

VENTURUS4TECH

UMA SALA DE AULA DIFERENTE



Introdução ao desenvolvimento iOS Nativo e Aplicações Híbridas



Instrutores:



- Formado em Engenharia da Computação pela UNICAMP
- Desenvolvedor iOS e Android há 6 anos.
- Líder de Inovação, trabalhando no Venturus desde 2011.
- gustavo.cazangi@venturus.org.br



- Sistemas de Informação pela PUC-Campinas
- Desenvolvedor iOS há 3 anos.
- No Venturus desde Janeiro de 2015.
- silvano.junior@venturus.org.br



Interface de Desenvolvimento (IDE)

Para desenvolver aplicativos para iOS e todas as outras plataformas da Apple, é preciso utilizar o Xcode.

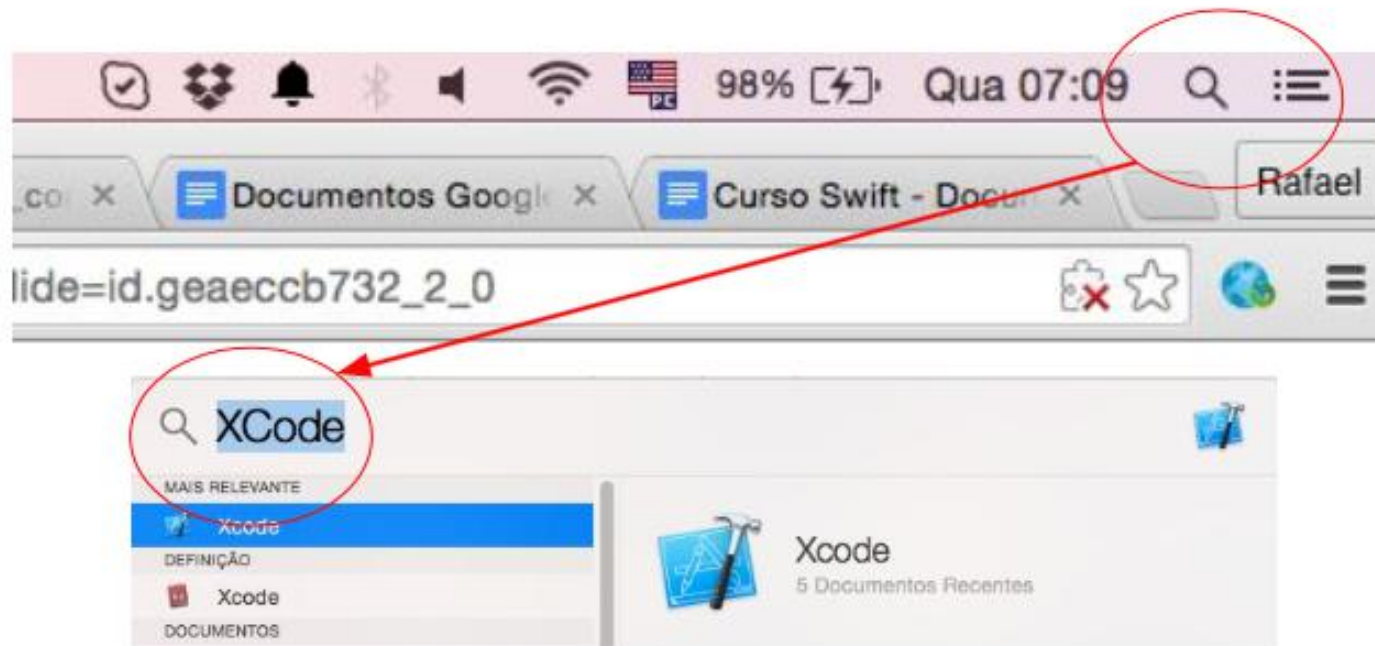
Ele é gratuito, pode ser baixado na Mac App Store e só está disponível para OSX, portanto não é possível desenvolver em Windows ou Linux.

A linguagem de desenvolvimento atual é o Swift, mas ainda é possível desenvolver em Objective-C.



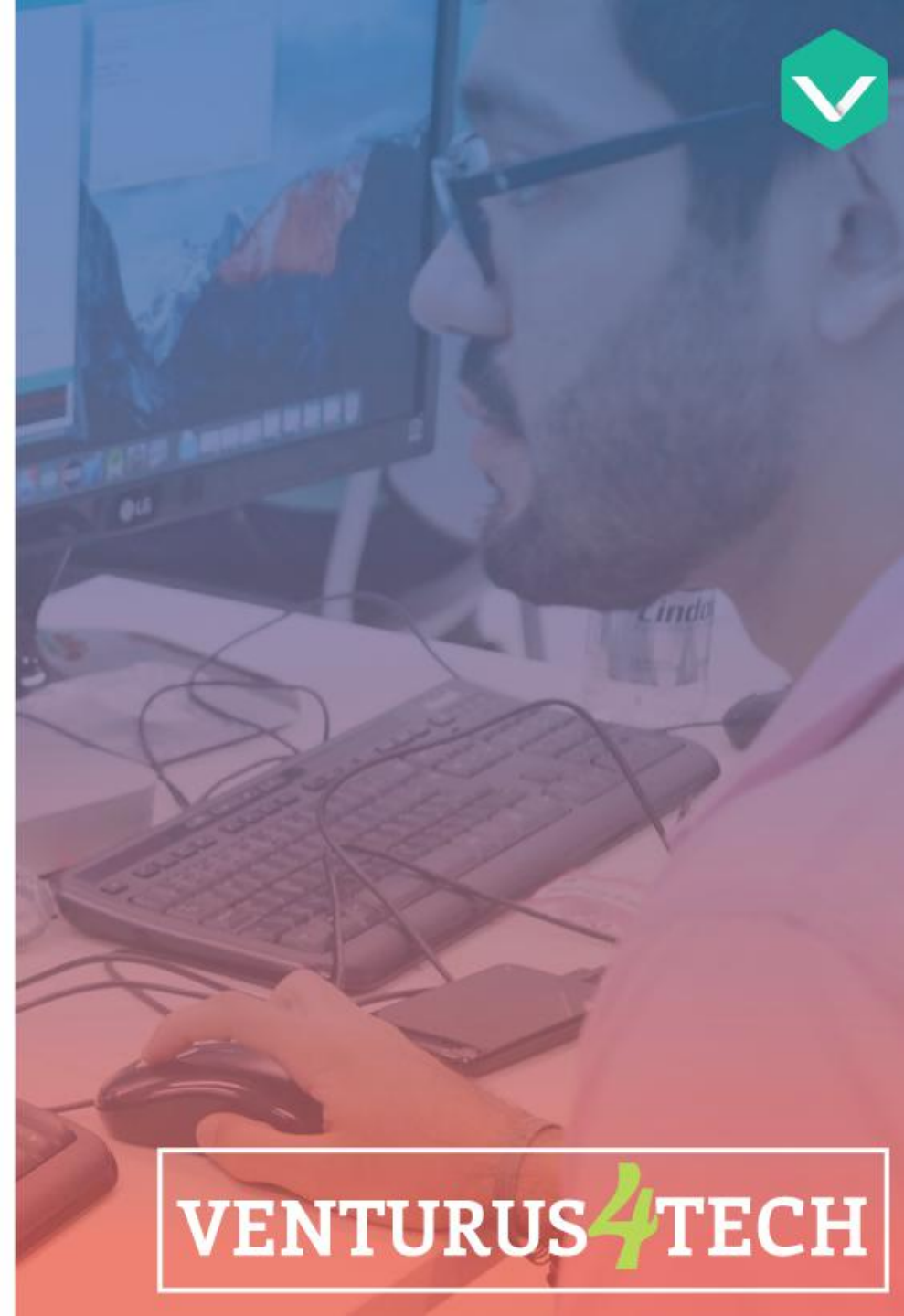
XCode

Existem várias maneiras de abrir o XCode, vamos usar o Spotlight do MAC OS X, para isso clique na lupa no canto superior direito da tela, um campo texto vai ser aberto, digite XCode e aperte Enter.



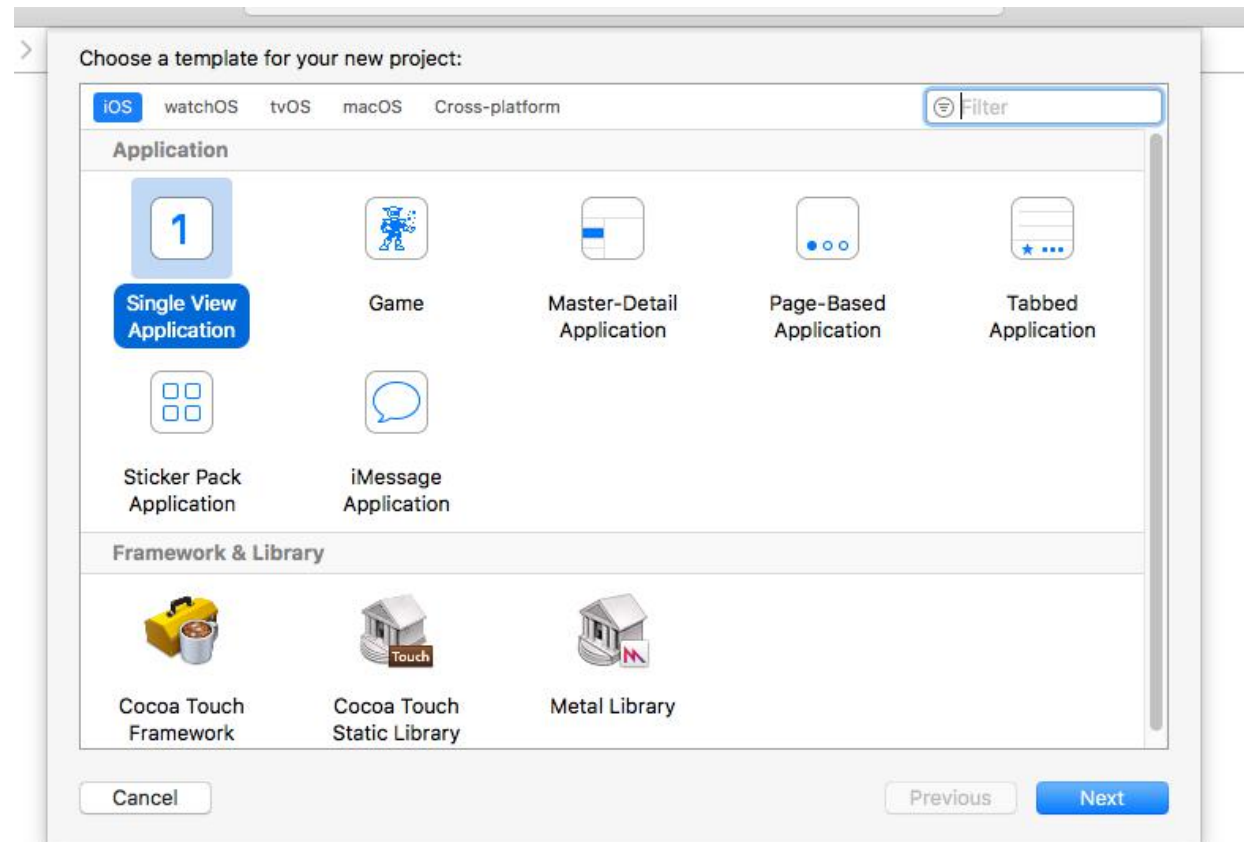
XCode

Vamos criar um novo projeto, clicando em “Create New Xcode project”



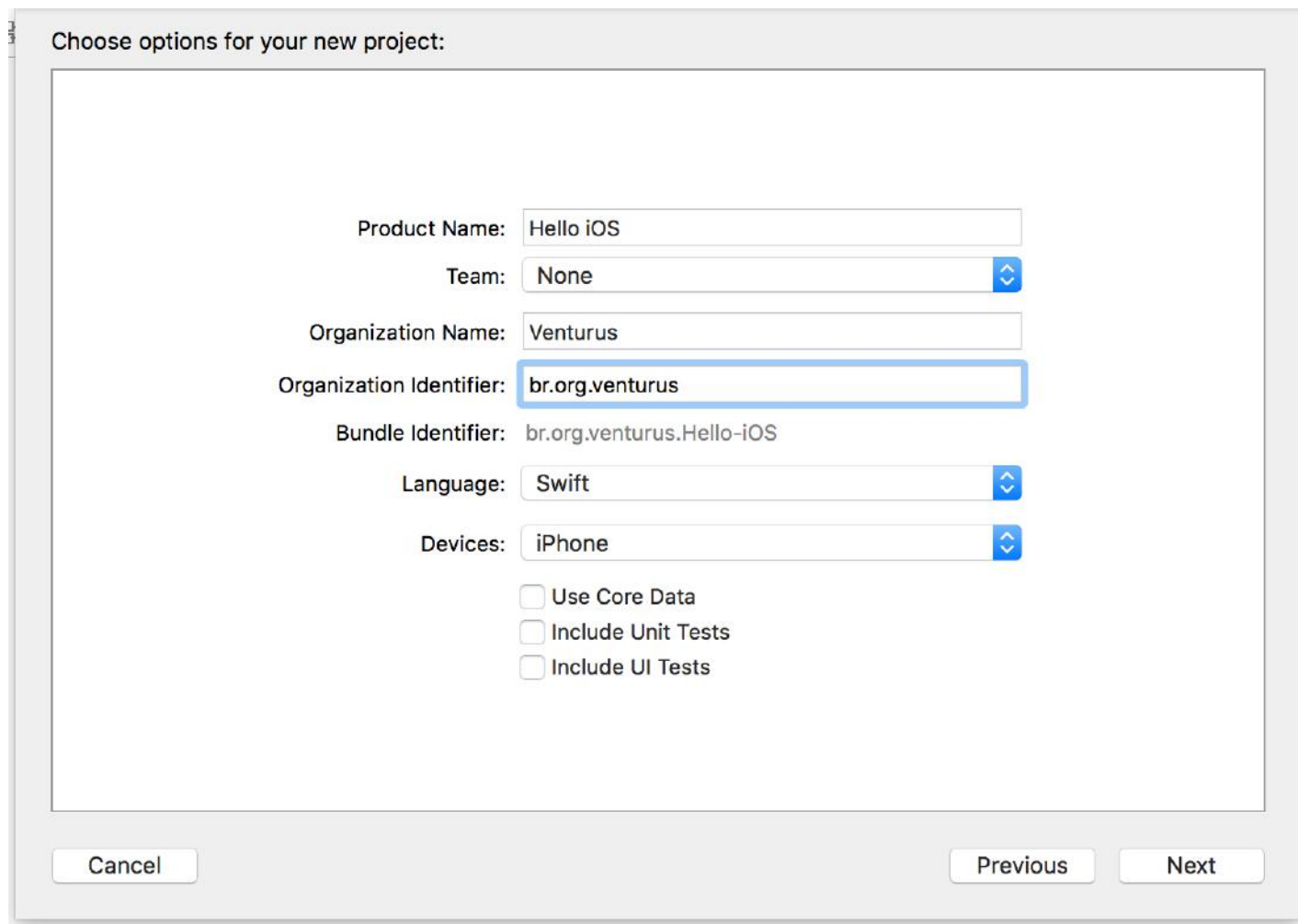
Xcode

Vamos escolher o template de projeto chamado Single View Application, nativo do iOS para fazermos o nosso primeiro aplicativo.



Hello iOS

Selecione as opções do projeto como mostrado na figura e clique em Next.



Choose options for your new project:

Product Name: Hello iOS

Team: None

Organization Name: Venturus

Organization Identifier: br.org.venturus

Bundle Identifier: br.org.venturus.Hello-iOS

Language: Swift

Devices: iPhone

☐ Use Core Data

☐ Include Unit Tests

☐ Include UI Tests

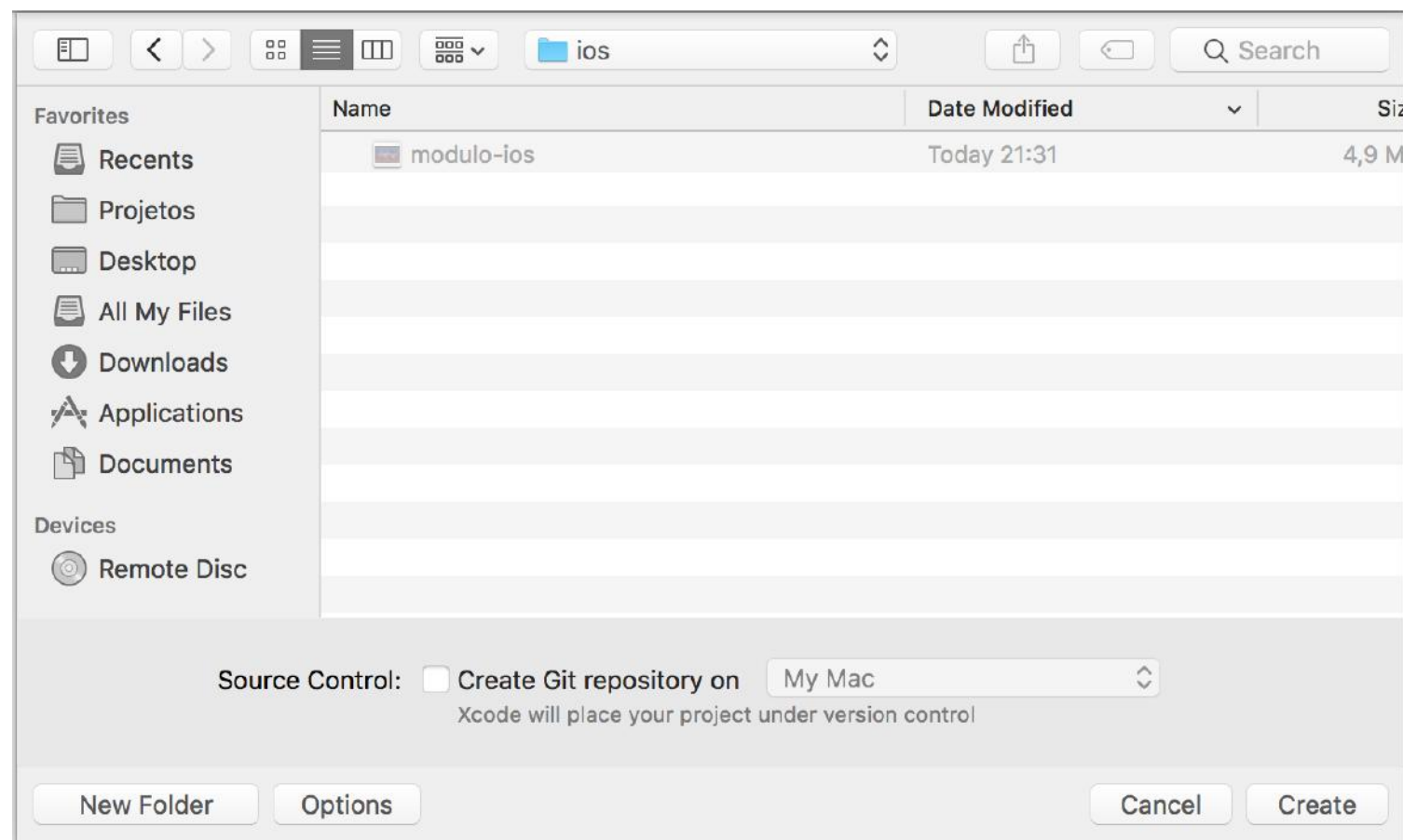
Cancel Previous Next





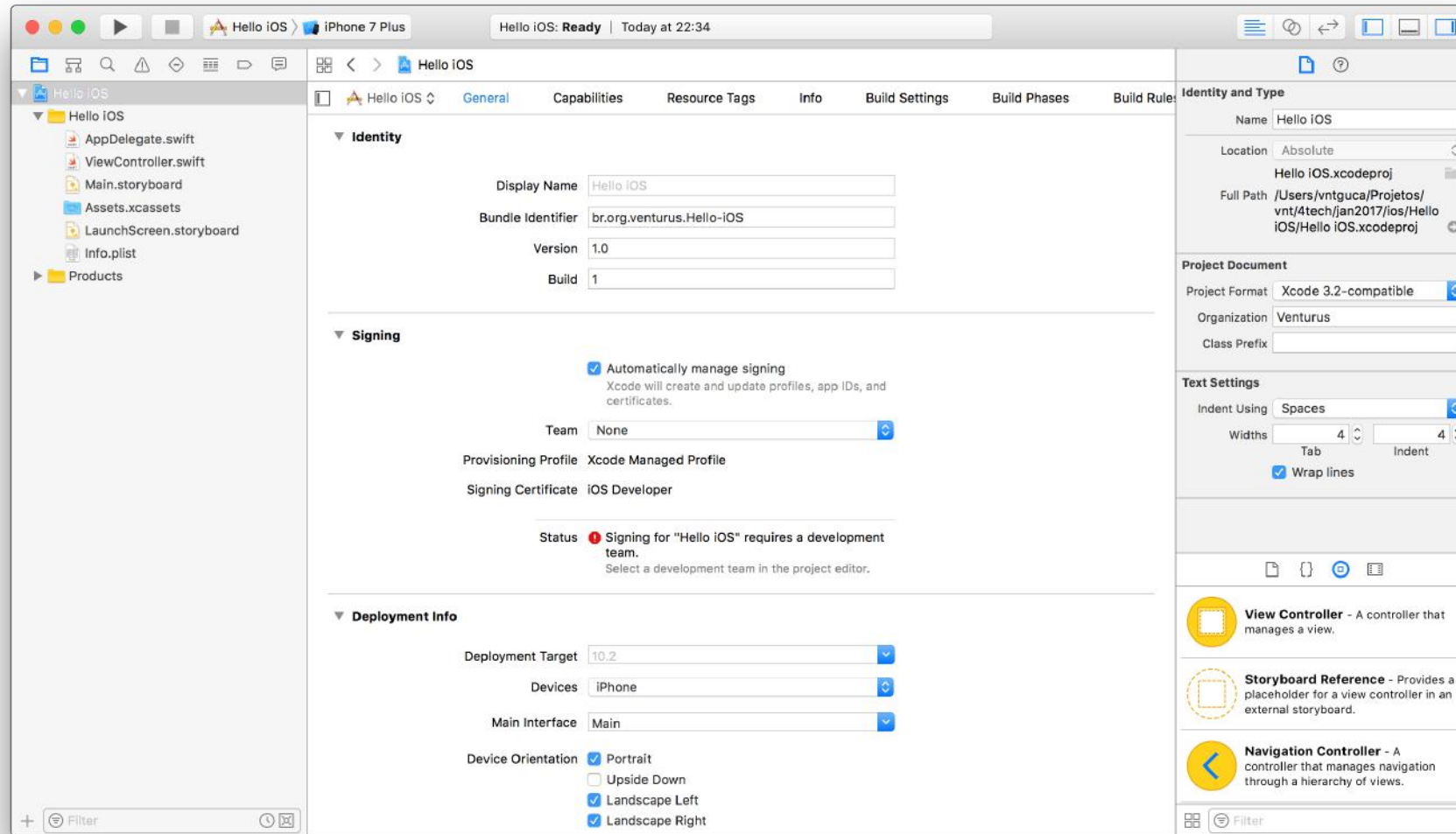
Hello iOS

Selecione uma pasta para salvar o projeto e clique em Create.



XCode

Principais elementos da tela do Xcode.



Design Pattern MVC

Para utilizarmos de maneira correta os frameworks da plataforma iOS, precisamos entender o conceito MVC, ou Model View Controller.

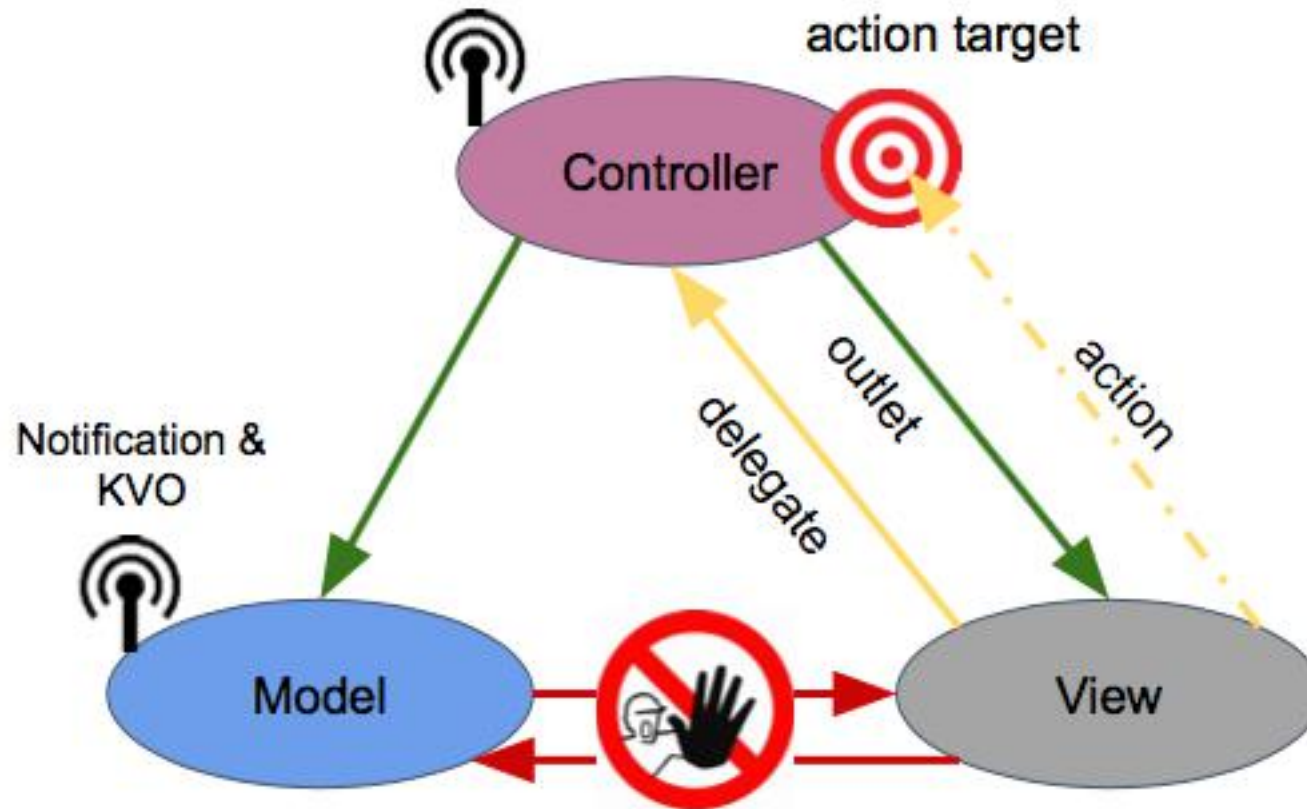
Seguindo esse conceito dividimos as aplicações são divididas em 3 partes:

- **Model:** representa os dados manipulados pela aplicação.
- **View:** é a implementação da interface do usuário que irá exibir os dados do modelo.
- **Controller:** é a parte da aplicação que manipula os dados do modelo para serem exibidos na interface do usuário.



Design Pattern MVC

A comunicação entre as partes do MVC acontecem da seguinte maneira:



Design Pattern MVC

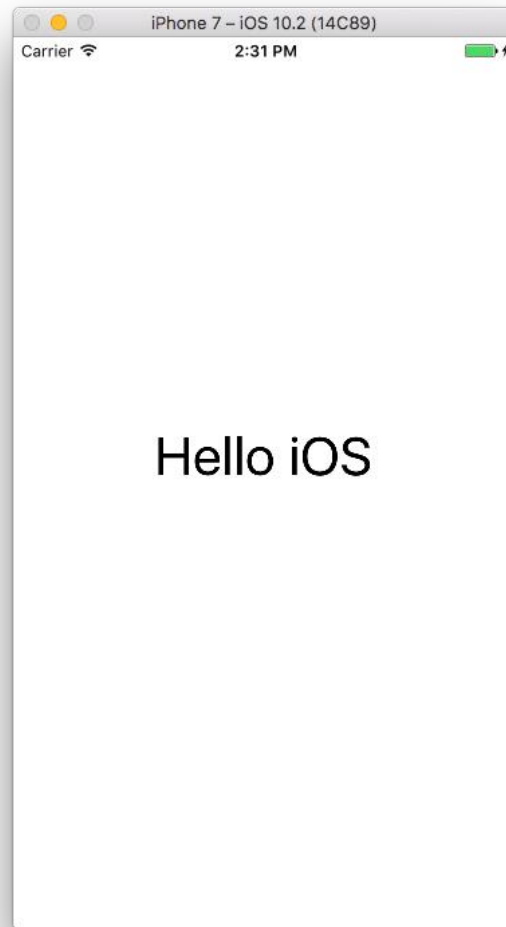
O Xcode já gera alguns arquivos com base no conceito de MVC na criação do projeto, podemos então classificar estes arquivos da seguinte maneira:

- **View:** `Main.storyboard`, esse arquivo contém a implementação da view do projeto, ou seja os elementos gráficos.
- **Controller:** `ViewController.swift`, esse arquivo contém a classe `ViewController` que representa o controller da aplicação.

E o Model? Neste projetos simples não temos nenhum elemento de modelo por enquanto.

Hello iOS

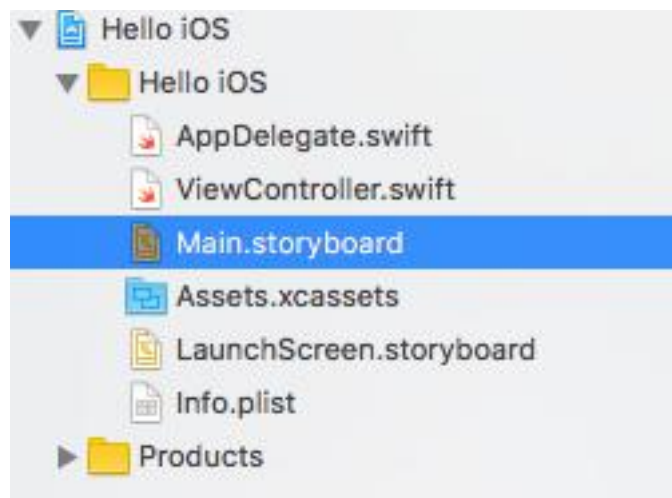
Nosso objetivo neste primeiro contato será criar uma tela simples com um texto “Hello iOS”, conforme a tela a seguir:



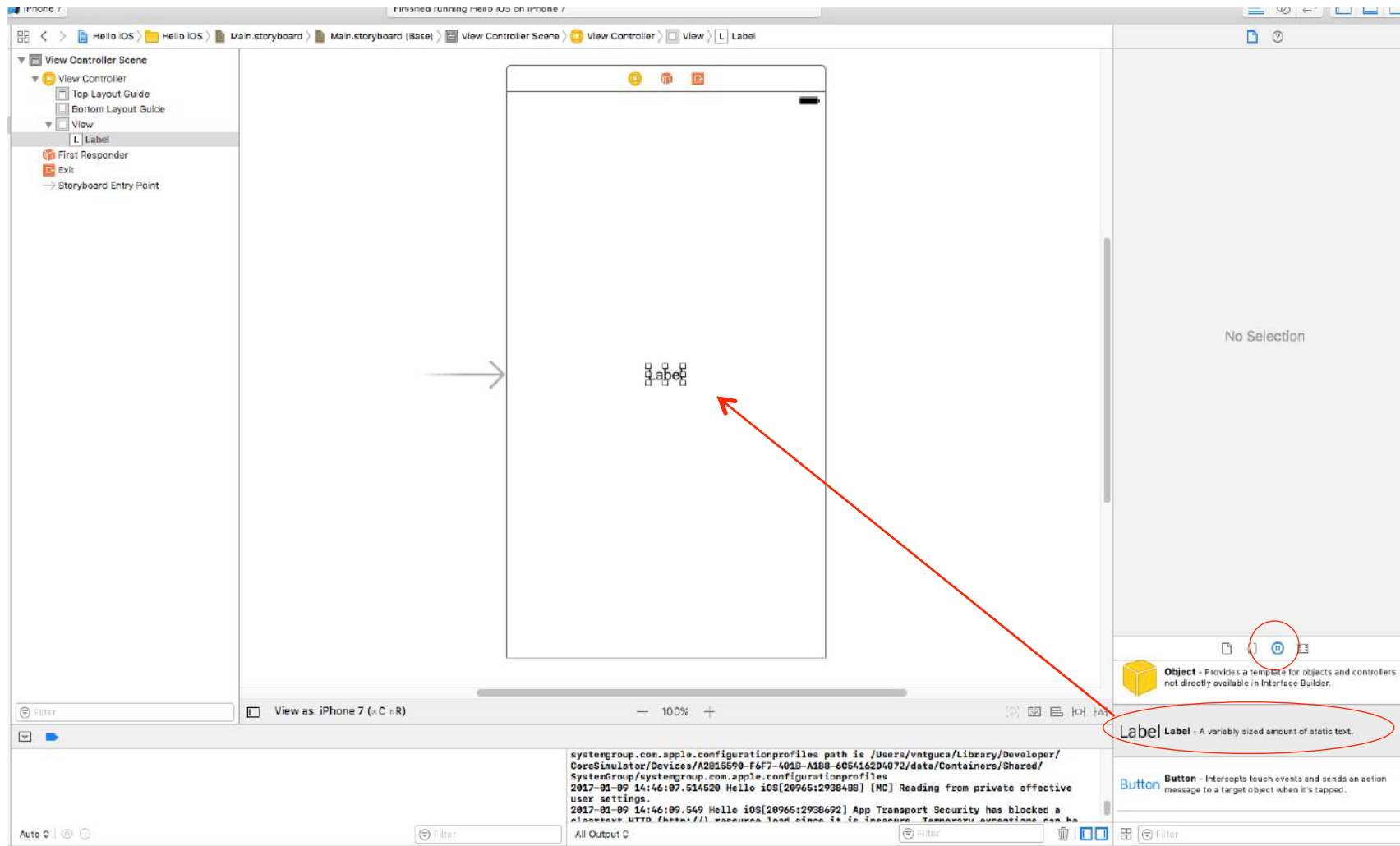
Adicionando um Label

Como vimos no padrão MVC, sabemos que o arquivo `Main.storyboard` contem a implementação da view, onde os elementos gráficos que vão ser posicionados na tela da aplicação.

Vamos abrir o arquivo com um clique único do botão esquerdo do mouse.



Adicionando um Label



Escolha o componente Label, no canto inferior direito e arraste para a interface gráfica da tela.

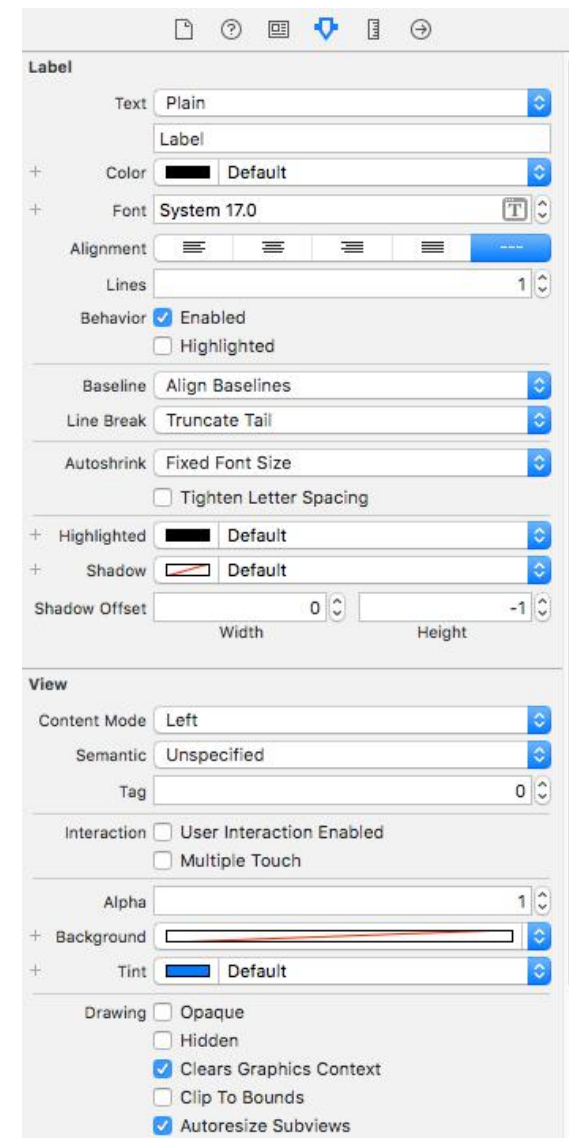
Conforme a imagem, posicione o elemento no meio da tela.

Adicionando um Label

Agora precisamos alterar o texto para “Hello iOS”, além de ajustar outros parâmetros do texto, como o tamanho, alinhamento e cor.

Podemos fazer estes ajustes na aba de inspeção de atributos, do lado direito da tela.

Vamos alterar o texto e o tamanho da fonte, e se necessário, ajustaremos também o tamanho do elemento na interface gráfica.



Rodando o Hello iOS

Pronto!

Fizemos o nosso primeiro app, que exibe “Hello iOS” na tela.

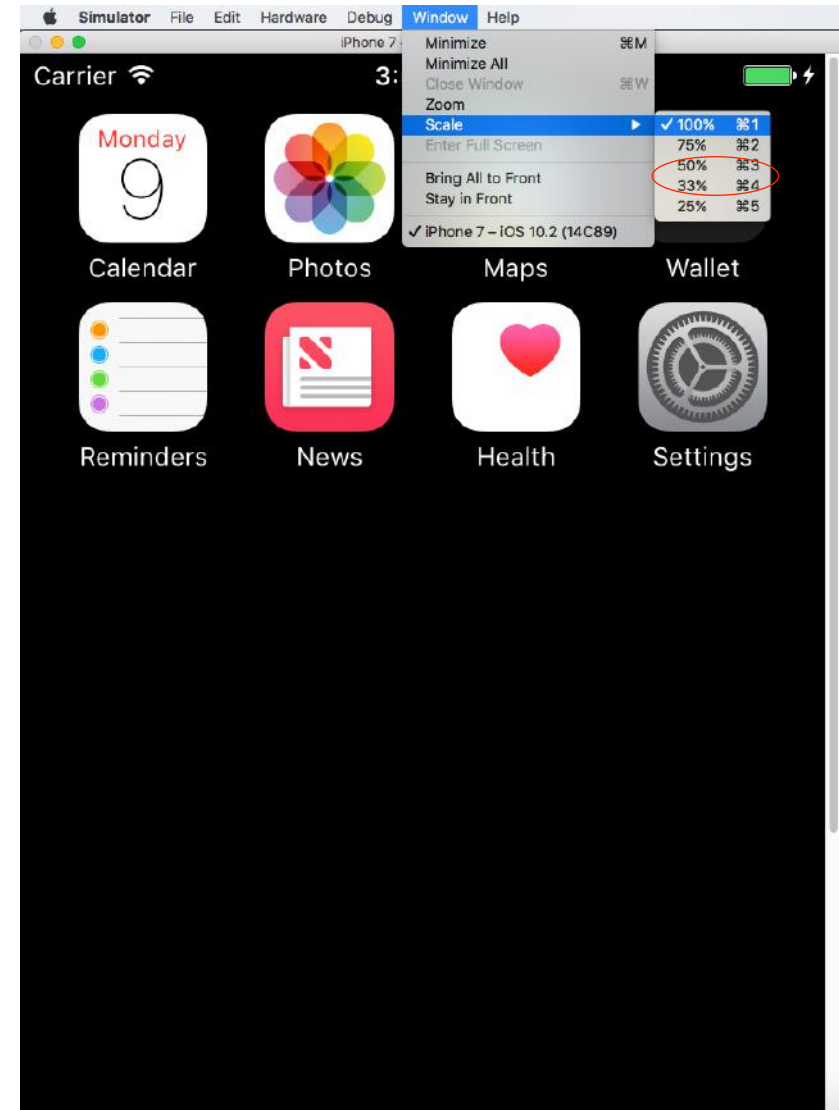
Vamos rodar o aplicativo no simulador do iPhone 7, selecionando este iphone na lista de simuladores e em seguida clicando no botão play.



Rodando o Hello iOS

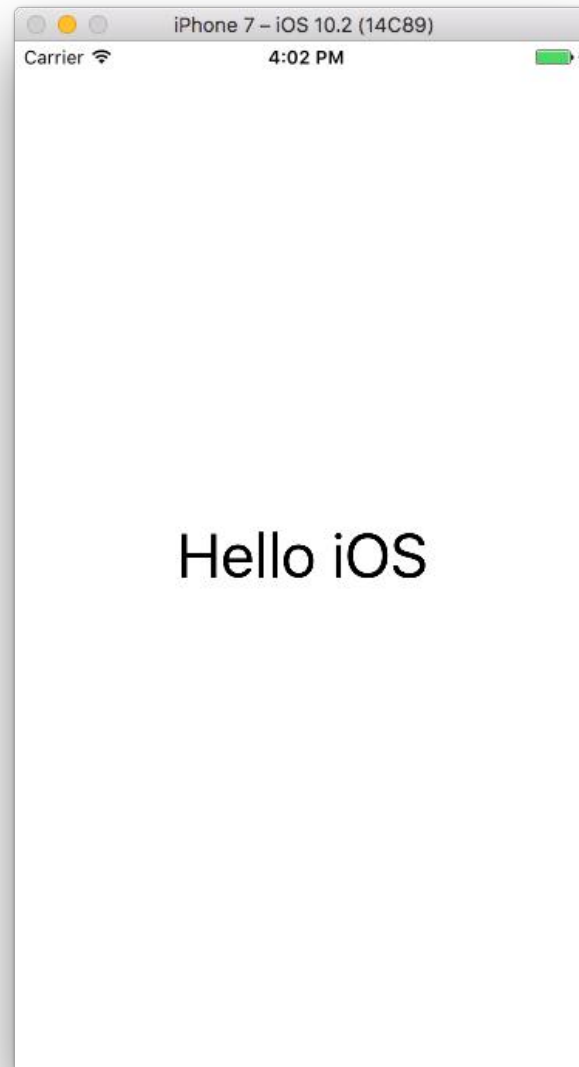
Normalmente o simulador abre com o tamanho real (em pixels) do dispositivo, o que significa uma tela enorme.

Para facilitar a visualização vamos colocar um zoom de 50%, clicando em Window - Scale - 50%, o u simplesmente pressionando Command (Windows) + 3 no teclado.





Rodando o Hello iOS



Agora sim!

**O app já está
rodando no
Iphone 7!**

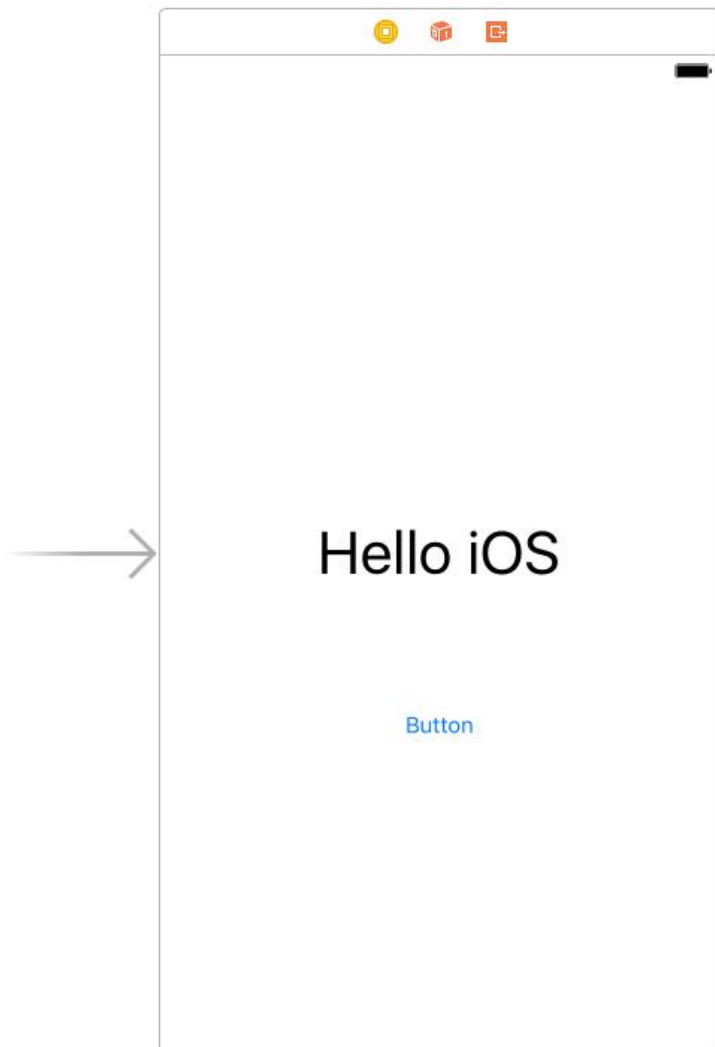


Adicionando outros controles

Vamos aprimorar o nosso aplicativo incluindo um botão para mudar a cor do texto quando o usuário clicar?

Escolha o componente Button e arraste para a interface gráfica da tela, posicionando abaixo do texto.

IBAction



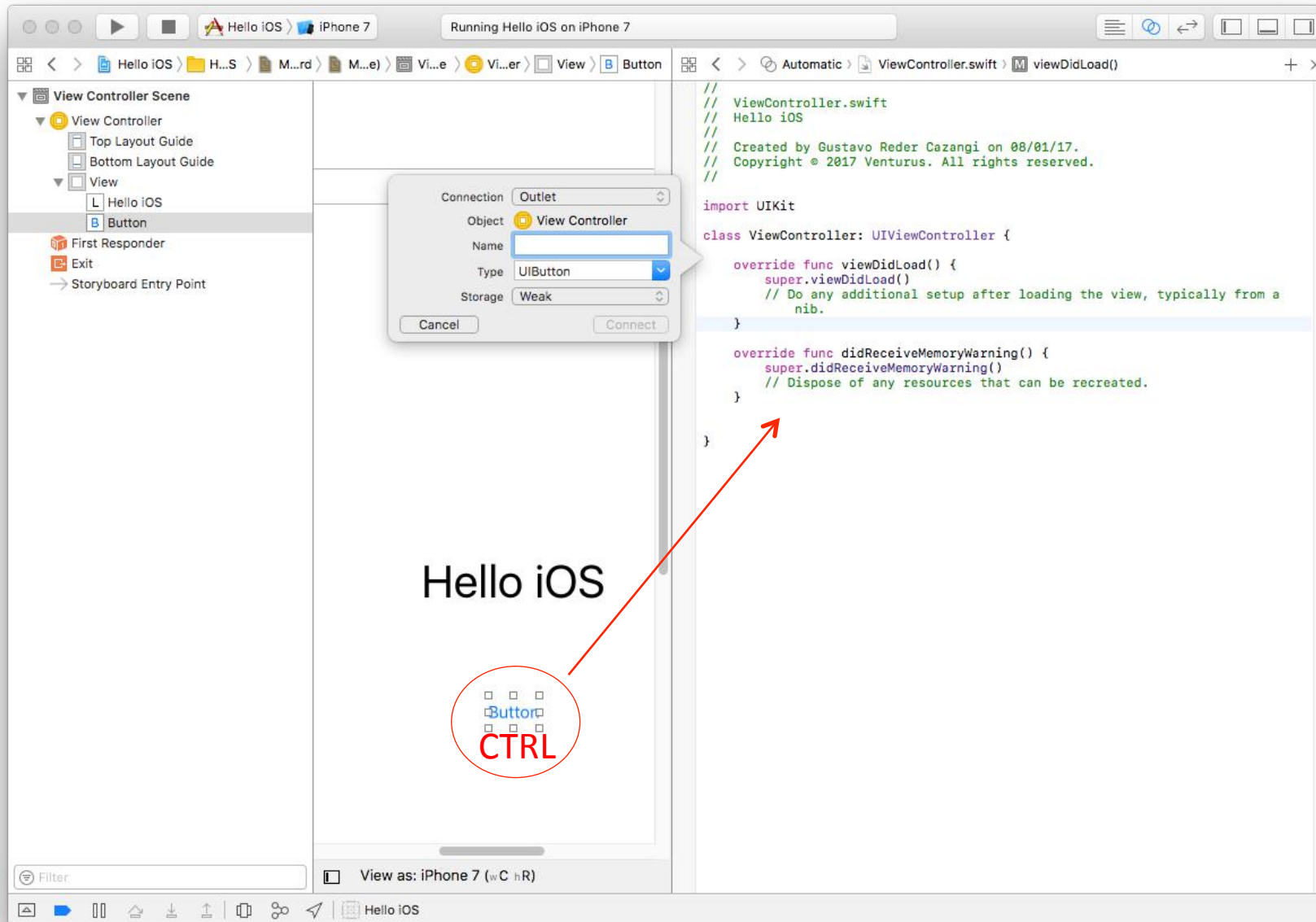
Agora que temos o botão precisamos programar qual será a ação executada quando este botão for clicado.

Para isso, criaremos um IBAction na classe de controle desta tela, lembrando do MVC, estamos tratando da classe `ViewController.swift`.

Um IBAction é um método que será executado quando um determinado evento ocorrer.



IBAction



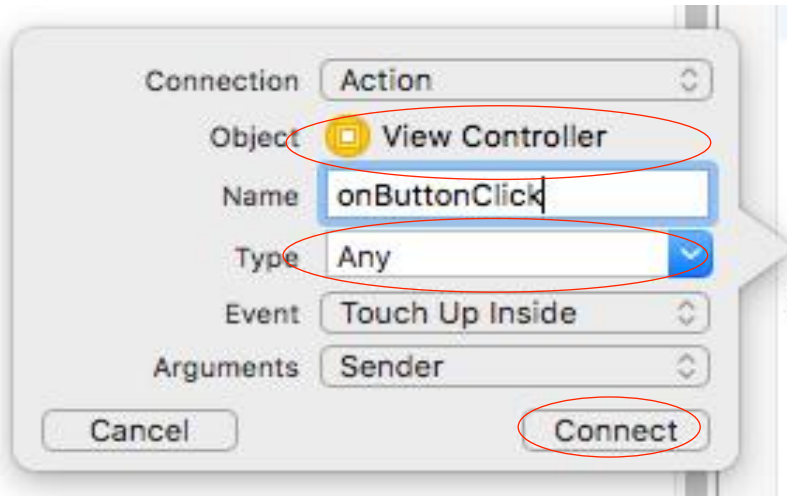
Vamos alternar para a visualização do assistente de edição.

Em seguida com a tecla CTRL apertada, vamos arrastar o botão em direção à classe ViewController e soltar.



IBAction

Selecione a conexão do tipo Action, preencha o nome do método e clique em Connect.



O código do IBAction será gerado na classe ViewController.swift.



IBOutlet

Para editar as propriedades do texto quando o botão for clicado, precisamos de uma referência para esse label.

Vamos criar uma nova conexão, agora do tipo IBOutlet, segurando CTRL e arrastando o “Hello iOS” para a classe ViewController.

IBOutlet

Selecione a conexão do tipo Outlet, preencha o nome e clique em Connect.



A variável IBOutlet será gerada na classe ViewController.



Alterando a cor do texto

Agora que temos um evento (IBAction) para o clique do botão e uma referência (IBOutlet) para o label, podemos programar a mudança de cor.

Todos os parâmetros do label e de outros elementos gráficos podem ser alterados em tempo de execução inclusive o próprio texto.

Uma possível implementação é esta ao lado.

```
class ViewController: UIViewController {  
    @IBOutlet weak var label: UILabel!  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
        // Do any additional setup after loading the nib.  
    }  
  
    override func didReceiveMemoryWarning() {  
        super.didReceiveMemoryWarning()  
        // Dispose of any resources that can be recreated.  
    }  
  
    @IBAction func onClick(_ sender: Any) {  
        if (label.textColor != UIColor.green) {  
            label.textColor = UIColor.green;  
        } else {  
            label.textColor = UIColor.red;  
        }  
    }  
}
```



Rodando o Hello iOS colorido

Vamos agora rodar o novo código do aplicativo, clicando no play do simulador novamente.

A execução anterior será interrompida e o novo código será executado.

Ao clicar no botão, a cor do texto deverá ser alterada.



Criando uma variável

Para completar esse aplicativo de introdução vamos criar uma variável que armazena o número de vezes que o botão foi clicado.

No Swift uma variável pode ser declarada da seguinte maneira:

```
var clicks = 0
```

Não é preciso declarar explicitamente o tipo da variável, o compilador consegue inferir que trata-se de um inteiro a partir da declaração. O ; no final também é opcional, e por convenção não é usado normalmente.

Já uma constante é declarada com o modificador let, exemplo:

```
let maximo = 40
```



Contador de clicks

Por fim, para contar o número de clicks no botão, podemos atualizar o valor da variável clicks a cada vez que o evento `onButtonClick` é executado:

```
@IBAction func onButtonClick(_ sender: Any) {  
    if (label.textColor != UIColor.green) {  
        label.textColor = UIColor.green;  
    } else {  
        label.textColor = UIColor.red;  
    }  
    clicks = clicks + 1;  
}
```



Exercício

Já ouviu falar do jogo do 1,2,3, PIM? Se não ouviu não tem problema, a ideia é simples, o jogador deve ir falando os números em sequência, a partir de 1, sempre trocando os múltiplos de 4 por PIM, ou seja:

1,2,3, PIM, 5, 6, 7, PIM ... pelo menos até 40.

A tarefa é simples, programe o botão para alterar o texto do label para o número de cliques, sempre mostrando PIM em vermelho, ao invés dos múltiplos de 4.

Dica: para converter um inteiro em uma string, utilize a declaração:

`"\"(meuInteiro)\""`

VENTURUS4TECH



Componentes Nativos

Além do label e do botão, existem muitos outros componentes gráficos nativos, como listas, switchs, campos de texto e outros que são usados para construir aplicações.

Entre eles existe um componente chamado WebView. Uma WebView é capaz de exibir código Web (HTML, CSS, Javascript...) de maneira formatada tal qual um browser.

Vamos criar um novo projeto, do tipo Single View Application para explorar um pouco mais esses componentes. File – New – Project ...



Componentes Nativos

Assim como no Hello iOS vamos configurar o projeto conforme a tela a seguir:

Product Name:

Team:

Organization Name:

Organization Identifier:

Bundle Identifier:

Language:

Devices:

☐ Use Core Data

☐ Include Unit Tests

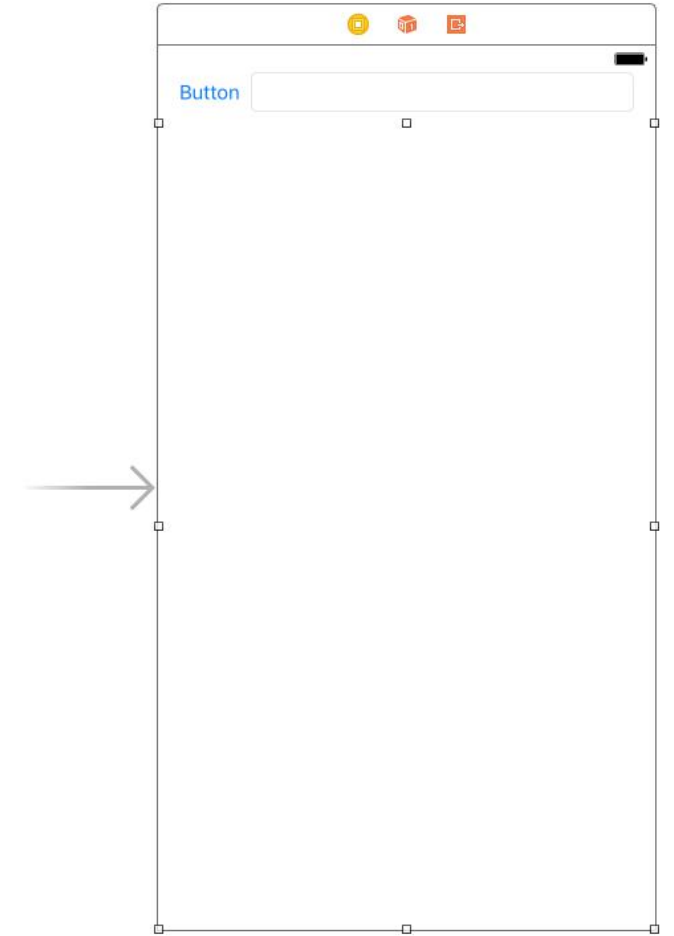
☐ Include UI Tests



Componentes Nativos

Vamos agora adicionar agora um botão, um campo de texto e uma webview à nossa aplicação.

Arraste estes itens e os posicione na tela da aplicação, ajuste também os tamanhos de cada um na tela.





Criando as Referências

Novamente precisaremos de referências para os nossos elementos gráficos, para que a classe de controle possa executar operações sobre a nossa view.

Vamos criar então dois IBOutlets, um para o campo de texto e outro para a WebView.

Vamos criar também um IBAction para o evento de clique do botão.



Criando as Referências

Nossa classe ViewController ficará parecida com o código a seguir:

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var textField: UITextField!
    @IBOutlet weak var webView: UIWebView!

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

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

    @IBAction func onClick(_ sender: Any) {

    }

}
```

Carregando conteúdo na WebView

Vamos agora carregar um conteúdo web, por exemplo uma página como o UOL, na nossa WebView.

No método `viewDidLoad()`, vamos fazer uma requisição para que a WebView carregue o endereço <https://www.uol.com.br>.

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    let url = URL(string: "https://www.uol.com.br")  
    let request = URLRequest(url: url!)  
    webView.loadRequest(request)  
}
```



HTTPS e Optional

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    let url = URL(string: "https://www.uol.com.br")  
    let request = URLRequest(url: url!)  
    webView.loadRequest(request)  
}
```

Alguns detalhes importantes, a url passada para a WebView utiliza o protocolo HTTPS. A partir do iOS9 apenas conexões seguras são aceitas automaticamente, para reforçar a segurança das aplicações.

No URLRequest passamos o parâmetro url com um ! na frente, para forçar o uso do valor da variável, já que o Swift possui um tipo especial de variável chamado Optional.

Uma variável Optional pode possuir ou não valor. Suponha que ao invés da url correta tivéssemos passado um parâmetro inválido para o construtor da URL. A variável então seria nula, sem valor, e consequentemente a aplicação iria capotar quando tentássemos utilizar seu valor com a !.



Rodando o Hello WebView

Vamos então agora executar o Hello WevView, que deve carregar a página do UOL no simulador:



Um pouco mais sobre HTTPS

Normalmente durante o desenvolvimento não utilizamos HTTPS mas sim HTTP, pois para que um site HTTPS seja válido perante o browser, o mesmo precisa ter um domínio certificado por uma CA. Esse processo é burocrático e pode custar alguns dólares por ano.

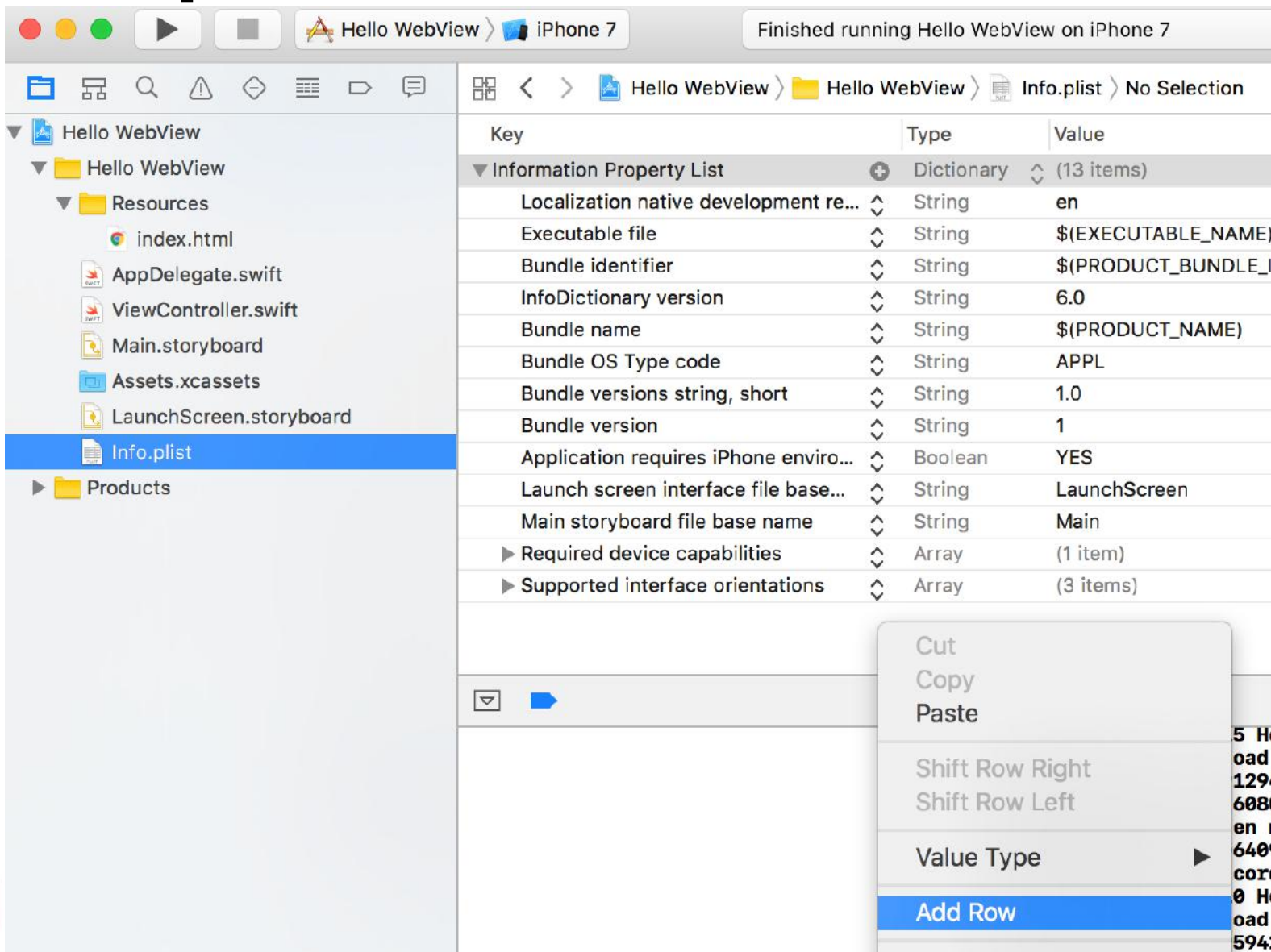
Como conseguimos lidar com a limitação de HTTPS então? O log de debug já nos dá uma dica:

```
2017-01-29 15:48:05.401 Hello WebView[76474:8763908] App Transport Security has blocked a cleartext HTTP (http://) resource load since it is insecure. Temporary exceptions can be configured via your app's Info.plist file.
```



Um pouco mais sobre HTTPS

Vamos editar o arquivo Info.plist, e incluir uma nova row:



The screenshot shows the Xcode interface with the 'Hello WebView' project selected. The 'Info.plist' file is open, displaying the 'Information Property List' with 13 items. A context menu is open over the list, showing options like 'Cut', 'Copy', 'Paste', 'Shift Row Right', 'Shift Row Left', 'Value Type', and 'Add Row' (which is highlighted).

Key	Type	Value
Information Property List (13 items)		
Localization native development re...	String	en
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_ID)
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle version	String	1
Application requires iPhone enviro...	Boolean	YES
Launch screen interface file base...	String	LaunchScreen
Main storyboard file base name	String	Main
Required device capabilities	Array	(1 item)
Supported interface orientations	Array	(3 items)

Um pouco mais sobre HTTPS

Escolhemos a chave “App Transport Security Settings”, adicionando também os dois sub-items “Allow Arbitrary Loads”, alterando seu valor para YES.

⌘ < > 📄 Hello WebView > 📁 Hello WebView > 📄 Info.plist > No Selection

Key	Type	Value
▼ Information Property List	Dictionary	(14 items)
▼ App Transport Security Settings	Dictionary	(2 items)
Allow Arbitrary Loads in Web Content	Boolean	YES
Allow Arbitrary Loads	Boolean	YES
Localization native development region	String	en
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0

Rodando novamente, o app já será capaz de carregar endereços HTTP.



Desafio

A partir dos conhecimentos que já aprendemos, e da aplicação com a webview que estamos desenvolvendo, que tal fazer um Browser simples, digamos que seu próprio Chrome para iOS?

Dicas:

Utilize o botão da interface gráfica como “Ir”.

Utilize o campo de texto para digitar a url.

Utilize este código para fechar o teclado:

```
textField.resignFirstResponder()
```



Multiplataforma

Até agora trabalhamos com componentes nativos do iOS, mas e se precisarmos desta mesma aplicação para Android, Windows Phone ou até para Desktop?

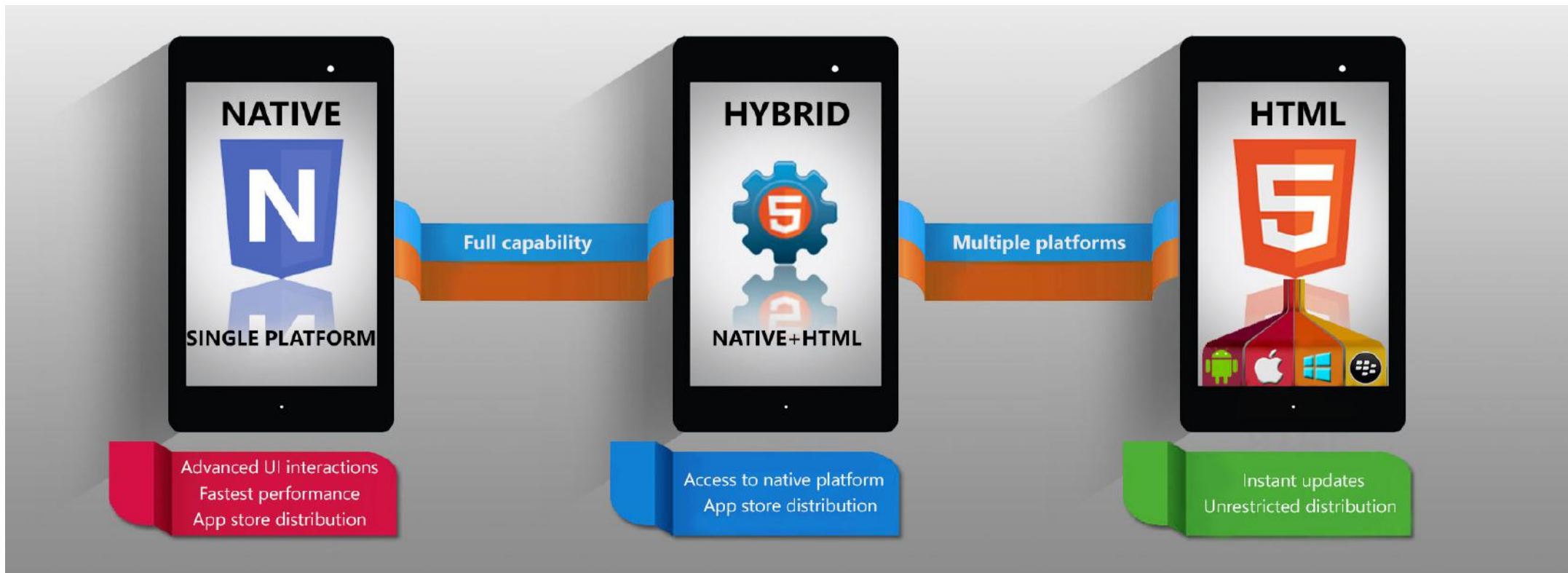
Com o desenvolvimento nativo, todo o trabalho precisa ser refeito, sequer um botão do iOS não pode ser reutilizado em outras plataformas e vice versa...

Esta é uma das principais demandas de desenvolvimento de aplicativos no mercado hoje, como reaproveitar o código e evitar o retrabalho em várias plataformas, reduzindo assim tempo de desenvolvimento e custo?



Multiplataforma

Tendo em vista essa necessidade de re-utilização, podemos classificar as abordagens de desenvolvimento de aplicações em três tipos diferentes:



Multiplataforma

HTML5 vs. Hybrid vs. Native

*Pesquisa com desenvolvedores em 2013



39%

SPEND TIME DEVELOPING
THE SAME APP/FEATURE FOR
MULTIPLE PLATFORMS



HTML5 is #1

CHOICE FOR BUILDING APPS FOR
MULTIPLE MOBILE PLATFORMS



Source: KendoUI.com

VENTURUS4TECH



Multiplataforma

Não existe uma “melhor” abordagem para todos os casos.

É preciso analisar com cuidado as necessidades de cada aplicativo para decidir qual é a melhor opção, com base nos prós e nos contras de cada cenário.



Multiplataforma – Aplicativos Híbridos

Dentro de cada uma das abordagens existem também variações no forma de desenvolvimento, que consistem em utilizar diferentes tecnologias ou ferramentas.

Em geral nos aplicativos híbridos, temos uma WebView (componente nativo) que exibe um conteúdo Web armazenado na própria aplicação.

Existem diversos frameworks que permitem que o desenvolvedor trabalhe apenas com a parte Web e que se encarregam de exportar a aplicação e gerar a parte nativa do aplicativo.



PhoneGap



Xamarin



Ionic ✓

VENTURUS4TECH

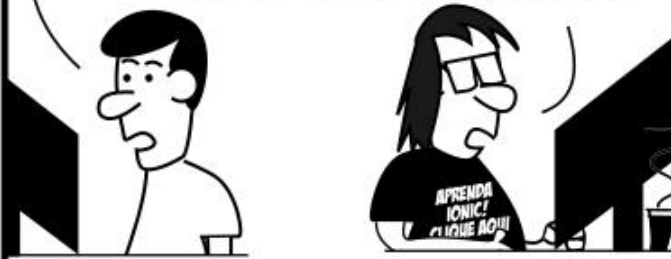
VIDA DE
PROGRAMADOR
COM.BR



Academia
WebApps

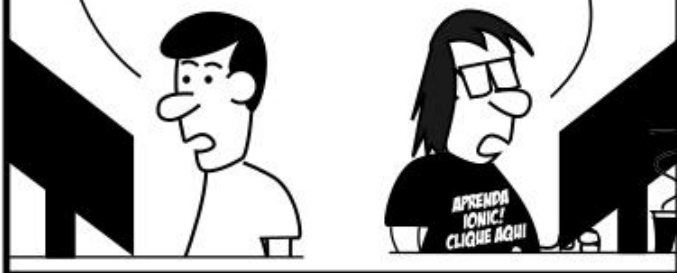
PENSEI EM COMEÇAR A PROGRAMAR
PRA CELULAR, MAS NÃO QUERIA
APRENDER JAVA, OBJECTIVE C E
MAIS UM MONTE DE COISAS...

CARA, POR QUE VOCÊ NÃO
USA **IONIC**? DAÍ VOCÊ NÃO
PRECISA APRENDER NEM
JAVA NEM OBJECTIVE C



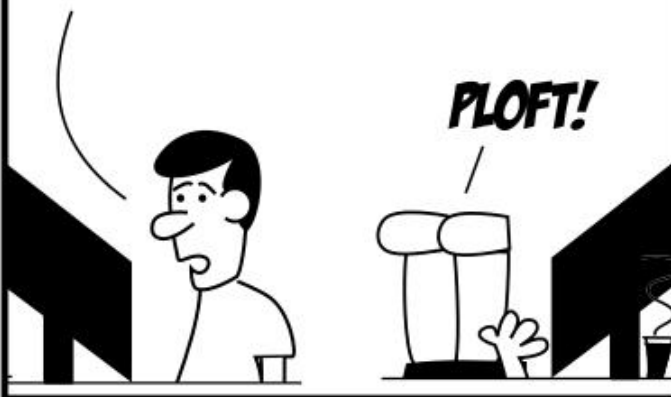
HMMM... GOSTEI DA IDEIA!
E COMO USA ISSO?

COM O IONIC VOCÊ PODE
CRIAR APPS PARA ANDROID
E IPHONE USANDO HTML E
JAVASCRIPT!



IH, DAÍ TEM QUE
APRENDER HTML
E JAVASCRIPT...
MELECA!

PLOFT!



Multiplataforma

VENTURUS4TECH

Multiplataforma – Aplicativos Híbridos

Voltando ao nosso aplicativo do browser, já temos uma parte pronta para executar um aplicativo híbrido: a WebView nativa já está preparada para exibir URLs da Web.

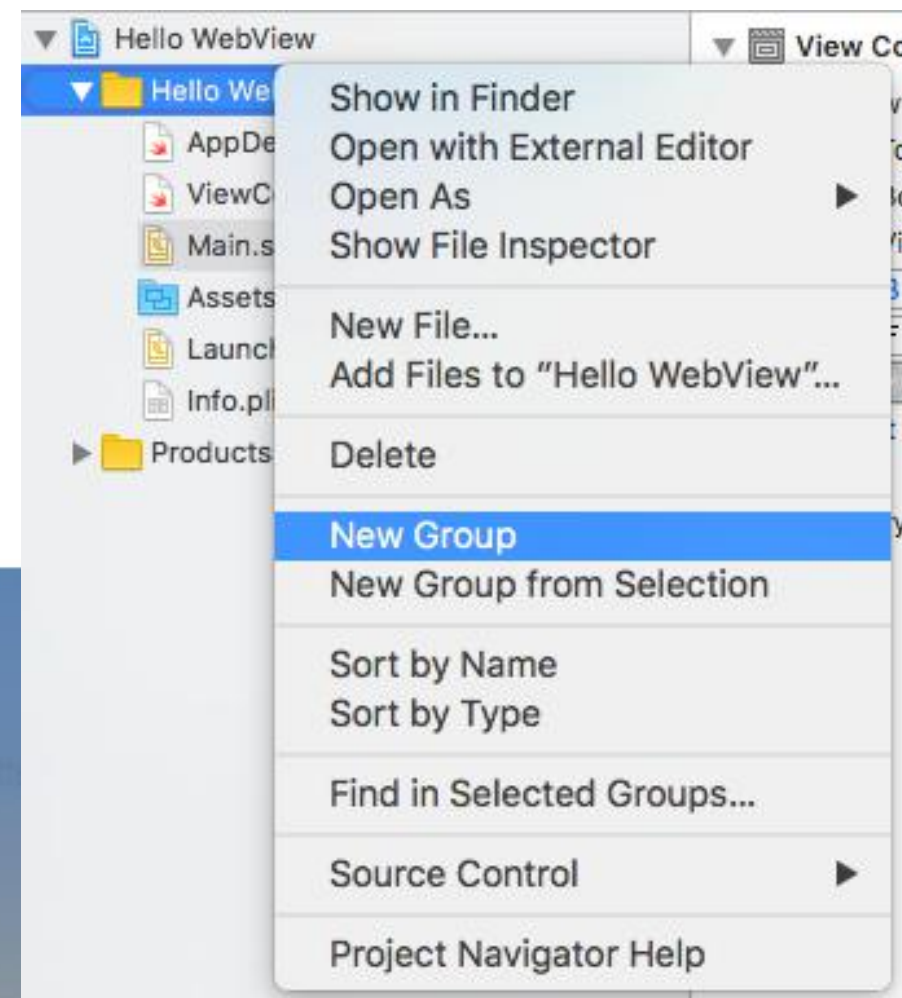


Vamos aprender agora como carregar conteúdo direto de arquivos HTML armazenados dentro do próprio aplicativo, para tornar nosso aplicativo híbrido e sem depender de internet.



Carregando os arquivos Web

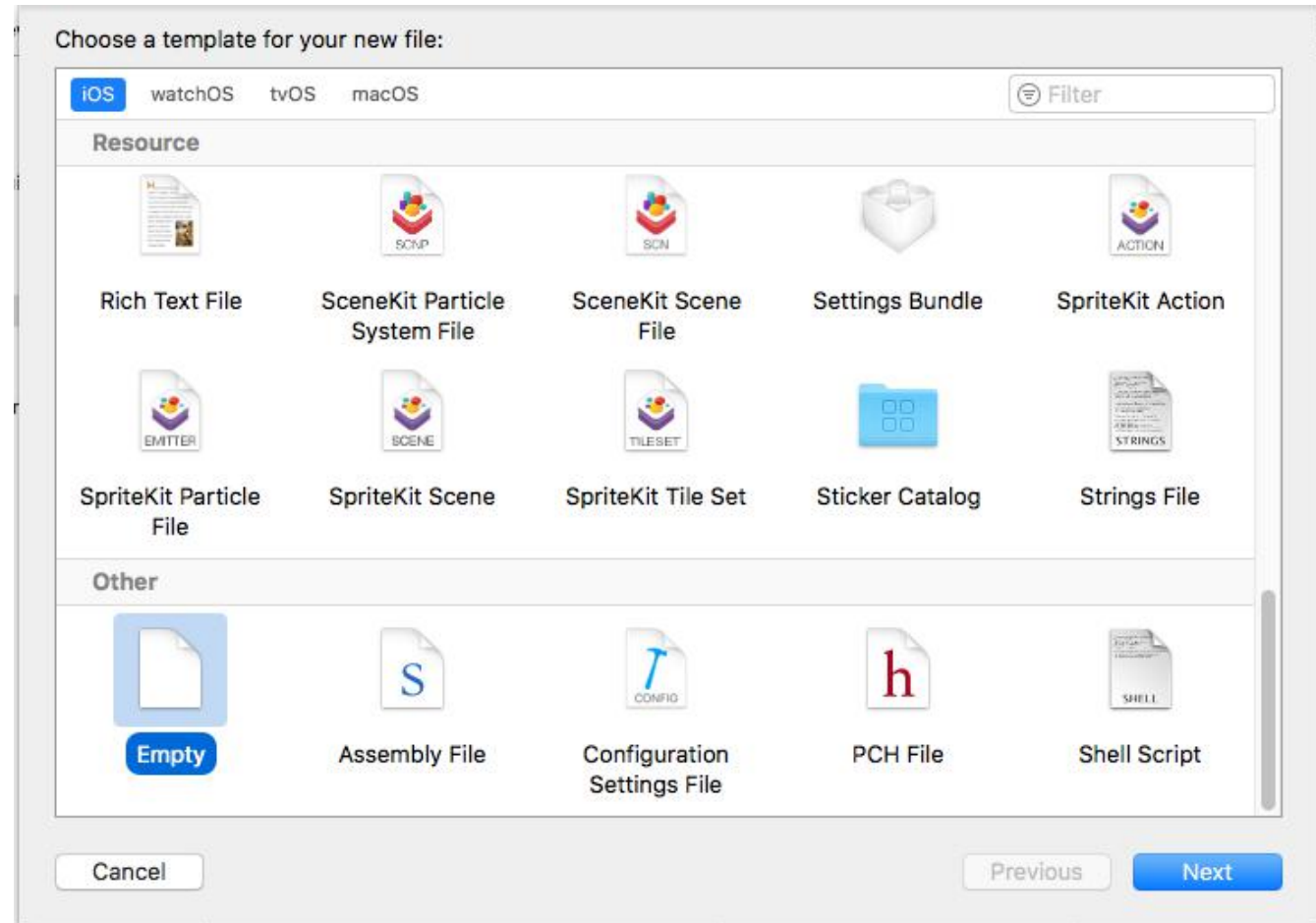
Primeiro vamos criar uma pasta chamada Resources, clicando com o botão direito sobre a pasta Hello WebView no project explorer, e selecionando “New Group”.



Carregando os arquivos Web

Agora sobre a pasta Resources, clicamos com o botão direito e criaremos um novo arquivo, em New File.

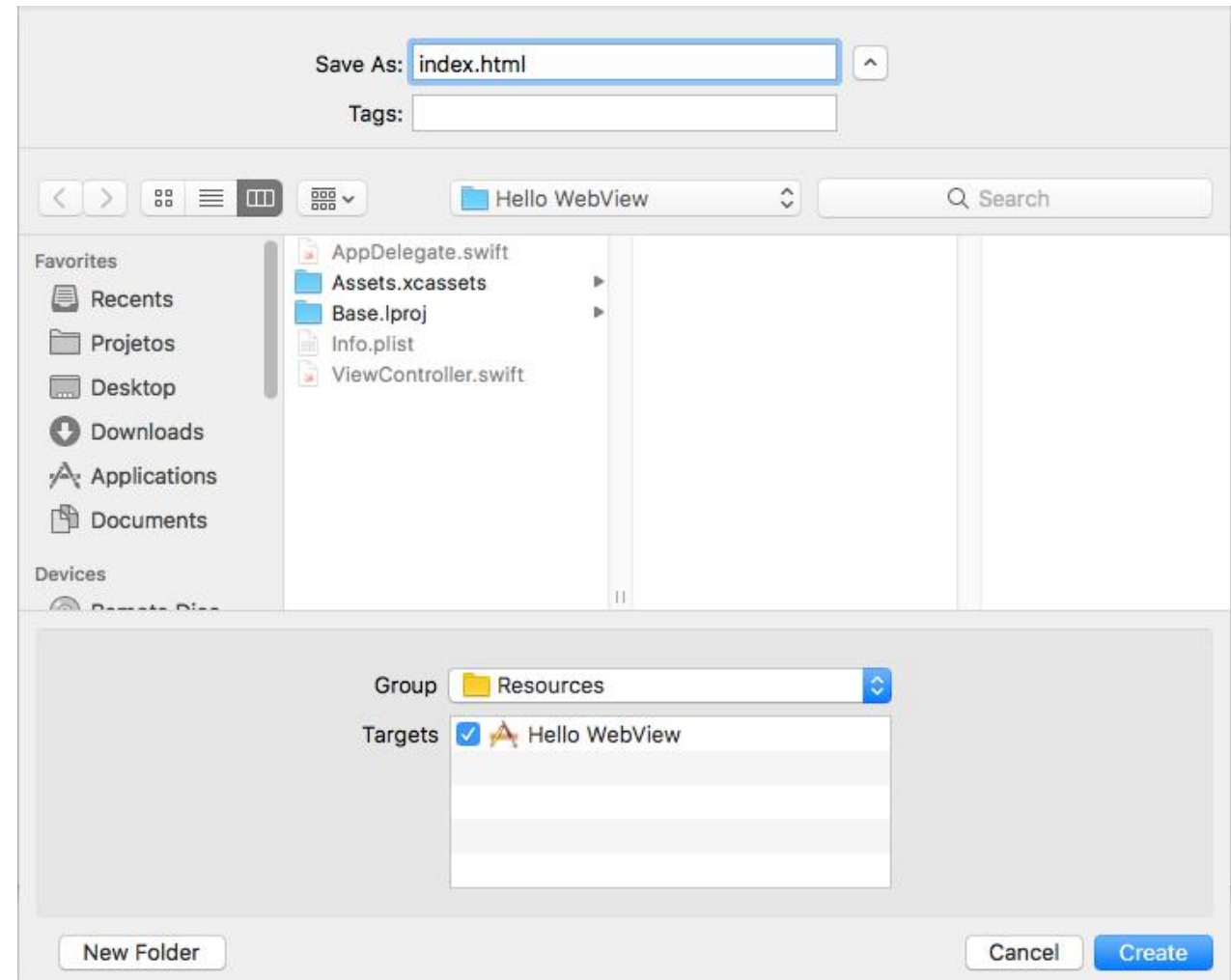
O template a ser selecionado é o Empty.



Carregando os arquivos Web

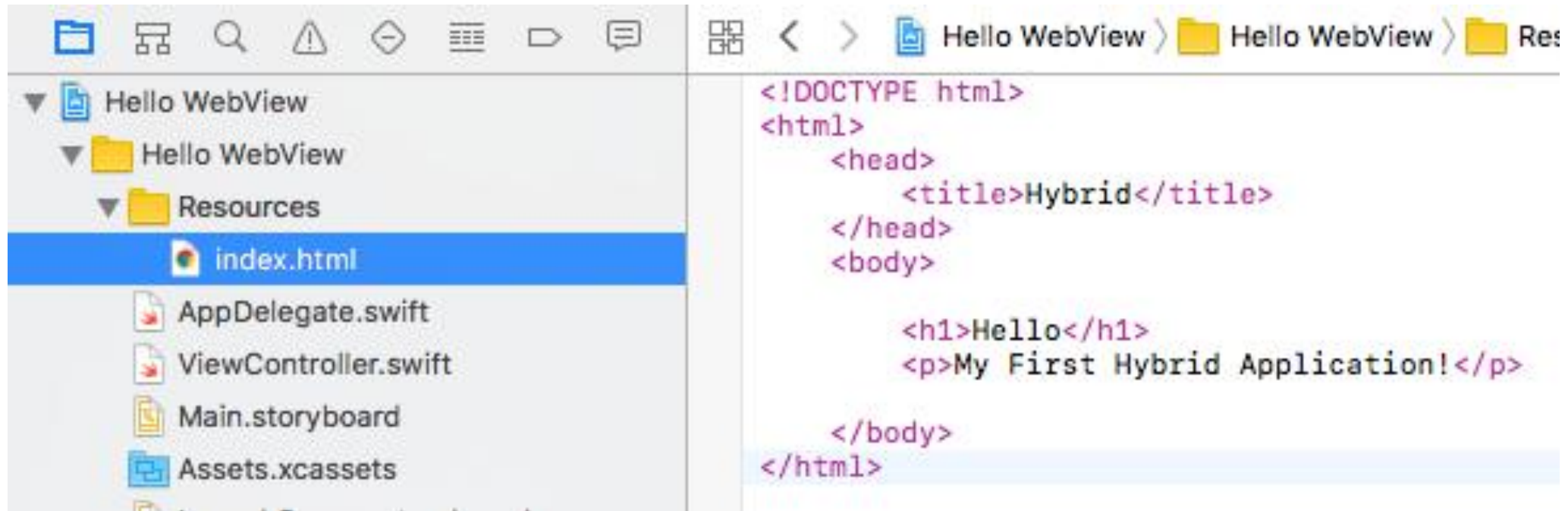
Na hora de salvar, coloque “index.html” no nome do arquivo, verifique se o Group é Resources e que o Target Hello WebView está marcado.

Por fim, pressione create.



Carregando os arquivos Web

O arquivo foi criado e está vazio, vamos adicionar um markup html simples para ser carregado na nossa webview.



Carregando o index.html na WebView

Quando um arquivo é adicionado ao projeto iOS, dizemos que ele passa a fazer parte do Bundle da aplicação.

O iOS fornece maneiras para encontrar o caminho dos arquivos que pertencem ao Bundle principal da aplicação, ou seja, ele dá uma URL para o arquivo desejado.

Para obter essa URL, vamos atualizar nosso método `viewDidLoad` da classe `ViewController`:

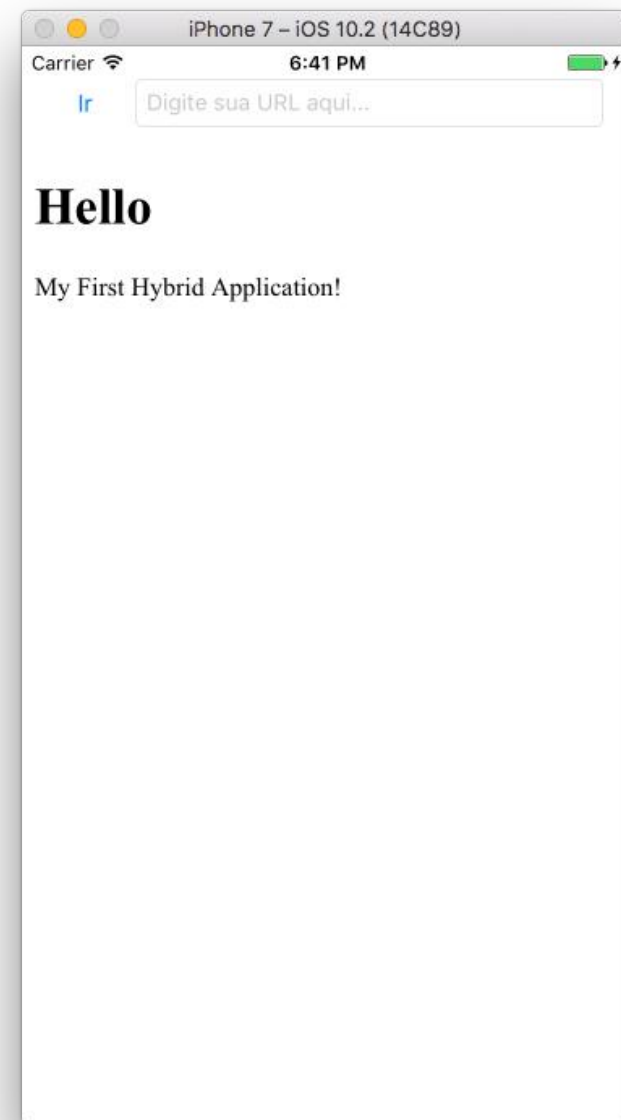
```
override func viewDidLoad() {  
    super.viewDidLoad()  
    //let url = URL(string: "https://www.facebook.com")  
  
    // IF Swift 2  
    //let url = NSBundle.mainBundle().URLForResource("privacy", withExtension:"html")  
  
    //Swift 3  
    let url = Bundle.main.url(forResource: "index", withExtension: "html")  
  
    let request = URLRequest(url: url!)  
    webView.loadRequest(request)  
}
```





Rodando o aplicativo Híbrido

Pronto, muito simples não?! Vamos rodar agora o aplicativo e testar se o HTML será carregado com sucesso na nossa aplicação.



Reaproveitando o código do módulo Web

No módulo web desenvolvemos um site capaz de enviar requests para a placa IoT para acender ou apagar o led.

Imagine se tivéssemos que reimplementar todos os requests e a view da lampada e do interruptor em cada uma das plataformas?



Exercício Final

Para fechar o módulo de iOS do Venturus 4Tech, vamos exercitar todos os conceitos que conhecemos até aqui para ter um aplicativo híbrido que seja capaz de acender e apagar o led da placa de IoT.

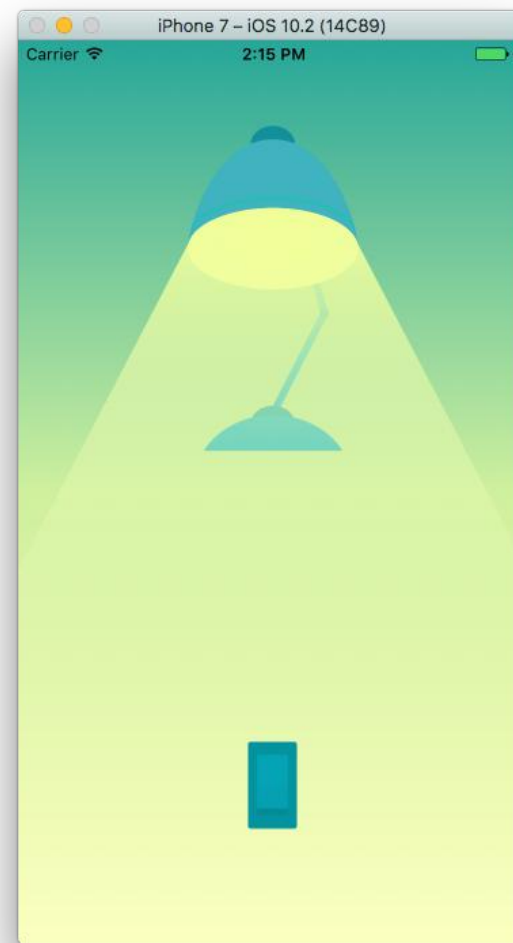
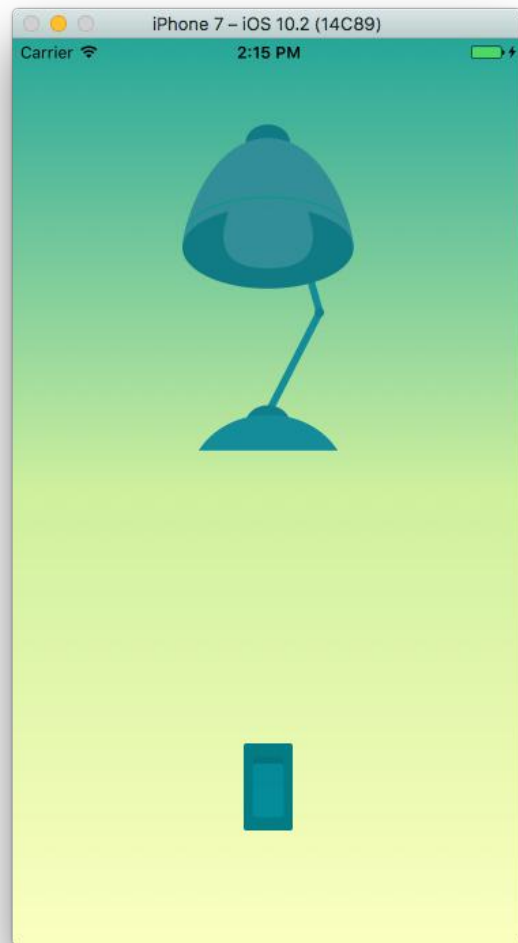
- Crie um novo projeto.
- Inclua uma WebView
- Importe os arquivos do módulo Web.
- Carregue o arquivo HTML na WebView.
- Execute no simulador e teste o envio de comandos de apagar e acender o led.





Exercício Final

O resultado deste exercício será:



Referências

- <https://developer.apple.com/reference/swift>
- <https://goo.gl/7Npp8k>
- <https://developer.apple.com/reference/uikit/uiwebview>
- <https://itunes.apple.com/br/course/desenvolvimento-ios-em-swift/id937721240>

