entender melhor o que vai em cada um desses arquivos. Primeiramente, vamos compreender melhor o arquivo `Contato.h`. O código gerado pelo `Xcode` é mínimo, vamos verificar linha a linha o que ele faz:

```
#import <Foundation/Foundation.h>
```

Essa instrução torna disponível, para os objetos do tipo `Contato`, as funcionalidades básicas do `iOS`, assim como alguns tipos como o `NSArray`, o `NSString` entre outras coisas.

```
@interface Contato : NSObject
```

O nome da classe vem logo após a diretiva `@interface`, depois disso é possível utilizar os `:` para definir qual será a classe de onde a nova herdará. Nesse caso, foi criada a classe `Contato` que herda de `NSObject`.

E, finalmente, vamos dizer que terminaram as definições necessárias até o momento para a nossa nova classe:

```
@end
```

As definições desse arquivo são suficientes para informar como funciona uma classe porém, essa é só a declaração. Para implementar as funcionalidades dessa classe usaremos o arquivo *.m*.

O arquivo de implementação: Contato.m

Nesse arquivo, criaremos realmente a lógica que representa um contato. Vamos ver qual o código mínimo criado pelo *Xcode*, linha por linha, já que são apenas 3:

Primeiramente, precisamos tornar disponível a declaração da classe `Contato` para que seja possível realizar a implementação:

```
#import "Contato.h"
```

As próximas duas linhas delimitam o trecho onde será escrito o código de implementação. Qualquer novo código relativo à classe `Contato` será inserido entre a instrução `@implementation Contato` e a diretiva `@end` :

```
@implementation Contato
```

```
@end
```

5.3 EXERCÍCIO - CRIAÇÃO DA CLASSE CONTATO

1. No *Xcode*, use a opção *File -> New -> File*, ou então utilize o atalho *Command + n*. Será exibida a primeira tela do *assistente* para a criação de arquivos. Vamos criar uma nova classe *Objective-C*, portanto escolha a opção *Cocoa Touch Class* e clique em *Next*.

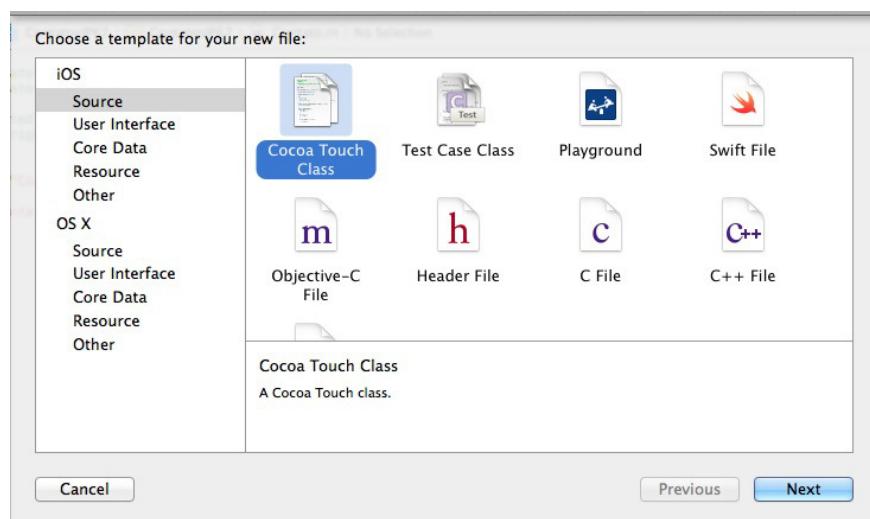


Figura 5.1: Nova classe Objective-C

VERSÕES DO XCODE E A DIFERENÇA DE TEMPLATES DISPONÍVEIS

É possível que a versão do Xcode utilizada por você seja diferente e, portanto, a tela do assistente exibida na figura acima seja também um pouco diferente. Isso acontece porque vez por outra essas telas e menus são alteradas quando é feita uma atualização do Xcode.

Não se preocupe, basta encontrar nas opções disponíveis a que permite criar uma nova classe Objective-C, e seguir em frente!

2. Na próxima tela serão definidos o nome da nova classe e também qual será a classe da qual a nova herdará. Defina `Contato` como valor para a opção `Class`, esse será o nome da nova classe. Escolha (ou digite) `NSObject` para a opção `Subclass of`, é importante herdar dessa classe para que os métodos que permitem criar novos objetos (entre outros) estejam disponíveis na classe `Contato`. Após realizar as configurações, clique em `Next`.

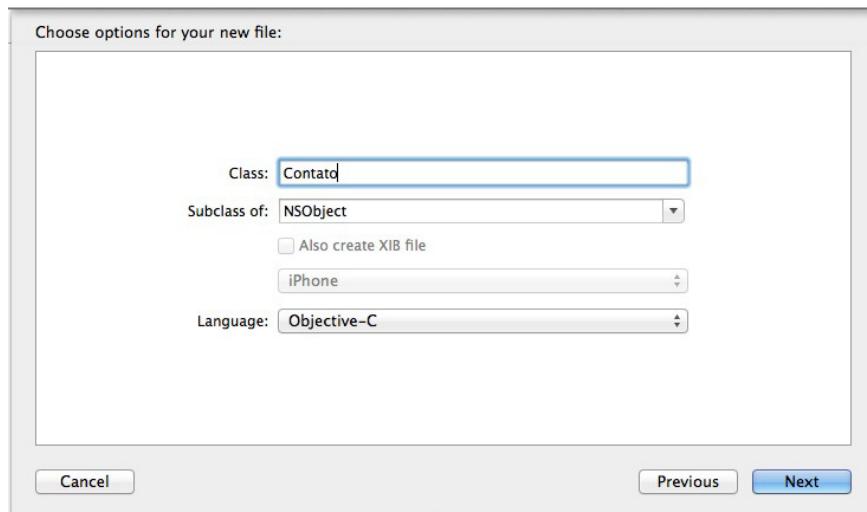


Figura 5.2: Herdando de NSObject

3. Será exibido uma mensagem perguntando se você quer criar um arquivo de Bridging-Header. Clique em `Create Bridging Header`.



Figura 5.3: Criando o arquivo bridging header

CRIANDO BRIDGING-HEADER MANUALMENTE

É possível criar o arquivo de bridging header manualmente, para isso basta seguir os seguintes passos no xcode: selecione o menu *File* -> *New* -> *File* feito isso basta selecionar *Header File* e no nome do arquivo aplicar o padrão de nomenclatura `NomeDoApp-Bridging-Header.h` no nosso caso `ContatosIP67-Bridging-Header.h`.

4. Agora, você pode definir o local em seu sistema de arquivos e em seu projeto onde quer gravar os arquivos relativos à classe `Contato`. Você pode selecionar as opções sugeridas pelo próprio *Xcode*, basta clicar no botão *Create*.

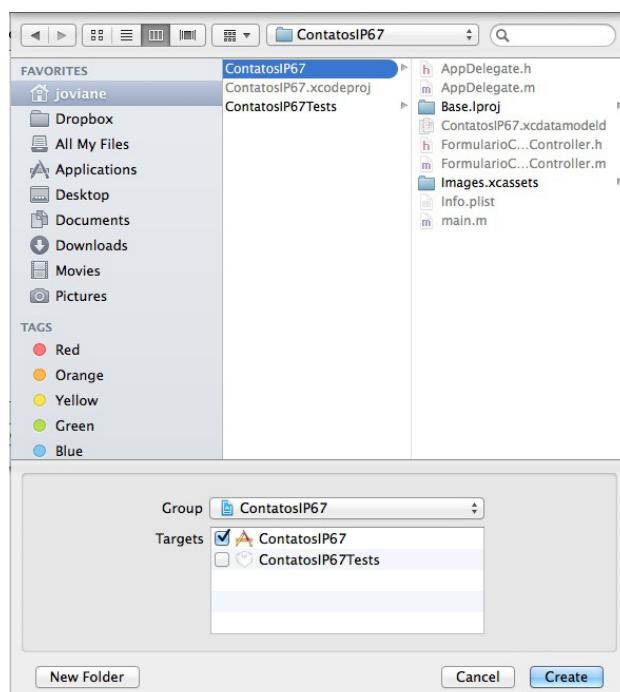


Figura 5.4: Gravando os arquivos

Ao final, você verá dois novos arquivos no *Project navigator* que são: `Contato.h` e `Contato.m`.

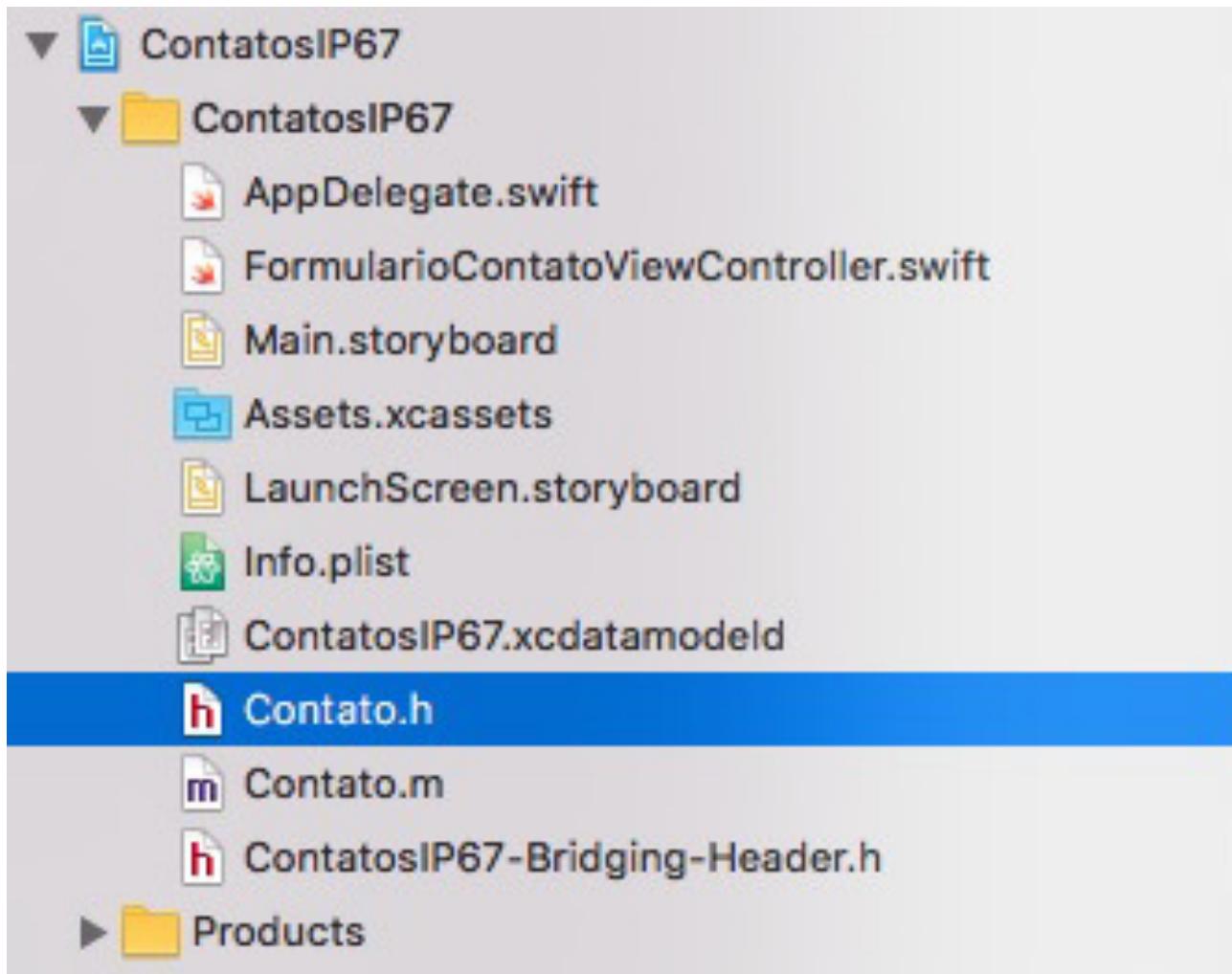


Figura 5.5: Gravando os arquivos

Agora que já sabemos a função dos arquivos com extensão `.h` e `.m`, vamos entender melhor qual tipo de instrução podemos usar em cada um desses arquivos.

5.4 INSTANCIANDO OBJETOS

Em Objective-C definimos o tipo das variáveis no momento em que elas são declaradas no código. Quando criamos uma variável que fará referência para uma instância de algum tipo específico, é preciso adicionar o caractere `*` antes do nome da variável. Para declarar uma variável que pode conter uma referência para um objeto do tipo `Contato`, poderíamos fazer o seguinte:

```
Contato *contato;
```

Para instanciar objetos a partir de uma classe, precisamos mandar uma mensagem (invocar um método) na própria classe para que seja realizada a alocação de memória para o novo objeto que será criado. Isso é feito por meio de uma chamada ao método `alloc` presente em toda classe. O `alloc`

retorna uma referência a um objeto, mas ele ainda não está pronto para uso. Todo objeto responde ao método `init`, que tem a lógica necessária para inicialização tornando-o pronto para uso. Em uma analogia simples, o `alloc` faz o trabalho básico do `new` no Java e C#, e o `init` possui o código do construtor. A seguir, um exemplo clássico de como podemos construir uma instância a partir da classe `Contato`:

```
Contato *novoContato = [[Contato alloc] init];
```

Se quisermos apenas instanciar um objeto sem configurar nada no construtor, podemos simplesmente utilizar o método `new` que internamente chama os métodos `alloc` e `init`:

```
Contato *novoContato = [Contato new];
```

Em Objective-C é comum encontrarmos classes que disponibilizam vários métodos para a criação de um objeto. O uso do prefixo `init` no nome do método é uma convenção para indicar que a responsabilidade de um método é criar um objeto já configurado. A classe `NSString` tem alguns exemplos disso, um dos mais utilizados é o método `initWithFormat`: que recebe uma string e um número variável de parâmetros, exatamente como faríamos para utilizar a função `NSLog`.

5.5 PROPRIEDADES E O GERENCIAMENTO DE MEMÓRIA

Decidimos criar uma classe `Contato` para armazenar os dados de um contato criado a partir do formulário e para possibilitar a criação de comportamentos relacionados a ele. Sabemos também como fazemos para criar um objeto a partir da classe `Contato` porém, para usarmos um `Contato`, precisamos estabelecer quais informações podem ser guardadas em um objeto deste tipo. Imagine que queremos colocar um nome no contato. Queremos fazer algo como:

```
Contato *novoContato = [Contato new];
[novoContato setNome: @"Bruce Wayne"];
NSLog(@"%@", [novoContato nome]); // O nome é Bruce Wayne
```

Note que o exemplo acima sugere utilizar o método `setNome` de um contato para atribuir um valor e o método `nome` para recuperar o valor armazenado. Isso é possível, mas para isso precisamos declarar e implementar os métodos no arquivo `Contato.m`. Precisamos implementar um método na classe `Contato` que permita a atribuição de um valor associado ao `nome`. E, da mesma forma, é preciso implementar um método que permita recuperar esse valor. Vamos implementar esses métodos no arquivo `Contato.m`, começando pelo método que vai permitir a atribuição.

Para que o *Objective-C* entenda que esse método será responsável pela atribuição da propriedade `nome`, precisamos usar uma convenção de nomenclatura. O nome desse método precisa ser `setNome:`. Esse nome recebe um parâmetro do mesmo tipo da propriedade declarada, nesse caso, um parâmetro do tipo `NSString`. O que fazer com esse parâmetro? Precisamos armazená-lo de alguma forma! Podemos guardá-lo em uma variável chamada... `nome`!

Primeiro declaramos a variável `nome` no arquivo `Contato.m`:

```

#import "Contato.h"

@implementation Contato

NSString *nome;

@end

```

Agora declaramos na classe o método que buscará o nome do contato, por convenção, ele terá o mesmo nome de nossa variável:

```

-(NSString *) nome{
    return nome;
}

```

E por último o método que colocará um novo nome no contato:

```

-(void) setNome:(NSString *) novoNome{
    nome = novoNome;
}

```

A classe ficará da seguinte forma:

```

#import "Contato.h"

@implementation Contato

NSString *nome;

-(NSString *) nome{
    return nome;
}

-(void) setNome:(NSString *) novoNome{
    nome = novoNome;
}

@end

```

Podemos agora atribuir e buscar o nome do jeito que queríamos anteriormente:

```

Contato *novoContato = [Contato new];
[novoContato setNome: @"Bruce Wayne"];
NSLog(@"O nome é: %@", [novoContato nome]); // O nome é Bruce Wayne

```

Perfeito! Agora bastaria repetir o processo para todas as outras propriedades mas como seguimos uma convenção ao declarar os nomes dos métodos, será possível usar essas propriedades da forma: `contato.nome = @"Bruce Wayne"`. Quando essa instrução for interpretada, irá gerar uma chamada ao método `setNome:` passando como parâmetro o objeto posicionado logo após o sinal de atribuição. O mesmo vale para o método de acesso. Ao invocarmos `contato.nome` geramos, na verdade, uma chamada a `[contato nome]`. Esta forma de acessar é chamada de `dot notation`:

```

// acesso a propriedade usando o "dot notation":
contato.nome = @"Bruce Wayne";
NSLog(@"O nome é: %@", novoContato.nome);

// é equivalente a uma chamada aos métodos:
[contato setNome:@"Bruce Wayne"];

```

```
NSLog(@"%@", [novoContato nome]);
```

Porém todas as demais propriedades que queremos guardar, teríamos que efetuar o mesmo processo: criar a variável, o método que busca o valor da variável e o que coloca o valor na variável. Isto é um processo trabalhoso, e felizmente temos uma forma de fazer isto automaticamente.

Em *Objective-C* existe um conceito chamado **propriedade**. Podemos pensar nas propriedades como uma forma de armazenar dados relativos a um determinado objeto. No caso de um contato queremos armazenar: um objeto do tipo `NSString` para representar o *nome*, outro para o *site* e assim por diante.

Para declarar uma propriedade, utilizamos a diretiva `@property` seguida no tipo da propriedade que será declarada e do nome que será usado para se referir a ela. Se quisermos que outros objetos, além do próprio contato, tenham acesso às propriedades relativas a ele, temos que declarar essas propriedades no arquivo `.h`. A declaração das propriedades de um contato será feita no arquivo `Contato.h` da seguinte forma:

```
#import <Foundation/Foundation.h>

@interface Contato : NSObject

@property NSString *nome;
@property NSString *telefone;
@property NSString *endereco;
@property NSString *site;

@end
```

Todas as propriedades estão declaradas! A diretiva `@property` já declara tudo que precisamos! Para a propriedade `nome` por exemplo, temos o método `nome` para buscar o valor, o método `setNome:` para colocar o valor e a variável privada `_nome`!

AS VERSÕES MAIS NOVAS DO XCODE E A DIRETIVA @SYNTHESIZE

Nas versões mais antigas do XCode, era necessário utilizar a diretiva `@synthesize` para implementar os métodos de acesso e atribuição para a propriedade e, também, a declaração da variável privada que armazenará o valor. Essa diretiva era utilizada no arquivo `.m`, portanto era necessário o arquivo `Contato.m` usando a diretiva `@synthesize` para implementar a propriedade `nome`.

Nas versões mais novas do Xcode, o compilador incluído gera automaticamente as diretivas `@synthesize`, seguindo o padrão recomendado pela Apple de iniciar o nome da variável de instância (*iVar*) usando um caractere underscore ("_"). A declaração equivalente seria:

```
@synthesize nome = _nome;
```

O uso do underscore é comum, pois ajuda a evitar o uso acidental da variável de instância, no lugar do getter/setter correspondente. Por exemplo, ao escrevermos o seguinte código em uma mensagem da classe Contato:

```
NSLog(@"Nome do Contato: %@", nome);
```

O compilador daria um erro, pois `nome`, nesse contexto, não existe. Devemos, então, explicitar que queremos acessar diretamente a variável de instância (usando `_nome`), ou seguir a prática recomendada de sempre acessar uma propriedade através de seu getter ou setter (usando `self.nome`).

Gerenciamento de memória e o trabalho do ARC

No iOS o gerenciamento da memória utilizada por um programa é realizado usando uma técnica conhecida como *contador de referências*. Com o iOS 5 foi lançado um recurso chamado *Automatic Reference Counting* ou simplesmente *ARC*. O *ARC* poupa o trabalho de enviar chamadas de método para um objeto relativos ao gerenciamento de memória, deixando o programador mais livre para se preocupar com a lógica do negócio que ele está construindo em uma aplicação.

GERENCIAMENTO DE MEMÓRIA E O OBJECTIVE-C

Muitas linguagens e ambientes de programação fornecem o recurso chamado *Garbage Collector*. O objetivo principal dessa funcionalidade é tirar do programador a responsabilidade por alocar e desalocar a memória utilizada pelo programa durante sua execução.

No *iOS* não há esse recurso e a ideia do ARC é facilitar a vida do desenvolvedor. O ARC trabalha em tempo de compilação inserindo chamadas de métodos necessárias para alocação e liberação de memória para os objetos criados durante a execução.

Para compreender melhor o assunto você pode consultar o seguinte post no blog da Caelum:
<http://blog.caelum.com.br/gerenciamento-de-memoria-e-o-arc-no-objective-c>.

Porém o *ARC* não faz tudo sozinho. Quando usamos a diretiva `@property` para criar propriedades, o que acontece nos bastidores é a geração de código no momento da compilação; para que o código seja gerado de forma adequada, precisamos informar como as propriedades irão se comportar quando forem atribuídas e quando retornarem os objetos referenciados por elas.

Quando declaramos um atributo como `@property NSString *nome;` estamos usando valores *default* para o gerenciamento de memória. Em um `UIViewController`, muitas vezes queremos declarar `IBOutlet`s para manipular elementos da interface. Uma forma de gerenciamento de memória possibilitada pelo *ARC*, que faz sentido para esse tipo de propriedade, é a referência fraca. Para criar uma propriedade que não aumenta o contador de referências para um objeto quando recebe uma atribuição, usamos o atributo `weak`:

```
@property(weak) NSString *nome;
```

Se não formos específicos quanto ao tipo de referência que queremos criar em uma propriedade, o valor *assign* será assumido, o que significa que uma atribuição não irá aumentar o contador de referências, mas se o objeto for liberado da memória é nossa responsabilidade descobrir isso e anular qualquer referência para esse objeto. Tudo isso é gerenciado automaticamente pelo tipo de referência `weak`.

Uma outra opção *pós-ARC* para declaração de uma propriedade é a referência do tipo `strong`. Podemos usar esse tipo de referência quando queremos garantir que o objeto referenciado pela propriedade estará disponível na memória durante todo o tempo de vida do objeto que possui a propriedade.

Acesso concorrente aos objetos: atomic e nonatomic

Também é possível definir se um objeto é seguro ou não para ser acessado por diversas *threads*

distintas ao mesmo tempo. Os elementos visuais em uma aplicação *iOS* são acessíveis exclusivamente pela *thread* principal, que é criada assim que o programa é carregado. Portanto, podemos considerar qualquer instância de `UIView` visível na tela como segura para acesso concorrente.

Por padrão, qualquer propriedade é criada com o atributo `atomic`, o que significa que o objeto não pode ser acessado por várias *threads*. Isso garante que nenhum objeto será modificado inadvertidamente por *threads* concorrentes, mas há um certo custo de performance envolvido, já que sempre que o objeto for acessado, o *iOS* precisa verificar se já não há uma *thread* utilizando o mesmo objeto.

Por isso, é bem comum declarar uma propriedade que se refere a uma instância de `UIView`, normalmente um `IBOutlet` como `nonatomic`. Pode até parecer contra intuitivo, mas como todo objeto do tipo `UIView` exibido na tela será acessível somente pela *thread* principal, não precisamos adicionar mais essa verificação:

```
@property(nonatomic, weak) IBOutlet UITextField *nome;
```

Em *Swift* a mesma declaração pode ser representada assim:

```
@IBOutlet weak var nome: UITextField!
```

UM PASSO DE CADA VEZ

Não se preocupe se toda essa informação for demais! No dia-a-dia você não precisa se preocupar com tudo isso o tempo todo.

Porém, é interessante saber que existem esses detalhes envolvidos com a criação de propriedades para que você tenha ferramentas para entender o que está acontecendo em seu programa quando isso for necessário.

5.6 INSTANCIANDO E USANDO OBJETOS

Podemos completar a classe `Contato` adicionando as propriedades relativas aos dados de um contato. Vamos usar o atributo `strong` na declaração para garantir que todos os dados do contato permaneçam na memória enquanto a instância de `Contato` que se refere a eles existir.

```
@property (strong) NSString *nome;
@property (strong) NSString *telefone;
@property (strong) NSString *endereco;
@property (strong) NSString *site;
```

Ótimo. Agora temos a declaração de um tipo chamado `Contato` que tem todas as características necessárias para armazenar os dados de um contato criado a partir de um formulário. Podemos, finalmente, alterar o código do método `pegarDadosDoFormulario`, responsável por extrair os dados digitados pelo usuário no formulário de cadastro. Atualmente, esse método está usando um formulário

que apenas imprime os dados das variáveis.

Porém, para isso será preciso criar um objeto do tipo `Contato`. O primeiro passo é invocar o método de classe `alloc` na classe `Contato`. Esse método fará operações como alocação de memória para o objeto e retornará uma instância do tipo `Contato`. No entanto, é preciso completar o processo de inicialização do objeto retornado invocando seu método `init`. O código final para a criação de um objeto do tipo `Contato` será o seguinte: `[[Contato alloc] init]`. Podemos também utilizar o método `new` que é um atalho para o `alloc` seguido de `init`.

RECEITA PARA CRIAÇÃO DE OBJETOS

Essa é a receita básica para a criação de novos objetos. Se você programa em Java, C# ou alguma linguagem parecida, pode pensar nesse conjunto de invocações de método como algo semelhante ao operador `new` para a criação de um novo objeto.

Agora, vamos alterar o método `pegaDadosDoFormulario` para usar uma instância de `Contato`. Sempre que utilizamos uma classe em um arquivo onde ela não foi declarada, é preciso importar o arquivo de cabeçalho que define essa classe. No nosso caso, será preciso importar o arquivo `Contato.h` no arquivo `ContatosIP67-Bridging-Header.h` para que possamos utilizar a classe de `Contato` em qualquer classe `Swift`. Para importar um arquivo de declaração, usamos a diretiva `#import`:

```
#import "Contato.h"
```

Depois de importar o cabeçalho da classe, tudo que ele define pode ser utilizado, portanto já podemos alterar o método `pegaDadosDoFormulario` para utilizar uma instância de `Contato`. Vamos alterar também este método para que utilize a *dot notation* ao usarmos as propriedades. Após a alteração, o método ficará assim:

```
@IBAction func pegaDadosDoFormulario(){
    let contato: Contato = Contato()

    contato.nome = self.nome.text!
    contato.telefone = self.telefone.text!
    contato.endereco = self.endereco.text!
    contato.site = self.site.text!
    print(contato)
}
```

Mas se rodarmos o código deste jeito, aparecerá no console algo como:

```
<Contato: 0x7f9338e2c560>
```

Seria interessante que os dados do `Contato` fossem mostrados, assim como acontece com a classe `NSString`. Existe um método que representa a descrição de um objeto e todo objeto possui este método. Este método é chamado de `description` e ele está definido na classe `NSObject`. Se

quisermos dar uma implementação diferente para ele, devemos sobrescrevê-lo em nossa classe. No arquivo `Contato.m` fazemos então:

```
- (NSString *)description {
    return [NSString stringWithFormat:@"Nome: %@", Telefone: %@,
        Endereço: %@", Site: %@", self.nome, self.telefone,
        self.endereco, self.site];
}
```

Agora podemos simplesmente rodar novamente a aplicação, que no log aparecerá:

```
Nome: Bruce Wayne, Telefone: 1199999-9999, Endereço: Ghotam City, Site: www.wayneenterprises.com
```

5.7 EXERCÍCIO - CRIANDO UM CONTATO A PARTIR DE UM FORMULÁRIO

Agora que já temos um conhecimento mais amplo sobre o que realmente acontece quando uma propriedade é declarada, vamos criar propriedades na classe `Contato`.

1. Vamos declarar as propriedades do `Contato`. Queremos garantir que, após instanciado, um objeto do tipo `Contato` tenha sempre disponível os seus próprios dados: **nome**, **telefone**, **endereco** e **site**. Portanto, vamos usar o atributo `strong` na declaração da propriedade para garantir que os objetos associados a essas propriedades não possam ser liberados da memória antes da instância de `Contato` que faz referência a elas.

Edite o arquivo `Contato.h` e adicione a declaração das propriedades:

```
@interface Contato : NSObject

@property (strong) NSString *nome;
@property (strong) NSString *telefone;
@property (strong) NSString *endereco;
@property (strong) NSString *site;

@end
```

2. Altere o arquivo `Contato.m` e adicione o método `description` para podermos ter uma representação dos dados de um contato:

```
@implementation Contato

-(NSString *)description {
    return [NSString stringWithFormat:@"Nome: %@", Telefone: %@,
        Endereço: %@", Site: %@", self.nome, self.telefone,
        self.endereco, self.site];
}

@end
```

3. Para utilizarmos a classe `Contato` é preciso importar seu cabeçalho no arquivo `ContatosIP67-Bridging-Header.h`. Adicione a declaração:

```
#import "Contato.h"
```

4. Agora altere o código do método `pegaDadosDoFormulario` para utilizar uma instância de contato para armazenar os dados do formulário. Na implementação do método:

- Crie uma instância de `Contato`:

```
let contato: Contato = Contato()
```

- Armazene os valores nas propriedades do objeto:

```
contato.nome = self.nome.text!
```

Ao final das alterações, essa será a implementação do método:

```
@IBAction func pegaDadosDoFormulario(){
    let contato: Contato = Contato()

    contato.nome = self.nome.text!
    contato.telefone = self.telefone.text!
    contato.endereco = self.endereco.text!
    contato.site = self.site.text!

    print(contato)
}
```

5. Tente adicionar um novo contato. O que acontece?

ARMAZENANDO REGISTROS NA MEMÓRIA: ARRAY

Muitas aplicações modernas precisam lidar com informação de algum tipo: um conjunto de nomes, uma coleção de contatos, uma lista de tarefas. Várias linguagens de programação fornecem algum tipo de estrutura de dados que permite organizar essa informação. No mundo orientado a objetos é comum que essas estruturas sejam implementadas com uma série de comportamentos que permitam agir sobre os dados, de maneira bem fácil, sem ter de se preocupar com a implementação interna.

No estado atual de nossa aplicação já é possível criar um objeto do tipo `Contato` a partir de um formulário. Porém, não estamos armazenando esses contatos em lugar algum, precisamos resolver isso antes de conseguir listar todos os contatos cadastrados, então mãos à obra!

6.1 ARMAZENANDO OBJETOS EM INSTÂNCIAS DE ARRAY

Em Swift há uma classe chamada `Array` que serve para armazenar uma coleção de qualquer tipo de objeto. Os objetos armazenados em um `Array` são indexados numericamente seguindo a ordem de inserção. Vamos criar um array com 3 strings, para isso, vamos utilizar um `Array` literal que recebe uma lista de objetos e retorna um novo array configurado com esses objetos:

```
let contatos: Array<String> = ["Batman", "Robin", "Coringa"]
```

Como vamos visualizar o que há dentro do array? Apenas para verificar se o `array` foi criado corretamente, basta "imprimir" na tela o conteúdo desse objeto. Vamos percorrer toda a lista e usar a função `print` para exibir o valor de cada item armazenado.

Existe um tipo de construção específica para "andar por uma lista", chamada *foreach*. Essa construção vai permitir escrever um código que pode ser lido da seguinte maneira: para **cada** string no array chamado `contatos`, crie uma variável `contato` use a função `print` para exibir o valor dessa variável. Veja a implementação:

```
for contato:String in contatos {
    print(@"Nome: \(contato)")
}
```

Você pode recuperar uma referência para qualquer objeto dentro dessa nossa coleção por meio de seu índice, como, por exemplo, `c:Contato = contatos[5]`.

Alterando o conteúdo de um Array

Quando declaramos uma variável com `let` definimos que esse objeto é imutável ou seja uma constante. Quando declaramos nosso array `let contatos: Array<String> = ["Batman", "Robin", "Coringa"]` temos um array que não podemos adicionar nem remover informações dele.

Vamos criar um novo array, mas agora utilizando um objeto mutável para que nossa lista possa ser atualizada durante a execução do programa. Para isso iremos utilizar a declaração `var` ao invés de `let` com isso temos uma `Array` mutável.

```
var nomes: Array<String> = ["Jobs", "Wozniak"]
```

Vamos atualizar essa lista adicionando dois nomes (dois objetos do tipo `String`). Para isso, usaremos o método `append`:

```
nomes.append("Ballmer")
nomes.append("Wayne")
```

Ops... adicionamos um nome por engano em nosso array! Sem problemas, como o array que estamos utilizando é mutável, além de adicionar também é possível remover objetos. Podemos remover o último nome usando o método `removeLast`:

```
nomes.removeLast()
```

Ou remover de uma determinada posição com o método `removeAtIndex` passando o índice que queremos remover:

```
nomes.remove(at:3)
```

Há ainda muitos outros métodos para auxiliar a manipulação dos objetos armazenados em nosso array.

6.2 ARMAZENANDO CONTATOS NA MEMÓRIA

Muito bem, agora que já sabemos como armazenar nossos contatos em memória usando uma instância de `Array`, podemos alterar a classe `FormularioContatoViewController` para guardarmos todos os nossos contatos. No arquivo `FormularioContatoViewController.swift` declaramos a propriedade `contatos` usando a diretiva `var`:

```
var contatos: Array<Contato>
```

6.3 CUSTOMIZANDO A INICIALIZAÇÃO DE OBJETOS: O MÉTODO INIT

Agora que temos uma propriedade para guardar os contatos, precisamos instanciá-la. Em que momento devemos fazer isso? Precisamos garantir que essa propriedade seja criada no mesmo momento em que o formulário for criado.

Toda vez que um objeto do tipo `UIViewController` é criado, o próprio *iOS* invoca o método `init` nesse objeto. Nossa classe `FormularioContatoViewController` herda de `UIViewController`, isso fica claro na declaração da classe: `class FormularioContatoViewController: UIViewController`. Portanto, podemos sobrescrever o método inicializador `init` em nossa classe e ter a garantia de que ele será invocado pelo *iOS*.

Porém, é preciso ter bastante cuidado quando sobrescrevemos o método `init`. A implementação original desse método é feita na classe `NSObject` e sua responsabilidade é tornar os objetos aptos a serem utilizados em um programa. Quando optamos por sobrescrevê-lo é preciso garantir que a implementação original continue sendo invocada.

Para invocar um método em uma classe pai na hierarquia de uma herança em *Swift* podemos utilizar a palavra chave `super`. Para invocar o método `init` original podemos usar o seguinte: `super.init()`.

O código a seguir demonstra um padrão bem comum de sobrescrita do método `init`, seria o equivalente ao **construtor** de outras linguagens.

```
override init() {
    // aqui fica o código das customizações

    super.init()
}
```

Dessa forma, toda a inicialização do objeto realizada na implementação original do método `init` em `NSObject` é utilizada em nosso próprio método `init`.

Mas se fizermos desta forma, nosso método `init` não será executado! Acontece que quem instancia o `FormularioContatoViewController` é o `Storyboard` e ele não utiliza o `init` para instanciar os `view controllers`. O `init` é um inicializador que não recebe nenhum parâmetro de entrada e o `Storyboard` precisa passar algumas informações para o objeto ser instanciado corretamente por isso ele utiliza outro construtor. Devemos usar então o construtor `initWithCoder:` para personalizar a criação de nosso objeto. Se tentássemos utilizar o método `init` o compilador não deixaria, pois não é o método inicializador designado:

```
required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
}
```

Como o método `initWithCoder` é o método designado para inicialização, devemos utilizar o modificador *required* ao invés *override*. Essa é a forma que temos para notificar ao compilador que o inicializador requerido/designado está sendo utilizado.

Agora que temos nosso próprio método inicializador, podemos simplesmente inicializar o `Array`:

```
required init?(coder aDecoder: NSCoder) {
    self.contatos = Array()
```

```
super.init(coder: aDecoder)
}
```

Perfeito, agora temos disponível um `Array` e podemos adicionar um novo contato criado a essa lista. Atualize o método `pegaDadosDoFormulario` e adicione o código para armazenar o `Contato` recém criado na lista:

```
@IBAction func pegaDadosDoFormulario(){
    let contato: Contato = Contato()

    contato.nome = self.nome.text!
    contato.telefone = self.telefone.text!
    contato.endereco = self.endereco.text!
    contato.site = self.site.text!

    contatos.append(contato)
}
```

Que tal? Direto ao ponto, certo? Mas será que cuidar deste array de contatos é responsabilidade do próprio formulário? Vamos precisar utilizar este mesmo array em outras partes de nossa aplicação, então onde podemos colocar este código?

6.4 BOAS PRÁTICAS DE ORIENTAÇÃO A OBJETOS: DIVIDINDO RESPONSABILIDADES

Vamos separar esta funcionalidade em uma classe que será responsável por cuidar do acesso aos dados de nossos contatos, deixando nosso código mais limpo. Esta classe segue um padrão de projetos conhecido como **Dao**, muito utilizado em linguagens Orientadas a Objetos.

Vamos criar uma classe chamada `ContatoDao` que cuidará de manipular o armazenamento dos contatos de nossa aplicação. Esta classe deverá ser filha também de `NSObject`. Com a classe criada, podemos transferir a criação do `NSMutableArray` de contatos para ela. Como este array somente poderá ser manipulado dentro da própria classe `ContatoDao` vamos deixar a propriedade como somente leitura, através da palavra `readonly`:

```
import Foundation

class ContatoDao: NSObject {
    var contatos: Array<Contato>
}
```

Agora temos que criar o método que adicionará o contato no array e inicializar o array `contatos` no método `init`, do mesmo modo que estávamos fazendo anteriormente no `init` do formulário.:

```
import Foundation

class ContatoDao: NSObject {
    var contatos: Array<Contato>
```

```

override init(){
    self.contatos = Array()
    super.init()
}

func adiciona(_ contato:Contato){
    contatos.append(contato)
}

}

```

Alteramos então o `FormularioContatoViewController` para utilizar esta nova classe e alteramos o `initWithCoder:` e o `pegaDadosDoFormulario`:

```

var dao:ContatoDao

required init?(coder aDecoder: NSCoder) {
    self.dao = ContatoDao()

    super.init(coder: aDecoder)

}

@IBAction func pegaDadosDoFormulario(){
    let contato: Contato = Contato()

    contato.nome = self.nome.text!
    contato.telefone = self.telefone.text!
    contato.endereco = self.endereco.text!
    contato.site = self.site.text!

    dao.adiciona(_ contato)
}

```

Mas toda vez estamos fazendo com que um objeto do tipo `ContatoDao` seja criado. Podemos garantir que este objeto seja único na memória utilizando outro padrão chamado **Singleton**. Se já tivermos instanciado o `ContatoDao`, utilizamos a mesma instância, caso contrário, instanciamos o objeto. Vamos também implementar o padrão **Factory** para gerar esta instância única. Existe uma convenção criada pela Apple para nomearmos métodos de fábrica. O método deve ser de classe e começar com o nome do objeto que queremos gerar, ou seja, o nome de nossa própria classe `ContatoDao`. O código ficará da seguinte forma seguindo esta convenção.

```

import Foundation

class ContatoDao: NSObject {

    static private var defaultDAO: ContatoDao!
    private var contatos:Array<Contato>

    static func sharedInstance() -> ContatoDao {
        if defaultDAO == nil {
            defaultDAO = ContatoDao()
        }
        return defaultDAO
    }
}

```

```

    }

    override private init(){
        self.contatos = Array()
        self.init()
    }

    func adiciona(_ contato:Contato){
        contatos.append(contato)
    }
}

```

Outro detalhe importante é que adicionamos o modificador *private* ao método `init`. Dessa forma nenhuma outra classe pode chamar esse método.

6.5 EXERCÍCIO - ARMAZENANDO CONTATOS COM NSMUTABLEARRAY

1. Crie a classe `ContatoDao`. Não se esqueça de colocar que ela é filha de `NSObject`.
 - Utilize o atalho *Command + N* para criar um novo arquivo. Na janela exibida, dentro do menu *Source*, selecione a opção *Cocoa Touch Class*. Clique em *Next*.
 - A próxima tela permite configurar o nome da nova classe. Preencha o campo *Class* com `ContatoDao` e o campo *Subclass of* com `NSObject` e a linguagem como `Swift`. Clique em *Next*.
 - Na próxima tela, você pode configurar o local no projeto onde quer criar sua classe, aceite a sugestão do *Xcode* clicando em *Create*.
2. Na classe `ContatoDao` declare as propriedades `contatos` do tipo `Array<Contato>` e `defaultDAO` do tipo `ContatoDao`. Não se esqueça de importar o framework `Foundation`.

```

import Foundation

class ContatoDao: NSObject {

    static private var defaultDAO: ContatoDao!
    var contatos: Array<Contato>
}

```

1. Implemente o método `adiciona`. Ele deve receber um contato e adicioná-lo no Array de contatos. Adicione também uma chamada ao `print` para vermos os contatos adicionados.

```

func adiciona(_ contato:Contato){
    contatos.append(contato)
}

```

1. Ainda no mesmo arquivo implemente os métodos para garantir que teremos somente uma única instância de `ContatoDao` na memória.

```

static func sharedInstance() -> ContatoDao{

```

```

    if defaultDAO == nil {
        defaultDAO = ContatoDao()
    }

    return defaultDAO
}

override private init(){
    self.contatos = Array()
    super.init()
}

```

1. Agora que já temos o dao pronto, vamos adequar a classe `FormularioContatoViewController`.
 - Adicione a declaração de uma propriedade do tipo `ContatoDao` chamada `dao`.
 - Sobrescreva o método `init` na classe `FormularioContatoViewController` e recupere a instância do `ContatoDao`
 - Altere o método `pegaDadosDoFormulario` para armazenar os contatos criados a partir dos dados do formulário utilizando o `dao`.

O código final deve ficar algo do tipo:

```

class FormularioContatoViewController : UIViewController {
    var dao:ContatoDao

    required init?(coder aDecoder: NSCoder) {
        self.dao = ContatoDao()

        super.init(coder: aDecoder)

    }

    @IBAction func pegaDadosDoFormulario(){
        let contato: Contato = Contato()

        contato.nome = self.nome.text!
        contato.telefone = self.telefone.text!
        contato.endereco = self.endereco.text!
        contato.site = self.site.text!

        dao.adiciona(_ contato)
    }
}

```

1. Execute o programa e insira um novo contato na lista, você verá a seguinte saída no console do `Xcode`:

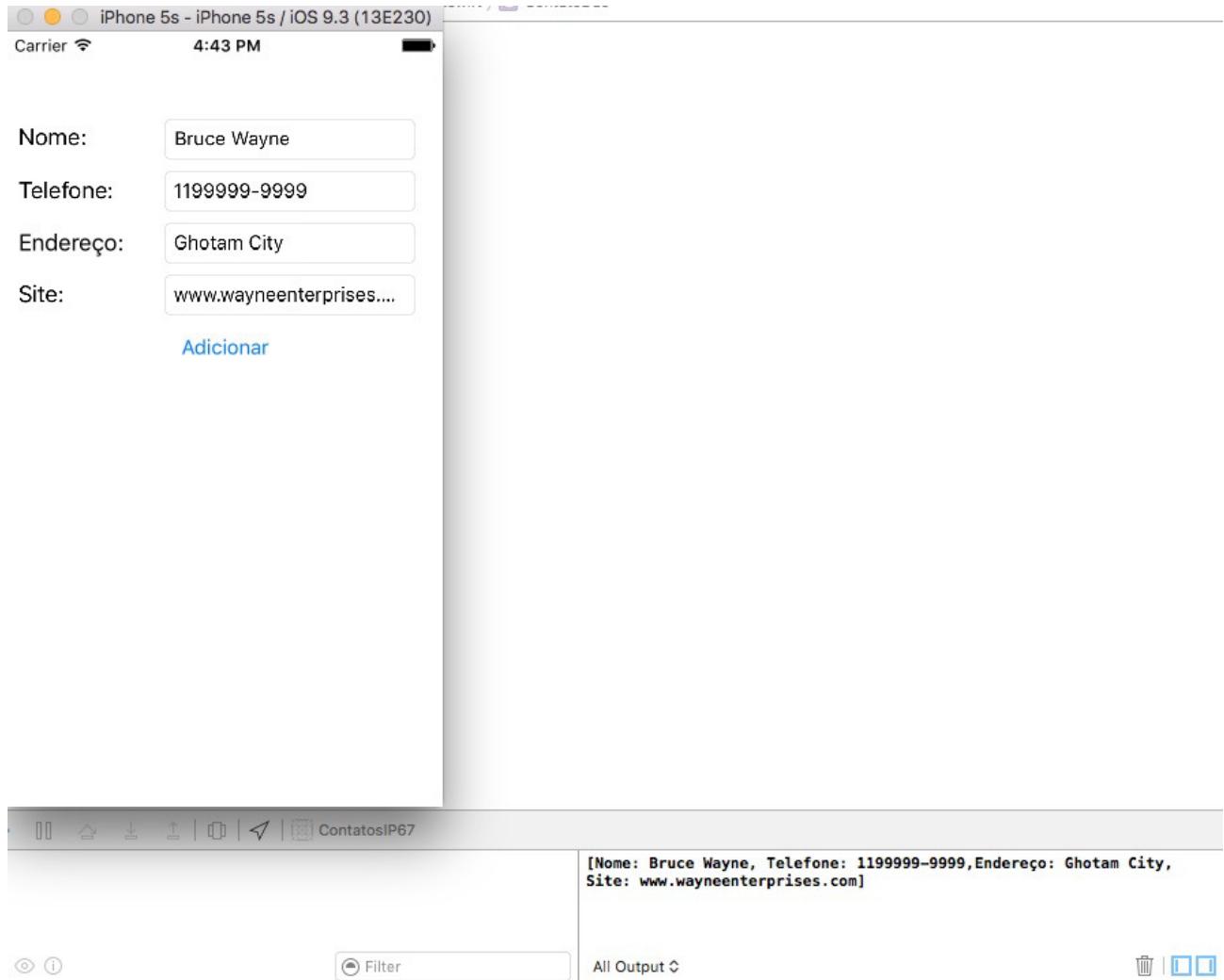


Figura 6.1: Armazenando contatos

UM COMPONENTE PARA LISTAR REGISTROS: UITABLEVIEWCONTROLLER

A aplicação já permite que um usuário cadastre novos contatos, porém, não há ainda nenhuma funcionalidade que permita a visualização desses cadastros. Para resolver isso usaremos um componente do *iOS* chamado `UITableViewController`. Um fato interessante sobre esse componente é que ele é o mesmo utilizado nos aplicativos nativos do *iOS*.

A vantagem de usar componentes desse tipo é que o usuário final da nossa aplicação vai saber como interagir com ela, mesmo que seja a primeira vez que abre a aplicação. Isso porque os aplicativos da própria Apple já fizeram um ótimo trabalho ensinando aos usuários como interagir com determinados componentes. Porém, isso também traz uma responsabilidade: quando usamos componentes amplamente utilizados, o ideal é que ele se comporte exatamente como nas aplicações nativas para que o usuário não tenha surpresas estranhas enquanto usa o aplicativo.

No *iOS* é bastante comum a utilização de tabelas para listar itens. Podemos ver essa funcionalidade em aplicativos como o *Music*, o *Clock*, o *Settings*, ou até mesmo o *Contacts* que é a inspiração para aplicação em que estamos trabalhando.



Figura 7.1: Exemplos de lista em aplicações nativas

Todos esses aplicativos tem em comum o uso de uma instância do tipo `UITableViewController` que, por sua vez, é apenas uma especialização da classe `UIViewController`. Lembra-se da classe `FormularioContatoViewController`? Na declaração dessa classe foi declarada uma herança com a classe `UIViewController`, o que torna o nosso formulário apto a se comunicar com a *view* colocada no arquivo `Main.storyboard` e capturar seus eventos.

7.1 VIEW CONTROLLERS SÃO APENAS CLASSES

Muito bem, o que precisamos fazer então é criar a lógica de listagem de contatos, mas não precisamos fazer isso do zero, pois vamos aproveitar toda a lógica que já foi implementada na classe `UITableViewController`. Nossa classe de listagem vai se chamar `ListaContatosViewController`.

Primeiramente, criaremos uma nova classe chamada `ListaContatosViewController`. No *Xcode* clique em *File* -> *New* -> *File* ou, simplesmente, use o atalho: *Command* + *N*. Na janela, selecione na coluna à esquerda *iOS* -> *Source*. Nas opções exibidas à direita, selecione *Cocoa Touch Class*. Na próxima janela, configure o nome da classe e, para a opção *Subclass of*, selecione `UITableViewController` e em linguagem selecione `Swift` e clique em *Next*. Feito isso importar as classes da biblioteca `UIKit`.

Uma tabela no *iOS* é representada por uma instância de `UITableView`. Essa classe descreve um componente visual que já tem diversos comportamentos associados à listagem como o *scroll* vertical, a seleção de um item, entre outras funcionalidades. A classe `UITableViewController` tem uma série de métodos que podem ser invocados pelo componente visual para informar ao *controller* que é preciso reagir a alguma interação do usuário, é por isso que nossa classe `ListaContatosViewController` herdar de `UITableViewController`. Dessa forma, temos um controller que será personalizado com a nossa lógica de listagem e, ao mesmo tempo, já tem a habilidade de se comunicar com o componente nativo de listagem do *iOS*.

```
import UIKit

class ListaContatosViewController: UITableViewController {
```

```
}
```

Perfeito, mas não há nenhuma tela no `storyboard` ou nenhum arquivo `xib` associado com essa classe. Precisamos fazer isso, certo? Vamos adicionar um `TableViewController` em nosso `storyboard`.

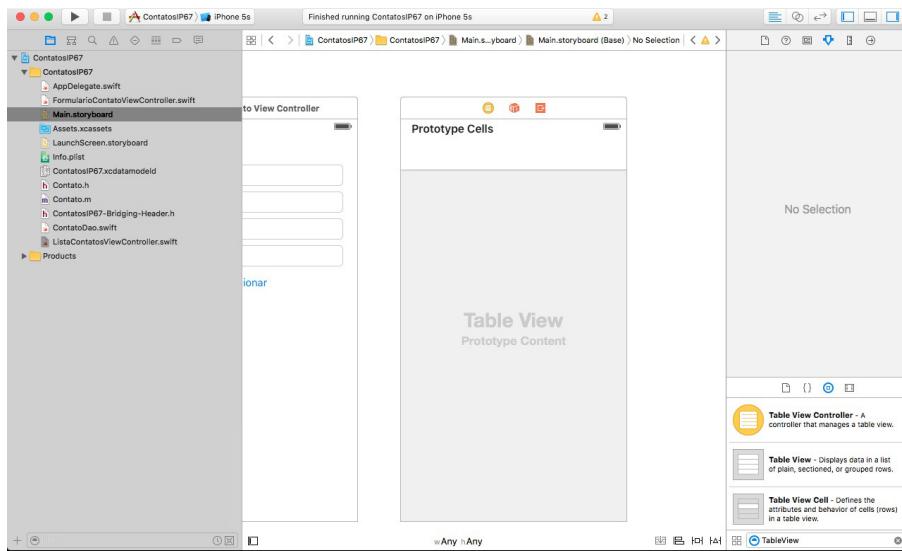


Figura 7.2: Adicionando TableViewController

Vamos associar nossa classe `ListaContatosViewController` ao nosso componente `TableViewController` na nossa `storyboard`.

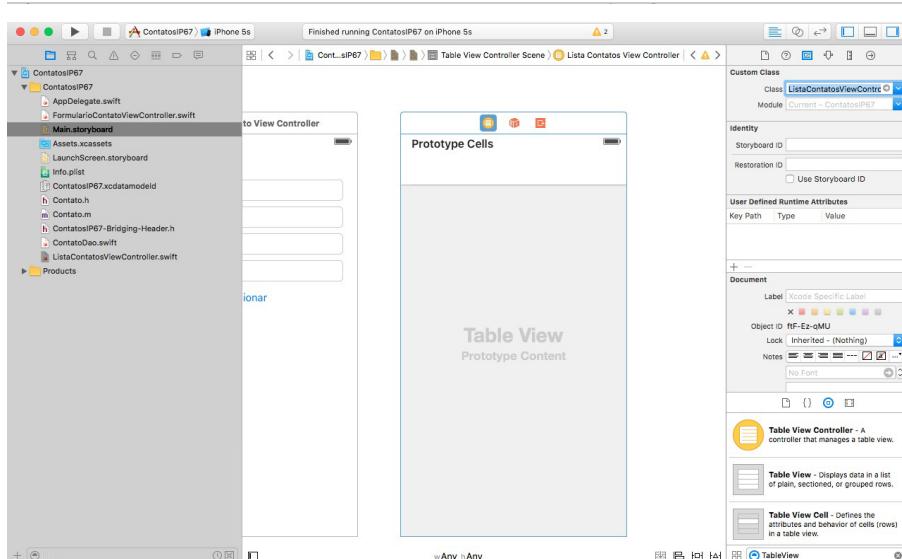


Figura 7.3: Associando TableViewController com nossa classe

Na verdade, não precisamos necessariamente de uma tela no `Storyboard` ou um arquivo `.xib`. Utilizamos os arquivos deste tipo apenas para facilitar a criação de componentes visuais. O *Interface Builder* é uma ótima ferramenta para construção de interfaces e facilita muito a criação de telas apenas arrastando componentes. Porém, todo componente visual nada mais é do que uma instância de um objeto.

Atualmente, a tela principal de nossa aplicação é o formulário. Para verificar se está tudo em ordem com o *controller* que criamos, vamos tornar o *TableViewController* a raiz em nossa aplicação.

Para isso iremos mover o ponto inicial da nossa aplicação para nosso *TableViewController*:

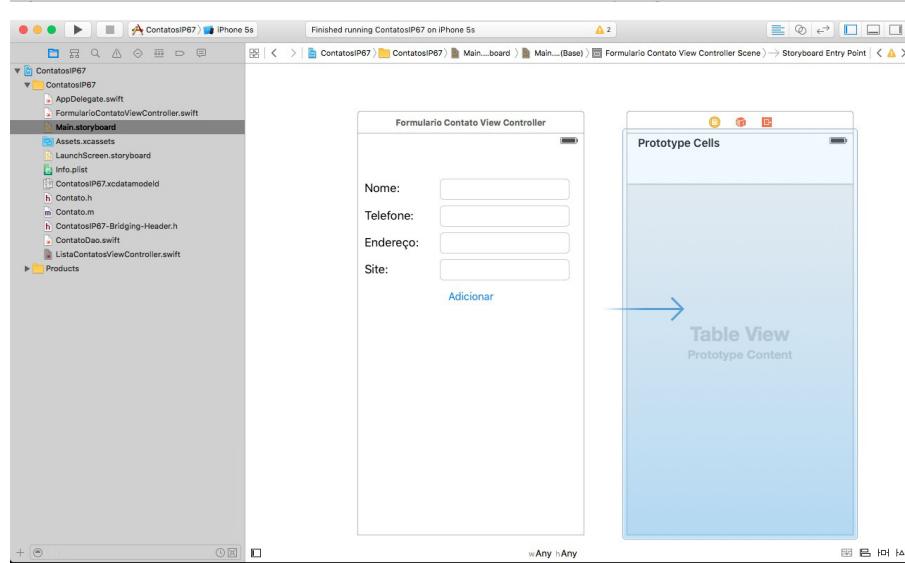


Figura 7.4: Alterando ponto inicial da nossa aplicação

Ao rodar o projeto no simulador, vemos o efeito de nossa alteração: uma tabela vazia é exibida assim que a aplicação é carregada, veja:

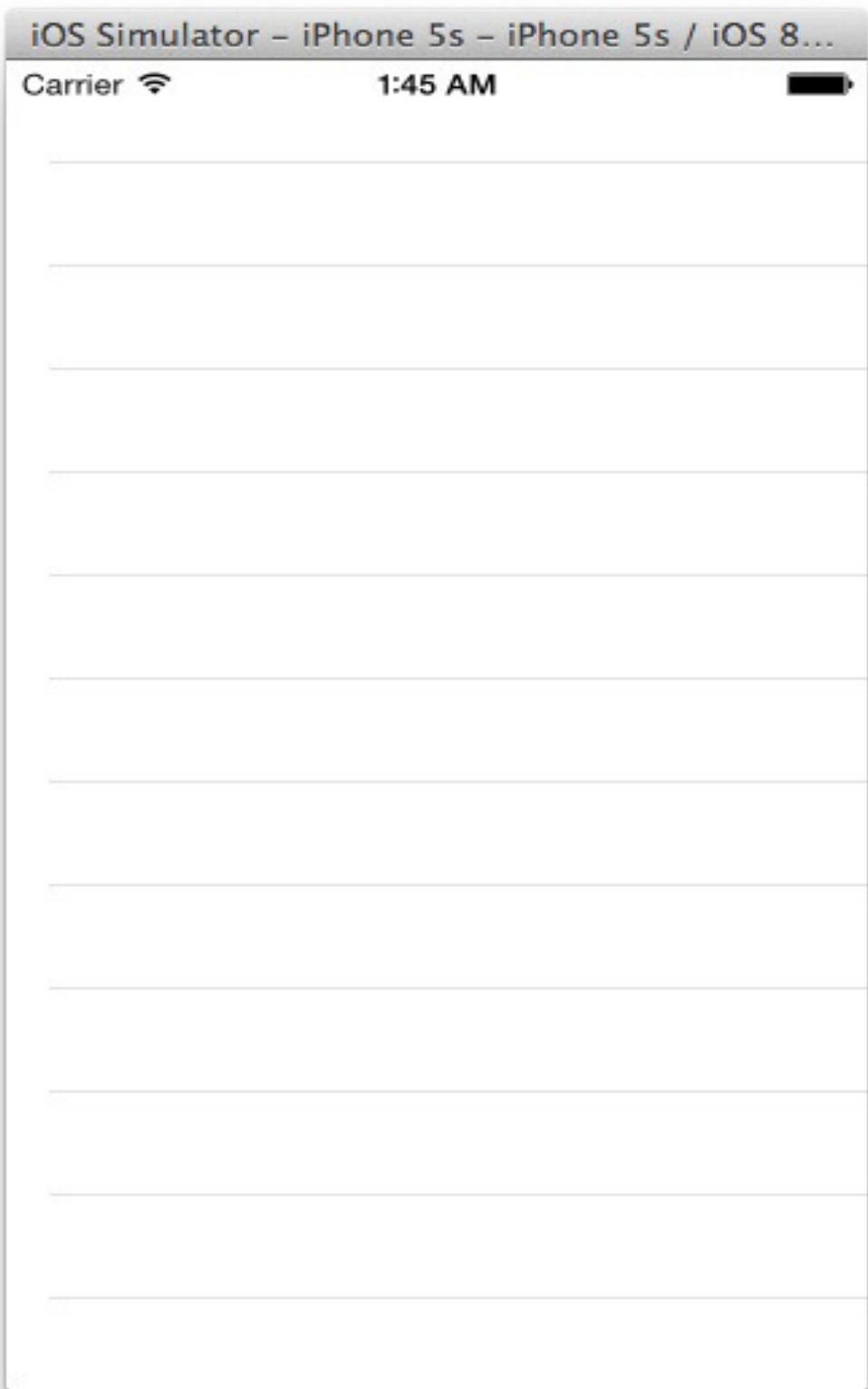


Figura 7.5: Primeira tela da aplicação com lista vazia

7.2 EXERCÍCIO - UM UITABLEVIEWCONTROLLER COMO TELA PRINCIPAL DA APLICAÇÃO

1. Crie uma nova classe *Swift* chamada `ListaContatosViewController` .
 - Utilize o atalho *Command + N* para criar um novo arquivo. Na janela exibida, dentro do menu *Source*, selecione a opção *Cocoa Touch Class*. Clique em *Next*.
 - A próxima tela permite configurar o nome da nova classe. Preencha o campo *Class* com `ListaContatosViewController` e o campo *Subclass of* com `UITableViewController` e em linguagem selecione `Swift` . Clique em *Next*.
 - Na próxima tela, você pode configurar o local no projeto onde quer criar sua classe, aceite a sugestão do *Xcode* clicando em *Create*.

2. Não esqueça de importar a biblioteca `UIKit` na sua `ListaContatosViewController` :

```
import UIKit

class ListaContatosViewController: UITableViewController {
```

```
}
```

3. Adicione um *TableViewController* no storyboard e associe a classe `ListaContatosViewController` com o *TableViewController*.

4. Altere o ponto inicial da sua aplicação para o *TableViewController*.

Rode a aplicação e verifique se a tabela vazia foi carregada no simulador como na figura abaixo:

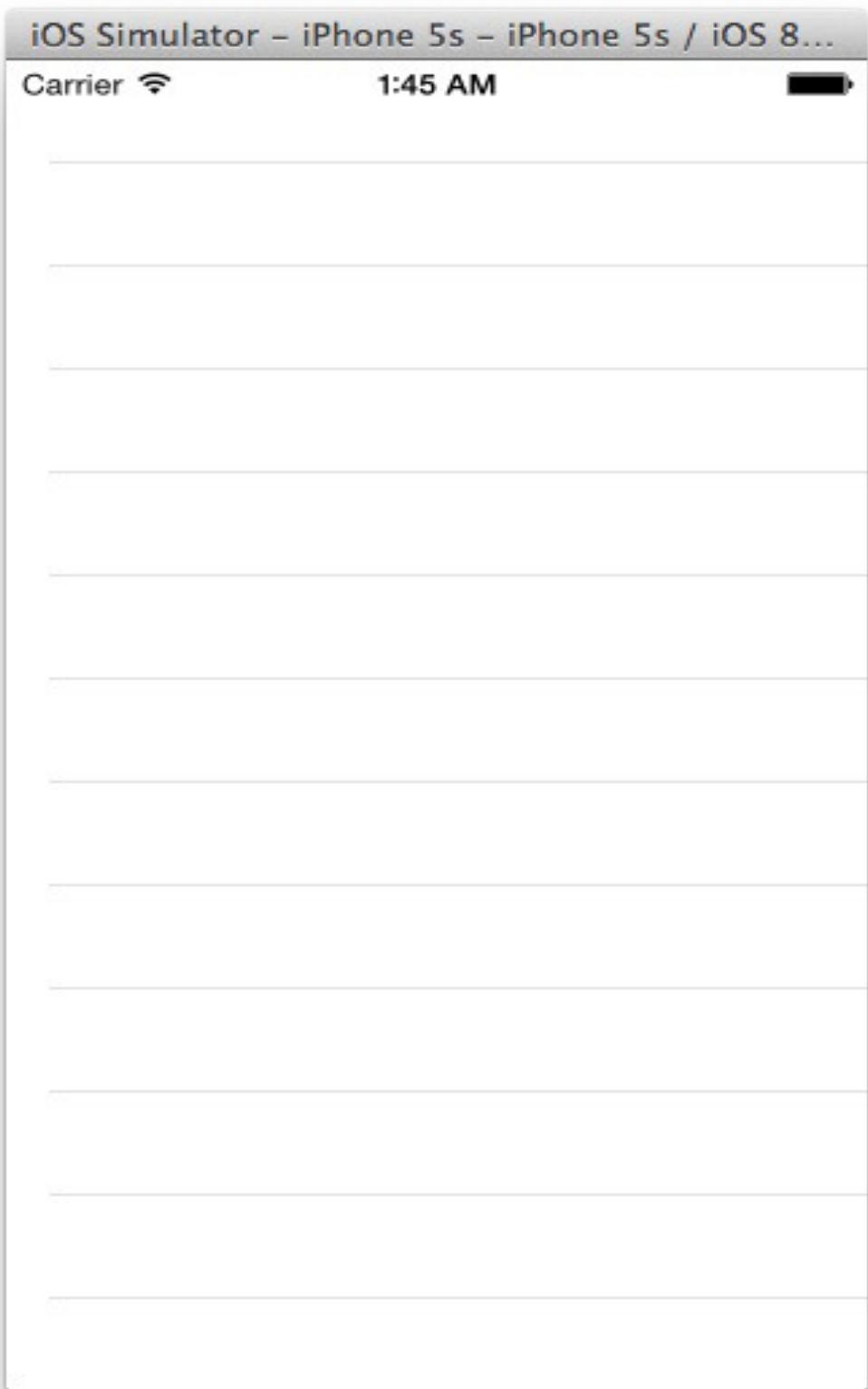


Figura 7.6: Listagem de contatos como tela principal da aplicação

Porém, isso não é o bastante pois a listagem está vazia. A seguir, vamos ver o que é preciso para visualizar os contatos cadastrados.

NAVEGANDO ENTRE TELAS DE FORMA INTUITIVA

Agora, sempre que a aplicação é carregada pelo usuário, a listagem é exibida. Se um usuário precisar cadastrar um contato, ele simplesmente... não consegue. Precisamos fornecer algum mecanismo que permita ao usuário navegar até a tela de cadastro, que já foi criada anteriormente.

Inspirados pelo aplicativo de contatos nativos do *iOS*, criaremos um botão na tela de listagem que exibe a tela de cadastro. Esse botão será exibido no topo da tela e terá o símbolo +. Quando o usuário clicar - ou usando o termo que aparece nos métodos da *api* de gestos do *iOS*: realizar um *tap* - nesse botão, o formulário vai aparecer sobre a atual tela de listagem de uma forma elegante, usando uma animação.

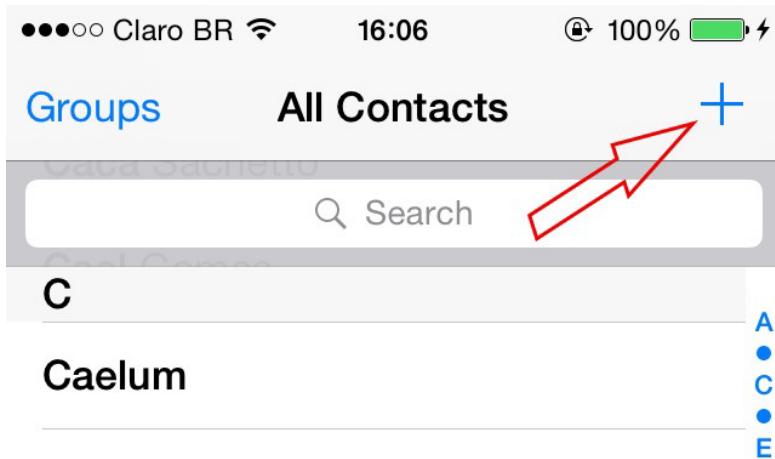


Figura 8.1: Destaque para o botão + na aplicação Contacts

8.1 UINAVIGATIONCONTROLLER: O COMPONENTE PADRÃO PARA NAVEGAR ENTRE TELAS

Parece um bocado de trabalho a ser feito, não? Será surpresa quando for dito que a maior parte dessas funcionalidades já estão prontas? Iremos nos basear em componentes nativos do sistema e o primeiro que precisamos conhecer é o que permite colocar essa *barra* no topo de uma tela. Além dessa barra visual, esse componente nos fornece a funcionalidade de navegação entre telas, permitindo voltar para as telas anteriores. O nome desse componente é `UINavigationController`. A figura abaixo ilustra o funcionamento interno desse componente:



Figura 8.2: Funcionamento do UINavigationController

O UINavigationController funciona como um *container* de telas e isso tem alguns impactos diretos na forma como criamos uma aplicação. Atualmente, queremos que a listagem seja a primeira coisa visualizada pelo cliente, mas também queremos que essa tela seja exibida dentro de um UINavigationController.

Quando um UINavigationController é instanciado, também é criado o componente visual que pode ser usado para conter telas, nessa parte visual já está embutida a barra de navegação que aparece no topo da tela quando usamos o *navigation*. O que faremos, então, é usar um NavigationController como o *view controller root* em nossa aplicação.

Adicione o NavigationController ao storyboard e apague TableViewController relacionado ao NavigationController.

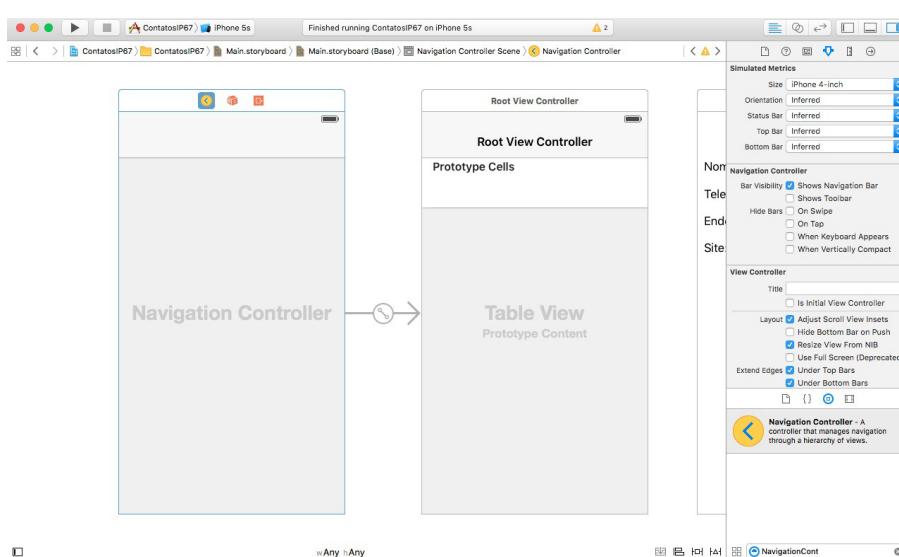


Figura 8.3: Adicionando NavController

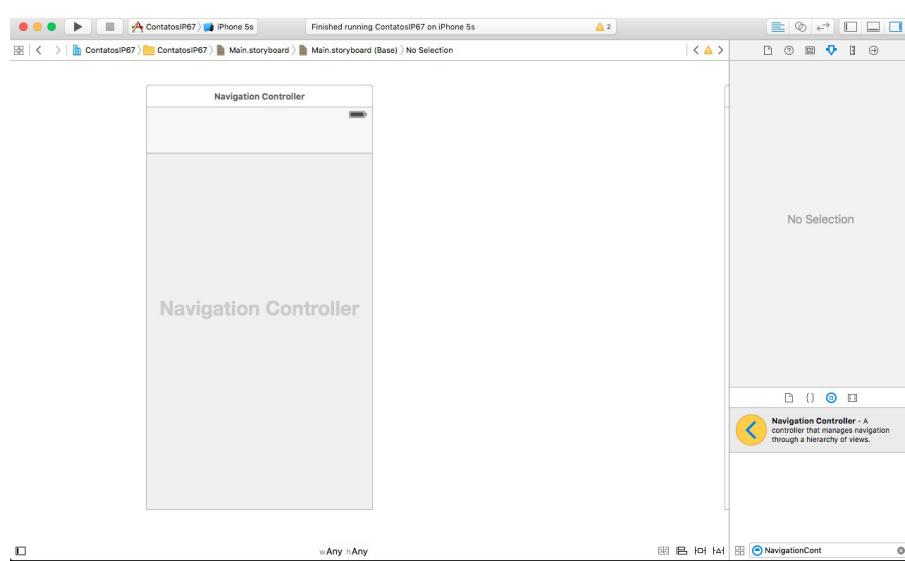


Figura 8.4: Removendo UITableViewController associado ao UINavigationController

Altere a raiz da aplicação para o `NavigationController`

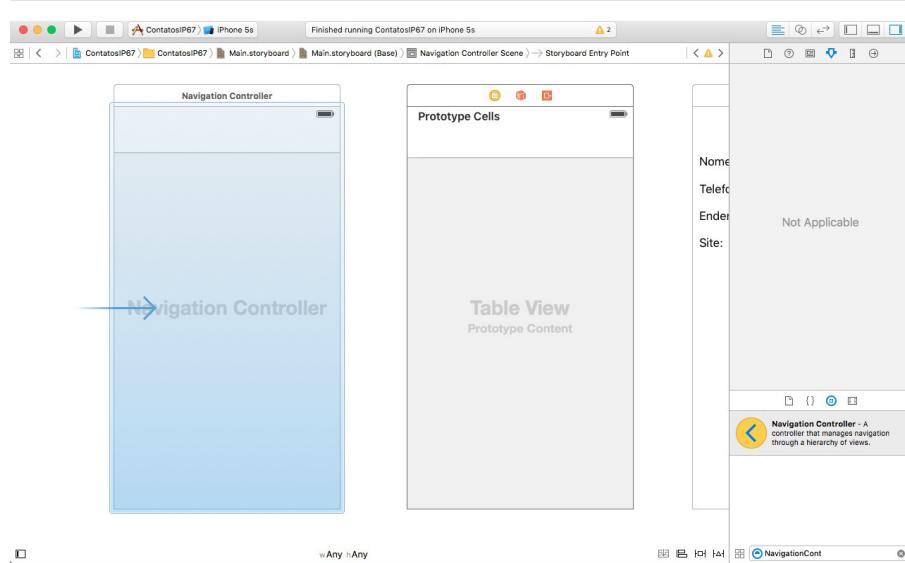


Figura 8.5: Alterando ponto inicial da aplicação para o NavigationController

Relacione o `NavigationController` ao nosso `TableViewController`. Para isso clique sobre o `NavigationController` pressione *CTRL* e mantenha pressionado, e arraste a seleção até o nosso `TableViewController`.

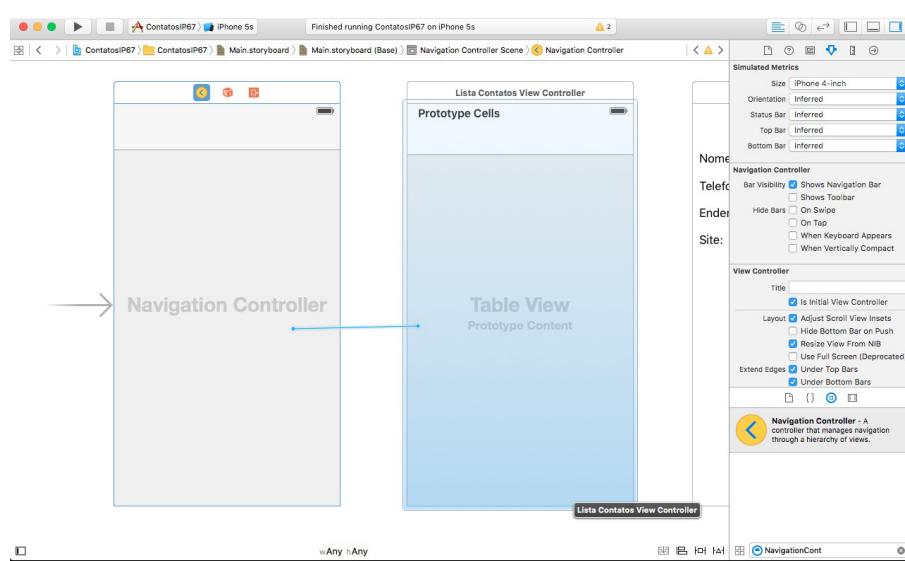


Figura 8.6: Relacionando UINavigationController ao nosso TableViewController

Selecione o relacionamento `root view controller`.

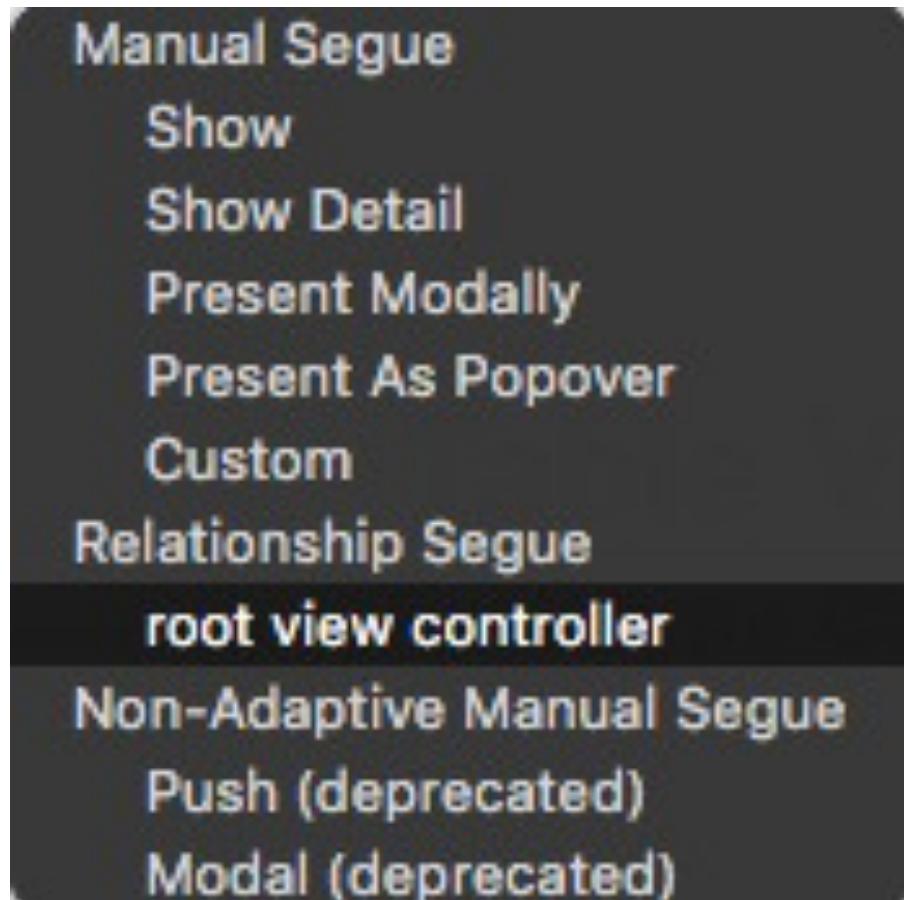


Figura 8.7: Selezionando tipo do relacionamento

Rode a aplicação e perceba que a lista agora é exibida dentro do *navigation controller*:

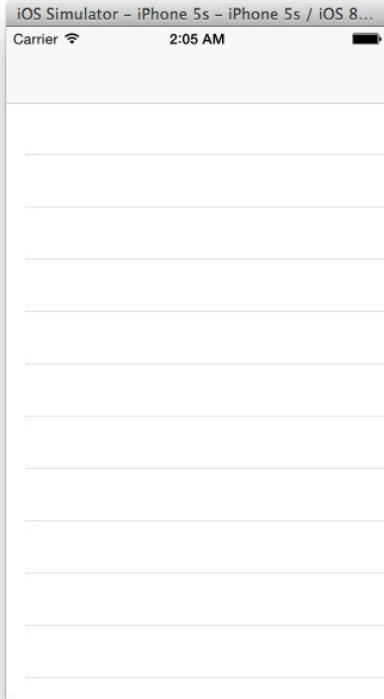


Figura 8.8: Lista exibida em um `UINavigationController`

Porém, perceba que nem o botão + , nem um título foi exibido no *navigation bar* no topo da tela.

8.2 CONFIGURANDO UM TÍTULO PARA UMA TELA EXIBIDA EM UM `UINavigationController`

A classe `UIViewController` nativamente está apta para apresentar sua *view* dentro de um `UINavigationController` . Isso é parte integrante do *iOS*. Sempre que uma classe herda de `UINavigationController` , herda também a propriedade `navigationItem` que se referencia a um objeto do tipo `UINavigationItem` . Podemos pensar nessa propriedade como uma referência para o controller, do ponto de vista de um *navigation controller* , e apenas se ele estiver dentro desse *navigation*.

Parece um pouco complicado, mas na verdade não é. Pense dessa forma: sempre que um `UINavigationController` vai exibir a *view* de um *controller* qualquer, ele pergunta para esse controller informações sobre esse item de navegação que será exibido. Uma das informações que o `UINavigationController` vai buscar sobre o item de navegação é o título que deverá ser exibido na `UINavigationBar`.

Todo *navigation item* tem uma propriedade chamada `title`, podemos atribuir uma `string` a essa propriedade para definir o título que será mostrado em um `UINavigationController` . Faremos isso

alterando a propriedade *title* diretamente no storyboard :

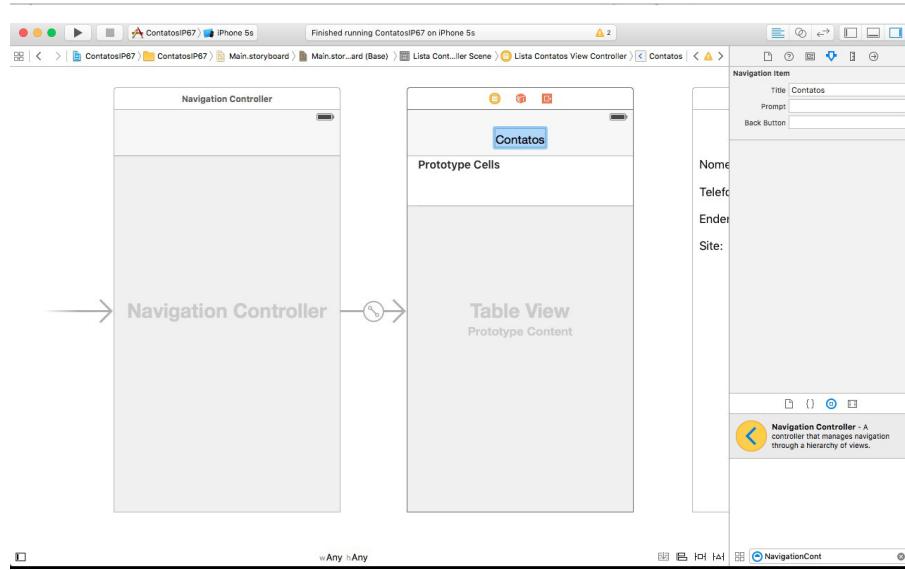


Figura 8.9: Alterando o título do navigationItem

Pronto, agora já temos um título relacionado com a lista, mas precisamos, também, adicionar o botão + ao *navigation bar*.

8.3 ADICIONANDO BOTÕES EM UM NAVIGATION BAR

Podemos exibir botões em um *navigation bar*, mas esses botões precisam ser de um tipo específico. Dentro do *iOS* foi criada uma classe justamente para definir botões que se adequam ao layout de um *UINavigationBar*. Será preciso instanciar esse botão com o uso de um pouco de código.

Os botões que podem ser exibidos em uma instância de *UINavigationBar* são do tipo *UIBarButtonItem*. Vamos adicionar um botão desse tipo, para isso no *Interface Builder* procure por *Bar Button Item* e arraste-o para o lado do título do nosso *Navigation Bar*.

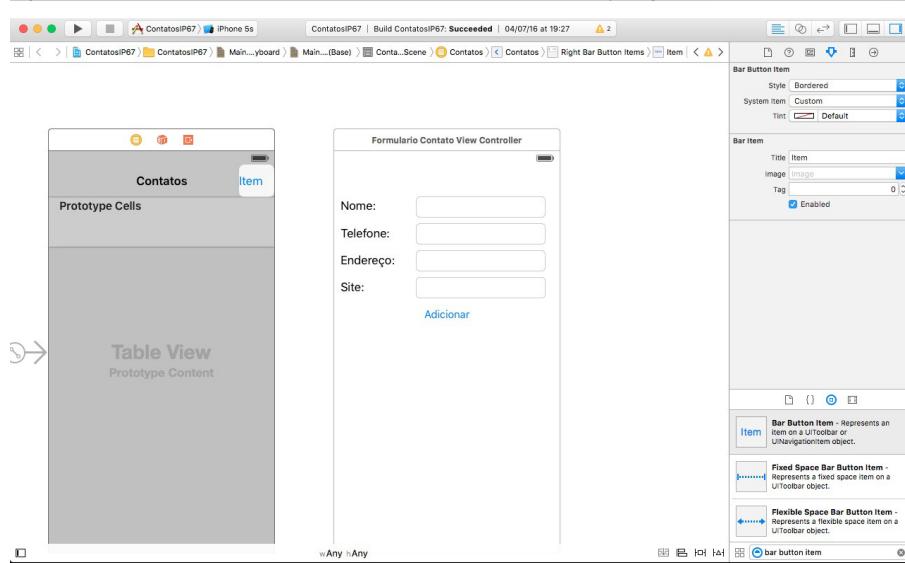


Figura 8.10: Adicionando botão no navigationItem

COMPONENTES VISUAIS SÃO OBJETOS

Vimos alguns componentes visuais até agora. A maioria deles nós definimos por meio de um arquivo `Storyboard`. Porém o arquivo é apenas uma forma de configurar e relacionar instâncias de objetos que herdam de `UIView`. O *Interface Builder* não é a única maneira de adicionar elementos visuais a uma aplicação, podemos criá-los de forma programática. Elementos visuais são apenas objetos com atributos e métodos, e criar um elemento visual nada mais é que instanciar um novo objeto. Poderíamos adicionar o botão usando a seguinte declaração:

```
let botaoExibirFormulario: UIBarButtonItem = UIBarButtonItem()
self.navigationItem.rightBarButtonItem = botaoExibirFormulario;
```

Lembre-se de que não há nenhum motivo para evitar a criação de elementos visuais utilizando o *Interface Builder*. É possível criar componentes apenas com código, mas sempre que possível você pode, e deve, usar as facilidades do *Interface Builder*.

Outro detalhe sobre o botão que queremos exibir é que ele precisa apresentar o sinal + assim como na aplicação *Contacts*. Existe um estilo de botão que podemos indicar ao instanciar *bar button itens*, ao usar esse estilo, o botão automaticamente será exibido com o sinal de +.

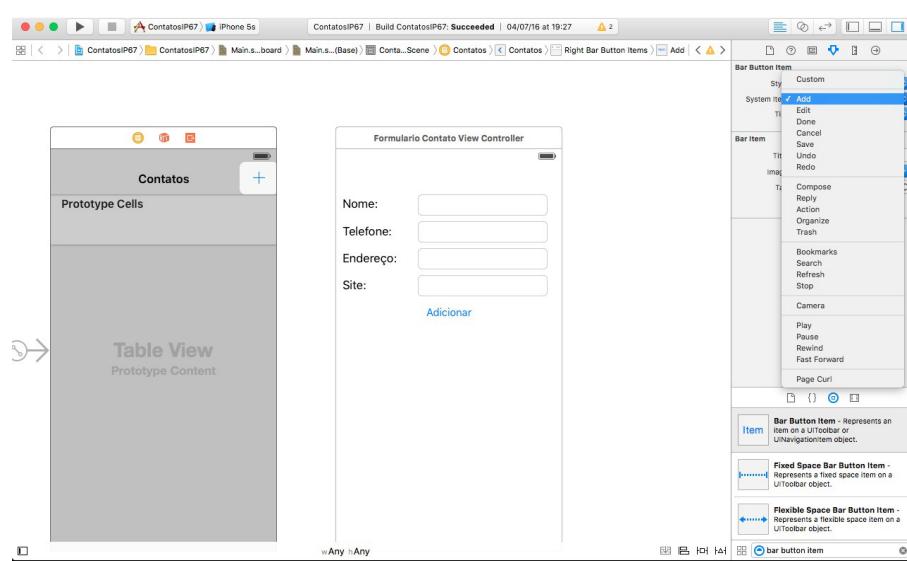


Figura 8.11: Modificando o estilo do barButtonItem

PORQUE USAR COMPONENTES VISUAIS PADRONIZADOS?

A vantagem de usar os estilos e componentes visuais nativos do *iOS* é usar "pistas visuais" bastante conhecidas do usuário da plataforma. Quando um usuário visualizar em nosso aplicativo o botão +, ele provavelmente vai deduzir que o botão aciona a funcionalidade de criação de registros.

Isso não impede que você customize completamente seus componentes visuais. Porém, a ideia de usar componentes padronizados ajuda a produtividade para desenvolvedores (ou mesmo equipes) que não sejam especialistas em design.

8.4 TRANSIÇÕES SIMPLES ENTRE TELAS

Agora que temos nosso botão, queremos que ao clicar nele seja exibido o nosso `FormularioContatoViewController`. Como temos tanto o nossa listagem quanto o nosso formulário no `storyboard` podemos criar uma transição entre as duas telas, essa transição é chamada de `segue`.

Para fazer isso vamos clicar no botão que adicionamos e pressionar `CTRL` e arrastar até nosso formulário.

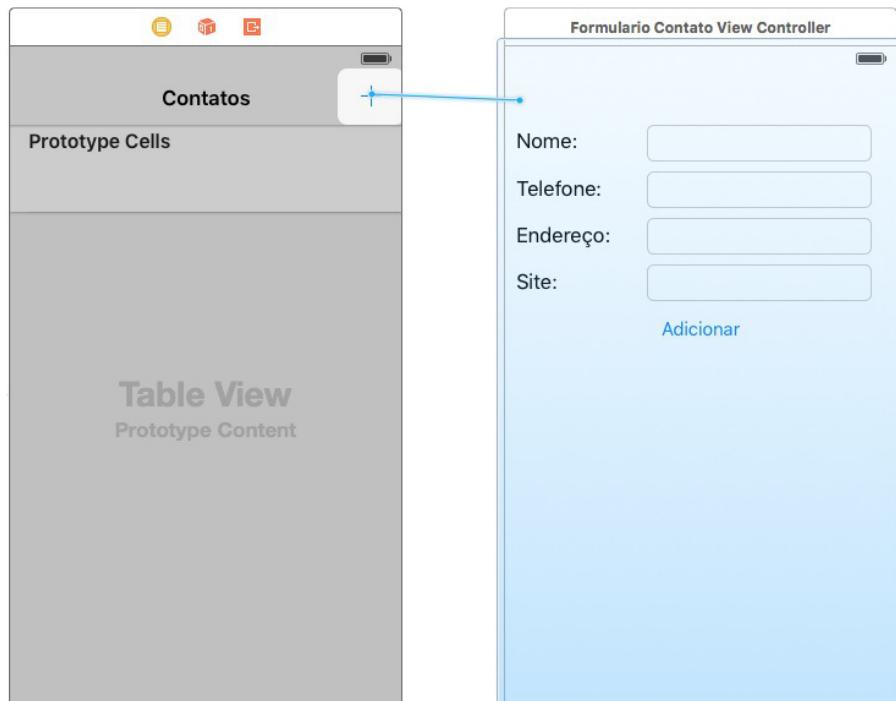


Figura 8.12: Criando transição entre telas

Feito isso vamos selecionar o tipo de transição que queremos utilizar.

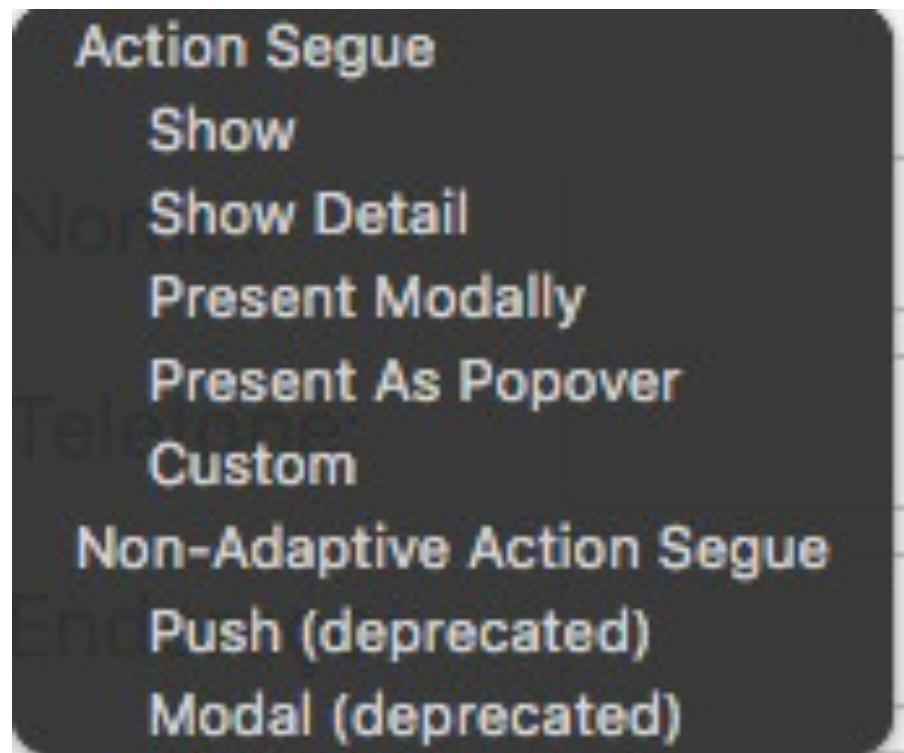


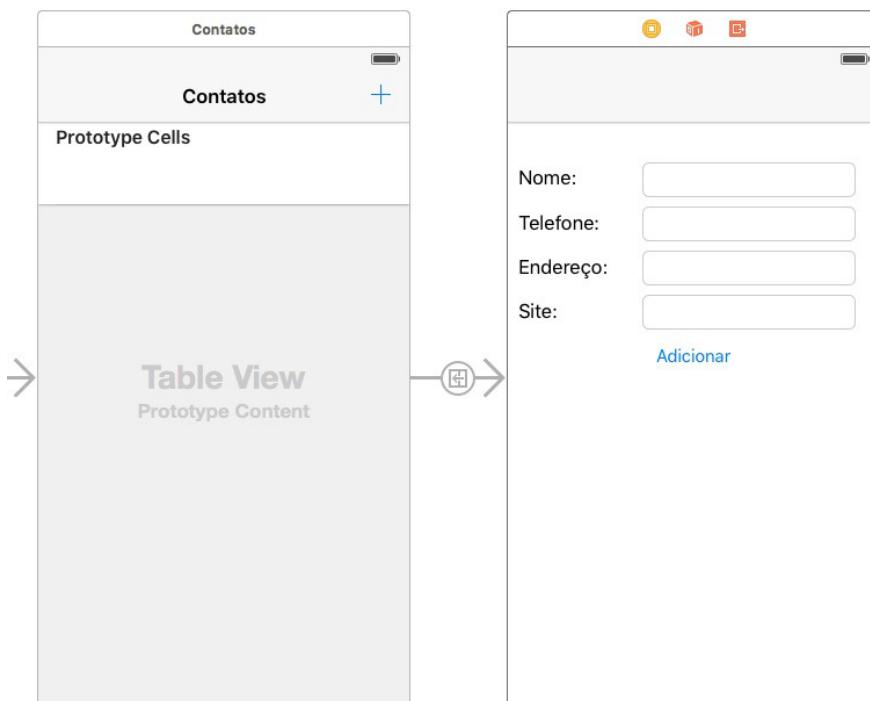
Figura 8.13: Selecionando a transição

TIPOS DE TRANSIÇÕES

- **Show** - Empurrado o conteúdo em cima do View Controller atual.
- **SHOW DETAIL** - Apresentar o conteúdo na área de detalhe. Se o aplicativo está exibindo uma tela/cena no modelo master e detail, o novo conteúdo substitui o detail atual. O símbolo é o mesmo utilizado no segue **Show**
- **PRESENT MODALLY** - Apresenta o conteúdo de forma modal.
- **PRESENT AS POPOVER** - Mostra a nova tela/cena, ligada à tela/cena anterior
- **CUSTOM** - Utilizada quando queremos um controle fino entre as transições e temos que programá-las manualmente.

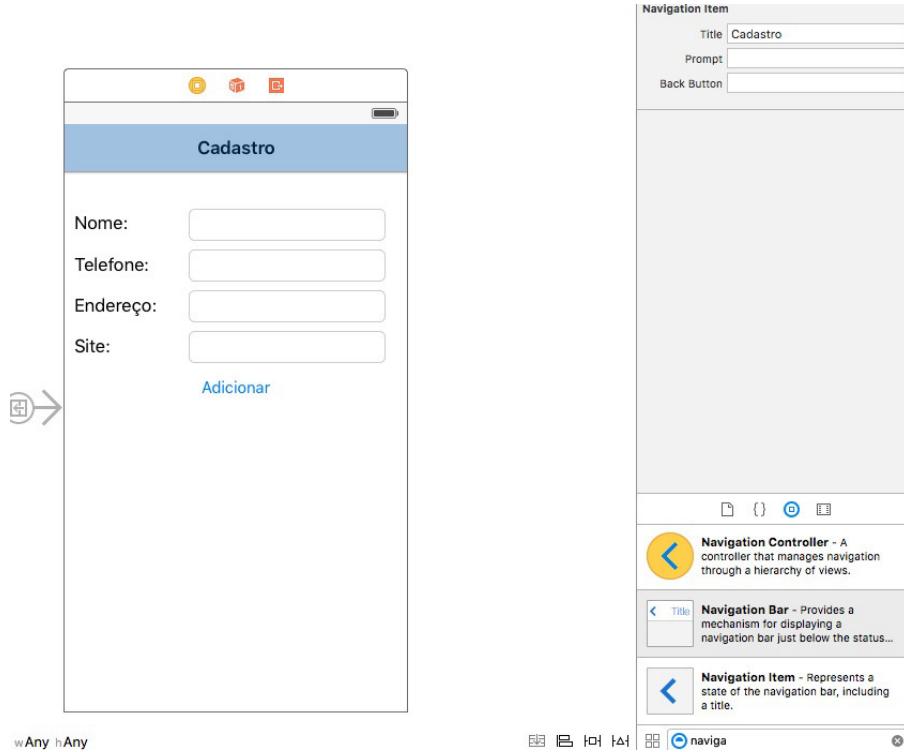
Mais detalhes em <https://goo.gl/DMqvYO>.

Para o nosso caso utilizaremos a transição **Show**. Feito isso nossos storyboard deve estar da seguinte forma:



Perceba que em nenhum momento definimos que nosso formulário está dentro de um `navigationController`. Mas mesmo assim foi adicionado uma barra de navegação no nosso

formulario. Isso ocorreu pois nossa listagem faz parte de um `navigationController` e como fizemos a transição entre nosso formulário e nossa listagem *hedamos* a funcionalidade de navegação. Para concluir vamos definir um título para nosso formulário, para isso adicione um `navigationBar` ao nosso formulário e altere o title do mesmo para **Cadastro**.



8.5 EXERCÍCIO - EXIBINDO O FORMULÁRIO A PARTIR DA LISTAGEM

- Vamos adicionar um *navigation controller* ao nosso *storyboard* :

- Selecione *navigation controller* no *interface builder* e arraste para nossa *storyboard*

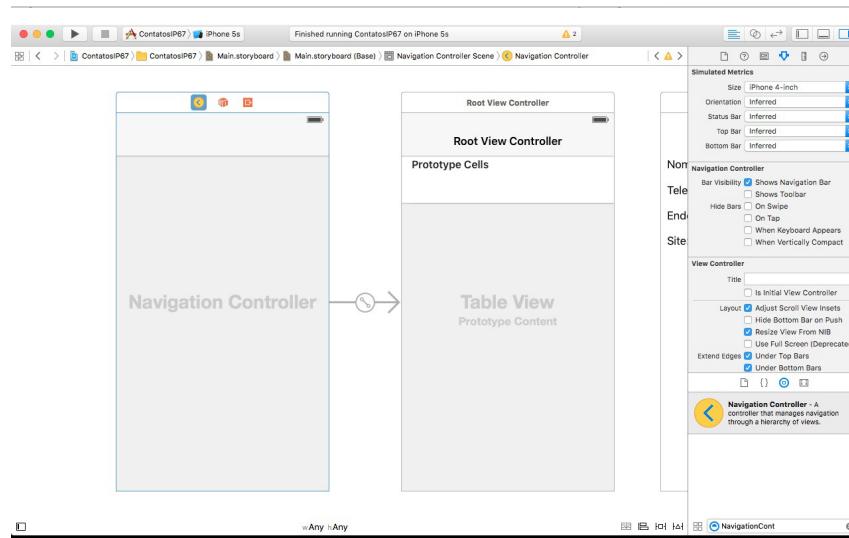


Figura 8.16: Adicionando NavController

- Altere a raiz da aplicação para nosso *navigation controller*

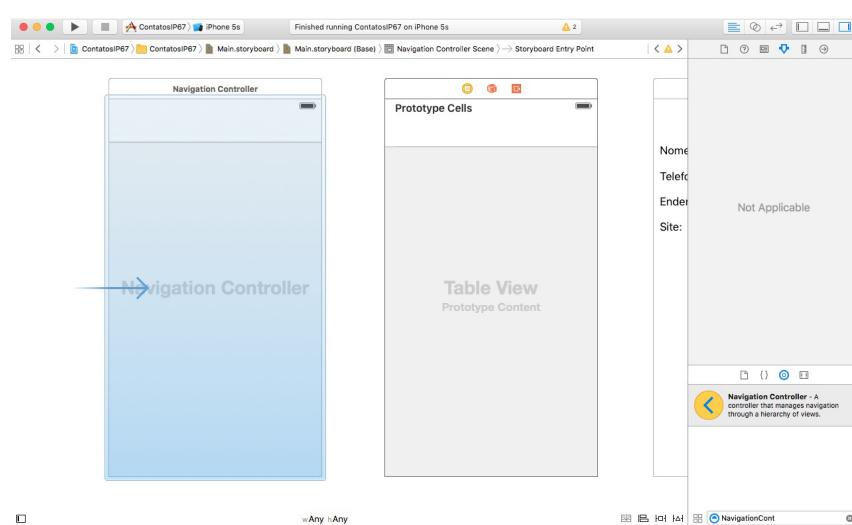


Figura 8.17: Alterando ponto inicial da aplicação para o *NavigationController*

2. Vamos relacionar nossa `ListaContatosViewController` com o nosso *navigation controller*:

- Apague o *tableview controller* relacionado ao *navigation controller*

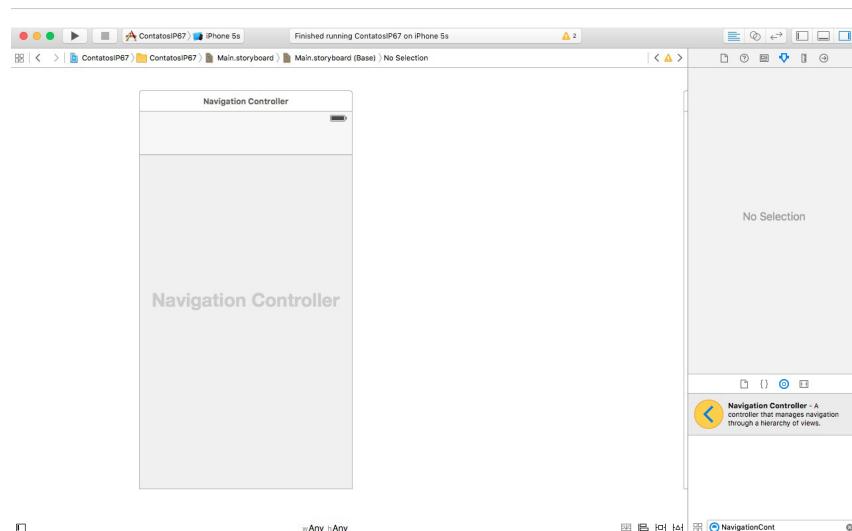


Figura 8.18: Removendo *TableViewController* associado ao *NavigationController*

- Relacione o *navigation controller* com `ListaContatosViewController`.

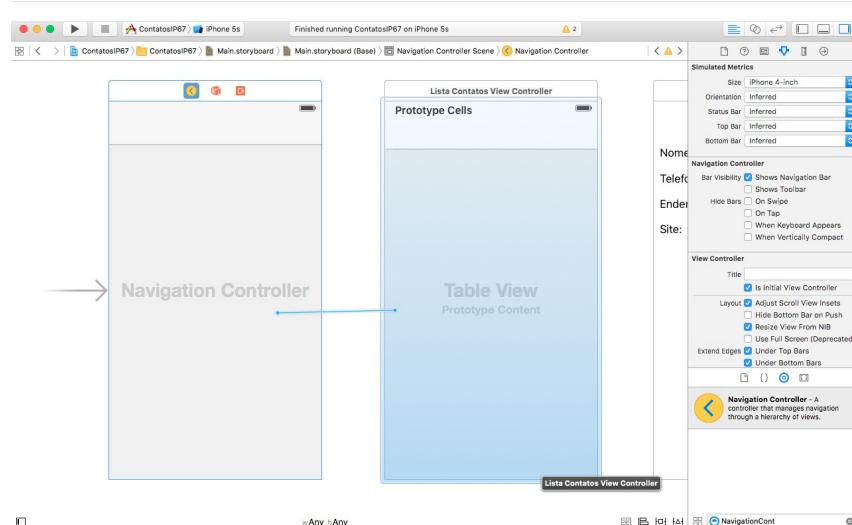
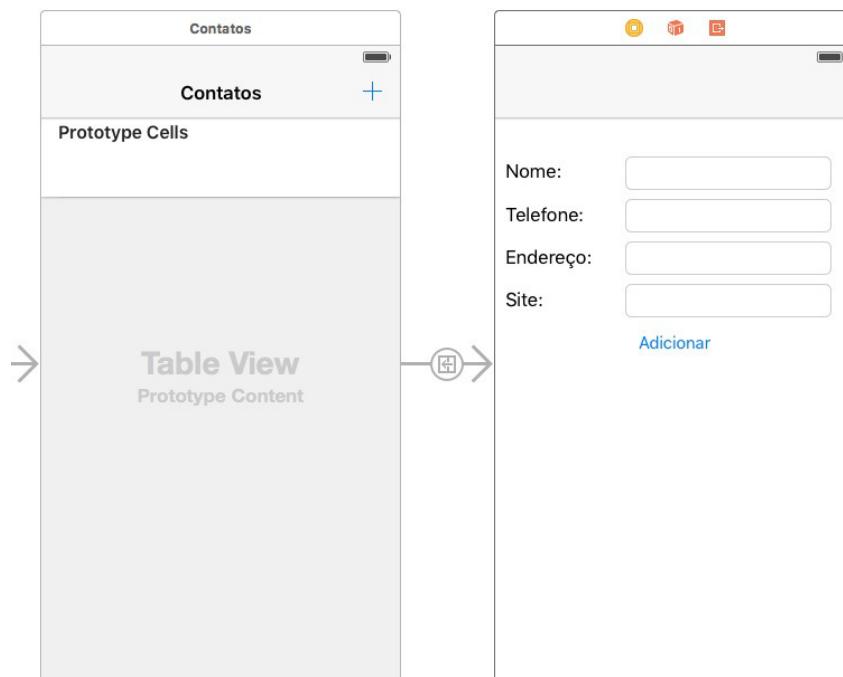


Figura 8.19: Relacionando NavigationController ao nosso TableViewController

3. Coloque um título e o botão chamado `Add` com o estilo *system style* na *navigation bar* da tela de `ListaContatosViewController`
4. Faça com que nosso *bar button item* exiba o formulário de cadastro de contatos.



5. Coloque um título na tela do formulário de cadastro

8.6 BOAS PRÁTICAS DE ORIENTAÇÃO A OBJETOS: DIVIDINDO

RESPONSABILIDADES

Vamos precisar que nosso botão adicione o contato na listagem. Para isso criaremos um método chamada `criaContato`, com a responsabilidade de armazenar um novo contato na listagem. Seu código poderia ser semelhante ao seguinte:

```
let contato: Contato = Contato()  
self.dao.adiciona(contato)
```

No entanto, para que o método `criaContato` tenha sentido, será preciso copiar para ela o código do método `pegaDadosDoFormulario`. Será? Vamos pensar um pouco nessa questão: qual deveria ser o objetivo do método `criaContato`? Não seria apenas adicionar uma determinada instância de `Contato` a uma lista? Por quê, então, adicionar mais responsabilidades a essa método? Podemos dividir essa responsabilidade, deixar que a mensagem `pegaDadosDoFormulario` faça essa parte do trabalho e mover para a mensagem `criaContato` tudo o que for relacionado com armazenar um `Contato`.

ORIENTAÇÃO A OBJETOS E DIVISÃO DE RESPONSABILIDADES

Essa estratégia de isolar responsabilidades diferentes é uma boa prática em Orientação a Objetos que, entre outras coisas, leva a um código mais fácil de manter e evoluir.

O foco desse curso não é específico em Orientação a Objetos, mas existem alguns debates bastante interessantes sobre o assunto que você pode conferir no blog da Caelum:

- Como não aprender Java e Orientação a Objetos: getters e setters
<http://blog.caelum.com.br/nao-aprender-oo-getters-e-setters>
- Compondo seu comportamento: herança, Chain of Responsibility e Interceptors
<http://blog.caelum.com.br/compondo-seu-comportamento-heranca-chain-of-responsibility-e-interceptors/>

Como poderia ser, então, a implementação desse método `criaContato`? Perceba que ela vai precisar se apoiar no método `pegaDadosDoFormulario` para obter o novo `Contato`. Poderíamos fazer a implementação da seguinte forma:

```
func criaContato() {  
    let contato: Contato = self.pegaDadosDoFormulario()  
    self.dao.adiciona(contato)  
}
```

Para esse código funcionar no projeto atual precisaremos fazer algumas alterações. Porém, note uma coisa importante que vamos ganhar com essa arquitetura: a nova mensagem `criaContato` nem sequer precisa saber que existe um formulário em nossa aplicação, não precisa entrar em contato com os elementos visuais que compõem esse formulário.

Isso significa que toda a lógica para buscar dados no formulário e transformar esses dados em um objeto do tipo `Contato` ficará dentro da mensagem `pegaDadosDoFormulario`. Se, no futuro, for preciso alterar o formulário, adicionar ou remover campos, essa alteração não vai interferir na mensagem `criaContato`. Podemos dizer que toda a lógica para transformar os dados do formulário em um objeto está **encapsulada** pelo método `pegaDadosDoFormulario`, que vai estabelecer um contrato: o de retornar um objeto do tipo `Contato`. Dessa forma a implementação desse método pode mudar a qualquer momento, desde que, no final, ela retorne um novo `Contato`.

Como usaremos o contato em mais do que um método, podemos ainda transformá-lo em uma propriedade e utilizá-lo em todos os métodos que precisamos.

Para conseguir esse isolamento de responsabilidades, vamos fazer as alterações necessárias em `pegaDadosDoFormulario`. É preciso criar a propriedade `contato` na classe e fazer com que o método popule esta propriedade, como a seguir:

```
class FormularioContatoViewController: UIViewController {  
    ...  
  
    var contato:Contato!  
  
    ...  
  
    func criaContato(){  
        self.pegaDadosDoFormulario()  
        dao.adiciona(contato)  
    }  
}
```

E dentro do método `pegaDadosDoFormulario`:

```
@IBAction func pegaDadosDoFormulario(){  
    self.contato = Contato()  
  
    self.contato.nome = self.nome.text!  
    // ... restante das atribuições  
}
```

Um último detalhe para que essa alteração fique completa: o método `pegaDadosDoFormulario` havia sido definida como um `IBAction`, para ser executada após um evento disparado pela `View`. Porém isso não é mais necessário, podemos dizer que o método agora é uma mensagem *comum*. Por isso, é preciso editar o arquivo `storyboard` e remover a referência entre o botão e o `IBAction`.

PARA SABER MAIS: REFERÊNCIAS QUEBRADAS EM UM ARQUIVO STORYBOARD OU XIB

Durante o processo de evolução de nosso aplicativo, surgiu a necessidade de remover um `IBAction`, o mesmo poderia ocorrer com um `IBOutlet`. Quando isso for necessário, lembre-se de editar o arquivo *storyboard* ou *xib* e remover a referência que sobrou por lá. Essa é uma fonte bastante comum de problemas quando estamos começando a desenvolver para a plataforma *iOS*.

A imagem abaixo ilustra que, mesmo não existindo mais a declaração de um determinado `IBAction` no arquivo `FormularioContatoViewController.h`, é preciso quebrar a conexão realizada no *storyboard* anteriormente.

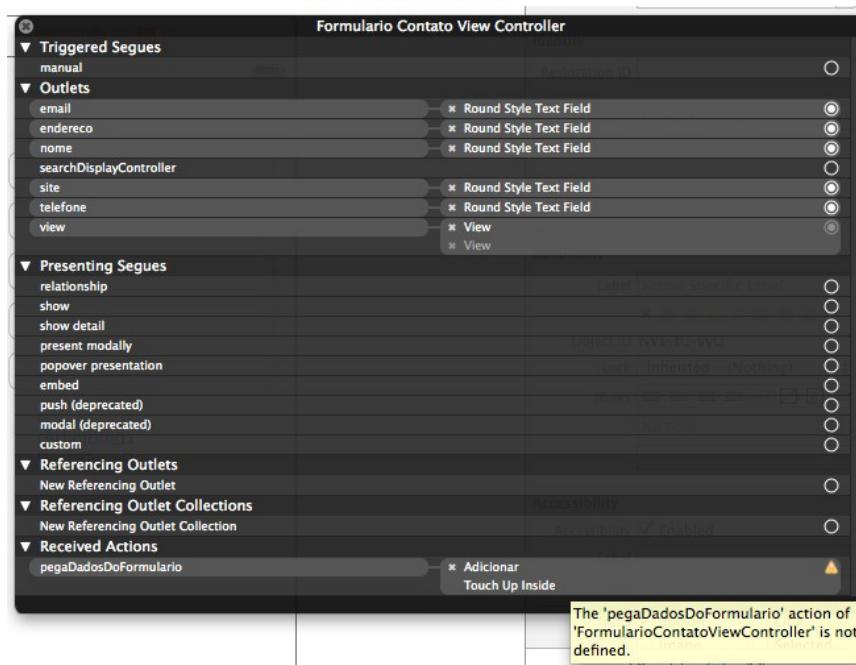


Figura 8.21: Referência precisa ser desfeita

Então, se vamos criar um botão específico na barra de navegação para adicionar um `Contato`, não precisamos mais do `UIButton`, sendo assim, podemos simplesmente excluir esse componente da interface, isso também vai excluir a referência para o `IBAction` feita por ele. Podemos remover também, a declaração `@IBAction` do método `pegaDadosDoFormulario`.

A essa altura, temos um método chamado `criaContato` totalmente funcional. Podemos, então, adicionar um novo `UIBarButtonItem` à barra de navegação e fazer com que ao clicar (efetuar um gesto de `tap`) nesse novo botão seja chamado o método `criaContato` que acabamos de criar. Para isso, vamos adicionar a declaração `@IBAction` na declaração do método `criaContato`.

```
@IBAction func criaContato(){
    self.pegaDadosDoFormulario()
```

```
        dao.adiciona(contato)
    }
```

Feito isso vamos adicionar o `UIBarButtonItem` e alterar o título para **Adiciona** e vincula-lo ao método `criaContato`.

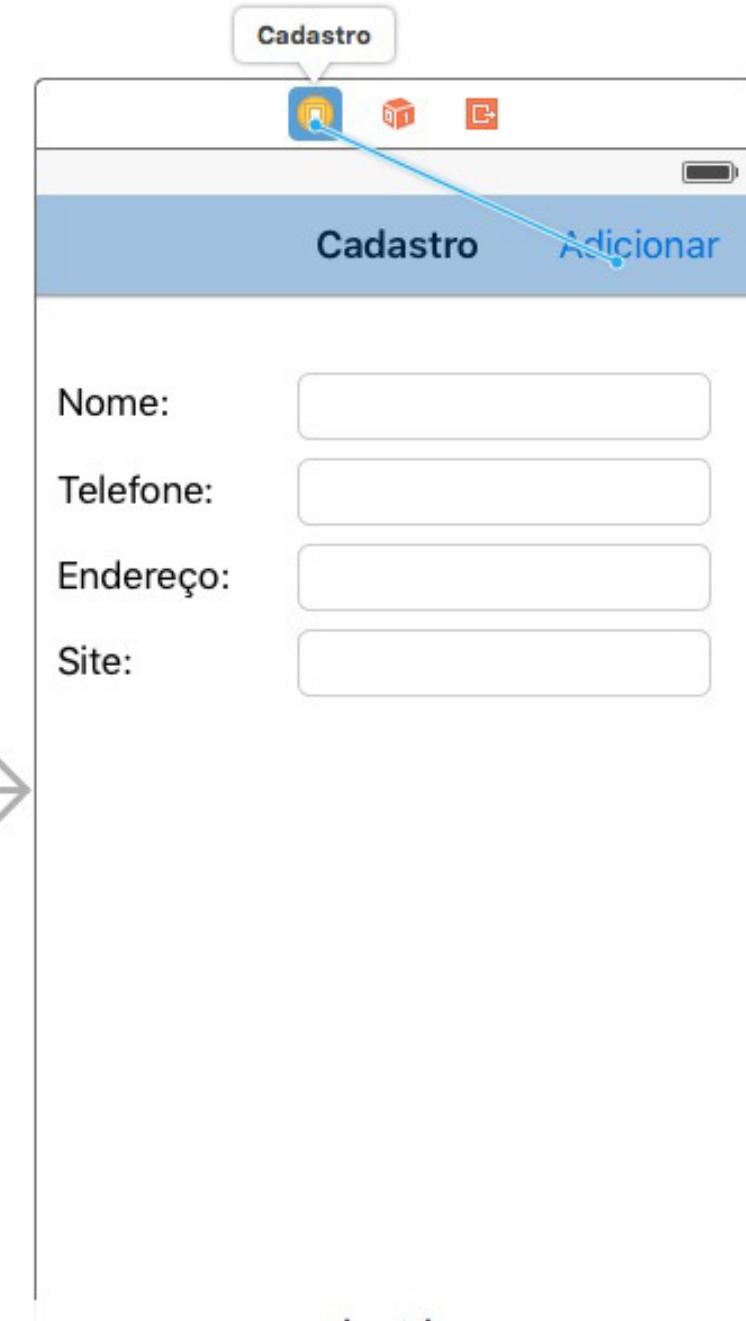


Figura 8.22: Referência precisa ser desfeita

Ao final dessas alterações, teremos um cadastro funcional! E a listagem que é a tela inicial da aplicação? O que faremos quando um `Contato` for cadastrado? Podemos voltar exibir a listagem novamente, certo? Como podemos fazer isso?

Vamos explorar o fato de que o formulário é exibido pelo *Navigation Controller*. (Devido ao fato de ter *herdado* a funcionalidade de navegação quando adicionamos a transição.) Podemos chamar o método `popViewControllerAnimated` para o `navigationController`, e ele se encarregará de voltar a navegação para a *View* anterior - no nosso caso, a listagem. O código pode ser o seguinte:

```
@IBAction func criaContato(){
    self.pegaDadosDoFormulario()
    dao.adiciona(contato)

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

Ao rodar a aplicação, você poderá clicar no botão + e ela terá a seguinte aparência:



Figura 8.23: Aparência atual da aplicação

Vamos fazer essas alterações. Mas, será que elas serão o suficiente?

8.7 EXERCÍCIO - USANDO UM UIBarButtonItem PARA ARMAZENAR UM CONTATO

1. Crie a propriedade `contato` no `FormularioContatoViewController.swift`:

```
var contato: Contato!
```

2. Crie o método `criaContato`, ele deve ser responsável apenas por armazenar uma instância de contato na lista. Implemente o método como a seguir:

```
func criaContato(){
    self.pegaDadosDoFormulario()
    dao.adiciona(contato)
}
```

3. Agora, podemos reaproveitar a lógica que atualmente está no método `pegaDadosDoFormulario`. Repare que a única função desse método é buscar os dados preenchidos em um formulário e popular uma instância de `Contato` com essa informação. É apenas isso que queremos que esse método faça.

Altere a declaração do método, pois não vamos mais usá-lo como um `IBAction`, e **remova** a chamada ao método `adiciona`:

```
@IBAction func pegaDadosDoFormulario() {
    // ... restante da implementação do método

    // não esqueça de apagar o adiciona aqui
}
```

Por fim, utilize a nova propriedade na implementação do método, para que o contato gerado possa ser usado por outros métodos nessa classe. O código final do método `pegaDadosDoFormulario` deverá ficar como o seguinte:

```
@IBAction func pegaDadosDoFormulario() {
    self.contato = Contato();

    self.contato.nome = self.nome.text!;
    self.contato.telefone = self.telefone.text!;
    self.contato.email = self.email.text!;
    self.contato.endereco = self.endereco.text!;
    self.contato.site = self.site.text!;
}
```

4. Agora que não precisamos mais de um `IBAction` para criar um contato, vamos remover a anotação `@IBAction` do método.

```
func pegaDadosDoFormulario() {
    //...
}
```

5. Removemos um `IBAction` do formulário, porém, ele ainda é referenciado pelo arquivo `Main.storyboard`, por meio do link com o botão de *Adicionar*. Como não precisamos mais desse botão, vamos simplesmente removê-lo. Edite o `storyboard`, selecione o botão e clique em *delete*. Feito isso, adicione um `Bar Button Item` no lado direito do título da barra de navegação.

Seu arquivo vai ficar parecido com o seguinte:

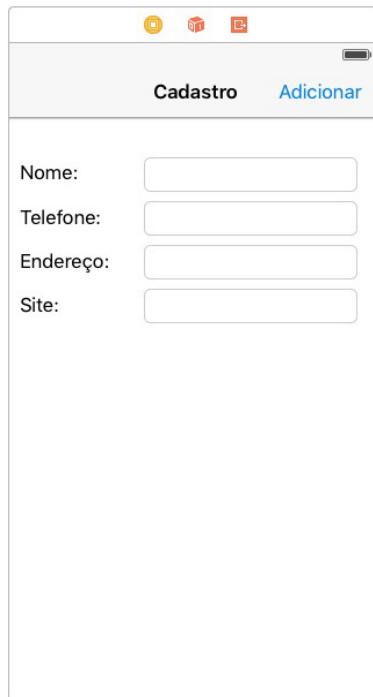


Figura 8.24: Tela de formulário com o botão Adicionar

6. Agora precisamos que quando o botão **Adicionar** receber um gesto de `Tap` seja executado o método `criaContato`. Para isso vamos adicionar a anotação `@IBAction` no método e vincular o botão com o método:

```
@IBAction func criaContato(){
    self.pegaDadosDoFormulario()
    dao.adiciona(contato)
}
```



Figura 8.25: Referência precisa ser desfeita

7. Se quiser rodar a aplicação agora, você verá o botão *Adiciona* na barra de navegação, e, cada vez que ele é clicado, também aparece uma mensagem no console do *Xcode* informando os dados dos contatos que estão atualmente na lista. A figura a seguir ilustra o comportamento atual do aplicativo:



Figura 8.26: Dados dos contatos cadastrados

8. Para finalizar, basta voltar a navegação para a lista com uma chamada ao `popViewControllerAnimated`. Essa chamada pode ser feita usando a propriedade `navigationController` do próprio formulário.

Faça isso no método `criaContato`, pois após inserir um novo `Contato` o usuário quer visualizar a lista atualizada.

```
@IBAction func criaContato(){  
    self.pegaDadosDoFormulario()  
    dao.adiciona(contato)  
    _ = self.navigationController?.popViewControllerAnimated(animated: true)  
}
```

Rode a aplicação, experimente cadastrar um novo `Contato`. Tudo está funcionando como deveria? O que realmente gostaríamos de ver após inserir um novo `Contato` é uma listagem populada com todos os contatos cadastrados e não uma lista vazia! É exatamente disso que vamos cuidar no próximo capítulo.

APRESENTANDO DADOS DA MESMA FORMA QUE AS APLICAÇÕES NATIVAS:UITABLEVIEW

O objetivo desse capítulo é mostrar o que é preciso para utilizar um `UITableViewController` para listar registros da mesma forma que as aplicações nativas do *iOS* fazem. Além do *view controller*, que já tem algumas funcionalidades prontas, também vamos ver como lidar com o componente visual que representa uma tabela, o `UITableView`.

Também vamos passar os objetos como referência, usar uma característica da linguagem, para facilitar o desenvolvimento da aplicação. Ao final do capítulo teremos uma aplicação que permite cadastrar e listar os contatos existentes.

9.1 UITABLEVIEWCONTROLLER: UM VIEW CONTROLLER PARA LISTAR REGISTROS

Para que cada célula da listagem apresente as informações de um contato precisaremos sobrescrever alguns métodos na classe `ListaContatosViewController`.

Os três métodos básicos necessários são responsáveis por informar quantas linhas e seções terá a tabela e qual a lógica para criar a visualização de cada uma dessas linhas. Esses métodos são `-tableView:numberOfRowsInSection` , `tableView:cellForRowAt` e o `numberOfSections` .

Esses métodos são definidos na classe `UITableViewController` , mas a implementação padrão não fornece o comportamento que desejamos, o que faz sentido. Podemos pensar nessa estratégia como um ponto de extensão para que o programador possa customizar o comportamento padrão de um componente visual já existente no *iOS*. Todo objeto do tipo `UITableViewController` tem uma referência para um objeto do tipo `UITableView` que é o componente visual para o qual o *view controller* dá suporte. Esse *table view* vai invocar os métodos de nosso *view controller* para buscar a informação necessária para se apresentar corretamente na tela.

Para conseguirmos visualizar os contatos, podemos obter a lista da classe `ContatoDao` . Vamos criar a propriedade `dao` :

```
import UIKit
```

```
class ListaContatosViewController: UITableViewController {  
    var dao: ContatoDao  
}
```

E em seguida buscar uma instância dentro do método `init`:

```
import UIKit  
  
class ListaContatosViewController: UITableViewController {  
    var dao: ContatoDao  
  
    required init?(coder aDecoder: NSCoder) {  
        self.dao = ContatoDao.sharedInstance()  
        super.init(coder: aDecoder)  
    }  
}
```

O método `tableView:numberOfRowsInSection:` informa a quantidade total de linhas na tabela - isso é importante para o cálculo do "scroll" que será feito pelo próprio componente visual `UITableView`. Em nossa aplicação este número será baseado no total de contatos cadastrados.

```
override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {  
    return self.dao.contatos.count  
}
```

O que é, exatamente, esse parâmetro `section` que apareceu em nosso código? O componente `UITableView` que estamos usando para listar os contatos tem a capacidade de dividir os registros em regiões, dando o nome de *section* para cada uma delas. Cada *section* pode ter um título e seus próprios registros.

Você pode implementar uma tabela exatamente como a da aplicação *Contacts* ilustrada na figura abaixo usando as *sections*.

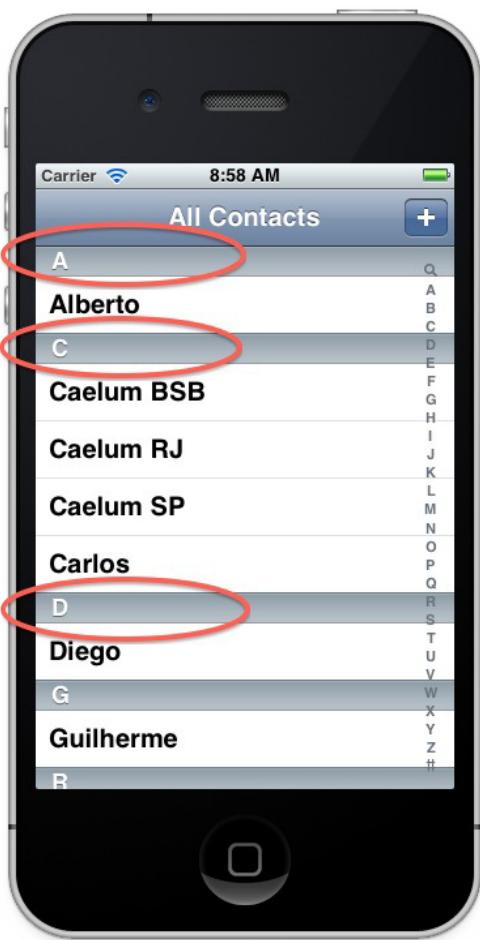


Figura 9.1: Total de contatos cadastrados

Para isso, você precisa checar o índice da *section* que foi passado como parâmetro na invocação do método `tableView:numberOfRowsInSection:`. Esse índice começa em 0 (zero) e fica a cargo do programa (e claro, do desenvolvedor) calcular quantos registros serão exibidos em qual índice. É esse o valor que será retornado pelo método `tableView:numberOfRowsInSection:`.

Como em nossa aplicação vamos utilizar apenas uma *section*, não precisaremos nos preocupar em calcular o total de registros para diferentes *sections*. E é por isso que será tão simples implementar o método `numberOfSections:`, que é o método que retorna o número de *sections* presentes em uma tabela. Como não precisamos nos preocupar com cálculos, vamos apenas informar ao iOS que nossa tabela tem apenas uma *section* retornando o valor **1**:

```
override func numberOfSections(in tableView: UITableView) -> Int {  
    return 1  
}
```

ESTE MÉTODO NÃO É NECESSÁRIO

O método `numberOfSections:` é opcional, por padrão a implementação da uma `UITableViewController` já retorna `1`, foi implementado apenas para entender o conceito de um `IndexPath`.

Pronto! Já implementamos o código que informa quantas linhas serão exibidas na tabela, agora é preciso fornecer a informação que será exibida em cada uma dessas linhas ou células. Para isso, vamos utilizar um objeto que representa uma linha de uma `UITableView`. Esse objeto é instância da classe `UITableViewCell`.

Dentro do iOS, temos alguns layouts de células prontos para utilizarmos, cada um com suas características próprias. Em nossa aplicação vamos utilizar um estilo que também é usado em outros aplicativos do iOS chamado `UITableViewCellStyleDefault`. Porém, existem alguns outros que você pode usar. Cada um deles vai variar em forma e maneira de apresentar o conteúdo na tabela. Veja, a seguir, uma lista de alguns estilos possíveis:

- `UITableViewCellStyleDefault`
- `UITableViewCellStyleValue1`
- `UITableViewCellStyleValue2`
- `UITableViewCellStyleSubtitle`



Figura 9.2: Estilos das células

O estilo que escolhemos contém um label alinhado à esquerda e uma imagem que é opcional. Temos acesso a esse label a partir do atributo `textLabel` de um `UITextViewCell`.

MAIS SOBRE OS LAYOUTS PARA `UITABLEVIEWCELL`

Para saber mais sobre os layout providos pelo iOS, faça uma busca dentro do **Window**, na opção **Documentation and API Reference** por `UITableViewCellStyle`.

Vamos sobrescrever o método `tableView:cellForRowAt:` na classe `ListaContatosViewController`. Esse é o método responsável por retornar uma instância de `UITableViewCell` para cada linha de uma tabela. Esse método recebe como parâmetro em `cellForRowAt:` um objeto do tipo `IndexPath`, que tem duas propriedades: `row` e `section`.

A ideia desse método é retornar uma instância de `UITableViewCell` com a informação que deverá ser exibida em determinada linha de uma determinada *section*.

Lembre-se de que nossa tabela terá uma única *section*, portanto, podemos ignorar a informação na propriedade *section* do parâmetro `cellForRowAt:`. O que nos interessa é a propriedade `row` que é um índice a partir de 0 (zero). Podemos, simplesmente, fazer um mapeamento desse índice para o registro na lista que queremos exibir na tabela. Qual é o contato da linha zero? É o contato retornado pela invocação `self.dao.contatos[0]`. Logo, para recuperar o contato que será exibido em determinada linha, podemos implementar a seguinte lógica:

```
let contato:Contato = self.dao.contatos[indexPath.row];
```

Porém manipular a lista de contatos é uma responsabilidade do `ContatoDao`! Vamos então criar um método que, dado um número, retorne o contato daquela posição.

Na classe `ContatoDao` vamos criar:

```
func buscaContatoNaPosicao(_ posicao:Int) -> Contato {
    return contatos[posicao]
}
```

Podemos voltar a classe da lista e fazer:

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let contato:Contato = self.dao.buscaContatoNaPosicao(indexPath.row)
}
```

Já temos a informação que gostaríamos de exibir, agora é preciso apresentar esses dados dentro de um `UITableViewCell`, para isso, vamos instanciar o objeto com o construtor `initWithStyle:reuseIdentifier:`. Já sabemos do que se trata o *style* que aparece nesse construtor,

vamos descobrir o papel de `reuseIdentifier:` em breve.

```
let cell:UITableViewCell = UITableViewCell(style: .Default, reuseIdentifier: nil)
```

Com isso, já podemos criar uma implementação inicial para o método `tableView:cellForRowAtIndexPath:`. Basta configurar o label do objeto `cell` com o nome do contato relativo àquela célula na tabela, veja:

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let contato:Contato = self.dao.buscaContatoNaPosicao(indexPath.row)
    let cell = UITableViewCell(style: .default, reuseIdentifier: nil)
    cell.textLabel?.text = contato.nome
    return cell
}
```

Ótimo, mas falta entender o papel do parâmetro `reuseIdentifier:` que surgiu no construtor do `UITableViewCell`. Qual será exatamente a função dessa `String`?

Esse valor está diretamente relacionado ao gerenciamento e economia de memória na construção de uma tabela. Esse parâmetro serve para ter apenas o número necessário de `UITableViewCell` para preencher visualmente a tabela para o usuário, mesmo que a quantidade de dados seja superior a esse número.

Conforme o usuário faz o *scroll* na tabela para revelar registros que estão invisíveis, podemos reciclar as células que não estão mais visíveis, configurar com a informação que precisa aparecer na linha que será exibida e, dessa forma, reutilizamos sempre um determinado número de células. As imagens abaixo ilustram como isso funciona:

- Primeiramente, uma célula é preparada para ser reciclada de acordo com a direção do *scroll* realizado pelo usuário.



Figura 9.3: Célula reciclável disponível

- A seguir, é preciso configurar a célula reciclada com a informação que será exibida



Figura 9.4: Célula é preparada com novos dados

- E, finalmente, o próprio UITableView exibe a célula reciclada.



Figura 9.5: Célula é reposicionada

Para tirar proveito dessa funcionalidade é preciso identificar a instância de `UITableViewCell` que será passível de reciclagem. É para isso que serve o parâmetro `reuseIdentifier`. Vamos usar uma *string* que será usada pela própria tabela para nos retornar uma instância de `UITableViewCell` pronta para reciclagem. Como não precisamos instanciar uma nova *string* para cada célula, já que o que importa é o valor dessa *string*, vamos declarar uma instância de `String` com o modificador `static`. Isso fará com que o objeto `String` armazenado nessa variável seja criado uma única vez e armazenado na memória.

Vamos alterar o código que instancia um `UITableViewCell` para passar essa *string* como parâmetro no construtor:

```
class ListaContatosViewController: UITableViewController {

    static let cellIdentifier:String = "Cell"
    ...

    override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        let contato:Contato = self.dao.buscaContatoNaPosicao(indexPath.row)

        let cell:UITableViewCell = UITableViewCell(style: .default, reuseIdentifier: ListaContatosViewController.cellIdentifier)

        cell.textLabel?.text = contato.nome

        return cell
    }
}
```

No entanto, dessa forma ainda estamos instanciando células para cada um dos registros na tabela, precisamos fazer algum tipo de verificação: só queremos instanciar um novo objeto se não há nenhum

disponível para reciclagem. Parte desse trabalho já é feito pelo componente `UITableView`. O método `dequeueReusableCellWithIdentifier` vai retornar a primeira instância de `UITableViewCell` pronta para reciclagem que ele encontrar. Se não houver nenhuma, ou em outras palavras, se ainda não há instâncias de `UITableViewCell` suficientes para a exibição de todos os registros que cabem na tela, então esse método retornará `nil`.

Portanto, vamos verificar o retorno do método `dequeueReusableCellWithIdentifier` e se nenhum objeto for retornado, vamos instanciar uma nova célula:

```
var cell:UITableViewCell? = tableView.dequeueReusableCell(withIdentifier: ListaContatosViewController.cellIdentifier)

if cell == nil {
    cell = UITableViewCell(style: .default, reuseIdentifier: ListaContatosViewController.cellIdentifier)
}
```

E isso é tudo o que precisamos para reciclar as células e um `tableView`, isso pode trazer um grande ganho de performance se a quantidade de registros a serem exibidos for grande.

Veja como ficará o código final do método `tableView:cellForRowAt::`:

```
override func tableView(tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let contato:Contato = self.dao.buscaContatoNaPosicao(indexPath.row)

    var cell:UITableViewCell? = tableView.dequeueReusableCell(withIdentifier: ListaContatosViewController.cellIdentifier)

    if cell == nil {
        cell = UITableViewCell(style: .default, reuseIdentifier: ListaContatosViewController.cellIdentifier)
    }

    cell!.textLabel?.text = contato.nome

    return cell!
}
```

Pronto! Isso é tudo o que faltava para listar os contatos em nossa tabela. Temos o cadastro a listagem interligados e funcionando. Os exercícios a seguir são o passo a passo de tudo o que discutimos nesse capítulo, preparado?

9.2 EXERCÍCIO - LISTANDO CONTATOS

1. Como a aplicação usará apenas uma `section` para listar todos os contatos, o método mais simples de implementação será o `numberOfSections`. Sobrescreva esse método na classe `ListaContatosViewController` retornando o valor 1. Isso vai indicar que essa lista tem apenas uma `section`:

```
override func numberOfSections(tableView: UITableView) -> Int {
    return 1
}
```

2. Também é preciso sobrescrever o método `tableView:numberOfRowsInSection:` utilizado pela instância de `UITableView` para descobrir o número total de linhas que serão exibidas em uma determinada *section*. Podemos buscar a lista do `ContatoDao` para obter essa informação.

- Na classe `ListaContatosViewController` crie um atributo para o dao

```
import UIKit

class ListaContatosViewController: UITableViewController {

    var dao:ContatoDao

}
```

- Ainda no `ListaContatosViewController`, obtenha uma instância de `ContatoDao` dentro do método `init`:

```
required init?(coder aDecoder: NSCoder) {
    self.dao = ContatoDao.ContatoDaoInstance()
    super.init(coder: aDecoder)
}
```

- Implemente o método retornando o resultado da invocação do método `count` em `self.contatos`:

```
override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    return dao.listaTodos().count
}
```

3. Precisamos criar o método para listar os contatos e a partir dele pegar a quantidade total, dentro da classe `ContatoDao`:

```
func listaTodos() -> [Contato] {
    return contatos
}
```

4. Agora é preciso sobrescrever o método `tableView:cellForRowAt:`, que terá a responsabilidade de configurar os dados que serão exibidos em cada linha da tabela.

Além disso, a implementação desse método precisa prever a questão do gerenciamento de memória. Uma estratégia comum é criar somente um número de células suficiente para exibir na tela em vez de uma célula para cada registro.

Os objetos do tipo `UITableView` já se encaixam nessa arquitetura fornecendo um método chamado `dequeueReusableCellWithIdentifier:` que vai retornar o primeiro objeto do tipo `UITableViewCell` com um identificador igual ao passado como parâmetro e pronto para reciclagem. Caso não exista nenhum objeto nessas condições, o método vai retornar `nil`.

Baseado nessas informações sobrescreva o método `tableView:cellForRowAt:`. O identificador

que vamos usar para as células será a string `Cell` :

```
override func tableView(tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell
Cell {
    var cell:UITableViewCell? = tableView.dequeueReusableCell(withIdentifier: ListaContatosViewController.cellIdentifier)

    if cell == nil {
        //obter uma nova instancia
    }

    return cell!
}
```

5. A seguir, precisamos criar um novo objeto do tipo `UITableViewCell` sempre que não houver uma célula pronta para reciclagem. As células de nossa tabela vão usar o estilo `UITableViewCellStyleDefault` e o identificador que já declaramos no método `tableView:cellForRowAt:`.

Edite o método `tableView:cellForRowAt:` e, no bloco do `if` criado para checar se há uma célula que pode ser reciclada, instancie uma célula e atribua à variável `cell` :

```
override func tableView(tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell
{
    var cell:UITableViewCell? = tableView.dequeueReusableCell(withIdentifier: ListaContatosViewController.cellIdentifier)

    if cell == nil {
        cell = UITableViewCell(style: .default, reuseIdentifier: ListaContatosViewController.cellIdentifier)
    }

    return cell!
}
```

6. Para configurar a célula com o nome do contato, basta recuperar o objeto que desejamos mostrar naquela linha no *array* de contatos. Os índices das linhas começam em 0, portanto podemos usar esse índice para recuperar as instâncias do *array*. Para isso, vamos criar o método que devolverá um contato de uma determinada posição, dentro da classe `ContatoDao` :

```
func buscaContatoNaPosicao(_ posicao:Int) -> Contato {
    return contatos[posicao]
}
```

7. Agora, já é possível recuperar uma instância de `Contato` e armazená-la em uma variável.

```
override func tableView(tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell
{
    let contato:Contato = self.dao.buscaContatoNaPosicao(indexPath.row)

    var cell:UITableViewCell? = tableView.dequeueReusableCell(withIdentifier: ListaContatosViewController.cellIdentifier)

    if cell == nil {
```

```

        cell = UITableViewCell(style: .default, reuseIdentifier: ListaContatosViewController.cellIdentifier)
    }

    return cell!
}

```

O próximo passo é usar essa instância para configurar o conteúdo da propriedade `cell.textLabel.text`. Vamos atribuir a *string* do nome do contato nessa propriedade para que ele apareça na lista. Após isso, atribua o nome do contato a propriedade `cell.textLabel.text`:

```

override func tableView(tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let contato: Contato = self.dao.buscaContatoNaPosicao(indexPath.row)

    var cell: UITableViewCell? = tableView.dequeueReusableCell(withIdentifier: ListaContatosViewController.cellIdentifier)

    if cell == nil {
        cell = UITableViewCell(style: .default, reuseIdentifier: ListaContatosViewController.cellIdentifier)
    }

    cell!.textLabel?.text = contato.nome

    return cell!
}

```

8. Se rodarmos agora, veremos que os dados não serão atualizados após a inserção. Para que a atualização ocorra, precisaremos recarregar a tabela toda vez que a tela de listagem for exibida. Então, no método `viewWillAppear:`, chamaremos o método `reloadData`:

```

override func viewDidAppear(_ animated: Bool) {
    self.tableView.reloadData()
}

```

Rode novamente o aplicativo e agora tudo deve funcionar como o esperado. Nossa aplicação já permite o cadastro e listagem de novos contatos e tudo de forma integrada!

USANDO O MODO DE EDIÇÃO NATIVO DO UITABLEVIEW

Precisamos implementar a funcionalidade que vai permitir excluir um contato da listagem. O componente `UITableView` tem um "modo edição" que facilitará muito a nossa vida! Para ativar esse modo em uma tabela, será preciso alterar o valor da propriedade `editing`.

Na classe `ListaContatosViewController` já existe uma propriedade `tableView` para referenciar a tabela onde são listados os contatos. Para ativar o modo edição nessa tabela podemos usar o código `self.tableView.editing = true`. Quando queremos que a tabela se torne "editável"? Precisaremos de uma ação do usuário para isso, certo? Portanto, podemos implementar essa funcionalidade como resposta a um tap em algum botão. Uma ideia seria criar, um novo `UIBarButtonItem` para editar os contatos.

10.1 PRECISA EDITAR UMA VIEW? USE O PADRÃO DO IOS!

A ideia de editar os dados apresentados por uma view em determinado ponto da navegação numa aplicação é algo muito comum em aplicações iOS. Tanto que, para facilitar a implementação dessa funcionalidade, existe uma propriedade do tipo `UIBarButtonItem` declarada na classe `UIViewController`. Como todo controller herda dessa classe, é possível usar a referência ao botão em qualquer controller por meio da propriedade chamada `editButtonItem`. Pelo nome da propriedade fica claro que a ideia é utilizar esse botão para edição, inclusive o label do botão já vem configurado como **Edit**.

Porém, o que acontece quando alguém dá um tap nesse botão? O método `setEditing:animated` é invocado (perceba também que ele está implementado na classe `UIViewController`). Classes que herdam de `UIViewController` podem sobrepor esse método para indicar qual será o comportamento de uma view ou componente visual quando entrar em modo de edição.

Nossa classe `ListaContatosViewController` herda de `UITableViewController` (que por sua vez herda de `UIViewController`). Note que esse controller foi criado especificamente para gerenciar e tratar os eventos gerados pelo componente `UITableView`. A ótima notícia é que a implementação do método `setEditing:animated` já foi feita na classe `UITableViewController`, só é preciso mostrar o botão na tela! Quando ele receber um tap, a tabela referenciada em nosso controller entrará em modo de

edição.

Lembre que o botão disponível em todos os controllers é do tipo `UIBarButtonItem`, isso quer dizer que ele só pode ser usado na view de `UINavigationController`. Portanto, vamos adicioná-lo no canto esquerdo em nossa barra de navegação exibida com a lista. Faremos isso no método `init` no arquivo `ListaContatosViewController`:

```
self.navigationItem.leftBarButtonItem = self.editButtonItem;
```

Neste momento, ao clicar no botão de edição teremos o seguinte comportamento:

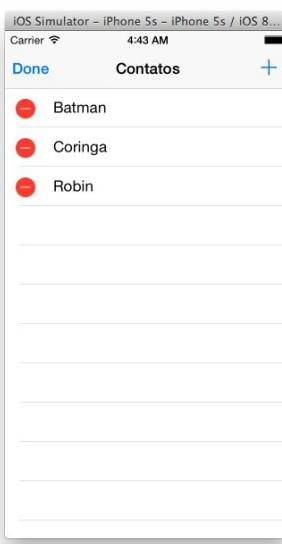


Figura 10.1: Tabela em edição

10.2 A FONTE DE DADOS DE UMA TABELA: O UITABLEVIEWDATASOURCE

Porém, ao clicar no ícone de remover, nada acontece; precisamos implementar um método para excluir o registro da tabela. Lembra quando implementamos um protocolo para garantir a existência de determinado método? Pois é exatamente assim que a classe `UITableViewcontroller` (da qual a nossa `ListaContatosViewController` herda) foi criada, em sua declaração está especificado que ela implementa `UITableViewDataSource`.

Uma tabela precisa de uma fonte de dados de onde serão extraídas as informações listadas. Em nossa aplicação, essa fonte é o array contido na propriedade `contatos` que atualmente possui uma lista de objetos do tipo `Contato`. Note que antes ele armazenava instâncias de `Dictionary`, e a tabela conseguia listar os dados da mesma forma, mas como isso é possível?

A fonte de dados de uma tabela é qualquer objeto que implemente o protocolo

`UITableViewDataSource` , um dos métodos que esse protocolo define é o chamado `tableView:cellForRowAt:` , lembra-se dele? Sobrescrevemos esse método em nossa classe `ListaContatosViewController` . Como a tabela sabe que é exatamente esse método que deve ser chamado para obter as informações de uma célula em determinada linha? A classe `UITableView` define uma propriedade chamada `dataSource` do tipo `UITableViewDataSource` . A classe `UITableView` ao inicializar a tabela também registra a si própria como o `dataSource` da tabela.

10.3 EXCLUÍNDO UM REGISTRO DA TABELA

O papel do `dataSource` é ser a fonte de dados para a tabela, sendo assim, para excluir um registro, é preciso remover esse registro do `dataSource` . Um dos métodos opcionais declarados no protocolo `UITableViewDataSource` é o chamado `tableView:commit:forRowAt:` . Ele recebe a instância da tabela, a opção de edição que foi selecionada pelo usuário (como excluir, por exemplo) e qual a linha que disparou o evento. Esse último parâmetro vai permitir saber exatamente qual registro deverá ser excluído.

O primeiro passo para excluir o registro é saber se essa foi a opção selecionada pelo usuário. Para isso, temos que investigar o valor do parâmetro recebido em `editingStyle`:

```
override func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
    if editingStyle == .delete {
        // lógica para excluir o registro da tabela
    }
}
```

Perfeito! Agora precisamos descobrir exatamente a linha em que estava o registro que o usuário pediu para excluir. Essa informação está disponível no parâmetro passado em `forRowAt:` , que é um objeto do tipo `IndexPath` . Objetos desse tipo têm uma propriedade chamada `row` , que contém o valor que precisamos. Objetos do tipo array possuem um método `remove(at:)` que exclui o item no índice passado como parâmetro. Como as linhas da tabela são indexadas da mesma forma que um array (primeiro item na posição zero) podemos usar o número da linha diretamente na remoção de um item do array. Vamos implementar o método que efetuará a remoção, na classe `ContatoDao` . Não se esqueça de colocar a declaração do método no arquivo ".h":

```
func remove(_ posicao:Int){
    contatos.remove(at:posicao)
}
```

Agora vamos utilizar o dao para removermos um contato:

```
override func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
    if editingStyle == .delete {
        self.dao.remove(indexPath.row)
        // lógica para excluir o registro da tabela
    }
}
```

```
}
```

Agora, é preciso avisar à tabela que os valores fornecidos pelo `dataSource` foram alterados, para que a lista exibida na tela seja atualizada. Vamos usar o método `reloadData` da classe `UITableView` para isso.

```
override func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
    if editingStyle == .delete {
        self.dao.remove(indexPath.row)
        tableView.reloadData()
    }
}
```

Pronto! Agora já é possível remover registros. Rode a aplicação para testar essa funcionalidade. Funciona, mas dá para melhorar a usabilidade da aplicação. Uma forma de facilitar para o usuário enxergar que o registro foi realmente removido (ou ainda que o registro removido foi exatamente o que ele selecionou) é usar uma animação ao remover a linha da tabela. Você ficaria surpreso se houvesse um método que já remova a linha com uma animação bem elegante? O método `deleteRows(at:with:)` cuidará disso.

Esse método recebe dois parâmetros: o primeiro é um array de objetos do tipo `IndexPath` e cada elemento nesse array representa uma linha que será removida. Já temos um objeto desse tipo, que é o parâmetro passado para o método `tableView:commit:forRowAt:..`. O segundo parâmetro é o tipo de animação que será usado para a remoção da linha, vamos usar `fade` em nossa aplicação. Nossa código deverá ficar assim:

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

        tableView.deleteRows(at: [indexPath], with: .fade)
    }
}
```

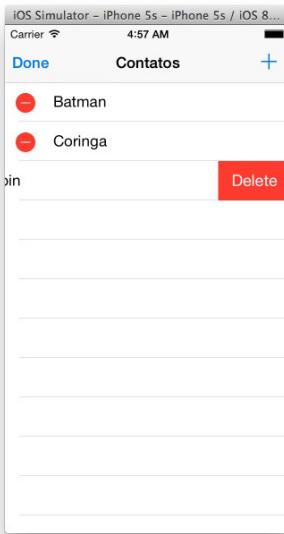


Figura 10.2: Deletando uma linha

OUTROS TIPOS DE ANIMAÇÃO PARA A EDIÇÃO DA TABELA

Além do `fade` existem vários outros tipos de animação que você pode utilizar. Para saber quais modelos de animação estão disponíveis, procure por `UITableViewRowAnimation` na documentação.

10.4 EXERCÍCIO - REMOVENDO UM CONTATO

1. Precisamos fornecer o mecanismo para colocar uma tabela em modo de edição. Para isso, vamos tornar visível o botão armazenado na propriedade `editButtonItem` de nossa instância de `UITableViewController`.

Na classe `ListaContatosViewController`, edite o método `init` e atribua a propriedade `self.navigationItem.leftBarButtonItem` o botão de editar nativo da lista:

```
required init?(coder aDecoder: NSCoder) {
    self.dao = ContatoDao.ContatoDaoInstance()
    self.navigationItem.leftBarButtonItem = self.editButtonItem
    super.init(coder: aDecoder)
}
```

2. Agora é preciso sobrescrever o método que trata os eventos de um `UITableView` em modo de edição.

Adicione a sobreescrita do método abaixo ao arquivo `ListaContatosViewController`:

```
override func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
    // ... código para excluir um registro
}
```

3. É necessário ter o método que efetuará a remoção do registro. Implementaremos na classe `ContatoDao`:

```
func remove(_ posicao:Int){
    contatos.remove(at:posicao)
}
```

4. Em seguida, é preciso verificar se a ação de edição disparada pelo usuário foi excluir. Se foi, basta retirar o contato selecionado do `array` de contatos.

Edite o método `tableView:commit:forRowAt::` para adicionar essa verificação:

```
override func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
    if editingStyle == .delete {
        self.dao.remove(indexPath.row)
    }
}
```

5. E, finalmente, para que o usuário tenha uma experiência mais agradável, vamos usar uma animação para indicar, com maior clareza, que determinada linha está desaparecendo da tabela.

Para isso, edite novamente o método responsável pelas ações disparadas na tabela em modo de edição e, dessa vez, adicione a invocação do método `deleteRows(at:with:)` na instância de `UITableView`. A implementação final na classe `ListaContatosViewController` ficará como a seguir:

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

        tableView.deleteRowsAtIndexPaths([indexPath], withRowAnimation: .Fade)
    }
}
```

Rode a aplicação e verifique se tudo está funcionando normalmente. Você consegue excluir registros da tabela? Ótimo. E que tal editar os dados de um contato? Vamos cuidar disso no próximo capítulo.

SELECIONANDO REGISTROS EM UM UITABLEVIEW: EDITANDO OS DADOS DE UM CONTATO

Até o momento, não há uma forma de visualizar os dados de um contato. Seguindo os conceitos de interação com o usuário estabelecidos pelo componente `UITableView`, poderíamos tornar a tabela clicável. Ao clicar em uma linha da tabela, o usuário é levado para uma tela onde pode visualizar e, se for o caso, editar os dados de determinado contato.

11.1 SELECIONANDO UM REGISTRO DA TABELA

O primeiro passo será permitir que a tabela seja clicada e fazer a aplicação reagir de alguma forma a esse clique, ou melhor dizendo, a esse *tap*. Quando uma linha da tabela for selecionada com um *tap*, o próprio `UITableView` vai disparar uma chamada ao método `tableView:didSelectRowAtIndexPath` de seu *view controller*.

O parâmetro passado a `didSelectRowAtIndexPath` é um objeto do tipo `IndexPath`. Esse objeto tem duas propriedades: `row` e `section`. Como nossa tabela usa uma única `section`, temos que nos preocupar somente com o índice da linha que foi clicada, justamente o valor da propriedade `row`. Com base nesse índice podemos recuperar a instância de `Contato` relacionada aquela linha e guardá-la como uma propriedade da lista, veja:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {  
    let contato: Contato = dao.buscaContatoNaPosicao(indexPath.row)  
  
    print("Nome: \(contato.nome)")  
}
```

Depois de descobrir o contato selecionado pelo usuário, podemos partir para a exibição dos dados.

11.2 EXERCÍCIO - RECUPERANDO REGISTRO SELECIONADO EM UM UITABLEVIEW

1. Crie a propriedade para armazenar a referência para o contato selecionado:

```
import UIKit
```

```

class ListaContatosViewController: UITableViewController {

    static let cellIdentifier:String = "Cell"
    var dao:ContatoDao

}

```

2. Queremos capturar o evento de tap na tabela e, de acordo com a linha selecionada pelo usuário, descobrir qual o Contato associado àquela linha.

Sobrescreva o método `tableView:didSelectRowAtIndexPath` na classe `ListaContatosViewController` e recupere o objeto `Contato` referente à linha selecionada. Por hora, apenas use a função `print` para checar se o nome do `Contato` recuperado está correto, logo a seguir vamos criar a funcionalidade de edição.

```

override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
    let contatoSelecionado = dao.buscaContatoNaPosicao(indexPath.row)

    print("Nome: \(contatoSelecionado.nome)")
}

```

Confira na imagem abaixo o resultado esperado após o exercício:

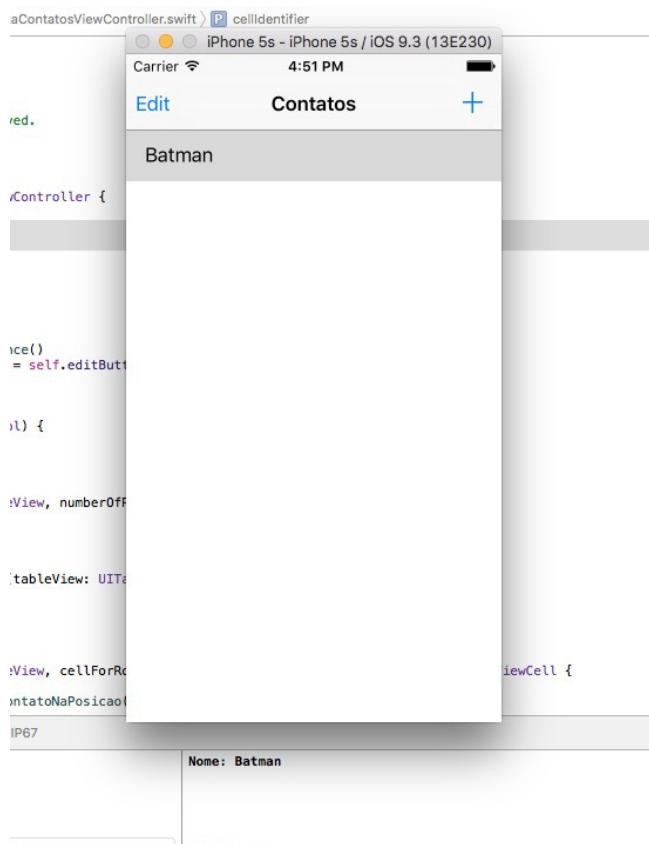


Figura 11.1: Linha selecionada em um UITableView

11.3 EXIBINDO UM FORMULÁRIO JÁ PREENCHIDO

Como vamos usar a mesma tela para apresentar os dados e também editá-los, podemos usar a tela de formulário já existente para isso. Quando o usuário selecionar um contato na lista, vamos exibir o formulário preenchido com os dados daquele contato. Dessa forma, o usuário pode, além de visualizar, editar os dados de qualquer contato.

Para fazer isso precisamos de alguma forma identificar nosso formulário, para conseguir chama-lo de forma programática. Para isso vamos adicionar um identificador no nosso formulário diretamente na nossa storyboard :

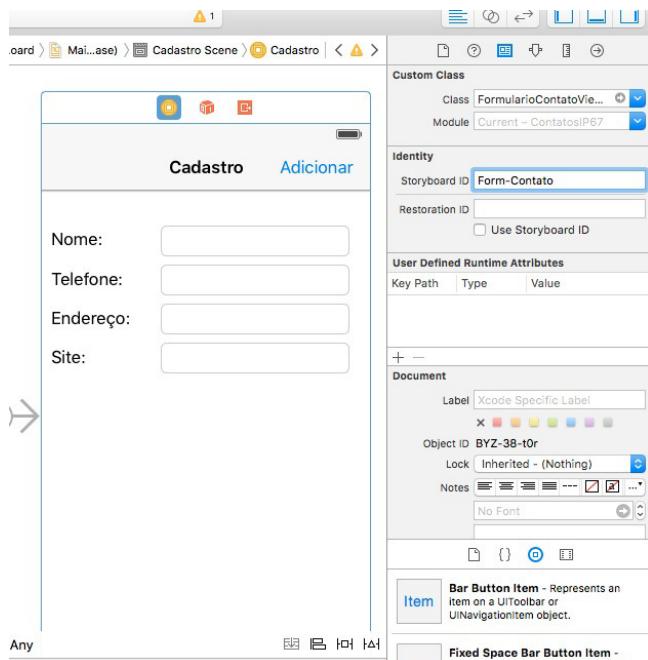


Figura 11.2: Adicionando identificador ao formulário

Agora vamos alterar a implementação do método `tableView:didSelectRowAtIndexPath` por uma lógica que exiba o formulário, setando o contato que queremos visualizar ou editar. Basta substituir o atual `print` nesse método pelo código a seguir:

```
let storyboard: UIStoryboard = UIStoryboard(name: "Main", bundle: nil)

let formulario = storyboard.instantiateViewController(withIdentifier: "Form-Contato") as! FormularioContatoViewController

self.navigationController?.pushViewController(formulario, animated: true)
```

Mas faltou definir o contato que será exibido no formulário para isso altere o código para:

```
let storyboard: UIStoryboard = UIStoryboard(name: "Main", bundle: nil)

let formulario = storyboard.instantiateViewController(withIdentifier: "Form-Contato") as! FormularioContatoViewController

formulario.contato = contatoSelecionado

self.navigationController?.pushViewController(formulario, animated: true)
```

Como essa lógica de exibir o formulário envolve pegar storyboard instanciar o formulário e definir o contato que será exibido no formulário e depois efetuar a navegação. Vamos separar as responsabilidades, vamos criar um método chamado `exibeFormulario` que faz toda essa lógica para gente:

```
func exibeFormulario(_ contato:Contato) {
    let storyboard: UIStoryboard = UIStoryboard(name: "Main", bundle: nil)

    let formulario = storyBoard.instantiateViewController(withIdentifier: "Form-Contato") as! FormularioContatoViewController

    formulario.contato = contato

    self.navigationController?.pushViewController(formulario, animated: true)
}
```

Com isso a declaração do método `tableView:didSelectRowAtIndexPath` fica somente com a responsabilidade de pegar o contato selecionado e chamar o método `exibeFormulario`:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {

    let contatoSelecionado = dao.buscaContatoNaPosicao(indexPath.row)

    self.exibeFormulario(contatoSelecionado)

}
```

Perfeito, precisamos escolher um bom momento para preencher o formulário. Pode parecer que o construtor é um bom lugar para isso, o problema é que no momento em que o construtor é chamado, os componentes visuais referenciados pelos `outlets` podem não estar disponíveis ainda.

Uma melhor maneira de alterar o estado inicial de componentes visuais é sobrescrevendo o método `viewDidLoad` que é invocado em todo *view controller* quando os elementos visuais já estão todos em memória, mas ainda não apareceram na tela. Se, no momento da invocação desse método, houver algum contato armazenado em `self.contato` saberemos que o formulário está em modo edição e poderemos preencher os campos com os dados desse contato. Na classe `FormularioContatoViewController` podemos então acrescentar este método.

```
override func viewDidLoad() {
    super.viewDidLoad()

    if contato != nil {
        self.nome.text = contato.nome
        self.telefone.text = contato.telefone
        self.endereco.text = contato.endereco
        self.site.text = contato.site
    }
}
```

Ótimo, vamos implementar a funcionalidade de edição:

11.4 EXERCÍCIOS - PASSANDO UM CONTATO PARA O FORMULÁRIO

1. Altere o método `viewDidLoad`, que é invocado assim que uma nova tela associada a um `viewController` estiver pronta para ser exibida, na classe `FormularioContatoViewController`.

Faça uma verificação e, se houver uma instância de `Contato` atribuída à propriedade `contato`, use os dados desse objeto para preencher o formulário.

```
override func viewDidLoad() {
    super.viewDidLoad()

    if contato != nil {
        self.nome.text = contato.nome
        self.telefone.text = contato.telefone
        self.endereco.text = contato.endereco
        self.site.text = contato.site
    }
}
```

2. Para que possamos passar o contato para o formulário, crie o método `exibeFormulario` para chamar o `setter` de contato do form:

```
func exibeFormulario(_ contato:Contato) {
    let storyboard: UIStoryboard = UIStoryboard(name: "Main", bundle: nil)

    let formulario = storyboard.instantiateViewController(withIdentifier: "Form-Contato") as!
    FormularioContatoViewController

    formulario.contato = contato

    self.navigationController?.pushViewController(formulario, animated: true)
}
```

1. Edite a classe `ListaContatosViewController` e altere a implementação do método `tableView:didSelectRowAtIndexPath`. Essa implementação usará o método `exibeFormulario`, que devemos alterar anteriormente para passar o contato selecionado para a classe `FormularioContatoViewController`. Ao final, o método deverá estar como o seguinte:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {

    let contatoSelecionado = dao.buscaContatoNaPosicao(indexPath.row)

    self.exibeFormulario(contatoSelecionado)

}
```

Rode a aplicação. Agora você já pode ver o formulário preenchido com os dados de um contato quando clica sobre uma linha na tabela. Ótimo, mas o que acontece quando você clica em *Adicionar*? Bem, se olharmos friamente para nosso programa, podemos dizer que acontece o que já era previsto: um novo contato é criado.

Gostaríamos de alterar a funcionalidade e o título desse botão, pois, quando estivermos editando um contato, queremos apenas alterar os dados que foram recuperados da lista.

11.5 ESCOLHENDO A AÇÃO DE UM BOTÃO NO FORMULÁRIO

E se o usuário alterar os dados e quiser gravar as alterações? Precisamos fornecer um botão específico para isso. Já sabemos que a propriedade `contato` só estará preenchida quando o formulário for usado no modo de edição. Podemos, então, assumir que, sempre que esta propriedade estiver setada, ao invés de exibir o botão de `Adicionar` deve exibir um botão na barra de navegação que será responsável por atualizar (e não criar) um contato.

Levando em consideração que esse botão precisa ser dinâmico, como definir a ação do toque do botão?

Swift uma linguagem dinâmica - buscando métodos em tempo de execução

Ao criarmos um `UIBarButtonItem` precisamos dizer o que vai acontecer quando o usuário clicar no elemento em si. Vamos instruir o iOS para invocar um método chamado `atualizaContato` (que vamos implementar mais adiante). Para isso, vamos usar um seletor.

Em Swift é possível representar um método por meio da marcação `#selector`. Essa funcionalidade é usada quando precisamos invocar um método, mas só é possível descobrir seu nome em tempo de execução. Um seletor é utilizado para realizar uma busca ao método de um objeto pelo nome utilizado na criação do seletor. Selectors são case sensitive. Se o método receber um ou mais parâmetros, precisamos especificá-los no nome do método. O primeiro parâmetro usamos `_` os demais são nomeados e separados por `:`, sem espaços. Suponha que tenhamos um método com a seguinte assinatura: `func atualizaContato(contato:Contato, nome:String, sobrenome:String)` teríamos o seguinte seletor `#selector(atualizaContato(_:nome:sobrenome:))`. Podemos armazenar o retorno da construção de um `#selector` em uma variável do tipo `Selector`: `let nomeDoMetodo:Selector = #selector(atualizaContato)`. Além de dizer que o tap (clique) no botão deverá invocar o método `atualizaContato`, também precisamos dizer em qual objeto esse método será invocado. Podemos, inclusive, implementar esse método no próprio controlador e instruir o iOS a invocar o método no objeto atual. A classe `UIBarButtonItem` declara um construtor que recebe um seletor e o objeto onde o método especificado deverá ser executado (além de uma string para ser usada como título do botão). Já que o método para exibir o formulário será declarado no próprio controller da listagem, vamos usar `self` como o objeto destino (`target`) para a invocação do método.

Portanto, vamos editar o método `viewDidLoad` e alterar dinamicamente o botão na barra de navegação que vai invocar o método chamado `atualizaContato`.

```
override func viewDidLoad() {
    super.viewDidLoad()

    if contato != nil {
        self.nome.text = contato.nome
        self.telefone.text = contato.telefone
        self.endereco.text = contato.endereco
        self.site.text = contato.site
```

```

        let botaoAlterar = UIBarButtonItem(title: "Confirmar", style: .plain, target: self, action: #selector(atualizar))

        self.navigationItem.rightBarButtonItem = botaoAlterar

    }

}

...

func atualizaContato(){
    pegaDadosDoFormulario()

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

```

O método `atualizaContato` precisa obter os dados do formulário e atualizar a instância de contato na lista de contatos e, por fim, esconder novamente o formulário.

Antes de implementar esse método, vamos relembrar alguns fatos: em *Swift* os objetos são passados como referência. Significa que, se um objeto for alterado dentro de um método, essa alteração se reflete no objeto original, criado fora do método, que foi passado como parâmetro. Na verdade o objeto passado como parâmetro é o mesmo que existe fora do método.

Dito isso, podemos dar uma revisada no seguinte trecho de código:

```

override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
    let contatoSelecionado = dao.buscaContatoNaPosicao(indexPath.row)

    self.exibeFormulario(contatoSelecionado)
}

```

Perceba que o `Contato` passado como parâmetro para o formulário foi obtido da lista, que é a mesma lista criada ou recuperada pela classe `ContatoDao`. Significa que, se fizermos alguma alteração nesse objeto `contato` dentro do formulário, na verdade vamos causar uma alteração no objeto armazenado na lista. Perfeito, é exatamente o que precisamos para editar um contato. Basta armazenar os dados do formulário no objeto `contato` existente e pronto! O contato foi editado.

Para isso, faremos uma alteração no método `pegaDadosDoFormulario`. Se o `Contato` for passado como parâmetro, não queremos criar um novo objeto, apenas atualizar os dados do objeto existente. Porém, se nenhum contato foi passado, significa que o formulário está no "modo criação" de contato, e temos que instanciar um novo objeto do tipo `Contato` para armazenar na lista.

```

func pegaDadosDoFormulario(){
    if contato == nil {
        self.contato = Contato()
    }

    self.contato.nome = self.nome.text!
    self.contato.telefone = self.telefone.text!
    self.contato.endereco = self.endereco.text!
}

```

```

    self.contato.site = self.site.text!
}


```

Com isso, temos as funcionalidades de criação e edição implementadas na classe `FormularioContatoViewController`.

11.6 EXERCÍCIOS - BOTÃO ESPECÍFICO PARA A AÇÃO DE EDITAR

Por mais estranho que possa parecer, a lógica que já temos no momento é mais que suficiente para implementarmos a edição de um contato. Basta algumas pequenas alterações.

- Atualize o método `pegaDadosDoFormulario`. Ele só deverá criar uma nova instância de `Contato` caso a ação atual não seja de edição. Como saber disso? Verificando se há ou não um `Contato` instanciado na propriedade `contato`.

Após as alterações, o método ficará como a seguir:

```

func pegaDadosDoFormulario(){
    if contato == nil {
        self.contato = Contato()
    }

    self.contato.nome = self.nome.text!
    self.contato.telefone = self.telefone.text!
    self.contato.endereco = self.endereco.text!
    self.contato.site = self.site.text!

}

```

Isso fará com que o `Contato` armazenando na propriedade `contato` seja uma nova instância, em caso de criação, ou a instância de contato já existente na lista, em caso de edição.

- O próximo passo é implementar um método que vai atribuir os dados atuais do formulário à instância de contato já presente na lista e depois esconder o formulário.

Para isso, crie um método chamado `atualizaContato` com a seguinte implementação:

```

func atualizaContato(){
    pegaDadosDoFormulario()

    _ = self.navigationController?.popViewControllerAnimated(true)

}

```

- Edita o método `viewDidLoad` para quando verificarmos que possuímos uma instância de `Contato`, seja substituído o botão de *Adicionar* por uma instância do tipo `UIBarButtonItem` com o título *Confirmar* à barra de navegação. Um clique nesse botão deve causar a execução do método `atualizaContato`. Além disso vamos fazer que caso tenha uma instância de `Contato` vamos popular o nosso formulário com o contato existente:

```
override func viewDidLoad() {
```

```
super.viewDidLoad()

if contato != nil {
    self.nome.text = contato.nome
    self.telefone.text = contato.telefone
    self.endereco.text = contato.endereco
    self.site.text = contato.site

    let botaoAlterar = UIBarButtonItem(title: "Confirmar", style: .plain, target: self, action: #selector(atualizar))

    self.navigationItem.rightBarButtonItem = botaoAlterar
}

}
```

Rode a aplicação e agora você deve ser capaz de visualizar e editar os dados de um contato. Todas as outras implementações devem continuar funcionando.

ENTENDENDO DELEGATION E IMPLEMENTANDO O PADRÃO COM PROTOCOLOS

Em toda a plataforma *iOS* é usado um padrão arquitetural conhecido como *delegation*. A ideia desse padrão é permitir que um objeto *delegue* para outro mais especializado a responsabilidade por cuidar de alguma tarefa específica.

Por exemplo, após editar ou adicionar um `Contato` em um formulário, queremos destacar esse registro de alguma forma na tabela, para que o usuário tenha algum *feedback* visual relativo a sua ação. Será que faz sentido o formulário interferir na forma como o componente `UITableView` se apresenta, para fazer esse destaque na tela?

Teríamos que passar uma referência do *table view* para dentro do formulário - como já fazemos atualmente com o `DAO` e a instância de contato. Além disso, seria preciso levar para dentro do formulário o conhecimento sobre como uma `UITableView` é desenhado na tela. Claramente, essa é uma responsabilidade da lista.

O mecanismo do *delegate* vai facilitar essa implementação, criando um meio de comunicação entre o formulário e a lista; dessa forma, o formulário poderá simplesmente informar à lista que um contato foi criado ou alterado. Poderá, inclusive, dizer qual foi o contato. E a lista, por sua vez, poderá reagir de acordo.

12.1 DELEGATE: UM PADRÃO USADO POR TODO O IOS

Um *delegate* dentro do *iOS* nada mais é que um objeto que responde para determinado método. No nosso exemplo, poderíamos estabelecer que um *delegate* para o formulário é qualquer objeto que responda para os métodos `contatoAdicionado:` e `contatoAtualizado:`. Qual é o mecanismo para isso? Em *Swift*, como podemos criar uma classe que obedece a um contrato? Se você pensou em protocolo, acertou em cheio.

Vamos declarar o protocolo `FormularioContatoViewControllerDelegate`.

Para criar um protocolo, basta criar um arquivo (`.swift`) com a diretiva `protocol` seguida do nome do protocolo. Dentro do corpo da diretiva `protocol` podemos definir os métodos que precisam

ser implementados por qualquer classe que se comprometa com esse protocolo, veja:

```
import Foundation

protocol FormularioContatoViewControllerDelegation {
    func contatoAdicionado(_ contato: Contato)
    func contatoAtualizado(_ contato: Contato)
}
```

DEFININDO MÉTODOS OPCIONAIS

Em um protocolo, além de definir métodos **requeridos**, como fizemos anteriormente, podemos definir métodos que são opcionais, dessa forma quem seguir o contrato pode, ou não, implementar o método descrito.

```
@objc protocol FormularioContatoViewControllerDelegate {
    func contatoAdicionado(_ contato: Contato)
    func contatoAtualizado(_ contato: Contato)

    optional func metodoOpcional()
}
```

Por padrão, todos os métodos definidos são obrigatórios, por isso não colocamos anteriormente.

Perfeito, mas isso não resolve a questão do `delegate`. Precisamos de um mecanismo para dizer que o formulário aceita um objeto que implemente o protocolo `FormularioContatoViewControllerDelegate`. Porém, qual será o tipo desse objeto? Pois essa é a sacada, não importa o tipo, desde que esse objeto implemente o protocolo desejado. Quem usa um `delegate` está interessado apenas nos métodos que aquele objeto fornece de acordo com determinado protocolo.

Para ficar mais claro, veja como ficará a declaração da propriedade `delegate` do formulário:

```
var delegate:FormularioContatoViewControllerDelegation?
```

Perfeito, agora podemos assumir que, se um `delegate` estiver presente em um formulário, podemos invocar tanto o método `contatoAdicionado`: quanto `contatoAtualizado`: nesse `delegate`. Sendo assim, podemos atualizar a implementação dos métodos `criaContato` e `atualizaContato` do formulário para ficarem como a seguir:

```
func criaContato() {
    // ...
    self.delegate?.contatoAdicionado(novoContato);
}
```

```

func atualizaContato() {
    // ...
    self.delegate?.contatoAtualizado(contatoAtualizado);
}

```

Ótimo, mas quem será o delegate do formulário? Já sabemos que é qualquer objeto, não importa o tipo, que implemente um determinado protocolo. No entanto, ainda não sabemos quem é esse objeto. Precisamos implementar esse protocolo em alguma classe.

E claro, pelo nome do protocolo já temos uma boa pista de qual classe será essa.

```
class ListagemTableViewController: UITableViewController, FormularioContatoViewControllerDelegation
```

Após garantir que a classe `ListaContatosViewController` implementa o protocolo `FormularioContatoViewControllerDelegate`, será necessário implementar os métodos definidos nesse protocolo. Vamos criar uma implementação bem simples por enquanto, apenas para garantir que tudo está funcionando. Vamos recuperar o índice na lista em que um contato está armazenado.

Como os métodos do `delegate` recebem um contato como parâmetro, podemos exibir o nome do contato no log:

```

func contatoAtualizado(_ contato:Contato) {
    print("contato atualizado: \(contato.nome)");
}

func contatoAdicionado(_ contato:Contato) {
    print("contato adicionado: \(contato.nome)");
}

```

Para que esses métodos sejam invocados na instância de `ListaContatosViewController` que exibiu o formulário na tela, será preciso indicar esse objeto como delegate do formulário. Perceba que precisamos indicar a própria instância de lista como delegate do formulário que é criado dentro da lista. Um `delegate` de formulário nada mais é que uma propriedade. Ou seja, após instanciar um objeto do tipo `FormularioContatoViewController`, basta atribuir `self` ao `delegate` desse novo objeto formulário:

```

func exibeFormulario(_ contato:Contato) {
    let storyboard: UIStoryboard = UIStoryboard(name: "Main", bundle: nil)

    let formulario = storyboard.instantiateViewController(withIdentifier: "Form-Contato") as! FormularioContatoViewController

    formulario.contato = contato
    formulario.delegate = self
    self.navigationController?.pushViewController(formulario, animated: true)
}

```

Mas... E quando adicionarmos um contato? Como nos registramos como sendo `delegate` na inclusão? Como nossa alteração está sendo feita de forma programática, foi fácil nos registrar como sendo o `delegate` do nosso formulário.

Porém nossa inclusão fizemos via `interface builder` através de `segue`, como faremos para registrar nesse caso? A resposta é simples **usando um método específico para preparar nosso segue**. Toda classe que herde de `UIViewController` pode chamar o método `prepare(for segue:sender)`. Esse método é invocado no momento em que a transição entre as duas telas ligadas pelo `segue` acontece.

Ou seja antes da transição do `segue` ocorrer, esse método é invocado. Com esse método temos acesso ao objeto `UIStoryboardSegue` que representa nosso `segue` no `interface builder`. Através dele temos acesso a diversas informações e uma delas é a instância da tela de destino (no nosso caso o formulário).

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {  
  
    if let formulario = segue.destination as? FormularioContatoViewController {  
        formulario.delegate = self  
    }  
  
}
```

Porém nossa tela pode ter mais de um `segue`, e precisamos nos registrar somente no `segue` que liga ao formulário. Para isso vamos adicionar um identificador ao `segue` no `storyboard`. Basta selecionar o `segue` no `storyboard` e adicionar um `Identifier` na aba `Attribute Inspector`.

Feito isso, programáticamente podemos verificar se o `segue` que está sendo executado tem o identificador que colocamos via `interface builder`, em caso positivo registramos o `delegate`:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {  
    if segue.identifier == "FormSegue" {  
  
        if let formulario = segue.destination as? FormularioContatoViewController {  
            formulario.delegate = self  
        }  
  
    }  
  
}
```

DELEGATION

Em nosso exemplo, usando a propriedade `delegate`, armazenamos no controller do formulário uma referência para o controller da listagem. Posteriormente, invocamos um comportamento na listagem para que ela reaja à criação de um `Contato`. Esse conceito de delegar para outros objetos parte do comportamento de acordo com eventos que estão ocorrendo no sistema é muito presente nos frameworks nativos do iOS e do próprio Mac OS X.

É o padrão observer sendo aplicado. Vemos bastante isso no Java e no C# por meio das interfaces.

Você pode ler mais a respeito disso na documentação da Apple:
<https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/CocoaFundamentals/CommunicatingWithObjects/CommunicateWithObjects.html>

Por que recuperar o índice do `array` na implementação inicial que fizemos dos métodos do protocolo? Vamos descobrir em seguida, mas antes, que tal um pouco de exercícios para esquentar os motores?

12.2 EXERCÍCIO - CRIANDO UM DELEGATE

1. Declare um protocolo chamado `FormularioContatoViewControllerDelegate`. O protocolo terá dois métodos declarados: `contatoAtualizado:` e `contatoAdicionado:`. Ambos recebem um `Contato` como parâmetro.

Veja como ficará o código do novo protocolo:

```
import Foundation

protocol FormularioContatoViewControllerDelegate {
    func contatoAtualizado(_ contato:Contato)
    func contatoAdicionado(_ contato:Contato)
}
```

2. Faça com que a classe `ListaContatosViewController` implemente o protocolo `FormularioContatoViewControllerDelegate`. Para isso, edite a declaração da classe no cabeçalho para que o código fique como o seguinte:

```
import UIKit

class ListaContatosViewController: UITableViewController, FormularioContatoViewControllerDelegate
```

Após isso, o `Xcode` vai alertar que a implementação da classe não está completa, pois faltam alguns

métodos definidos no protocolo, vamos fazer isso!

3. Implemente os métodos definidos no protocolo. Como temos acesso ao objeto do tipo `Contato` usado no formulário, será possível descobrir qual o índice desse objeto no *array* de contatos por meio do método `indexof`:

Primeiramente, implemente o método que poderá ser invocado para informar que um novo contato foi adicionado:

```
func contatoAtualizado(_ contato:Contato) {  
    print("contato atualizado: \(contato.nome)");  
}
```

4. Implemente, também, o método que pode ser chamado quando um contato é criado.

```
func contatoAdicionado(_ contato:Contato) {  
    print("contato adicionado: \(contato.nome)");  
}
```

5. Agora, edite o arquivo `FormularioContatoViewController.swift` para criar uma propriedade chamada `delegate` que fará referência a um objeto que implemente o protocolo `FormularioContatoViewControllerDelegate`.

```
var delegate:FormularioContatoViewControllerDelegation?
```

6. Edite o método que adiciona um contato para verificar se há um `delegate` disponível e, em caso positivo, invocar o método `contatoAdicionado`: nesse `delegate`:

```
@IBAction func criaContato(){  
    self.pegaDadosDoFormulario()  
    dao.adicionaContato(contato)  
  
    self.delegate?.contatoAdicionado(contato)  
  
    _ = self.navigationController?.popViewController(animated: true)  
}
```

7. Implemente o método de atualização, com a mesma lógica do método que acabamos de implementar:

```
func atualizaContato(){  
    pegaDadosDoFormulario()  
  
    self.delegate?.contatoAtualizado(contato)  
  
    _ = self.navigationController?.popViewController(animated: true)  
}
```

8. Agora podemos configurar o `delegate` do formulário quando ele é instanciado na classe `ListaContatosViewController`. Faremos isso atualizando o método `exibeFormulario` adicionando a seguinte linha de código logo após instanciar o formulário:

```
formulario.delegate = self;
```

Após as edições, o método `exibeFormulario` deverá ficar como a seguir:

```
func exibeFormulario(_ contato:Contato) {
    let storyboard: UIStoryboard = UIStoryboard(name: "Main", bundle: nil)

    let formulario = storyBoard.instantiateViewController(withIdentifier: "Form-Contato") as! FormularioContatoViewController

    formulario.delegate = self
    formulario.contato = contato

    self.navigationController?.pushViewController(formulario, animated: true)
}
```

9. Devemos lembrar de nos registramos também na inclusão de um contato, quando acessamos nosso formulário via `segue`. Adicione um identificador para o `segue` com o nome de **FormSegue**. Sobrescreva o método `prepare(for segue:sender)` e registre o delegate apenas se o `segue` tiver o identificador **FormSegue**:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if segue.identifier == "FormSegue" {

        if let formulario = segue.destination as? FormularioContatoViewController {
            formulario.delegate = self
        }
    }
}
```

Ao rodar a aplicação, será possível editar e criar contatos normalmente, porém, a única novidade é uma mensagem no *log*. Parece que não é muito, mas fizemos muitas coisas.

Além de criar um novo protocolo e fazer com que a classe `ListaContatosViewController` o implemente, também usamos o padrão `delegate` e a compreensão desse padrão facilitará entender muita coisa sobre o funcionamento nativo das bibliotecas nativas do iOS.

A seguir, vamos usar essa infraestrutura para dar um destaque à linha que foi atualizada no formulário, isso vai melhorar a usabilidade do nosso aplicativo.

12.3 DESTACANDO UMA LINHA NA TABELA VIA SELEÇÃO

Parte da motivação em criar um `delegate` no formulário era conseguir destacar na lista o contato recém criado ou alterado no formulário, vamos resolver isso.

Toda *table view* tem um método chamado `selectRowAtIndexPath:animated:scrollPosition::` que permite selecionar uma linha programaticamente. O parâmetro `animated`, bastante conhecido, recebe um `Bool` que indica se a alteração na tela deverá ser animada ou não. Já o `scrollPosition` permite fazer o *scroll* da tabela até a posição selecionada. Podemos usar constantes para definir onde essa linha selecionada ficará posicionada após o *scroll*: no fim da tela, no topo ou no centro. Usaremos a terceira

opção passando `.middle`.

O parâmetro que ficou sem explicação é o `selectRow(at: IndexPath)`, o tipo de objeto a ser passado aqui é `IndexPath`, um objeto com duas informações: `row` e `section`. Para saber a linha que precisamos selecionar, é preciso saber o índice da tabela, que é exatamente o mesmo índice de um contato no `array` de contatos! É por isso que nos preocupamos em descobrir o índice de um objeto na tabela.

Vamos criar um método dentro da classe `ContatoDao` que retornará o índice do `Contato` selecionado.

```
func buscaPosicaoDoContato(contato: Contato) -> Int {  
    return contatos.index(of: contato)!  
}
```

Faremos uso disso agora. Veja, como exemplo, a alteração que poderemos fazer no método `contatoAtualizado:`. Primeiramente, será necessário instanciar um `IndexPath`. Depois disso, podemos invocar o método que faz a seleção na tabela:

```
func contatoAtualizado(_ contato: Contato) {  
    let posicao = dao.buscaPosicaoDoContato(contato)  
  
    let indexPath = IndexPath(row: posicao, inSection: 0)  
  
    self.tableView.selectRow(at: indexPath, animated: true, scrollPosition: .middle)  
}
```

Só há um problema aqui. Quando o método do `delegate` (no caso do exemplo, o `contatoAtualizado:`) é invocado, pode ser que a tabela nem sequer esteja carregada na memória, ainda. Podemos sobrescrever o método `viewDidAppear:` que é invocado imediatamente após o componente visual estar visível na tela. Parece um ótimo lugar para movermos a nossa lógica de seleção. Porém, será preciso armazenar o índice da linha que queremos destacar ou o `IndexPath` da mesma. Então vamos criar uma propriedade:

```
var linhaDestaque: IndexPath?
```

Agora vamos atualizar os métodos do `delegate`:

```
func contatoAdicionado(_ contato: Contato) {  
    self.linhaDestaque = IndexPath(row: dao.buscaPosicaoDoContato(contato), inSection: 0)  
}  
  
func contatoAtualizado(_ contato: Contato) {  
    self.linhaDestaque = IndexPath(row: dao.buscaPosicaoDoContato(contato), inSection: 0)  
}
```

E, por fim, sobrescrever o método `viewDidAppear:`

```
override func viewDidAppear(animated: Bool) {  
    self.tableView.selectRow(at: self.linhaDestaque!, animated: true, scrollPosition: .middle)  
}
```

E, após isso, a tabela vai "rolar" até a posição indicada pelo parâmetro `indexPath`. Que tal, parece

interessante? Então vamos ver isso na prática, vamos lá!

12.4 EXERCÍCIO - SELECIONANDO UMA LINHA NA TABELA PROGRAMATICAMENTE

1. Na classe `ContatoDao`, vamos declarar o método que busca o contato de uma determinada posição:

```
func buscaPosicaoDoContato(_ contato: Contato) -> Int {  
    return contatos.index(of: contato)!  
}
```

2. Crie uma propriedade para armazenar a última linha manipulada pelo formulário, seja para edição ou para criação de um contato. No arquivo `ListaContatosViewController` adicione a seguinte linha:

```
var linhaDestaque: IndexPath?
```

3. Implemente novamente os métodos do protocolo para armazenar o índice da linha que precisa ser destacada. Eles ficarão como a seguir:

```
func contatoAdicionado(_ contato: Contato) {  
    self.linhaDestaque = IndexPath(row: dao.buscaPosicaoDoContato(contato), inSection: 0)  
}  
  
func contatoAtualizado(_ contato: Contato) {  
    self.linhaDestaque = IndexPath(row: dao.buscaPosicaoDoContato(contato), inSection: 0)  
}
```

4. Vamos sobrescrever um método que é invocado em todo *view controller* quando o componente visual associado a ele terminou de ser exibido. Vamos implementar uma lógica no método `viewDidAppear`: que usa o último índice que precisa ser destacado na tabela. Com esse índice, criaremos um objeto do tipo `IndexPath` para o método que permite selecionar uma linha na tabela programaticamente.

Veja como vai ficar o código:

```
override func viewDidAppear(animated: Bool) {  
    self.tableView.selectRow(at: self.linhaDestacada!, animated: true, scrollPosition: .middle)  
}
```

Ao Rode a aplicação perceba que deu um erro assim que o aplicativo apareceu. Vamos resolver isso!

5. Precisamos verificar se a propriedade `linhaDestaque` tem ou não valor. Pois com base nesse valor vamos decidir se há ou não alguma linha que precisa de destaque. Para isso podemos usar a instrução `if let`:

```
override func viewDidAppear(animated: Bool) {  
    if let linha = self.linhaDestacada {  
  
        self.tableView.selectRow(at: linha, animated: true, scrollPosition: .middle)  
    }  
}
```

6. Após marcarmos a linha precisamos resetar o atributo `linhaDestaque` para que na proxima transição ele não selecione novamente a linha. Para isso podemos atribuir `nil` para `linhaDestaque`. Ou usar uma forma um pouco mais elegante com `Optional.none` ou simplesmente `.none` :

```
override func viewDidAppear(animated: Bool) {
    if let linha = self.linhaDestacada {

        self.tableView.selectRow(at: linha, animated: true, scrollPosition: .middle)
        self.linhaDestacada = Optional.none
    }
}
```

Perceba que, com o código acima, caso alguma linha seja destacada a tabela fará o *scroll* para essa posição!

Teste a aplicação novamente, tudo deverá fluir muito bem.

12.5 REMOVENDO A SELEÇÃO DA LINHA

Da forma que fizemos marcamos a linha alterada mas não a desmarcamos, para desmarca-la precisamos usar o método `deselectRow(at:animated)` podemos fazer isso logo após selecionarmos a linha:

```
override func viewDidAppear(animated: Bool) {
    if let linha = self.linhaDestacada {

        self.tableView.selectRow(at: linha, animated: true, scrollPosition: .middle)
        self.tableView.deselectRow(at: linha, animated: true)
        self.linhaDestacada = Optional.none
    }
}
```

O problema com essa abordagem é o tempo entre uma instrução e outra ocorre muito rápido, e com isso provavelmente não vamos conseguir ver a linha sendo marcada ou desmarcada. Uma outra abordagem seria utilizar algum tipo de `sleep` para criar um atraso entre as duas instruções, porém dessa forma vamos gerar outro problema. Pois a `thread` principal ficará esperando o `sleep` terminar antes de prosseguir, o que fará com que nossa tela fique congelada durante esse `sleep`.

Toda a parte de nossa `view` é executada na `thread` principal, e sempre que temos que processar algo demorado ou nesses casos onde queremos esperar, para executar um determinado código. Devemos criar uma `thread` alternativa para evitar o congelamento da `thread` principal (deixando nosso aplicativo inoperante até o término da tarefa). No iOS temos uma api específica para essa finalidade de lidar com execução em threads separadas, ou agendamento de tarefas futuras na `thread` principal ou em threads separadas. Essa api chamase *GDC (Grand Central Dispatch)*.

Para conseguir o resultado que queremos, podemos pedir para o sistema operacional agendar uma

operação, para ser executada na thread principal. Dessa forma até esse agendamento finalizar (*deadline*), não ficaremos com nossa thread principal congelada. Vamos alterar nosso código para usar esse agendamento assíncrono:

```
override func viewDidAppear(animated: Bool) {
    if let linha = self.linhaDestacada {

        self.tableView.selectRow(at: linha, animated: true, scrollPosition: .middle)

        DispatchQueue.main.asyncAfter(deadline: .now() + .seconds(1)) {
            self.tableView.deselectRow(at: linha, animated: true)
            self.linhaDestacada = .none
        }
    }
}
```

PARA SABER MAIS
[acesse o link:](https://developer.apple.com/library/ios/documentation/Performance/Reference/GCD_libdispatch_Ref/)
https://developer.apple.com/library/ios/documentation/Performance/Reference/GCD_libdispatch_Ref/

Com o código acima estamos pedindo para o sistema operacional esperar um segundo antes de desmarcar a linha.

12.6 EXERCÍCIO - DESELECCIONANDO ASSÍNCRONAMENTE

1. Vamos alterar nosso código para desmarcar a linha de forma assíncrona:

```
override func viewDidAppear(animated: Bool) {
    if let linha = self.linhaDestacada {

        self.tableView.selectRow(at: linha, animated: true, scrollPosition: .middle)

        DispatchQueue.main.asyncAfter(deadline: .now() + .seconds(1)) {
            self.tableView.deselectRow(at: linha, animated: true)
            self.linhaDestacada = .none
        }
    }
}
```

USANDO RECONHECIMENTO DE GESTOS. INTEGRAÇÃO COM TELEFONE, MAPAS E MAIS

À medida que os dispositivos móveis ficam cada vez mais inteligentes, a função de "fazer ligações" torna-se apenas uma comodidade. Quando desenvolvemos um aplicativo para o iOS, por exemplo, muitas vezes nem lembramos que o usuário pode usar esse aplicativo em um iPhone que, por coincidência, também faz ligações! Brincadeiras à parte é importante conhecer os meios de integração com funcionalidades como fazer ligação, enviar SMS e outras desse tipo.

13.1 O EVENTO LONG PRESS: PARA SUA APLICAÇÃO SE COMPORTAR COMO AS NATIVAS

Um evento bastante natural para o usuário de aplicativos *iOS* é o **longPress**, por mais que ele não saiba que isso é um evento. Na verdade, os aplicativos que já vêm instalados no iOS, fornecidos pela própria Apple, já fizeram o trabalho de acostumar os usuários com o **longPress**. Muitas vezes ele já espera que se der um tap em alguma região da tela e mantiver essa seleção, isso deve causar a exibição de um menu com funcionalidades extras, normalmente um menu será exibido para a escolha de alguma ação.

Eventos no iOS são representados por objetos, e todo objeto do tipo `UIView` é capaz de lançar eventos representando a interação do usuário com a aplicação. O interessante é que podemos "avisar" um `UIView` para nos comunicar um evento, em outras palavras isso significa cadastrar um método para que seja possível receber uma mensagem com o evento sendo passado como parâmetro.

Existe uma classe chamada `UILongPressGestureRecognizer` que implementa a lógica de receber uma mensagem da view quando ocorre um evento do tipo **longPress**. O interessante é que podemos informar a um objeto do tipo `gestureRecognizer` para invocar um método implementado por nós quando o evento ocorrer.

Vamos criar um método chamado `exibirMaisAcoes` : para ser invocado quando um **longPress** for disparado, ele vai receber o objeto que representa o evento, uma instância de `UILongPressGestureRecognizer` . Esse `gesture recognizer` será cadastrado na `UITableView` da listagem de contatos.

Primeiramente, vamos alterar a view da classe `ListaContatosViewController` para que ela passe a nos informar quando um evento `longPress` for disparado. Adicione o código no método `viewDidLoad` da `ListaContatosViewController`:

```
override func viewDidLoad() {
    super.viewDidLoad()

    let longPress = UILongPressGestureRecognizer(target: self, action: #selector(exibirMaisAcoes(gesture:)))
    self.tableView.addGestureRecognizer(longPress)
}
```

Perceba que adicionamos o `UILongPressGestureRecognizer` na `tableView`, mas tratamos o evento na `ListaContatosViewController`, pois passamos a própria instância do controller (representada por `self`) como parâmetro para o target onde o selector de `exibirMaisAcoes` será invocado.

Precisamos, agora, implementar o método `exibirMaisAcoes` quando o usuário segurar por mais de meio segundo em um registro de contato.

```
func exibirMaisAcoes(gesture:UIGestureRecognizer){
```

```
}
```

Perceba que recebemos um objeto do tipo `UIGestureRecognizer` na chamada do método, nele temos diversas informações importantes sobre o que o usuário fez para disparar o evento.

A primeira informação importante que precisamos saber é em qual estado está o evento; no nosso caso, queremos fazer algo assim que o evento for iniciado, para isso verificaremos se o `state` do `UIGestureRecognizer` é `UIGestureRecognizerStateBegan`.

```
func exibirMaisAcoes(gesture: UIGestureRecognizer){
    if gesture.state == .began {
        // lógica para descobrir o contato
    }
}
```

Precisamos saber em qual contato o usuário estava interessado quando disparou o evento. Para isso, vamos usar o ponto na tela em que o usuário disparou o evento, em outras palavras, onde exatamente o usuário fez o tap sobre a `tableView`. A partir dessa informação é possível descobrir em qual linha da listagem o usuário estava clicando quando ocorreu o evento e assim recuperar o `Contato` desejado.

```
func exibirMaisAcoes(gesture: UIGestureRecognizer){
    if gesture.state == .began {
        let ponto = gesture.location(in: self.tableView)
    }
}
```

Um método muito útil de um `tableView` vai permitir saber exatamente qual linha da tabela está localizada no `ponto` do tap `longPress` disparado pelo usuário, esse método é o `indexPathForRowAt:`.

O retorno desse método é um `IndexPath` que podemos utilizar para resgatar um contato da lista a partir do seu atributo `row`:

```
func exibirMaisAcoes(gesture: UIGestureRecognizer){

    let ponto = gesture.location(in: self.tableView)
    if let indexPath = self.tableView.indexPathForRow(at:ponto){
        self.contatoSelecionado = self.dao.buscaContatoNaPosicao(indexPath.row)
    }
}
```

Agora, com o Contato em mãos podemos perguntar ao usuário o que ele deseja fazer com as informações contidas no registro.

- Fazer ligação
- Abrir site com a url
- Mostrar endereço no mapa

Vamos utilizar o componente `UIAlertController` para possibilitar ao usuário escolher quais das opções ele deseja executar.

Vale lembrar que o componente `UIAlertController` com o estilo `.actionSheet` permite a criação de diversos botões com um método cada.

Já sabemos como esse componente funciona, portanto vamos para a implementação. Poderíamos declarar a implementação destas ações diretamente na classe `ListaContatosViewController` porém estariamos misturando diversas responsabilidades diferentes na mesma classe. Vamos então extrair este comportamento para uma nova classe. Crie uma nova classe chamada `GerenciadorDeAcoes` Lembre-se que precisamos importar a biblioteca `UIKit` também:

A implementação atual de nosso método `exibirMaisAcoes`: já identifica qual é o contato no qual o usuário está interessado, ou seja, sobre o qual foi realizado o `longPress`. Precisamos exibir o `action sheet` com opções. Como já temos um `Contato`, podemos usá-lo para configurar o texto do `action sheet` baseado no nome. Mas quem tem o contato é a própria lista. Temos que fazer com que o gerenciador receba este contato. Como o gerenciador depende deste contato para funcionar, vamos criar um construtor que receba o contato como parâmetro e uma propriedade para armazená-lo:

O código do construtor ficará como:

```
import UIKit

class GerenciadorDeAcoes: NSObject {

    let contato:Contato

    init(do contato:Contato) {
        self.contato = contato
    }
}
```

```
}
```

Agora vamos criar o método que mostrará as opções de fato e declarar o `UIAlertController` nele. Como o `UIAlertController` precisa de um *controller* para ser exibido, vamos pedi-lo como parâmetro deste novo método e armazena-lo em um atributo pois utilizaremos ele mais a frente:

```
var controller: UIViewController!

exibirAcoes(em controller: UIViewController){
    self.controller = controller
}
```

Vamos fazer agora com que a classe `ListaContatosViewController` instancie o `GerenciadorDeAcoes` com o contato como parâmetro e chame este método passando a referência para o *controller*. Vamos também criar uma propriedade para armazenar a instância do `GerenciadorDeAcoes`:

```
func exibirMaisAcoes(gesture: UIGestureRecognizer){

    let ponto:CGPoint = gesture.location(in: self.tableView)
    let indexPath:IndexPath? = self.tableView.indexPathForRow(at:ponto)

    self.contatoSelecionado = self.dao.buscaContatoNaPosicao(indexPath!.row)

    let acoes = GerenciadorDeAcoes(do: contatoSelecionado)
    acoes.exibirAcoes(em: self)
}
```

Um próximo passo instanciar `UIAlertController` e declarar cada um dos botões e seus respectivos método. Dentro de `acoesDoController` vamos instanciar um objeto do tipo `UIAlertController` e definir o estilo como `.actionSheet`. Como o título do nosso `UIAlertController` vamos passar o nome do nosso contato. Para o nosso caso não vamos precisar especificar uma mensagem então iremos passar `nil` e em `preferredStyle` informaremos `.actionSheet`.

```
exibirAcoes(em controller: UIViewController) {
    self.controller = controller

    let alertView = UIAlertController(title: self.contato.nome, message: nil, preferredStyle: .actionSheet)
}
```

Feito isso vamos adicionar um botão para cada ação: **Ligar**, **Visualizar Site**, **Abrir mapa** e um botão para **Cancelar**. Além disso precisamos dizer que queremos exibir o `UIAlertController`, e para isso precisamos dizer em qual controller iremos exibi-lo.

```
exibirAcoes(em controller: UIViewController) {

    self.controller = controller

    let alertView = UIAlertController(title: self.contato.nome, message: nil, preferredStyle: .actionSheet)

    let cancelar = UIAlertAction(title: "Cancelar", style: .cancel, handler: nil)
```

```

        let ligarParaContato = UIAlertAction(title: "Ligar", style: .default){ action in
            self.ligar()
        }

        let exibirContatoNoMapa = UIAlertAction(title: "Visualizar No Mapa", style: .default) { action in
            self.abrirMapa()
        }

        let exibirSiteDoContato = UIAlertAction(title: "Visualizar Site", style: .default){ action in
            self.abrirNavegador()
        }

        alertView.addAction(cancelar)
        alertView.addAction(ligarParaContato)
        alertView.addAction(exibirContatoNoMapa)
        alertView.addAction(exibirSiteDoContato)

        self.controller.present(alertView, animated: true, completion: nil)
    }
}

```

Ainda precisamos implementar todos esses métodos, mas antes vamos deixar o esqueleto de todos pronto:

```

private func ligar() {
}

private func abrirSite() {
}

private func mostrarMapa() {
}

```

13.2 EXERCÍCIO - GERENCIANDO O EVENTO DE LONGPRESS E UTILIZANDO O UIACTIONSSHEET

1. Gerenciando o evento de longPress .

- Implemente o método `exibirMaisAcoes` : para descobrir em qual linha da `tableView` o usuário disparou o evento de `longPress` . A partir dessa linha vamos buscar qual **Contato** era exibido naquele momento.

```

func exibirMaisAcoes(gesture: UIGestureRecognizer){
    if gesture.state == .began{

        let ponto = gesture.location(in: self.tableView)
        if let indexPath: IndexPath? = self.tableView.indexPathForRow(at:ponto) {
            self.contatoSelecionado = self.dao.buscaContatoNaPosicao(indexPath.row)
        }

    }
}

```

- No método `viewDidLoad` da `ListaContatosViewController` , crie um objeto do tipo

```
UILongPressGestureRecognizer e o adicione como um gestureRecognizer da tableView .
```

```
override func viewDidLoad() {
    super.viewDidLoad()

    let longPress = UILongPressGestureRecognizer(target: self, action: #selector(exibirAcoes(gesture:)))
    self.tableView.addGestureRecognizer(longPress)
}
```

1. Crie a classe `GerenciadorDeAcoes`, filha de `NSObject`. Declare o construtor `init(do contato :)`, a propriedade `contato` e o método `exibirAcoes()` que será utilizado pela `ListaContatosViewController`. Lembre-se de importar a biblioteca `UIKit`

```
import UIKit

class GerenciadorDeAcoes: NSObject {

    let contato: Contato
    var controller: UIViewController!

    init(do contato: Contato) {
        self.contato = contato
    }

    exibirAcoes(em controller: UIViewController) {
        self.controller = controller
    }
}
```

- Implemente o método `exibirAcoes()` para que exiba um `UIAlertController` com estilo `.actionSheet` com as opções de **Ligar**, **Visualizar site** e **Abrir Mapa**.

```
``` swift
```

```
exibirAcoes(em controller: UIViewController) {

 self.controller = controller

 let alertView = UIAlertController(title: self.contato.nome, message: nil, preferredStyle: .actionSheet)

 let cancelar = UIAlertAction(title: "Cancelar", style: .cancel, handler: nil)

 let ligarParaContato = UIAlertAction(title: "Ligar", style: .default){ action in
 self.ligar()
 }

 let exibirContatoNoMapa = UIAlertAction(title: "Visualizar No Mapa", style: .default) { action in
 self.abrirMapa()
 }
```

```

 let exibirSiteDoContato = UIAlertAction(title: "Visualizar Site", style: .default){ action in
 self.abrirNavegador()
 }

 alertView.addAction(cancelar)
 alertView.addAction(ligarParaContato)
 alertView.addAction(exibirContatoNoMapa)
 alertView.addAction(exibirSiteDoContato)

 self.controller.present(alertView, animated: true, completion: nil)
 }

 ...
}

* Declare o esqueleto dos métodos para cada ação:

```swift
private func ligar() {

}

private func abrirSite() {

}

private func mostrarMapa() {

}
```

```

1. Altere o método `exibirMaisAcoes:` para que ele utilize a propriedade que acabamos de declarar.

```

func exibirMaisAcoes(gesture: UIGestureRecognizer){

 let ponto = gesture.location(in: self.tableView)

 if let indexPath = self.tableView.indexPathForRow(at:ponto){

 let contato = self.dao.buscaContatoNaPosicao(indexPath.row)

 let acoes = GerenciadorDeAcoes(do: contato)

 acoes.exibirAcoes(em: self)
 }
}

```

### 13.3 UTILIZANDO URIS PARA INTEGRAÇÃO COM OUTROS APLICATIVOS

Precisamos, agora, implementar os métodos que fazem as possíveis ações do usuário como ligar, enviar e-mail, abrir um site e mostrar o endereço do contato cadastrado. Perceba que todas as ações já estão implementadas nativamente, vamos apenas utilizar as aplicações que têm cada uma dessas funcionalidades.

Porém, precisamos passar parâmetros para explicar em detalhes para o aplicativo nativo exatamente o que queremos. Por exemplo, para o aplicativo de ligação precisamos informar para qual número queremos ligar, no nosso caso, o número do contato.

Para isso vamos utilizar uma funcionalidade presente no *iOS*, a capacidade de cadastrar uma **URL** para um aplicativo. Os aplicativos nativos tem **URLs** específicas que podemos usar para executá-los, passando os parâmetros que vão interferir na forma como eles se comportam. Interessante notar que essa é uma funcionalidade nativa e poderíamos associar uma **URL** à nossa aplicação, dessa forma outros aplicativos poderiam chamá-la para executar alguma funcionalidade específica de acordo com os parâmetros.

Se quisermos fazer uma ligação, devemos antes testar se o nosso dispositivo tem essa capacidade (o iPhone é um dispositivo com telefone, mas o iPod touch por exemplo não tem essa funcionalidade). Para isso é preciso criar uma *URL* com a **URI** do recurso e disparar um `sharedApplication` em *UIApplication*.

Para obter detalhes sobre o dispositivo em que a aplicação está rodando, usamos a classe *UIDevice*. O método *current* retorna um objeto que representa o dispositivo. Uma das informações que podemos obter sobre o dispositivo é o modelo, veja:

#### *GerenciadorDeAcoes*

```
private func ligar() {

 let device = UIDevice.current

 if device.model == "iPhone" {
 // se for iPhone, podemos pedir para fazer um telefonema
 } else {
 // precisamos comunicar ao usuário que essa
 // funcionalidade não está disponível
 }
}
```

Um componente nativo muito utilizado para comunicação rápida com os usuários de aplicativos *iOS* é o *UIAlertController* com o estilo *.alert*. Ele representa um popup que exibe uma mensagem sobreposta à tela, de forma que o usuário possa ler sem problemas, clicar em *ok* para dispensar o popup e continuar utilizando o aplicativo. Podemos usar essa funcionalidade para avisar ao usuário de que não será possível efetuar a ligação. Vamos configurar um *UIAlertController* com título e mensagem e exibir esse **alerta**:

#### *GerenciadorDeAcoes*

```
private func ligar() {

 let device = UIDevice.current

 if device.model == "iPhone" {
 // se for iPhone, podemos pedir para fazer um telefonema
 }
}
```

```

 }else {
 let alert = UIAlertController(title: "Impossível fazer Ligações", message: "Seu dispositivo não é um iPhone", preferredStyle: .alert)

 self.controller.present(alert, animated: true, completion: nil)
 }
 }
}

```

E agora o momento esperado! Como será que enviamos essa URL para o iOS de forma que ele dispare a execução de uma nova aplicação? Além de tudo passando os parâmetros que desejamos? É mais fácil do que pode parecer, a classe *UIApplication* vai disponibilizar uma instância desse mesmo tipo por meio do método *sharedApplication*. Com esse objeto em mãos podemos invocar o *open*: que recebe a URL do aplicativo. E quanto aos parâmetros? Eles serão enviados na própria URL!

Para facilitar a implementação, vamos criar um método especializado em receber uma URL e executar a chamada ao **shared**. Vamos chamar esse método de *abrirAplicativo:com:*. Ele vai receber uma **string** com a URL desejada e transformá-la em um objeto do tipo *URL*, que é o tipo de parâmetro aceito pelo método *open* da **shared**:

#### *GerenciadorDeAcoes*

```

private func abrirAplicativo(com url:String){
 UIApplication
 .shared
 .open(URL(string: url)!, options: [:], completionHandler: nil)
}

```

Vamos atualizar o método *ligar* para que ele gere a nova URL e, por fim, invoque o método *abrirAplicativo:com:*. Note que a URL será formada por "tel:", que é específica para o aplicativo de telefone, e o número de telefone desejado:

#### *GerenciadorDeAcoes*

```

private func ligar() {

 let device = UIDevice.current

 if device.model == "iPhone" {
 abrirAplicativo(com: "tel:" + contato.telefone)
 }else {
 let alert = UIAlertController(title: "Impossível fazer Ligações", message: "Seu dispositivo não é um iPhone", preferredStyle: .alert)

 self.controller.present(alert, animated: true, completion: nil)
 }
}

```

Pronto! E para abrir um site? Todo dispositivo iOS tem o Safari instalado nativamente, portanto, não precisamos fazer nenhuma verificação. Como isolamos a lógica de pedir para a aplicação passar uma URL para o **shared**, torna-se simples implementar a parte do site:

---

```
private func abrirNavegador() {
```

```

var url = contato.site!

if !url.hasPrefix("http://") {
 url = "http://" + url
}

abrirAplicativo(com: url)
}

```

Para mostrar o mapa, a *URL* será `http://maps.google.com/maps?q=` mais o endereço do contato. Como a implementação nativa do iOS já vai tentar utilizar o GPS do dispositivo, se ele existir, o aplicativo de mapas será aberto já com um indicador de posição apontando o endereço atual no mapa. Se não houver um GPS disponível, será utilizada a conexão com a internet do usuário para acesso ao site **Google Maps**.

```

private func abrirMapa() {
 let url = ("http://maps.google.com/maps?q=" + self.contato.endereco!).addingPercentEncoding(withAllowedCharacters: CharacterSet.urlQueryAllowed)!

 abrirAplicativo(com: url)
}

```

## 13.4 EXERCÍCIO - INTEGRAÇÃO COM OUTROS APLICATIVOS ATRAVÉS DE URI'S

1. Na classe `GerenciadorDeAcoes` crie o método `abrirAplicativo:com`:

```

```swift
private func abrirAplicativo(com url:String){

    UIApplication
        .shared
        .open(URL(string: url)!, options: [:], completionHandler: nil)

}
```

```

1. Faça com que o método `ligar` efetue uma chamada para o telefone do contato se o dispositivo for `iPhone`, caso contrário exiba um alerta. Informando que o dispositivo não pode fazer chamadas:

```

private func ligar(){
 let device = UIDevice.current

 if device.model == "iPhone"{
 print("UUID \(device.identifierForVendor!)")
 abrirAplicativo(com: "tel:" + self.contato.telefone!)
 }else{
 let alert = UIAlertController(title: "Impossível fazer Ligações", message: "Seu dispositivo não é um iPhone", preferredStyle: .alert)

 self.controller.present(alert, animated: true, completion: nil)

 }
}

```

2. Faça com que o método `abrirNavegador` exiba o site do contato no navegador:

```
private func abrirNavegador(){
 var url = contato.site!

 if !url.hasPrefix("http://") {
 url = "http://" + url
 }

 abrirAplicativo(com: url)
}
```

3. Faça com que o método `abrirMapa` exiba no navegador o endereço do contato:

```
private func abrirMapa(){
 let url = ("http://maps.google.com/maps?q=" + self.contato.endereco!).addingPercentEncoding(withAllowedCharacters: CharacterSet.urlQueryAllowed)!

 abrirAplicativo(com: url)
}
```

# TIRANDO PROVEITO DO HARDWARE: INTERAGINDO COM A CÂMERA

Agora que o nosso cadastro está completo, que tal alterar o formulário para permitir que o usuário selecione uma imagem para um contato? Vamos adicionar um novo componente ao formulário, para isso será preciso "criar espaço" para ele no layout.

## Atualizando o formulário

Vamos editar o arquivo *Main.storyboard*. Na tela de edição selecione um componente do tipo *ImageView* e o posicione da maneira que desejar no aplicativo.

O componente *ImageView* nada mais é do que um container que pode exibir imagens. Podemos comparar o *ImageView* à uma moldura.

Não remova os componentes existentes para não perder as configurações que foram realizadas no projeto, apenas adicione o novo e reposicione os existentes.

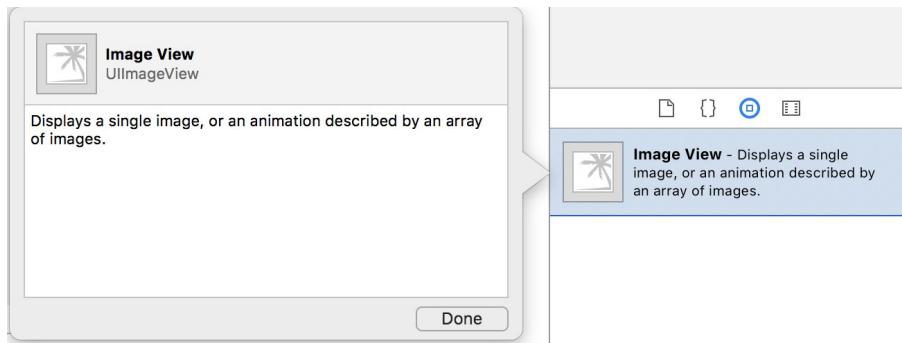


Figura 14.1: ImageView

Vamos implementar essa funcionalidade de forma que o usuário possa clicar no image view para escolher uma imagem, e, quando ele terminar, vamos colocar a foto selecionada no image view.

Crie um `IBOutlet` para o *ImageView* no arquivo *FormularioContatoViewController*. Queremos que nosso *imageView* ao receber um tap, o aplicativo deverá permitir que o usuário selecione uma imagem. Porém por padrão o *imageView* não é um componente interativo, ou seja, ele só é usado para exibir informações. Para habilitar interações nele, vamos alterar o atributo `User Interaction Enabled` na view *Attribute Inspector*.

Precisamos agora que seja invocado um método quando nossa `imageView`. Para isso iremos criar um gesto de `Tap` e associa-lo à nossa `imageView`. E no selector desse gesto vamos chamar o método `selecionaFoto`:

```
override func viewDidLoad() {
 //... restante da implementação

 let tap = UITapGestureRecognizer(target: self, action: #selector(selecionarFoto(sender:)))
 self.imageView.addGestureRecognizer(tap)
}

func selecionarFoto(sender: AnyObject) {
```

## 14.1 USANDO A BIBLIOTECA DE IMAGENS NATIVA DO DISPOSITIVO

Nosso objetivo agora é permitir que o usuário selecione uma imagem. Já sabemos que o iOS roda no *iPhone*, no *iPad* e no *iPod Touch*. O *iPod* não tem uma câmera embutida, enquanto os outros dois dispositivos, sim. Portanto, nossa aplicação deverá tomar uma decisão: se o dispositivo tiver uma câmera disponível, vamos deixar que o usuário escolha entre tirar uma foto ou escolher uma da biblioteca local. Caso contrário, a aplicação vai abrir a biblioteca automaticamente.

Sempre que você criar uma aplicação para interagir com a câmera de um dispositivo, terá que checar se essa funcionalidade está realmente disponível e fornecer uma alternativa. Isso é uma exigência da **Apple**.

Portanto, inicialmente, precisamos verificar se existe uma câmera disponível no dispositivo. Existe uma classe no iOS específica para lidar com eventos relacionados à câmera e à biblioteca de imagens chamada `UIImagePickerController`. Podemos usar o método `isSourceTypeAvailable`: para saber se a câmera está disponível:

```
@IBAction func selecionaFoto(sender: AnyObject) {

 if UIImagePickerController.isSourceTypeAvailable(.camera) {
 //câmera disponível
 } else {
 //vamos apresentar a biblioteca para o usuário.
 }
}
```

Vamos resolver, primeiramente, a vida do usuário que está rodando o aplicativo em um dispositivo sem câmera. Vamos instanciar um novo `UIImagePickerController`. O interessante é que essa classe já implementa a maior parte do comportamento que precisamos, serão necessárias apenas algumas configurações:

```
let imagePicker = UIImagePickerController()
```

Temos que informar qual a fonte de imagens: câmera ou biblioteca do usuário, como estamos

preocupados com o usuário que não tem câmera nesse momento, a fonte será a biblioteca (`UIImagePickerControllerSourceType.photoLibrary` ou `.photoLibrary`).

```
let imagePicker = UIImagePickerController()
pickerController.sourceType = .photoLibrary
```

Uma funcionalidade interessante é permitir que o usuário ajuste a imagem escolhida, para isso vamos atribuir `true` para a propriedade `allowsEditing`.

```
let imagePicker = UIImagePickerController()
pickerController.sourceType = .photoLibrary
pickerController.allowsEditing = true
```

No momento em que o usuário selecionar a imagem, o `UIImagePickerController` vai enviar uma mensagem para seu delegate; vamos definir o próprio controller como delegate e, na sequência, implementar o método que será invocado. A imagem selecionada pelo usuário será enviada como parâmetro para esse método.

```
let imagePicker = UIImagePickerController()
pickerController.sourceType = .photoLibrary
pickerController.allowsEditing = true
pickerController.delegate = self;
```

A propriedade `delegate` de `UIImagePickerController` está definida como qualquer objeto que implemente os protocolos `UINavigationControllerDelegate` e `UIImagePickerControllerDelegate`. Vamos declarar que nossa classe `FormularioContatoViewController` implementa esses protocolos.

### *FormularioContatoViewController*

```
import UIKit

class FormularioContatoViewController: UIViewController,
 UINavigationControllerDelegate,
 UIImagePickerControllerDelegate
```

Será necessário implementar o método `imagePickerController:didFinishPickingMediaWithInfo:` que será invocado no processo de **delegation**. Faremos isso em instantes.

Além disso, é preciso exibir a view relacionada com o `UIImagePickerControllerDelegate`, para isso usaremos o método `present:animated:completion:`. Então, o método `selecionaFoto:` estará assim:

```
func selecionaFoto(sender: AnyObject) {

 if UIImagePickerController.isSourceTypeAvailable(.camera) {
 //câmera disponível
 }else {
 let imagePicker = UIImagePickerController()
 pickerController.sourceType = .photoLibrary
 pickerController.allowsEditing = true
 pickerController.delegate = self
 }
}
```

```
 self.present(imagePicker, animated: true, completion: nil)
 }
}
```

Quando a imagem for selecionada, será enviada a mensagem `imagePickerController:pickerDidFinishPickingMediaWithInfo:` para o delegate. Esse método recebe informações sobre a imagem selecionada pelo usuário. Como o delegate é a própria classe `FormularioContatoViewController` e já definimos que ela implementa os protocolos necessários, vamos apenas implementar o método:

#### *FormularioContatoViewController*

```
func imagePickerController(picker: UIImagePickerController, didFinishPickingMediaWithInfo info: [String : AnyObject]) {
}
```

Primeiramente, vamos recuperar um objeto que representa a imagem selecionada pelo usuário. Essa informação está armazenada no Dictionary passado como parâmetro em `didFinishPickingMediaWithInfo:`:

#### *FormularioContatoViewController*

```
func imagePickerController(picker: UIImagePickerController, didFinishPickingMediaWithInfo info: [String : AnyObject]) {
 if let imageSelecionada = info[UIImagePickerControllerEditedImage] as? UIImage {
 }
 // restante da implementação
}
```

O nosso próximo passo é alterar o `imageView` para adicionar a imagem escolhida pelo usuário no próprio `imageView`:

#### *FormularioContatoViewController*

```
func imagePickerController(picker: UIImagePickerController, didFinishPickingMediaWithInfo info: [String : AnyObject]) {
 if let imageSelecionada = info[UIImagePickerControllerEditedImage] as? UIImage {
 self.imageView.image = imagemSelecionada
 }
 // restante da implementação
}
```

Feito! Precisamos apenas esconder a biblioteca de imagens do usuário agora. Note que a instância de `UIImagePickerController` foi passada como parâmetro para nosso método. Vamos enviar uma mensagem para esse objeto para que ele esconda a própria view usando uma animação. Usaremos o método `dismiss:animated:completion:`.

#### *FormularioContatoViewController*

---

```
func imagePickerController(picker: UIImagePickerController, didFinishPickingMediaWithInfo info: [String : AnyObject]) {
 if let imageSelecionada = info[UIImagePickerControllerEditedImage] as? UIImage {
 self.imageView.image = imagemSelecionada
 }

 picker.dismiss(animated: true, completion: nil)
}
```

## ADICIONANDO IMAGENS À BIBLIOTECA DO SIMULADOR

Existe um **técnica alternativa**, ou em outras palavras, **um macete** para gravar uma imagem na biblioteca de imagens do simulador. Ele não tem uma câmera disponível, mas é possível gravar imagens na biblioteca a partir do *Safari*.

Rode o simulador, pode ser inclusive com a aplicação que estamos desenvolvendo e clique no botão *home*. Abra o *Safari* e navegue para qualquer site que tenha uma imagem que você queira usar para esse teste.

Dê um *long tap* sobre a imagem desejada. Um *action sheet* será exibido com opções de ações para aquela imagem específica. Selecione a opção *Save Image*:



Figura 14.2: Long tap sobre imagem no Safari

Pronto! A imagem está na biblioteca de imagens. Você pode conferir se deu tudo certo acessando o aplicativo *Photos*.

Caso esteja utilizando o simulador do iOS 8, não precisa se preocupar pois ele já contém algumas imagens por padrão.

## 14.2 EXERCÍCIO - SELECIONANDO UMA IMAGEM DA BIBLIOTECA

1. Adicione o botão que vai representar a imagem do contato na tela de formulário. Basta arrastar o

componente e o posicionar da maneira desejada no arquivo `Main.storyboard`.

Veja a seguir uma sugestão de layout:



Figura 14.3: Layout de layout para o imageView

2. Crie um `IBOutlet` chamado `imageView` para o `imageView` criado. É por meio desse *outlet* que poderemos alterar a imagem exibida quando ela for escolhida pelo usuário.
3. Após adicionar a declaração do *outlet* no arquivo `FormularioContatoViewController`, lembre-se de editar o arquivo `.storyboard` e fazer a ligação do novo *outlet* com a `imageView` na interface.
4. Agora precisamos permitir que nosso `imageView` ao ser tocado ele invoke um método. Para isso precisamos alterar o atributo **User Interaction Enabled** na view `Attribute Inspector`.
5. Vamos criar um gesto de `Tap` e associa-lo ao nosso `imageView`. No final do método `viewDidLoad` adicione as seguintes instruções:

```
override func viewDidLoad() {
 //...
 let tap = UITapGestureRecognizer(target: self, action: #selector(selecionarFoto(sender:))
}
 self.imageView.addGestureRecognizer(tap)
}
```

6. Ótimo, agora vamos implementar o método `selecionaFoto:`:

```
func selecionaFoto(sender: AnyObject) {
```

```

 if UIImagePickerController.isSourceTypeAvailable(.camera) {
 //câmera disponível
 }else {
 //usar biblioteca
 }
}

```

7. Por enquanto, não vamos nos preocupar em dar suporte à câmera, faremos isso mais adiante ainda nesse capítulo. Portanto, vamos instanciar um `UIImagePickerController` no bloco `else` do `if` que verifica se é possível ou não utilizar a câmera para obter imagens.

Uma coisa importante é que para usar um `UIImagePickerController`: é preciso definir um delegate. Um método será invocado nesse delegate quando a imagem for selecionada.

Instancie um `UIImagePickerController`, definindo a biblioteca como fonte de dados. Defina o próprio objeto formulário como delegate desse controller. E por fim apresente o controller criado como um modal.

```

func selecionaFoto(sender: AnyObject) {

 if UIImagePickerController.isSourceTypeAvailable(.camera) {
 //câmera disponível
 }else {
 let imagePicker = UIImagePickerController()
 pickerController.sourceType = .photoLibrary
 pickerController.allowsEditing = true
 pickerController.delegate = self

 self.present(imagePicker, animated: true, completion: nil)
 }
}

```

8. No exercício anterior, definimos o objeto formulário como delegate de uma instância de `UIImagePickerController`. O próprio Xcode está reclamando com um *warning* porque o formulário não implementa os protocolos necessários para ser delegate de um *image picker*.

Para resolver isso, na declaração da classe, indique que os protocolos serão implementados:

```

class FormularioContatoViewController: UIViewController,
 UINavigationControllerDelegate ,
 UIImagePickerControllerDelegate

```

9. O *image picker* invocará o método `imagePickerController:didFinishPickingMediaWithInfo:` em seu `delegate` quando o usuário selecionar uma imagem.

Implemente esse método no arquivo `FormularioContatoViewController.m`.

```

func imagePickerController(picker: UIImagePickerController,
 didFinishPickingMediaWithInfo info: [String : AnyObject]) {

}

```

10. Agora altere a implementação para recuperar um objeto do tipo `UIImage` a partir da seleção de

imagem feita pelo usuário.

```
func imagePickerController(picker: UIImagePickerController,
 didFinishPickingMediaWithInfo info: [String : AnyObject]) {

 if let imageSelecionada = info[UIImagePickerControllerEditedImage] as? UIImage {
 self.imageView.image = imageSelecionada
 }
}
```

11. Por fim, use a imagem selecionada pelo usuário para alterar a aparência do botão e esconda novamente o modal do *image picker*.

```
func imagePickerController(picker: UIImagePickerController,
 didFinishPickingMediaWithInfo info: [String : AnyObject]) {

 if let imageSelecionada = info[UIImagePickerControllerEditedImage] as? UIImage {
 self.imageView.image = imageSelecionada
 }

 picker.dismiss(animated: true, completion: nil)
}
```

Rode a aplicação. Está tudo funcionando, certo? Será que está mesmo? Você vai notar que, ao alterar um contato e atribuir uma imagem a ele, essa imagem vai desaparecer assim que você voltar para a tela de listagem. Vamos corrigir isso!

## 14.3 ARMAZENANDO OBJETOS DO TIPO IMAGEM

Precisamos armazenar a imagem de um contato na memória. Vamos cuidar disso agora.

Precisamos adicionar uma nova propriedade à classe `Contato` para armazenar o objeto do tipo `UIImage` que usaremos para armazenar a imagem. Vamos chamar essa propriedade de `foto`. Crie a declaração para essa propriedade:

```
@property (strong) UIImage *foto;
```

Agora, precisamos cuidar do formulário, o método `pegaDadosDoFormulario` passa a ter a responsabilidade de obter, também, a imagem associada ao botão (se houver alguma) e armazenar esse objeto na propriedade `foto` do contato. Para isso usamos a propriedade `image` do nosso `imageView`:

```
func pegaDadosDoFormulario(){
 if contato == nil {
 self.contato = Contato()
 }

 self.contato.foto = self.imageView.image

 // ... restante das atribuições
}
```

Pronto! Rode a aplicação, edite contatos, associe imagens a eles, volte para a lista e os edite novamente. Dessa vez, a imagem permanece associada ao registro, mas não é exibida, certo? Precisamos adicionar uma verificação. Se um contato editado já possui uma imagem, então botão já aparece com aquela imagem.

Adicione a seguinte lógica ao método `viewDidLoad` do formulário:

```
override func viewDidLoad() {
 super.viewDidLoad()

 if contato != nil {
 self.nome.text = contato.nome
 self.telefone.text = contato.telefone
 self.endereco.text = contato.endereco
 self.site.text = contato.site

 if let foto = self.contato.foto {
 self.imageView.image = foto
 }
 }

 let botaoAlterar: UIBarButtonItem = UIBarButtonItem(title: "Confirmar", style: .Plain, target
: self, action: #selector(atualizaContato))

 self.navigationItem.rightBarButtonItem = botaoAlterar
}

}
```

Ótimo, agora a imagem está gravada em memória.

## 14.4 EXERCÍCIO - ARMAZENANDO A IMAGEM DO CONTATO

1. Adicione a diretiva para a criação da nova propriedade `foto` do tipo `UIImage` no arquivo `Contato.h`.

```
#import <UIKit/UIKit.h>

@interface Contato : NSObject
// outras propriedades
@property (strong) UIImage *foto;
@end
```

2. Além dos dados digitados pelo usuário, o método `pegaDadosDoFormulario` da classe `FormularioContatoViewController` também precisa recuperar a imagem selecionada pelo usuário. Essa imagem fica armazenada na propriedade `image` do `imageView`.

Edite o método para que ele fique como a seguir:

```
func pegaDadosDoFormulario(){
 if contato == nil {
 self.contato = Contato()
 }

 self.contato.foto = self.imageView.image
```

```

 self.contato.nome = self.nome.text!
 self.contato.telefone = self.telefone.text!
 self.contato.endereco = self.endereco.text!
 self.contato.site = self.site.text!
 }

```

3. Por fim, ao exibir o formulário para um contato já existente, queremos que a imagem associada a ele (se existir alguma) apareça no imageView.

Atualize o método `viewDidLoad` e adicione uma verificação: se a `image` existir no contato, ela deve ser utilizada no `imageView`:

```

override func viewDidLoad() {
 super.viewDidLoad()

 if contato != nil {
 self.nome.text = contato.nome
 self.telefone.text = contato.telefone
 self.endereco.text = contato.endereco
 self.site.text = contato.site

 if let foto = contato.foto{
 self.imageView.image = self.contato.foto
 }

 let botaoAlterar = UIBarButtonItem(title: "Confirmar", style: .plain, target: self, action: #selector(atualizar))

 self.navigationItem.rightBarButtonItem = botaoAlterar
 }
}

```

Rode a aplicação e faça alguns testes! Tudo deve estar funcionando perfeitamente agora.

## 14.5 UIALERTCONTROLLER - ACTIONSHEET: UMA FORMA NATIVA DE SELECIONAR UMA AÇÃO

Agora, vamos resolver a vida de quem está executando o programa em um dispositivo que tem a câmera disponível. Vamos permitir que o usuário escolha entre tirar uma nova foto ou simplesmente escolher uma da biblioteca. Existe uma funcionalidade no iOS específica para exibir uma lista de opções para o usuário. Cada ação é representada por um botão. Essa funcionalidade é a mesma utilizada pela Apple para criar os aplicativos nativos do iOS.

Para implementar isso, devemos utilizar uma instância da classe `UIAlertController` e definir o estilo `.actionSheet`.

Para instanciar um `UIAlertController` precisamos informar um título para a view que vai aparecer sobre a tela atual uma mensagem opcional e o estilo de alerta que queremos que seja exibido. Existem dois tipos de alertas o `.alert` que exibe um *popup* e o `.actionSheet` que exibe uma lista de

ações para o usuário interagir. No nosso caso usaremos `.actionSheet`.

Podemos definir vários botões em um alert com estilo `.actionSheet`, no nosso exemplo serão necessários apenas dois: "Tirar foto" e "Escolher da biblioteca". Veja como fica a configuração:

```
let alert = UIAlertController(title: "Escolha foto do contato", message: self.contato.nome, preferredStyle: .actionSheet)

let cancelar = UIAlertAction(title: "Cancelar", style: .cancel, handler: nil)

let tirarFoto: UIAlertAction = UIAlertAction(title: "Tirar Foto",
 style: .default){ (action) in
 //implementação da ação do botão aqui
}

let tirarFoto = UIAlertAction(title: "Tirar Foto", style: .default){ (action) in
 //implementação da ação do botão aqui
}

alert.addAction(cancelar)
alert.addAction(tirarFoto)
alert.addAction(escolherFoto)
```

Além disso, será preciso configurar como será apresentado esse `alert`. Vamos completar o código:

```
let alert = UIAlertController(title: "Escolha foto do contato", message: self.contato.nome, preferredStyle: .actionSheet)

let cancelar = UIAlertAction(title: "Cancelar", style: .cancel, handler: nil)

let tirarFoto: UIAlertAction = UIAlertAction(title: "Tirar Foto",
 style: .default){ (action) in
 //implementação da ação do botão aqui
}

let tirarFoto = UIAlertAction(title: "Tirar Foto", style: .default){ (action) in
 //implementação da ação do botão aqui
}

alert.addAction(cancelar)
alert.addAction(tirarFoto)
alert.addAction(escolherFoto)

self.present(alert, animated: true, completion: nil)
```

## 14.6 ACESSANDO A CÂMERA DO DISPOSITIVO

Vamos configurar um novo `UIImagePickerController` dependendo da opção escolhida pelo usuário, vamos configurar a fonte de dados para esse **image picker** como sendo a câmera ou a biblioteca. Por fim, vamos exibir o **image picker**.

Antes de mais nada, vamos instanciar um novo **image picker** e configurar seu delegate para ser o controller. Também vamos habilitar a opção de ajuste da imagem selecionada pelo usuário:

### *FormularioContatoViewController*

```
let imagePicker = UIImagePickerController()
imagePicker.allowsEditing = true
imagePicker.delegate = self
```

Vamos adicionar comportamento para os botões alterando onde será a fonte de dados do **image picker** e exibi-lo:

### *FormularioContatoViewController*

```
if UIImagePickerController.isSourceTypeAvailable(.camera) {

 let imagePicker = UIImagePickerController()
 imagePicker.allowsEditing = true
 imagePicker.delegate = self

 let alert = UIAlertController(title: "Escolha foto do contato", message: self.contato.nome, preferredStyle: .actionSheet)

 let cancelar = UIAlertAction(title: "Cancelar", style: .cancel, handler: nil)

 let tirarFoto = UIAlertAction(title: "Tirar Foto", style: .default){ (action) in
 imagePicker.sourceType =
 .camera
 }

 let escolherFoto = UIAlertAction(title: "Escolher da biblioteca", style: .default){ (action) in
 imagePicker.sourceType = .photoLibrary
 }

 alert.addAction(cancelar)
 alert.addAction(tirarFoto)
 alert.addAction(escolherFoto)

 self.present(alert, animated: true, completion: nil)
} else {
 //restante da implementação
}
```

Perceba que o código para instanciar um novo **image picker** é exatamente o mesmo código que fizemos quando o *device* não tem camera. Vamos melhorar um pouco nosso código removendo as duplicidades:

```
private func pegarImage(da sourceType: UIImagePickerControllerSourceType){

 let imagePicker = UIImagePickerController()
 imagePicker.allowsEditing = true
 imagePicker.delegate = self
 imagePicker.sourceType = sourceType

 self.present(imagePicker, animated: true, completion: nil)
}

func selecionaFoto(sender: AnyObject) {

 if UIImagePickerController.isSourceTypeAvailable(.camera){
```

```

 let alert = UIAlertController(title: "Escolha foto do contato", message: self.contato.nome, preferredStyle: .actionSheet)

 let cancelar = UIAlertAction(title: "Cancelar", style: .cancel, handler: nil)
 let tirarFoto = UIAlertAction(title: "Tirar Foto", style: .default){ (action) in
 self.pegarImage(da: .camera)
 }

 let escolherFoto = UIAlertAction(title: "Escolher da biblioteca", style: .default){ (action) in
 self.pegarImage(da: .photoLibrary)
 }

 alert.addAction(cancelar)
 alert.addAction(tirarFoto)
 alert.addAction(escolherFoto)

 self.present(alert, animated: true, completion: nil)

 }else{
 pegarImage(da: .photoLibrary)
 }
}

```

Perceba que configuramos como **delegate** do **image picker** o controller `FormularioContatoViewController`. Será que falta implementar algum método? Na verdade já fizemos isso! O método `imagePickerController:didFinishPickingMediaWithInfo:` já foi implementado com a lógica que precisamos.

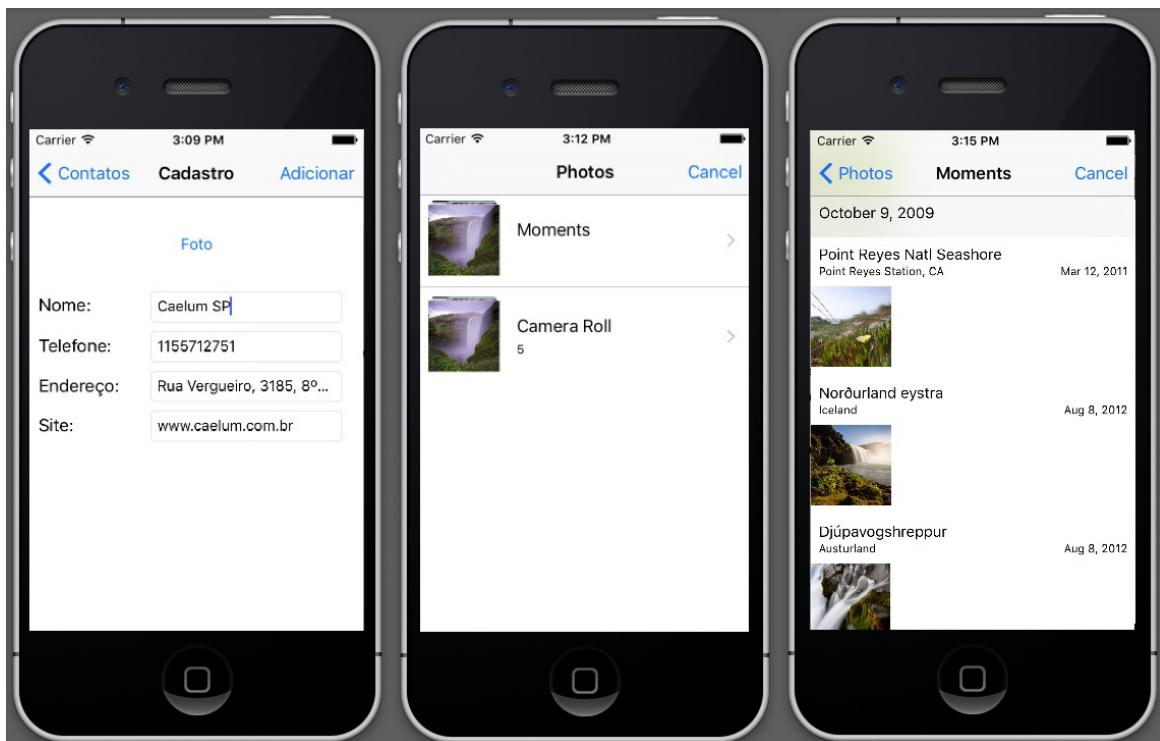


Figura 14.4: Formulário com imagem

## 14.7 EXERCÍCIO (OPCIONAL): ESCOLHENDO ENTRE CÂMERA OU

## BIBLIOTECA

1. Quando a câmera estiver disponível para uso no dispositivo, permitiremos que o usuário faça a escolha entre câmera ou biblioteca para selecionar a foto. Para isso, usaremos novamente o `UIAlertController`.

Altere o método `selecionaFoto:` do formulário. Caso a câmera esteja disponível, instancie um `UIAlertController` com estilo `.actionSheet` com o título "Escolha a foto do contato". Ele deve exibir duas opções: *tirar foto* ou *escolher da biblioteca*:

```
private func pegarImage(da sourceType: UIImagePickerControllerSourceType){

 let imagePicker = UIImagePickerController()
 imagePicker.allowsEditing = true
 imagePicker.delegate = self
 imagePicker.sourceType = sourceType

 self.present(imagePicker, animated: true, completion: nil)
}

func selecionaFoto(sender: AnyObject) {

 if UIImagePickerController.isSourceTypeAvailable(.camera){

 let alert = UIAlertController(title: "Escolha foto do contato", message: self.contato
.nome, preferredStyle: .actionSheet)

 let cancelar = UIAlertAction(title: "Cancelar", style: .cancel, handler: nil)
 let tirarFoto = UIAlertAction(title: "Tirar Foto", style: .default){ (action) in
 self.pegarImage(da: .camera)
 }

 let escolherFoto = UIAlertAction(title: "Escolher da biblioteca", style: .default){ (action) in
 self.pegarImage(da: .photoLibrary)
 }

 alert.addAction(cancelar)
 alert.addAction(tirarFoto)
 alert.addAction(escolherFoto)

 self.present(alert, animated: true, completion: nil)

 }else{
 pegarImage(da: .photoLibrary)
 }
}
```

## VISUALIZANDO O MAPA E APRESENTANDO MAIS DE UMA TELA

Agora que a aplicação está completa e conseguimos tirar proveito do hardware, podemos adicionar funcionalidades interessantes ao aplicativo: imagine visualizar todos os contatos em um mapa, mostrando cada contato como um pino, vendo o quanto próximo cada contato está do outro. É exatamente o que faremos!

Primeiramente, precisamos ter o mapa para poder adicionar os pinos. Entretanto, no iOS temos algo pronto e vamos aprender como utilizar e gerenciar um mapa da mesma forma que o aplicativo nativo de mapa faz.

### 15.1 CRIANDO UMA TELA COM O ELEMENTO MKMAPVIEW

O componente usado na aplicação nativa de mapa no iOS é o *MKMapView* e é ele que iremos utilizar, porém, inicialmente, precisamos de uma tela para poder exibi-lo. Sendo assim, como fizemos ao desenhar o formulário e a listagem de contatos, precisamos de um novo *UIViewController*.

Para isso, vá em *New -> File* e crie uma classe chamada **ContatosNoMapViewController** que será subclasse de *UIViewController* e terá a lógica para adicionar os pontos no mapa.

Com os arquivos criados, vamos abrir o *storyboard* e editar a nossa interface, vamos adicionar um *ViewController* vinculá-lo a nossa classe **ContatosNoMapViewController** depois busque no *Object Library* um elemento visual do tipo *MapKit View* e arraste-o para nosso *ViewController*, deixando-o ocupar toda a tela.

Se tentarmos rodar neste momento, não conseguiremos visualizar o mapa pois essa classe está presente em um *framework* a parte, o **MapKit** e para utilizá-la precisamos importá-lo e criar um outlet para nosso componente. Para isso vamos criar uma classe, associa-la ao nosso *ViewController* e dentro vamos importar o *MapKit*

```
import UIKit
import MapKit

class ContatosNoMapViewController: UIViewController {

 @IBOutlet weak var mapa: MKMapView!
```

}

## 15.2 EXERCÍCIO - CRIANDO OUTRA VIEW E ADICIONANDO O MAPA

1. Crie uma nova *UIViewController* para adicionarmos o elemento de mapa.
  - Vá em *File* -> *New* -> *File* (cmd+n) e crie uma nova Classe chamada **ContatosNoMapaViewController**, faça com que ela seja **Subclass of UIViewController**.
2. Adicionando um elemento do tipo *MKMapView* à interface
  - Busque o elemento visual *MapKit View* no *Object Library*, arraste-o para a tela e faça com que ele ocupe todo o espaço da *View*.
  - Deixe a interface da seguinte forma:

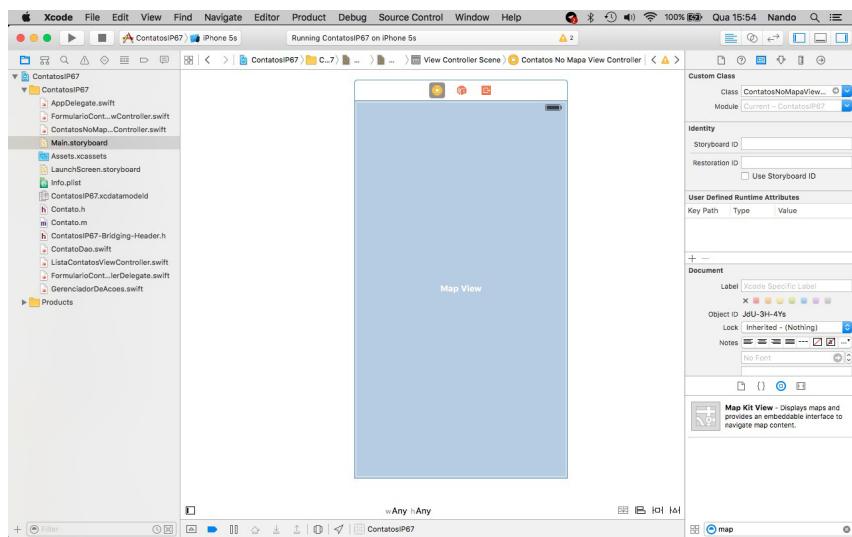


Figura 15.1: Adicionando o elemento de Mapa

3. No arquivo **ContatosNoMapaViewController.swift**, importe o **MapKit**, que possui tudo o que precisamos para colocar o mapa na tela e crie um *outlet* para nosso componente:

```
import UIKit
import MapKit

class ContatosNoMapaViewController: UIViewController {

 @IBOutlet weak var mapa: MKMapView!
}
```

E agora, onde vamos visualizar a tela que criamos? A Apple tem um componente para isso!

## 15.3 MOSTRANDO VÁRIAS TELAS COM O ELEMENTO

## UITABBARCONTROLLER

O nosso usuário já é capaz de visualizar uma tela inicial: a listagem de contatos. Durante o desenvolvimento, aprendemos como utilizar a `UINavigationController` para adicionar a funcionalidade de navegação entre diversas telas e também como apresentar uma tela por cima de outra de forma modal, utilizando o método `present:animated:completion:` presente em toda `UIViewController`.

No entanto, nenhuma dessas maneiras possibilita nosso usuário visualizar mais de uma tela ao mesmo tempo. O interessante seria manter mais de uma tela aberta, tornando fácil a navegação entre elas.

Queremos, também, que nosso usuário veja mais de uma tela e, entre as transições, que o estado da tela anterior continue o mesmo. Para isso, o iOS nos provê um `UIViewController` específico, o `UITabBarController`. Esse elemento é utilizado em vários aplicativos desenvolvidos pela Apple e é capaz de fazer o que queremos.

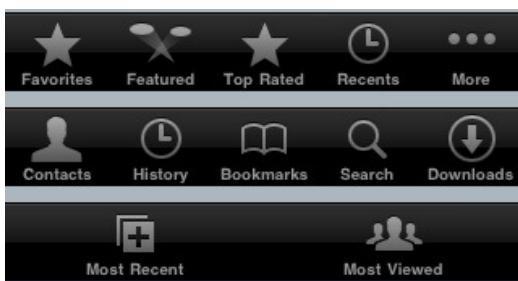


Figura 15.2: Elemento UITabBar

Como uma `UITabBarController` é filha de `UIViewController`, podemos torná-la a tela principal do aplicativo já que o próprio elemento fará a transição entre todas as *view controllers* que declaramos.

Vamos alterar nossa aplicação para exibir as telas que desejamos.

## 15.4 EXERCÍCIO - CRIANDO UM UITABBARCONTROLLER E EXIBINDO MAIS DE UMA TELA

### 1. Utilizando `UITabBarController`.

- Abra o arquivo *storyboard*. Vamos adicionar um `TabBarController` e vamos alterar o ponto inicial da nossa aplicação para ele.
- Apague as duas telas que estão relacionadas com nosso `TabBarController`
- Relacione o `TabBarController` com o **Navigation Controller** da listagem de contatos. (O tipo desse relacionamento é `Relationship Segue`)

- Faça com que a tela que tem o mapa se torne um **Navigation Controller**. Para isso selecione a tela acesse o menu *Editor > Embed In > Navigation Controller*.
- Relacione o fluxo do mapa (que acabamos de criar) com nosso `TabBarController`, da mesma forma que fizemos com o fluxo da listagem.
- Ao término nosso *layout* no `Storyboard` deve estar da seguinte maneira:

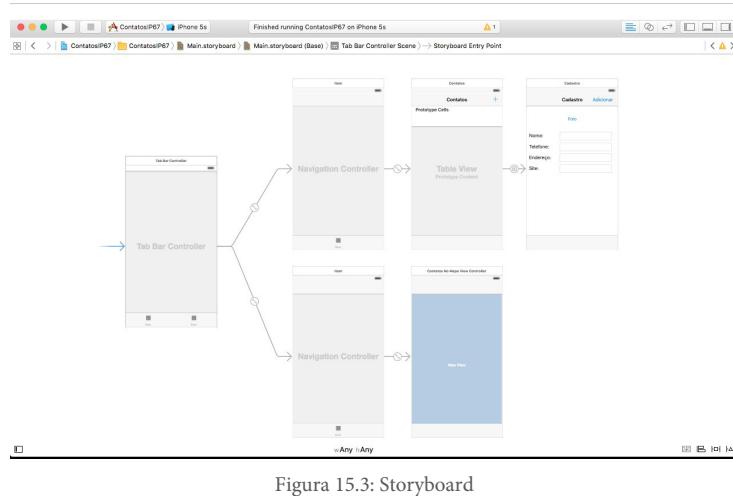


Figura 15.3: Storyboard

- Ao rodar a aplicação, veremos as duas telas sendo exibidas para o usuário com um elemento que facilita a transição entre elas.

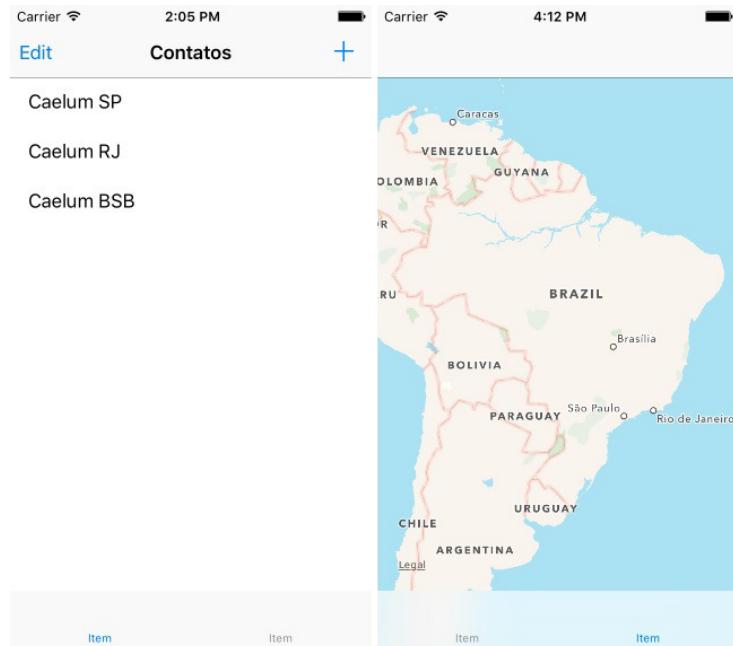


Figura 15.4: Aplicação

- Perceba que o título está como *item* ou podem estar em branco e não temos nenhum ícone sendo mostrado. Como podemos melhorar?

## 15.5 ALTERANDO OS ÍCONES DE UMA UITABBARCONTROLLER

Será que para alterar os botões e títulos que a `UITabBarController` exibe precisamos alterar algum atributo da própria `UITabBarController`?

Essa seria a forma mais natural de pensar, não? Porém, não é assim que acontece, e faz sentido.

Quando uma `UITabBarController` exibe um `UIViewController`, a própria `tabBar` pergunta para a `viewController` que está sendo exibida qual botão deve ser mostrado, este botão deve conter a imagem e o título desejado.

Portanto, precisamos alterar o texto dos botões diretamente nas telas de *Listagem* e *Mapa*.

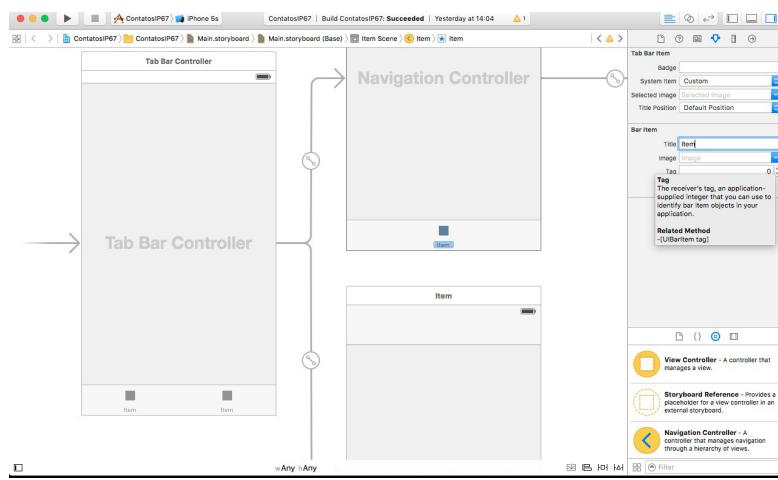


Figura 15.5: Botões TabBarController

Existem alguns estilos de botões prontos com imagem e título, para usá-los basta selecionar algum deles em *System Item*:

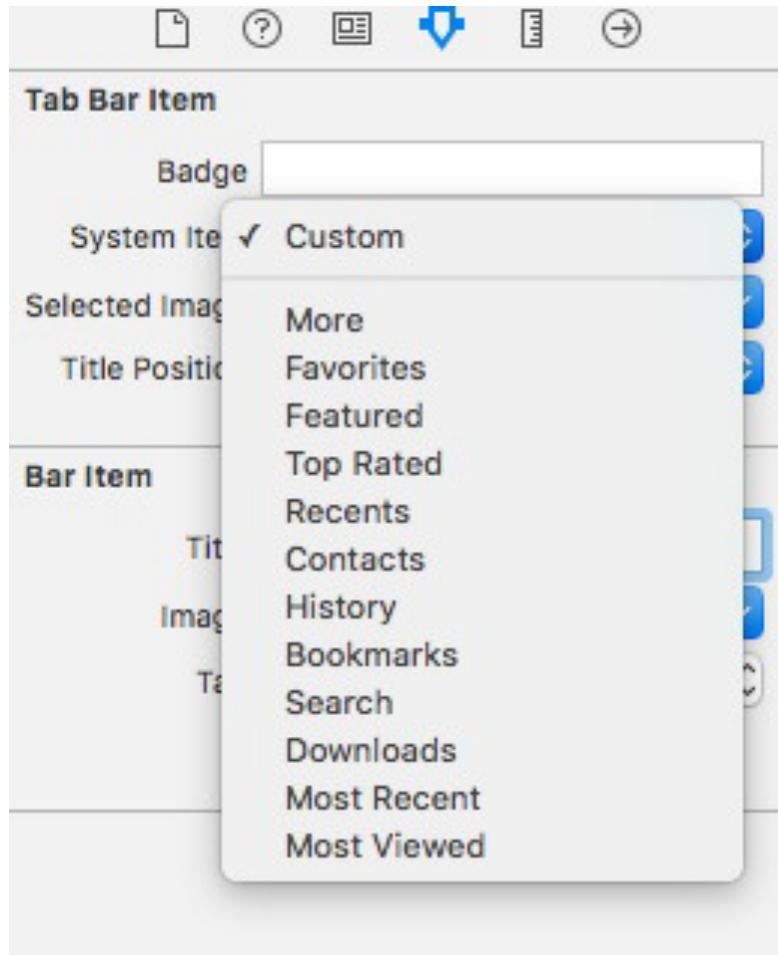


Figura 15.6: Estilos pré-definidos para TabBarItem

#### BOTÕES CUSTOMIZADOS

Ao criarmos um botão usando os estilos prontos no *iOS*, não conseguimos modificar o título nem a imagem do botão.

Como queremos criar um botão com a nossa cara, precisamos passar um título e uma imagem customizada para o elemento. Entretanto, primeiramente, temos que conseguir carregar uma imagem. Caso essa imagem esteja dentro do nosso projeto, o processo é bem simples.

## 15.6 CARREGANDO UMA IMAGEM DO PROJETO PARA A APLICAÇÃO

Toda imagem presente em nosso projeto está disponível para ser utilizada dentro da aplicação. Imagens no *iOS* são representadas pela classe `UIImage` e por facilidade existe um método de classe que recebe uma `String` que é o nome da imagem. Se essa imagem estiver dentro do projeto, perfeito, ela

será carregada e estará pronta para ser utilizada.

```
let imagem:UIImage = UIImage(named:"nome_da_imagem")
```

O **HIG** da Apple manda que as imagens mostradas em uma *UITabBar* tenham o tamanho de 30x30px e, em telas retina display, 60x60px. Lembrando que as imagens serão trocadas automaticamente caso o nome siga o padrão da Apple: terminando com @2x, a imagem lista-contatos.png vira lista-contatos@2x.png, por exemplo.

## 15.7 EXERCICIO: ADICIONANDO BOTÕES NA UITABBAR

### 1. Adicionando imagens ao projeto

- Clique no arquivo **Assets.xcassets**
- Clique no botão + e selecione a opção **Import**

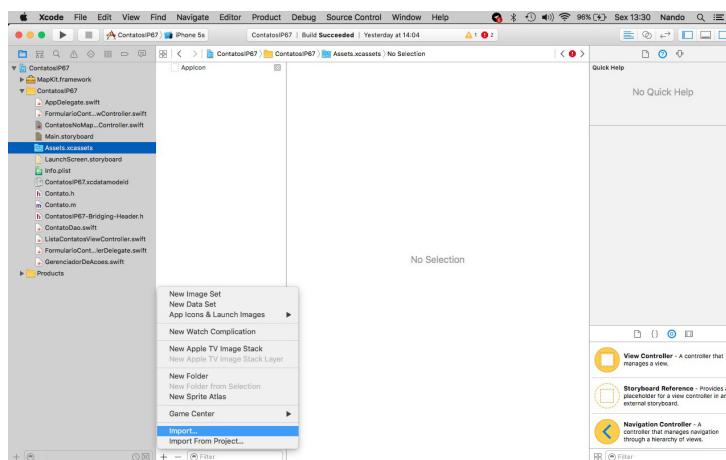


Figura 15.7: Importando imagens

- Selecione os arquivos abaixo na pasta da Caelum de seu Desktop e clique em **Open**:
- lista-contatos.png
- lista-contatos@2x.png
- mapa-contatos.png
- mapa-contatos@2x.png

### 2. Alterando nossas Telas para exibir um *UITabBarItem* customizado.

- Abra o *storyboard* selecione o *navigationController* referente à *Listagem de Contatos* e altere os atributos:
  - *Title* para *Contatos*
  - *Image* para *lista-contatos*

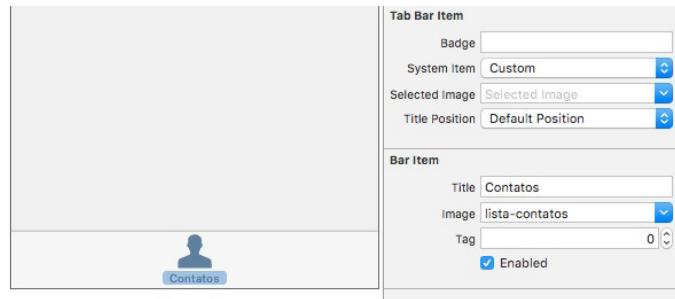


Figura 15.8: Botão lista contatos

- Abra o `storyboard` selecione o `navigationController` referente à *Contatos no Mapa* e altere os atributos:
  - *Title* para *Mapa*
  - *Image* para *mapa-contatos*

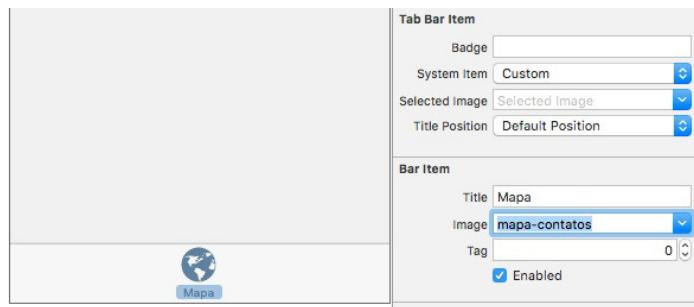


Figura 15.9: Botão lista contatos

- Rode a aplicação, que deverá ter a seguinte cara:

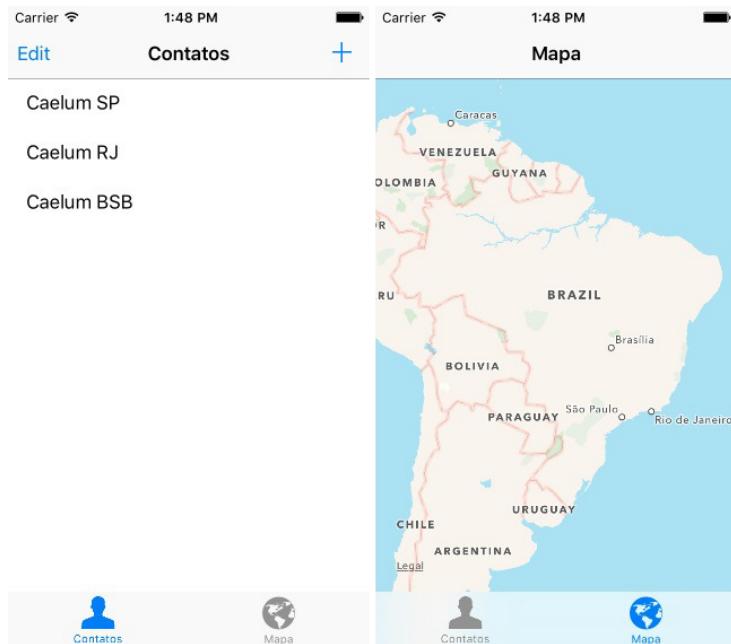


Figura 15.10: Aplicação

#### PARA SABER MAIS: MUDANDO DE ABAS DE FORMA PROGRAMÁTICA

Podemos trocar a aba selecionada na *UITabBar* dizendo qual índice queremos que seja mostrado, para isso podemos passar para a *UITabBarController* o valor desejado.

```
tabBarController.selectedIndex = 0;
```

## 15.8 MOSTRANDO A LOCALIZAÇÃO ATUAL DO USUÁRIO

Antes de adicionar a localização dos nossos contatos no mapa, seria interessante ver a sua localização atual, porém, perceba que no iOS temos uma aplicação de mapa e nela já temos um botão que faz algo parecido!



Figura 15.11: MKUserTrackingBarButtonItem

Esse botão, além de mostrar a localização atual do usuário, consegue utilizar o giroscópio do aparelho para mostrar para qual direção no mapa o usuário está apontando. Interessante, não?

E o mais legal: temos uma classe pronta que faz exatamente isso chamada

```
MKUserTrackingBarButtonItem .
```

A classe `MKUserTrackingBarButtonItem` possui um construtor que recebe o mapa que ele terá controle: `mapView:` que recebe como argumento um objeto do tipo `MKMapView`. Note que é o mesmo tipo do mapa que adicionamos na tela `ContatosNoMapaViewController`.

Como precisamos acessar o objeto que está instanciado na `view`, precisamos de um `@IBOutlet` na classe `ContatosNoMapaViewController` para o nosso mapa.

Precisamos esperar que tudo esteja conectado entre nossa `view` e os atributos que declaramos; isso ocorre quando o método `viewDidLoad` é chamado. Por esse motivo, vamos criar nosso botão apenas nesse momento, sobrescrevendo o método `viewDidLoad` da classe `ContatosNoMapaViewController`.

```
@IBOutlet weak var mapa: MKMapView!

override func viewDidLoad() {
 super.viewDidLoad()

 let botaoLocalizacao = MKUserTrackingBarButtonItem(mapView: self.mapa)

 self.navigationItem.rightBarButtonItem = botaoLocalizacao
}
```

Mas se tentarmos rodar nossa aplicação, mesmo assim a localização atual não será obtida. Isto acontece pois no iOS 8, para utilizar a localização do usuário, é necessário pedir uma autorização primeiramente. Podemos pedir dois tipos de autorização: buscar a localização somente quando a app estiver em uso, ou buscar a localização mesmo quando a app estiver em *background*. Além disso precisamos cadastrar o pedido de autorização dentro do arquivo `Info.plist`:

No `Info.plist` podemos ter as duas chaves para solicitação de autorização:

- `NSLocationAlwaysUsageDescription` - Para uso em *background*
- `NSLocationWhenInUseUsageDescription` - Para uso enquanto o app estiver aberto

Agora temos que solicitar a autorização dentro do método `viewDidLoad` utilizando a classe `CLLocationManager`. Com ele temos dois métodos para solicitar a autorização:

- `requestAlwaysAuthorization` - Solicita autorização para uso da localização em *background*
- `requestWhenInUseAuthorization` - Solicita autorização para usar a localização enquanto o app estiver aberto

Vamos criar uma propriedade para armazenar o `CLLocationManager` para que possamos solicitar a autorização para usar a localização enquanto o app estiver aberto:

```
@IBOutlet weak var mapa: MKMapView!

let locationManager = CLLocationManager()

override func viewDidLoad() {
```

```

super.viewDidLoad()

self.locationManager.requestWhenInUseAuthorization()

let botaoLocalizacao = MKUserTrackingBarButtonItem(mapView: self.mapa)

self.navigationItem.rightBarButtonItem = botaoLocalizacao
}

```

Perfeito, agora ao rodarmos a aplicação já visualizaremos o botão de localização do usuário. Ao clicarmos, o aparelho buscará a localização atual do usuário e mostrará o ponto.

Se estivermos utilizando o simulador para rodar nossa aplicação qual será a localização atual que o aparelho vai utilizar?

## 15.9 SIMULANDO UMA LOCALIZAÇÃO COM O XCODE

Como estamos utilizando o simulador para rodar a nossa aplicação, precisamos dizer ao aparelho qual a localização que ele deve utilizar quando o aplicativo pedir.

Rode a aplicação, verifique se você está vendo a área de debug do XCode, localize o botão para simular uma localização e escolha uma das definidas.

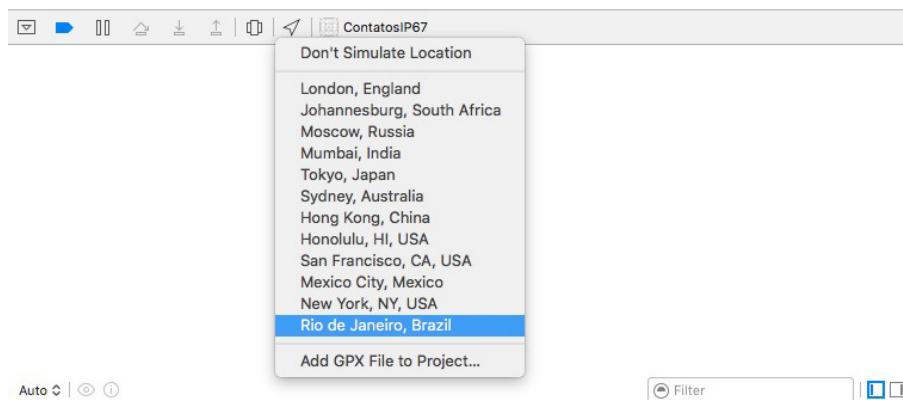


Figura 15.12: Simulando uma localização

Perfeito, ao escolher uma das opções dadas pelo XCode e clicarmos no botão para localizar o usuário, o aplicativo vai pedir permissão para utilizar sua localização atual. Ao aprovar poderemos visualizar nossa localização simulada, neste caso, o Rio de Janeiro.

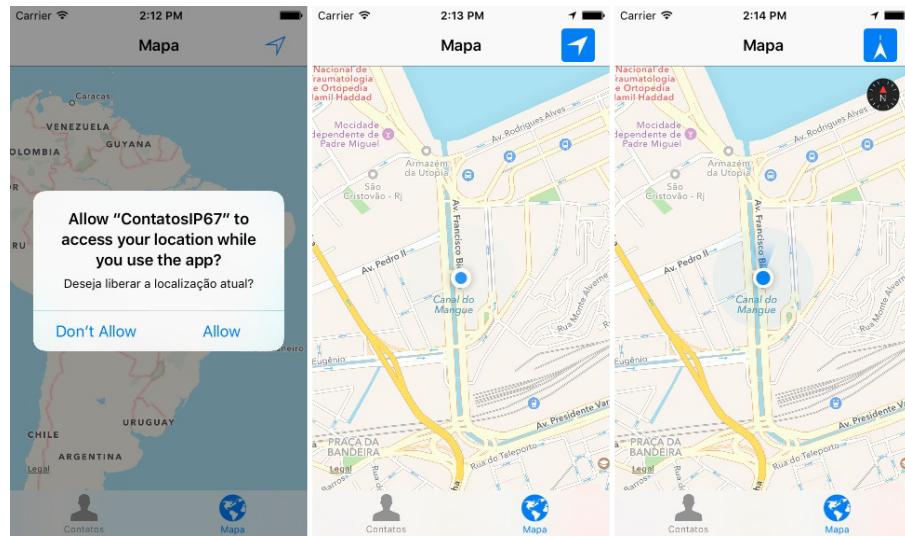


Figura 15.13: Permissão do usuário

Podemos criar uma localização customizada diferente daquelas disponibilizadas pelo XCode.

Vamos criar um novo arquivo *GPX*. Para isso, vá em *File -> New -> File*, selecione o menu de *Resource* na aba de iOS e, então, **GPX File**. Chame-o de **Vila Mariana**.

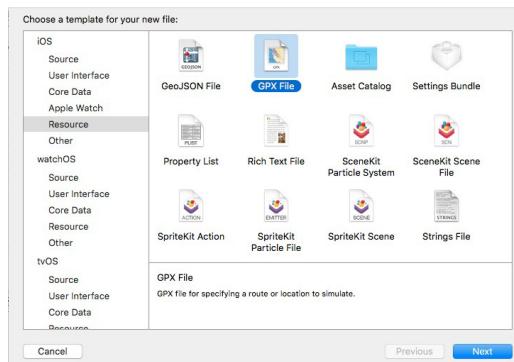


Figura 15.14: Arquivo GPX

Edito o arquivo para conter as coordenadas próximas à Caelum:

```
<?xml version="1.0"?>
<gpx version="1.1" creator="Xcode">
 <wpt lat="-23.588453" lon="-46.632103">
 <name>Vila Mariana</name>
 </wpt>
</gpx>
```

Perfeito, agora podemos selecionar o ponto declarado no arquivo *GPX* como a localização simulada:



Figura 15.15: Arquivo GPX

### ARQUIVOS GPX

Um arquivo **GPX** é um formato padronizado designado para transferir dados de GPS entre aplicações, podendo descrever um ou mais pontos geográficos.

## 15.10 EXERCICIO: LOCALIZANDO O USUÁRIO E ARQUIVO GPX

### 1. Criando um **IBOutlet** para acessar o mapa.

- Utilizaremos a classe *MKUserTrackingBarItem* para mostrar a localização do usuário no mapa, mas primeiramente precisaremos ter acesso ao elemento visual, ou seja, precisaremos de um *@IBOutlet* do tipo *MKMapView*.
- Declare um *@IBOutlet* para acessarmos o mapa por meio do nosso código, chame a propriedade de mapa:

```
@IBOutlet weak var mapa: MKMapView!
```

- Faça o binding entre o *@IBOutlet* e o mapa no storyboard .
- Agora temos acesso ao elemento visual pelo nosso código.

### 2. Criando um **MKUserTrackingBarItem** e adicionando o *navigationItem*.

- Sobrescreva o método *viewDidLoad* da classe *ContatosNoMapaViewController* para criarmos um novo objeto do tipo *MKUserTrackingBarItem* e utilizá-lo na barra de navegação.

```
@IBOutlet weak var mapa: MKMapView!
```

```
override func viewDidLoad() {
```

```

 super.viewDidLoad()

 let botaoLocalizacao = MKUserTrackingBarButtonItem(mapView: self.mapa)

 self.navigationItem.rightBarButtonItem = botaoLocalizacao
 }

```

- Crie uma propriedade para armazenarmos a instância de `CLLocationManager` .

```
let locationManager = CLLocationManager()
```

- Edite o método `viewDidLoad` para que ele solicite a localização atual do usuário.

```

@IBOutlet weak var mapa: MKMapView!

let locationManager = CLLocationManager()

override func viewDidLoad() {
 super.viewDidLoad()

 self.locationManager.requestWhenInUseAuthorization()

 let botaoLocalizacao = MKUserTrackingBarButtonItem(mapView: self.mapa)

 self.navigationItem.rightBarButtonItem = botaoLocalizacao
}

```

3. Ainda com esse código não é possível obter a localização do aparelho do usuário, pois esquecemos de dizer o que aparecerá no alerta de permissão.

Dentro do arquivo `Info.plist`, clique em qualquer parte livre com o botão direito do mouse e em `Add Row`. Na primeira coluna, colocaremos o seguinte valor da nossa chave:

```
NSLocationWhenInUseUsageDescription
```

Na terceira coluna, diremos o valor que nossa chave possuirá:

```
Deseja liberar a localização atual?
```

Manteremos a coluna do meio inalterada, pois o tipo do nosso valor é `string`, o mesmo valor que já está lá.

4. Criando um GPX e simulando a localização do usuário.

- Para simular uma coordenada diferente das que o XCode provê precisamos criar um arquivo `GPX`. Para isso, vá em `File -> New -> File`, na aba `Resource` selecione `GPX`. Chame-o de `Vila Mariana`, altere o arquivo para que ele passe as coordenadas próximas à Caelum.

```

<?xml version="1.0"?>
<gpx version="1.1" creator="Xcode">
 <wpt lat="-23.588453" lon="-46.632103">
 <name>Vila Mariana</name>
 </wpt>
</gpx>

```

- Rode a aplicação e selecione o ponto que declararmos no `GPX`.



Figura 15.16: Ponto Vila Mariana

- Busque a localização do usuário clicando no botão que adicionamos e veja o mapa navegar até o ponto declarado. Se estivéssemos em um *iPhone* ou *iPod* poderíamos utilizar o giroscópio do aparelho para mostrar onde o usuário está apontando.

# BUSCANDO A LOCALIZAÇÃO GEOGRAFICA DO USUÁRIO

Agora que a aplicação possui uma tela para visualizar os contatos no mapa, precisamos transformar as informações de endereço do contato em coordenadas geográficas.

Precisamos adicionar dois novos campos de texto no formulário, um para latitude e outro para longitude, mas ao fazer isso, estaremos infringindo uma das regras da Apple em relação a apresentação de formulários: quando um campo de texto é selecionado, o teclado aparece na nossa frente, se ele esconder alguma informação do formulário a Apple provavelmente não aprovará o aplicativo.

O comportamento que a Apple espera, num momento desses, pode ser ilustrado com a figura:

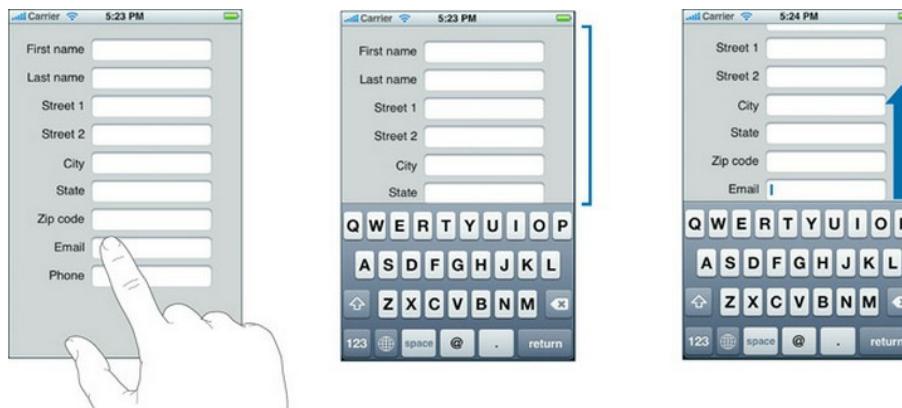


Figura 16.1: Teclado escondendo informações

Perceba que a Apple recomenda que o formulário suba de tal forma que seja possível ver o campo escondido pelo teclado.

Existe um componente nativo que pode ser usado quando precisamos apresentar mais informações do que cabem na tela, como no caso do formulário ilustrado acima. A ideia de um *scroll view* é criar uma view maior que a tela e que pode ser arrastada para cima ou para baixo para revelar o conteúdo invisível.

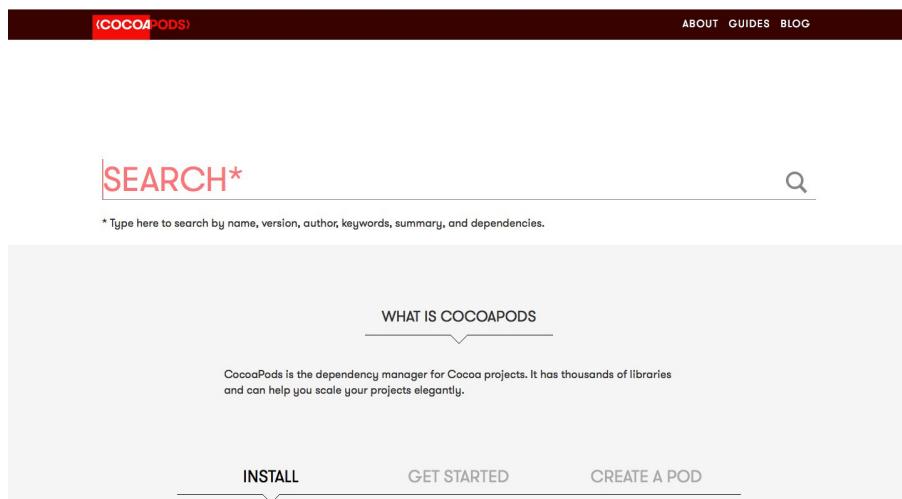
Este componente é o `UIScrollView` mas mesmo utilizando ele, nossa aplicação ainda não estará em conformidade com as regras de usabilidade da *Apple*, pois a única coisa que ele faz é permitir que a tela tenha um *scroll*. Ainda será preciso fazer com que o *scroll* mude de posição automaticamente

quando o teclado aparecer na tela.

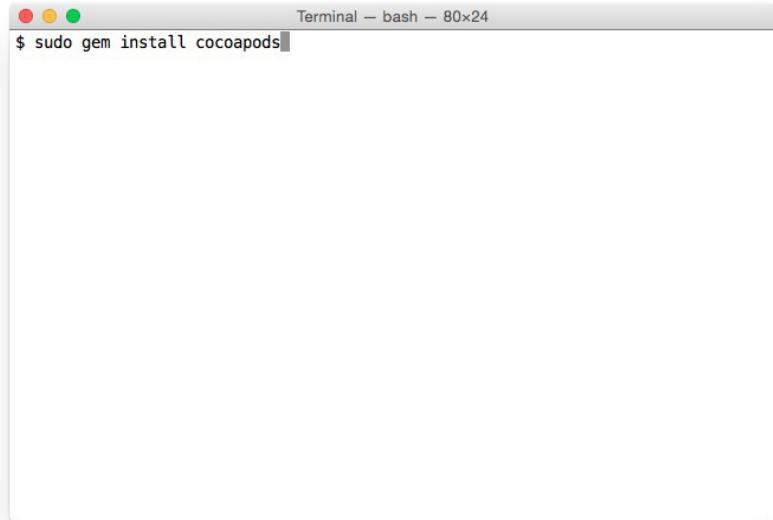
Para termos o *scroll* automático, precisaríamos descobrir sempre que o teclado é exibido ou ocultado, depois teríamos que descobrir qual o campo atual sendo editado para enfim conseguirmos efetuar o *scroll*. Mas o quanto deveríamos *scrollar*? O suficiente para que o campo fique visível na tela, o que envolve uma série de cálculos para conseguir fazer uma funcionalidade que deveria ser bem mais simples. Será que realmente precisamos fazer tudo isso? Isso já deveria estar pronto! Vamos usar alguém que já se preocupou com esta funcionalidade antes! Só precisamos saber como colocar um componente existente em nosso projeto.

## 16.1 GERENCIANDO DEPENDÊNCIAS COM COCOAPODS

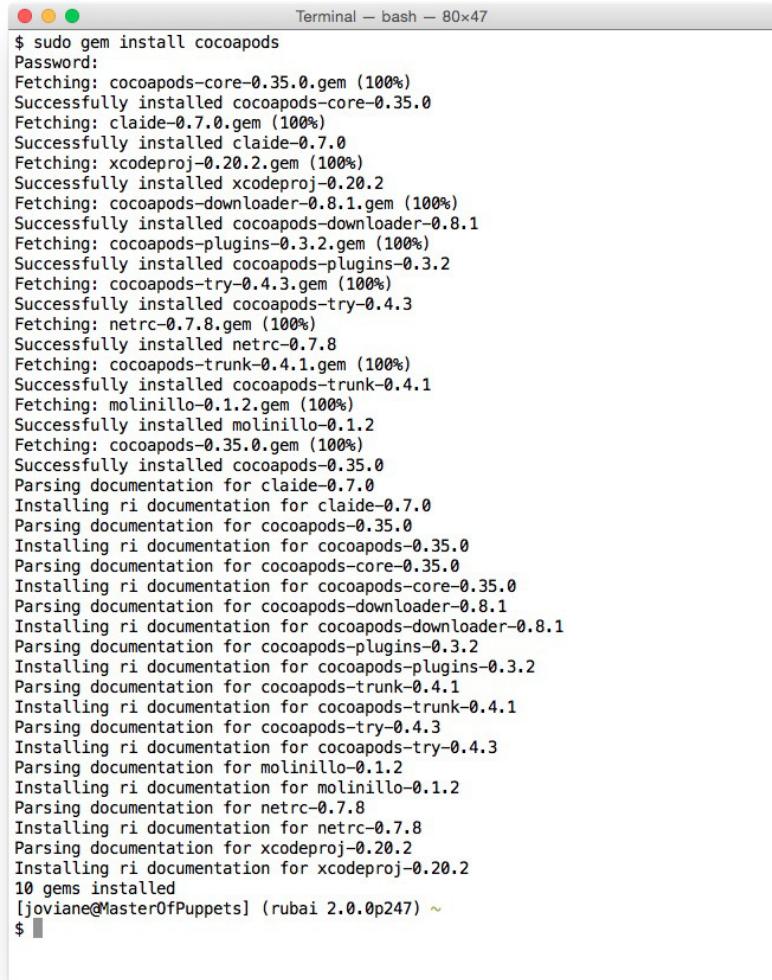
Pensando sempre em reaproveitamento de códigos e funcionalidades, a própria comunidade de Objective-C criou um gerenciador de dependências para conseguirmos usar outras bibliotecas em nossos projetos de uma forma bem mais simples. Este gerenciador é chamado de **CocoaPods** e pode ser acessado pelo site <http://cocoapods.org/>.



Nele existe um catálogo de bibliotecas prontas com um monte de informações sobre elas, como o site, a documentação, qual a versão atual e etc. Para podermos usufruir do **CocoaPods**, precisamos instalá-lo em nosso sistema. Fazemos isto no terminal digitando o comando `sudo gem install cocoapods`.



Será necessário digitar a sua senha para que a instalação seja efetuada. Ao final da instalação, aparecerá a mensagem de sucesso.



```
Terminal — bash — 80x47
$ sudo gem install cocoapods
Password:
Fetching: cocoapods-core-0.35.0.gem (100%)
Successfully installed cocoapods-core-0.35.0
Fetching: claide-0.7.0.gem (100%)
Successfully installed claide-0.7.0
Fetching: xcodeproj-0.20.2.gem (100%)
Successfully installed xcodeproj-0.20.2
Fetching: cocoapods-downloader-0.8.1.gem (100%)
Successfully installed cocoapods-downloader-0.8.1
Fetching: cocoapods-plugins-0.3.2.gem (100%)
Successfully installed cocoapods-plugins-0.3.2
Fetching: cocoapods-try-0.4.3.gem (100%)
Successfully installed cocoapods-try-0.4.3
Fetching: netrc-0.7.8.gem (100%)
Successfully installed netrc-0.7.8
Fetching: cocoapods-trunk-0.4.1.gem (100%)
Successfully installed cocoapods-trunk-0.4.1
Fetching: molinillo-0.1.2.gem (100%)
Successfully installed molinillo-0.1.2
Fetching: cocoapods-0.35.0.gem (100%)
Successfully installed cocoapods-0.35.0
Parsing documentation for claide-0.7.0
Installing ri documentation for claide-0.7.0
Parsing documentation for cocoapods-0.35.0
Installing ri documentation for cocoapods-0.35.0
Parsing documentation for cocoapods-core-0.35.0
Installing ri documentation for cocoapods-core-0.35.0
Parsing documentation for cocoapods-downloader-0.8.1
Installing ri documentation for cocoapods-downloader-0.8.1
Parsing documentation for cocoapods-plugins-0.3.2
Installing ri documentation for cocoapods-plugins-0.3.2
Parsing documentation for cocoapods-trunk-0.4.1
Installing ri documentation for cocoapods-trunk-0.4.1
Parsing documentation for cocoapods-try-0.4.3
Installing ri documentation for cocoapods-try-0.4.3
Parsing documentation for molinillo-0.1.2
Installing ri documentation for molinillo-0.1.2
Parsing documentation for netrc-0.7.8
Installing ri documentation for netrc-0.7.8
Parsing documentation for xcodeproj-0.20.2
Installing ri documentation for xcodeproj-0.20.2
10 gems installed
[joviane@MasterOfPuppets] (rubai 2.0.0p247) ~
$
```

Após a instalação, precisamos completar o setup do **CocoaPods** executando ainda no terminal, o comando `pod setup`. Este comando irá colocar as referências para todos os *pods*, como são chamadas as bibliotecas gerenciadas pelo **CocoaPods**, em nossa máquina.

Com o **CocoaPods** instalado e configurado, precisamos agora dizer que queremos utilizá-lo em nosso projeto. Fazemos isso novamente pelo terminal. Devemos entrar no diretório de nosso projeto, e executar o comando `pod init`. Este comando criará um arquivo chamado *Podfile*, que é onde colocaremos os *pods* que queremos utilizar em nosso projeto.

```

Terminal - bash - 80x24
$ cd caelum/iOS/ContatosIP67/
[joviane@MasterOfPuppets] (rubai 2.0.0p247) ~/caelum/iOS/ContatosIP67
$ pod init
[joviane@MasterOfPuppets] (rubai 2.0.0p247) ~/caelum/iOS/ContatosIP67
$ ls
ContatosIP67 ContatosIP67Tests
ContatosIP67.xcodeproj Podfile
[joviane@MasterOfPuppets] (rubai 2.0.0p247) ~/caelum/iOS/ContatosIP67
$

```

Pronto, agora podemos utilizar a biblioteca que fará o *scroll* do teclado.

## 16.2 INSTALANDO TPKEYBOARDVOIDING

Um dos *pods* mais utilizados no desenvolvimento *iOS* é o **TPKeyboardAvoiding**. Ele faz exatamente o que precisamos, o *scroll* do teclado automático e também já transforma nossa *view* em uma *scroll view*.

Para utilizá-lo devemos editar o arquivo *Podfile* e acrescentar dentro do bloco que contém o nome de nosso projeto, a linha `pod "TPKeyboardAvoiding"`, que é o *pod* que queremos utilizar. O *Podfile* deve ficar da seguinte forma:

```

Uncomment this line to define a global platform for your project
platform :ios, "6.0"

target "ContatosIP67" do
 pod "TPKeyboardAvoiding"
end

target "ContatosIP67Tests" do
end

```

Feito isso, pedimos para o **CocoaPods** buscar em seu repositório a versão mais recente do *pod* e instalá-lo em nosso projeto, através do comando `pod install`.

```
Terminal - bash - 80x24
$ pod install
Analyzing dependencies

CocoaPods 0.36.0.beta.2 is available.
To update use: sudo gem install cocoapods
Downloading dependencies
Installing TPKeyboardAvoiding (1.2.6)
Generating Pods project
Integrating client project

[!] From now on use `ContatosIP67.xcworkspace`.
[joviane@MasterOfPuppets] (rubai 2.0.0p247) ~/caelum/iOS/ContatosIP67
$
```

A partir de agora, sempre que quisermos abrir um projeto que utiliza o **CocoaPods** devemos abri-lo pelo arquivo `ContatosIP67.xcworkspace` ao invés do `ContatosIP67.xcproject`, então será necessário fechar o projeto e abrir novamente pelo arquivo correto.

Toda vez que quisermos utilizar um *pod* novo, só precisamos editar o arquivo *Podfile* do projeto e em seguida executar o comando `pod install`.

Agora que instalamos o *pod*, vamos utilizá-lo em nosso projeto.

## 16.3 MELHORANDO NOSSO FORMULÁRIO COM TPKEYBOARDVOIDINGSCROLLVIEW

A classe que possibilita que a nossa *view* tenha o *scroll* automático é a `TPKeyboardAvoidingScrollView`. O primeiro fato importante a saber é que esta classe herda de `UIScrollView` que por sua vez, herda de `UIView`. Isso quer dizer que uma instância desse tipo pode ser associada como a propriedade *view* de qualquer *view controller*.

Como o *Identity Inspector* permite alterar a classe de determinado componente no próprio *Interface Builder*, faremos exatamente isso com a *view* do formulário. Vamos alterar sua classe para `TPKeyboardAvoidingScrollView`. Dessa forma, o componente visual que abriga todos os campos de formulário passa a ser uma instância de `TPKeyboardAvoidingScrollView`.

Com isso, ganhamos a habilidade de adicionar mais componentes do que é possível visualizar em uma tela do dispositivo. Ganhamos também, o tratamento para eventos de *scroll* e além de tudo, ainda ganhamos o *scroll* automático ao trocarmos de campo! Tudo isso sem precisarmos fazer um monte de

cálculos!

## 16.4 EXERCÍCIO: UTILIZANDO COCOAPODS E TPKEYBOARDVOIDING

1. Como o CocoaPods já está instalado em nossas máquinas, somente é necessário efetuar o setup dele. Para isto, no terminal (representado pelo símbolo \$) digite:

```
$ pod setup
```

2. Ainda pelo terminal, entre no diretório de seu projeto e inicialize o CocoaPods digitando:

```
$ pod init
```

Será criado neste momento o arquivo *Podfile*

3. Edite o arquivo *Podfile* e acrescente que queremos instalar o *pod* do `TPKeyboardAvoiding` dentro do target com o nome de nosso projeto. O arquivo deverá ficar da seguinte forma:

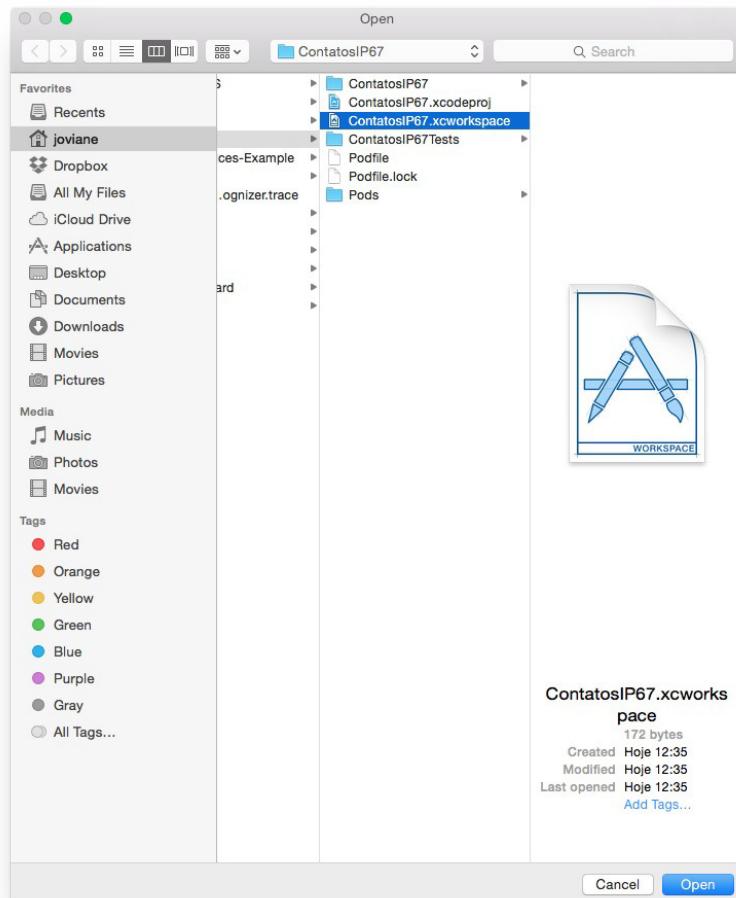
```
target "ContatosIP67" do
 pod "TPKeyboardAvoiding"
end
```

4. No terminal novamente, devemos pedir para que o CocoaPods instale o *pod* que acabamos de informar no *Podfile*. Fazemos isto executando o comando:

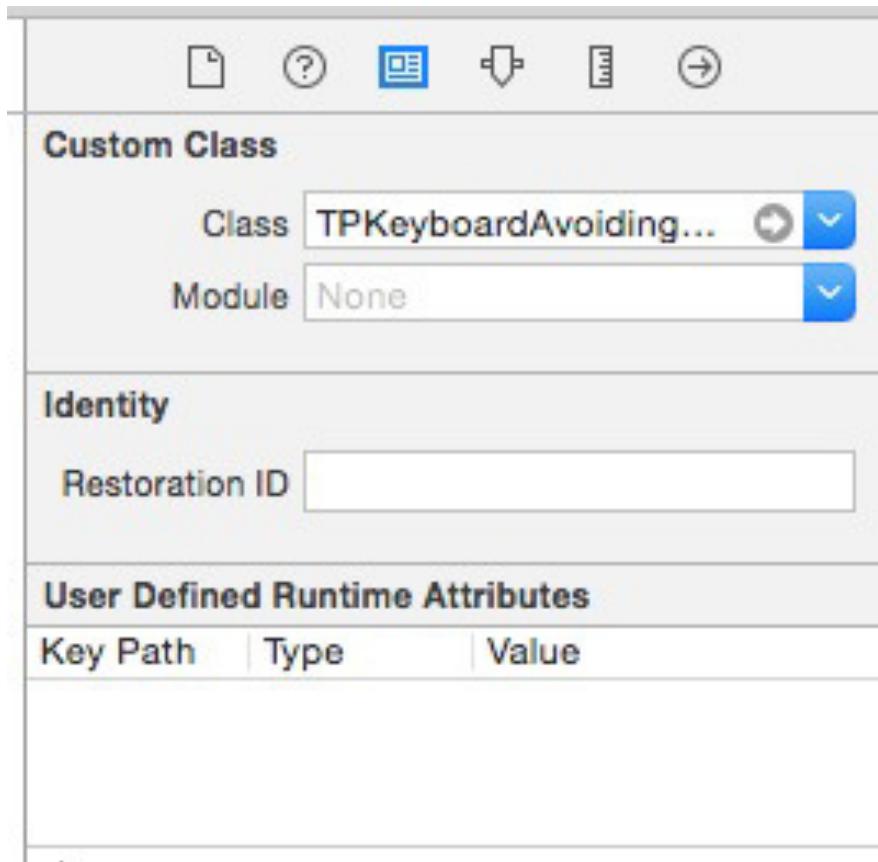
```
$ pod install
```

5. Agora que temos o *pod* instalado, vamos reabrir nosso projeto, desta vez utilizando o arquivo com a extensão `.xcworkspace`.

- No Xcode, vá em *File -> Close Project* para fechar o projeto atual
- Agora vamos reabri-lo, vá novamente em *File -> Open* aparecerá a tela com os arquivos que podem ser abertos.



- Selecione o arquivo `ContatosIP67.xcworkspace`. Pronto agora o projeto está aberto corretamente.
6. Vamos utilizar o `TPKeyboardAvoiding` em nosso formulário. No arquivo `Main.storyboard`, clique sobre a tela do formulário e vá no menu *Identity Inspector*. Nele existe uma seção chamada *Custom class*. No campo *Class* digite `TPKeyboardAvoidingScrollView` e aperte Enter.



7. Execute a aplicação. Troque de um campo para o outro e verifique que o *scroll* está funcionando corretamente.

## 16.5 BUSCANDO AS COORDENADAS DO USUÁRIO A PARTIR DO SEU ENDEREÇO

Perfeito, agora que o nosso formulário se comporta como Apple manda, podemos prosseguir para o objetivo principal: encontrar a localização dos contatos e visualizá-los no mapa.

O processo de encontrar as coordenadas geográficas a partir de um endereço chama-se **geocode**, e a Apple deixou tudo pronto em um **framework** - o *CoreLocation*.

O *CoreLocation* cuida de tudo o que envolva coordenadas, GPS e georreferenciamento de dados.

Ao importá-lo no projeto, teremos disponível a classe **CLGeocoder** - com ela podemos geocodificar um endereço para encontrar a latitude e longitude que ele representa.

## GEOCODE REVERSO

A classe `CLGeocoder` do framework `CoreLocation` é capaz encontrar as coordenadas geográficas a partir um endereço e também o contrário - encontrar o endereço que coordenadas geográficas representam. Para saber mais, veja a documentação da Apple.

A classe `CLGeocoder` é bem simples: para realizar um geocode basta instanciá-la e invocar o método `geocodeAddressString:completionHandler:` que recebe uma `String` como argumento, que será utilizado para encontrar um ou mais pontos geográficos que ele representa.

O `completionHandler`: é um *closure* que recebe os seguintes parâmetros:

- `[CLPlacemark]?` - Um array opcional de objetos do tipo `CLPlacemark`
- `Error?` - Um objeto que representa o erro também opcional

Passaremos um trecho de código, um *closure*, onde trataremos o retorno do geocode, receberemos os endereços que forem encontrados e, caso aconteça um erro, também teremos como tratá-lo.

Este trecho de código (*closure*) será executado apenas quando a operação for finalizada.

Veja como vamos utilizar um bloco para fazer o geocode:

```
let geocoder = CLGeocoder()

geocoder.geocodeAddressString("Rua Vergueiro, 3185, São Paulo, São Paulo, Brasil")
{ (resultados, error) in
 // aqui vamos tratar o retorno do geocode, podemos pensar nesse trecho
 // de código como um método, pois ele será executado apenas no final da operação.
}
```

Nós recebemos, no argumento do *closure*, o resultado da operação - um `Array` com as informações do geocode. Recebemos, também, um argumento dizendo se um erro ocorreu ou não, podendo tratá-lo caso algo dê errado.

Se tudo estiver correto e o endereço passado seja encontrado, diversas instâncias do tipo `CLPlacemark` serão passadas dentro do `Array` que recebemos no *closure*.

A classe `CLPlacemark` possui informações sobre o endereço buscado - como latitude e longitude.

```
@IBAction func buscaCoordenadas(sender: UIButton) {

 let geocoder = CLGeocoder()

 geocoder.geocodeAddressString(self.endereco.text!) { (resultado, error) in

 if error == nil && resultado?.count > 0 {
 let placemark = resultado![0]
 let coordenada = placemark.location!.coordinate
```

```

 print("latitude: \u201c(coordenada.latitude), longitude: \u201c(coordenada.longitude)\u201d)

 }

}

}

```

Altere o formulário de contatos para que, a partir do clique de um botão, faça o geocode do endereço digitado pelo usuário.

Altere o formulário de cadastro de contatos na *storyboard* da seguinte forma:

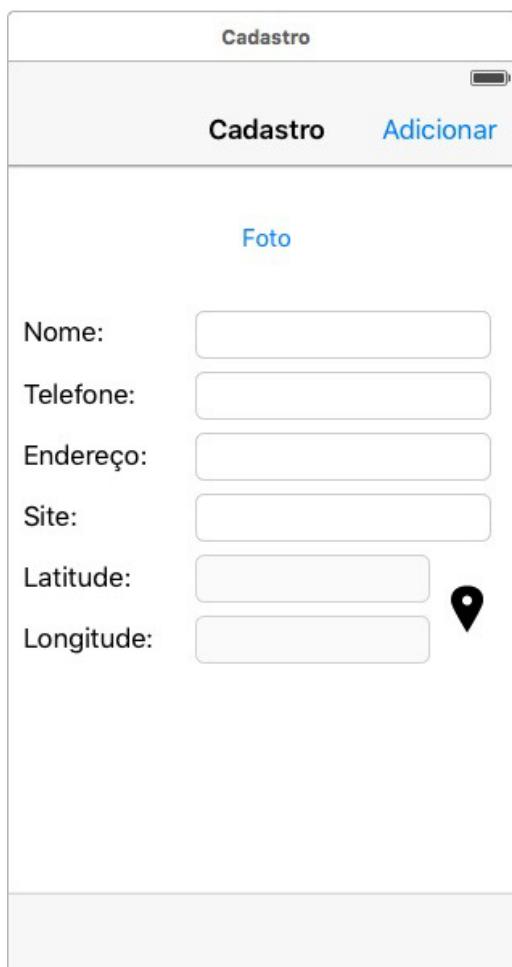


Figura 16.9: Botão geocode

Percebeu que o botão está com uma cara diferente? Nós alteramos o seu ícone e podemos fazer isso diretamente no *Interface Builder*:

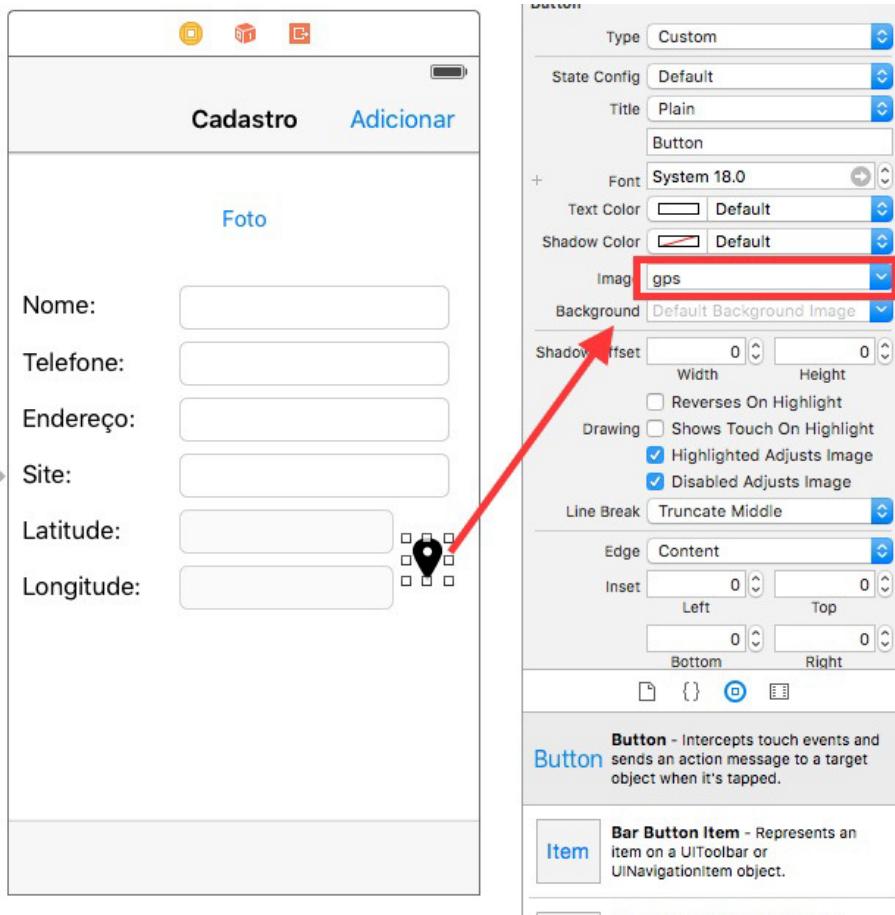


Figura 16.10: Customizando o ícone e um botão

Para efetuar o geocode, declare um `@IBAction` para o clique do botão, lembre-se de declarar o método e fazer a conexão no *storyboard*.

Ao efetuar o geocode, atualize o formulário com os dados encontrados.

```
@IBAction func buscaCoordenadas(sender: UIButton) {
 let geocoder = CLGeocoder()
 geocoder.geocodeAddressString(self.endereco.text!) { (resultado, error) in
 if error == nil && resultado?.count > 0 {
 let placemark = resultado![0]
 let coordenada = placemark.location!.coordinate
 self.latitude.text = coordenada.latitude.description
 self.longitude.text = coordenada.longitude.description
 }
 }
}
```

## 16.6 EXERCICIO: FAZENDO GEOCODE DO ENDEREÇO DO USUÁRIO

1. Alterando o formulário para realizar o geocode.

- Clique no arquivo **Assets.xcassets**
- Clique no botão + e selecione a opção **Import**
- Copie para a aplicação o ícone para customizar o botão e clique em **Open**:
- **gps.png**
- **gps@2x.png**
- A partir de agora, a imagem estará disponível para utilizarmos. Adicione um botão no formulário e altere suas configurações para exibir a imagem que acabamos de copiar.

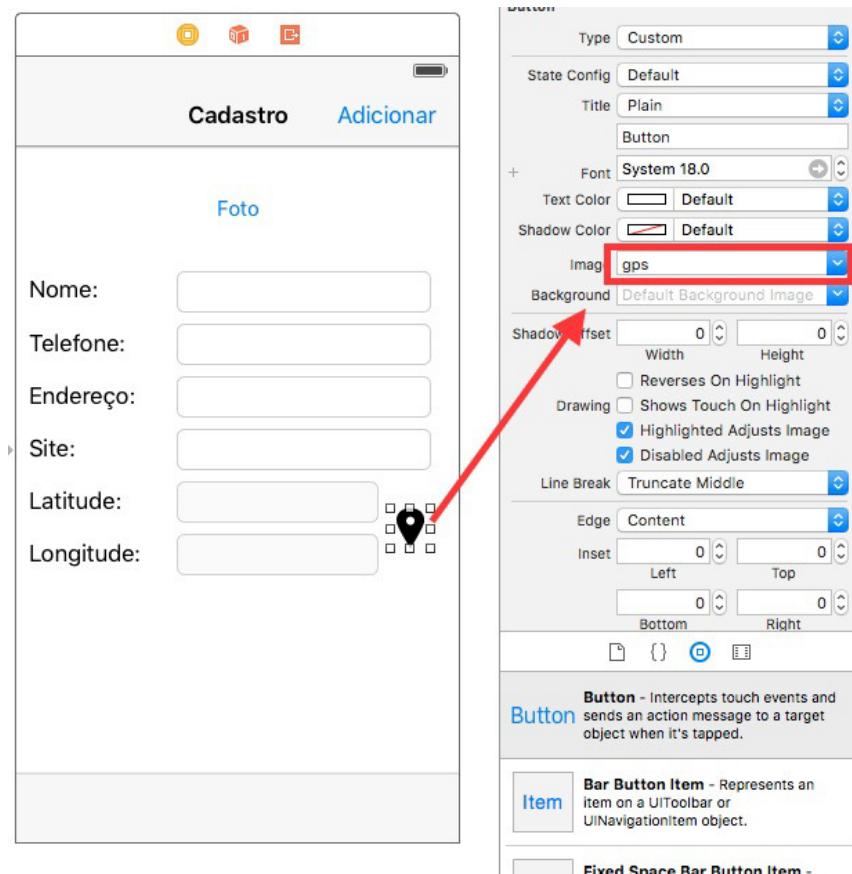


Figura 16.11: Customizando o ícone e um botão

- Declare um `@IBAction` com o nome `buscarCoordenadas:` para efetuar o geocode quando o usuário pressionar o botão, lembre-se de conectar tudo corretamente.

2. Efetuando o geocode a partir do endereço do usuário.

- Importe o framework `CoreLocation` no *FormularioContatoViewController*:

```

import CoreLocation

◦ No @IBAction declarado faça o geocode do endereço do contato, seria interessante validar se o
usuário entrou com a informação de endereço ou não.

@IBAction func buscaCoordenadas(sender: UIButton) {

 let geocoder = CLGeocoder()

 geocoder.geocodeAddressString(self.endereco.text!) { (resultado, error) in

 if error == nil && resultado?.count > 0 {
 let placemark = resultado![0]
 let coordenada = placemark.location!.coordinate

 self.latitude.text = coordenada.latitude.description
 self.longitude.text = coordenada.longitude.description
 }
 }
}

```

### 3. Armazenando as coordenadas do endereço no Contato.

- Adicione duas novas *properties* para guardar as informações de latitude e longitude no contato.

```

//Contato.h
#import <UIKit/UIKit.h>
#import <Foundation/Foundation.h>
@interface Contato : NSObject

@property (strong) NSString *nome;
@property (strong) NSString *endereco;
@property (strong) NSString *telefone;
@property (strong) NSString *site;
@property (strong) UIImage *foto;
//novas propriedades
@property (strong) NSNumber *latitude;
@property (strong) NSNumber *longitude;

@end

```

- Com as novas propriedades, altere o formulário para passar as informações de latitude e longitude contidas no formulário para o contato.

Para isso altere o método `pegaDadosDoFormulario` da classe **FormularioContatoViewController**.

```

func pegaDadosDoFormulario(){
 if contato == nil {
 self.contato = Contato()
 }

 if self.botaoAdcionaImage.backgroundImageForState(.Normal) != nil {
 self.contato.foto = self.botaoAdcionaImage.backgroundImageForState(.Normal)
 }
}

```

```

 }

 self.contato.nome = self.nome.text!
 self.contato.telefone = self.telefone.text!
 self.contato.endereco = self.endereco.text!
 self.contato.site = self.site.text!

 if let latitude = Double(self.latitude.text!) {
 self.contato.latitude = latitude as NSNumber
 }

 if let longitude = Double(self.longitude.text!) {
 self.contato.longitude = longitude as NSNumber
 }

}

```

- Precisamos alterar a forma com que o formulário é preenchido quando um contato está sendo editado. Para isso vamos alterar o arquivo `FormularioContatoViewController` e preencher o campo de latitude e longitude assim que a tela estiver carregada.

```

override func viewDidLoad() {
 super.viewDidLoad()

 if contato != nil {
 self.nome.text = contato.nome
 self.telefone.text = contato.telefone
 self.endereco.text = contato.endereco
 self.site.text = contato.site
 self.longitude.text = contato.longitude?.description
 self.latitude.text = contato.latitude?.description

 if let foto = contato.foto {
 self.imageView.image = foto
 }

 let botaoAlterar = UIBarButtonItem(title: "Confirmar", style: .plain, target: self, action: #selector(atualizar))

 self.navigationItem.rightBarButtonItem = botaoAlterar
 }
}

```

- Rode a aplicação e faça o geocode de alguns contatos cadastrados, é necessário preencher os dados corretos do endereço, contendo a cidade.

#### VALIDANDO OS DADOS DO USUÁRIO

Seria interessante validar se o usuário preencheu os dados do endereço antes de fazer o geocode, mostrando um alerta caso não tenham sido preenchidos.

## 16.7 MELHORANDO A USABILIDADE DO USUÁRIO COM UIACTIVITYINDICATORVIEW

## UIACTIVITYINDICATORVIEW

Perceba que a operação de buscar as coordenadas geográficas do endereço não é efetuada imediatamente quando clicamos no botão.

Tudo é feito de forma *assíncrona*, ou seja, a resposta da operação de geocode não é retornada logo quando chamamos o método `geocodeAddressString:completionHandler:`, justamente para não forçar o usuário a esperar o retorno da operação.

Porém seria interessante dar um *feedback* para o usuário que algo está acontecendo, pois ao clicar no botão e não ver nenhuma alteração visual o comportamento esperado do usuário seria apertar novamente o botão para ver se algo acontece.

No mundo web um elemento visual ficou muito conhecido por dar a sensação de que algo ainda está sendo efetuado, os *spinners*, e podemos encontrá-los em diversos sites onde operações são efetuadas e queremos dizer para o usuário que algo ainda está sendo feito, evitando que quem utiliza o sistema tente fazer a operação novamente.

A Apple incorporou um *spinner* pronto para ser utilizado dentro de nossas aplicações e é representado pela classe `UIActivityIndicatorView` que pode nos ajudar a dar um *feedback* interessante pro usuário.

Vamos arrastar um novo componente visual em cima do botão de geocode e animá-lo assim que a operação começar.

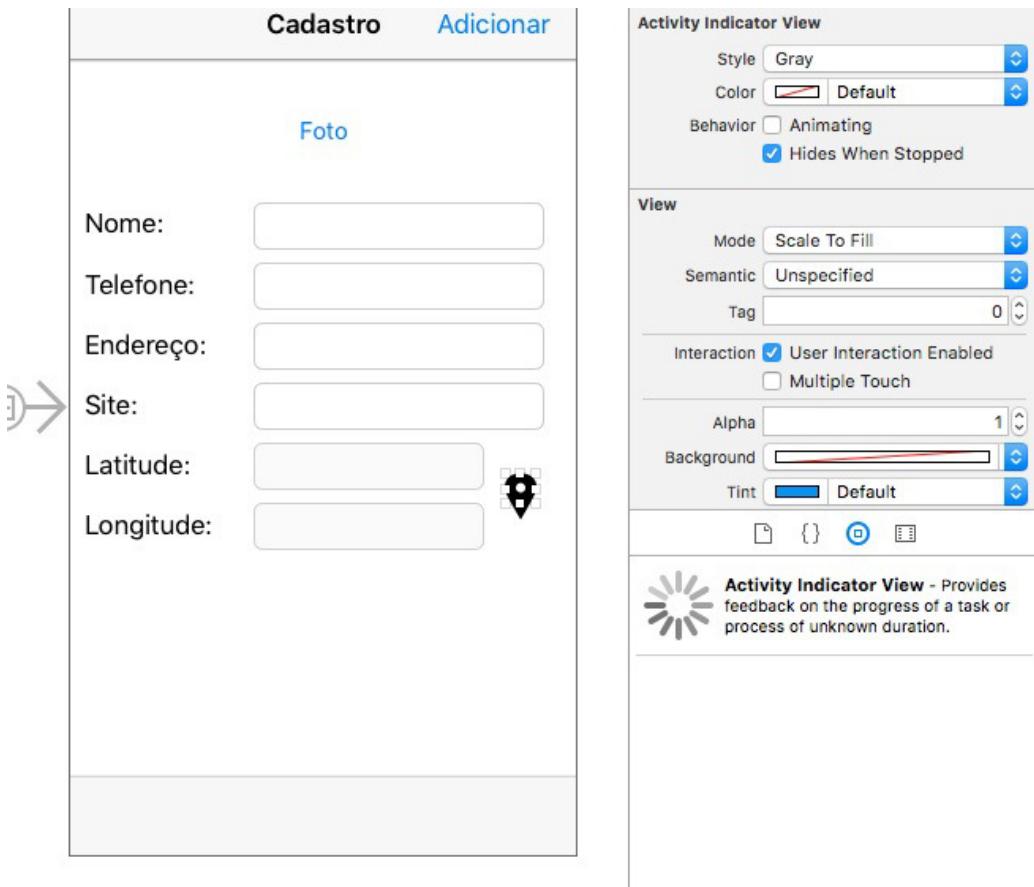


Figura 16.12: UIActivityIndicatorView

Perceba que alteraremos a propriedade *Hides When Stopped* do elemento para que ele se esconda automaticamente quando não estiver se movimentando.

Para conseguir animar o elemento assim que a operação de geocode for iniciada, precisaremos do acesso ao elemento visual, ou seja, temos criar um *IBOutlet* para ele:

```
@IBOutlet weak var loading: UIActivityIndicatorView!
```

E logo que o geocode for iniciado, ou seja, quando o usuário clicar no botão, vamos animá-lo utilizando o método *startAnimating*.

```
@IBAction func buscaCoordenadas(sender: UIButton) {
 self.loading.startAnimating()
 let geocoder = CLGeocoder()
 // restante da implementação
}
```

Para que o *spinner* pare e se esconda vamos chamar um outro método - *stopAnimating*, e como marcamos a opção para que ele se esconda quando estiver parado, o elemento ficará invisível automaticamente. Porém chamaremos este método apenas quando a operação terminar, isto é, dentro do *closure* - quando recebermos os dados da operação.

```

@IBAction func buscaCoordenadas(sender: UIButton) {

 self.loading.startAnimating()

 let geocoder = CLGeocoder()

 geocoder.geocodeAddressString(self.endereco.text!) { (resultado, error) in

 if error == nil && resultado?.count > 0 {
 let placemark = resultado![0]
 let coordenada = placemark.location!.coordinate

 self.latitude.text = coordenada.latitude.description
 self.longitude.text = coordenada.longitude.description
 }

 self.loading.stopAnimating()
 }
}

```

Podemos também bloquear o botão para que não seja possível repetir a operação a menos que tenha terminado o processo de geocode. Como nosso `@IBAction` está associado à um botão podemos pegar a referencia ao componente `UIButton` a partir do `sender`. Com essa referencia podemos desabilitar o botão ao iniciar o processo de geocode e habilitar ao termino do processo:

```

@IBAction func buscaCoordenadas(sender: UIButton) {

 self.loading.startAnimating()
 sender.isEnabled = false

 let geocoder = CLGeocoder()

 geocoder.geocodeAddressString(self.endereco.text!) { (resultado, error) in

 if error == nil && resultado?.count > 0 {
 let placemark = resultado![0]
 let coordenada = placemark.location!.coordinate

 self.latitude.text = coordenada.latitude.description
 self.longitude.text = coordenada.longitude.description
 }

 self.loading.stopAnimating()
 sender.isEnabled = true
 }
}

```

## 16.8 EXERCÍCIO: ADICIONANDO UM SPINNER AO EFETUAR O GEOCODE

1. Utilizando o *Interface Builder*, edite o arquivo `Main.storyboard` e adicione um *Activity Indicator*

*View* logo em cima do botão de geocode. Vamos animá-lo assim que a operação começar.

Lembre-se de marcar a opção *Hides When Stopped* para que o elemento fique invisível enquanto não estiver animando.

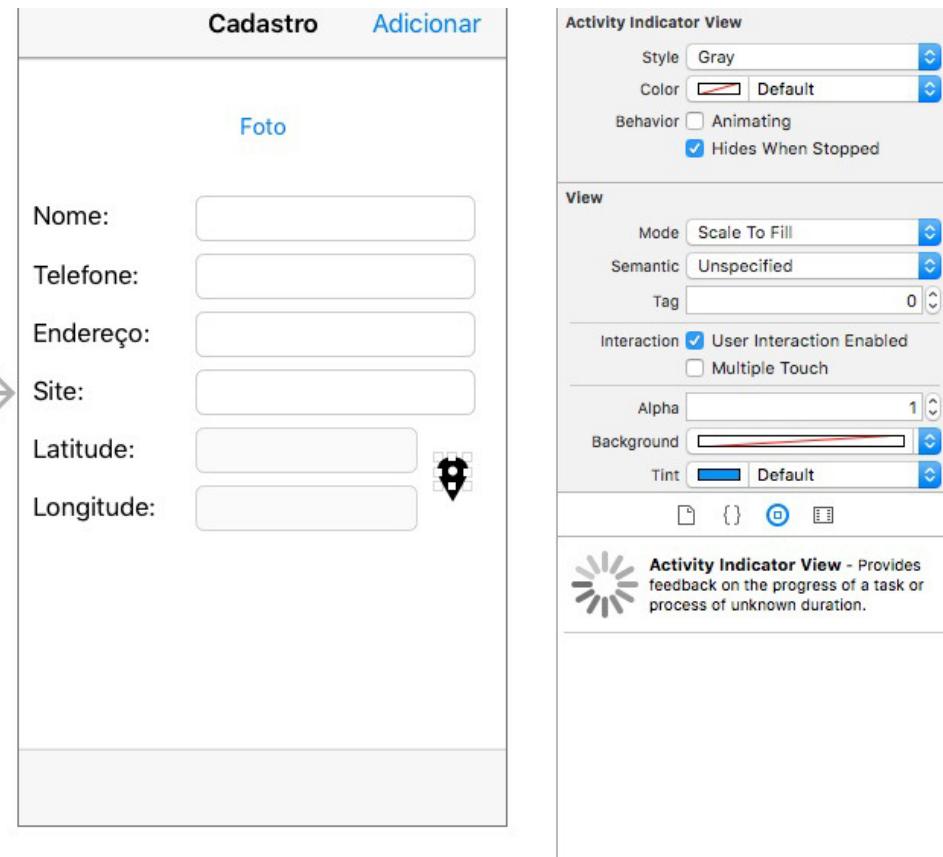


Figura 16.13: UIActivityIndicatorView

- Para conseguir animá-lo precisamos acessar a sua referência, logo, um novo *IBOutlet*. Lembre-se de conectar corretamente.

```
@IBOutlet weak var loading: UIActivityIndicatorView!
```

- Faça a chamada do método *startAnimating* na referência para o *UIActivityIndicatorView* que acabamos de criar, assim que a operação de geocode for iniciada e assim que a operação terminar faça com que o *spinner* pare de se animar chamando o método *stopAnimating*.

```
@IBAction func buscaCoordenadas(sender: UIButton) {

 self.loading.startAnimating()
 sender.isEnabled = false

 let geocoder = CLGeocoder()

 geocoder.geocodeAddressString(self.endereco.text!) { (resultado, error) in

 if error == nil && resultado?.count > 0 {

 }
}
```

```

let placemark = resultado![0]
let coordenada = placemark.location!.coordinate

self.latitude.text = coordenada.latitude.description
self.longitude.text = coordenada.longitude.description
}

self.loading.stopAnimating()
sender.isEnabled = true

}

}

```

## 16.9 VISUALIZANDO OS CONTATOS NO MAPA

O aplicativo de mapas do iOS exibe informações como pinos e, com os dados de latitude e longitude do nosso contato, podemos fazer o mesmo.

Veja como o aplicativo nativo faz:

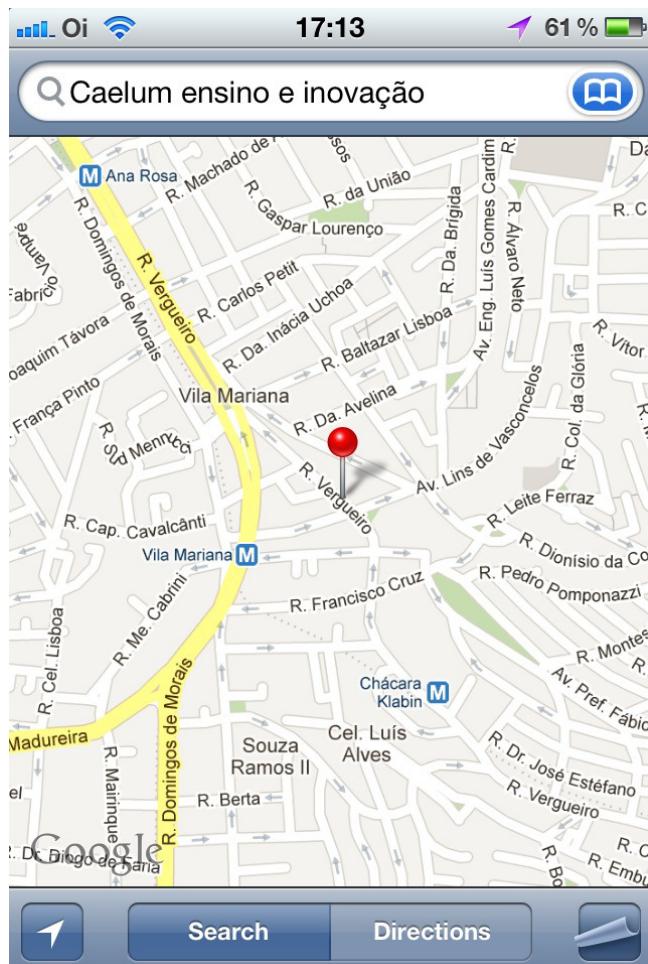


Figura 16.14: Localização Caelum SP

Vamos pensar: quais as informações básicas que um pino precisa para aparecer no mapa? Apenas a

coordenada geográfica, concorda? Se quisermos adicionar um contato no mapa, vamos precisar dizer para o mapa qual a coordenada que o nosso contato representa.

Para o mapa, não importa qual o tipo de informação que ele vai exibir, a única coisa que ele precisa saber é se o objeto possui uma coordenada geográfica e o resto será feito.

O protocolo **MKAnnotation** serve exatamente para isso, certificar-se que o objeto exibido possui uma coordenada.

Vamos dar uma olhada nele:

```
public protocol MKAnnotation : NSObjectProtocol {

 // Center latitude and longitude of the annotation view.
 // The implementation of this property must be KVO compliant.
 public var coordinate: CLLocationCoordinate2D { get }

 // Title and subtitle for use by selection UI.
 optional public var title: String? { get }

 optional public var subtitle: String? { get }
}
```

Perceba que ele nos obriga a implementar um *getter* para a *@property* `coordinate`.

Vamos fazer com que nosso contato siga este protocolo. Para isso, basta alterar a declaração da classe **Contato** fazendo com que ele siga esse contrato.

```
#import <Foundation/Foundation.h>
#import <MapKit/MKAnnotation.h>
@interface Contato : NSObject <MKAnnotation>

@property (strong) NSString *nome;
@property (strong) NSString *endereco;
@property (strong) NSString *telefone;
@property (strong) NSString *site;
@property (strong) UIImage *foto;
@property (strong) NSNumber *latitude;
@property (strong) NSNumber *longitude;

@end
```

Ao compilar o projeto teremos um **warning** gerado por este contrato, pois não implementamos algo que foi declarado com *requirido*(que não foi marcado como *optional*). Não estamos seguindo o que o contrato manda.

Precisamos então fazer com que a classe de contato faça tudo que o protocolo declara, precisamos criar uma localização geográfica a partir das informações de latitude e longitude que o contato já possuí. Utilizaremos a função `CLLocationCoordinate2DMake` que recebe dois atributos do tipo `double` e nos retorna um `CLLocationCoordinate2D` que representa exatamente a coordenada do contato.

```
#import "Contato.h"

@implementation Contato
```

```

- (CLLocationCoordinate2D)coordinate {
 return CLLocationCoordinate2DMake(
 [self.latitude doubleValue],
 [self.longitude doubleValue]
);
}

@end

```

Legal, agora nosso contato sabe prover as informações que o mapa precisa para apresentá-lo.

Vamos tratar de adicionar os contatos no mapa e, para isso, a classe `MKMapView` nos provê alguns métodos:

```

open func addAnnotation(_ annotation: MKAnnotation)

open func addAnnotations(_ annotations: [MKAnnotation])

open func removeAnnotation(_ annotation: MKAnnotation)

open func removeAnnotations(_ annotations: [MKAnnotation])

open var annotations: [MKAnnotation] { get }

```

Perceba que podemos adicionar qualquer objeto que siga o protocolo `MKAnnotation`, essa é uma forma interessante de não se acoplar a um objeto e sim a um contrato. Dessa forma, qualquer objeto que siga o protocolo é capaz de ser adicionado no mapa, e foi o que fizemos com o **Contato**.

Vamos adicionar um novo atributo na classe `ContatosNoMapaViewController` para guardar a referência de todos os contatos cadastrados no aplicativo:

```

import UIKit
import MapKit

class ContatosNoMapaViewController: UIViewController {

 @IBOutlet weak var mapa: MKMapView!

 let locationManager = CLLocationManager()

 let contatos: [Contato] = Array()

}

```

## 16.10 QUAL O MELHOR MOMENTO PARA ADICIONAR OS CONTATOS NO MAPA?

Em qual momento vamos adicionar os contatos no mapa? Temos que nos preocupar com algumas coisas, por exemplo, quando um contato é adicionado ou removido? Como fazer com que isso reflita no mapa?

A forma mais simples é adicionar todos os contatos no mapa quando a tela ficar visível e os remover quando a tela deixar de ser exibida. Dessa forma, não teremos que tomar conta quando um contato for

alterado e/ou removido, pois sempre estaremos vendo os dados atualizados.

Para isso, vamos sobrescrever o método `viewWillAppear:` e `viewWillDisappear:` da classe `ContatosNoMapaViewController` para sempre adicionar e remover os contatos quando a tela for exibida e escondida.

```
override func viewWillAppear(animated: Bool) {
 self.mapa.addAnnotations(self.contatos)
}

override func viewWillDisappear(animated: Bool) {
 self.mapa.removeAnnotations(self.contatos)
}
```

#### DIFERENÇA ENTRE VIEWDIDLAD E VIEWWILLAPPEAR

Quando sobrescrevemos o método `viewDidLoad`: ele é chamado apenas quando a nossa `UIViewController` é carregada, diferentemente do método `viewWillAppear`: e `viewWillDisappear`, esses métodos são chamados todas as vezes que a tela é apresentada e sempre que ela é escondida, respectivamente. Portanto quando o usuário alternar entre telas utilizando a `UITabBar` os métodos serão chamados.

Agora, temos que preencher nosso array com todos os contatos que temos no nosso Dao e passarmos para a propriedade `contatos`. Vamos fazer isso dentro do método `viewWillAppear`:

```
import UIKit
import MapKit

class ContatosNoMapaViewController: UIViewController {

 var contatos: [Contato] = Array()
 let dao = ContatoDao.ContatoDaoInstance()

 override func viewWillAppear(animated: Bool) {
 self.contatos = dao.lista()
 self.mapa.addAnnotations(self.contatos)
 }
}
```

Pronto, ao rodar a aplicação, devemos ver os contatos que possuem coordenadas aparecendo como pinos no mapa.



Figura 16.15: Pinos dos contatos

## 16.11 EXERCICIOS: MOSTRANDO OS CONTATOS NO MAPA

1. Faça com que a classe Contato siga o protocolo `MKAnnotation`. Desta forma, a classe `Contato` sabe prover as informações necessárias para o mapa exibi-lo como um pino.

Lembre-se de importar o protocolo.

```
#import <Foundation/Foundation.h>
#import <MapKit/MKAnnotation.h>
@interface Contato : NSObject <MKAnnotation>

@property (strong) NSString *nome;
@property (strong) NSString *endereco;
@property (strong) NSString *telefone;
@property (strong) NSString *site;
@property (strong) UIImage *foto;
@property (strong) NSNumber *latitude;
@property (strong) NSNumber *longitude;

@end
```

Implemente o `getter` para a `@property` definida no protocolo. Vamos criar uma coordenada a partir dos dados de latitude e longitude utilizando a função `CLLocationCoordinate2DMake`.

```

#import "Contato.h"

@implementation Contato

- (CLLocationCoordinate2D)coordinate {
 return CLLocationCoordinate2DMake([self.latitude doubleValue],
 [self.longitude doubleValue]);
}

@end

```

2. Adicione uma *propriedade* na classe *ContatosNoMapaViewController* para guardar a referência para todos os contatos cadastrados.

```

import UIKit
import MapKit

class ContatosNoMapaViewController: UIViewController {
 @IBOutlet weak var mapa: MKMapView!
 var contatos: [Contato] = Array()

 ...
}

```

3. Precisamos buscar os contatos do Dao. Crie a instância de `ContatoDao` e atribua os contatos para a propriedade `contatos` :

```

import UIKit
import MapKit

class ContatosNoMapaViewController: UIViewController {
 @IBOutlet weak var mapa: MKMapView!
 var contatos: [Contato] = Array()
 let dao: ContatoDao = ContatoDao.ContatoDaoInstance()

 override func viewDidAppear(animated: Bool) {
 self.contatos = dao.lista()
 self.mapa.addAnnotations(self.contatos)
 }
}

```

4. Ainda na classe *ContatosNoMapaViewController*, sobrescreva os métodos `viewWillAppear:` e `viewWillDisappear:` para adicionar e remover todos os contatos do mapa quando a tela for apresentada e escondida.

```

override func viewDidAppear(animated: Bool) {
 self.contatos = dao.lista()
 self.mapa.addAnnotations(self.contatos)
}

override func viewWillDisappear(animated: Bool) {
 self.mapa.removeAnnotations(self.contatos)
}

```

Rode a aplicação e faça o geocode do endereço de todos os contatos cadastrados.

Mude para a tela *ContatosNoMapaViewController* e veja os contatos sendo apresentados como pinos.

## 16.12 CUSTOMIZANDO E ADICIONANDO COMPORTAMENTO AO PINO

Até o momento, podemos visualizar todos os contatos no mapa mas sem muita interação com eles, concorda? Uma funcionalidade interessante seria poder selecionar um pino e ver algumas informações sobre o contato que ele representa, que tal?

O aplicativo nativo de mapas do iOS tem uma abordagem interessante, ao clicar em um pino, aparece um "balão" logo em cima do pino com algumas informações sobre o que ele apresenta.

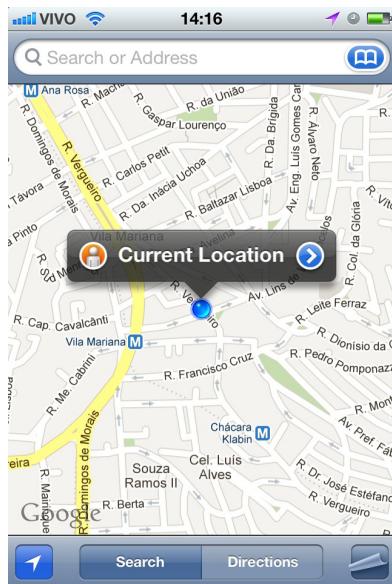


Figura 16.16: Interagindo com pinos

Vamos relembrar o protocolo de mapas que fizemos a classe de *Contato* seguir:

```
public protocol MKAnnotation : NSObjectProtocol {

 // Center latitude and longitude of the annotation view.
 // The implementation of this property must be KVO compliant.
 public var coordinate: CLLocationCoordinate2D { get }

 // Title and subtitle for use by selection UI.
 optional public var title: String? { get }

 optional public var subtitle: String? { get }
}
```

Perceba que apenas um *property\_s* é obrigatória e o restante é *\_optional* e, se o objeto que segue esse protocolo implementar esses métodos opcionais, o mapa vai cuidar de tirar proveito das informações adicionais passadas, apresentando-as em uma bolha.

Para isso, é necessário implementar os *getter\_s* na classe de contato, retornando o *\_nome* e o *email* para o **title** e o **subtitle**:

```
#import "Contato.h"
```

```

@implementation Contato

- (CLLocationCoordinate2D)coordinate {
 return CLLocationCoordinate2DMake([self.latitude doubleValue],
 [self.longitude doubleValue]);
}

- (NSString *)title {
 return self.nome;
}

- (NSString *)subtitle {
 return self.site;
}

@end

```

Rode a aplicação e veja que o vai acontecer: o mapa vai perceber que passamos dados adicionais ao implementar o protocolo e apresentará as informações seguindo o padrão da Apple, como balões.



Figura 16.17: Callout dos contatos

Para customizar o balão e dar a nossa cara para a aplicação, precisamos fazer algumas mudanças:

- Passar para o atributo *delegate* da classe *MKMapView* um objeto que siga o protocolo *MKMapViewDelegate*, esse protocolo define a forma que o mapa vai interagir com esse objeto.
- Para simplificar, vamos fazer a classe *ContatosNoMapViewController* seguir o protocolo, ela será responsável por customizar as informações exibidas pelo mapa.
- Implementar o método *mapView:viewFor:* retornando uma instância da classe

```
MKAnnotationView .
```

O método `mapView:viewFor:` é responsável por devolver o objeto que representa a informação no mapa para a `annotation` em questão, logo receberemos no argumento deste método todos os contatos cadastrados.

Esse método recebe todas as informações que o mapa precisa exibir, portanto, vamos receber também um objeto da classe que representa a localização do usuário: a `MKUserLocation` (que também implementa protocolo `MKAnnotation`). A localização do usuário aparece no mapa quando clicamos no botão de localização que adicionamos anteriormente.

A documentação da Apple pede para não criarmos nenhuma `MKAnnotationView` quando a `annotation` passada for da localização do usuário, portanto, precisamos validar se a classe do objeto passado é `MKUserLocation` e retornar `nil` caso verdadeiro.

```
func mapView(_ mapView: MKMapView, viewFor annotation: MKAnnotation) -> MKAnnotationView? {
 if annotation is MKUserLocation {
 return nil
 }
}
```

Neste método precisamos retornar um objeto do tipo `MKAnnotationView`, que representa a forma mais básica para se mostrar informações em um mapa. Como um pino possui diversas configurações, teríamos muito trabalho para replicar seu comportamento apenas para exibir uma foto adicional. Para facilitar o nosso trabalho, a Apple deixou disponível uma classe que é filha de `MKAnnotationView` e se comporta como um pino: a `MKPinAnnotationView`. Vamos utilizá-la:

```
func mapView(_ mapView: MKMapView, viewFor annotation: MKAnnotation) -> MKAnnotationView? {
 if annotation is MKUserLocation {
 return nil
 }

 let identifier:String = "pino"

 pino = MKPinAnnotationView(annotation: annotation, reuseIdentifier: identifier)

 return pino
}
```

Perceba que passamos no construtor do objeto um argumento chamado `identifier`, é a mesma abordagem que utilizamos para criar células para uma `UITableView` e economizar recursos do aparelho.

Quando um elemento visual apresentado no mapa deixar de ser visto, em vez de criar um novo elemento para mostrar a nova informação, reutilizaremos o objeto criado anteriormente apenas trocando as informações que ele exibe, economizando memória.

Vamos aterar o código para verificar se algum elemento do mapa deixou de ser visualizado para poder reutilizá-lo.

```

func mapView(_ mapView: MKMapView, viewFor annotation: MKAnnotation) -> MKAnnotationView? {

 if annotation is MKUserLocation {
 return nil
 }

 let identifier:String = "pino"

 var pino:MKPinAnnotationView

 if let reusableAnnotation = mapView.dequeueReusableCell(withIdentifier: identifier) as?
 MKPinAnnotationView{
 pino = reusablePin
 }else{
 pino = MKPinAnnotationView(annotation: annotation, reuseIdentifier: identifier)
 }

 return pino
}

```

Ao rodar a aplicação veremos exatamente o mesmo que antes, porém, agora podemos customizar o balão que aparece para o usuário da forma que quisermos.

Vamos dar uma olhada no que podemos alterar em um balão:



Figura 16.18: Callout dos pinos

Veja que, além de título e subtítulo, temos também duas imagens, uma em cada extremidade do balão, podemos acessar essas *views* a partir das propriedades `leftCalloutAccessoryView` e `rightCalloutAccessoryView`, para o lado esquerdo e direito respectivamente.

Vamos alterar essas *views* para exibir a foto do contato, caso ela tenha sido cadastrada. Porém, como a imagem do contato pode ser maior do que o tamanho suportado pelo balão, precisamos tomar cuidado para não exceder o seu tamanho, que é de *32px*.

Para isso, vamos criar um objeto do tipo `UIImageView`, essa classe é filha de `UIView` e tem por finalidade exibir uma imagem para o usuário. Vamos associá-la, então, ao atributo `leftCalloutAccessoryView` do pino que criamos.

```

func mapView(_ mapView: MKMapView, viewFor annotation: MKAnnotation) -> MKAnnotationView? {

 if annotation is MKUserLocation {
 return nil
 }

```

```

}

let identifier:String = "pino"

var pino:MKPinAnnotationView

if let reusableAnnotation = mapView.dequeueReusableCell(withIdentifier: identifier) as?
MKPinAnnotationView{
 pino = reusablePin
}else{
 pino = MKPinAnnotationView(annotation: annotation, reuseIdentifier: identifier)
}

if let contato = annotation as? Contato{

 pino.pinTintColor = UIColor.red
 pino.canShowCallout = true

 if let foto = contato.foto {
 let frame = CGRect(x: 0.0, y: 0.0, width: 32.0, height: 32.0)
 let imagemContato = UIImageView(frame: frame)

 imagemContato.image = foto

 pino.leftCalloutAccessoryView = imagemContato
 }

}

return pino
}

```

Ao rodar a aplicação, veremos a imagem do contato à esquerda do balão.



Figura 16.19: Balão customizado

#### CUSTOMIZAÇÕES AVANÇADAS

Ao implementar o protocolo `MKMapViewDelegate` podemos alterar diversos comportamentos do mapa. Podemos também estender a classe `MKAnnotationView` para criar um balão só nosso, totalmente customizado. Busque informações na documentação da Apple e veja as possibilidades.

### 16.13 EXERCICIO: CUSTOMIZANDO A APARÊNCIA DO MAPA

1. Ao customizar o pino precisamos passar algumas informações adicionais do contato e ao permitir a interação com o pino essa regra se torna obrigatória. Por isso precisamos implementar os dois métodos opcionais declarados no protocolo `MKAnnotation` - `title` e `subtitle`.

Para isso, edite a classe de **Contato** para retornar o nome e o site para os atributos opcionais que o protocolo descreve.

```
#import "Contato.h"
```

```

@implementation Contato

- (CLLocationCoordinate2D)coordinate {
 return CLLocationCoordinate2DMake(
 [self.latitude doubleValue],
 [self.longitude doubleValue]
);
}

- (NSString *)title {
 return self.nome;
}

- (NSString *)subtitle {
 return self.site;
}

@end

```

2. Altere a classe `ContatosNoMapaViewController`, faça ela seguir o protocolo `MKMapViewDelegate`. E no método `viewDidLoad` faça com que o delegate do `@IBOutlet` do mapa receba a referência de `self`

```

import UIKit
import MapKit

class ContatosNoMapaViewController: UIViewController, MKMapViewDelegate {
 @IBOutlet weak var mapa: MKMapView!
 var contatos: [Contato] = Array()

 override func viewDidLoad() {
 super.viewDidLoad()

 self.mapa.delegate = self

 locationManager.requestWhenInUseAuthorization()

 let botaoLocalizacao = MKUserTrackingBarButtonItem(mapView: self.mapa)

 self.navigationItem.rightBarButtonItem = botaoLocalizacao
 }

 ...
}

```

3. Implemente o método `mapView:viewFor:` na classe `ContatosNoMapaViewController`, nele vamos criar um `MKPinAnnotationView` tratando de reutilizá-lo corretamente; também precisamos validar se a classe do parâmetro passado é do tipo `MKUserLocation`, pois a Apple nos obriga a retornar `nil` caso seja.

```

func mapView(_ mapView: MKMapView, viewFor annotation: MKAnnotation) -> MKAnnotationView? {

 if annotation is MKUserLocation {
 return nil
 }

 let identifier:String = "pino"

```

```

var pino:MKPinAnnotationView

if let reusablePin = mapView.dequeueReusableCellWithIdentifier(identifier) as? MKPin-
nAnnotationView {
 pino = reusablePin
} else{
 pino = MKPinAnnotationView(annotation: annotation, reuseIdentifier: identifier)
}

if let contato = annotation as? Contato {

 pino.pinTintColor = UIColor.red
 pino.canShowCallout = true

 let frame = CGRect(x: 0.0, y: 0.0, width: 32.0, height: 32.0)
 let imagemContato = UIImageView(frame: frame)

 imagemContato.image = foto

 pino.leftCalloutAccessoryView = imagemContato
}

return pino
}

```

4. Rode a aplicação e selecione um contato no mapa que possua uma imagem, veremos ela sendo apresentada no lado esquerdo do balão.



Figura 16.20: Balão customizado

# TRABALHANDO COM BANCO DE DADOS UTILIZANDO O CORE DATA

Nossa aplicação está quase completa pois ainda falta persistirmos os contatos.

Poderíamos salvá-los em arquivo, porém se o número de contatos cadastrados aumentasse drasticamente, digamos, passando dos milhares, poderíamos ter grandes problemas de performance, pois estamos sempre carregando todos os contatos em memória.

Agora imagine que, além do número de registros aumentar, também precisemos efetuar buscas avançadas - por exemplo, buscar todos os contatos que começam com uma letra específica, ou todos que moram em *São Paulo*.

Perceba que isso se tornaria outro problema de performance, pois teríamos que fazer tudo isso em memória, sendo assim, carregar e filtrar se tornaria um problema.

## 17.1 BANCO DE DADOS, O SQLITE

Para manter as informações de uma aplicação, seja para permitir consultas futuras, buscas avançadas, relatórios detalhados e alterações nos dados, podemos tirar proveito do *SQLite*, um banco de dados relacional muito pequeno e rápido - presente em todos os dispositivos com **iOS**.

Com o crescimento das aplicações mobile, os aplicativos começaram a utilizar o *SQLite* para solucionar os problemas de performance, porém outras dificuldades foram encontradas.

Como o *SQLite* é uma biblioteca escrita *C*, o seu código não é tão simples de se entender e principalmente de dar manutenção. Além disso, ao se trabalhar com banco de dados relacionais precisamos escrever as consultas em uma linguagem específica, o **SQL**.

Portanto, além de escrever em *Objective-C*, tínhamos que dar manutenção nos códigos do *SQLite* (em *C*) e nas consultas no banco de dados (feitas em *SQL*).

Complicado para desenvolver e dar manutenção, concorda?

## 17.2 MAPEAMENTO OBJETO RELACIONAL

Os problemas descritos anteriormente foram enfrentados muito antes das aplicações mobile tomarem a proporção que tomaram, em aplicativos desktop e até em grandes sistemas corporativos.

Trabalhar com *SQL* (em qualquer banco de dados) gera ruído, pois desenvolvemos a aplicação em uma linguagem **orientada a objetos** e armazenamos os dados de forma relacional, um paradigma totalmente diferente.

Para tentar unir os dois mundos, foram criadas ferramentas de **mapeamento objeto-relacional** (ORM), que tentam eliminar a necessidade de se trabalhar com *SQL* dentro de um mundo **orientado a objetos**. Para isso, precisamos entender algumas coisas:

- Ao pensar em uma tabela, em banco de dados estamos querendo representar alguma informação. Já, no mundo **orientado a objetos**, para representar um dado pensamos em classes.
- Podemos associar as linhas de cada registro inseridas em uma tabela como **instâncias** de uma classe.
- As colunas de uma tabela representam os atributos de um objeto.
- Buscas avançadas serão efetuadas em objetos, de forma simples, sem precisar manipular *SQL*.

As ferramentas de *ORM* estão presentes em todas as plataformas, seja para desenvolvimento mobile ou não.

E não foi diferente no **iOS**, a Apple percebeu a necessidade e implementou uma ferramenta muito interessante, o **Core Data**, que serve exatamente para isso - eliminar o ruído gerado ao escrever buscas com *SQL*, facilitando o desenvolvimento e a manutenção de aplicações que usem grandes volumes de dados.

Para desenhar o nosso modelo de dados vamos tirar proveito de uma ferramenta do *XCode*, o *Data Modeler*.

### 17.3 UTILIZANDO O DATA MODELER

Para evitar trabalhar com tabelas, o *XCode* introduziu o **Data Modeler**, uma forma simples de desenhar relacionamentos.

Como ao criarmos nosso projeto já deixamos marcada a opção `Use Core Data` foi criado um arquivo chamado `.xcdatamodeld` em nosso projeto. Ao selecioná-lo vamos perceber uma tela diferente das que já estamos acostumados a usar.

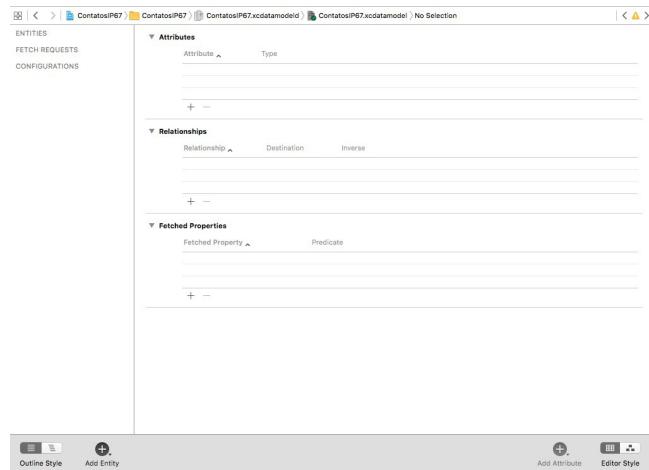


Figura 17.1: Visualizando um Data Model

Para poder mapear uma classe com uma tabela do banco de dados, vamos criar uma *Entity*. Pense em uma entidade como uma classe normal, onde cada instância dessa classe representa um registro na tabela.

No editor gráfico do *Data Model* podemos adicionar uma *Entity* clicando em *Add Entity*, vamos chamar essa entidade de **Contato**.

Ainda no editor gráfico, podemos adicionar os atributos do contato, eles representam colunas na tabela. Repare que, além de dizer o nome do atributo, podemos alterar o seu tipo, refletindo no tipo da coluna criada no banco.

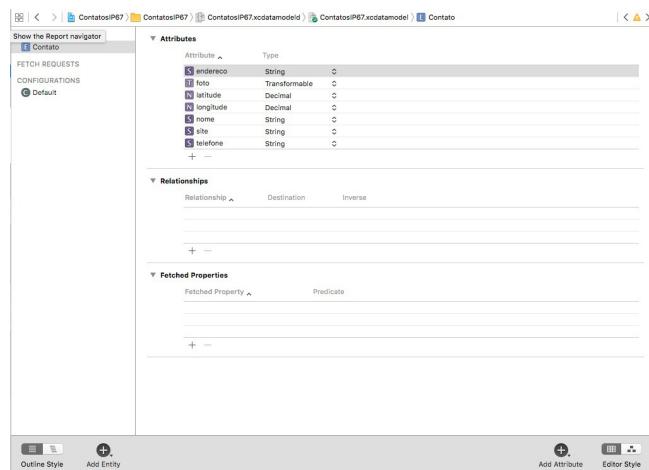


Figura 17.2: Criando o Contato

## 17.4 EXERCÍCIO - CRIANDO O MODELO DE CONTATO, UTILIZANDO O DATA MODELER

1. Adicionando uma nova entidade no *ContatosIP67.xcdatamodeld* e configurando suas propriedades.

- Selecione o arquivo *ContatosIP67.xcdatamodeld* para visualizá-lo no *Data Modeler*.
- Adicione uma nova *Entity* chamada **Contato** clicando em *Add Entity*.
- Adicione todas as propriedades que temos no contato atual. Configure os tipos das propriedades corretamente:
- **String** para: `endereco` , `nome` , `site` , `telefone`
- **Decimal** para: `latitude` e `longitude`
- **Transformable** para: `foto`

#### PROPRIEDADES PARA UM ATRIBUTO

Selecione um atributo declarado no *Data Modeler* da *Entity*, na aba de utilities busque a opção *Data Modeler Inspector*, nela podemos visualizar as configurações da propriedade selecionada.

Podemos adicionar regras de validações, se o atributo é opcional ou não, valor inicial, entre outras funcionalidades.

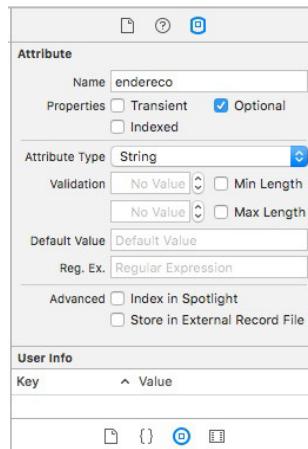


Figura 17.3: Propriedades dos atributos

Agora, com o modelo criado e tudo mapeado, podemos entender como utilizar o framework.

## 17.5 ENTENDENDO COMO OBTER O CONTEXTO DO CORE DATA

Com o modelo de dados criado, podemos entender como o *Core Data* vai nos ajudar a **não** escrever *SQL*, sempre trabalhando com objetos.

Primeiramente, precisamos entender como conseguimos acessar o contexto do *Core Data*, este

contexto é onde tudo acontece - onde objetos são salvos, buscas efetuadas e atualizações feitas, refletindo diretamente no banco de dados.

O contexto do *Core Data* precisa saber qual mapeamento ele precisa gerenciar, ou seja, com qual *Data Model* ele vai trabalhar.

O interessante é que o framework de **ORM** da Apple nos possibilita ir além, podemos ter um mapeamento que não reflete apenas para um banco de dados, poderíamos, por exemplo, utilizar o mapeamento do *Data Model* para gerenciar dados em memória ou até uma implementação totalmente customizada, sem alterar a forma que trabalhamos com o framework.

Para conseguir tirar proveito de toda essa capacidade de customização, precisamos entender melhor algumas classes que fazem parte do framework - `NSManagedObjectModel` , `NSPersistentStoreCoordinator` e `NSPersistentStore` - após entendê-las poderemos focar no contexto do *Core Data* a classe `NSManagedObjectContext` .

- `NSManagedObjectModel` - responsável por representar um modelo de dados, no caso é o arquivo que criamos e configuramos com as nossas entidades, o `xcdatamodeld`. Este arquivo é compilado para o formato `mmod` e pode ser encontrado na pasta principal do aplicativo.

```
let modelURL = Bundle.main.url(forResource: "ContatosIP67", withExtension: "momd")
let managedObjectModel = NSManagedObjectModel(contentsOf: modelURL)!
```

- `NSPersistentStoreCoordinator` - uma ponte entre o modelo e onde os dados são gerenciados, com ele poderemos armazenar as informações em diversas formas - em um banco de dados SQL, em memória ou desenvolver uma forma completamente customizada.

```
let persistentStoreCoordinator = NSPersistentStoreCoordinator(managedObjectModel: self.managedObjectModel)

let url = FileManager.default.urls(for: .documentDirectory, in: .userDomainMask)[0].appendingPathComponent("SingleViewCoreData.sqlite")
```

- `NSPersistentStore` - o tipo de armazenamento. No nosso caso, um banco de dados *SQL*.

```
let persistentStoreCoordinator = NSPersistentStoreCoordinator(managedObjectModel: self.managedObjectModel)

let url = FileManager.default.urls(for: .documentDirectory, in: .userDomainMask)[0].appendingPathComponent("SingleViewCoreData.sqlite")

persistentStoreCoordinator.addPersistentStore(ofType: NSSQLiteStoreType, configurationName: nil,
at: url, options: nil)
```

Perceba que buscamos dentro da pasta de documentos da aplicação um arquivo `.sqlite`, este é o nosso banco de dados e será criado automaticamente pelo framework.

Veja que, ao adicionar um `persistentStore`, dissemos ao `coordinator` que a `URL` passada no parâmetro aponta para um banco de dados, por isso passamos o tipo `NSSQLiteStoreType` .

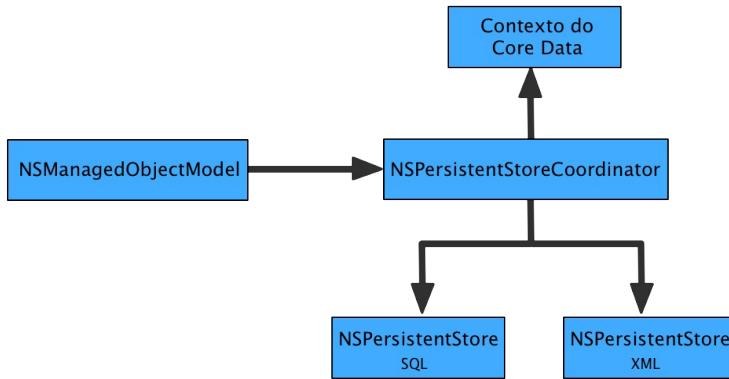


Figura 17.4: Modelo de objetos do Core Data

### OUTROS TIPOS DE PERSISTENTSTORE

Por padrão a Apple trabalha com quatro tipos de persistência de dados utilizando o *Core Data*

- NSSQLiteStoreType - banco de dados SQL
- NSXMLStoreType - arquivo em XML
- NSBinaryStoreType - arquivo binário
- NSInMemoryStoreType - em memória

Procure a vantagem de cada um deles lendo a documentação da Apple.

Perfeito, agora podemos criar um `NSManagedObjectContext` a partir do `NSPersistentStoreCoordinator`.

```

let coordinator = self.persistentStoreCoordinator
var managedObjectContext = NSManagedObjectContext(concurrencyType: .mainQueueConcurrencyType)
managedObjectContext.persistentStoreCoordinator = coordinator

```

Com a chegada do Swift 3 toda essa parte foi encapsulada e agora temos um único objeto chamado `NSPersistentContainer`. Ele recebe o nome do arquivo `mod` que queremos usar e para inicializar todo o contexto usamos o método `loadPersistentStores`, que recebe como argumento um `closure` com o `storeDescription` que foi configurado para o `NSPersistentContainer` e um `error` em caso ter ocorrido algum problema ao inicializar o contexto podemos tratar dentro do `closure`.

O método `loadPersistentStores` é responsável por fazer toda a mágica citada anteriormente.

```

lazy var persistentContainer: NSPersistentContainer = {
 let container = NSPersistentContainer(name: "Contato")
 container.loadPersistentStores(completionHandler: { (storeDescription, error) in
 if let error = error as NSError? {
 fatalError("Unresolved error \(error), \(error.userInfo)")
 }
 })
}

```

```

 })
 }

 return container
}()
```

Esse tipo de código se torna tão repetitivo que ao criar um novo projeto podemos marcar a opção *Use Core Data*. Dessa forma, um *template* de projeto com as configurações básicas para se acessar o *contexto* é gerado, porém é importante entender o código que o *Xcode* nos provê.

Como ao criarmos nossa aplicação já dissemos que queríamos utilizar o *Core Data*, todo o código que cria o `NSPersistentContainer` já foi criado pelo próprio *Xcode* dentro de nossa classe `AppDelegate`. Dessa forma não precisamos digitar todo este monte de código novamente!

Agora vamos pensar um pouco, quem de nossas classes é responsável por cuidar do acesso aos dados de um `Contato`? O próprio `ContatoDao`! Como não temos nenhum outro Dao em nossa aplicação, podemos isolar todo este código dentro do `ContatoDao` e ele que continuará sendo responsável pelo *CRUD* de `Contato`.

## 17.6 EXERCÍCIO - ISOLANDO OS MÉTODOS DE COREDATA

1. Vamos mover as propriedades e os métodos que foram declarados no `AppDelegate` para uma classe `CoreDataUtil`:

- Abra o arquivo `AppDelegate` e copie o trecho de código abaixo e coloque dentro do arquivo `CoreDataUtil` (não esqueça de importar `CoreData` na classe `CoreDataUtil`):

```

// MARK: - Core Data stack

lazy var persistentContainer: NSPersistentContainer = {
 /*
 The persistent container for the application. This implementation
 creates and returns a container, having loaded the store for the
 application to it. This property is optional since there are legitimate
 error conditions that could cause the creation of the store to fail.
 */

 let container = NSPersistentContainer(name: "Contato")
 container.loadPersistentStores(completionHandler: { (storeDescription, error) in
 if let error = error as NSError? {
 // Replace this implementation with code to handle the error appropriately.
 // fatalError() causes the application to generate a crash log and terminate. You
 should not use this function in a shipping application, although it may be useful during devel-
 opment.
 /*
 Typical reasons for an error here include:
 * The parent directory does not exist, cannot be created, or disallows writing.
 * The persistent store is not accessible, due to permissions or data protection w-
 hen the device is locked.
 * The device is out of space.
 * The store could not be migrated to the current model version.
 Check the error message to determine what the actual problem was.
 */
 })
}
```

```

 fatalError("Unresolved error \(error), \(error.userInfo)")
 }
})

return container
}()

// MARK: - Core Data Saving support

func saveContext () {

 let context = persistentContainer.viewContext
 if context.hasChanges {
 do {
 try context.save()
 } catch {
 // Replace this implementation with code to handle the error appropriately.
 // fatalError() causes the application to generate a crash log and terminate. You
 // should not use this function in a shipping application, although it may be useful during devel-
 // opment.
 let nserror = error as NSError
 fatalError("Unresolved error \(nserror), \(nserror.userInfo)")
 }
 }
}

```

- Quando movemos essa parte do código para a classe *CoreDataUtil* o código da classe *AppDelegate* parou de funcionar, pois o *AppDelegate* estava usando uma referencia para um método que removemos. /= Para resolver esse problema vamos comentar a linha que fazia referencia ao método:

Comente ou remova a declaração *self.saveContext()*

```

func applicationWillTerminate(_ application: UIApplication) {
 // Called when the application is about to terminate. Save data if appropriate. See also
 applicationDidEnterBackground:.

 // Saves changes in the application's managed object context before the application termi-
 nates.

 //self.saveContext()
}

```

- Ao termino sua classe *CoreDataUtil* deve ficar assim:

```

import CoreData

class CoreDataUtil {

 // MARK: - Core Data stack

 lazy var persistentContainer: NSPersistentContainer = {
 /*
 The persistent container for the application. This implementation
 creates and returns a container, having loaded the store for the
 application to it. This property is optional since there are legitimate
 error conditions that could cause the creation of the store to fail.
 */

 let container = NSPersistentContainer(name: "Contato")
 container.loadPersistentStores(completionHandler: { (storeDescription, error) in

```

```

 if let error = error as NSError? {
 // Replace this implementation with code to handle the error appropriately.
 // fatalError() causes the application to generate a crash log and terminate.
 // You should not use this function in a shipping application, although it may be useful during development.

 /*
 Typical reasons for an error here include:
 * The parent directory does not exist, cannot be created, or disallows writing.
 * The persistent store is not accessible, due to permissions or data protection settings.
 * The device is out of space.
 * The store could not be migrated to the current model version.
 Check the error message to determine what the actual problem was.
 */
 fatalError("Unresolved error \(error), \(error.userInfo)")
 }
 })

 return container
}()

// MARK: - Core Data Saving support

func saveContext () {

 let context = persistentContainer.viewContext
 if context.hasChanges {
 do {
 try context.save()
 } catch {
 // Replace this implementation with code to handle the error appropriately.
 // fatalError() causes the application to generate a crash log and terminate.
 // You should not use this function in a shipping application, although it may be useful during development.
 let nserror = error as NSError
 fatalError("Unresolved error \(nserror), \(nserror.userInfo)")
 }
 }
}

```

- Agora para poder usar as funcionalidades do *Core Data* vamos fazer com que nosso *DAO* herde da nossa classe *CoreDataUtil*.

```

import Foundation

class ContatoDao: CoreDataUtil {
 ...
}

```

2. Rode a aplicação, tudo deve continuar funcionando normalmente.

Agora temos a base para prosseguir e entender como utilizar o contexto do *Core Data*.

## 17.7 INSERINDO CONTATOS NO BANCO DE DADOS

---

A partir da classe `NSManagedObjectContext` podemos criar novos registros e buscar informações salvas.

Um registro para o *Core Data* é representado pela classe `NSManagedObject`. É possível trabalhar com ela diretamente, porém, essa classe é genérica e pode salvar qualquer informação, seja um Contato ou qualquer outro mapeamento.

Como a classe `NSManagedObject` é genérica, para criar uma instância precisamos dizer qual entidade esse registro vai representar, apontando para um mapeamento feito no arquivo `Modelo_Contatos.xcdatamodeld`.

Este apontamento é feito utilizando a classe `NSEntityDescription`, ela possui um construtor que recebe qual modelo ela deve representar e em qual contexto será utilizada, devolvendo uma instância de `NSManagedObject`.

```
let contato = NSEntityDescription.insertNewObject(forEntityName: "Contato", into: self.persistentContainer.viewContext)
```

Para preencher as informações desse registro, a classe `NSManagedObject` possui o método `- setValue:forKey:`, para associar um valor para qualquer atributo.

Para criar um registro do tipo **Contato** chegamos ao seguinte código:

```
let contato = NSEntityDescription.insertNewObject(forEntityName: "Contato", into: self.persistentContainer.viewContext)

contato.setValue("John Doe", forKey: "nome")
contato.setValue("www.johndoe.com", forKey: "site")
```

Podemos enxergar um problema recorrente nessa abordagem, trabalhar diretamente com `NSManagedObject` pode ser comparado ao se trabalhar com `NSDictionary` para representar informações, pois estamos trabalhando com **chave e valor** e não com um objeto. Que tal aprender a maneira correta?

## 17.8 TRABALHANDO CORRETAMENTE COM A CLASSE NSMANAGEDOBJECT

Para inserir contatos no banco utilizando o *Core Data* podemos trabalhar apenas com a classe `NSManagedObject`, mas nós temos uma classe para representar um **Contato**, não precisamos trabalhar com um conjunto de **chave e valor**.

Para transformar a classe de **Contato** em um `NSManagedObject` vamos mudar a sua declaração, em vez de herdar de `NSObject` vamos fazer com que a classe **Contato** seja filha de `NSManagedObject` - ganhando todos os seus comportamentos avançados.

```
@interface Contato : NSManagedObject<MKAnnotation>
```

Ao alterar a declaração da classe *Contato* ela passa a ter todas as funcionalidades que um *NSManagedObject* possui, podendo ser gerenciado diretamente no contexto do *Core Data*.

Porém, como a classe *NSManagedObject* é a responsável por gerenciar os atributos daquela instância, precisamos alterar a forma com que os atributos da classe de *Contatos* são gerados. Para falar a verdade não vamos mais gerar esses acessores, vamos deixar que tudo seja feito em tempo de execução - responsabilidade da classe *NSManagedObject*.

Para passar a responsabilidade para a classe *NSManagedObject* vamos adicionar a marcação `@dynamic` nas propriedades. Dessa forma, vamos dizer que os acessores para os atributos serão gerados dinamicamente, assim o compilador fica tranquilo sabendo que tudo será gerado em tempo de execução pelo *Core Data*.

```
#import "Contato.h"
@implementation Contato

// vamos deixar a responsabilidade com o _Core Data_, utilizando o @dynamic
@dynamic nome, telefone, email, endereco, site, latitude, longitude, foto;
```

Além disso, precisamos alterar o mapeamento da classe no seu *Data Model*, alterar qual classe a *Entity* **Contato** deve referenciar. Podemos fazer isso editando o arquivo *Modelo\_Contatos.xcdatamodeld* e alterando a classe da entidade na aba *Data Model Inspector*. Veja que isso é interessante quando não queremos que a classe tenha o mesmo nome da *Entity*.

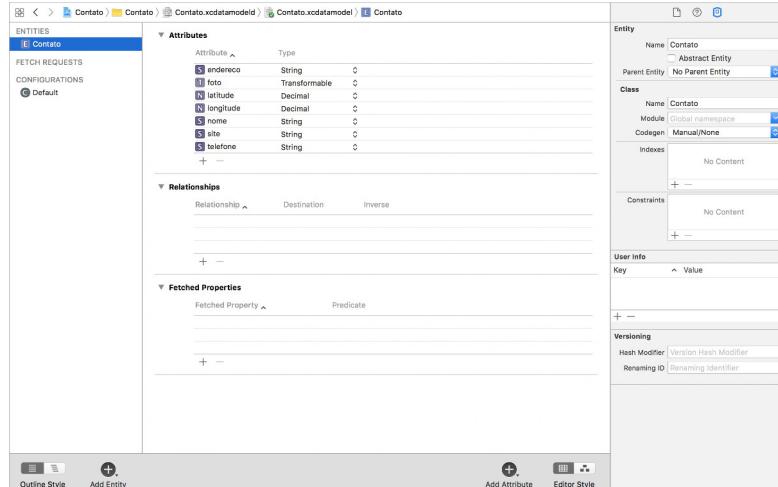


Figura 17.5: Alterando o Data Model

Dessa forma para criar um *Contato* já gerenciado pelo *Core Data* chegamos ao seguinte código:

```
let contato:Contato = NSEntityDescription.insertNewObject(forEntityName: "Contato", into: self.persistentContainer.viewContext) as! Contato

contato.nome = "John Doe"
contato.site = "www.johndoe.com"
```

Agora todo contato criado já estará sendo gerenciado pelo *Core Data*, porém, para que tudo seja

salvo e persistindo no banco, precisamos salvar o contexto - enquanto isso todas as alterações estavam sendo feitas apenas em memória.

A classe `NSManagedObjectContext` possui o método `save:`, que caso algum erro ocorra é lançado uma `Exception` e nesse caso temos que tratar.

```
func saveContext () {

 let context = persistentContainer.viewContext
 if context.hasChanges {
 do {
 try context.save()
 } catch {
 // Replace this implementation with code to handle the error appropriately.
 // fatalError() causes the application to generate a crash log and terminate. You should
not use this function in a shipping application, although it may be useful during development.
 let nserror = error as NSError
 fatalError("Unresolved error \(nserror), \(nserror.userInfo)")
 }
 }
}
```

Perfeito, agora podemos migrar a nossa aplicação para utilizar o *Core Data* para armazenar os contatos cadastros no banco de dados.

## HABILITANDO O LOG DO CORE DATA

Para visualizar as operações que o *Core Data* faz com o *SQLite* precisamos habilitar o seu log.

- Altere o *scheme* do projeto, vá em *Product -> Schema ->Edit Scheme* ou pelo atalho ⌘ + <.
- Na configuração de Run & Debug, altere para aba de *Arguments*.
- Adicione um novo parâmetro no campo *Arguments Passed On Launch*.  
com.apple.CoreData.SQLDebug 1

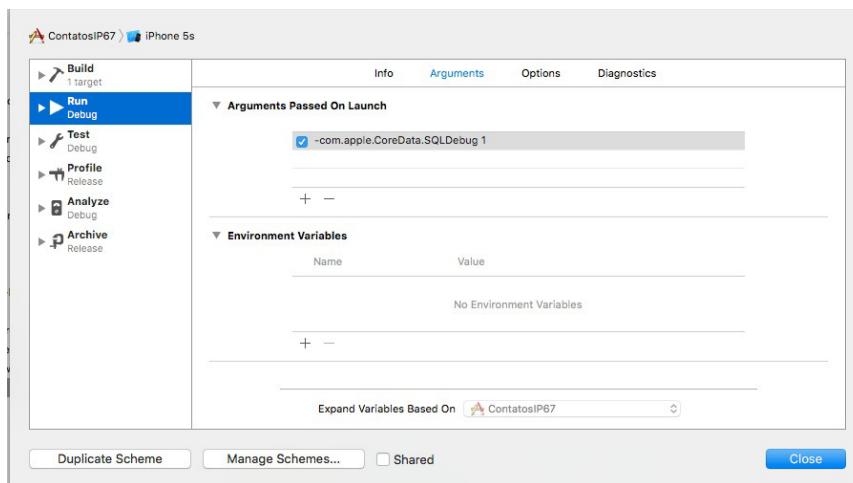


Figura 17.6: Core Data Debug

## 17.9 EXERCÍCIO - CONTATOS COMO NSMANAGEDOBJECT

1. Altere a declaração da classe *Contato* para que ela seja filha de *NSManagedObject*, dessa forma o *Core Data* poderá gerenciar as os atributos do nosso objeto.

```
#import <Foundation/Foundation.h>

@interface Contato : NSManagedObject <MKAnnotation>

// declaração das propriedades
```

2. Vamos deixar o *Core Data* responsável por gerar os acessores para as propriedades; então, adicione a diretiva `@dynamic` na implementação das propriedades. Ao colocar a marcação `@dynamic` nas propriedades, os acessores são criados apenas quando o registro é inserido pelo *Core Data*, deixando que ele gerencie os atributos.

```
#import "Contato.h"

@implementation Contato
```

```

@dynamic nome, telefone, endereco, site, latitude, longitude, foto;
// restante da implementação

```

3. Abra o modelo de contatos que criamos (*Modelo\_Contatos.xcdatamodeld*) e altere a classe da **Entity** Contato para a classe de Contato.

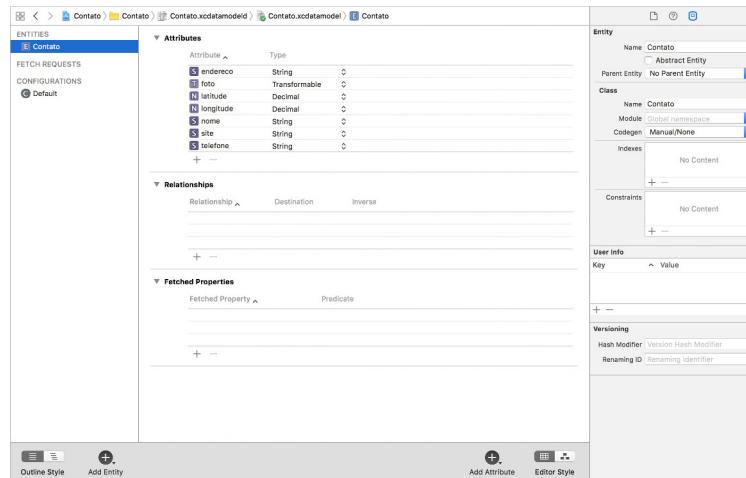


Figura 17.7: Alterando o Data Model

Agora que a aplicação já está pronta para utilizar o *Core Data* e sabemos como inserir novos contatos, que tal adicionar alguns dados iniciais no aplicativo? Para quando o usuário acessar a aplicação pela primeira vez algumas informações já venham presentes.

Porém, como podemos garantir que essa operação só será executada uma vez? Afinal, se inserirmos dados iniciais todas as vezes que a aplicação abrir geraremos informações duplicadas, concorda?

Precisamos de alguma forma guardar a informação de que a carga foi feita, para nas próximas, não efetuá-la. Que tal guardar essa informação nas configurações do aplicativo?

## 17.10 ARMAZENANDO CONFIGURAÇÕES COM USERDEFAULTS

A classe `UserDefault`s pode ser utilizada para salvar preferências e/ou configurações do aplicativo, como por exemplo, se já efetuamos ou não a inserção dos dados. Essas preferências continuam a existir mesmo se o aplicativo for fechado e pertence apenas a nossa aplicação.

Com a classe podemos armazenar informações de diversos tipos:

- String
- URL
- Data
- Array
- Dictionary

- Int
- Float
- Double
- Bool

Que tal salvar um *Bool* para saber se os dados iniciais foram inseridos ou não?

Para utilizar as preferências do aplicativo, precisamos de uma **instância** da classe `UserDefault`s, para isso, vamos chamar o atributo `standard` da própria classe, que retorna as configurações daquele aplicativo, não tendo a necessidade de instanciar manualmente o objeto.

```
let configuracoes = UserDefaults.standard
```

Vamos salvar um *Bool* para uma chave qualquer, para que depois possamos acessar o valor dessa chave e saber se já efetuamos a carga dos dados.

```
let configuracoes = UserDefaults.standard
configuracoes.set(true, forKey: "dados_inseridos")
```

Para que os dados sejam realmente salvos temos que pedir que as informações sejam sincronizadas, fazemos isso chamando o método `synchronize`.

```
let configuracoes = UserDefaults.standard
configuracoes.set(true, forKey: "dados_inseridos")
configuracoes.synchronize();
```

Para buscar um valor dentro das configurações do aplicativo, podemos utilizar o método `bool(forKey:)`, que retorna um valor *booleano* para uma chave, retornando `false` caso o valor não exista.

```
let configuracoes = UserDefaults.standard
let dadosInseridos = configuracoes.bool(forKey: "dados_inseridos")

if !dadosInseridos {
 //efetuar a inserção dos dados no banco
}
```

Perfeito, agora sabemos como inserir dados e como evitar que a operação ocorra mais de uma vez, criaremos uma carga inicial no aplicativo e teremos a certeza que será feita apenas uma vez.

## 17.11 EXERCÍCIOS - CARREGANDO INFORMAÇÕES PARA O BANCO APENAS UMA VEZ COM USERDEFAULTS

1. Crie um método na classe `ContatoDao`, nele vamos criar alguns registros de contato. Lembre-se de chamar o método `saveContext`.

Para que essa carga seja efetuada apenas uma vez, vamos utilizar a classe `UserDefault`s.

```
func inserirDadosIniciais(){
 let configuracoes = UserDefaults.standard
```

```

let dadosInseridos = configuracoes.bool(forKey: "dados_inseridos")

if !dadosInseridos {

 let caelumSP = NSEntityDescription.insertNewObject(forEntityName: "Contato",
into: self.persistentContainer.viewContext) as! Contato

 caelumSP.nome = "Caelum SP"
 caelumSP.endereco = "São Paulo, SP, Rua Vergueiro, 3185";
 caelumSP.telefone = "01155712751";
 caelumSP.site = "http://www.caelum.com.br";
 caelumSP.latitude = -23.5883034
 caelumSP.longitude = -46.632369

 self.saveContext()

 configuracoes.set(true, forKey: "dados_inseridos")

 configuracoes.synchronize()

}

}

```

2. Precisamos chamar o método criado para inserir os dados iniciais, vamos alterar o construtor da classe `ContatoDao` para logo que a instância for criada, os dados já sejam carregados:

```

override private init(){
 self.contatos = Array()

 super.init()

 self.inserirDadosIniciais()
}

```

Se quiser, aproveite para habilitar o debug do *Core Data* para imprimir as instruções geradas por ele no console.

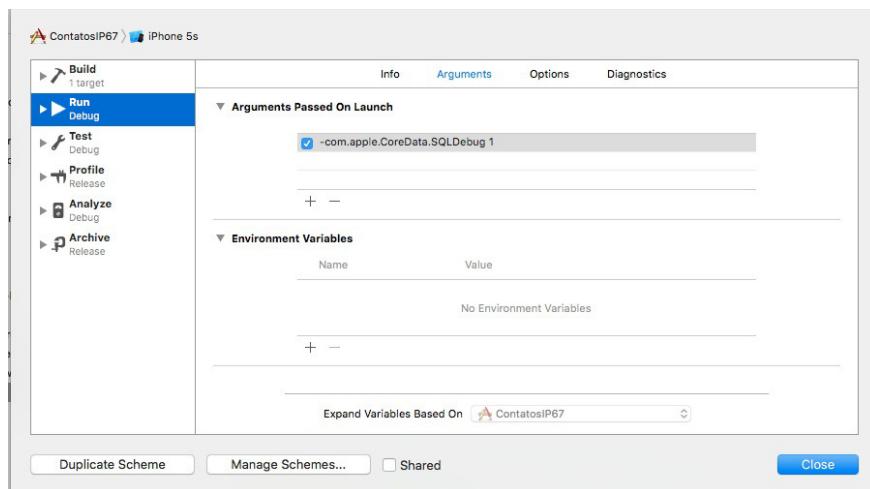


Figura 17.8: Core Data Debug

3. Rode a aplicação mais de uma vez e valide que os dados são inseridos apenas uma vez pois estamos

controlando utilizando a classe `User Defaults` .

Agora que temos os dados iniciais inseridos, podemos listar os contatos cadastrados para visualizá-los.

Para buscar informações diretamente no banco de dados, vamos aprender mais algumas classes do *Core Data* - `NSFetchRequest` , `NSSortDescriptor` - entre outras.

## 17.12 BUSCANDO DADOS COM NSFETCHREQUEST

Ao desenvolver aplicações, sejam elas mobile ou não, precisamos buscar dados das mais diversas formas. Seja para exibir um relatório ou buscar os contatos cadastrados - esses dados normalmente precisam ser ordenados, agrupados e somados das mais diversas formas - e o *Core Data* pode nos auxiliar - evitando escrever *SQL* e não tendo que lidar com todos os dados em memória.

Para efetuar buscas o *Core Data* nos disponibiliza a classe `NSFetchRequest` e, para listar todos os contatos cadastrados no banco, podemos executar o seguinte código:

```
let busca = NSFetchRequest<Contato>(entityName: "Contato")
```

Tendo a busca declarada podemos executá-la diretamente no contexto:

```
do {
 self.contatos = try self.persistentContainer.viewContext.fetch(busca)
} catch let error as NSError {
 print("Fetch Falhou: \(error.localizedDescription)")
}
```

Veja que podemos passar um argumento para o método `error`, dessa forma, podemos saber quando um erro na busca ocorreu, tratando quando necessário.

Porém ao efetuar buscas o *Core Data* não assume nenhuma regra de ordenação, a busca efetuada anteriormente não teria nenhuma regra específica, podendo variar cada vez que executada, precisamos dizer por qual atributo queremos que a busca seja ordenada.

Para descrever a ordem em que os dados são buscados, podemos utilizar a classe `NSSortDescriptor` que possui um método de classe para ordenar por um atributo da classe - sendo de forma crescente ou não.

```
let orderPorNome = NSSortDescriptor(key: "nome", ascending: true)
```

Podemos criar mais de uma regra de ordenação e passar para a instância de `NSFetchRequest` que vai tratar de executar a busca com as regras passadas.

```
let busca = NSFetchRequest<Contato>(entityName: "Contato")
let orderPorNome = NSSortDescriptor(key: "nome", ascending: true)
busca.sortDescriptors = [orderPorNome]
```

```

do {
 self.contatos = try self.persistentContainer.viewContext.fetch(busca)
}catch let error as NSError {
 print("Fetch Falhou: \(error.localizedDescription)")
}

```

Perfeito, agora podemos listar todos os contatos para exibir na aplicação.

## 17.13 EXERCÍCIOS - LISTANDO CONTATOS COM NSFETCHREQUEST

- Crie um método na classe `ContatoDao` para efetuar a busca de todos os contatos cadastrados ordenados pelo nome.

```

func carregaContatos(){
 let busca = NSFetchedResultsController<Contato>(entityName: "Contato")

 let orderPorNome = NSSortDescriptor(key: "nome", ascending: true)

 busca.sortDescriptors = [orderPorNome]

 do {
 self.contatos = try self.persistentContainer.viewContext.fetch(busca)
 }catch let error as NSError {
 print("Fetch Falhou: \(error.localizedDescription)")
 }
}

```

- Chame o método declarado no momento que o `ContatoDao` for instanciado. Não precisamos mais instanciar o `Array` para o atributo de contatos pois já atribuímos o resultado da busca para a propriedade `contatos`:

```

override private init(){
 super.init()

 self.inserirDadosIniciais()

 self.carregaContatos()
}

```

Rode a aplicação e veja os dados sendo exibidos.

## 17.14 EXERCÍCIOS - MIGRANDO A APLICAÇÃO PARA O CORE DATA

- Agora que o `Contato` é um `NSManagedObject` não podemos mais criá-lo apenas chamando seu construtor, dependemos de uma instância de `NSManagedObjectContext` para poder instanciá-lo.

Precisamos alterar a nossa aplicação, já que no formulário de contatos instanciamos um novo registro chamando seu construtor.

Para não precisarmos passar o contexto do `Core Data` de um lado para o outro, vamos criar um método em nosso `ContatoDao` que retorne um `Contato` já gerenciado.

```
func novoContato() -> Contato{
```

```
 return NSEntityDescription.insertNewObject(forEntityName: "Contato", into: self.persistentContainer.viewContext) as! Contato
 }
```

2. No método `pegaDadosDoFormulario` da classe `FormularioContatoViewController` altere a forma com que instanciamos um novo `Contato` para utilizar o método criado anteriormente.

```
func pegaDadosDoFormulario(){
 if contato == nil {
 self.contato = dao.novoContato()
 }
 // restante da implementação
}
```

3. Para salvarmos o contexto do *Core Data*, adicione a chamada ao método `saveContext` assim que a aplicação entrar em *background*, na classe *AppDelegate*.

```
func applicationDidEnterBackground(_ application: UIApplication) {
 // Use this method to release shared resources, save user data, invalidate timers, and store
 // enough application state information
 // to restore your application to its current state in case it is terminated later.
 // If your application supports background execution, this method is called instead of applicationWillTerminate: when the user quits.

 ContatoDAO.sharedInstance().saveContext()

}
```

# CONSUMINDO WEBSERVICES EM NOSSA APLICAÇÃO

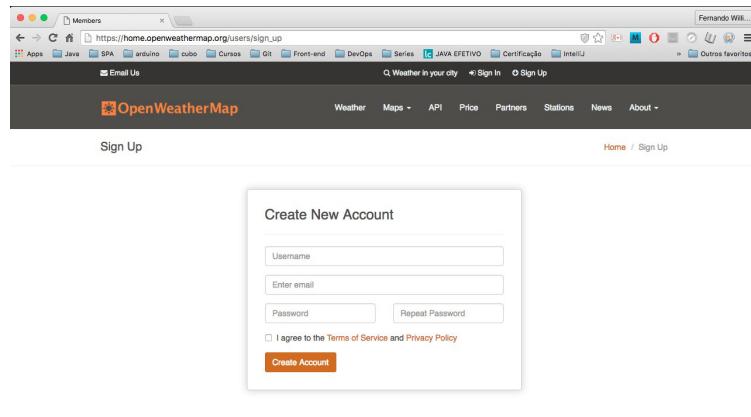
Quando desenvolvemos aplicativos mobile uma tarefa bem comum é integrar o App com alguma serviço para pegar e\ou enviar informações. Então mão na massa para consumir um WebService em nossa aplicação!

## 18.1 O SERVIÇO

Em nosso aplicativo iremos consumir um serviço de meteorologia. A ideia saber como está o clima no endereço do nosso contato. Para isso vamos usar uma **API Open Source** Open Weather Map . Para podermos utilizar essa API temos que nos registrar para obter uma **API KEY (APPID)** Vamos nos cadastrar então:

## 18.2 EXERCICIO: EFETUANDO CADASTRO

1. Acesse o link [http://home.openweathermap.org/users/sign\\_up](http://home.openweathermap.org/users/sign_up)



2. Será exibido um *popup* para informarmos onde pretendemos usar essa api.

How and where will you use our API?

Hi! We are doing some housekeeping around thousands of our customers. Your impact will be much appreciated. All you need to do is to choose in which exact area you use our services.

Company	<input type="text"/>
* Purpose	Mobile apps development

**Cancel** **Save**

3. Após preencher o *popup* estaremos na *home* do nosso usuário. Clique no link **API Keys**

My Home

Setup API keys My Weather Stations My Services My Payments Billing plans Map editor Block logs Logout

NEW! Find your API keys in the special sheet **API keys**

4. Nessa tela temos um campo com *key* onde temos nossa **API Key**. Ela será usada para podermos consumir o serviço

API keys

Setup API keys My Weather Stations My Services My Payments Billing plans Map editor Block logs Logout

NEW! You can generate as much API keys as needed for your subscription. We accumulate the total loading from all of them.

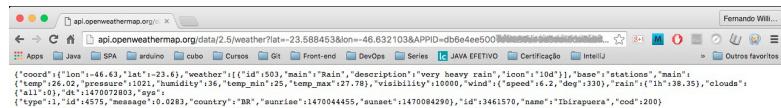
Key	Name	Create key
db6e4ee50[REDACTED]	Default	<input type="button" value="Create key"/> * Name <input type="text"/> <input type="button" value="Generate"/>

5. Para verificar se está tudo funcionando vamos consumir o serviço informando uma latitude, longitude e o **APPID**.

- Latitude = -23.588453
- Longitude = -46.632103

[http://api.openweathermap.org/data/2.5/weather?  
lat=-23.588453&lon=-46.632103&APPID=SUA\\_APP\\_KEY\\_AQUI&units=metric](http://api.openweathermap.org/data/2.5/weather?lat=-23.588453&lon=-46.632103&APPID=SUA_APP_KEY_AQUI&units=metric)

Quando acessamos esse link em nosso navegador teremos um retorno parecido com esse:



The screenshot shows a browser window with the URL [api.openweathermap.org/data/2.5/weather?lat=-23.588453&lon=-46.632103&APPID=db6e4ee500](http://api.openweathermap.org/data/2.5/weather?lat=-23.588453&lon=-46.632103&APPID=db6e4ee500). The page displays a JSON object representing weather data for Ibirapuera, São Paulo. The JSON structure includes coordinates, weather conditions, base data, stations, clouds, visibility, wind, and other parameters.

```
{"coord": {"lon": -46.63, "lat": -23.6}, "weather": [{"id": 503, "main": "Rain", "description": "very heavy rain", "icon": "10d"}], "base": "stations", "stations": [{"name": "Ibirapuera", "humidity": 36, "temp_min": 15, "temp_max": 27.78}, {"name": "Ibirapuera", "humidity": 36, "temp_min": 15, "temp_max": 27.78}], "clouds": {"all": 10}, "dt": 1470072803, "sys": {"message": 0.0293, "country": "BR", "sunrise": 1470044455, "sunset": 1470084290}, "id": 3461570, "name": "Ibirapuera", "cod": 200}
```

## DOCUMENTAÇÃO DA API E QUICK START

Você pode acessar o link e seguir o passo-a-passo para se cadastrar e utilizar a API <http://openweathermap.org/appid>

Você pode encontrar toda a documentação da api que iremos utilizar no link abaixo: <http://openweathermap.org/current>

Note que o retorno é um *JSON*. Esse formato de retorno é muito comum em integrações, por ser muito leve e versátil.

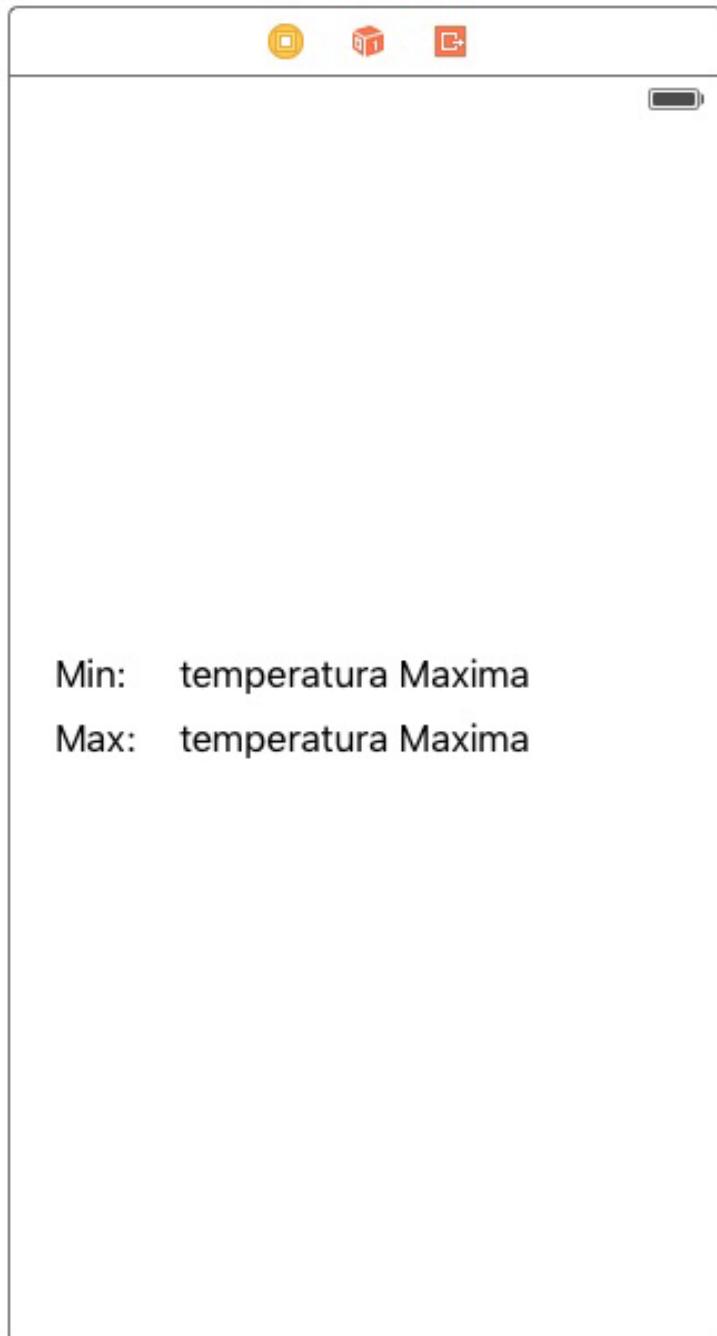
Agora que já conseguimos efetuar requisições para nosso serviço web. Temos que fazer essa requisição por dentro do nosso aplicativo e transformar o *JSON* retornado pelo webservice em algum objeto.

Vamos lá!

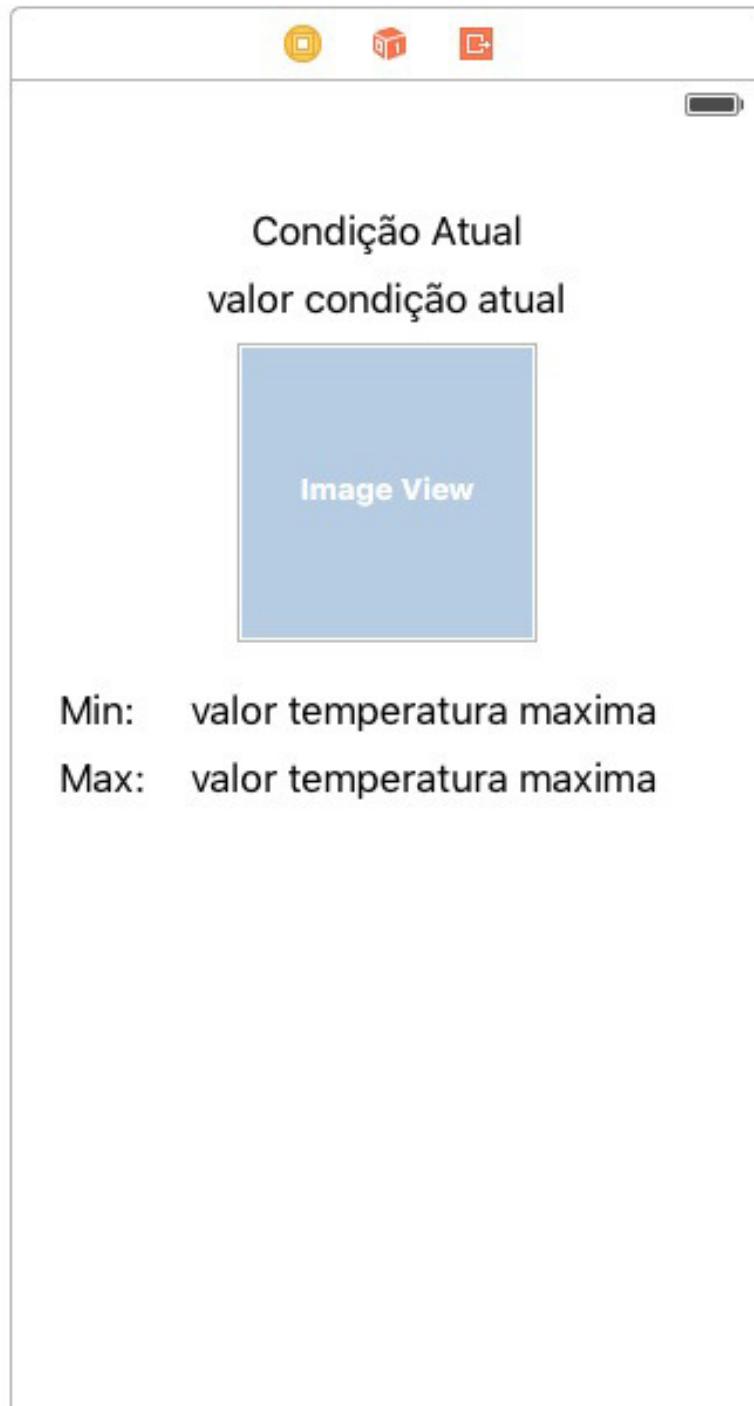
## 18.3 PREPARANDO NOSSO APP PARA CONSUMIR O SERVIÇO

Dentro do nosso *app* teremos uma tela para exibir a temperatura minima, maxima e a condição atual e além disso uma imagem do site para simbolizar a condição do tempo.

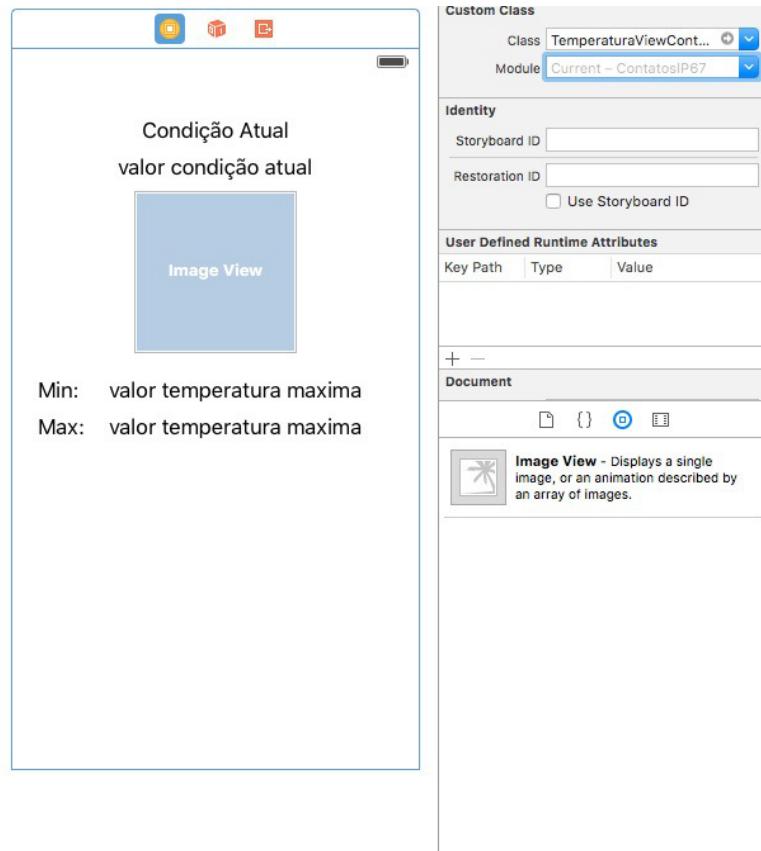
No nosso storyboard vamos adicionar um *ViewController* dentro desse controller vamos adicionar um label *Min* um label *Max* e para cada um deles, vamos adicionar um outro label na frente que recebera os valores das temperaturas minima e maxima:



Além disso vamos adicionar um label para condição atual e valor da condição atual, e um *imageView*. Ao término teremos uma tela com o seguinte layout:



Vamos criar uma classe para nosso controller e associar o controller e a classe.



Feito isso vamos associar o nossos labels de valores com nossa classe para podermos manipular o conteúdo deles e também associar o *imageView*.

```
import UIKit

class TemperaturaViewController: UIViewController {

 @IBOutlet weak var labelCondicaoAtual: UILabel!
 @IBOutlet weak var labelTemperaturaMaxima: UILabel!
 @IBOutlet weak var labelTemperaturaMinima: UILabel!
 @IBOutlet weak var imageView: UIImageView!

 ...

}
```

Perceba que nossa tela não está associada a nenhuma outra tela como chegaremos até ela?

Além de conseguirmos acessar telas através de *segue*, também conseguimos acessar uma tela programaticamente. E para esse caso usaremos dessa forma. Vamos adicionar uma nova opção dentro no nosso *Gerenciador de Ações*.

Dentro da classe *Gerenciador de Ações* vamos adicionar uma nova ação dentro do método *acoesDoController*:

```
func exibirAcoes(em controller: UIViewController) {
```

```

self.controller = controller

let alertView = UIAlertController(title: self.contato.nome, message: nil, preferredStyle: .actionSheet)

let cancelar = UIAlertAction(title: "Cancelar", style: .cancel, handler: nil)

let ligarParaContato = UIAlertAction(title: "Ligar", style: .default){ action in
 self.ligar()
}

let exibirContatoNoMapa = UIAlertAction(title: "Visualizar No Mapa", style: .default) { action in
 self.abrirMapa()
}

let exibirSiteDoContato = UIAlertAction(title: "Visualizar Site", style: .default){ action in
 self.abrirNavegador()
}

let exibirTemperatura = UIAlertAction(title: "Visualizar Clima", style: .default){ action in
 self.exibirTemperatura()
}

alertView.addAction(cancelar)
alertView.addAction(ligarParaContato)
alertView.addAction(exibirContatoNoMapa)
alertView.addAction(exibirSiteDoContato)
alertView.addAction(exibirTemperatura)

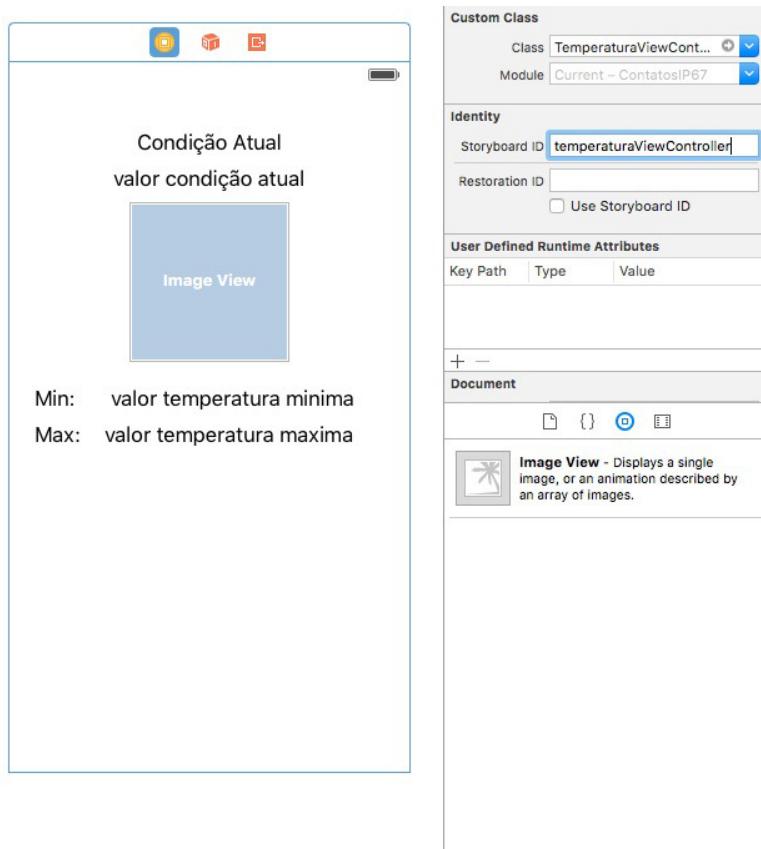
self.controller.present(alertView, animated: true, completion: nil)
}

```

Agora precisamos criar o método `mostrarClima` e dentro dele pegar uma instancia do nosso controller.

Para isso o `storyboard` tem um método `instantiateViewControllerWithIdentifier` que recebe um identificador e a partir dele ele pega uma instancia do `viewController`.

Para adicionar o identificador vamos no `storyboard` selecionamos o `viewController` que queremos colocar o identificador. Vamos agora acessar a aba *identity inspector* e dentro do grupo *Identity* vamos colocar um valor para o campo *Storyboard ID*.



Agora que temos o identificador vamos para o método `mostraClima` e instanciar nosso `viewController`. Para pegarmos a referência do nosso Storyboard acessamos a propriedade `storyboard` de um `viewController`. E a partir do `storyboard` invocamos o método `instantiateViewControllerWithIdentifier` passando como parâmetro o `id` que definimos na `storyboard`. Como esse método retorna um `viewController` precisamos converte-lo para o nosso `temperaturaViewController`.

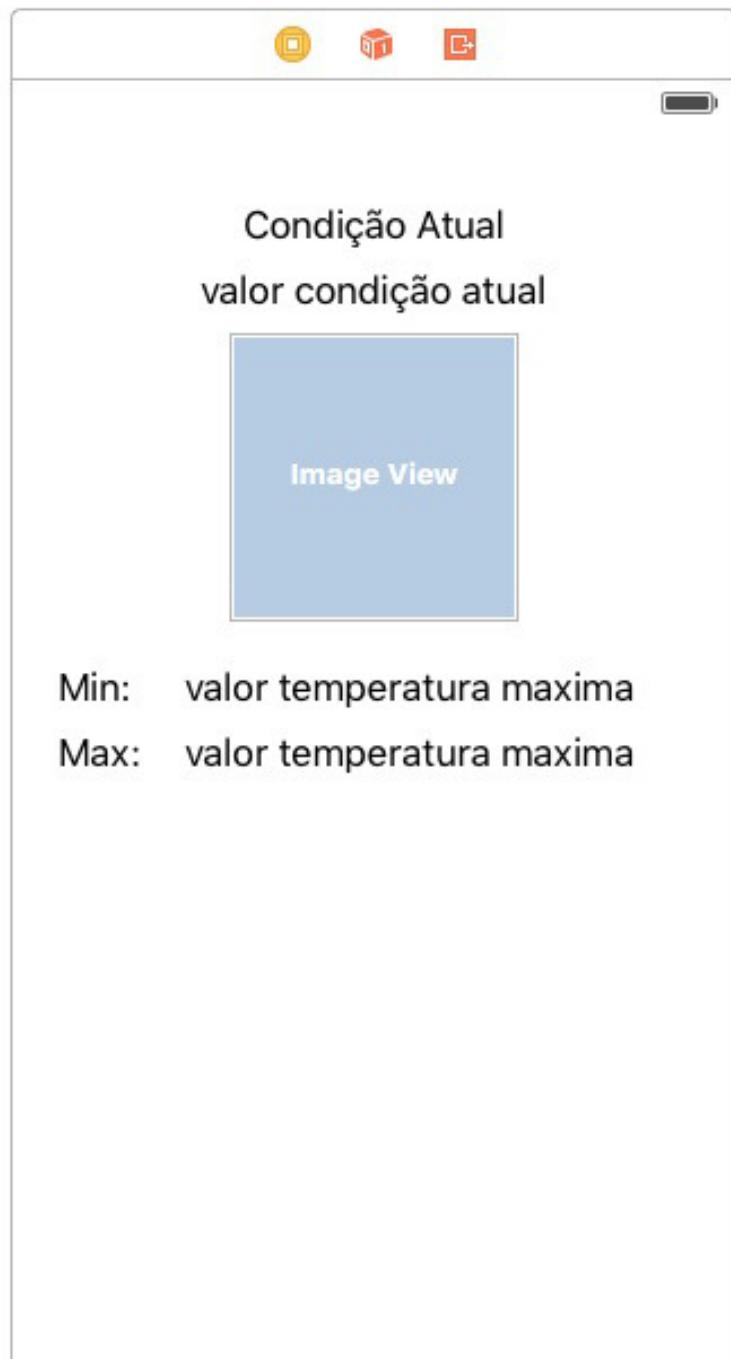
```
func exibirTemperatura(){
 let temperaturaViewController = controller.storyboard?.instantiateViewController(withIdentifier: "temperaturaViewController") as! TemperaturaViewController

 controller.navigationController?.pushViewController(temperaturaViewController, animated: true)
}
```

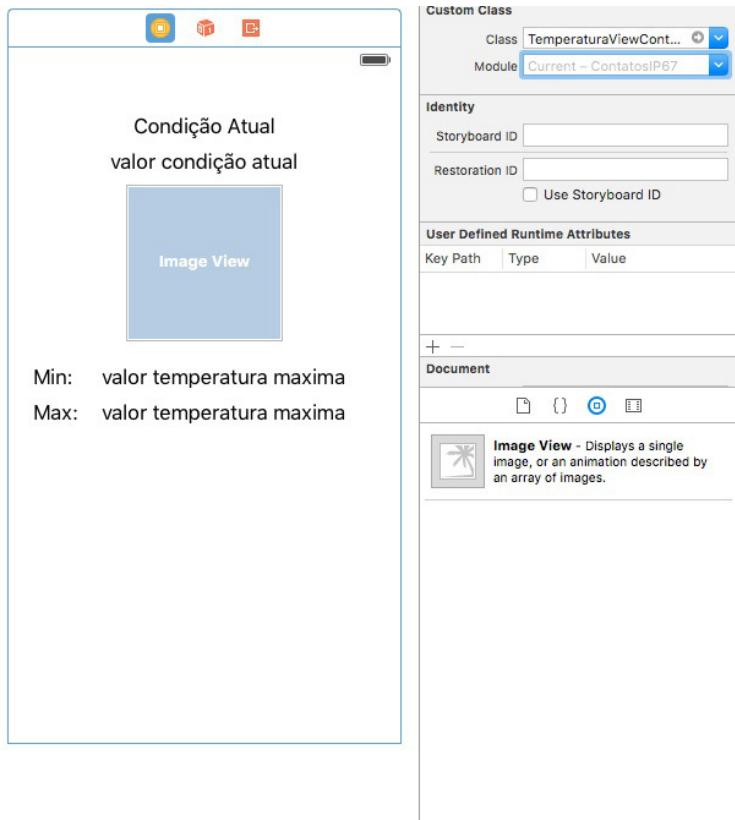
Pronto dessa forma conseguimos acessar nossa tela, agora só falta consumir o webservice e exibir as informações. Mas antes vamos colocar em prática tudo o que falamos até aqui.

## 18.4 EXERCÍCIO: CRIANDO E ACESSANDO A TELA DE TEMPERATURA

- Crie uma tela que tenha o seguinte layout:



2. Crie uma classe `TemperaturaViewController` que herde de `UIViewController` e associe a tela com a classe



3. Associe também os labels de valores e o imageView com a nossa classe:

```
import UIKit

class TemperaturaViewController: UIViewController {

 @IBOutlet weak var labelCondicaoAtual: UILabel!
 @IBOutlet weak var labelTemperaturaMaxima: UILabel!
 @IBOutlet weak var labelTemperaturaMinima: UILabel!
 @IBOutlet weak var imageView: UIImageView!

 ...
}
```

4. Adicione uma nova ação dentro da nossa classe Gerenciador de Ações

```
func exibirAcoes(em controller: UIViewController) {
 self.controller = controller
 ...

 let exibirTemperatura = UIAlertAction(title: "Visualizar Clima", style: .default){ action in
 self.exibirTemperatura()
 }

 alertView.addAction(exibirTemperatura)
}
```

```

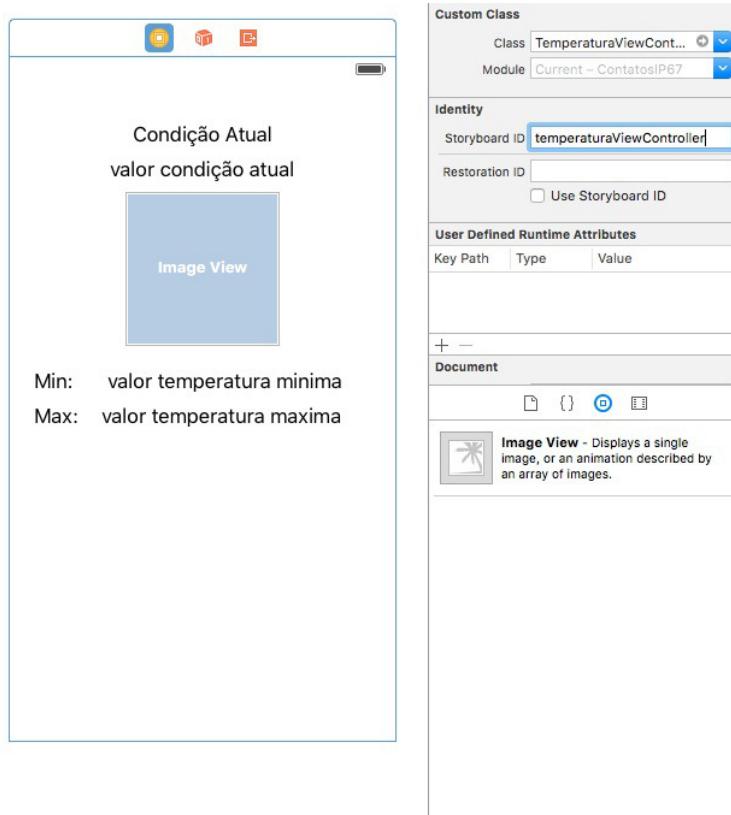
}

func exibirTemperatura(){

}

```

5. Defina um *ID* para nossa tela dentro do storyboard .



6. Implemente o método `mostrarClima` :

```

func mostrarClima(){
 let temperaturaViewController = controller.storyboard?.instantiateViewController(withIdentifier
 "temperaturaViewController") as! TemperaturaViewController

 controller.navigationController?.pushViewController(temperaturaViewController, animated: true)
}

```

## 18.5 CONSUMINDO O WEBSERVICE

Dentro da nossa classe `TemperaturaViewController` teremos que pegar os valores de *latitude* e *longitude* do contato selecionado e a partir desses dados montar a *URL* que iremos utilizar no nosso webservice.

Para pegar esses dados de *latitude* e *longitude* vamos criar uma propriedade contato dentro da nossa

classe `TemperaturaViewController` e no método `mostraClima` vamos definir o conteúdo dessa propriedade com o contato selecionado.

```
import UIKit

class TemperaturaViewController: UIViewController {

 @IBOutlet weak var labelCondicaoAtual: UILabel!
 @IBOutlet weak var labelTemperaturaMaxima: UILabel!
 @IBOutlet weak var labelTemperaturaMinima: UILabel!
 @IBOutlet weak var imageView: UIImageView!

 var contato: Contato?
 ...
}
```

Na classe `GerenciadorDeAcoes` vamos definir o conteúdo desse contato:

```
func mostrarClima(){
 let temperaturaViewController = controller.storyboard?.instantiateViewController(withIdentifier: "temperaturaViewController") as! TemperaturaViewController

 temperaturaViewController.contato = self.contato

 controller.navigationController?.pushViewController(temperaturaViewController, animated: true)
}
```

Agora que temos nosso contato dentro da nossa classe vamos adicionar uma string fixa com a *URL* base que queremos acessar. (Ou seja toda a url exceto os dois parâmetros *lat, lon*) Além disso também precisamos de uma *URL* base para pegar a imagem.

```
import UIKit

class TemperaturaViewController: UIViewController {

 @IBOutlet weak var labelCondicaoAtual: UILabel!
 @IBOutlet weak var labelTemperaturaMaxima: UILabel!
 @IBOutlet weak var labelTemperaturaMinima: UILabel!
 @IBOutlet weak var imageView: UIImageView!

 var contato: Contato?

 let URL_BASE = "http://api.openweathermap.org/data/2.5/weather?APPID=_SEU_APPID_AQUI_&units=metric"
 let URL_BASE_IMAGE = "http://openweathermap.org/img/w/"
 ...
}
```

Dentro do método `viewDidLoad` vamos criar um objeto *URL* nos baseando nessa *URL* base e adicionando os parâmetros *lat* e *lon* com os dados do nosso contato.

```
override func viewDidLoad() {
 super.viewDidLoad()

 if let contato = self.contato {
 if let endpoint = URL(string: URL_BASE + "&lat=\(contato.latitude ?? 0)&lon=\(contato.longitude ?? 0)") {
```

```
 }
 }
}
```

Agora o que precisamos é fazer uma requisição para esse endpoint e tratar o retorno dessa requisição para isso vamos usar a classe `URLSession`. Essa classe nos permite fazer requisições para uma determinada *URL* e nos da uma forma de tratar o retorno.

Para utilizar `URLSession` precisamos definir algumas configurações, para o nosso caso usaremos as configurações padrões.

```
override func viewDidLoad() {
 super.viewDidLoad()

 if let contato = self.contato {
 if let endpoint = URL(string: URL_BASE + "&lat=\(contato.latitude ?? 0)&lon=\(contato.longitude ?? 0)") {
 let session = URLSession(configuration: .default)

 }
 }
}
```

Para conseguir efetuar essa requisição vamos criar uma tarefa que será executada de forma assíncrona faremos isso utilizando o método `dataTask:with`. Depois de criar a tarefa precisamos executá-la e utilizaremos o método `resume` da classe `URLSessionDataTask`. Para tratarmos o retorno dessa requisição usaremos um *closure* que recebe 3 parâmetros e não tem nenhum retorno:

- Data - o retorno da requisição no nosso caso o *JSON* de retorno (*Data*)
- Response - Objeto *URLResponse* que podemos converter para *NSHTTPURLResponse* onde temos o response http propriamente dito
- Error - caso tenha acontecido algum erro na requisição será armazenado nesse parâmetro

```
override func viewDidLoad() {
 super.viewDidLoad()

 if let contato = self.contato {
 if let endpoint = URL(string: URL_BASE + "&lat=\(contato.latitude ?? 0)&lon=\(contato.longitude ?? 0)") {

 let session = URLSession(configuration: .default)

 let task = session.dataTask(with: endpoint) { (data, response, error) in

 if let httpResponse = response as? HTTPURLResponse {
 // Aqui dentro devemos tratar o retorno da requisição
 }

 task.resume()
 }
 }
 }
}
```

```
}
```

Para sabermos se a requisição foi bem sucedida vamos verificar se `statusCode` da `response` é `200` em caso negativo vamos imprimir o motivo no console.

```
override func viewDidLoad() {
 super.viewDidLoad()

 if let contato = self.contato {
 if let endpoint = URL(string: URL_BASE + "&lat=\(contato.latitude ?? 0)&lon=\(contato.longitude ?? 0)") {

 let session = URLSession(configuration: .default)

 let task = session.dataTask(with: endpoint) { (data, response, error) in

 if let httpResponse = response as? HTTPURLResponse {
 if httpResponse.statusCode == 200 {
 }else{
 print("ocorreu algum problema com a requisição")
 }
 }

 task.resume()
 }
 }

 task.resume()
 }
}
```

A primeira coisa que precisamos fazer caso a requisição tenha sido bem sucedida é converter o retorno JSON para algum objeto que conseguimos manipular. Se olharmos a estrutura do JSON de retorno vemos que sempre temos uma chave associada à um valor bem parecido com um *Dictionary*:

```

1 {
2 "coord":{
3 "lon":-46.63,
4 "lat":-23.6
5 },
6 "weather":[
7 {
8 "id":503,
9 "main":"Rain",
10 "description":"very heavy rain",
11 "icon":"10d"
12 }
13],
14 "base":"stations",
15 "main":{
16 "temp":26.02,
17 "pressure":1021,
18 "humidity":36,
19 "temp_min":25,
20 "temp_max":27.78
21 },
22 "visibility":10000,
23 "wind":{
24 "speed":6.2,
25 "deg":330
26 },
27 "rain":{
28 "1h":38.35
29 },
30 "clouds":{
31 "all":0
32 },
33 "dt":1470072803,
34 "sys":{
35 "type":1,
36 "id":4575,
37 "message":0.0283,
38 "country":"BR",
39 "sunrise":1470044455,
40 "sunset":1470084290
41 },
42 "id":3461570,
43 "name":"Ibirapuera",
44 "cod":200
45 }

```

E para transformar um *JSON* em um *Dictionary* (ou *Dictionary*) temos uma classe chamada `JSONSerialization` e nela temos um método chamado `JSONObjectWithData`, que recebe um objeto `Data` e devolve `AnyObject` que podemos converter para nosso *Dictionary* (ou *Dictionary*).

O unico detalhe é que esse método lança uma exceção e temos que trata-la. No nosso caso vamos simplesmente imprimir no console, caso seja lançada uma exceção.

```

let task = session.dataTask(with: endpoint) { (data, response, error) in
 if let httpResponse = response as? HTTPURLResponse {

 if httpResponse.statusCode == 200 {

 do{
 if let json = try JSONSerialization.jsonObject(with: data!, options:[]) as? [String:AnyObject] {
 }
 }catch let error as NSError {
 print("Não foi possível fazer o parse do JSON: \(error.localizedDescription)")
 }
 }else{
 print("A requisição não foi bem sucedida: \(error!.localizedDescription)")
 }
 }
 }

task.resume()

```

Ao observarmos o *JSON* as informações de temperatura minima e maxima que queremos estão dentro do chave `main`, e as informações de condição atual e icone estão dentro de um array dentro de `weather`.

Agora que temos um *Dictionary* vamos manipula-lo para chegar nas informações necessárias.

```

let task = session.dataTask(with: endpoint) { (data, response, error) in
 if let httpResponse = response as? HTTPURLResponse {

 if httpResponse.statusCode == 200 {

 do{
 if let json = try JSONSerialization.jsonObject(with: data!, options:[]) as? [String:AnyObject] {
 let main = json["main"] as! [String:AnyObject]
 let weather = json["weather"]![0] as! [String:AnyObject]
 let temp_min = main["temp_min"] as! Double
 let temp_max = main["temp_max"] as! Double
 let icon = weather["icon"] as! String
 let condicao = weather["main"] as! String

 }
 }catch let error as NSError {
 print("Não foi possível fazer o parse do JSON: \(error.localizedDescription)")
 }
 }else{
 print("A requisição não foi bem sucedida: \(error!.localizedDescription)")
 }
 }
 }

task.resume()

```

Agora que temos quase todas as informações o que precisamos é atualizar nossa tela. Porém como foi dito essa task será executada de maneira assincrona ou seja em uma `thread` separada. Porém nossa

tela está sendo executada na `thread` principal.

O que queremos é que de alguma forma conseguimos chavar entre a `thread` separada e a principal.

Para fazer isso utilizaremos novamente o *GDC (Grand Central Dispatch)*

Através do `DispatchQueue` vamos fazer uma chamada assíncrona para a `main thread`. Para isso usamos a seguinte invocação `DispatchQueue.main.async` agora basta usar o *closure*, e implementar o que queremos que seja executado na `main thread`.

```
let task = session.dataTask(with: endpoint) { (data, response, error) in
 if let httpResponse = response as? HTTPURLResponse {

 if httpResponse.statusCode == 200 {

 do{
 if let json = try JSONSerialization.jsonObject(with: data!, options:[]) as? [String:AnyObject] {
 let main = json["main"] as! [String:AnyObject]
 let weather = json["weather"]![0] as! [String:AnyObject]
 let temp_min = main["temp_min"] as! Double
 let temp_max = main["temp_max"] as! Double
 let icon = weather["icon"] as! String
 let condicao = weather["main"] as! String

 DispatchQueue.main.async {

 self.labelCondicaoAtual.text = condicao
 self.labelTemperaturaMinima.text = temp_min.description + "°"
 self.labelTemperaturaMaxima.text = temp_max.description + "°"

 self.labelCondicaoAtual.isHidden = false
 self.labelTemperaturaMinima.isHidden = false
 self.labelTemperaturaMaxima.isHidden = false

 }

 }
 }catch let error as NSError {
 print("Não foi possível fazer o parse do JSON: \(error.localizedDescription)")
 }
 }else{
 print("A requisição não foi bem sucedida: \(error!.localizedDescription)")
 }
 }
}

task.resume()
```

O ultimo passo que falta codificar para integração é pegar o ícone e colocar no `imageView` Para isso precisaremos fazer uma nova requisição e o resultado da requisição colocar no `imageView`. Vamos isolar essa lógica dentro de um método chamado `pegaImagem`:

```
let task = session.dataTask(with: endpoint) { (data, response, error) in
```

```

if let httpResponse = response as? HTTPURLResponse {

 if httpResponse.statusCode == 200 {

 do{
 if let json = try JSONSerialization.jsonObject(with: data!, options:[]) as? [String:AnyObject] {
 let main = json["main"] as! [String:AnyObject]
 let weather = json["weather"]![0] as! [String:AnyObject]
 let temp_min = main["temp_min"] as! Double
 let temp_max = main["temp_max"] as! Double
 let icon = weather["icon"] as! String
 let condicao = weather["main"] as! String

 DispatchQueue.main.async {

 self.labelCondicaoAtual.text = condicao
 self.labelTemperaturaMinima.text = temp_min.description + "°"
 self.labelTemperaturaMaxima.text = temp_max.description + "°"
 self.pegaImagen(icon)

 self.labelCondicaoAtual.isHidden = false
 self.labelTemperaturaMinima.isHidden = false
 self.labelTemperaturaMaxima.isHidden = false

 }

 }
 }catch let error as NSError {
 print("Não foi possível fazer o parse do JSON: \(error.localizedDescription)")
 }
 }else{
 print("A requisição não foi bem sucedida: \(error!.localizedDescription)")
 }
}
}

task.resume()

```

Agora vamos implementar o método `pegaImagen`:

```

private func pegaImagen(_ icon:String){
 if let endpoint = URL(string: URL_BASE_IMAGE + icon + ".png") {
 let session = URLSession(configuration: .default)
 let task = session.dataTask(with: endpoint){ (data, response, error) in

 if let httpResponse = response as? HTTPURLResponse {

 if httpResponse.statusCode == 200 {
 DispatchQueue.main.async {
 print("Exibindo Imagem")
 self.imageView.image = UIImage(data: data!)
 }
 }
 }
 }
 }

 task.resume()
}

```

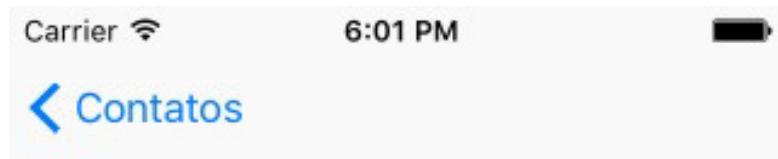
}

Para que nosso **APP** possa efetuar requisições precisamos adicionar uma entrada no nosso arquivo `info.plist`.

Vamos adicionar a seguinte entrada no nosso arquivo `info.plist`



Agora nosso **APP** consegue efetuar requisições e ao executarmos ele e acessar a tela de clima temos o seguinte resultado:



## 18.6 EXERCÍCIO: IMPLEMENTANDO INTEGRAÇÃO

1. Declare uma propriedade para o `Contato` dentro da classe `TemperaturaViewController`

```
import UIKit

class TemperaturaViewController: UIViewController {

 @IBOutlet weak var labelCondicaoAtual: UILabel!
 @IBOutlet weak var labelTemperaturaMaxima: UILabel!
 @IBOutlet weak var labelTemperaturaMinima: UILabel!
 @IBOutlet weak var imageView: UIImageView!
```

```

 var contato:Contato?
 ...
}

```

2. Na classe `GerenciadorDeAcoes` passe o contato selecionado para nosso controller

```

func mostrarClima(){
 let temperaturaViewController = controller.storyboard?.instantiateViewController(withIdentifier
 "temperaturaViewController") as! TemperaturaViewController

 temperaturaViewController.contato = self.contato

 controller.navigationController?.pushViewController(temperaturaViewController, animated: true)
}

```

3. Dentro do método `viewDidLoad` use `URLSession` para consumir o webservice ```swift override

```

func viewDidLoad() {

 super.viewDidLoad()

 if let contato = self.contato {

 if let endpoint = URL(string: URL_BASE + "&lat=\(contato.latitude ?? 0)&lon=\(contato.longitude ?? 0)") {

 let session = URLSession(configuration: .default)
 print(endpoint)
 let task = session.dataTask(with: endpoint) { (data, response, error) in
 if let httpResponse = response as? HTTPURLResponse {

 if httpResponse.statusCode == 200 {

 do{
 if let json = try JSONSerialization.jsonObject(with: data!, options:[]) as? [String:AnyObject] {

 let main = json["main"] as! [String:AnyObject]
 let weather = json["weather"]![0] as! [String:AnyObject]
 let temp_min = main["temp_min"] as! Double
 let temp_max = main["temp_max"] as! Double
 let icon = weather["icon"] as! String
 let condicao = weather["main"] as! String

 DispatchQueue.main.async {

 print(condicao)
 print(temp_min)
 print(temp_max)
 print(icon)

 self.labelCondicaoAtual.text = condicao
 self.labelTemperaturaMinima.text = temp_min.description + "°"
 self.labelTemperaturaMaxima.text = temp_max.description + "°"
 self.pegaImagen(icon)

 self.labelCondicaoAtual.isHidden = false
 self.labelTemperaturaMinima.isHidden = false
 self.labelTemperaturaMaxima.isHidden = false
 }
 }
 }
 }
 }
 }
 }
 }
}

```

```

 }
 }
} catch let erro as NSError {
 print(erro.localizedDescription)
}

}

task.resume()

}

}

```

1. Implemente o método `pegaImagen`

```

```swift
private func pegaImagen(_ icon:String){
    if let endpoint = URL(string: URL_BASE_IMAGE + icon + ".png") {
        let session = URLSession(configuration: .default)
        let task = session.dataTask(with: endpoint){ (data, response, error) in

            if let httpResponse = response as? HTTPURLResponse {

                if httpResponse.statusCode == 200 {
                    DispatchQueue.main.async {
                        print("Exibindo Imagem")
                        self.imageView.image = UIImage(data: data!)
                    }
                }
            }
        }

        task.resume()
    }
}

```

1. Por fim vamos adicionar uma nova entrada no `info.plist` para que seja possível efetuar requisições através do nosso APP.



APÊNDICE A: DISPONIBILIZANDO SEU APLICATIVO NA APPLE APP STORE

Então chegou o momento de deixar o mundo saber sobre o seu aplicativo. Após uma bateria de testes, você decide que está pronto para disponibilizar de graça ou mesmo ganhar algum dinheiro com seu novo programa para iOS. É o momento de mergulhar no processo de aprovação da Apple.

Nesse capítulo, vamos descrever alguns passos necessários que talvez o ajude a caminhar por esse terreno que muitas vezes pode ser um tanto pantanoso.

19.1 APP STORE: A PLATAFORMA OFICIAL DE DISTRIBUIÇÃO DE APLICAÇÕES DA APPLE

A Apple fornece uma plataforma de distribuição que você pode usar para disponibilizar os seus aplicativos para qualquer pessoa que use um dispositivo rodando o iOS. Para que alguém use essa plataforma, é preciso um cadastro chamado **Apple ID**, esse cadastro é utilizado para autenticação no aplicativo *App Store* presente tanto nos computadores que usam o sistema operacional **Mac OS X**, quanto nos dispositivos rodando o **iOS**.



Figura 19.1: App Store para iOS

O aplicativo *App Store* tem funcionalidades para facilitar a busca, compra e instalação de aplicações. Se uma atualização é fornecida pelo fabricante, então o usuário recebe um aviso de que a aplicação pode ser atualizada. Tudo é feito pelo próprio dispositivo.

Quem gerencia e garante a disponibilidade da App Store é a própria Apple e existem regras para que sua aplicação possa ser disponibilizada nessa plataforma.

19.2 BUROCRACIAS E DIFICULDADES PARA TESTAR SUA APLICAÇÃO EM UM DISPOSITIVO

Como vimos no decorrer do curso, o Xcode, plataforma para desenvolvimento de aplicativos para iOS é gratuito. Para fazer o download é preciso a criação de uma conta de desenvolvedor que pode ser feita em uma parte específica do site da Apple chamada *Apple Developer Center*. Após o cadastro, você tem acesso ao conteúdo relacionado com desenvolvimento de aplicativos como documentação, manuais e até mesmo o download de ferramentas como o Xcode.

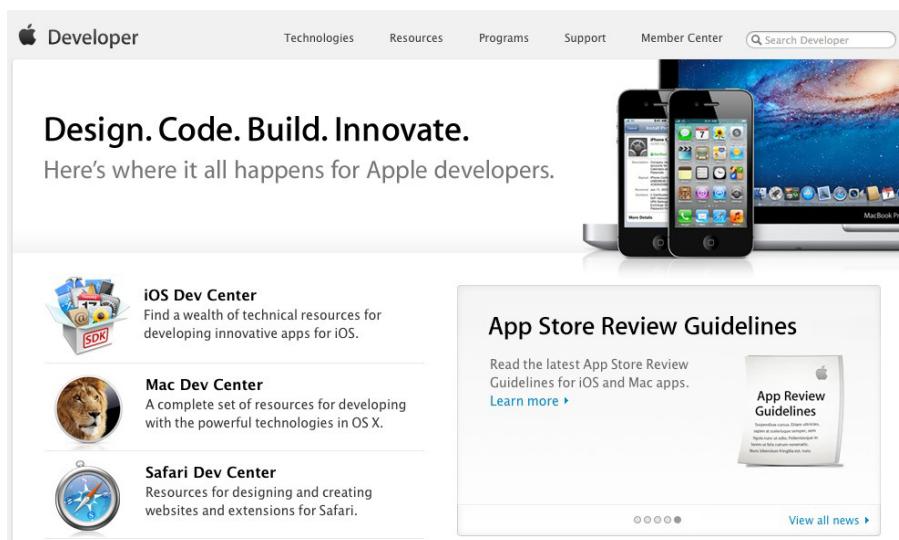


Figura 19.2: Apple Developer Center

Porém, para enviar sua aplicação para a **App Store** é preciso comprar uma licença que dá direito a fazer parte do **Developer Program**. O **iOS Developer Program** custa \$99 por ano. Com essa licença você ganha um certificado digital com o qual é possível assinar suas aplicações, é essa assinatura que permite a instalação de uma aplicação em um dispositivo. Ou seja, para testar o aplicativo construído no Xcode em um aparelho, será preciso fazer parte do **Developer Program**.

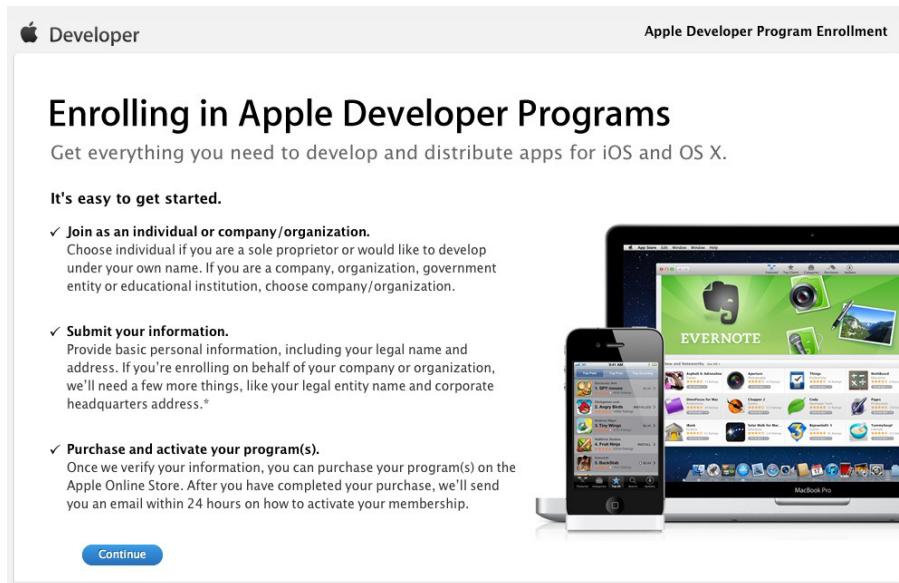


Figura 19.3: Inscrição no Programa de Desenvolvimento

Existem alguns "modelos" de inscrição no programa de desenvolvedor como "Empresa" ou "Individual". Após escolher o "perfil" da sua conta você pode escolher entre associar a conta a um Apple ID existente ou criar um específico para o programa de desenvolvedor.

Figura 19.4: Formulário de Cadastro

Até o momento em que esse texto foi escrito há uma "agradável" surpresa para desenvolvedores do Brasil que se cadastraram no programa. Para confirmar o cadastro e poder realizar o pagamento usando um cartão de crédito internacional, é preciso enviar um fax. Isso mesmo, um fax. De papel! Durante o processo de cadastro, é exibido um link para download do PDF modelo que deve ser utilizado como **template** para o tal fax.

Ao conversar com pessoas que utilizam o programa, a experiência, apesar de um pouco "atípica", é

bem tranquila. Após enviar o fax, em poucas horas ou, no máximo até o próximo dia, um e-mail de confirmação é enviado e você pode seguir com o processo de ativação da licença.

SERVIÇOS ONLINE DE ENVIO DE FAX

Encontrar um aparelho de fax nos dias de hoje não é uma coisa comum em qualquer casa que tenha um telefone. Uma possibilidade é utilizar algum serviço que envia um fax a partir de documentos digitalizados.

Existem inclusive alguns sites que oferecem o serviço gratuitamente em troca de um anúncio no rodapé do documento que será enviado.

Após completar o cadastro no programa, será permitido a criação de um **perfil de testes** no site. Esse perfil pode ser instalado em um dispositivo e, a partir de então, qualquer aplicação assinada com esse perfil pode ser instalada no dispositivo diretamente pelo Xcode.

Após isso, será preciso criar um **iTunes Connect App Record**. Esse cadastro vai permitir a visualização de relatórios relativos às vendas e downloads de seu aplicativo, entre outras informações. É também esse cadastro que permite o envio de uma aplicação para a **App Store**, o interessante é que você só precisa do **iTunes** para tudo isso.

Para distribuir a aplicação, é preciso criar um arquivo específico para a distribuição, isso é feito com o Xcode. É esse arquivo que será usado para publicação por meio do **iTunes Connect**.

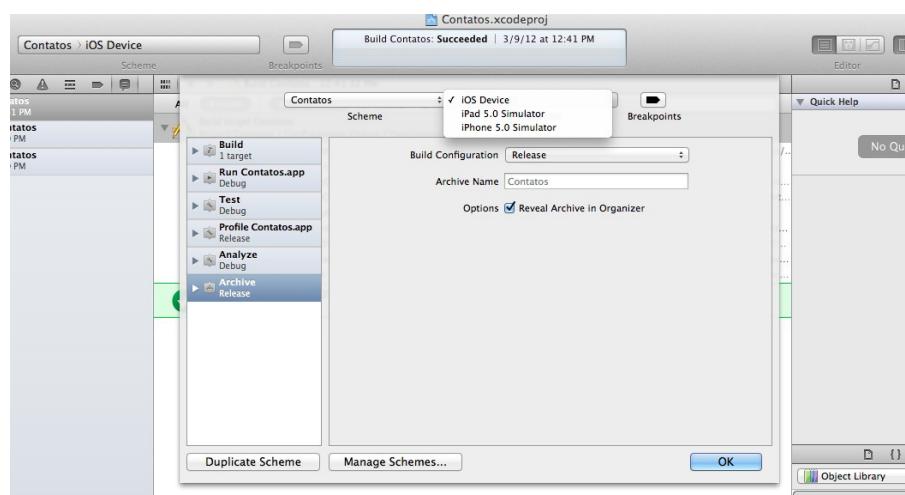


Figura 19.5: Criando arquivo de distribuição

19.3 SOBRE OS GUIDELINES DA APPLE PARA DESENVOLVIMENTO DE APLICAÇÕES IOS

Desde o início da App Store para iOS, a Apple ficou conhecida por ser rigorosa nos quesitos de aprovação de um aplicativo. Hoje parece que as regras estão um pouco menos rígidas, mas, mesmo assim, é preciso ter um mínimo de atenção e cuidado com o desenvolvimento da aplicação, do contrário ela pode ser reprovada para a publicação (claro, isso não o impede de rever os problemas apontados e reenviar o aplicativo para publicação).

Boa parte das regras são referentes à usabilidade da aplicação. A Apple, historicamente, é preocupada com a experiência que os usuários terão ao usar seus aplicativos, os guidelines para desenvolvimento, que são utilizados para avaliar as aplicações submetidas, são reflexos de uma tentativa de manter esse padrão de qualidade.

Podemos olhar para o documento chamado `iOS Human Interface Guidelines` como um conjunto de dicas e ideias que podem garantir uma boa usabilidade a aplicação baseada nisso.

Exemplos de dicas sobre como tratar gesturas fornecidas no documento:

- O gesto de tap deve funcionar como um clique de mouse (para selecionar)
- O duplo tap deve causar zoom in ou out, conforme o estado corrente da aplicação
- O shake, se implementado, deve ser usado como undo e redo

Perceba que você pode usar essas gesturas de outras formas, e não necessariamente ter sua aplicação recusada, o documento lhe fornece informação sobre como a Apple idealizou o próprio sistema para interação com os usuários.

A princípio tudo isso pode parecer um pouco intimidador, mas não se detenha, vale a pena passar pelo processo para ter sua aplicação publicada. Publique suas aplicações e conte-nos sobre seus resultados!