iOS Swift - Frameworks 300

iOS Swift - Frameworks 300

Autoria:
Tiago Santos de Souza
Roberto Rodrigues Junior

Revisão:
Tiago Santos de Souza

Design, edição e produção:
Tiago Santos de Souza

Edição nº3
Janeiro 2018

Sumário

Apresentação

Depois de conhecermos a sintaxe do **Swift** e aprendermos a utilizar os principais recursos relacionados ao desenvolvimento da **interface gráfica** para os nossos aplicativos, chegamos ao ponto onde colocaremos em prática a utilização dos principais **frameworks** disponíveis.

Mas afinal, o que é um **framework?**

Framework nada mais é do que uma reunião de recursos relacionado a um assunto específico.

Durante o módulo 200, trabalhamos especificamente com um framework, o **UIKit.** Esse framework nos auxiliou a construir os objetos de interface que serviram de interação com o usuário e o framework **Foundation.**

Durante o módulo 300, teremos a disposição outros frameworks, com diferentes objetivos. Como exemplo, podemos citar o framework **MapKit** que nos auxiliará no **trabalho com mapas.**

A ideia a partir daí, é tornar possível a construção de aplicativos mais dinâmicos e complexos.

Vamos passar por componentes como **reconhecimentos de gestos, utilização do acelerômetro, geolocalização usando o GPS,** até componentes mais avançados, como **Push Notifications** e armazenamento de dados com **Sandbox.**

Nossa base de desenvolvimento continuará sendo o **Xcode.** Devemos nos atentar apenas para a **importação** correta dos frameworks a serem utilizados.

Quando trabalhamos com diferentes frameworks, cada um se comportará de forma diferenciada. Para alguns, temos a disposição elementos gráficos que implementarão a interface, por outro lado, temos outros frameworks que se baseiam apenas em linhas de código.

Bons estudos!

Capítulo 1

Visualizando prévias de arquivos com QuickLook

Seção 1-1
QuickLook

O framework **QuickLook** é utilizado para exibirmos prévias de arquivos de extensões diversas, seja um **pdf, iWork,** ou arquivo do **Microsoft Office.**

Este framework nos permite mais controle sobre a visualização de arquivos, permitindo que o conteúdo seja exibido no contexto de um navigation controller ou de forma modal.

Podemos exibir os seguintes formatos:

- iWork;

- Documentos Microsoft Office (Office '97 ou posterior);

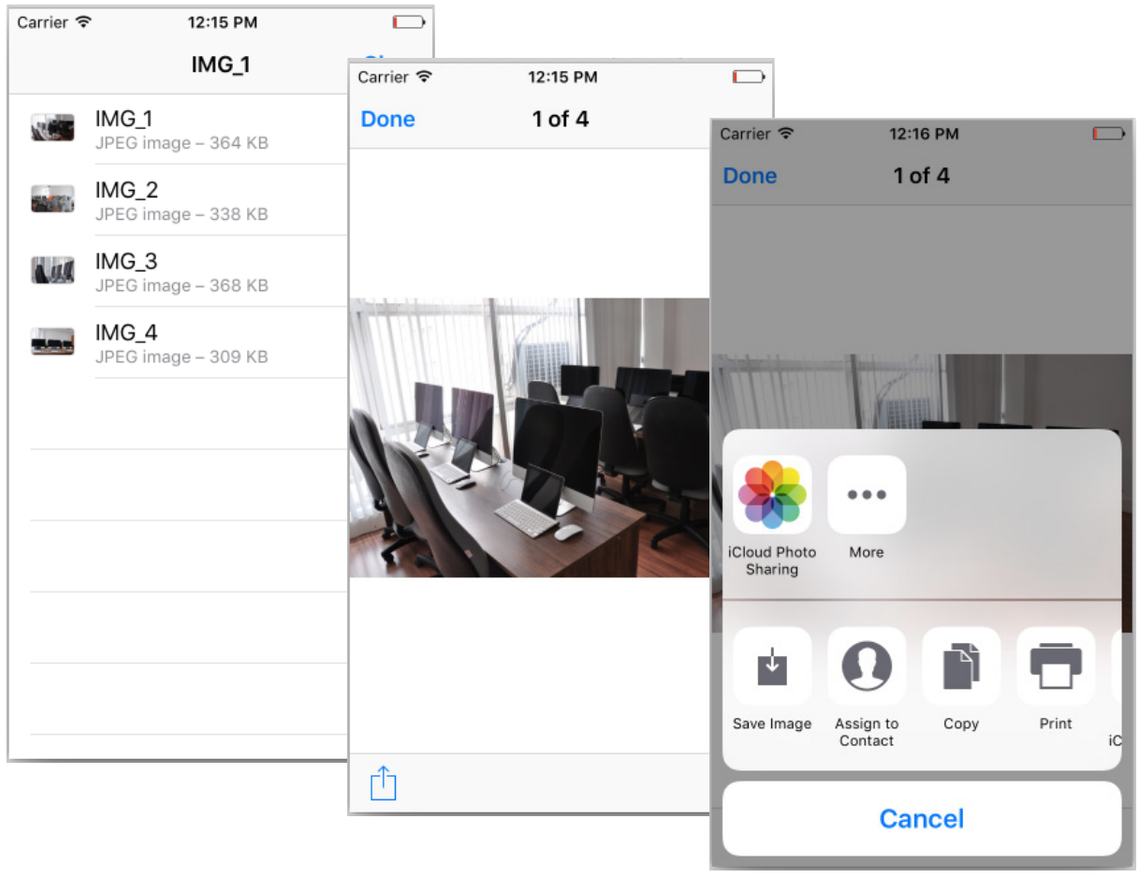
- Documentos Rich Text Format (RTF);

- PDF;

- Imagens;

- Arquivos CSV.

Quando colocamos diversos arquivos para serem pré visualizados em sequência, menus relacionados a navigation controller serão exibidos:



Por ser um framework, o QuickLook não está incluído ao nosso projeto na sua criação, portanto devemos importar o framework logo no inicio do nosso código.

```
//Início do Projeto
import UIKit

//Importando o QuickLook
import QuickLook
```

Seção 1-2

QLPreviewController

Herança: NSObject > UIResponder > UIViewController > QLPreviewController

Um objeto da classe **QLPreviewController** provê uma view para a pré-visualização de itens relacionados ao Quick Look.

Podemos exibir objetos do QLPreviewController através de um **navigationController** ou apresentá-los de maneira modal usando um **presentModalViewController : animated.**

Principais propriedades

Propriedade	Tipo	Descrição
dataSource	QLPreviewController-DataSource?	Define o datasource
delegate	QLPreviewController-Delegate?	Define o delegate
currentPreviewItem	QLPreviewItem?	Retorna o item que está sendo exibido no Quick Look.

Principais métodos

```
//Método que recarrega as informações no QuickLook:
.reloadData()


//Método que atualiza a exibição do item que está
sendo exibido:
.refreshCurrentPreviewItem()
```

```
//Método que retorna se um item pode ser exibido pelo
QuickLook:
.canPreviewItem(item: QLPreviewItem!) -> Bool
```

Protocolos para DataSource e Delegate

A definição dos dados e interações que farão parte de um QLPreviewController faz uso de protocolos **DataSource e Delegate.**

Ambos protocolos devem ser adotados pela classe **View Controller** ou pela classe que será utilizada para controlar a View que conterá o elemento:

```
extension ClasseVController : QLPreviewControllerDataSource{

    //Declarações dos métodos

}


extension ClasseVController : QLPreviewControllerDelegate{

    //Declarações dos métodos

}
```

Métodos de QLPreviewControllerDataSource

```
//Método que retorna ao QuickLook preview controller o
número de itens na preview navigation list:
func numberOfPreviewItems(in controller:
QLPreviewController) -> Int
```

```
//Método chamado quando o QuickLook preview controller
precisa exibir um item específico em uma determinada posição:
func previewController(_ controller: QLPreviewController,
previewItemAt index: Int) -> QLPreviewItem
```

Métodos de QLPreviewControllerDelegate

```
//Método que retorna ao delegate quando um QuickLook preview
está para ser exibido ou retirado para proporcionar um
efeito de zoom:
func previewController(_ controller: QLPreviewController,
frameFor item: QLPreviewItem, inSourceView view:
AutoreleasingUnsafeMutablePointer<UIView?>) -> CGRect
```

```
//Método que retorna ao delegate quando o preview controller
está para ser fechado:
func previewControllerWillDismiss(_ controller:
QLPreviewController)
```

```
//Método que retorna ao delegado quando o preview controller
fechou:
func previewControllerDidDismiss(_ controller:
QLPreviewController)
```

```
//Método chamado pelo QuickLook preview antes de tentar
carregar uma URL:
func previewController(_ controller: QLPreviewController,
shouldOpen url: URL, for item: QLPreviewItem) -> Bool
```

Capítulo 2

Interação com a web

WebKit e SafariServices

WKWebView

Herança: NSObject > UIResponder > UIView > WKWebView

A classe **WKWebView** é utilizada quando precisamos apresentar um conteúdo da web dentro da nossa aplicação. Basicamente, será criada uma view que exibe em seu interior o conteúdo de um endereço web.

Através dessa view também é possível avançar ou regredir no histórico do navegador dentro do app, agregando essas funções a objetos como botões.

Principais propriedades

Propriedade	Tipo	Descrição
isLoading	Bool	Indica se um conteúdo está sendo carregado
canGoBack	Bool	Indica se a WebView é capaz de retornar para uma página anterior
canGoFoward	Int	Indica se a WebView é capaz de avançar para uma página posterior
navigationDelegate	WKNavigation Delegate?	Define/Retorna o objeto delegate

Principais métodos

```
//Método que carrega um conteúdo na webview a partir de uma
requisição:
.load(request: URLRequest)


//Método que para o carregamento de qualquer conteúdo que
esteja sendo carregado pela Webview:
.stopLoading()
```

```
//Método que carrega a página anterior da lista de páginas
acessadas na Webview:
.goBack()


//Método que carrega a próxima página da lista de páginas
acessadas na Webview:
.goForward()


//Método que recarrega o conteúdo da Webview:
.reload()
```

Protocolo de WKNavigationDelegate

A WKWebView utiliza um protocolo **delegate**. Esse protocolo nos auxilia a realizar tarefas como comunicar quando um conteúdo de uma pagina inicia o carregamento, ou quando o conteúdo termina de ser carregado.

O protocolo deve ser adotado pela classe **View Controller** ou pela classe que será utilizada para controlar o elemento:

```
extension ViewController : WKNavigationDelegate {

      //Declarações dos métodos

}
```

Principais métodos de WKNavigationDelegate

```
//Método que comunica o delegate que o conteúdo começou a
ser carregado na webView:
func webView(_ webView: WKWebView,
didStartProvisionalNavigation navigation: WKNavigation!)
```

```
//Método que comunica o delegate que o conteúdo de uma
webView terminou de ser carregado:
func webView(_ webView: WKWebView, didFinish navigation:
WKNavigation!)
```

```
//Método que comunica o delegate caso ocorra algum erro ao
carregar um conteúdo:
func webView(_ webView: WKWebView,
didFailProvisionalNavigation navigation:
WKNavigation!, withError error: Error)
```

Seção 2-2

Safari Services

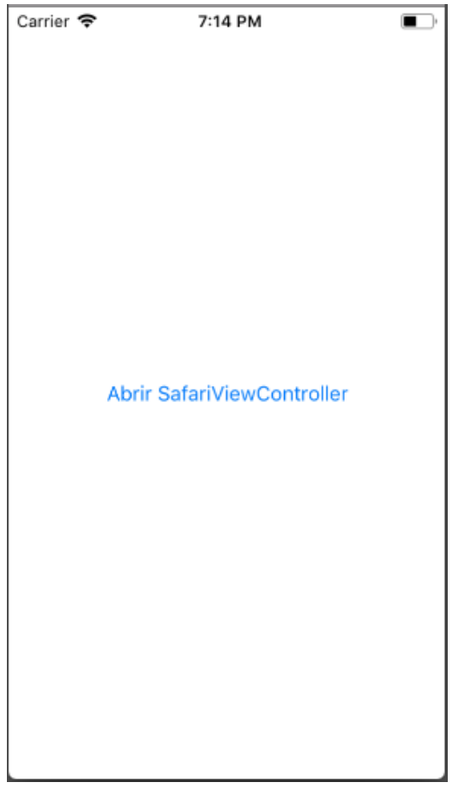
Utilizamos o framework **Safari Services** para integrar comportamentos do navegador **Safari** da Apple ao nosso aplicativo.

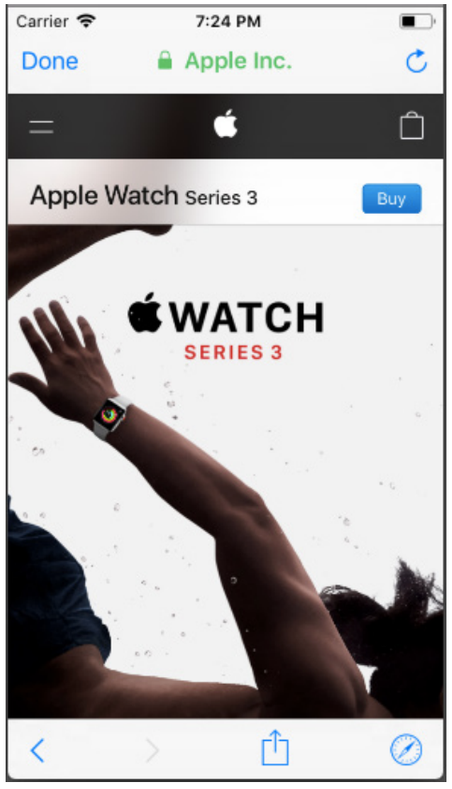
A partir do momento que acessamos uma **SFSafariViewController**, temos a disposição uma interface idêntica aquela fornecida pelo aplicativo Safari, porém mais dinâmica e limpa.

Os usuários podem navegar na web a partir o objeto SFSafariViewController, podendo retornar ao conteúdo do seu aplicativo no momento que quiser.

Por ser um framework, o Safari Services não está incluído ao nosso projeto na sua criação, portanto devemos importar o framework logo no inicio dos arquivos de classe que acessarão os recursos da estrutura.

```
//Início do Projeto
import UIKit

//Importando o Safari Services
import SafariServices
```





Seção 2-3
SFSafariViewController

Herança: NSObject > UIResponder > UIViewController > SFSafariViewController

A classe **SFSafariViewController** inclui recursos do navegador Safari, tais como Reader, preenchimento automático, detecção de sites fraudulentos, e bloqueio de conteúdo. Ele compartilha cookies e outros dados do site com o aplicativo Safari instalado no dispositivo.

A atividade do usuário e interação com SFSafariViewController não são visíveis para a sua aplicação, que não terá acesso a dados de preenchimento automático, histórico de navegação, ou dados do site.

Principais propriedades

Propriedade	Tipo	Descrição
preferredControl-TintColor	UIColor	Define a cor dos botões de controle da barra de navegação
preferredBar-TintColor	UIColor	Define a cor de fundo para a barra de navegação e ferramentas
delegate	SFSafariViewController-Delegate?	Define/Retorna o objeto delegate

Protocolo de SFSafariViewControllerDelegate

A **SFSafariViewController** utiliza o protocolo **delegate** para implementar eventos personalizados de manipulação para uma view. O protocolo deve ser adotado pela classe **View Controller** ou pela classe que será utilizada para controlar o elemento:

```
extension ClassVController: SFSafariViewControllerDelegate{

      //Declaração dos métodos
}
```

Principais métodos de SFSafariViewControllerDelegate

```
//Método que informa ao delegate que um endereço URL foi
totalmente carregado:
func safariViewController(_ controller:
SFSafariViewController, didCompleteInitialLoad
didLoadSuccessfully: Bool)
```

```
//Método que informa ao delegate um botão de ação foi
pressionado pelo usuário:
func safariViewController(_ controller:
SFSafariViewController, activityItemsFor URL: URL, title:
String?) -> [UIActivity]
```

```
//Método que informa ao delegate quando o usuário deixou a
SFSafariViewController:
func safariViewControllerDidFinish(_ controller:
SFSafariViewController)
```

Capítulo 3

Capturando e resgatando fotos

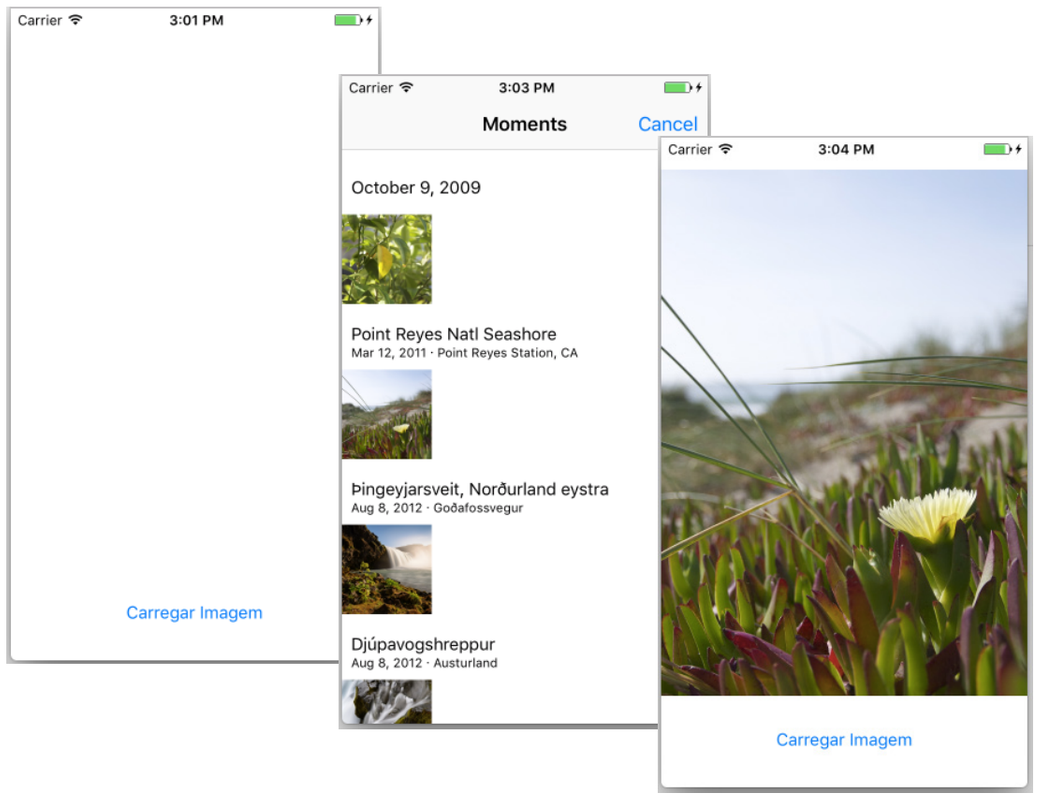
Seção 3-1

UIImagePickerController

Herança: NSObject > UIResponder > UIViewController > UINavigationController > UIImagePickerController

A classe **UIImagePickerController** gerencia o sistema de captura de imagens e vídeos em dispositivos que suportem esses recursos. Além disso, a classe oferece o acesso as mídias salvas anteriormente para uso dentro do aplicativo.

Podemos escolher se o **UIImagePickerController** será utilizado para capturar fotos ou apresentar as já salvas. Essa escolha é feita através da propriedade **sourceType.**



Principais propriedades

Propriedade	Tipo	Descrição
cameraCaptureMode	UIImagePickerController-CameraCaptureMode	Define/Retorna o tipo de captura
cameraDevice	UIImagePickerController-CameraDevice	Define/Retorna a câmera usada
cameraFlashMode	UIImagePickerController-CameraFlashMode	Define/Retorna o tipo de flash
delegate	UIImagePickerController-Delegate? e UINavination-ControllerDelegate?	Define/Retorna o objeto delegate
sourceType	UIImagePickerController-SourceType	Define/Retorna o tipo do ImagePicker (captura ou resgate de imagens)
mediaTypes	[String]	Define/Retorna um array com tipos de mídia a ser acessado

Protocolo de UIImagePickerControllerDelegate

A **UIImagePickerController** utiliza o protocolo **UIImagePickerControllerDelegate** para avisar quando uma imagem é selecionada ou ocorre o cancelamento da seleção. O protocolo deve ser adotado pela classe **View Controller** ou pela classe que será utilizada para controlar o elemento.

Como o **UIImagePicker** trabalhará com a manipulação de Navigation Controller, também é importante a utilização do protocolo **UINavigationControllerDelegate.**

```
extension ClasseViewController :  UIImagePicker-
ControllerDelegate {

      //Declaração dos Métodos
}


extension ClasseViewController :  UINavigationController-
Delegate  {

      //Declaração dos Métodos
}
```

Principais métodos de UIImagePickerControllerDelegate

```
//Método que notifica ao delegate que o usuário selecionou
uma imagem:
func imagePickerController(_  picker: UIImagePicker-
Controller, didFinishPickingMediaWithInfo info:
[String : Any])

// Método que notifica ao delegado que o usuário cancelou a
seleção de uma imagem:
func imagePickerControllerDidCancel(_ picker:
UIImagePickerController)
```

Principais métodos

```
//Método que retorna se uma determinada câmera do dispositivo
está disponível:
class func isCameraDeviceAvailable (UIImagePicker-
ControllerCameraDevice) -> Bool


//Método que retorna se a câmera selecionada no dispositivo
possuí flash disponível:
class func isFlashAvailable(for cameraDevice:
UIImagePickerControllerCameraDevice) -> Bool


//Método que inicia captura de vídeo:
func startVideoCapture() -> Bool


//Método que encerra a captura do vídeo:
func stopVideoCapture() -> Bool


//Método que caputra uma foto:
func takePicture()
```

#Dica!

É importante salientar que o acesso do usuário à câmera e biblioteca de fotos só serão permitidas a partir do momento que essas autorizações forem solicitadas ao arquivo **info.plist.**

Capítulo 4

Reconhecimento de gestos

Seção 4-1

UIGestureRecognizer

Herança: NSObject > UIGestureRecognizer

Apesar do suporte a toques a partir de objetos **UITouch** presentes nos métodos de **UIViewController,** existem limitações que dificultam o desenvolvimento de aplicativos que necessitem de interações de toque mais avançadas. Para preencher essa lacuna e fornecer mais recursos, foram criados os gestos: elementos controlados pela classe **UIGestureRecognizer** e classes da sua cadeia hierárquica

Uma das principais vantagens do uso de gestos é a associação direta ao objeto que será tocado. Isso além de facilitar a programação, enxuga os códigos necessários para essas interações.

Outra vantagem dessa abordagem é o fato do usuário ter acesso a gestos mais complexos do que simples toques, como os **gestos de pinça (zoom), giro (rotate) e arrastar (pan).**

Existem seis classes padrões para controlar e responder a gestos:

- UITapGestureRecognizer: Evento de tocar o objeto de forma breve;

- UILongPressGestureRecognizer: Evento de pressionar e manter o pressionamento sobre o objeto;

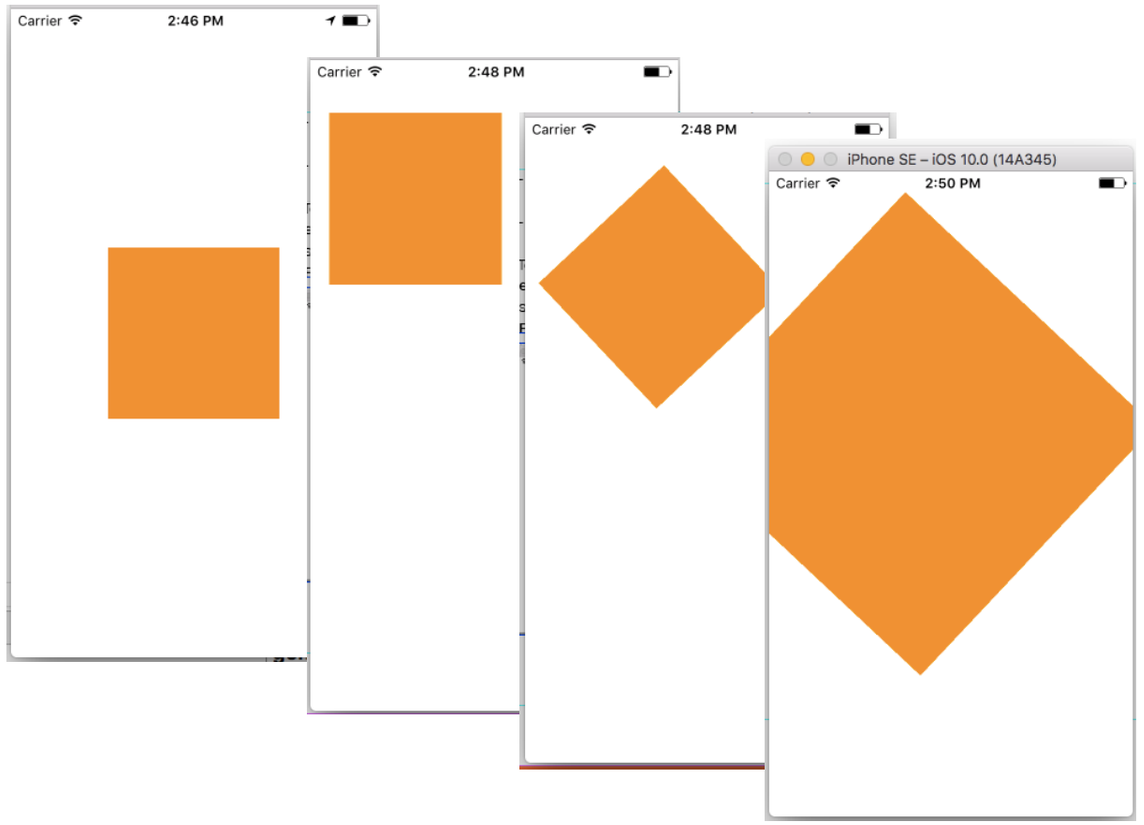
- UIPanGestureRecognizer: Evento de arrastar o toque do objeto;

- UIPinchGestureRecognizer: Evento de escalonar o objeto;

- UISwipeGestureRecognizer: Evento de deslizar sobre o objeto;

- UIRotationGestureRecognizer: Evento de girar o objeto.

Todos os gestos possuem propriedades para definir a **quantidade de toques, delegate** e seu **estado atual.** Além disso, cada um tipo possui seus recursos como valores de transformação para **Pinch** e **Rotation**, posição para **Pan** e direção para **Swipe.**



Principais propriedades

Propriedade	Classe	Tipo	Descrição
numberOfTapsRequired	UITapGestureRecognizer UILongPressGestureRecognizer	Int	Define / Retorna a quantidade de taps necessários para o gesto ser reconhecido
numberOfTouchesRequired	UITapGestureRecognizer UISwipeGestureRecognizer UILongPressGestureRecognizer	Int	Define / Retorna a quantidade de dedos necessários para o gesto ser reconhecido
scale	UIPinchGestureRecognizer	CGFloat	Define/Retorna o fator de escala em relação aos pontos dos dois toques em coordenadas de tela
velocity	UIPinchGestureRecognizer UIRotationGestureRecognizer	CGFloat	Retorna a velocidade do pinch no fator de escala por segundo
rotation	UIRotationGestureRecognizer	CGFloat	Define/Retorna a rotação do gesto em graus radianos
direction	UISwipeGestureRecognizer	UISwipeGesture-RecognizerDirection	Define/Retorna a direção para o reconhecimento do gesto swipe
maximumNumberOfTouches	UIPanGestureRecognizer	Int	Define / Retorna a quantidade mínima de dedos necessários para o gesto ser reconhecido
minimumNumberOfTouches	UIPanGestureRecognizer	Int	Define / Retorna a quantidade máxima de dedos a ser utilizada no gesto
minimumPressDuration	UILongPressGestureRecognizer	CFTimeInterval	Define / Retorna a quantidade mínima de tempo que os dedos devem permanecer em toque para que o gesto seja reconhecido

Protocolo de UIGestureRecognizerDelegate

A **UIGestureRecognizerDelegate** utiliza o protocolo **delegate** para afinar o comportamento de reconhecimento de gestos em um aplicativo. O protocolo deve ser adotado pela classe **View Controller** ou pela classe que será utilizada para controlar a View que conterá os gestos.

```
extension ClasseViewController : UIGestureRecognizer-
Delegate{

      //Declaração dos métodos
}
```

Temos a disposição a propriedade **delegate,** do tipo **UIGestureRecognizerDelegate?** que informa todos os eventos de gestos causados na aplicação.

Principais métodos de UIGestureRecognizerDelegate

```
//Método que indica ao delegate que um reconhecedor de
gestos deve começar a interpretar toques:
func gestureRecognizerShouldBegin(_ gestureRecognizer:
UIGestureRecognizer)


//Método que indica ao delegate se um gesto de reconhecimento
deve receber um objeto representando um toque:
func gestureRecognizer(_ gestureRecognizer:
UIGestureRecognizer, shouldReceive touch: UITouch) -> Bool

//Método que indica ao delegado se reconhecedores devem ser
autorizados a reconhecer gestos simultaneamente:
func gestureRecognizer(_ gestureRecognizer:
UIGestureRecognizer, shouldRecognizeSimultaneously-
With otherGestureRecognizer: UIGestureRecognizer) -> Bool
```

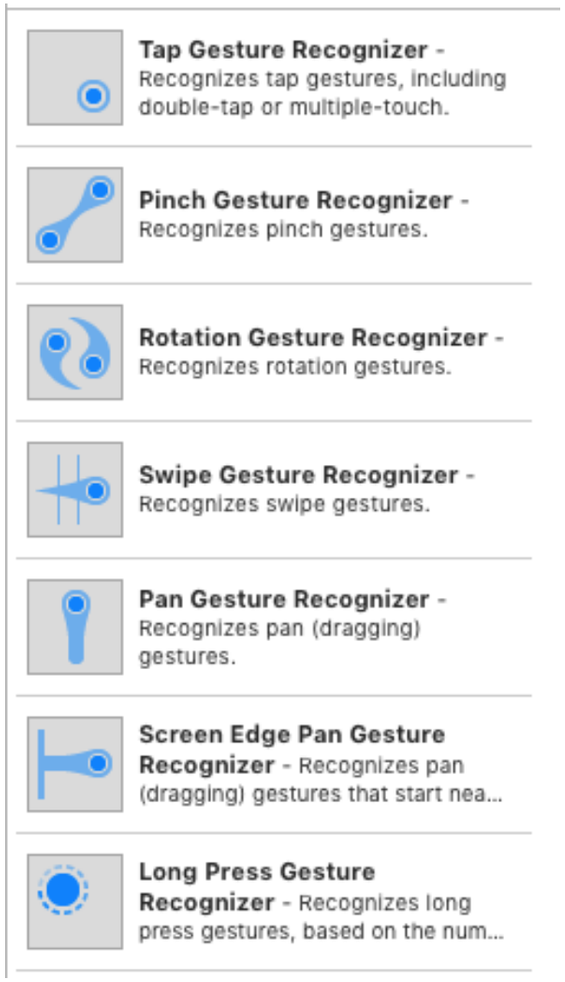
Principais métodos

```
// Método que associa um objeto UIGestureRecognizer a um
objeto UIView com o seletor definido:
func addTarget(_ target: Any,  action: Selector)

//Método que retorna a posição do toque na View definida:
func location(in view: UIView?) -> CGPoint


//Método que remove uma associação do gesto com o objeto:
func removeTarget(_ target: Any?, action: Selector?)
```

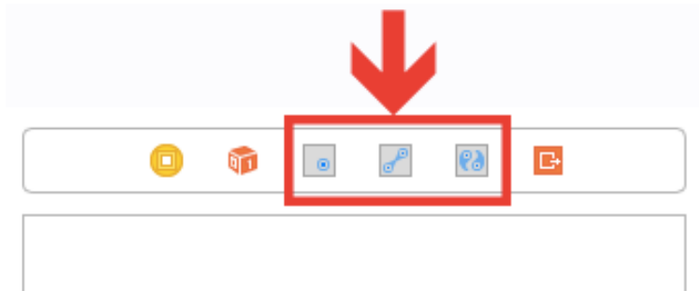
Gestos aplicados a interface

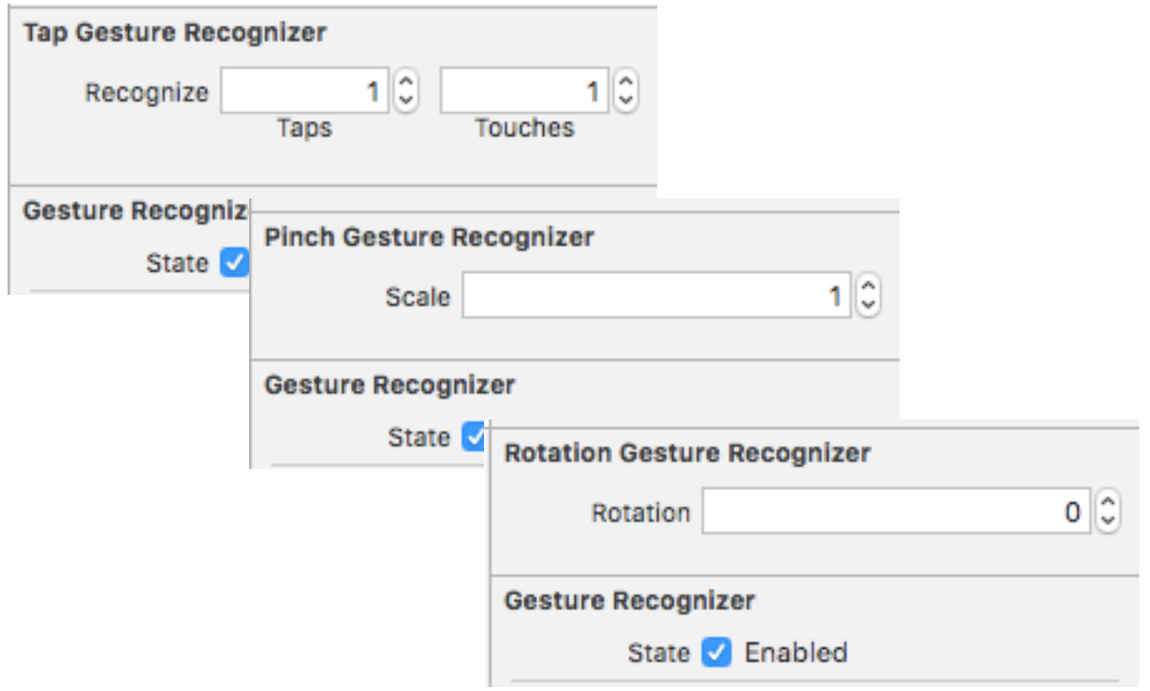
Além de trabalhar com as classes de reconhecimento de gestos de forma programática, podemos dispor os gestos diretamente na interface gráfica.

Podemos arrastar os elementos de reconhecimento de gestos disponíveis e soltá-los sobre os objetos que se deseja interagir.



Quando arrastamos um elemento de reconhecimento de gestos para um objeto da interface, esse elemento fica disponível para manipulação, integração com o código e atribuição de propriedades junto ao título da View Controller:





Capítulo 5

Gravando e carregando arquivos

Seção 5-1
Sandbox

Dependendo do tipo de aplicativo que desenvolvemos, precisamos fazer o **armazenamento local de informações.**

Seja para guardar um dado da fase em que o usuário parou em um jogo, ou então para armazenar um perfil de usuário, o recurso de salvar arquivos em de forma local se mostra muito importante e recorrente.

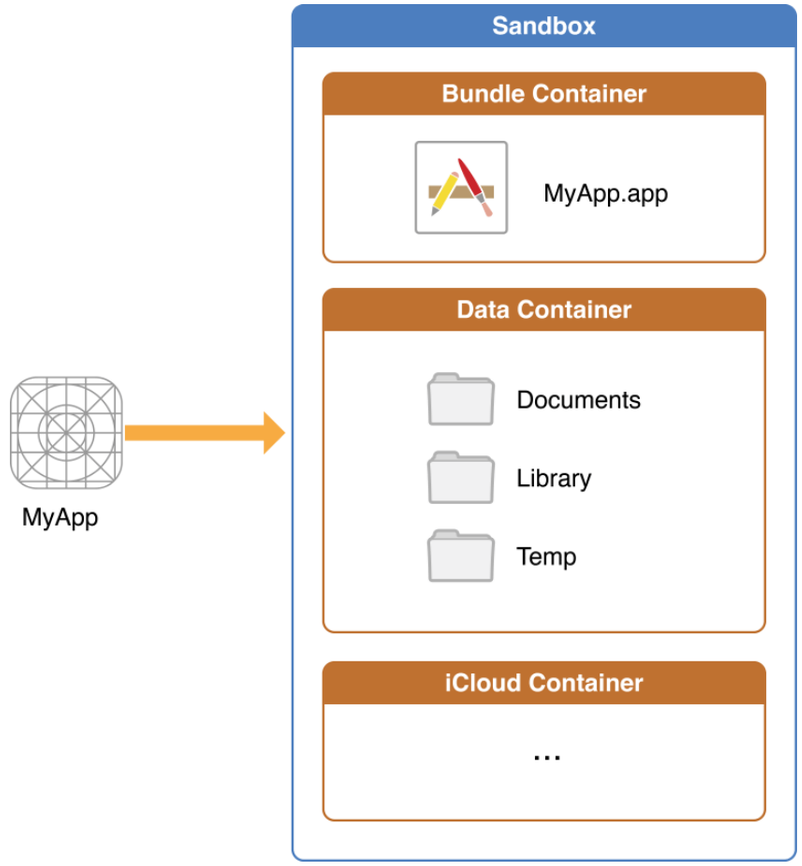
Existe todo um **sistema de segurança** que deve ser levado em consideração quando se faz gravação de dados no dispositivo, e nesse capítulo vamos trabalhar com o conceito de **Sandbox.**

Sandbox é o nome de um modelo de segurança implementado nos **aplicativos iOS** que impede que uma aplicação tenha acesso irrestrito a partes críticas dos dados tanto do sistema operacional, quanto do usuário.

O uso de Sandbox é uma das técnicas que garante a alta taxa de segurança de aplicativos iOS em comparação a outras plataformas computacionais.

Sua estrutura é definida para que um projeto tenha algumas pastas controladas para armazenar, gravar e ler arquivos.

O modelo de Sandbox divide um aplicativo em **quatro** diretórios principais, chamados de **Data Container,** são eles: **NomeDoAplicativo.app, Documents, Library e Temp.**



O diretório do **NomeDoAplicativo.app** concentra os recursos necessários para a compilação e geração do projeto final. Nele estão inclusos arquivos de certificado digital e assinaturas eletrônicas.

O diretório recomendável para armazenamento de arquivos em tempo de execução é o **Documents.** Quando um aplicativo tem a demanda de criar um arquivo de perfil ou configurações, salvar uma imagem ou mesmo um arquivo de banco de dados, a pasta a ser usada será a **Documents.**

#Dica!

Os arquivos salvos no diretório **Documents** podem ser sincronizados com o **iCloud.**

A pasta **Library** representa o nível superior para todos os arquivos que não são de dados do usuário. Ou seja é o local onde os aplicativos e outros módulos de código armazenam seus arquivos de dados personalizados.

Por ultimo temos a pasta **Temp,** que deve ser utilizada para gravar arquivos temporários que não precisam persistir entre os fechamentos e aberturas de seu aplicativo.

Seu aplicativo deve remover arquivos a partir deste diretório quando eles não são mais necessários, no entanto o sistema pode purgar esta pasta quando o seu aplicativo não está em execução.

Seção 5-2

Acessando o diretório Documents

Agora que sabemos que o diretório **Documents** é o local recomendado para guardar seus arquivos, vamos ver como acessar essa pasta:

Funcão para acessar o diretório raiz do Sandbox

A Função `NSHomeDirectory()` retorna uma **string** com o caminho até a raiz de diretórios do **Sandbox** do aplicativo.

```
func NSHomeDirectory() -> String
```

Podemos criar um objeto para armazenar esse caminho:

```
let caminhoRaiz = NSHomeDirectory()
```

Função para concatenar um caminho a raiz

A função `appendingPathComponent` é utilizada para complementar um determinado caminho a uma string.

```
func appendingPathComponent(_ str: String) -> String
```

Podemos utilizar essa função para acessar a pasta Documents:

```
let caminhoParaDocuments = (caminhoParaRaiz as NSString).
appendingPathComponent("Documents")
```

#Dica!

Para comprovar a eficácia dos itens indicados, recomenda-se um print do `caminhoParaDocuments,` e posterior abertura do endereço no Finder.

Seção 5-3

FileManager

Herança: NSObject > FileManager

A classe **FileManager** é responsável pelo gerenciamento de arquivos e diretórios no armazenados de forma interna em dispositivos iOS. Entre outros recursos, podemos criar/apagar/renomear/mover tanto arquivos quanto diretórios.

Principais métodos

```
//Método que retorna qual é o diretório home do usuário
específico:
func homeDirectory(forUser userName: String) -> URL?


//Método que cria um diretório com atributos no caminho
especificado:
func createDirectory(atPath path: String,
withIntermediateDirectories createIntermediates: Bool,
attributes: [String : Any]? = nil) throws


//Método que remove o arquivo ou pasta do caminho definido:
func removeItem(atPath path: String) throws


//Método que retorna um objeto compartilhado no sistema:
func defaultManager() -> FileManager


//Método que retorna o conteúdo do arquivo definido:
func contents(atPath path: String) -> Data?


//Método que cria um arquivo no caminho especificado e com
o conteúdo definido:
func createFile(atPath path: String, contents data: Data?,
attributes attr: [String : Any]? = nil) -> Bool


//Método que retorna o nome do arquivo do caminho definido:
func componentsToDisplay(forPath path: String) -> [String]?


//Método que retorna se o arquivo do caminho definido existe:
func fileExists(atPath path: String) -> Bool


//Método que copia um item de um caminho específico para
outro caminho:
func copyItem(atPath srcPath: String, toPath dstPath:
String) throws


//Método que move um item de um caminho específico para
outro caminho:
func moveItem(atPath srcPath: String, toPath dstPath:
String) throws
```

#Dica!

Os métodos com final **thows** são aqueles que podem ter seus possíveis erros captados e tratados. Para tal, utilize a estrutura de tratamento de erros `do{} catch{}`.

Capítulo 6

Multiprocessamento

Seção 6-1

Multiprocessamento

Dependendo das rotinas que seu aplicativo implemente, alguns blocos de código podem ter um **tempo de execução** mais **demorado** do que outros blocos.

Sem o devido tratamento, uma rotina que demore para ser concluída pode **travar** completamente a **interface** da sua aplicação.

Devemos entender que não acontecerá o travamento por erro de código, mas sim o **congelamento** da interface gráfica enquanto determinada tarefa está sendo resolvida.

Quando nossa aplicação chega a esse ponto, surge a necessidade de indicarmos **processamentos paralelos,** ou seja, a necessidade da nossa aplicação trabalhar com **multiprocessamento** de tarefas.

Podemos estipular filas (**queues**) de tarefas a serem executadas, tanto em primeiro quanto em segundo plano

Nesse capítulo vamos abordar os conceitos necessários para adotar **multiprocessamento** em seus aplicativos implementando códigos que sejam executados paralelamente às rotinas principais.

Main Queue

Para entendermos a necessidade de multiprocessamento, vamos detalhar como o processamento das informações acontece:

De maneira resumida, um programa de computador (ou um aplicativo para iOS) pode ser dividido em duas categorias principais: **processo e subprocesso.**

Definimos como **processo** o programa ou aplicativo em si. Aqui está contido todo o **core** do seu programa, pois quando um processo carrega, o aplicativo abre. Se o processo é encerrado, o aplicativo é fechado.

Um dos termos que usamos para esse processo é **Main Queue,** por representar essa execução contínua dos códigos do aplicativo.

Podemos dizer que a **Main Queue** concentra as **principais tarefas** do aplicativo, inclusive o controle da interface gráfica.

Processos paralelos (Global)

Vamos imaginar uma situação que além de todo o código que já está rodando no seu aplicativo, você precise de uma nova tarefa, como fazer o download ou equalizar um vídeo por exemplo.

De maneira simplista, vamos imaginar o **Main Queue** girando o tempo todo para realizar as suas **tarefas essenciais.**

Se você adicionar uma rotina que exija um processamento mais longo, esse processo vai **parar o giro** do Main Queue para poder realizar essa **nova tarefa.** Nessa situação a interface gráfica será congelada, e sua aplicação ficará **travada** até que este processamento seja concluído.

Para resolver este problema precisamos de uma execução fora do loop principal, para isso usamos um espaço que podemos denominar de **Global.**

A ideia é que essa **tarefa** não influencie o loop principal, pois ela roda em **outro local** do processador, o espaço **Global,** e consequentemente chamamos a nova tarefa de **thread.**

Com essa estrutura, podemos rodar códigos em segundo plano sem influenciar o Main Queue.

Seção 6-2

GCD (Grand Central Dispatch)

GCD (sigla para **Grand Central Dispatch**) é uma coleção de recursos para tornar ainda mais fácil e poderoso o trabalho com multiprocessamento.

Implementando **GCD** em nossos projetos, temos acesso a execução de **código em múltiplos núcleos,** controles para **tarefas síncronas e assíncronas,** gerenciamento dos espaços **Main** e **Global**, entre outras vantagens.

Um dos pontos principais do GCD é que ele controla internamente o uso de **múltiplos núcleos de processamento.** Se o dispositivo possuir este recurso, o GCD vai procurar otimizar o máximo de performance que puder.

DispatchQueue

Um dos conjuntos de recursos provenientes do GCD é o **DispatchQueue,** com eles podemos gerenciar a execução de itens de trabalho. Cada item será submetido a uma fila (**queue**) e processados em um pool de **threads** gerenciados pelo sistema.

Com o **DispatchQueue** podemos gerenciar tanto os recursos executados na **Main** ou em **Global Queue.**

Os processos executados na **Main** podem ser gerenciados utilizando a propriedade **.main** no GCD.

```
class var main: DispatchQueue { get }
```

Já os processos executados no **Global** podem ser gerenciados utilizando o método **.global** no GDC.

```
class func global(qos: DispatchQoS.QoSClass = default)
-> DispatchQueue
```

É importante salientar, que os processos de renderização da interface gráfica estão todos em Main Queue. Mais adiante, quando tratarmos de processos que envolvam o resgate de algumas informações via Web, alguns dos métodos propostos terão seus resultados todos processados em Global Queue.

Quando um processo é executado em Global Queue, e esse resultado precisa ser refletido em elementos da interface gráfica, é necessário que o resultado pro processo seja devolvido para Main Queue.

Trabalhando com Threads

Herança: NSObject > Thread

Um objeto **Thread** controla um segmento de execução. O trabalho com Threads pode ser usado para dividir um grande trabalho em diversos trabalhos menores, o que pode levar a um aumento de desempenho em dispositivos multi-core.

Principais pripriedades

Propriedade	Tipo	Descrição
isExecuting	Bool	Retorna quando um segmento está sendo executado
isFinished	Bool	Retorna quando um segmento acabou de ser executado
isCancelled	Bool	Retorna quando a execução de um segmento foi cancelada

Principais métodos

```
//Método que bloqueia a thread até uma data específica:
class func sleep(until date: Date)


//Método que bloqueia a thread por um espaço de tempo
especificado em segundos:
class func sleep(forTimeInterval ti: TimeInterval)
```

```
//Método que cancela a thread:
func cancel()
```

Capítulo 7

Lendo dados XML e JSON

Seção 7-1

Trabalhando com dados externos

Dependendo do aplicativo que se desenvolva, precisamos carregar dados externos ao nosso código. Neste tipo de situação o ideal é carregarmos estes dados através de arquivos **XML** ou **JSON.**

Tanto o **XML** como o **JSON,** são utilizados para compartilharmos dados entre diferentes aplicações e podem ser utilizados no desenvolvimento **iOS** tanto de forma local, como um arquivo no **Bundle (pacote),** como de forma remota, com um arquivo proveniente da internet.

O **XML** (eXtensible Markup Language) é uma linguagem de marcação para a criação de documentos com dados organizados de forma hierárquica e que permite definir os elementos de marcação.

JSON (Javascript Object Notation) é um tipo de arquivo de texto que tem como função facilitar a troca de dados entre sistemas.

O objetivo do **JSON** é similar ao do **XML,** no aspecto de fornecer uma forma de trocar dados entre sistemas. Entretanto, a sua estrutura visa um arquivo mais leve e conciso. Por este motivo, nos últimos anos as aplicações para dispositivos portáteis usam cada vez mais JSON.

Temos a disposição alguns recursos para a transposição de dados provenientes desses arquivos. Caso seja necessário, podemos transformar dados de Swift em dados de linguagem de marcação.

XMLParser

Herança: NSObject > XMLParser

A classe responsável por realizar a leitura e a analise de dados provenientes de arquivos XML é a **XMLParser**.

Os objetos da classe XMLParser são responsáveis por analisar um documento XML. A Classe possuí o protocolo **XMLParserDelegate** que retorna os itens analisados em um documento XML.

O protocolo deve ser adotado pela classe **View Controller** ou pela classe que será utilizada para controlar a View que conterá os dados XML.

```
extension  ClasseViewController : XMLParserDelegate {
      // Declaração dos métodos
}
```

Principais pripriedades

Propriedade	Tipo	Descrição
delegate	XMLParserDelegate ?	Define o delegado para receber mensagens sobre o processo de analise

Principais métodos

```
//Método que inicia a analise de um XML com um NSData
especificado:
init(data: Data)
```

```
//Método que inicia a analise de um XML com uma URL
especificada:
convenience init?(contentsOf url: URL)
```

```
//Método que retorna um booleano dizendo se a analise foi
ou não realizada:
func parse() -> Bool
```

Principais métodos XMLParserDelegate

```
//Método que comunica o delegado que um documento começou
a ser analisado:
optional func parserDidStartDocument(_ parser: XMLParser)
```

```
//Método que comunica o delegado quando um documentoterminou
de ser analisado:
optional func parserDidEndDocument(_ parser: XMLParser)
```

```
//Método que retorna ao delegate quando em análise de um XML
é encontrada uma tag inicial de um determinado elemento:
optional func parser(_ parser: XMLParser, didStartElement
elementName: String, namespaceURI: String?, qualifiedName
qName: String?, attributes attributeDict: [String : String]
= [:])
```

```
//Método que retorna ao delegate quando em análise de um XML
é encontrada uma tag final de um determinado elemento:
optional func parser(_ parser: XMLParser, didEndElement
elementName: String, namespaceURI: String?, qualifiedName
qName: String?)
```

```
//Método que retorna ao delegate uma sequência de caracteres
que representa todos ou parte dos caracteres do elemento
atual:
optional func parser(_ parser: XMLParser, foundCharacters
string: String)
```

```
//Método que retorna ao delegate uma sequência algum erro
encontrado ao interpretar o documento:
optional func parser(_ parser: XMLParser, parseErrorOccurred
parseError: Error)
```

Seção 7-3
JSONSerialization

Herança: NSObject > JSONSerialization

Utilizamos a classe **JSONSerialization** em Swift para converter arquivos **JSON** em objetos **Foundation**, e vice versa.

Principais métodos

```
//Método que retorna um objeto Foundation a partir dos
dados de um JSON:
func jsonObject(with data: Data, options opt:
JSONSerialization.ReadingOptions = []) throws -> Any

//Método que retorna dados JSON a partir de um objeto
Foundation
func data(withJSONObject obj: Any, options opt:
JSONSerialization.WritingOptions = []) throws -> Data
```

#Dica!

Os métodos com final **thows** são aqueles que podem ter seus possíveis erros captados e tratados. Para tal, utilize a estrutura de tratamento de erros `do{} catch{}`.

Codable

Uma das grandes novidades na versão 4.0 da linguagem Swift foi o typealias **Codable.** O Codable envolve dois protocolos responsáveis por fazer a serialização de deserialização de dados de forma simples e eficaz.

Basicamente podemos serializar um JSON diretamente dentro de um objeto de Swift, ou podemos transformar objetos em JSON.

Para tal, basta que a estrutura orientada a objeto tenha adotado o typealias Codable:

```
struct Endereco : Codable {
    let cep : String
    let logradouro : String
}
```

Instancias desse objeto serão capazes de receber ou enviar dados para JSON.

Classe JSONEncoder

Podemos utilizar objetos da classe **JSONEncoder** para transformar dados internos da nossa aplicação em JSON.

Utilizaremos como base a estrutura já descrita anteriormente:

```
//Externando um JSON a partir de um Objeto
let enderecoSaida = Endereco(cep: “01418100”, logradouro:
“Alameda Santos”)
let encoderJSON = JSONEncoder()
let json = try! encoderJSON.encode(enderecoSaida)

print(String(data: json, encoding: String.Encoding.utf8)!)

//Resultado: {“cep”:”01418100”,”logradouro”:”Alameda
Santos”}
```

Classe JSONDecoder

Podemos utilizar objetos da classe **JSONDecoder** para serializar dados JSON e importá-los para nossa aplicação. Utilizaremos como base a estrutura já descrita anteriormente:

```
//Serializando um JSON para dentro da aplicação
let meuJSON = URL(string: “https://viacep.com.br/
ws/01418100/json/”)
let decoderJSON = JSONDecoder()
let enderecoFinal = try! decoderJSON.decode(Endereco.self,
from: Data(contentsOf: meuJSON!))

//Resultado: Endereco(cep: “01418-100”, logradouro: “Alameda
Santos”)
```

Os métodos encode e decode são **thows** e podem ter seus possíveis erros captados e tratados com a estrutura de tratamento `do{} catch{}`.

Capítulo 8

Trabalhando com contatos

Seção 8-1

Introdução

Para trabalharmos com os contatos dentro dos nossos aplicativos, temos a disposição os recursos dos frameworks **Contacts** e **ContactsUI.**

Os recursos relacionados ao framework Contacts já estão incluídos em ContactsUI. Para utilizar os recursos relacionados a contatos devemos importar ContactsUI para os arquivos das respectivas classes que irão trabalhar com os contatos:

```
import ContactsUI
```

As estruturas **Contacts** e **ContactsUI** fornecem a Swift acesso as informações gravadas nos contatos do usuário. Como a maioria dos aplicativos lê informações de contato sem fazer nenhuma alteração, essas estruturas podem otimizar o uso seguro em thread-safe, fazendo com que as informações de contado sejam acessadas sem que aconteçam alterações.

Por outro lado, caso seja necessário, podemos dar acesso total a implementação de novos contatos ou até mesmo a possibilidade de apagar algum item.

Elementos como Labels, TextsFields e botões podem interagir com a lista de contatos, resgatando suas informações ou implementando novos itens a lista.

As próximas seções tratarão da captação de itens dos contatos e a manipulação desses itens.

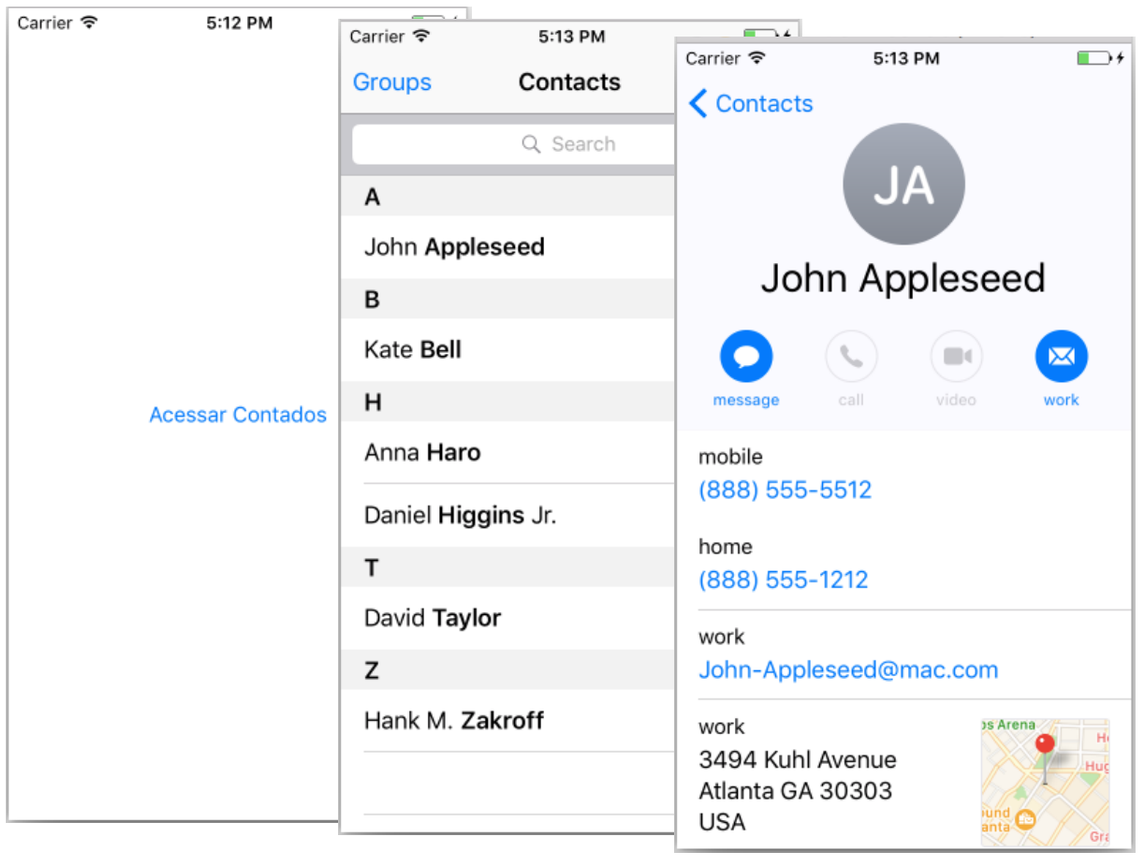
#Dica!

É importante salientar que o acesso do usuário para alterações dos itens de contatos só será permitido a partir do momento que a autorização seja solicitada ao arquivo **info.plist.**

Acessando os contatos

Vamos utilizar a estrutura **ContactsUI** para acessar os dados dos itens da lista de contatos do dispositivo., Vamos ao exemplo a seguir:

No storyboard, vamos colocar um objeto **button** e a partir dele vamos acessar os contatos.



Agora vamos ao arquivo ViewControll.swift, e fazer o lançamento dos códigos:

```
import UIKit

//Importando o framework
import ContactsUI

class ViewController : UIViewController {

    //Criar a action a partir do botão
    @IBAction func selecionarCont(_ sender: UIButtom) {

        let contactPicker = CNContactPickerViewController()

         contactPicker.delegate = self

            self.present(contactPicker, animated: true,
            completion: nil)
        }
}
```

A classe **CNContactPickerViewController** cria um objeto ViewController que gerencia a exibição do seletor de contatos.

Essa classe permite que o usuário selecione um ou mais contatos (ou suas propriedades) na lista de contatos exibidos. A partir daí, o aplicativo terá acesso apenas à seleção final do usuário.

Podemos captar as interações com a view de contatos através de um protocolo delegate. O protocolo **CNContactPickerDelegate** deve ser adotado pela classe **View Controller** ou pela classe que será exibirá a view controller de contatos, pois será através dele que a escolha e a importação dos dados serão feitas posteriormente.

```
extension  ClasseViewController : CNContactPickerDelegate {

      // Declaração dos métodos
}
```

Principais métodos de CNContactPickerDelegate

```
//Método disparado quando o botão Cancel é pressionado na
busca de um contato:
func contactPickerDidCancel(_ picker: CNContactPicker-
ViewController)
```

```
//Método disparado quando um contato é selecionado no picker:
func contactPicker(_ picker: CNContactPickerViewController,
didSelect contact: CNContact)
```

```
//Método disparado quando um item de um contato é selecionado:
func contactPicker(_ picker: CNContactPickerViewController,
didSelect contactProperty: CNContactProperty)
```

```
//Método disparado quando múltiplos contatos são selecionados:
func contactPicker(_ picker: CNContactPickerViewController,
didSelect contacts: [CNContact])
```

```
//Método disparado quando o picker de contatos é fechado:
func contactPickerDidClose(_ picker: CNContactPicker)
```

Seção 8-3

Classes de Contacts

A estrutura **Contacts** é utilizada para fazer alterações nos itens dos contatos, como criar, apagar objetos. Dentro da estrutura temos a disposição classes que nos auxiliam nas tarefas relacionadas às alterações de contatos.

Classes CNContact e CNMutableContact

A classe **CNContact** é um objeto de valor imutável que controla as propriedades de contato, como nome, imagem ou números de telefone.

Também temos a classe **CNMutableContact**, que assim como CNContact controla propriedades de um contato, mas de forma mutável.

Como principais propriedades para essas classes temos:

Propriedade	Tipo	Descrição
birthday	DateComponents?	Acesso ao item **Data de Aniversário** do contato
givenName	String	Acesso ao item **Nome** do contato
familyName	String	Acesso ao item **Sobrenome** do contato
phoneNumbers	[CNLabeledValue <CNPhoneNumber>]	Acesso aos itens de **Telefone** do contato
emailAddresses	[CNLabeledValue <NSString>]	Acesso ao item **E-mail** do contato

Classe CNSaveRequest

A classe **CNSaveRequest** cria uma nova solicitação de salvamento para cada operação relacionada a itens de contato.

Como principais métodos para essa classe temos:

```
//Método que adiciona o contato especificado para o
armazenamento na lista de contatos:
func add(_ contact: CNMutableContact,  toContainerWith-
Identifier identifier: String?)
```

```
//Método que atualiza um contato existente na lista de
contatos:
func update(_ contact: CNMutableContact)
```

```
//Método que apaga um contato da lista:
func delete(_ contact: CNMutableContact)
```

Classe CNContactStore

A classe **CNContactStore** é responsável buscar e salvar contatos, grupos e contêineres.

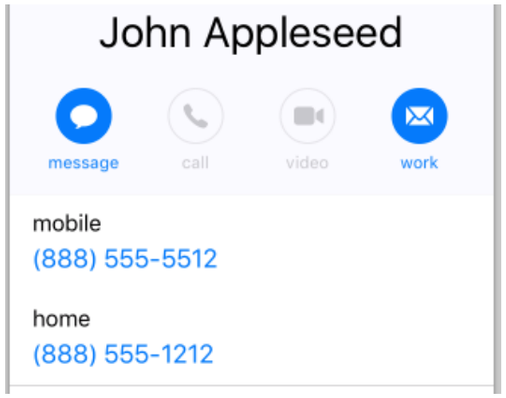
Como principal método para essa classe temos:

```
func execute(_ saveRequest: CNSaveRequest) throws
//Método que executa uma solicitação de salvamento e retorna
sucesso se obteve sucesso
```

Classe CNLabeledValue

E por ultimo temos a classe **CNLabeledValue**, que define um objeto de valor imutável. Esse objeto será associado a uma propriedade de contato levando em consideração uma determinada **label.**

Como exemplo, podemos citar o item telefone, que disponibiliza diferentes campos para implementação de números **home** ou **mobile.**



Capítulo 9

Trabalhando com mapas e localização

Seção 9-1

Introdução

Neste capítulo vamos aprender como utilizar os frameworks **MapKit** e **CoreLocation** para criar mapas, anotações e receber informações como a posição e direção do dispositivo com relação a coordenadas globais, ou através do sistema de **GPS.**

O framework **CoreLocation** é responsável por determinar a latitude e a longitude atuais de um dispositivo. Com ele podemos configurar e agendar a entrega de eventos relacionados à localização. Também podemos utilizar o hardware do dispositivo para nos fornecer sua localização utilizando o sistema de geolocalização (GPS).

Já o framework **Mapkit** permite que nossa aplicação exiba mapas para o usuário, possibilitando a criação de pontos customizados e afins.

Quando importamos o framework MapKit, todos os recursos relacionadoa a CoreLocation são importados automaticamente.

Antes de utilizarmos estas informações, precisamos verificar se o dispositivo possuí os elementos de **hardware necessários,** mesmo sabendo que na atualidade, quase todos os dispositivos possuem estes recursos, fazer uma **verificação** é considerada uma boa prática.

#Dica!

É importante salientar que o acesso do usuário ao ponto de localização do dispositivo só será permitido a partir do momento que a autorização seja solicitada ao arquivo **info.plist.**

Seção 9-2

Classe CLLocationManager

Herança: NSObject > CLLocationManager

A classe **CLLocationManager** é responsável por estabelecer parâmetros para receber dados de localização e definir o inicio ou termino do envio das informações de localização do dispositivo.

Para utilizar as informações de localização, é preciso saber se o dispositivo possui essa capacidade e pedir autorização ao usuário para acessarmos suas informações de localização. Atualmente a maioria dos dispositivos possuem tecnologia de localização.

Para a utilização dos recursos relacionados a localização, devemos fazer a importação do framework **CoreLocation** logo no início do arquivo **.swift** que utilizará as classes e protocolos:

```
import CoreLocation
```

A classe **CLLocationManager** faz uso do protocolo **delegate,** que define os métodos usados para receber as informações de localização de um objeto CLLocationManager. É importante adotar o protocolo junto a classe da View Controller que conterá itens de localização.

```
extension ClasseViewController : CLLocationManagerDelegate{

      //Declaração dos métodos
}
```

Principais Propriedades

Propriedade	Tipo	Descrição
delegate	CLLocationManager-Delegate?	Retorna o objeto delegate
desiredAccuracy	CLLocationAccuracy	Define/Retorna a precisão do localizador
location	CLLocation?	Retorna a ultima posição registrada pelo localizador
heading	CLHeading?	Retorna a última direção registrada pelo localizador
headingOrientation	CLDeviceOrientation	Define/Retorna a direção do device em uma navegação.
allowsBackground-LocationUpdates	Bool	Define/Retorna se o aplicativo receberá atualizações de localização se estiver em backgorund

Principais métodos

```
//Método que solicita a permissão para o aplicativo utilizar
as informações de localização em primeiro plano:
func requestWhenInUseAuthorization()


// Método que solicita para o usuário a permissão para que
as informações de localização fiquem sempre habilitadas:
func requestAlwaysAuthorization()


// Método que retorna o estatus de autorização relacionado
a localização:
class func authorizationStatus() -> CLAuthorizationStatus


//Método que retorna se o dispositivo possuí capacidade de
enviar informações de localização:
class func locationServicesEnabled() -> Bool


//Método que inicia a atualização da informação sobre
localização da localização do usuário:
func startUpdatingLocation()


//Método que encerra a atualização da informação sobre a
localização do usuário:
func stopUpdatingLocation()


//Método que inicia a atualização da informação da direção
do usuário:
func startUpdatingHeading()


//Método que inicia a atualização da informação da direção
do usuário:
func stopUpdatingHeading()
```

Principais métodos de CLLocationManagerDelegate

```
//Método que comunica ao delegate quando a nova informação
de localização está disponível:
optional func locationManager(_ manager: CLLocationManager,
didUpdateLocations locations: [CLLocation])


//Método que comunica ao delegate que o location manager
não está habilitado para receber informações de localização:
optional func locationManager(_ manager: CLLocationManager,
didFailWithError error: Error)


//Método que comunica ao delegado quando uma nova informação
de direção está disponível:
optional func locationManager(_ manager: CLLocationManager,
didUpdateHeading newHeading: CLHeading)
```

Estrutura auxiliar para criação de coordenadas

Algumas estruturas auxiliares são necessárias para enquadrar determinada localização. Dentre elas temos a disposição a estrutura `CLLocationCoordinate2D.` Essa struc dispõe de um método para criarmos coordenadas baseadas em latitude e longitude:

```
func CLLocationCoordinate2DMake(_ latitude:
CLLocationDegrees, _ longitude: CLLocationDegrees)
-> CLLocationCoordinate2D
```

Seção 9-3

Trabalhando com mapas

Herança: NSObject > UIResponder > UIView > MKMapView

O framework **Mapkit** fornece uma interface para anexar mapas diretamente em views. O framework também suporta anotações no mapa, adição de camadas e a marcação para uma determinada coordenada através de pinos de localização.

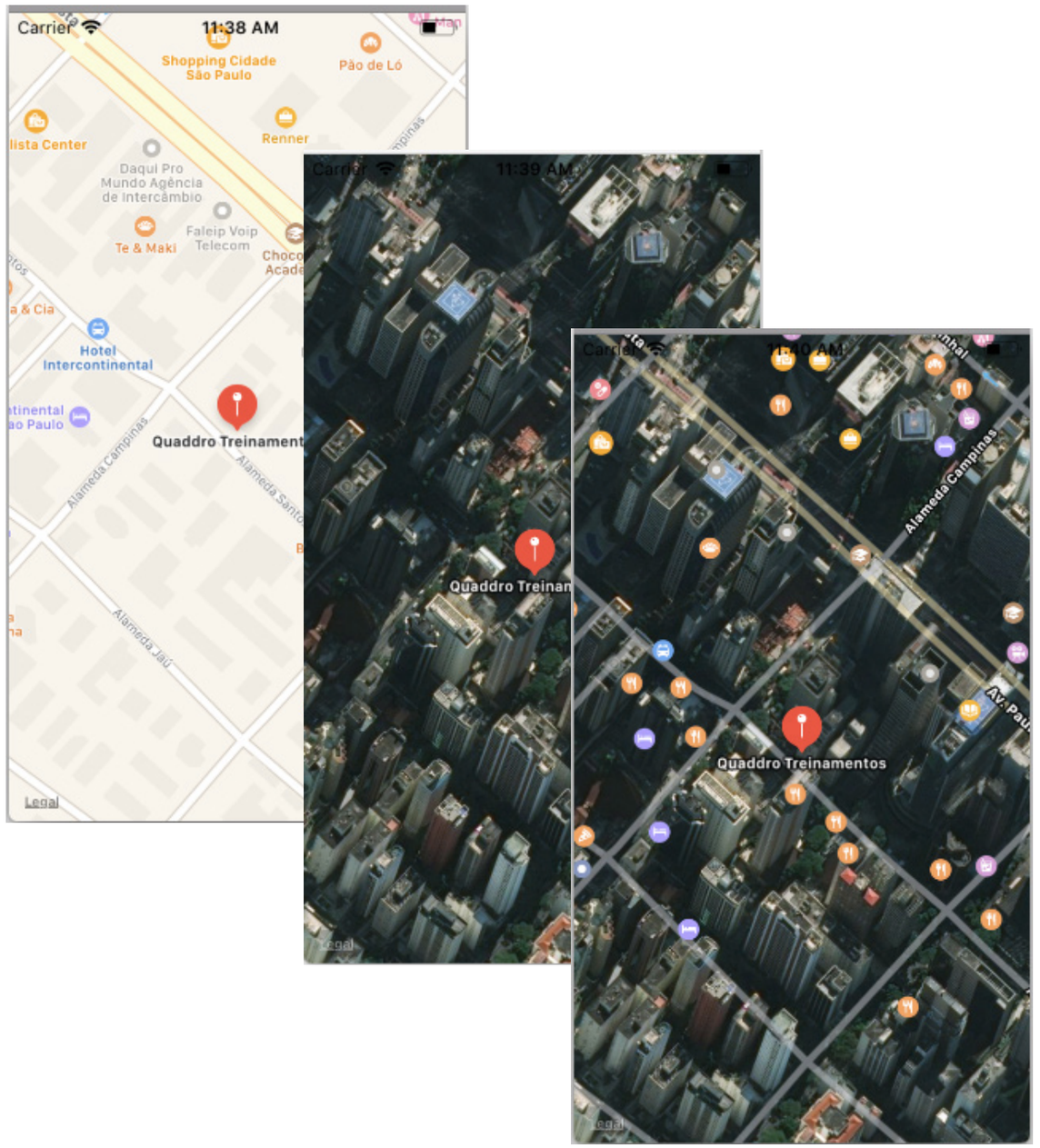
Podemos utilizar a classe **MKMapView** para exibir informações e manipular conteúdos do mapa na nossa aplicação. Podemos centralizar o mapa em uma determinada coordenada, especificar o tamanho de uma área que deseja exibir e anotar o mapa com informações customizadas.

Outro ponto importante a ressaltar é a integração que o MapKit tem com o **CoreLocation,** onde temos a opção de seguir a localização do nosso dispositivo. Quando importamos o framework MapKit, todas as opções do CoreLocation também são automaticamente importadas.

Todos os arquivos de classe que trabalharão com mapas deverão ter o framework importado.

```
import MapKit
```

Visualmente temos a disposição três formas diferentes de apresentação do mapa: **Padrão mapa, satélite e híbrido:**



Principais propriedades

Propriedade	Tipo	Descrição
delegate	MKMapView-Delegate?	Retorna o objeto delegate
mapType	MKMapType	Define/Retorna o tipo de mapa a ser exibido. Standard, Satellite e Hybrid
isZoomEnabled	Bool	Define/Retorna se o zoom está habilitado
isScrollEnabled	Bool	Define/Retorna se o scroll está habilitado
isRotateEnabled	Bool	Define/Retorna se o rotação está habilitada
showsPointsOfInterest	Bool	Define/Retorna se o mapa exibirá pontos de interesse
showsUserLocation	Bool	Define/Retorna se o mapa deve tentar exibir a posição do usuário
centerCoordinate	CLLocation-Coordinate2D	Define/Retorna a coordenada central a ser exibida no mapa

Principais métodos

```
//Método que define uma quantidade de zoom no mapa:
func MKCoordinateSpanMake(_ latitudeDelta:
CLLocationDegrees, _ longitudeDelta: CLLocationDegrees)
-> MKCoordinateSpan


//Método que define uma região baseada em coordenadas e zoom
func MKCoordinateRegionMake(_ centerCoordinate:
CLLocationCoordinate2D, _ span: MKCoordinateSpan)
-> MKCoordinateRegion

//Método que implementa uma região uma região definida para
o mapa com a opção de animar:
func setRegion(_ region: MKCoordinateRegion, animated: Bool)


//Método que define a posição central do mapa com a opção de
animar:
func setCenter(_ coordinate: CLLocationCoordinate2D,
animated: Bool)


//Método que define a forma como o usuário fará o rastreamento
da localização:
func setUserTrackingMode(_ mode: MKUserTrackingMode,
animated: Bool)


//Método que adiciona uma anotação definida ao mapa:
func addAnnotation(_ annotation: MKAnnotation)


//Método que remove uma anotação especificada no mapa:
func removeAnnotation(_ annotation: MKAnnotation)
```

Seção 9-4

Adicionando pinos ao mapa

Herança: NSObject > MKShape > MKPointAnnotation

Caso necessário, podemos indicar pontos no nosso mapa através de pinos de indicação. O **MapKit** possui uma classe responsável pela criação e manipulação desses objetos

Podemos ainda, anexar a esse ponto uma anotação, que será exibida assim que que pino for tocado, veja o método **addAnnotation** da classe **MKMapView.**

A classe **MKPointAnnotation** define um objeto de anotação localizado em um ponto específico definido para ele.

Existe apenas uma única propriedade nessa classe, que define a localização do pino através de uma coordenada proveniente de um objeto de **CoreLocation:**

Propriedade	Tipo	Descrição
coordinate	CLLocation-Coordinate2D	Define/Retorna a posição do pino em coordenadas

Para os textos relacionados a anotação, podemos utilizar duas propriedades de **MKShape:**

Propriedade	Tipo	Descrição
title	String?	Define/Retorna o título do quadro de anotação
subtitle	String?	Define/Retorna o subtítulo ou texto auxiliar do quadro de anotação

Capítulo 10

Core Motion

Seção 10-1
Introdução

Com o framework **Core Motion** podemos fazer uso dos sensores de movimento do nosso dispositivo, o acelerômetro e giroscópio.

Podemos utilizar essas informações para criar jogos, ou mesmo aproveitar a captação de movimentos em suas aplicações. Com isso, temos uma alternativa de interação sem que seja necessário tocar na tela.

Além da utilização dos sensores de movimento, temos a disposição diferentes classes para os sensores como **pedômetro e altímetro.**

É necessário, porém, antes de utilizarmos estes recursos, verificarmos se eles estão disponíveis no dispositivo para que possam ser utilizados. Fazemos isso durante a execução da aplicação.

Acelerômetro

O **acelerômetro** é o sensor utilizado para resgatarmos informações da intensidade de um movimento de um corpo em determinada direção, quando um movimento é realizado. No nosso caso, estamos nos referindo a movimentação do dispositivo em si, quando movimentamos o dispositivo para cima, baixo, esquerda ou direita.

Giroscópio

O **giroscópio** permite que um dispositivo avalie e determine a direção dos movimentos de rotação do dispositivo.

Um sensor giroscópico geralmente executa funções de reconhecimento de gestos. Além disso, os giroscópios em smartphones ajudam a determinar a posição e orientação do aparelho.

O giroscópio capta os movimentos de rotação dos dispositivos.

Para utilizarmos os recursos de movimentação devemos importar o framework Core Motion no arquivo de classe que irá dispor interação com movimentos:

```
import CoreMotion
```

#Dica!

Tanto o acelerômetro quanto o giroscópio utilizam o mesmo framework (CoreMotion).

Verificando o hardware

Herança: NSObject > CMMotionManager

Antes de utilizarmos os dados dos sensores, precisamos verificar se eles estão disponíveis e ativos. A classe **CMMotionManager** possui propriedades que nos ajudam a verificar estas informações.

Através desta classe, podemos verificar se o acelerômetro e o giroscópio, existem ou estão ativos.

Propriedade	Tipo	Descrição
isAccelerometer-Available	Bool	Retorna se o acelerômetro está disponível
isAccelerometer-Active	Bool	Retorna se o acelerômetro está atualizando dados
isGyroAvailable	Bool	Retorna se o giroscópio está disponível
isGyroActive	Bool	Retorna se o giroscópio está recebendo dados

Acelerômetro

A classe **CMMotionManager,** além de nos fornecer a propriedades para verificarmos a viabilidade em utilizar os sensores de **acelerômetro** e giroscópio, também é a responsável pelas propriedades e métodos que retornarão as informações captadas pelos sensores.

A seguir abordaremos as propriedades e métodos responsáveis por captar as informações relativas ao acelerômetro.

Principais propriedades

Propriedade	Tipo	Descrição
accelerometer-UpdateInterval	TimeInterval	Define/Retorna o intervalo de atualização utilizado pelo acelerômetro
accelerometerData	CMAccelerometer-Data?	Retorna as últimas informações captadas pelo acelerômetro.

Principais métodos

```
//Método que inicia a atualização do acelerômetro em uma
thread específica, e utiliza um bloco para retornar as
informações do sensor:
func startAccelerometerUpdates(to queue: OperationQueue,
withHandler handler: @escaping CMAccelerometerHandler)
```

```
//Método que inicia as atualizações do acelerômetro sem
um bloco:
func startAccelerometerUpdates()
```

```
//Método que encerra as atualizações do acelerômetro:
func stopAccelerometerUpdates()
```

Giroscópio

Assim como o acelerômetro, o **giroscópio** faz uso da classe **NSMotionManager** para informar ao objeto quais informações estão sendo recebidas pelo sensor, assim como definir informações como taxa de atualização.

Principais propriedades

Propriedade	Tipo	Descrição
gyro-UpdateInterval	TimeInterval	Define/Retorna o intervalo de atualização utilizado pelo giroscópio
gyroData	CMGyroData?	Retorna as últimas informações captadas pelo giroscópio

Principais métodos

```
//Método que inicia a atualização do giroscópio em uma
thread específica, e utiliza um bloco para retornar as
informações do sensor:
func startGyroUpdates(to queue: OperationQueue,
withHandler handler: @escaping CMGyroHandler)
```

```
//Método que inicia as atualizações do giroscópio sem um
bloco:
func startGyroUpdates()
```

```
//Método que encerra as atualizações do giroscópio:
func stopGyroUpdates()
```

Seção 10-5
Trabalhando com altímetro

Heranças: NSObject > CMAltimeter
NSObject > CMLogItem > CMAltitudeData

Utilizamos um objeto da classe **CMAltimeter** quando desejamos fornecer as informações de altitude para nosso aplicativo.

Como nem todos os dispositivos possuem este sensor, é importante utilizarmos o método **isRelativeAltitudeAvailable** antes de usarmos as possíveis informações.

Após checarmos se o sensor está disponível, utilizamos o método **startRelativeAltitudeUpdates** para recebermos as informações do sensor, recebendo o retorno das informações no bloco especificado.

O Core Motion retorna estas informações apenas enquanto o aplicativo está aberto, independente de ser no foreground ou background. Se o aplicativo estiver suspenso, a atualização das informações é suspensa temporariamente até que a aplicação retorne para foreground.

Além dos métodos da classe CMAltimeter, podemos utilizar as propriedades da classe **CMAltitudeData** para resgatar dos dados relacionados a altitude captada pelo dispositivo

Principais métodos de CMAltimeter

```
//Método que retorna se o device possui o altímetro:
class func isRelativeAltitudeAvailable() -> Bool
```

```
//Método que inicia as atualizações do sensor usando uma
determinada thread e retorna essas informações para o bloco:
func startRelativeAltitudeUpdates(to queue: OperationQueue,
withHandler handler: @escaping CMAltitudeHandler)
```

```
//Método que encerra a atualização do altímetro:
func stopRelativeAltitudeUpdates()
```

Propriedades de CMAltitudeData

Propriedade	Tipo	Descrição
relativeAltitude	NSNumber	Retorna o ultimo valor de altitude indicado (metros)
pressure	NSNumber	Retorna o ultimo valor de pressão atmosférica indicado (kilopascal)

Seção 10-6

Trabalhando com pedômetro

Heranças: NSObject > CMPedometer
NSObject > CMPedometerData

Utilizamos a classe **CMPedometer** quando precisamos de informações de quantos passos o usuário deu, quantos metros percorreu, ou quantos andares subiu ou desceu. Utilizamos os métodos **isStepCountingAvailable, isDistanceAvailable e isFloorCountingAvailable** para verificarmos se estas informações estão disponíveis.

Para resgatar os valores relacionados ao pedômetro contamos com a classe **CMPedometerData** e suas propriedades.

Principais métodos de CMPedometer

```
// Método que retorna se o contador de passos está disponível
no dispositivo:
class func isStepCountingAvailable() -> Bool

//Método que retorna se a estimativa de distância está
disponível no dispositivo:
class func isDistanceAvailable() -> Bool

//Método que retorna se o contador de andares está disponível
no dispositivo:
class func isFloorCountingAvailable() -> Bool

//Método que inicia o envio de informações referente ao
pedestre para o aplicativo:
func startUpdates(from start: Date, withHandler handler:
@escaping CMPedometerHandler)

//Método que encerra o envio de informações referentes ao
pedestre:
func stopUpdates()

//Método que retorna as informações coletadas entre duas
datas:
func queryPedometerData(from start: Date, to end: Date,
withHandler handler: @escaping CMPedometerHandler)
```

Propriedades de CMPedometerData

Propriedade	Tipo	Descrição
startDate	Date	Retorna a data de início da contagem
stopDade	Date	Retorna a data de início da contagem
numberOfSteps	NSNumber	Retorna a quantidade de passos contados
distance	NSNumber?	Retorna a distância percorrida (metros)
currentCadence	NSNumber?	Retorna a taxa de passos por segundo
floorsAscended	NSNumber?	Retorna a quantidade aproximada de andares em subida
floorsDescended	NSNumber?	Retorna a quantidade aproximada de andares em descida

Capítulo 11

Trabalhando com áudio e vídeo

Seção 11-1

Introdução

Para enriquecer aplicativos, podemos incluir além de textos e imagens, recursos de áudio e vídeo.

Nesse capítulo vamos aprender a reproduzir arquivos de áudio e vídeo assim como implementar recursos para a captura e manipulação de áudios.

O framework **AVFoundation** fornece recursos para a captura, manipulação e reprodução de arquivos de áudio e vídeo. Com ele podemos reproduzir arquivos de sons e vídeos definidos pelo programador e incluídos no aplicativo.

Outro framework com qual vamos trabalhar é o **AVKit,** com ele podemos criar views para reprodução de mídia, com controles completos de reprodução, navegação de capítulos e suporte para legendas.

#Dica!

Para gravar vídeo sem a necessidade de controle de formato e configuração avançadas, a recomendação é usar **UIKit** com a classe **UIImagePickerController.**

É importante que os frameworks sejam importados nos arquivos de classe servirão de base para os códigos que trabalharão com áudio e vídeo.

```
import AVKit
import AVFoundation
```

#Dica!

Os framworks de trabalho com áudio e vídeo aceitam o arquivos com as mesmas extensões suportadas pelo sistema iOS. Arquivos mais utilizados como mp3, mp4 e m4a por exemplo, são 100% compatíveis.

AVFoundation

O framework **AVFoundation** fornece recursos para a captura, manipulação e reprodução de arquivos de áudio e vídeo.

Podemos utilizar as classes de AVFoundation para implementar sons e vídeos aos nossos aplicativos. Digamos que alguns dos nossos botões irão emitir sons assim que forem tocados, por exemplo.

Os recursos de AVFoundation também podem ser utilizados na criação de jogos, players personalizados, aplicativos com recursos de acessibilidade, etc. Dentre as principais classes do framework, podemos citar:

AVAsset

Recurso para trabalhar com uma lista de faixas destinadas a serem reproduzidas ou processadas em conjunto, podendo-se trabalhar com diferentes tipos de mídias, incluindo áudios, vídeos, textos e legendas.

AVAudioMix

Um objeto **AVAudioMix** gerencia os parâmetros de entrada para mixar faixas de áudio. Ele permite que o processamento de áudio seja executado durante a reprodução ou outras operações.

AVAudioPlayer

Cria um reprodutor de áudio e fornece a reprodução automática de um arquivo de áudio do projeto ou alocado em um endereço URL.

AVAudioRecorder

Fornece a capacidade de gravação de áudio em seu aplicativo. Junto a essa classe temos propriedades e métodos para trabalhar com pausas, gravação por um determinado tempo estabelecido, etc.

AVAudioUnitEQ

Uma subclasse de **AVAudioUnitEffect** que implementa um equalizador multibanda em seu aplicativo, com controle de ganho do efeito equalizador.

AVAudioUnitReverb

Uma subclasse **AVAudioUnitEffect** que implementa um efeito de reverb (eco), ao áudio que está sendo executado.

O framework AVFoundation contém uma quantidade imensa de classes com diferentes funções relacionadas ao trabalho com áudio e vídeo dentro de um aplicação. Recomenda-se a leitura da documentação oficial para um maior entendimento dessas múltiplas classes.

https://developer.apple.com/av-foundation/

AVAudioPlayer

Herança: NSObject > AVAudioPlayer

A classe **AVAudioPlayer** fornece recursos para a reprodução de dados de áudio em seu aplicativo.

Entre os principais recursos da classe AVAudioPlayer estão:

- Tocar sons com qualquer duração;

- Criar loops de áudios;

- Tocar múltiplos sons simultaneamente.

Principais propriedades

Propriedade	Tipo	Descrição
currentTime	TimeInterval	Define/Retorna o ponto atual em segundos
data	Data?	Retorna a representação de dados
delegate	AVAudioPlayer-Delegate?	Define/Retorna o objeto delegate
duration	TimeInterval	Retorna a duração total em segundos
enableRate	Bool	Define se a reprodução pode alterar o rate
numberOfLoops	Int	Define/Retorna a quantidade de repetições
rate	Float	Define/Retorna a rate de reprodução
volume	Float	Define/Retorna o valor do volume entre 0.0 e 1.0
isPlaying	Bool	Retorna quando um áudio está sendo executado

Principais métodos

```
//Método que executa um áudio:
func play() -> Bool

//Método que executa um áudio começando em um ponto:
func play(atTime time: TimeInterval) -> Bool

//Método que pausa a reprodução de um áudio:
func pause()

//Método que para a reprodução de um áudio:
func stop()

//Método que prepara o som para execução com o pré carregamento
de buffer:
func prepareToPlay() -> Bool
```

AVAudioPlayerDelegate

A classe **AVAudioPlayer** pode fazer uso do protocolo **AVAudioPlayerDelegate.**

Todos os métodos neste protocolo são opcionais. Eles permitem que um delegate responda a pausas de áudio, erros de decodificação de áudio e à conclusão da reprodução de um som.

Caso seja necessária a sua utilização, o protocolo deve ser adotado junto a classe que cuidará da view controller:

```
extension ClasseViewController : AVAudioPlayerDelegate {

      // Declaração dos métodos
}
```

Métodos de AVAudioPlayerDelegate

```
//Método que diz ao delegate quando um som parou de tocar:
optional func audioPlayerDidFinishPlaying(_ player:
AVAudioPlayer, successfully flag: Bool)

//Método que retorna ao delegate quando acontece um erro de
decodificação:
optional func audioPlayerDecodeErrorDidOccur(_ player:
AVAudioPlayer, error: Error?)
```

AVAudioRecorder

Herança: NSObject > AVAudioRecorder

A classe **AVAudioRecorder** fornece recursos para a gravação de áudio dentro do aplicativo.

Entre os principais recursos de **AVAudioRecorder** estão: gravar sem limite de tempo, gravar com limite de tempo, pausar e continuar uma gravação.

Para gravar um áudio é necessário criar um dicionário com no mínimo as chaves: **AVEnconderAudioQualityKey, AVEncoderBitRateKey, AVNumberOfChannelsKey e AVSampleRateKey,** que controlam respectivamente a qualidade do áudio, a taxa de bit rate, a quantidade de canais e a quantidade de hertz do áudio a ser gravado.

Principais propriedades

Propriedade	Tipo	Descrição
currentTime	TimeInterval	Retorna o ponto atual da gravação em segundos
delegate	AVAudioRecorder-Delegate?	Define/Retorna o objeto delegate
settings	Dictionary	Retorna as configurações de gravação
url	URL	Retorna o endereço do arquivo associado à gravação do áudio
isRecording	Bool	Retorna quando um áudio está sendo gravado

Principais métodos

```
//Método que inicia a gravação do áudio:
func record() -> Bool


//Método que pausa a gravação:
func pause()


//Método que encerra a gravação:
func stop()


//Método que inicia a gravação com uma duração especificada:
func record(forDuration duration: TimeInterval) -> Bool


//Método que apaga a última gravação efetuada dentro do
mesmo tempo de execução:
func deleteRecording() -> Bool


//Método que cria o arquivo de áudio e prepara o sistema
para a gravação:
func prepareToRecord() -> Bool
```

AVAudioRecorderDelegate

A classe **AVAudioRecorder** pode fazer uso do protocolo **AVAudioRecorderDelegate.**

Todos os métodos neste protocolo são opcionais. Eles permitem que um delegate responda a pausas de áudio, erros de decodificação de áudio e à conclusão da gravação de um som.

Caso seja necessária a sua utilização, o protocolo deve ser adotado junto a classe que cuidará da view controller:

```
extension ClasseViewController : AVAudioRocorderDelegate {

      // Declaração dos métodos
}
```

Métodos de AVAudioRecorderDelegate

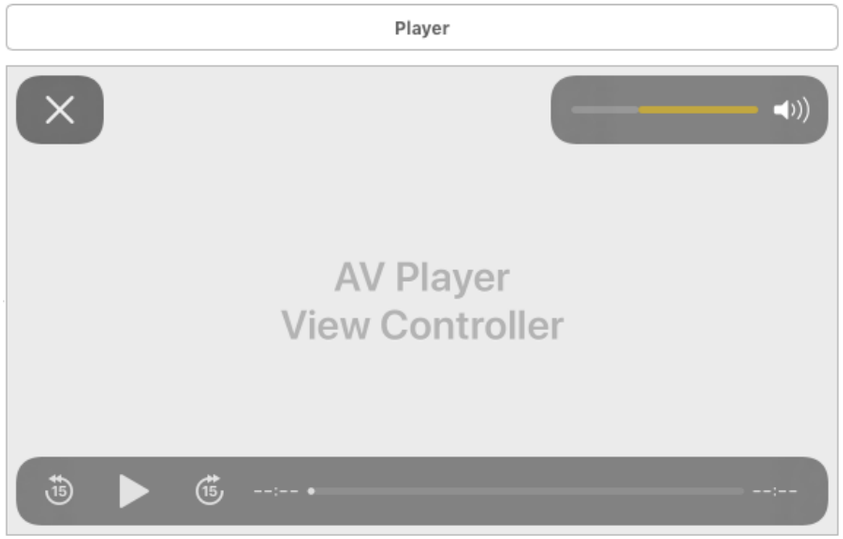
```
//Método que retorna ao delegate quando uma gravação é
encerrada:
optional func audioRecorderDidFinishRecording(_ recorder:
AVAudioRecorder, successfully flag: Bool)

//Método que retorna ao delegate quando ocorre um erro na
codificação da gravação:
optional func audioRecorderEncodeErrorDidOccur(_ recorder:
AVAudioRecorder, error: Error?)
```

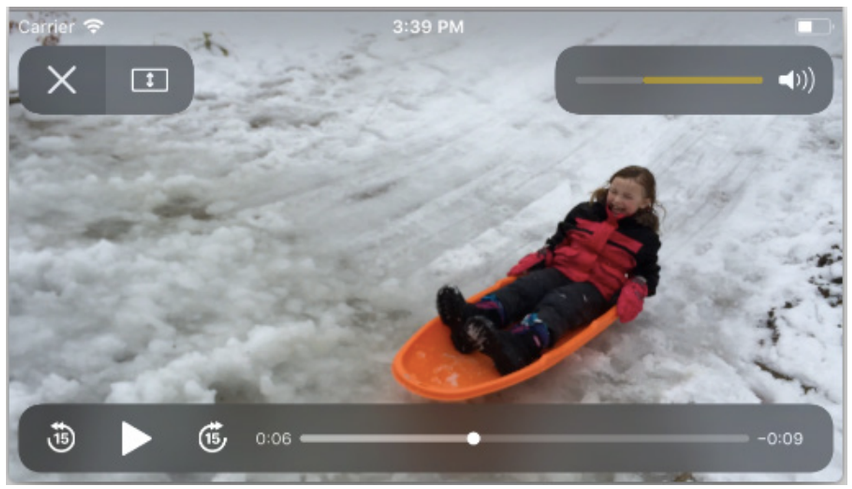
#Dica!

É importe que uma estrutura de **Sandbox** seja desenvolvida para **persistir os arquivos de áudio gravados.** Dessa forma podemos resgatar e reproduzir esses arquivos conforme as necessidade do aplicativo.

AVKit

O framework **AVKit** fornece uma interface de alto nível para reproduzir conteúdos de áudio e vídeo, com ele podemos criar views e view controllers para reprodução de mídia, com controles completos de reprodução, navegação de capítulos e suporte para legendas.





O trabalho com o AVKit é simples e direto, pois exige poucas linhas de código e ao mesmo tempo obtemos uma interface completa para execução de arquivos de mídia.

Outro ponto importante a destacar, é que além de trabalhar com os dispositivos iPhone e iPad, também podemos utilizar o AVKit para dispositivos que trabalhem com o tvOS.

Dentre as principais classes de AVKit, podemos citar:

AVCaptureView

Subclasse de **NSView** que pode ser usada para exibir controles de interface de usuário padrão para capturar dados de mídia.

AVPlayerView

Subclasse de **NSView** usada para exibir o conteúdo visual de um objeto **AVPlayer** e apresentar controles padrão para gerenciar sua reprodução.

AVPlayerViewController

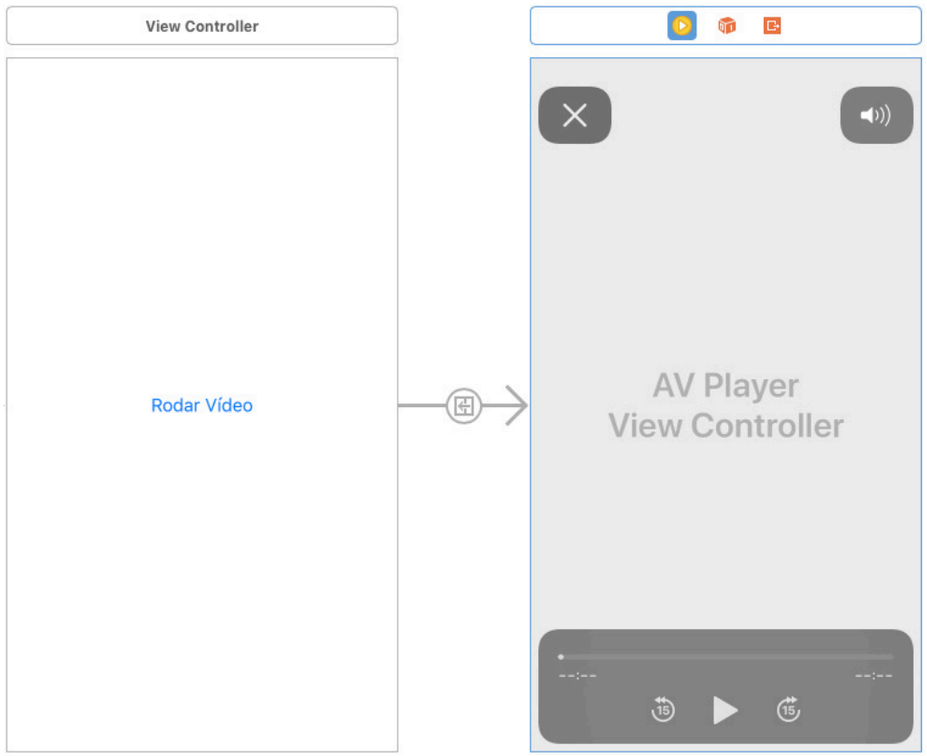
Exibe o conteúdo de mídia em um objeto AVPlayer juntamente com os controles de reprodução fornecidos pelo sistema em uma nova view controller.

AVPlayerViewController

Herança: NSObject > UIResponder > UIViewController > AVPlayerViewController

O uso do **AVPlayerViewController** facilita a adição de recursos de reprodução de mídia, combinando o estilo e os recursos dos players dos nativos do iOS com a sua aplicação.

Um dos recursos mais interessantes da classe é o suporte ao AirPlay com outros dispositivos Apple.



Principais propriedades

Propriedade	Tipo	Descrição
player	AVPlayer?	Define o conteúdo de mídia para o controller do player.
showsPlayback-Controls	Bool	Define/Retorna se os controles de playback estarão disponíveis
videoBounds	CGRect	Retorna o tamanho e a posição atuais da imagem de vídeo

#Dica!

Como o **AVPlayerViewController** é uma classe de framework do sistema, seus aplicativos de reprodução adotam automaticamente a estética e recursos de futuras atualizações do sistema operacional sem qualquer trabalho adicional.

Capítulo 12

Compartilhamento de dados

Seção 12-1

UIActivityViewController

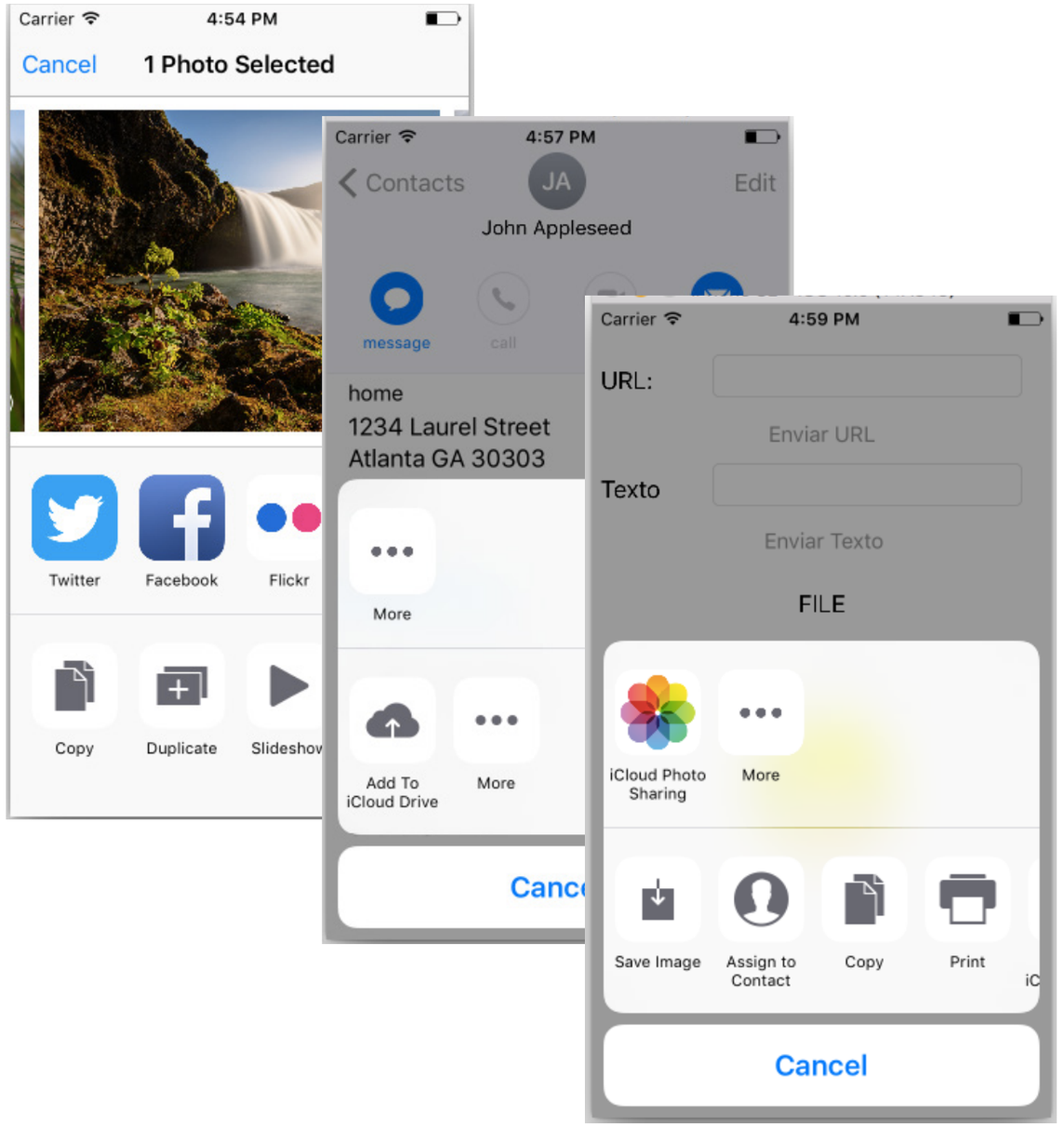
Herança: NSObject > UIResponder > UIViewController > UIActivityViewController

A tecnologia **Airdrop** dos dispositivos iOS foi uma grande revolução implantada pela Apple. Com ele podemos fazer o compartilhamento de informações entre dispositivos, sejam imagens, textos, urls, entre outros tipos de informações.

Utilizamos a classe **UIActivityViewController**, para realizar o compartilhamento de dados via Airdrop, ou a partir de outros meios de compartilhamento, dependendo dos recursos instalados no dispositivo.

Podemos compartilhar dados com **Facebook, WhatsApp** e **Twitter**. Cada meio de compartilhamento tem suas limitações e cabe ao desenvolvedor apontar quais meios serão utilizados e quais serão excluídos para compartilhar a informação.

Para a utilizar a classe, basta criar um objeto e usar o inicializador que recebe um **Array** contendo as informações/arquivos que serão enviados. Após isso, basta exibir o view controller.



#Dica!

Os recursos disponíveis para compartilhamento variam de acordo com os aplicativos instalados em cada dispositivo.

O simulador não fornece acesso ao compartilhamento por AirDrop. É interessante testar o recurso com um dispositivo plugado ao seu computador.

Instanciando a classe UIActivityViewController

Podemos criar um objeto do tipo **UIActivityViewController,** e aproveitar sua instância para implementar os elementos a serem compartilhados. Vamos a um exemplo:

```
var minhaActivity : UIActivityViewController!

func enviarArquivo() {

   let urlImagem = Bundle.main.url(forResource: “imagem”,
   withExtension: “png”)

   minhaActivity = UIActivityViewController(activityItems:
   [urlImagem!], applicationActivities: nil)

   self.present(minhaActivity, animated: true,
   completion: nil)
}
```

Principais propriedades

Propriedade	Tipo	Descrição
excluded-ActivityTypes	[UIActivityType]?	Define/Retorna a lista de serviços que não estarão disponíveis

Principais métodos

```
//Método que inicia o view controller recebendo as informações
e arquivos e quais meios serão utilizados para compartilhar:
init(activityItems: [Any],  applicationActivities:
[UIActivity]?)
```

Capítulo 13

Mensagens de e-mail e SMS

Seção 13-1

Introdução

Seja por uma necessidade de compartilhamento social ou mesmo um recurso de complemento de dados, diversos aplicativos fazem disparos de mensagens para outros usuários.

Esses envios podem ser por **e-mail,** ou **SMS** e nesse capítulo vamos trabalhar com essas formas de envio de informações relacionadas a mensagens.

O framework **MessageUI** fornece recursos para a tela de exibição e envio de mensagens de texto via e-mail ou SMS.

Uma vantagem do uso do MessageUI, é a possibilidade de fazer esses disparos de emails e SMS sem que o usuário precise sair do aplicativo.

Quando o ViewController de redação e envio de mensagens é exibido, ele é carregado por cima do conteúdo do aplicativo, desse modo quando a janela é removida, ela simplesmente sai de cima do antigo conteúdo.

Esse recurso garante uma melhor experiência para o usuário, que não precisa sair de um aplicativo e entrar em outro, e também para o programador, que tem mais controle do seus sistemas durante o envio de mensagens.

As principais classes de trabalho para o envio de e-mails e SMS são respectivamente: **MFMailComposeViewController** e **MFMessageComposeViewController.**

#Dica!

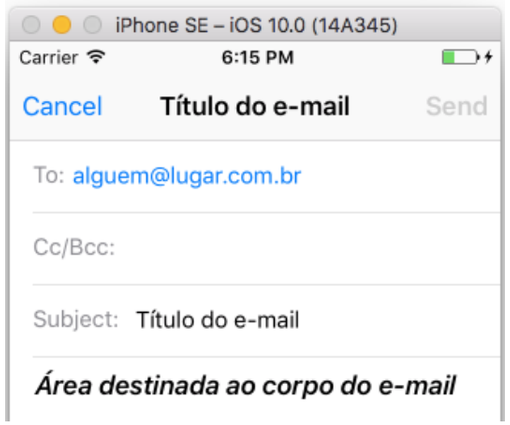
O Simulador do XCode tem certas limitações para apresentar as interfaces relacionadas a SMS e e-mail. Para uma boa simulação é recomendável a utilização de um dispositivo conectado ao seu computador.

Seção 13-2
MFMailComposeViewController

Herança: NSObject > UIResponder > UIViewController > UINavigationController > MFMailComposeViewController

A classe **MFMailComposeViewController** fornece uma interface gráfica que gerencia automaticamente a redação, edição e envio de mensagens por e-mail.

O ViewController possui métodos para preenchimento de campos para destinatários principais, destinatários com cópias e anexos, corpo da mensagem, bem como botões de envio e cancelar.



Principais propriedades

Propriedade	Tipo	Descrição
mailCompose-Delegate	MFMailCompose-ViewController Delegate?	Define o objeto delegate

Principais métodos

```
//Método que Retorna um bool indicando se o dispositivo
atual é capaz de enviar e-mail:
class func canSendMail() -> Bool


//Método que define uma array de destinatários para o
determinado campo:
func setToRecipients(_ toRecipients: [String]?)


//Método que define uma array de destinatários para o
cópias do e-mail:
func setCcRecipients(_ ccRecipients: [String]?)


//Método que define o campo assunto do email:
func setSubject(_ subject: String)


//Método que define o texto do corpo da mensagem, podendo
aceitar tags em HTML se definido:
func setMessageBody(_ body: String, isHTML: Bool)


//Método que adiciona uma determinada informação como
anexo à mensagem:
func addAttachmentData(_ attachment: Data, mimeType: String,
fileName filename: String)
```

MFMailComposeViewControllerDelegate

A classe **MFMailComposeViewController** pode fazer uso do protocolo **MFMailComposeViewControllerDelegate.**

Caso seja necessária a sua utilização, o protocolo deve ser adotado junto a classe que cuidará da view controller:

```
extension ClasseViewController :
MFMailComposeViewControllerDelegate {

      // Declaração dos métodos
}
```

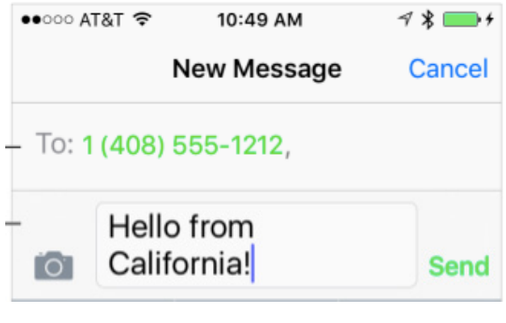
Método de MFMailComposeViewControllerDelegate

```
optional func mailComposeController(_ controller:
MFMailComposeViewController, didFinishWith result:
MFMailComposeResult, error: Error?)
//Método que notifica ao delegate que o usuário saiu da tela
de composição com o resultado definido
```

Seção 13-3
MFMessageComposeViewController

Herança: NSObject > UIResponder > UIViewController > UINavigationController > MFMessageComposeViewController

A classe **MFMessageComposeViewController** fornece uma interface gráfica que gerencia automaticamente a redação, edição e envio de mensagens por SMS. O ViewController possui propriedades para preenchimento de campos para destinatários e corpo da mensagem, bem como botões de envio e cancelar.



Principais Propriedades

Propriedade	Tipo	Descrição
messageCompose-Delegate	MFMessageCompose-ViewController-Delegate?	Define o objeto delegate
recipients	[String]?	Define/Retorna os destinatários
subject	String?	Define/Retorna o assunto da mensagem
body	String?	Define/Retorna o conteúdo da mensagem

Principais métodos

```
//Método que retorna um Booleano indicando se o dispositivo
está apto a enviar SMS:
class func canSendText() -> Bool


//Método que retorna um Booleano indicando se o dispositivo
está apto a enviar anexos por SMS:
class func canSendAttachments() -> Bool


//Método que notifica ao delegado que o usuário saiu da tela
de composição com o resultado definido:
func addAttachmentData(_ attachmentData: Data, typeIdentifier
uti: String, filename: String) -> Bool
```

MFMessageComposeViewControllerDelegate

A classe **MFMessageComposeViewController** pode fazer uso do protocolo **MFMessageComposeViewControllerDelegate.**

Caso seja necessária a sua utilização, o protocolo deve ser adotado junto a classe que cuidará da view controller:

```
extension ClasseViewController :
MFMessageComposeViewControllerDelegate {

      // Declaração dos métodos
}
```

Método de MFMessageComposeViewControllerDelegate

```
//Método que notifica ao delegate que o usuário saiu da tela
de composição com o resultado definido:
func messageComposeViewController(_ controller:
MFMessageComposeViewController, didFinishWith result:
MessageComposeResult)
```

Capítulo 14

Trabalhando com notificações

Seção 14-1

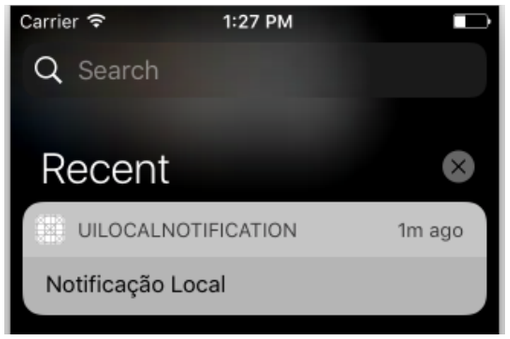
Introdução

Um dos recursos mais funcionais para avisar usuários sobre novidades ou mesmo para pedir que volte ao aplicativo são as **Notificações.**

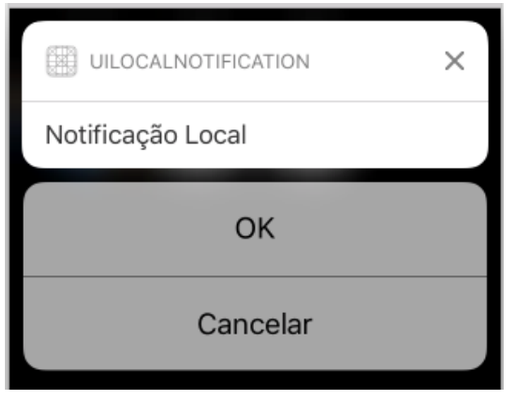
Existem deferentes tipos de notificações e nesse capítulo vamos complementar nossas aplicações com esses recursos.

Temos a disposição, desde notificações internas ao nosso aplicativo, que avisa ao centro de notificações, determinadas ações escolhidas pelo programador, passando por notificações locais, com agendamento por data, até aquelas que fazem uso de um servidor para disparos remotos.

As notificações locais e de servidores fazem uso da interface padrão para notificações adotadas pelo iOS, inclusive com o aviso sonoro característico de notificações, e podem ter sua aparência alterada conforme as diferentes versões do sistema.



Temos também a possibilidade de colocar botões nas nossas notificações que podem disparar operações diversas, bem como imagens ou outros elementos.



As notificações internas do aplicativo são controladas pela classe **NotificationCenter,** que é um objeto do framework principal **Foundation.**

Para notificações locais ou de servidores, temos a disposição o framework **UserNotifications,** que deve ser importado no arquivo que conterá os objetos que trabalharão com as notificações:

```
import UserNotifications
```

É importante salientar, que o trabalho com notificações remotas (Push Notifications), só é possível com uma conta de desenvolvedor paga. Abordaremos seu funcionamento adiante.

Seção 14-2

NotificationCenter

Herança: NSObject > NotificationCenter

Um objeto **NotificationCenter** fornece um mecanismo para transmitir informações dentro de um aplicativo. Esse objeto é essencialmente uma tabela de despacho de notificações.

Estruturalmente, cada programa possui um objeto **notification center,** dessa forma, quando for necessário seu uso em um aplicativo, devemos utilizar esse objeto, já que é automaticamente criado pelo sistema.

O uso do NotificationCenter é surpreendentemente simples e extremamente útil para situações onde você precisa para enviar mensagens de notificação acopladas ao código.

Principais métodos

```
//Método que adiciona um registro no NotificationCenter
para aguardar uma notificação:
func addObserver(_ observer: Any, selector aSelector:
Selector, name aName: NSNotification.Name?, object anObject:
Any?)


//Método que publica uma notificação:
func post(_ notification: Notification)
```

```
//Método que publica uma notificação definindo seu nome obejto
disparador e informações a serem acrescentadas:
func post(name aName: NSNotification.Name, object anObject:
Any?, userInfo aUserInfo: [AnyHashable : Any]? = nil)


//Método que remove observadores:
func removeObserver(_ observer: Any)


//Método que remove uma notificação com nome definido:
func removeObserver(_ observer: Any, name aName:
NSNotification.Name?, object anObject: Any?)
```

Seção 14-3

UserNotifications

O framework **UserNotifications** é utilizada para exibir e o tratar o lançamento de notificações locais e remotas relacionados ao aplicativo.

As classes desse framework são utilizadas para agendar a entrega de notificações locais com base em condições específicas, como tempo ou local determinado.

Também podemos utilizar esse framework para receber e potencialmente modificar notificações locais e remotas quando são entregues ao dispositivo.

Como Principais classes do framework UserNotifications, podemos citar:

UNUserNotificationCenter

Gerencia as atividades relacionadas à notificação do aplicativo ou alguma extensão do aplicativo. Adiante, temos alguns métodos importantes dessa classe:

```
//Método que faz a requisição da autorização para envios de
notificações:
func requestAuthorization(options: UNAuthorizationOptions
= [], completionHandler: @escaping (Bool, Error?) -> Void)


//Método que define as categorias de notificações a serem
utilizadas na aplicação:
func setNotificationCategories(_ categories:
Set<UNNotificationCategory>)


//Método que adiciona uma notificação ao notification center:
func add(_ request: UNNotificationRequest,
withCompletionHandler completionHandler: ((Error?) ->
Void)? = nil)


//Método que remove da fila, todas as notificações pendentes
de resposta:
func removeAllPendingNotificationRequests()


//Método que remove da fila, todas as notificações entregues:
func removeAllDeliveredNotifications()


//Método que remove da fila, notificações especificadas:
func removeDeliveredNotifications(withIdentifiers identifiers:
[String])
```

Como essa classe pode fazer uso do protocolo delegate, a propriedade **delegate** também pode ser utilizada.

UNCalendarNotificationTrigger

Aciona a entrega de uma notificação na data e hora especificadas. Podemos utilizar o método inicializador para efetuar a tarefa:

```
convenience init(dateMatching dateComponents: DateComponents,
repeats: Bool)
```

UNLocationNotificationTrigger

Aciona a entrega de uma notificação quando o usuário atinge a localização geográfica especificada;

UNMutableNotificationContent

Fornece o conteúdo editável de uma notificação. Temos a disposição propriedades para preenchimento desse conteúdo como **title, body e sound.**

UNNotification

Armazena a data em que uma determinada notificação foi entregue;

UNNotificationAction

Define uma tarefa a executar em resposta a uma notificação entregue. Com essa classe podemos colocar botões em nossa notificação. O método inicializador pode ser utilizado para a criação de uma ação:

```
convenience init(identifier: String, title: String, options:
UNNotificationActionOptions = [])
```

UNNotificationCategory

Define os tipos de notificações suportadas pelo seu aplicativo e as ações personalizadas exibidas para cada tipo, O método inicializador pode ser utilizado para a criação de uma categoria:

```
convenience init(identifier: String, actions:
[UNNotificationAction], intentIdentifiers: [String], options:
UNNotificationCategoryOptions = [])
```

UNNotificationContent

Armazena o conteúdo de uma notificação local ou remota;

UNNotificationRequest

Engloba conteúdos a notificação relacionadas as condições que acionam sua entrega;

UNNotificationResponse

Contém a resposta do usuário a uma notificação acionável;

UNNotificationSound

Habilita ou não o som a ser reproduzido quando uma notificação é entregue.

#Dica!

As classes que compõem o framework UserNotifications são numerosas e repletas de métodos e propriedades. Para um índice completo de todos os itens, recomenda-se e leitura da documentaçào oficial disponível em **https://developer.apple.com/reference/usernotifications**

Protocolo UNUserNotificationCenterDelegate

Quando interagimos com uma notificação, as suas interações podem ser captadas e retornadas ao aplicativo através do protocolo **UNUserNotificationCenterDelegate.**

Todos os métodos neste protocolo são opcionais. Eles permitem que um delegate responda a entrada ou resposta a uma notificação.

Caso seja necessária a sua utilização, o protocolo deve ser adotado junto a classe que cuidará das notificações, vamos a um exemplo:

```
extension ClasseViewController : UNUserNotificationCenter-
Delegate {

      // Declaração dos métodos
}
```

Métodos de UNUserNotificationCenterDelegate

```
//Método que é executado quando uma notificação está para
ser entregue:
optional func userNotificationCenter(_ center:
UNUserNotificationCenter,  willPresent notification:
UNNotification, withCompletionHandler completionHandler:
@escaping (UNNotificationPresentationOptions) -> Void)
```

```
//Método que é executado quando uma resposta a uma ação da
notificação acontece:
optional func userNotificationCenter(_ center:
UNUserNotificationCenter, didReceive response:
UNNotificationResponse, withCompletionHandler
completionHandler: @escaping () -> Void)
```

Seção 14-4

Push Notifications

A partir do momento que temos uma conta de desenvolvedor Apple paga, temos a disposição alguns recursos. Um deles é a capacidade de implementar o recebimento de notificações remotas na nossa aplicação.

Temos que separar o trabalho com notificações remotas em dois ramos distintos. Uma ponto, é a capacidade de **Receber** notificações remostas, e o outro ponto fica por conta do **Envio** desse tipo de notificação.

O intuito dessa seção é mostrar como habilitamos o recurso de **Recebimento** de notificações remotas em nossas aplicações, deixando como dica a utilização dos serviços do provedor **One Signal** para envios, como veremos mais adiante.

Habilitando o recebimento de Push Notifications

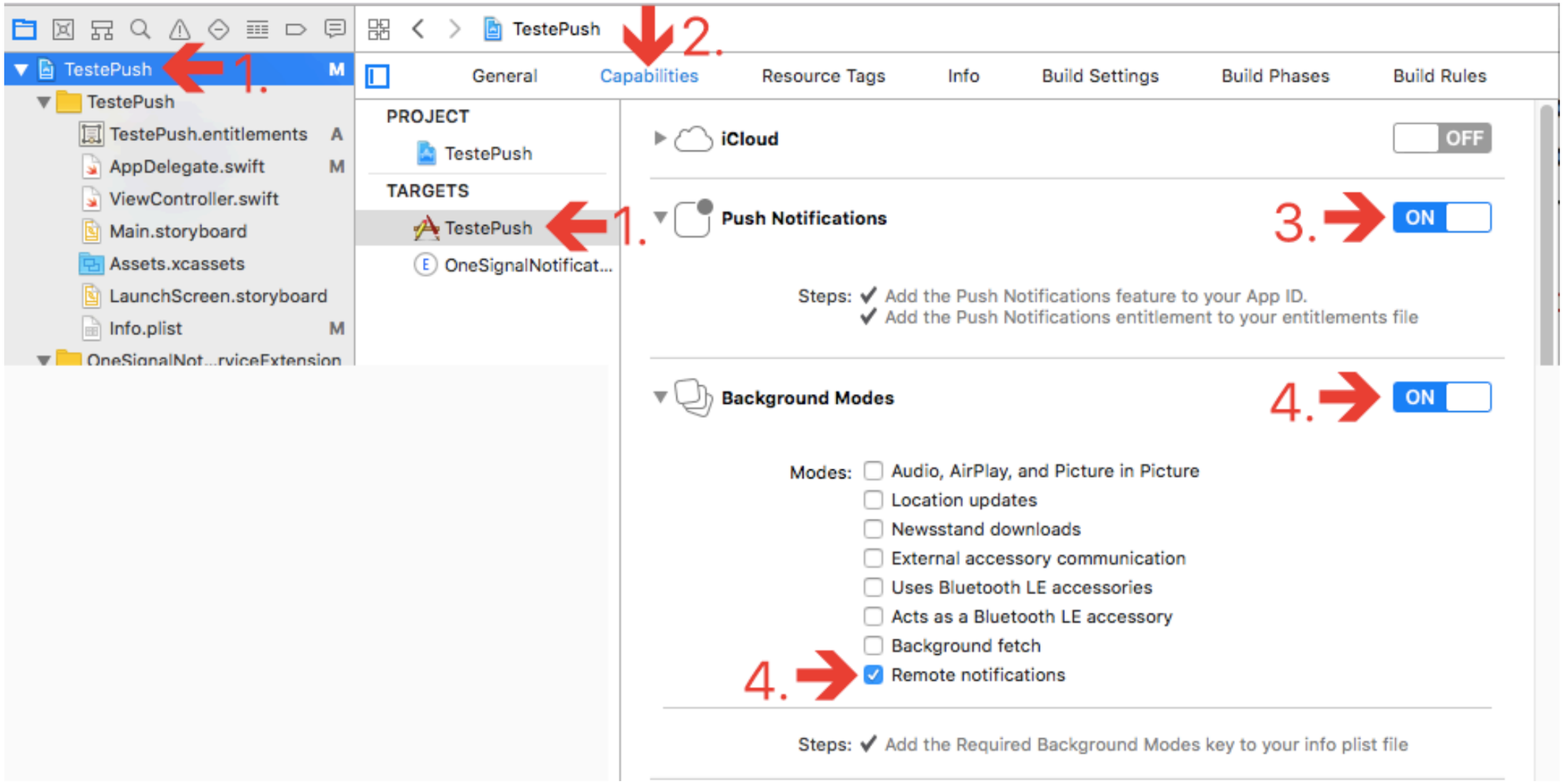
Para habilitar o recebimento de Push Notifications na aplicação, devemos proceder os seguintes passos:

1. Escolha o **pacote** do projeto, e em seguida o **Target** que cuida dos recursos de projeto;

2. Nas guias, escolha a opção **Capabilities;**

3. Dentre o grupo de opções, habilite aquela que cuida de **Push Notifications;**

4. Habilite também a opção que cuida de **Background Modes,** e nela marque a caixa **Remote notifications;**



Servidor One Signal

Um dos recursos para envio de push disponíveis é o One Signal. Recomendamos os tutoriais adiante para a realização de envios:

- Criar os certificados necessários:

https://documentation.onesignal.com/docs/generate-an-ios-push-certificate

- Fazer o Setup do SDK:

https://documentation.onesignal.com/docs/ios-sdk-setup

- Enviando notificações:

https://documentation.onesignal.com/docs/testing-mobile-push-notifications

Capítulo 15

Trocando dados com conexões Multipeer

Seção 15-1

Introdução

Podemos criar uma conexão entre dois dispositivos para trocar informações através do framework **Multipeer Connectivity,** responsável por gerenciar todos os detalhes da conexão entre os aparelhos.

O framework **Multipeer Connectivity** fornece suporte para usarmos recursos de comunicação entre dispositivos através de redes **Wi-fi, Wi-fi peer to peer e bluetooth,** para enviar dados como mensagens, streaming e arquivos diversos.

Para trabalhar com o framework Multipeer connectivity precisamos conhecer suas quatro etapas mais importantes:

1. Peer (MCPeerID class): Um peer nada mais é que um dispositivo disponível, programaticamente falando. Usamos a instância desta classe para inicializarmos as outras posteriormente;

2. Sessão (MCSession Class): Uma sessão permite a comunicação entre dispositivos pareados;

3. Browser (MCNearbyServiceBrowser class): É utilizada para rastrear dispositivos nas proximidades e convidá-los para uma sessão;

4. Advertiser (MCNearbyServiceAdvertiser): É utilizada para tornar o dispositivo visível e aceitar ou não convites de conexão.

Seguimos uma lógica simples para realizar a conexão:

Um dispositivo usa o browser para encontrar um outro dispositivo que está nas proximidades. Após encontrar, podemos enviar um convite para iniciar uma sessão, que pode ser aceita ou não.

O framework Multipeer Connectivity é composto por diversas classes que nos auxiliam a conectar dispositivos. Nesse capítulo vamos abordar as principais classes de conexão.

O framwork que deve ser importado no arquivo que conterá os objetos que trabalharão com as conexões:

```
import MultipeerConnectivity
```

Identificando o device com MCPeerID

Herança: NSObject > MCPeerID

A classe **MCPeerID** é responsável pela criação do objeto que identifica um dispositivo para a utilização do framework Multipeer Connectivity.

A identificação nominal do dispositivo não pode ter mais de **63 bytes utilizando UTF-8.**

Propriedade

Propriedade	Tipo	Descrição
displayName	String	Retorna o display name definido

Método

```
//Método inicializador que cria um objeto MCPeerID e recebe
como parâmetro o display name do peer:
init(displayName myDisplayName: String)
```

Iniciando sessões com MCSession

Herança: NSObject > MCSession

Um objeto da classe **MCSession** habilita e gerencia a comunicação de todos os peers em uma sessão Multipeer Connectivity

Para criarmos uma sessão, primeiro precisamos criar um **MCPeerID** object, que representa o peer local. Depois, precisamos adicionar os demais peers, seja através da busca ou manualmente.

Por último, aguardamos a chamado do método específico do delegate, para sabermos quando a sessão foi estabelecida.

Principais propriedades

Propriedade	Tipo	Descrição
delegate	MCSession-Delegate?	Define o delegate
myPeerID	MCPeerID	Retorna o identificador do dispositivo
connectedPeers	[MCPeerID]	Retorna um array com todos os dispositivos conectados

Principais métodos

```
//Método que cria uma sessão Multipeer Connectivity:
convenience init(peer myPeerID: MCPeerID)


//Método que cria uma sessão Multipeer Connectivity,
fornecendo informações de segurança:
init(peer myPeerID: MCPeerID, securityIdentity identity:
[Any]?, encryptionPreference: MCEncryptionPreference)
```

```
//Método que conecta um peer manualmente:
func connectPeer(_peerID:MCPeerID,withNearbyConnectionData
data: Data)


//Método que cancela a tentativa de conexão com um peer:
func cancelConnectPeer(_ peerID: MCPeerID)


//Método que envia uma mensagem encapsulada em um objeto
Data para peers próximos:
func send(_ data: Data, toPeers peerIDs: [MCPeerID], with
mode: MCSessionSendDataMode) throws
```

```
//Método que envia o conteúdo de uma URL para um peer:
func sendResource(at resourceURL: URL, withName resourceName:
String, toPeer peerID: MCPeerID, withCompletionHandler
completionHandler: ((Error?) -> Void)? = nil) -> Progress?


//Método que desconecta o dispositivo de uma sessão
func disconnect()
```

Protocolo MCSessionDelegate

Quando uma seção é iniciada, cada passo dentro da seção pode ser captado através do protocolo **MCSessionDelegate.**

Caso seja necessária a sua utilização, o protocolo deve ser adotado junto a classe que cuidará das notificações, vamos a um exemplo:

```
extension ClasseViewController : MCSessionDelegate {

      // Declaração dos métodos
}
```

Métodos do MCSessionDelegate

```
//Método que comunica ao delegate quando um objeto NSData é
recebido de um peer próximo:
func session(_ session: MCSession, didReceive data: Data,
fromPeer peerID: MCPeerID)
```

```
//Método que comunica ao delegate quando o peer local começa
a receber conteúdos de um peer próximo:
func session(_ session: MCSession, didStartReceivingResource-
WithName resourceName: String, fromPeer peerID: MCPeerID,
with progress: Progress)


//Método que comunica ao delegate quando o peer local termina
de receber conteúdos de um peer próximo:
func session(_ session: MCSession, didFinishReceiving-
ResourceWithName resourceName: String, fromPeer peerID:
MCPeerID, at localURL: URL, withError error: Error?)
```

```
//Método que comunica o delegate quando um peer próximo
abre uma conexão de transmissão de bytes com o peer local:
func session(_ session: MCSession, didReceive stream:
InputStream, withName streamName: String, fromPeer peerID:
MCPeerID)


//Método que comunica o delegado quando o status de um peer
próximo é alterado:
func session(_ session: MCSession, peer peerID: MCPeerID,
didChange state: MCSessionState)
```

MCNearbyServiceBrowser

Herança: NSObject > MCNearbyServiceBrowser

O objeto da classe **MCNearbyServiceBrowser** é responsável por procurar serviços oferecidos por dispositivos próximos através do **Wi-fi, peer to peer Wi-fi e bluetooth.**

Com ele é possível convidar estes dispositivos para uma conexão Multipeer Connectivity.

Principais Propriedades

Propriedade	Tipo	Descrição
delegate	MCNearbyService-BrowserDelegate?	Define o delegate
myPeerID	MCPeerID	Retorna o identificador do dispositivo
serviceType	String	Retorna o tipo de serviço que o objeto está procurando

Principais Métodos

```
//Método a buscar por peers:
func startBrowsingForPeers()

//Método que para a busca por peers:
func stopBrowsingForPeers()

//Método que convida um peer encontrado a se juntar a uma
conexão Multipeer:
func invitePeer(_ peerID: MCPeerID, to session: MCSession,
withContext context: Data?, timeout: TimeInterval)
```

Protocolo MCNearbyServiceBrowserDelegate

Podemos captar os momentos onde se encontra dispositivos próximos através do protocolo **MCNearbyServiceBrowserDelegate.**

Caso seja necessária a sua utilização, o protocolo deve ser adotado junto a classe que cuidará das notificações, vamos a um exemplo:

```
extension ClasseViewController : MCNearbyServiceBrowser-
Delegate {

      // Declaração dos métodos
}
```

Métodos de métodos MCNearbyServiceBrowserDelegate

```
//Método que retorna ao delegate quando um peer próximo é
encontrado:
func browser(_ browser: MCNearbyServiceBrowser, foundPeer
peerID: MCPeerID, withDiscoveryInfo info: [String : String]?)


//Método que retorna ao delegate quando um peer é perdido:
func browser(_ browser: MCNearbyServiceBrowser, lostPeer
peerID: MCPeerID)
```

Seção 15-5
MCNearbyServiceAdvertiser

Herança: NSObject > MCNearbyServiceAdvertiser

A classe **MCNearbyServiceAdvertiser** publica um aviso para um serviço específico que seu aplicativo fornece através do framework Multipeer Connectivity e notifica ao delegate sobre convites de peers próximos.

Para utilizarmos a classe, antes de criarmos avisos de serviços, precisamos criar um **MCPeerID** para identificar o aplicativo e os dispositivos próximos.

Principais Propriedades

Propriedade	Tipo	Descrição
delegate	MCNearbyService-AdvertiserDelegate?	Define o delegate
myPeerID	MCPeerID	Retorna o identificador do dispositivo
serviceType	String	Retorna o tipo de serviço que o objeto está avisando

Principais Métodos

```
//Método que inicia os avisos de serviços fornecidos pelo peer local:
func startAdvertisingPeer()
```

```
//Método que termina os avisos de serviços fornecidos pelo peer local:
func stopAdvertisingPeer()
```

Protocolo MCNearbyServiceAdvertiserDelegate

Podemos captar os momentos onde se recebe convites de dispositivos próximos através do protocolo **MCNearbyServiceAdvertiserDelegate.**

Caso seja necessária a sua utilização, o protocolo deve ser adotado junto a classe que cuidará das notificações, vamos a um exemplo:

```
extension ClasseViewController : MCNearbyServiceAdvertiser-
Delegate {

      // Declaração dos métodos
}
```

Métodos MCNearbyServiceAdvertiserDelegate

```
//Método que comunica ao delegate quando um convite para se juntar a sessão é recebido por um peer próximo:
func advertiser(_ advertiser: MCNearbyServiceAdvertiser, didReceiveInvitationFromPeer peerID: MCPeerID, withContext context: Data?, invitationHandler: @escaping (Bool, MCSession?) -> Void)
```

Seção 15-6
MCBrowserViewController

Herança: NSObject > MCBrowserViewController

A classe **MCBrowserViewController** é responsável por apresentar os dispositivos próximos ao usuário e permitir que o usuário convide estes dispositivos para criar uma sessão.

Principais Propriedades

Propriedade	Tipo	Descrição
delegate	MCBrowserView-ControllerDelegate?	Define o delegate
browser	MCNearbyService-Browser?	Retorna o objeto que é usado para descobrir peers
session	MCSession	Retorna as sessões que os usuários convidados estão conectados
maximumNumber-OfPeers	Int	Define/Retorna o numero máximo de peers

Principais Métodos

```
//Método que inicia um browser view controller com o browser
e a sessão definidos:
init(browser: MCNearbyServiceBrowser, session: MCSession)
```

Protocolo MCBrowserViewControllerDelegate

Podemos captar os momentos onde de exibição do view controller de busca através do protocolo **MCBrowserViewControllerDelegate.**

Caso seja necessária a sua utilização, o protocolo deve ser adotado junto a classe que cuidará das notificações, vamos a um exemplo:

```
extension ClasseViewController : MCBrowserViewController-
Delegate {

      // Declaração dos métodos
}
```

Métodos MCBrowserViewControllerDelegate

```
//Método que retorna ao delegado quando um novo peer
é encontrado, para que este decida se ele será exibido na
interface do usuário
optional func browserViewController(_ browserViewController:
MCBrowserViewController, shouldPresentNearbyPeer peerID:
MCPeerID, withDiscoveryInfo info: [String : String]?)
-> Bool
```

```
//Método que retorna ao delegate quando um browser view
controller é removido com peers conectados em uma sessão:
func browserViewControllerDidFinish(_ browserView-
Controller: MCBrowserViewController)



//Método que retorna ao delegate quando um browser view
controller é cancelado:
func browserViewControllerWasCancelled(_ browserView-
Controller: MCBrowserViewController)
```

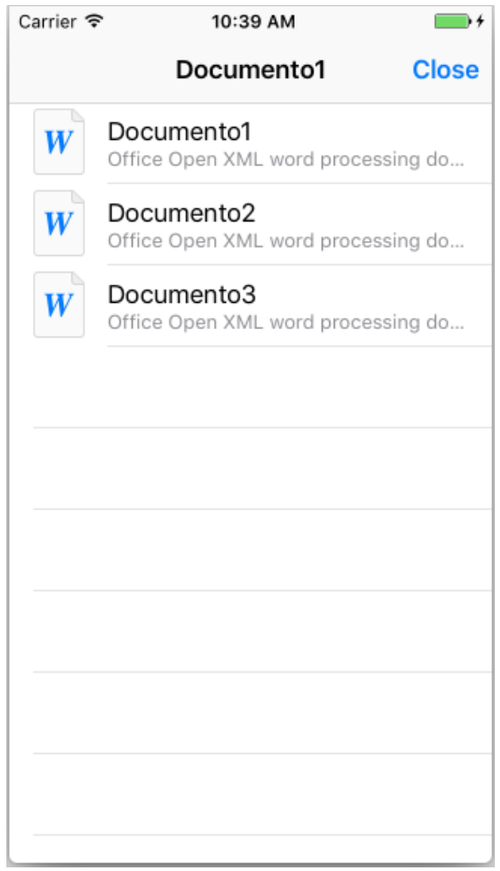
Apêndice 1

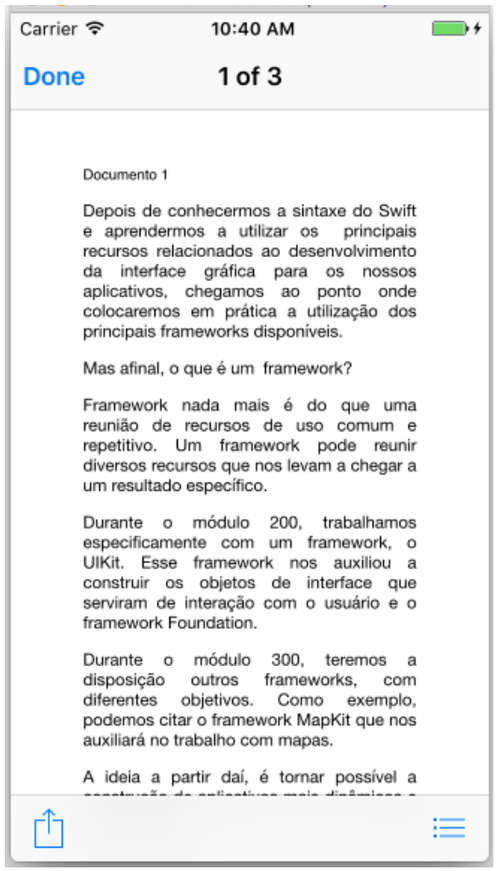
Caderno de exercícios

Visualizando com o Quick Look

Exercício 1

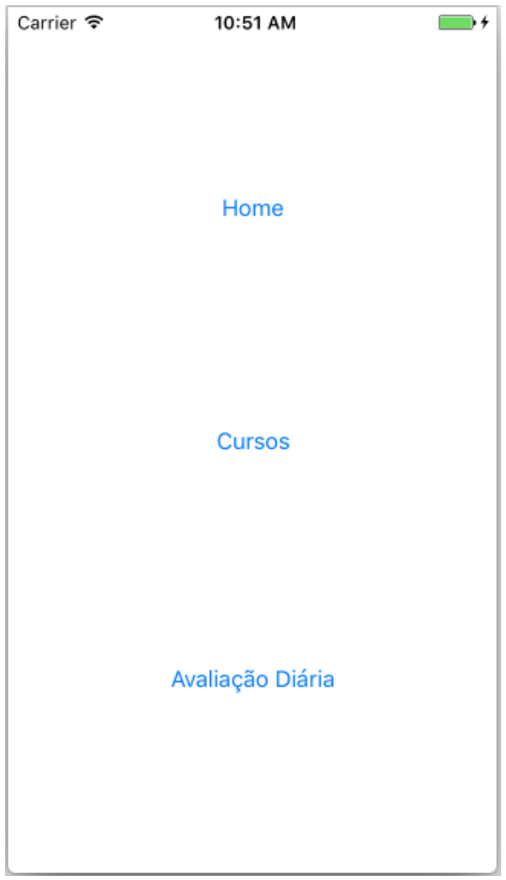
Dado três arquivos de extensão .docx, exiba-os utilizando os conceitos vistos em **Quick Look.**

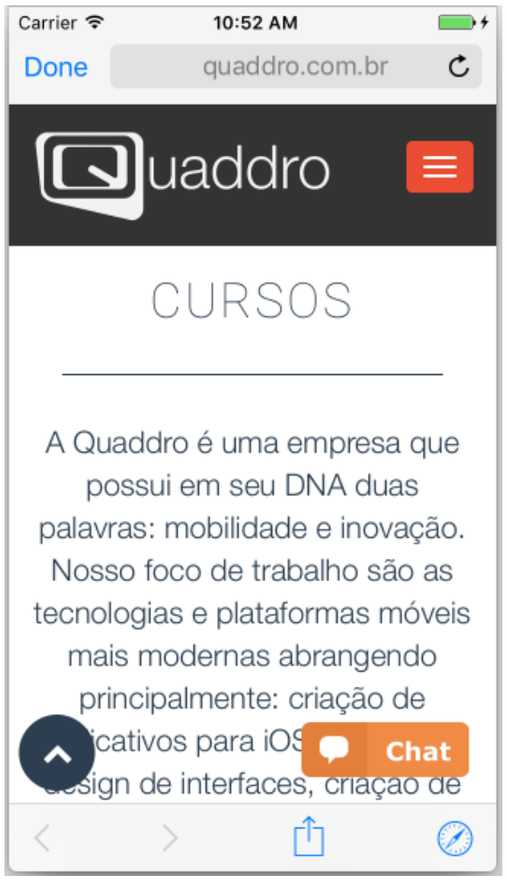




Exercício 1

Com os conceitos de **SafariViewController**, crie uma aplicação que contenha três botões, um botão direciona para a página home da Quaddro, o outro para a página de cursos e o outro para a avaliação diária.

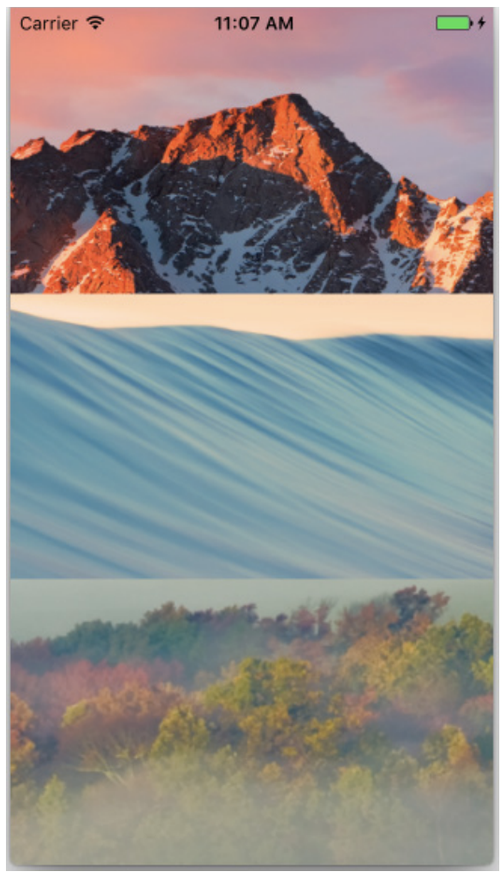


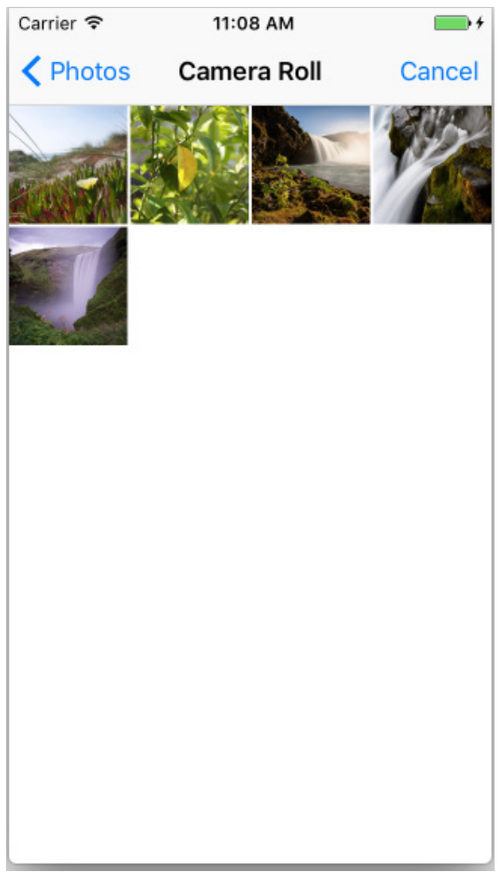


Capturando e resgatando fotos

Exercício 1

Crie uma aplicação que contenha três **imageViews** com uma imagem inicial. Ao efetuar um gesto de **LongPress** em cada uma delas o rolo da câmera é apresentado e o usuário pode escolher outra foto pra preencher o espaço. Utilize conceitos de **UIImagePickerController.**

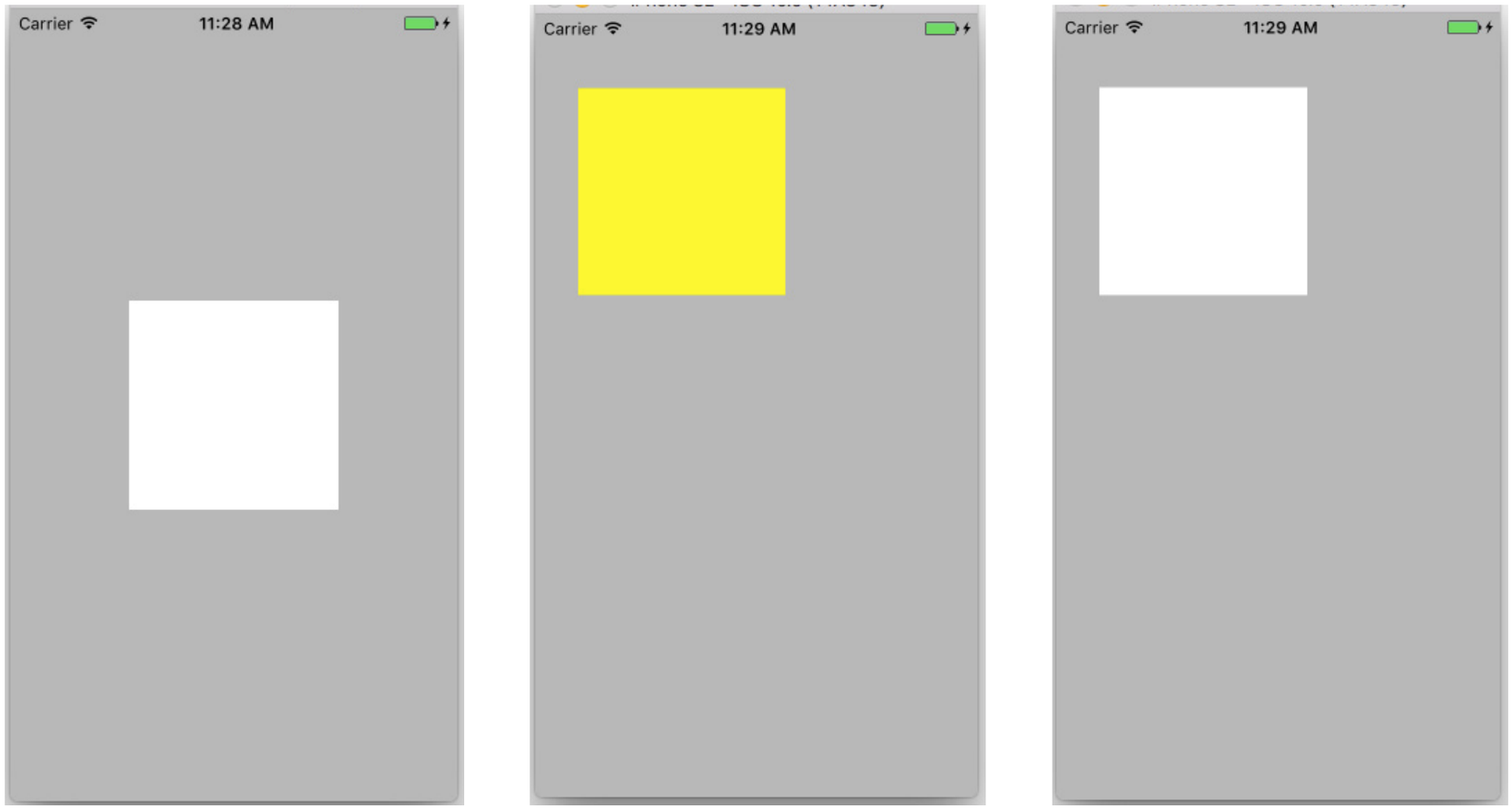




Reconhecimento de gestos

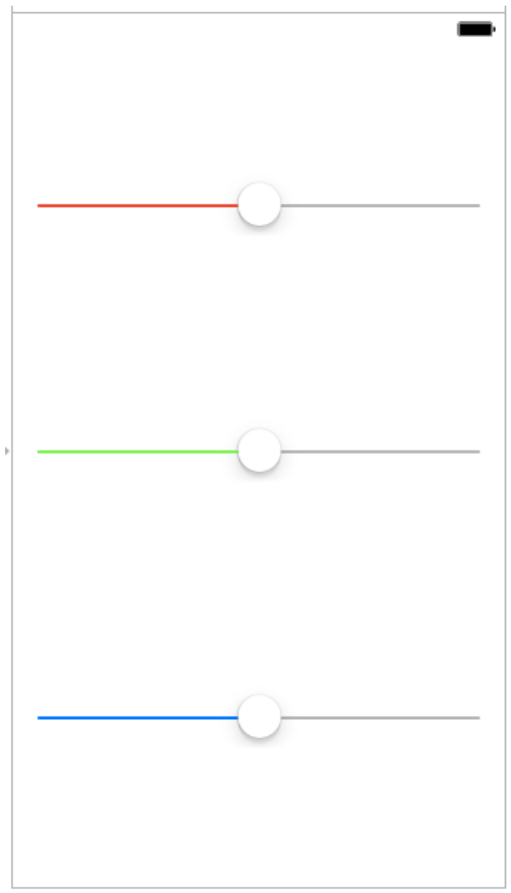
Exercício 1

Crie uma aplicação com um objeto **view** que se inicie ao centro. Ao executar um **DOUBLE TAP** a mesma deverá mudar de cor e possibilitará a alteração de sua posição com um gesto do tipo **PAN.** Um segundo doubleTap voltará o objeto para a cor inicial e a fixará a posição atual.



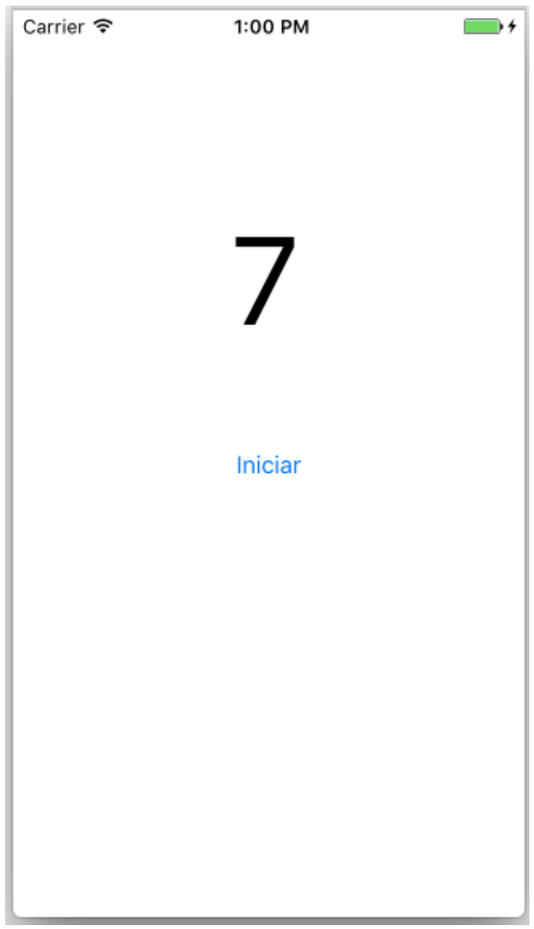
Exercício 1

Crie uma aplicação que tenha seu **background color** alterado baseado na escolha em RGB feita por três **sliders**. Quando a aplicação for reiniciada a cor deve ser a mesma da última escolha. Utilize conceitos de **Sandbox.**



Exercício 1

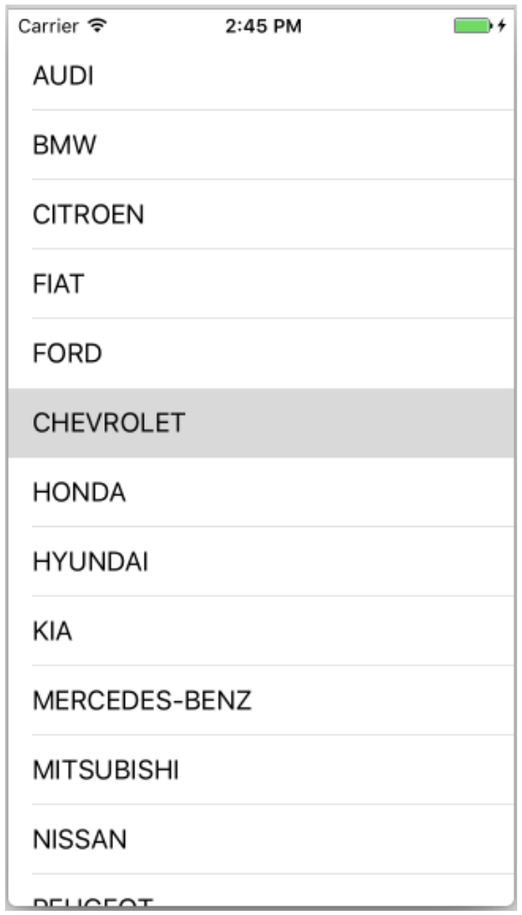
Crie uma aplicação que contenha uma **label** e um **button**. A mesma deve gerar um processo de 10 segundos que atualiza a label a cada segundo. Para simulação pode ser usado um **sleep** de **Thread**. Os processos devem ser gerenciados com **GCD** afim de não travar a interface gráfica.



Lendo dados XML e JSON

Exercício 1

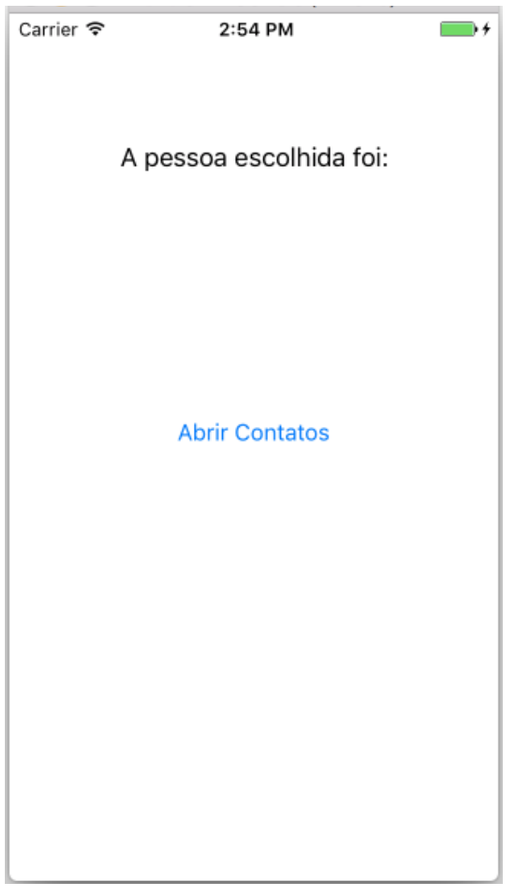
Dado um arquivo **JSON** que contém informações a respeito de marcas de carros, montar uma **tableView** apenas com o nome das marcas. (Lidas direto do arquivo JSON).

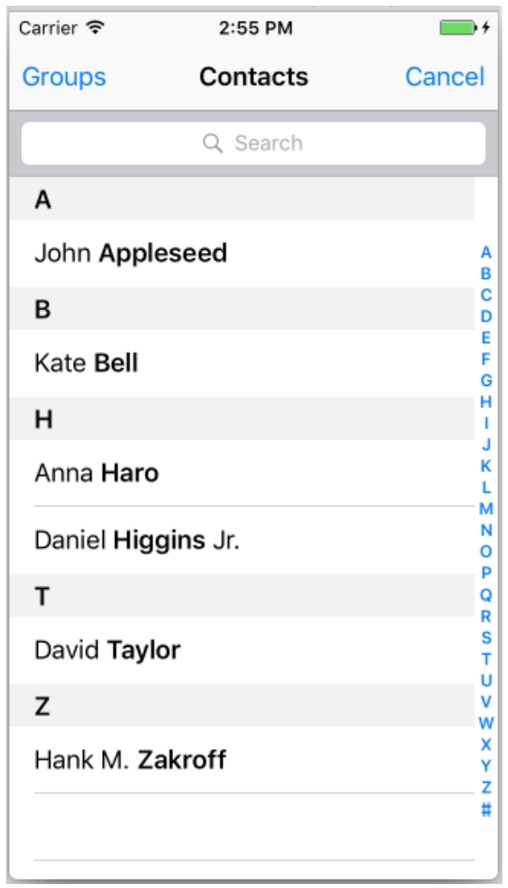


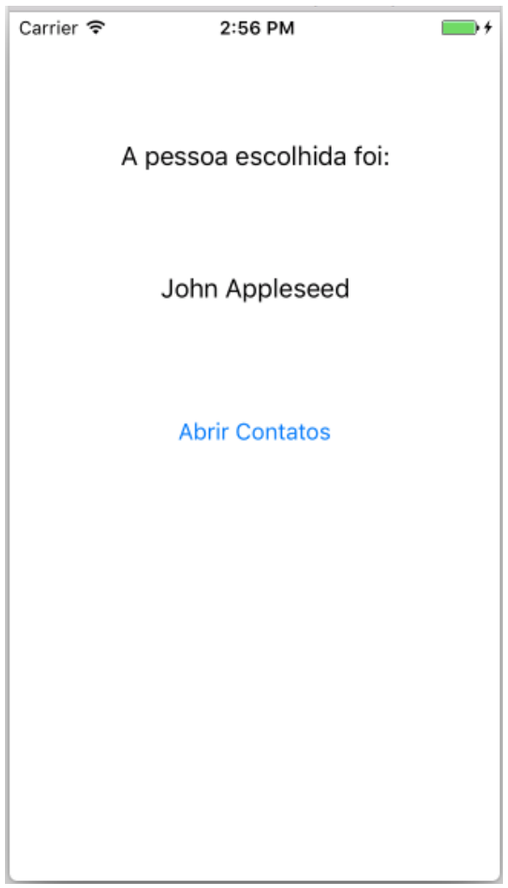
Trabalhando com contatos

Exercício 1

Crie uma aplicação que contenha um **button** que abra a lista de contatos. Quando uma escolha for feita , deve-se atribuir a uma **label** o **nome** e o **sobrenome** do contato escolhido.







Trabalhando com mapas e localização

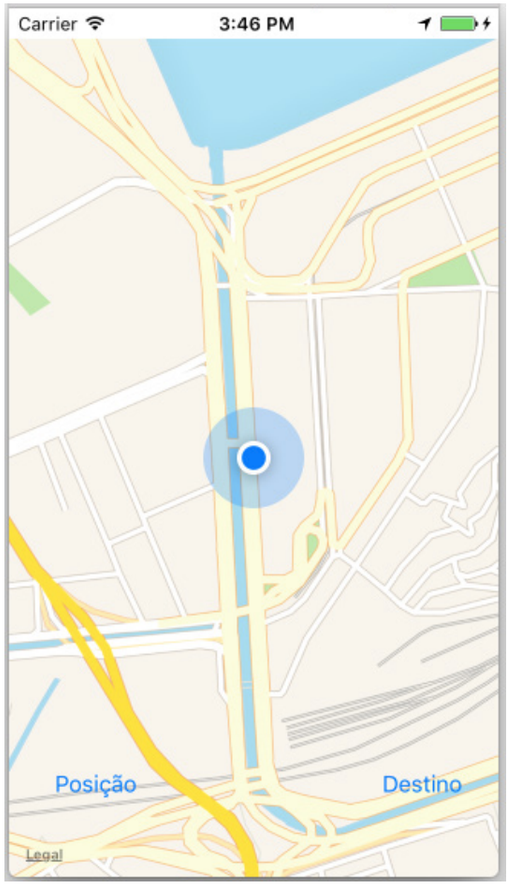
Exercício 1

Crie uma aplicação que possua um mapa de tela inteira e mostre a localização atual do usuário. Configure um pino na localização da Quaddro Treinamentos. A aplicação deve possuir dois botões:

- Botão Posição: Leva o mapa até a posição atual do usuário com uma distância (altura) pequena.

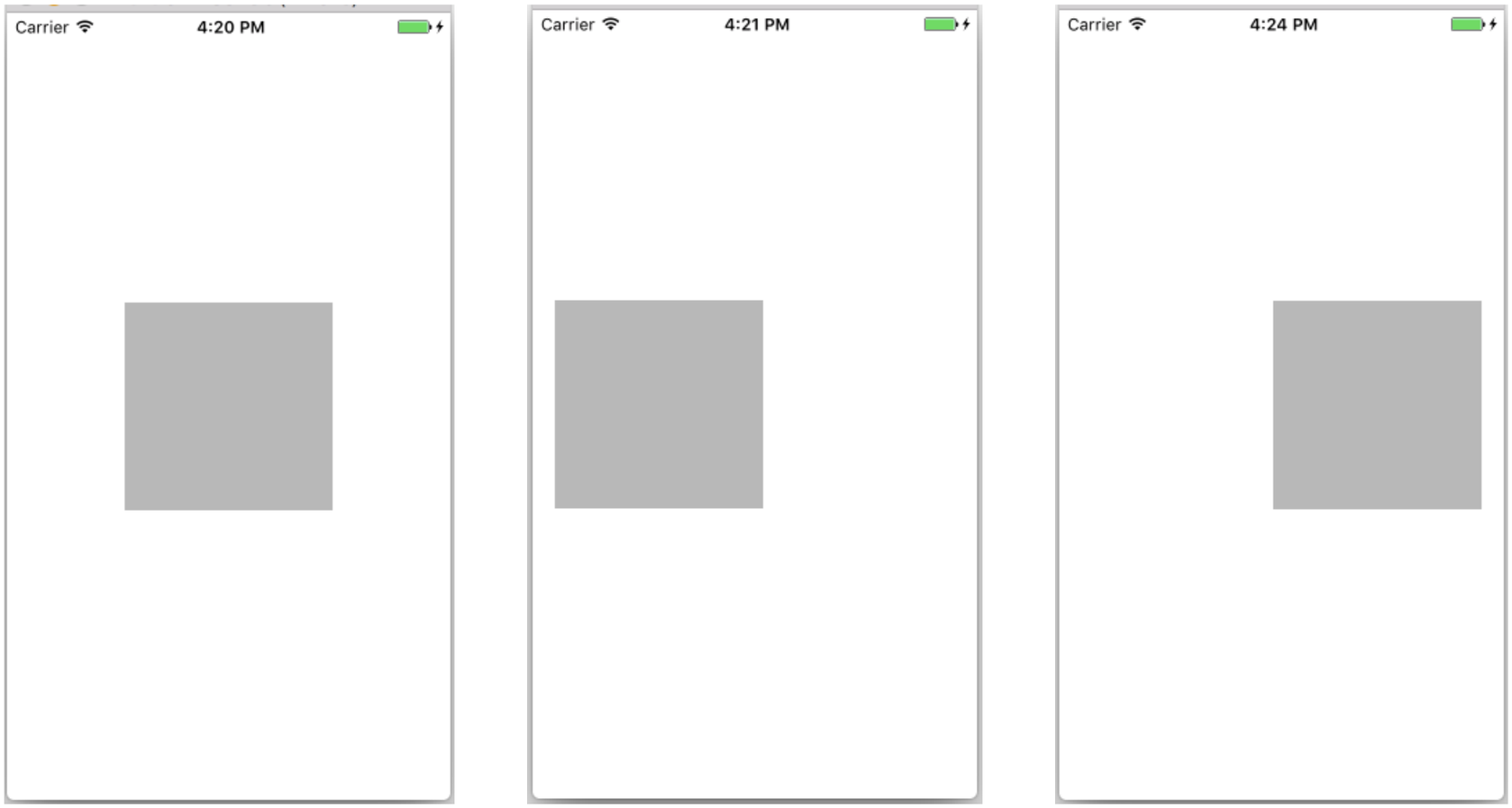
- Botão Destino: Leva o mapa até a posição do pino de destino, com uma distância (altura) pequena.

Coordenadas Quaddro: -23.566200, -46.652563



Exercício 1

Criar uma aplicação com um objeto **view** ao centro. Esse objeto deverá se mover no eixo horizontal conforme movimentação do dispositivo. Respeitando uma margem de 10 pts da view principal.



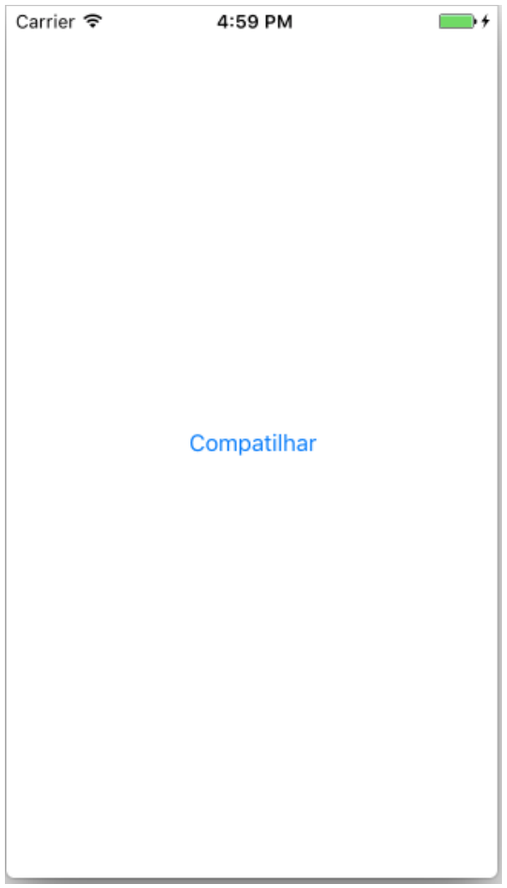
Trabalhando com áudio e vídeo

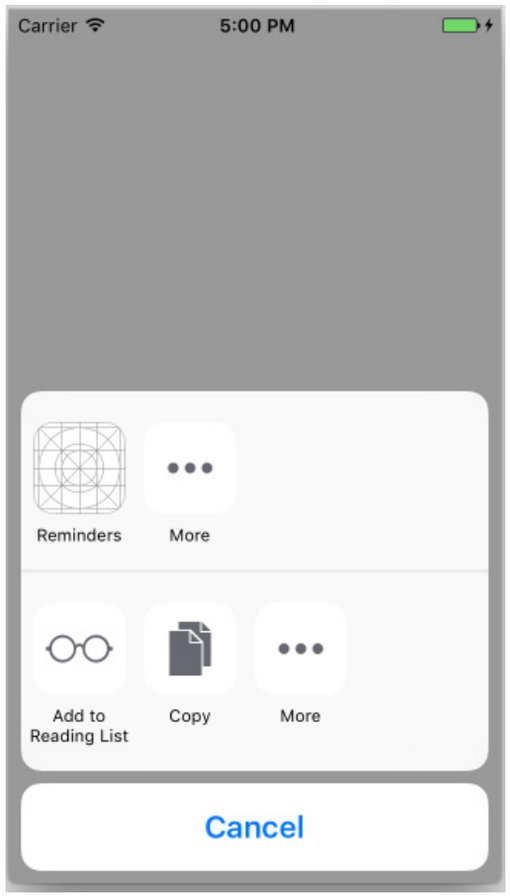
Exercício 1

Crie uma aplicação que fique tocando uma música de fundo, não é necessário interface gráfica.

Exercício 1

Dada uma URL (endereço de site), crie uma interface que contenha um **button** que ao ser clicado exiba um **UIActivityViewController** com as possibilidades de compartilhamento disponíveis.



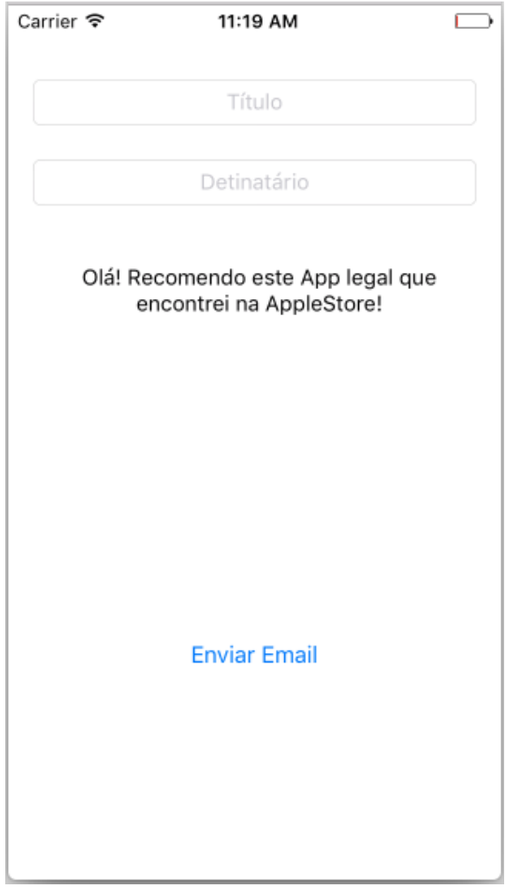


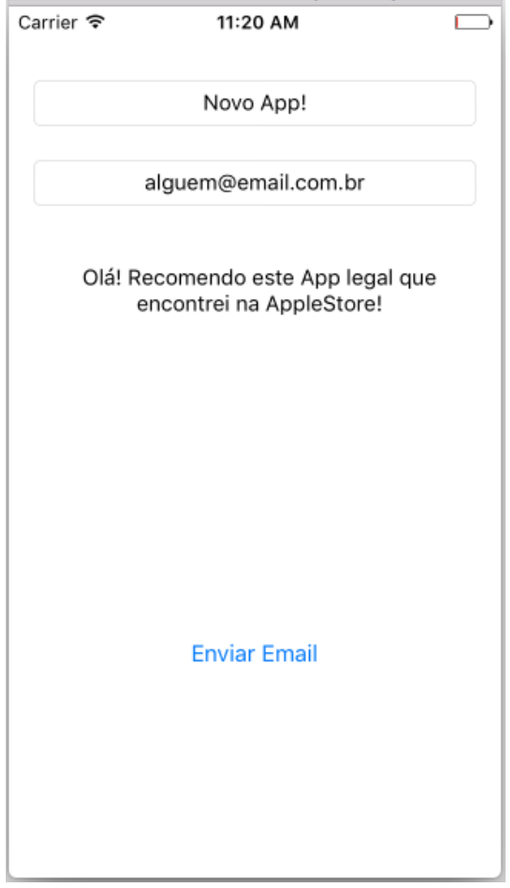
Mensagens de e-mail, SMS e redes

Exercício 1

Crie uma aplicação com um **button,** uma **textView,** e duas **textFields.** A ideia é que o usuário preencha o título da mensagem e o destinatário, sendo que o conteúdo do email será inicialmente de responsabilidade da aplicação conforme o texto escrito na textview.

Ao clicar no button, será exibido a interface padrão de envio de e-mails com os respectivos campos preenchidos.

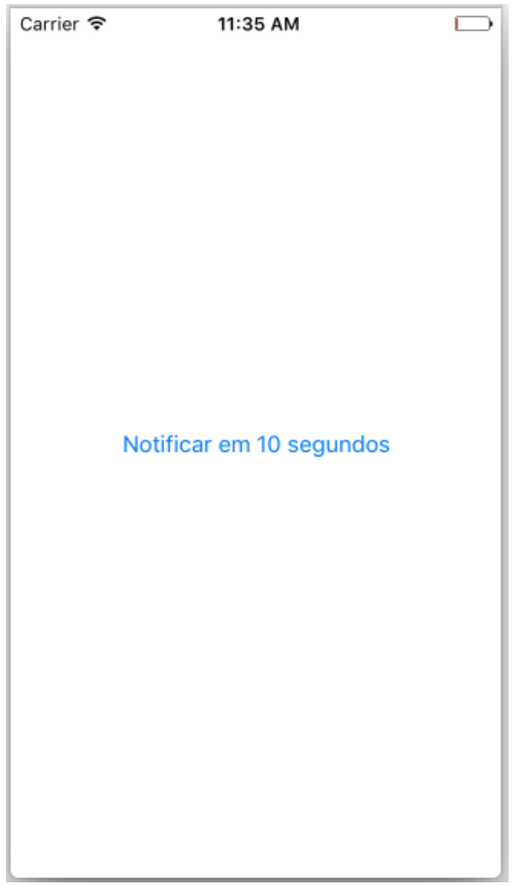




Trabalhando com notificações

Exercício 1

Criar uma interface que contenha um **button** que agende uma notificação para os próximos 10 segundos.



Trocando dados com Multipeer

Exercício 1

Utilizando conceitos de **Multipeer Connectivity,** crie uma aplicação que fará um **chat** entre dispositivos.

Essa aplicação deve ter um **button** para efetuar a busca de dispositivos disponíveis, um **textField** para digitarmos nossas linhas de papo, e um **textView** para visualizarmos o desenvolvimento do chat.