



Projeto de Aplicações para iOS

Bruno Marçal Lacerda Fonseca

2018

Projeto de Aplicações para iOS

Bruno Marçal Lacerda Fonseca

© Copyright do Instituto de Gestão e Tecnologia da Informação.

Todos os direitos reservados.

Sumário

Capítulo 1. Introdução	6
Cenário de aplicações para dispositivos móveis iOS	6
App Store	7
Apple Developer Program	7
Padrão MVC – Model View Controller	8
Introdução ao XCode	10
 Capítulo 2. Introdução ao Swift.....	11
História da linguagem.....	11
Hello World.....	11
Entendendo e praticando	15
Variáveis e Contantes (var x let)	20
Variáveis opcionais (operadores ! e ?)	21
Criando Classes	26
Organizando a estrutura do projeto.....	31
Construtores.....	38
Passagem de parâmetros	45
Protocolos	52
Tipo Any e AnyObject e conversão de tipos (cast).....	59
Array e Dictionary.....	61
Extensions (category).....	67
xCode.....	72
 Capítulo 3. Navegação entre telas, Storyboards, UINavigationController	76

Storyboards.....	76
Guias.....	80
Navegação entre telas	87
A Classe UINavigationController	107
Exceptions.....	116
Capítulo 4. O aplicativo carro	122
Criando o projeto.....	122
Incrementando nosso projeto.....	124
TableviewController.....	132
Cocoapods	144
TableviewController – Parte 2	150
Tela de detalhe dos carros.....	175
Usando Mapkit e UIWebView	183
Capítulo 5. Entendendo o uso de Blocos / Closures	199
Blocos / Closures	199
Usando Closures para animações	206
Capítulo 6. Persistência.....	211
Classe UserDefaults.....	211
Escrita e leitura de arquivos	217
Banco de dados SQLite	222
Criando classe utilitária para SQLite	231
Praticando com SQLite	242
Capítulo 7. APIs proprietárias.....	263

Capítulo 8. Apple TV	273
Desenvolvendo para Apple TV	274
Capítulo 9. Certificados e Provisionamentos	284
Criando Certificado	284
Criando AppID	293
Incluindo Devices	296
Criando Provisionamento	299
iTunes Connect	304
Referências	310

Capítulo 1. Introdução

O mercado de desenvolvimento de aplicativos mobile está em alta, e a demanda por bons profissionais está cada vez maior. O iOS se destaca como uma das principais plataformas de desenvolvimento mobile do mercado, pois possui muitos recursos que permitem criar aplicativos (apps) diferenciados que proporcionam uma ótima experiência ao usuário final.

Uma das principais características do iOS é a sua única e exclusiva integração com o hardware da Apple, o que garante um ótimo desempenho, sem falar que é o sistema operacional mais seguro do mundo, pois não existem vírus que atuem nesse sistema.

O iPhone foi lançado em 2007, durante o WWDC (Apple Worldwide Developers Conference), que é o evento para desenvolvedores organizado pela Apple todo ano. Mas o SDK para desenvolvimento de aplicativos foi lançado no ano seguinte, em 2008, junto com a abertura da App Store, a loja de aplicativos da Apple que facilita do processo de distribuição.

Mas o que realmente impulsionou o desenvolvimento de aplicativos foi a App Store, permitindo a fácil armazenagem, distribuição dos aplicativos e o principal, a remuneração aos desenvolvedores.

Cenário de aplicações para dispositivos móveis iOS

Primeiramente, para desenvolvermos aplicativos são necessários dois requisitos:

1. Um computador MAC (Macbook, iMac, Macmini), mas fiquem tranquilos, será possível acompanhar todos o conteúdo do curso sem a necessidade de um MAC.
2. Conta de desenvolvedor na Apple (Apple Developer Program).

App Store

A App Store é a loja de aplicativos da Apple, todos os apps estão disponíveis nessa loja para serem baixados, ela é dividida por país e categorias, ficando muito fácil encontrar o app que deseja.

Para acessar a App Store é necessário utilizar um programa da Apple chamado iTunes, que pode ser baixado diretamente do site da Apple e para sistemas MAC ou Windows. A App Store também pode ser acessada através de dispositivos iOS (iPhone, iPad etc.).

Link para baixar o iTunes: <https://www.apple.com/br/itunes/download/>

Apple Developer Program

Programa do desenvolvedor Apple, é um programa destinado aos interessados em desenvolver apps para a plataforma Apple, ou seja, iOS.

1. iOS Developer Program

Destinado a publicação de apps na App Store como pessoa física, empresa, instituição governamental ou educacional.

2. iOS Developer Enterprise Program

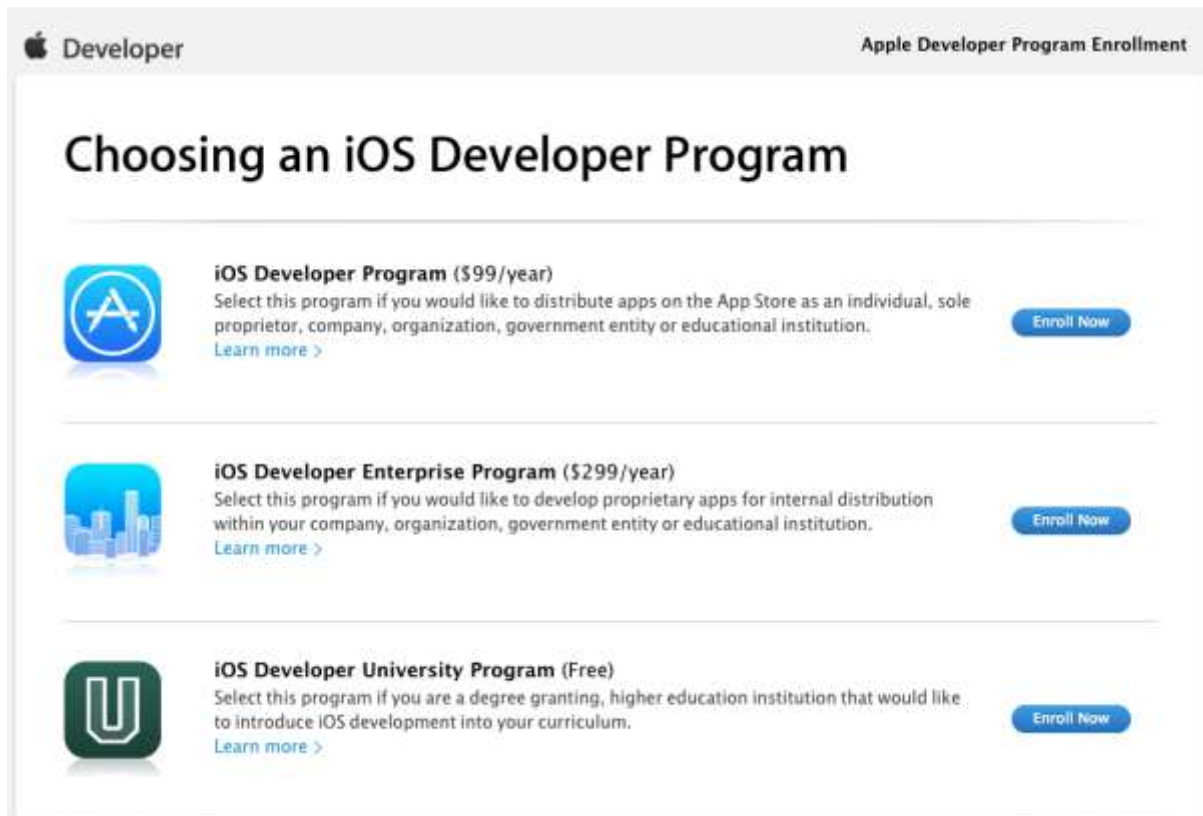
Destinado a desenvolvimento de apps proprietárias para distribuição interna a uma empresa ou instituição.

Obs.: essa opção não publica apps na App Store, seria necessário criar uma “store” para a empresa.

3. iOS Developer University Program

Destinado a instituições de ensino que desejam incluir o desenvolvimento iOS em seu programa.

Figura 1 – Programas para desenvolvedores Apple



Padrão MVC – Model View Controller

Embora a linguagem de programação utilizada para desenvolver para iOS seja o Objective C, foram criados alguns frameworks para deixar o desenvolvimento mais produtivo. O primeiro é o Foundation, que contém um conjunto de classes escritas em Objective C (NSString, NSObject, NSDate, NSNumber, NSArray etc.) para tornar o desenvolvimento mais simples e rápido.

Em cima desse conjunto de classes foi criado o Cocoa Touch que é integrado ao ambiente de desenvolvimento do XCode e facilita a utilização de diversos recursos.

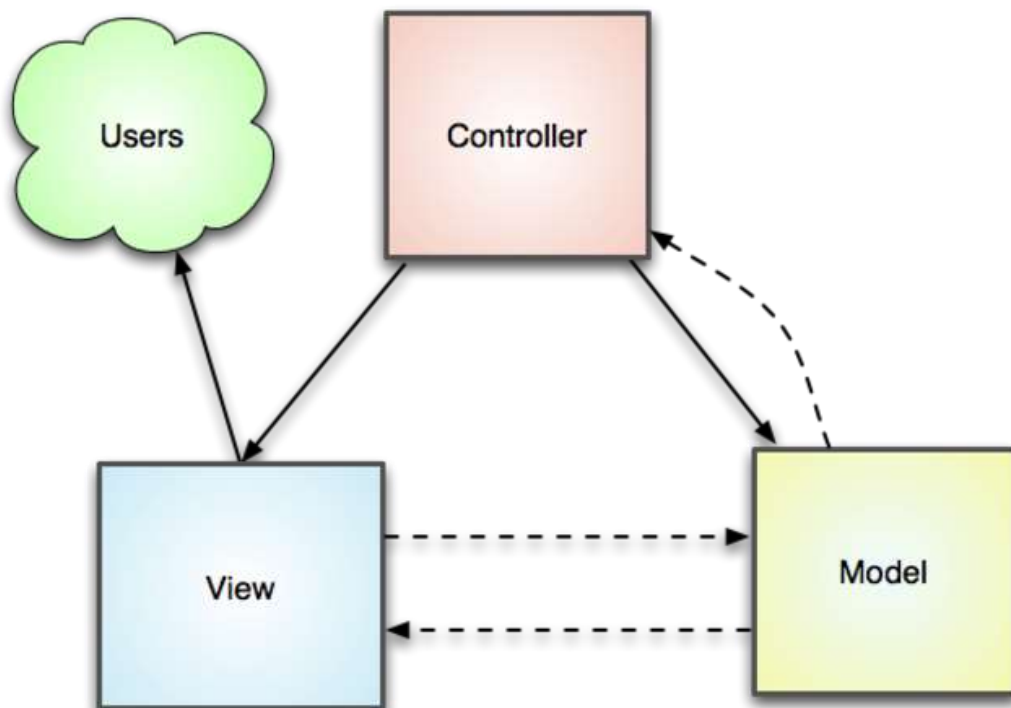
O Cocoa Touch é baseado no padrão MVC, que é uma maneira lógica de dividir o código que completa um aplicativo baseado em GUI.

O modelo MVC se divide em três categorias distintas:

- **Model (Modelo):** classes que possuem os dados de seu aplicativo, classes de persistência, classes comuns ao projeto.
- **View (Visualização):** formado de janelas, controles e outros elementos que o usuário pode interagir.
- **Controller (Controlador):** une o modelo e a visualização e é a lógica da aplicação, a regra de negócio do seu app que decide como manipular as entradas do usuário.

O modelo MVC ajuda a garantir a máxima reutilização do código. A camada de Controller são as classes filhas, como *UIViewController*, e basicamente são elas que vão definir o ciclo de vida das telas, tratar os eventos do usuário, controlar a navegação e é claro interagir com a camada Model, que possui as classes e os objetos responsáveis pela lógica de negócio da aplicação.

Figura 1.1 – Representação Modelo MVC



Introdução ao XCode

O XCode é a IDE oficial da Apple e vamos utilizá-lo durante todo o curso.

Para instalar o XCode acesse sua página oficial

<https://developer.apple.com/xcode/downloads/> ou, de um MAC, acesse o aplicativo Mac App Store (loja de aplicativos para MAC OS) e baixe o XCode.

Uma das vantagens de baixar apps da Mac App Store é a facilidade de atualizações, pois você receberá notificações sempre que existir uma nova versão disponível.

Depois de instalar é só executá-lo.

Figura 2 – Tela inicial do XCode



Capítulo 2. Introdução ao Swift

História da linguagem

Swift foi lançada na **WWDC 2014** (Apple Developers Conference) com uma sintaxe simples e moderna.

O que se nota logo de início é que Swift apresenta apenas um arquivo, de extensão **.swift**, ou seja, os arquivos **.h** e **.m** foram deixados extintos.

Também é possível que criar código em conjunto com **Objective C (Obj-C)**, pois classes do **Swift** podem chamar código escrito em **Obj-C**. Caso você tenha algum framework escrito em **Obj-C** você não precisa migrá-lo para Swift, pode simplesmente utilizá-lo em seu projeto.

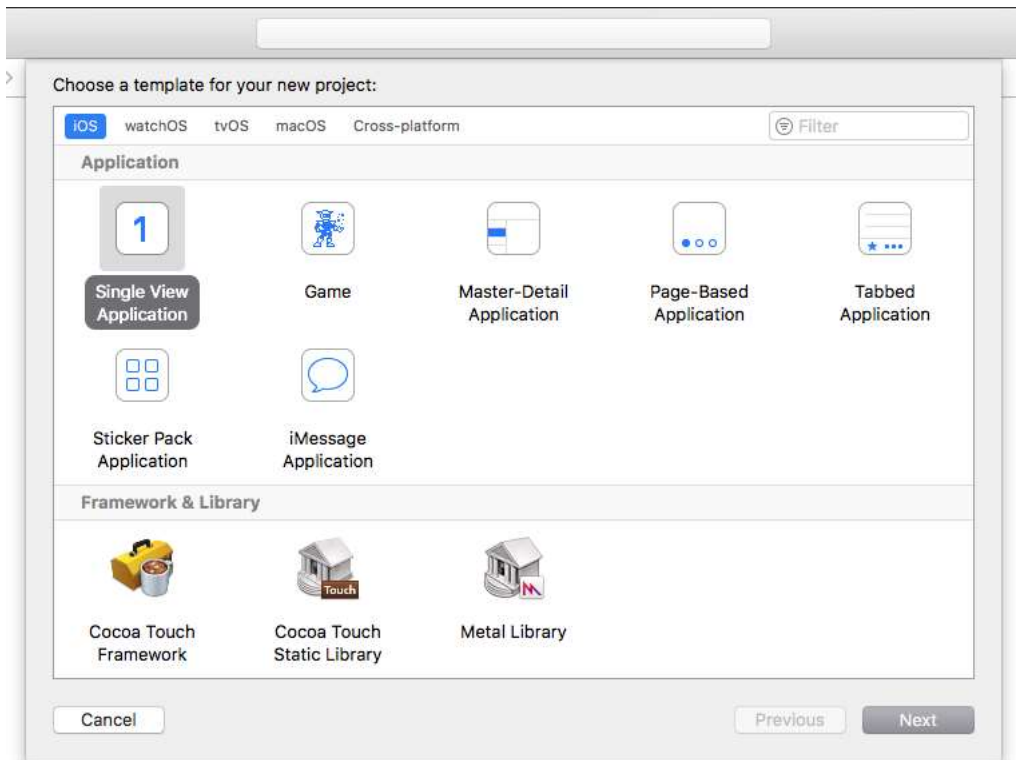
Hello World

Importante: os exemplos mais simples de código podem ser testados no endereço abaixo: https://www.tutorialspoint.com/compile_swift_online.php.

Lembrando que podem variar a versão da linguagem Swift.

Abra o XCode selecionando o **wizard File > New > Project** e selecione o projeto como mostrado na figura 3.

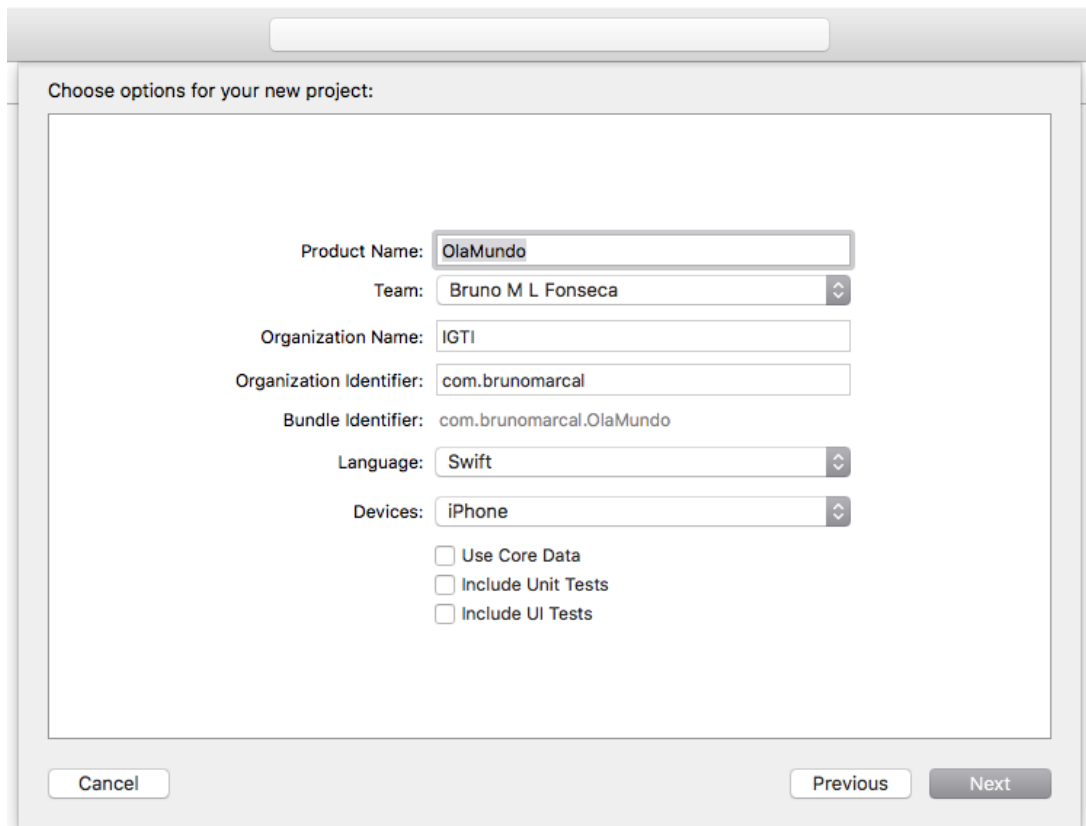
Figura 3 – Criando um projeto no XCode



Na próxima tela entre com os dados do projeto conforme figura 4.

Importante: o **Organization Identifier** é o campo que identifica sua empresa (recomendado que seja o site da empresa ao contrário) e o campo **Bundle Identifier** é gerado a partir dele, sendo o identificador único do seu app.

Figura 4 – Criando o projeto OlaMundo



Choose options for your new project:

Product Name: OlaMundo

Team: Bruno M L Fonseca

Organization Name: IGTI

Organization Identifier: com.brunomarcas

Bundle Identifier: com.brunomarcas.OlaMundo

Language: Swift

Devices: iPhone

☐ Use Core Data

☐ Include Unit Tests

☐ Include UI Tests

Cancel Previous Next

A figura 4 exibe propriedades do projeto que ficam visíveis sempre que selecionamos a raiz do projeto ou criamos um. Nesta tela podemos configurar o identificador do projeto, o tipo de projeto (iPhone, iPad ou Universal), etc.

Na tabela abaixo vemos detalhes de todas essas propriedades.

Item	Descrição
Bundle Identifier	Identificador único do app.
Team	AppleID usado para o desenvolvimento (só precisaremos disso quando formos enviar o app para o dispositivo)
Devices	Permite escolher o tipo de projeto (iPhone, iPad ou Universal)
Organization Name	Identifica a organização podendo ser: Nome da sua empresa, nome do seu cliente, pode também ser o seu próprio nome se

	estiver criando um app pessoal.
Organization Identifier	Identificador da organização (sua, do seu cliente pessoal etc.) Apple recomenda que seja o site da sua empresa / organização ao contrário. Ex.: com.apple
Bundle Identifier	Identificador único do seu app, com base no campo Organization Identifier e Product Name é criado este campo para representar um ID único para o seu app dentro da Apple e dentro do device que for instalado.
Language	Qual linguagem de programação base seu projeto terá.
Devices	Para qual device será o app, podendo ser: iPhone, iPad ou Universal.
Checkboxes	Se seu projeto vai ter projetos de testes e se usa Core Data ORM de banco de dados embarcado da Apple.

Na figura 5 podemos visualizar a estrutura do projeto, com o clássico função main para executar uma aplicação.

Figura 5 – Arquivo ViewController.swift



Entendendo e praticando

Para exemplificar melhor vamos usar um novo recurso que existe na linguagem Swift, que é o arquivo **.playground**, um arquivo para testes rápidos, vejamos abaixo, na figura 6, 7 e 8, como criá-lo.

Figura 6 – Criando arquivo .playground

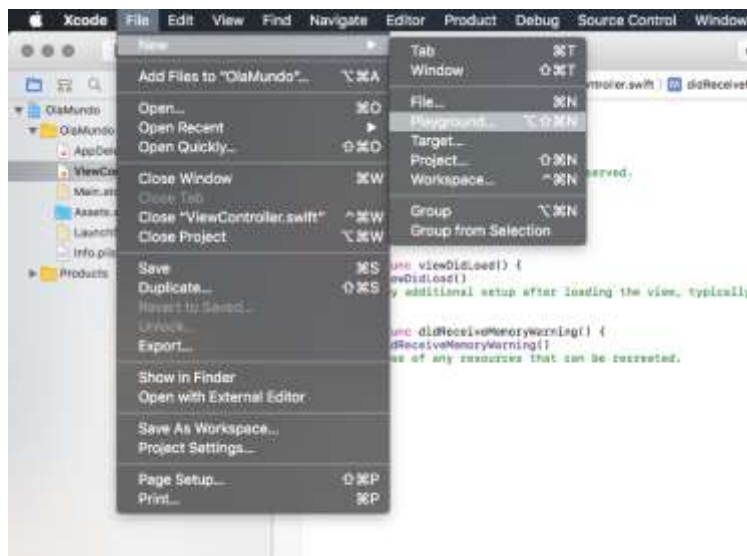


Figura 7 – Dando nome ao arquivo .playground

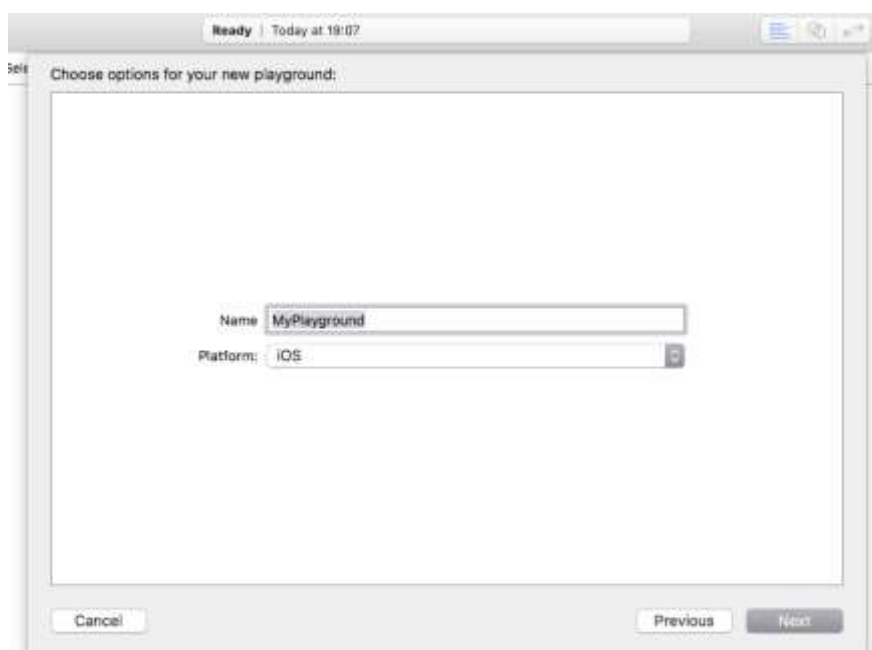
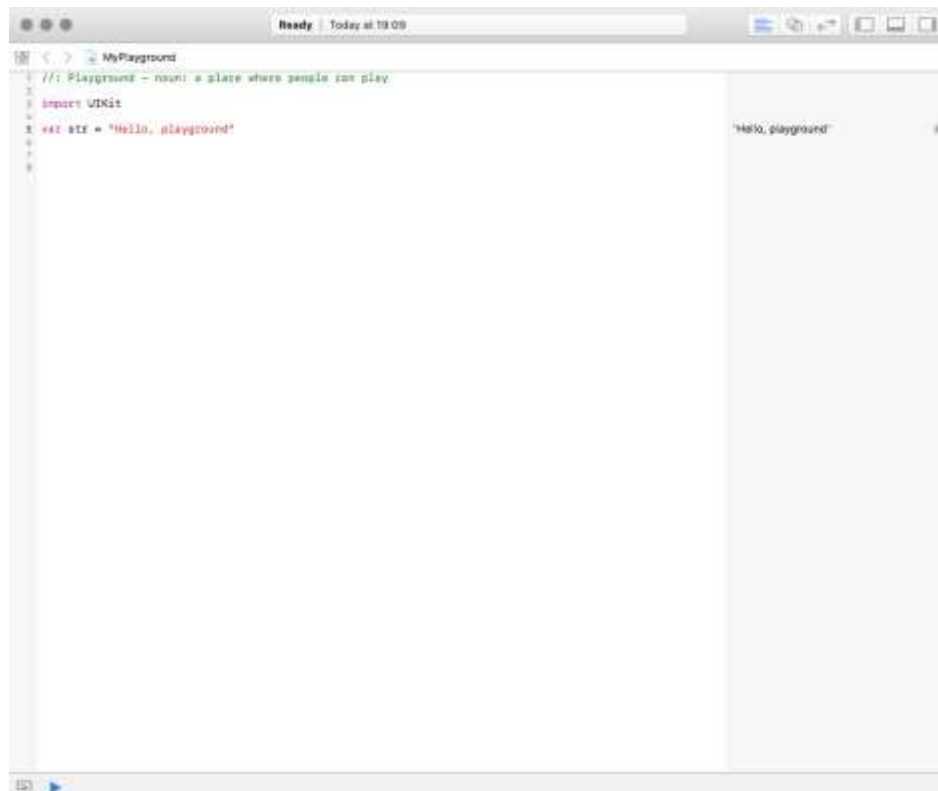


Figura 8 – Aparência do arquivo .playground



Importante: tudo que for escrito no arquivo **.playgroud** poderá ser visualizado logo na coluna ao lado como mostra a figura 9. Este tipo de arquivo serve para podermos testar implementações com mais rapidez.

Vamos explorar um pouco do arquivo **.playground** e de quebra aprender um pouco sobre os conceitos básicos de **Swift**.

Vamos criar uma função simples em **Swift** e utilizá-la.

```
func testeVar()
{
    var a = 1
```

```
var b = 2

var nome = "Bruno"

print("Olá \ \(nome), o resultado da soma é: \ \(a+b)")

}

testeVar()
```

Repare que temos três variáveis, que foram declaradas usando a palavra reservada **var** e logo atribuímos valor a essas variáveis, por último imprimimos no console uma string concatenando as variáveis e, por fim, fazendo uma soma dos valores das variáveis **a** e **b**.

Para executar simplesmente chamamos nossa variável.

Importante

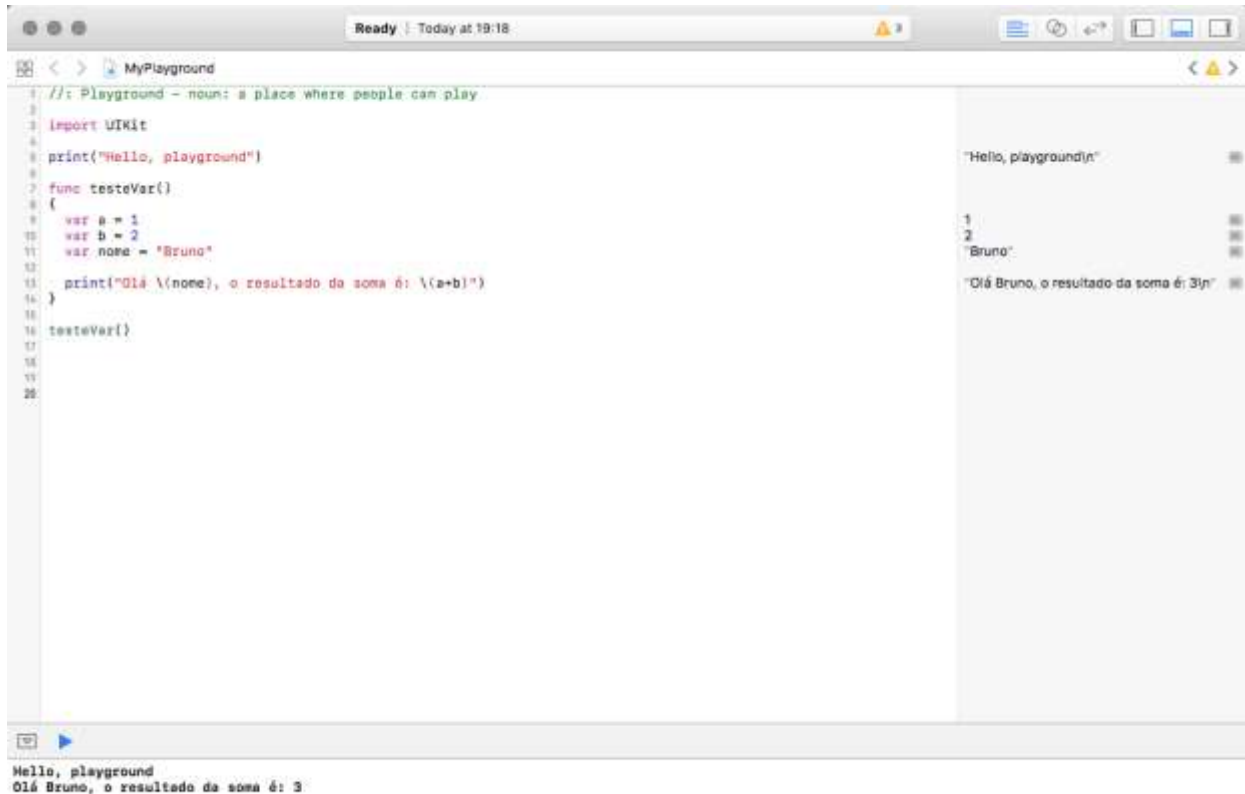
Para criarmos uma função utilizamos a sintaxe **func nomeFuncao() -> tipo retorno**, mas o tipo pode ser omitido se não for necessário, ou seja, se for **void**.

O Swift contém tipagem fraca de variáveis, ou seja, não é necessário declarar o tipo da variável, o compilador saberá qual o tipo da variável com base no seu conteúdo.

Para imprimir variáveis e até expressões deve-se usar a sintaxe **\(nome_variavel)**.

Veja como ficará nosso arquivo **.playground**.

Figura 9 – Criação da função testeVar()



Veja o exemplo do arquivo .playground mas com o tipo declarado.

```
func testeVar()
```

```
{
```

```
    var a:Int = 1
```

```
    var b:Int = 2
```

```
    var nome:String = "Bruno"
```

```
    print("Olá \(nome), o resultado da soma é: \(a+b)")
```

```
}
```

```
testeVar()
```

Enfim, podemos declarar ou não o tipo da variável mas nem sempre é bom fazer isso.

Dica: Quando se trata de objetos é melhor declaramos o tipo da variável, pois facilita a leitura do código.

Variáveis e Constantes (var x let)

No **Swift**, uma variável é declarada com a palavra reservada **var** antes do nome, e uma variável pode ser o seu valor alterado no decorrer do código.

No caso de uma variável nunca mudar como é o caso da variável **nome**, é recomendado que esta seja uma constante.

Sendo assim, nossa função ficaria como abaixo:

```
func testeVar()
{
    var a:Int = 1
    var b:Int = 2
    let nome:String = "Bruno"

    print("Olá \ \(nome), o resultado da soma é: \ \(a+b)")
}
```

```
}
```

```
testeVar()
```

Reparem que não usamos **var** na declaração da variável **nome** e sim a palavra **let**, pois é assim que o **Swift** entende que é a declaração de uma constante.

Dica: seguindo o guide de boas práticas da Apple para **Swift**, é recomendado declarar as variáveis e os objetos como constantes utilizando o **let**, pois isso permite ao compilador otimizar o código ao máximo. Utilize **var** somente se precisar alterar o valor da variável ou a instância do objeto.

Variáveis opcionais (operadores ! e ?)

O Swift possui o conceito de variáveis opcionais, que podem ter o valor nulo. Porém, para uma variável poder receber o valor nulo é preciso deixar isso explícito no código, caso contrário ocorrerá erro de compilação.

Veja a função abaixo:

```
func testeVarOpcional()
{
    let nome:String
    print("Olá \ \(nome)")
}
```

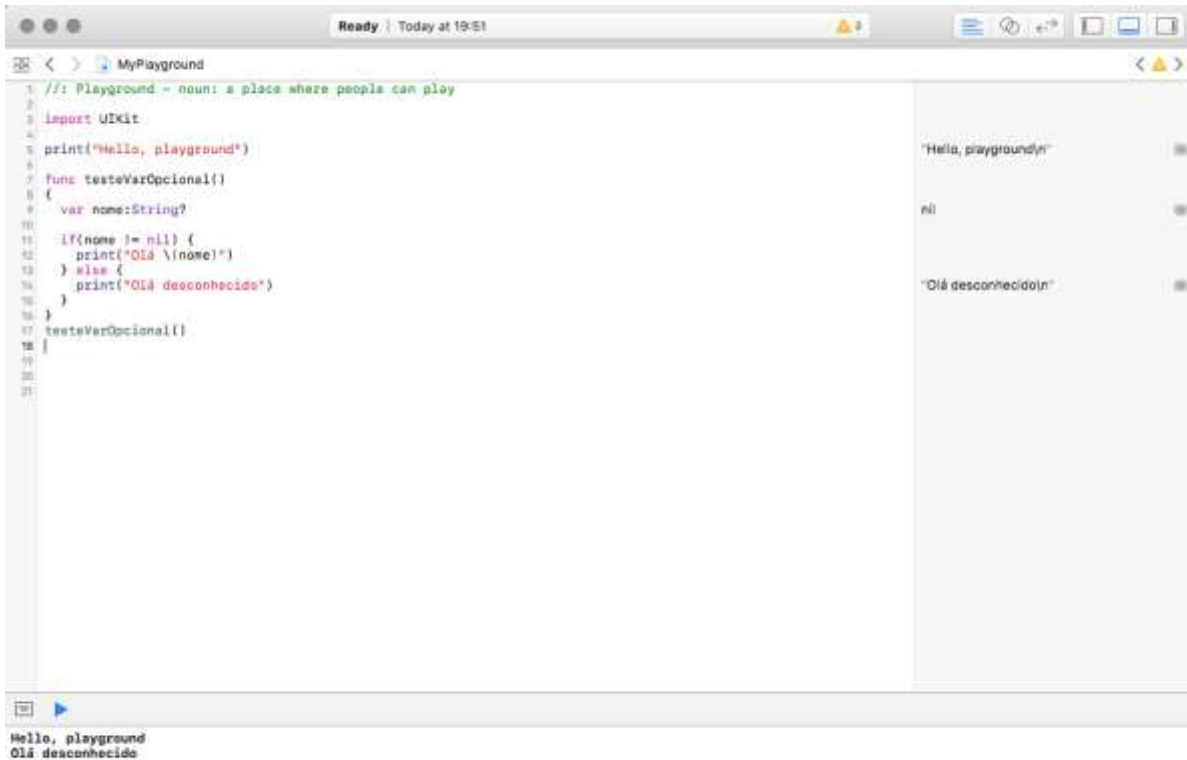
Não irá compilar, pois a variável **nome** não foi inicializada ou não foi explicitado que ela pode conter o valor nulo. Veja como ficará nossa função explicitando que a variável **nome** pode receber o valor nulo:

```
func testeVarOpcional()
{
    var nome:String?

    if(nome != nil)    {
        print("Olá \ \(nome)")
    } else {
        print("Olá desconhecido")
    }
}

testeVarOpcional()
```

Figura 10 – Utilizando variáveis opcionais



Veja o que acontece agora caso a variável **nome** seja iniciada:

```

func testeVarOpcional()
{
    var nome:String?

    nome = "Bruno"

    if(nome != nil)    {
        print("Olá \(nome)")
    }
}

```

```

    } else {

        print("Olá desconhecido")

    }

}

testeVarOpcional()

```

Figura 11 – Variável opcional sendo impressa



Observe que o texto “Bruno” não foi impresso, mas Optional(“Bruno”), que é o tipo da variável e não o valor.

Para imprimir o valor da variável opcional deve-se utilizar a sintaxe de exclamação “!” depois do nome da variável, conforme abaixo:

```

func testeVarOpcional()

{

    var nome:String?

    nome = "Bruno"

    if(nome != nil)    {

```



```

        print("Olá \(nome!)")

    } else {

        print("Olá desconhecido")

    }

}

testeVarOpcional()

```

Veja que a variável nome está sucedida de uma “!” que força o **unwrapping**, que é desempacotar a variável.

Importante:

O conceito de opcionais não existe em **C** ou **Obj-C**. O mais próximo no **Obj-C** é a capacidade de retornar nulo de um método que de outra forma retornaria um objeto, com **nil** como significando “**a ausência de um objeto válido**”. No entanto, isso só funciona para objetos – não funciona para estruturas –, tipos básicos de **C** ou valores de enumeração. Para esses tipos, os métodos **Obj-C** geralmente retornam um valor especial (como **NSNotFound**) para indicar a ausência de um valor. Essa abordagem pressupõe que o chamador do método sabe que há um valor especial para testar e lembra de verificar isso. Os opcionais de **Swift** permitem que você indique a ausência de um valor para qualquer tipo, sem a necessidade de constantes especiais.

Como muitas vezes a verificação de uma variável opcional torna-se necessária, o Swift contém uma sintaxe **if let variavel_opcional = variavel_com_valor**, que facilita essa declaração. Veja o código abaixo:

```

func testeVarOpcional()

```

```
{

    var nome:String?

    nome = "Bruno"

    if let strNome = nome    {

        print("Olá \ \(strNome)")

    } else {pl

        print("Olá desconhecido")

    }

}

testeVarOpcional()
```

A sintaxe **if let** já faz o **unwrapping** da variável dispensando o uso do **“!”**.

Importante:

Uma variável opcional é declarada com interrogação **“?”**. Para comparar se a variável está iniciada pode-se utilizar a condição **if (var != nil)** ou **if let**, e para imprimir o valor da variável opcional utiliza-se a exclamação **var!**.

Criando Classes

Agora iniciaremos a criação do nosso projeto, iremos construir um app do zero com as principais funcionalidades usadas no mercado.

Vamos começar criando uma classe chamada **Carro.swift**, que sofrerá alterações no decorrer do capítulo.

Vamos começar criando nosso projeto.

Abra o xCode e escolha a opção **criar novo projeto Xcode** (Create a new Xcode Project). Veja abaixo:

Figura 12 – Criando novo projeto no xCode



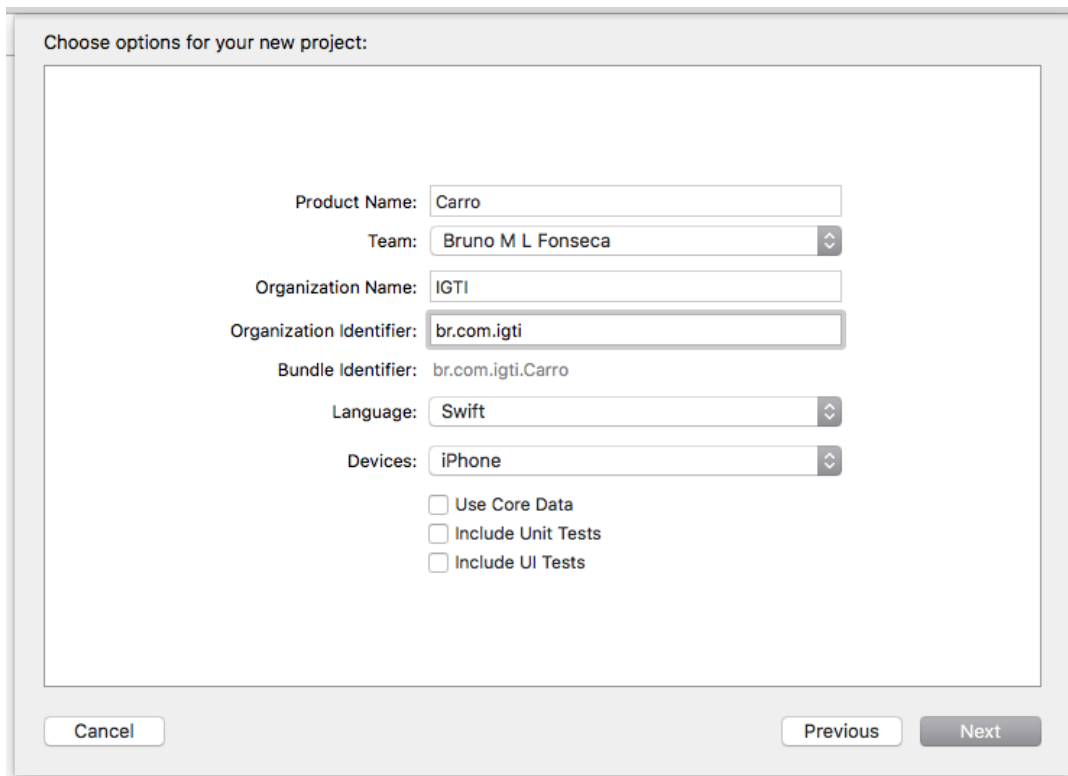
Feito isso, será apresentada a tela conforme a figura 13.

Figura 13 – Escolhendo o tipo do projeto



Feito isso, a tela da figura 14 será exibida.

Figura 14 – Detalhes do projeto



Choose options for your new project:

Product Name: Carro

Team: Bruno M L Fonseca

Organization Name: IGTI

Organization Identifier: br.com.igti

Bundle Identifier: br.com.igti.Carro

Language: Swift

Devices: iPhone

☐ Use Core Data

☐ Include Unit Tests

☐ Include UI Tests

Cancel Previous Next

A figura 14 nos mostra algumas configurações do projeto que vimos no capítulo 1. Vamos recapitular.

Product Name: Carro – é o nome do projeto.

Team: Campo onde aparecerá seu usuário vinculado à conta de desenvolvedor.

Organization Name: Nome da empresa, entidade etc. pode ser o seu nome mesmo.

Organization Identifier: Este é um campo que será usado pelo campo Bundle Identifier para gerar um identificado único para seu app, Apple recomenda que seja o site da sua empresa ou site pessoal ao contrário **ex.: br.com.igti**, desta forma garantimos que nosso app seja único.

Bundle Identifier: Com base no campo **Organization Identifier** esse campo é gerado definindo assim um identificador único para seu app. **Ex.: br.com.igti.Carro.**

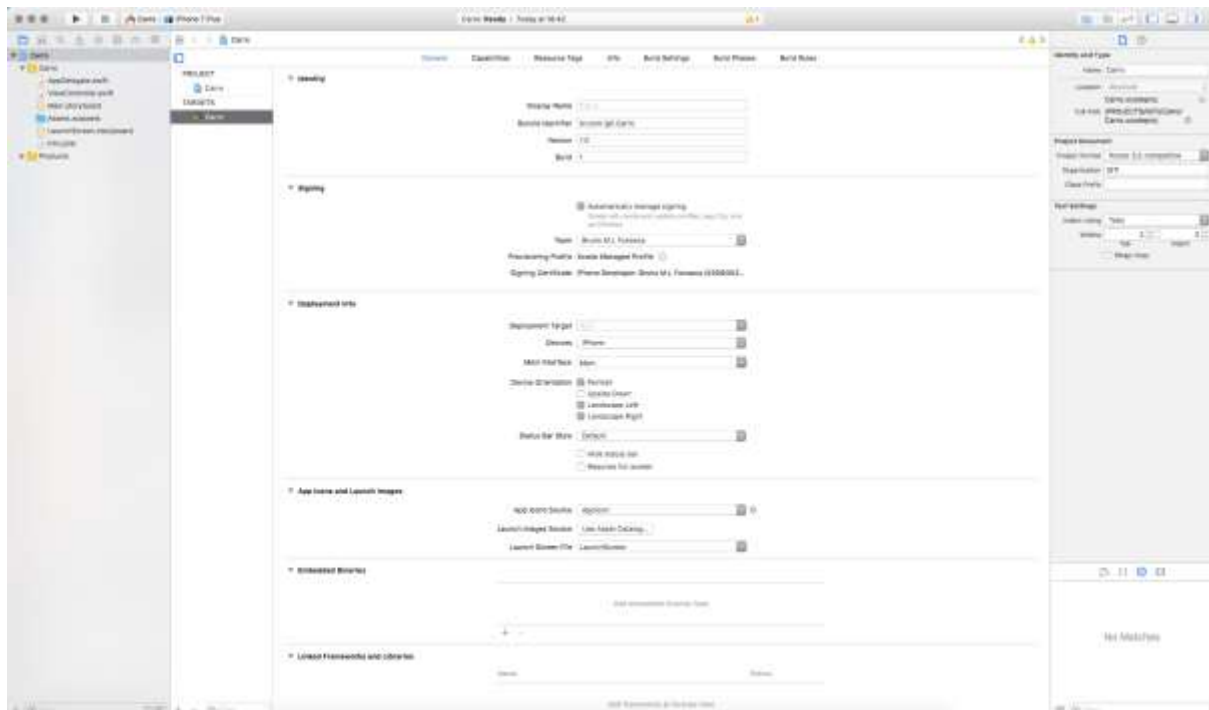
Language: Define qual a linguagem padrão do app.

Device: Define para qual device será criado o app (Universal, iPhone, iPad). Projeto que for definido como universal significa que é para iPhone e para iPad.

Checkbox: Campos que indicam se o projeto usará Core Data (ORM de gerenciamento de banco de dados embarcados), teste unitários e testes de UI (user interface).

Após a figura 14 irá aparecer mais uma tela indicando onde você deseja salvar seu projeto, é só escolher uma pasta e clicar no botão **Create** e a tela da figura 15 será exibida.

Figura 15 – Estrutura inicial do projeto Carro



A figura 15 exibe o **xCode** com toda a estrutura inicial do nosso projeto.

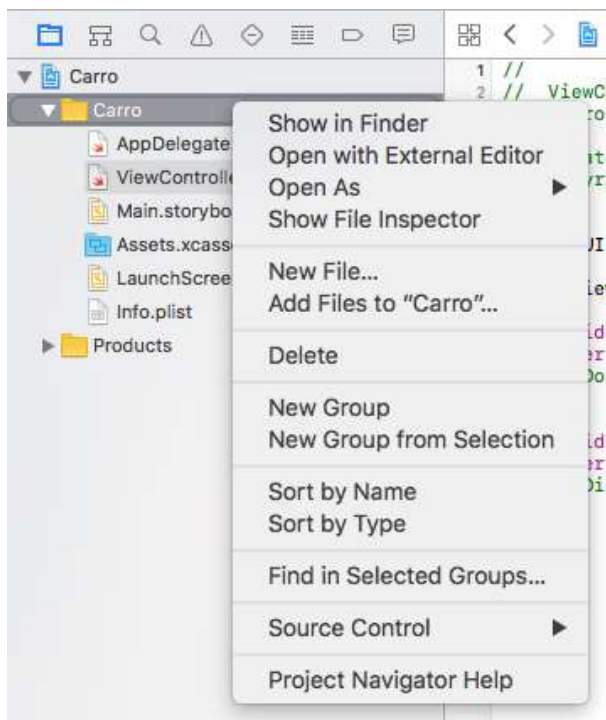
Vamos fazer uma pequena pausa para falarmos um pouco do **xCode** que é a IDE de desenvolvimento da Apple.

Organizando a estrutura do projeto

Antes de começarmos a criar nossas classes vamos organizar a estrutura de arquivos do nosso projeto, vejam que os arquivos estão soltos em apenas uma pasta, vamos então criar grupos para melhorar essa organização.

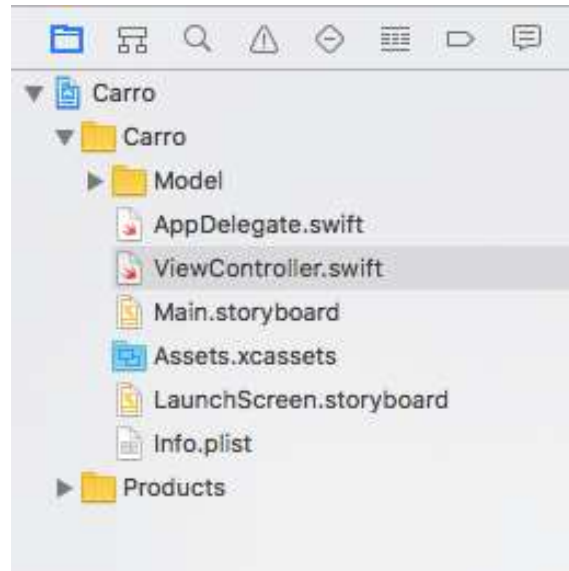
A figura 16 exibe a nossa estrutura de arquivos bem como um menu de contexto, para chegar a este menu é só clicar com o botão direito do mouse sobre a pasta Carro. Vamos então escolher a opção **New Group**.

Figura 16 – Criando grupos.



Dê ao grupo o nome de **Model**, pois é onde iremos criar nossas classe de modelo.

Figura 17 – Grupo Model criado.

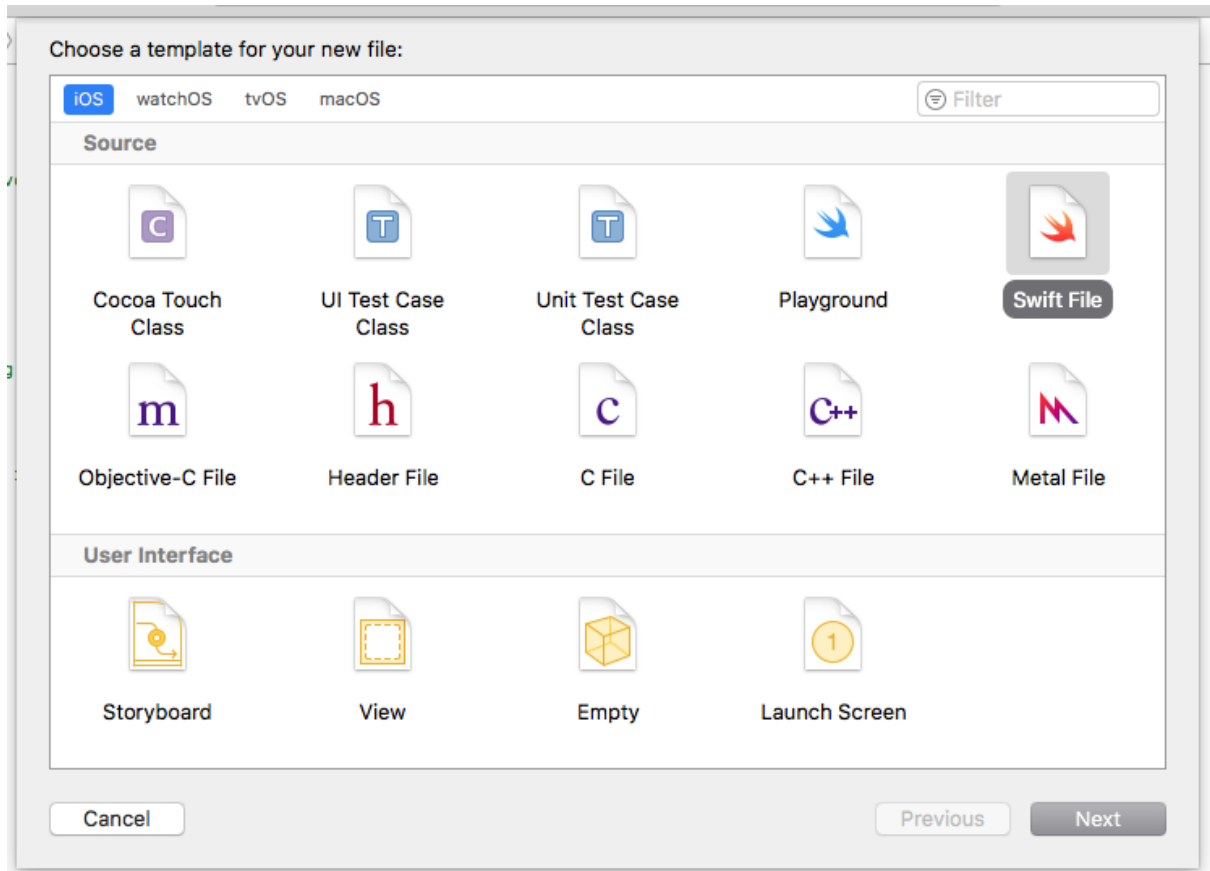


Importante: ao criar um grupo, o xCode cria uma pasta física para este grupo na estrutura do projeto.

Agora podemos criar nossa classe **Carro.swift**.

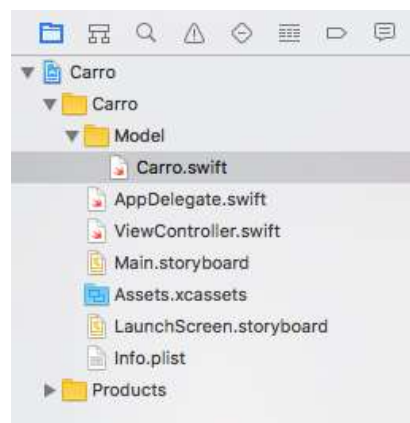
Clique com o botão direito do mouse em cima da pasta **Model** e escolha a opção **New File** (figura 16). Feito isso, escolha a opção **Swift File** exibida na figura 21.

Figura 21 – Escolha de template para criação de classe swift



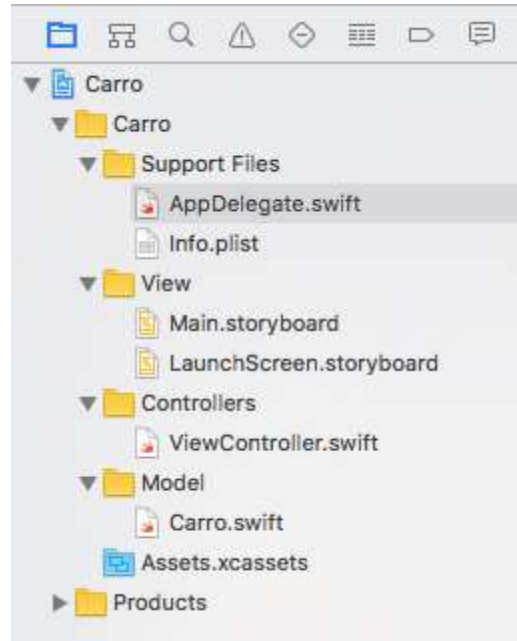
Na figura 22 podemos ver a organização estrutural do projeto até o momento.

Figura 22 – Organização do projeto até o momento



Bom, vamos organizar o restante do projeto conforme figura 23.

Figura 23 – Resultado da organização estrutural do projeto



Importante: Crie os grupos **Controllers** e **View**.

Crie um outro grupo chamado **Support Files** apenas para organização de alguns arquivos de projeto. Por fim, apenas arraste os arquivos para as respectivas pastas conforme figura 23.

Dica importante: Não arrastar o arquivo **Info.plist** (deixe-o onde está originalmente), existe um bug no xCode que quando alterar o caminho deste arquivo em específico não será alterado nas configurações do projeto, e será necessário alterar manualmente, então deixe-o onde está, Apple deverá corrigir esse bug na próxima atualização do xCode.

Agora podemos de fato criar nossa classe **Carro.swift**.

Carro.swift

```
import Foundation
```

```
class Carro {
```

```
    var nome:String?
```

```
    var ano:Int?
```

```
    init() {
```

```
        // Construtor padrão vazio
```

```
    }
```

```
}
```

Nota: no **Obj-C** toda classe é filha de **NSObject**. No **Swift**, isso é opcional, ou seja, uma classe não precisa ser filha de ninguém.

Note que as variáveis foram declaradas como opcionais com o símbolo “?”. Isso é necessário, pois não estamos inicializando a variável.

Vamos testar nossa classe `Carro.swift` recém criada, vá à classe `ViewController.swift`, esta classe é criada na concepção do projeto, então já possui todos os vínculos que precisamos com a parte visual do app (que será explicado posteriormente).

ViewController.swift

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {

        super.viewDidLoad()

        let c = Carro()

        c.nome = "Civic"

        c.ano = 2014

        print("Meu carro é \(c.nome!) e o ano é \(c.ano!)")

    }

    override func didReceiveMemoryWarning() {

        super.didReceiveMemoryWarning()

        // Dispose of any resources that can be recreated.

    }

}
```

Importante: A classe **ViewController.swift** é a primeira a ser chamada no app, pois está vinculada dessa forma, não se preocupe em saber como essa vinculação foi feita, pois faremos tudo isso posteriormente, estamos fazendo isso agora apenas para testar nossa classe **Carro.swift** recém criada já dentro da projeto /estrutura de um aplicativo.

Essa classe possui dois métodos **viewDidLoad()** e **didReceiveMemoryWarning()**. Veremos em detalhes esses métodos posteriormente.

Não precisamos do **import** no **Swift** quando queremos usar uma classe que criamos, pois **Swift** já automaticamente lê as classes do projeto e já podemos usá-las direito, assim como Java.

Dentro do método **viewDidLoad()**, inclua o código que é exibido acima.

Nós instanciamos a classe **Carro.swift** para uma variável **c**, veja que usamos **let** pois um objeto não irá mudar e sim suas propriedades e métodos.

Depois de instanciar setamos as variáveis **nome** e **ano** que criamos.

Em seguida imprimimos, com o método **print()**, a string “**Meu carro é xxxx e o ano é xxxx**”

Concatemos à string acima as propriedades **nome** e **ano**.

Para testarmos nosso projeto precisaremos compilá-lo, e isso pode ser feito clicando nos botões **command+R**.

O resultado abaixo será exibido.

Meu carro é Civic e o ano é 2014

Construtores

O construtor padrão da classe é o **init()**, mas, caso precise, é possível criar outros construtores.

Seguindo boas práticas de orientação a objetos, um construtor deve ser utilizado para garantir que um objeto seja inicializado corretamente. Portanto, como sabemos, todo carro tem um nome e um ano de fabricação, vamos pegar isso como exemplo.

Carro.swift

```
class Carro {  
  
    var nome:String  
  
    var ano:Int  
  
    init(nome:String, ano:Int) {  
        self.nome = nome  
        self.ano = ano  
    }  
}
```

No código acima, note que modificamos o construtor da classe para já iniciar as variáveis **nome** e **ano**.

Importante: o **self** do **Swift** é igual ao **this** do **java**, ou seja, se refere à própria classe.

Veja abaixo como ficará a classe **ViewController** para testarmos nossa classe **carro** modificada.

ViewController.swift

```
class ViewController: UIViewController {  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        let c = Carro(nome: "Civic", ano: 2014)  
        print("Meu carro é \(c.nome) e o ano é \(c.ano)")  
    }  
  
    override func didReceiveMemoryWarning() {  
        super.didReceiveMemoryWarning()  
  
        // Dispose of any resources that can be recreated.  
    }  
}
```

}

Veja que no momento de criação da instância da classe já passamos os parâmetros **nome** e **ano**.

O resultado abaixo será exibido:

Meu carro é Civic e o ano é 2014

No **Swift** também existe um construtor chamado de **Construtor de Conveniência**. Ele é um construtor auxiliar, que possui menos parâmetros para preencher os atributos com valores padrão.

Para criar um construtor de conveniência basta utilizar a palavra reservada **convenience init** e obrigatoriamente você irá delegar a criação do objeto a um construtor completo, ou seja, que não é de conveniência. Veja abaixo:

Carro.swift

```
class Carro {  
  
    var nome:String  
  
    var ano:Int  
  
    init(nome:String, ano:Int) {  
        self.nome = nome  
    }  
}
```



```
        self.ano = ano
    }

    convenience init(nome:String) {

        self.init(nome:nome, ano:2013)
    }
}
```

Note como ficará a classe **Carro** após a criação do construtor de conveniência, repare que para sua criação utilizamos a palavra **convenience**. Então na classe **Carro** temos o construtor padrão que preenche todos os atributos e o construtor de conveniência que inicializa somente o parâmetro **nome** e preenche o **ano** com um valor default.

Veja abaixo a classe **ViewController** para o teste desse construtor.

ViewController.swift

```
class ViewController: UIViewController {

    override func viewDidLoad() {

        super.viewDidLoad()
    }
}
```

```
let c = Carro(nome: "Civic")
```

```
print("Meu carro é \(c.nome) e o ano é \(c.ano)")
```

```
}
```

```
override func didReceiveMemoryWarning() {
```

```
    super.didReceiveMemoryWarning()
```

```
    // Dispose of any resources that can be recreated.
```

```
}
```

```
}
```

O resultado abaixo será exibido:

Meu carro é Civic e o ano é 2013

A partir do Swift 1.1 um construtor pode retornar valor nulo – isso é conhecido como **Failable Initializers**. Para criar esse tipo de construtor basta utilizar uma interrogação depois da palavra **init**, dessa forma: **init?**

Carro.swift

```
class Carro {
```

```
var nome:String
```

```
var ano:Int
```

```
init?(nome:String, ano:Int) {
```

```
    if(nome == "CarroNaoExiste") {
```

```
        return nil
```

```
    }
```

```
    self.nome = nome
```

```
    self.ano = ano
```

```
}
```

```
}
```

ViewController.swift

```
class ViewController: UIViewController {
```

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

```
        super.viewDidLoad()
```

```
var c = Carro(nome: "CarroNaoExiste", ano: 1935)
```

```
print("Carro A: \(c)")
```

```
if (c != nil) {
```

```
    print("Carro B: \(c!.nome), ano: \(c!.ano)")
```

```
}
```

```
c = Carro(nome: "Civic", ano: 2014)
```

```
print("Carro C: \(c)")
```

```
if (c != nil) {
```

```
    print("Carro D: \(c!.nome), ano: \(c!.ano)")
```

```
}
```

```
}
```

```
override func didReceiveMemoryWarning() {
```

```
    super.didReceiveMemoryWarning()
```

```
    // Dispose of any resources that can be recreated.
```

```
}
```

```
}
```

O resultado será:

Carro A: nil

Carro C: Optional(Carro.Carro)

Carro D: Civic, ano: 2014

Note que o primeiro carro “**A**” foi criado mas retornou nulo, o segundo carro “**B**” não foi criado pelo fato do carro “**A**” ser nulo, o terceiro carro “**C**” foi criado com sucesso, mas repare que ao utilizar o construtor **init?** o objeto retornado é **opcional**, portanto é preciso utilizar a sintaxe **c!** para acessar o objeto (unwrapping) ou utilizar a sintaxe **if let** conforme vimos anteriormente.

O quarto carro “**D**” foi criado e impresso com sucesso.

Passagem de parâmetros

Criar um método e passar parâmetros no **Swift** são atividades simples, e a sintaxe é parecida com a utilizada no construtor. Vamos praticar, veja a classe **Carro** abaixo:

Carro.swift

```
class Carro {  
  
    var nome:String  
  
    var ano:Int  
  
    init(nome:String, ano:Int) {  
  
        self.nome = nome  
    }  
}
```

```
self.ano = ano
```

```
}
```

```
func acelerarComVelocidade(velocidade: Int) {
```

```
    print("Acelerar para \ \(velocidade)")
```

```
}
```

```
func acelerarComVelocidade(velocidade: Int, distancia: Int){
```

```
    print("Acelerar para \ \(velocidade) km/h e Distância: \ \(distancia) metros")
```

```
}
```

```
}
```

Note que voltamos a classe com o construtor simples apenas e que adicionamos dois métodos de mesmo nome, um recebe um parâmetro e o outro recebe dois parâmetros.

Vamos agora alterar o classe **ViewController** para utilizar nossa nova classe **Carro**.

ViewController.swift

```
class ViewController: UIViewController {
```

```
override func viewDidLoad() {  
  
    super.viewDidLoad()  
  
  
    let c = Carro(nome: "Civic", ano: 2014)  
  
    c.acelerarComVelocidade(velocidade: 100)  
  
    c.acelerarComVelocidade(velocidade: 100, distancia: 500)  
  
}  
  
  
override func didReceiveMemoryWarning() {  
  
    super.didReceiveMemoryWarning()  
  
}  
  
}
```

Veja que criamos o objeto carro na variável **c** e chamamos seus dois métodos passando os parâmetros necessários.

O resultado será:

Acelerar para 100

Acelerar para 100 km/h e Distância: 500 metros

Importante: para passar parâmetros é necessário utilizar a sintaxe **parametro:valor**, mas é bem comum, em **Swift**, omitirmos a declaração do primeiro argumento utilizando um **underline** “_”.

Veja abaixo as classes alteradas.

Carro.swift

```
class Carro {  
  
    var nome:String  
  
    var ano:Int  
  
    init(nome:String, ano:Int) {  
  
        self.nome = nome  
  
        self.ano = ano  
  
    }  
  
    func acelerarComVelocidade(_ Int) {  
  
        print("Acelerar para \\\(velocidade)")  
  
    }  
  
    func acelerarComVelocidade(_: Int, distancia: Int){  
  
        print("Acelerar para \\\(velocidade) km/h e Distância: \\\(distancia)  
metros")  
  
    }  
}
```



```
}
```

ViewController.swift

```
class ViewController: UIViewController {  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        let c = Carro(nome: "Civic", ano: 2014)  
        c.acelerarComVelocidade(100)  
        c.acelerarComVelocidade(100, distancia: 500)  
    }  
  
    override func didReceiveMemoryWarning() {  
        super.didReceiveMemoryWarning()  
        // Dispose of any resources that can be recreated.  
    }  
}
```

Embora a sintaxe do underline “_” possa ser utilizada em todos os parâmetros, é mais comum que seja apenas no primeiro, pois o nome do método já indica o nome do parâmetro.

Isso se dá para manter uma certa coerência com **Obj-C** que sempre foi recomendado que o nome do método tivesse o nome do primeiro parâmetro, exatamente como **acelerarComVelocidade()**, (**Velocidade** seria o nome do primeiro parâmetro), mas em **Swift** podemos manter essa forma de criação de nomes para métodos ou não, isso fica a cargo do desenvolvedor. Logo você pode omitir ou não os nomes dos parâmetros.

Dica: não omita o nome dos parâmetros, pois isso dificulta mais o entendimento do código.

Muitas vezes um método pode retornar um valor e para isso a sintaxe é:

```
func nome_funcao(param1:tipo, param2:tipo, ...) -> retorno
```

Vamos adicionar na classe Carro um método **getDescription()** que retornará uma string.

Carro.swift

```
class Carro {  
  
    ...  
  
    func getDescription() -> String {
```

```

        let s = "Carro: \(self.nome), ano: \(self.ano)"

        return s
    }
}

```

Veja como chamar o método **getDescription()**, que retorna uma string.

ViewController.swift

```

class ViewController: UIViewController {

    override func viewDidLoad() {

        super.viewDidLoad()

        let c = Carro(nome: "Civic", ano: 2014)

        c.acelerarComVelocidade(150)

        c.acelerarComVelocidade(150, distancia: 1000)

        let desc = c.getDescription()

        print(desc)
    }
}

```

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

O resultado será:

Acelerar para 150

Acelerar para 150 km/h e Distância: 1000 metros

Carro: Civic, ano: 2014

Protocolos

Protocolo é uma maneira de estender a funcionalidade de uma classe, pode ser pensado como um escopo ou interface que define um conjunto de propriedades ou métodos. Muitos desenvolvedores que utilizam orientação a objetos estão familiarizados com o conceito de interfaces, bem protocolos em **Swift** e **Obj-C** são exatamente isso.

Obs.: os métodos e propriedades de um protocolo podem ser obrigatórios ou opcionais.

Para criar protocolos em **Swift** é utilizada a palavra reservada **Protocol**.

Vamos praticar, crie uma nova classe **Swift** dentro da pasta **Model** clique com o botão direito do mouse sobre a pasta e vá em **New File**, escolha a opção **Swift File**, de um nome para a classe que será **Combustivel**, como figura 24

Figura 24 – Criação da classe Combustivel.swift

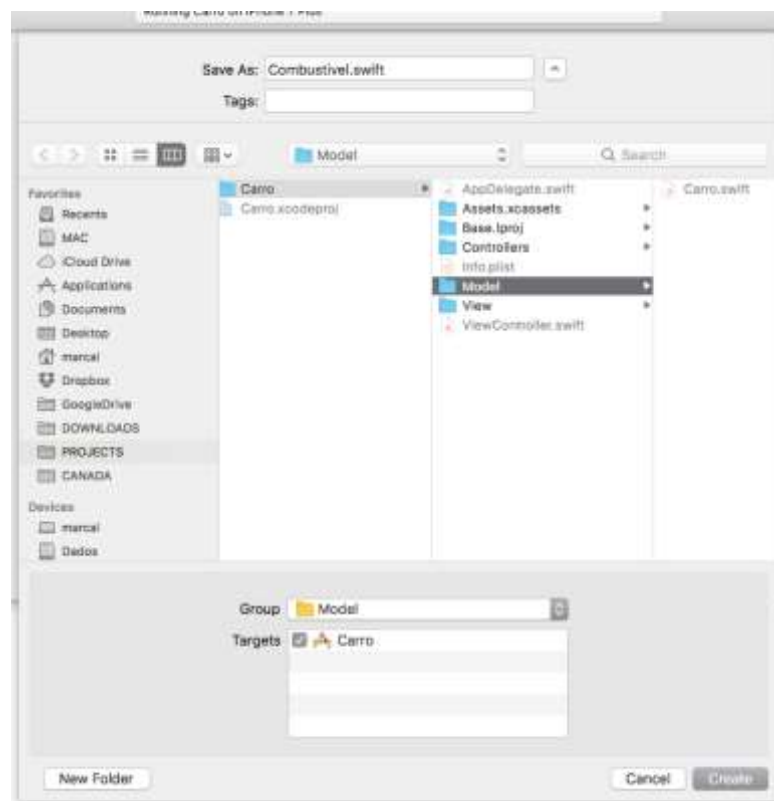
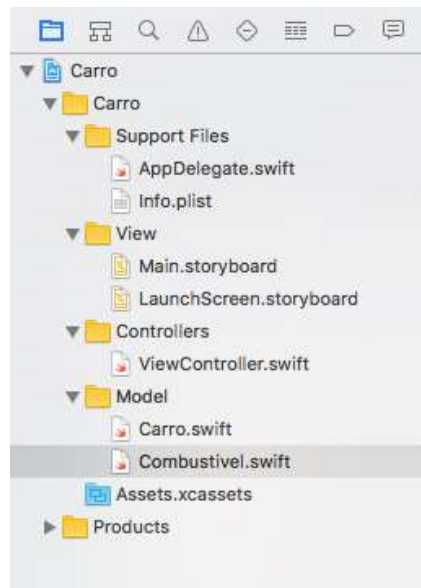


Figura 25 – Estrutura com a criação da classe Combustivel



Agora vamos começar a criação do nosso protocolo.

Combustivel.swift

```
import Foundation
```

```
protocol CombustivelAlcool {  
  
    func abastecerComAlcool()  
  
}
```

```
protocol CombustivelGasolina {  
  
    func abastecerComGasolina()
```

}

```
protocol CombustivelFlex: CombustivelAlcool, CombustivelGasolina {
```

}

```
class PostoDeGasolina {
```

```
    class func abastecerCarroComAlcool(_ tipo:CombustivelAlcool) {
```

```
        tipo.abastecerComAlcool()
```

```
    }
```

```
    class func abastecerCarroComGasolina(_ tipo:CombustivelGasolina) {
```

```
        tipo.abastecerComGasolina()
```

```
    }
```

```
}
```

Dica: note que os métodos **abastecerCarroComAlcool()** e **abastecerCarroComGasolina()** possuem a palavra reservada **class** antes da palavra **func**, isso indica que são métodos de classe ou estáticos.

Esta é a nossa classe **Combustivel**, feito isso agora precisamos alterar nossa classe **Carro** para que use nosso protocolo **Combustivel** recém criado.

Sintaxes:

Sintaxe para estender uma classe mãe:

```
class Carro : ClasseMae {
```

Sintaxe para criar um protocolo:

```
class Carro : Protocolo {
```

Notem que é a mesma sintaxe; para sucessivos protocolos somente separe-os por vírgula.

```
class Carro : ClasseMae, Protocolo1, Protocolo2 {
```

ou

```
class Carro : Protocolo1, Protocolo2 {
```

Importante: a regra é: caso precise estender uma classe mãe, esta deve vir antes do primeiro protocolo.

A seguir temos a classe Carro já com a implementação dos protocolos recém criados.

Carro.swift

```
class Carro : CombustivelFlex {
```

```
...
```

```
// MARK: - Protocolo CombustivelFlex
```



```
func abastecerComAlcool() {  
  
    print("Abastecendo com álcool")  
  
}  
  
func abastecerComGasolina() {  
  
    print("Abastecendo com gasolina")  
  
}  
  
}
```

Agora vamos alterar a classe ViewController para podermos testar.

ViewController.swift

```
class ViewController: UIViewController {  
  
  
    override func viewDidLoad() {  
  
        super.viewDidLoad()  
  
  
        let c = Carro(nome: "Civic", ano: 2014)  
  
        print("Carro: \(c.nome), ano: \(c.ano)")  
  
    }  
  
}
```

```
c.acelerarComVelocidade(190, distancia: 650)

PostoDeGasolina.abastecerCarroComAlcool(c)

PostoDeGasolina.abastecerCarroComGasolina(c)

}

override func didReceiveMemoryWarning() {

    super.didReceiveMemoryWarning()

    // Dispose of any resources that can be recreated.

}

}
```

Note que a sintaxe de **underline** foi utilizada no método **PostoDeGasolina()**, por isso não foi necessário passar o nome do primeiro argumento no método **abastecerCarroComGasolina()**, note também que o parâmetro passado para esse método é um método, que é nosso protocolo em si.

O retorno será:

Carro: Civic, ano: 2014

Acelerar para 190 km/h e Distância: 650 metros

Abastecendo com álcool

Abastecendo com gasolina

Esta é a forma mais simples para explicação e exemplificação de protocolos, sugiro que faça mais alguns testes para entender melhor o conceito.

Obs.: muitas classes nativas que usaremos no desenvolvimento de nosso app possuem protocolos, veremos no decorrer dos capítulos.

Tipo Any e AnyObject e conversão de tipos (cast)

Swift também possui os tipos de dados genéricos que são os mesmo em todas as linguagens: **Int, Float, Double, String, Bool**.

Como todos os tipos em **Swift** são tratados como objetos, podemos tratar todos da mesma forma.

O **Swift** também possui tipos especiais chamados **Any** e **AnyObject**.

O tipo **Any** pode representar qualquer tipo enquanto **AnyObject** pode representar qualquer classe ou estrutura.

A razão pela introdução de **Any** e **AnyObject** é o fato de ser compatível com o framework **Cocoa**, que é escrito em **C** e **Obj-C**. Em **Obj-C**, o tipo **id** representa qualquer tipo e para certificar que **Swift** e os frameworks **cocoa** possam trabalhar juntos foi introduzido os tipos **Any** e **AnyObject**.

Para exemplificar, veja o código abaixo.

ViewController.swift

```
class ViewController: UIViewController {  
  
    override func viewDidLoad() {  
  
        super.viewDidLoad()  
    }  
}
```

```

        testeAnyObject()

    }

    override func didReceiveMemoryWarning() {

        super.didReceiveMemoryWarning()

        // Dispose of any resources that can be recreated.

    }

    func testeAnyObject() {

        let qualquer:AnyObject = Carro(nome: "Civic", ano: 2014)

        let c = qualquer as! Carro

        print("Meu carro é um \(c.nome) e o ano dele é \(c.ano)")

    }

}

```

Veja que criamos um método ou função chamado **testeAnyObject()** e o chamamos no método **viewDidLoad()**.

Nosso método cria uma constante de nome **qualquer** do tipo **AnyObject** e logo em seguida criamos outra constante **c** que recebe o **cast** da constante **qualquer**, utilizamos o operador **as!** pois temos certeza que o nosso objeto é uma instância da classe **Carro**.

No **Swift 1.0** existia apenas uma palavra-chave **as** para o cast, mas a partir do **Swift 2.0** foram adicionadas mais duas, totalizando três, veja abaixo:

as: utilizado para fazer o upcasting (cast para o tipo mãe).

as?: faz o cast (ou downcasting) e retorna um tipo opcional, portanto você deve utilizar a opção caso tenha alguma dúvida quanto ao tipo do objeto, se o objeto não for do tipo correto é retornado nil.

as!: faz o cast (ou downcasting) quando você tem certeza de que é o tipo correto e faz o unwrapping da variável. Contudo, tome cuidado, pois se não for o tipo correto o aplicativo vai travar (crash).

Ao executar o projeto o resultado será o mostrado abaixo:

Meu carro é um Civic e o ano dele é 2014

Nota: No Swift 3 existem os tipos **Any** e **AnyObject**. **AnyObject**, que como vimos, pode representar qualquer objeto. O tipo **Any** é ainda mais genérico e pode representar objetos, tipos primitivos e até funções. Vale lembrar que a Apple recomenda usar esses tipos genéricos apenas se você realmente precisar deles. O melhor é utilizar os tipos corretos, por exemplo, usar **Carro** ao invés de **AnyObject**.

Array e Dictionary

Vamos falar um pouco dos tipos mais utilizados em **Swift**.

Array

Array é um array em qualquer lugar, então vamos para a prática.

ViewController.swift

```

class ViewController: UIViewController {

    override func viewDidLoad() {

        super.viewDidLoad()

        testeArray()

    }

    override func didReceiveMemoryWarning() {

        super.didReceiveMemoryWarning()

        // Dispose of any resources that can be recreated.

    }

    func testeArray() {

        let c1 = Carro(nome: "Civic", ano: 2014)

        let c2 = Carro(nome: "Punto", ano: 2010)

        let c3 = Carro(nome: "Peugeot 307", ano: 2004)

        var carros = [c1, c2, c3]

        carros.append(c1)

        print("Este array possui \(carros.count) carros")
    }
}

```

```
        for c in carros {  
            print(c.nome)  
        }  
    }  
}
```

Veja que criamos variáveis que receberam instâncias de seus respectivos carros.

E depois criamos um **array** chamado **carros**.

Feito isso usamos o método **append()** para adicionar um objeto **Carro** ao nosso array. Por fim, imprimimos a quantidade de carros, como o método **count()** e depois fizemos um **for** para percorrer cada índice dos carros, imprimindo o nome de cada um deles.

Observe que ao declarar o array foi omitido o tipo, que é **Array<Carro>**, pois **Swift** possui tipagem fraca e consegue descobrir o tipo da variável conforme o conteúdo da mesma.

Portanto o array de carros poderia ser declarado conforme abaixo:

```
var carros:Array<Carro> = [c1, c2, c3]
```

Caso queria declarar um array vazio e incluir os carros, veja o exemplo abaixo:

```
var carros:Array<Carro> = []  
  
carros.append(c1)
```

```
carros.append(c2)
```

```
carros.append(c3)
```

Ao compilar o projeto verá o seguinte:

Este array possui 4 carros

Civic

Punto

Peugeot 307

Civic

Dictionary

O conceito também é bastante conhecido, é uma estrutura de dados separada em chave:valor.

ViewController.swift

```
class ViewController: UIViewController {
```

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

```
        super.viewDidLoad()
```



```

        testeDictionary()

    }

    override func didReceiveMemoryWarning() {

        super.didReceiveMemoryWarning()

        // Dispose of any resources that can be recreated.

    }

```

```

func testeDictionary() {

    let c1 = Carro(nome: "Civic", ano: 2014)

    let c2 = Carro(nome: "Punto", ano: 2010)

    let c3 = Carro(nome: "Peugeot 307", ano: 2004)


    var carros = ["c1":c1, "c2":c2, "c3":c3]


    for (chave,valor) in carros {

        print("Chave: \(chave), Valor: \(valor.nome)")

    }


    if let carro = carros["c1"] {

        print("Carro: \(carro.nome), ano: \(carro.ano)")

    }

```

```
    }  
}  
}
```

Como fizemos com **Array**, também omitimos o tipo, que é **Dictionary<String,Carro>**. Caso queira declarar um dicionário vazio e depois adicionar os carros, basta usar o código abaixo:

```
var carros = Dictionary<String,Carro>()  
  
carros["c1"] = c1  
  
carros["c2"] = c2  
  
carros["c3"] = c3
```

Obs.: veremos mais a utilização de arrays e dicionários nos exemplos posteriores.

Ao compilar o retorno será:

Chave: c2, Valor: Punto

Chave: c1, Valor: Civic

Chave: c3, Valor: Peugeot 307

Carro: Civic, ano: 2014

Extensions (category)

Uma extensão no **Swift** funciona de modo similar a category no **Obj-C** e permite adicionar métodos dinamicamente em uma classe, sem criar uma classe filha.

Vamos criar uma nova classe, dentro da pasta **Model**, da mesma forma como criamos a classe **Combustivel.swift**, daremos o nome de **String+Extensions.swift**.

String+Extensions.swift

```
import Foundation

extension String {

    func trim() -> String {

        return self.trimmingCharacters(in: CharacterSet.whitespaces)

    }

    func url() -> URL {

        // Utiliza a sintaxe "!" pois o construtor de URL é do tipo init?

        return URL(string:self)!

    }

}
```

Importante: a classe **String** é marcada como final, ou seja, não pode ser criada uma subclasse dela. O conceito de categories permite adicionar um método numa classe sem a necessidade de estendê-la, o que é muito prático.

Repare que depois de criar uma instância de **URL** foi utilizada a sintaxe “!” para desempacotar a variável opcional, pois o construtor da classe **URL** é do tipo **init?**. sendo assim, para simplificar o código é utilizada a sintaxe “!” para não precisar testar se o retorno é **nil**.

Veja abaixo a utilização do método **trim()**.

Carro.swift

```
class ViewController: UIViewController {  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        testeAnyObject()  
    }  
  
    override func didReceiveMemoryWarning() {  
        super.didReceiveMemoryWarning()  
  
        // Dispose of any resources that can be recreated.  
    }  
}
```

}

func testeAnyObject() {

let qualquer:AnyObject = Carro(nome: " Civic ", ano: 2014)

let c = qualquer as! Carro

print("Meu carro é um [(c.nome)] e o ano dele é \ \(c.ano)")

print("Meu carro é um [(c.nome.trim())] e o ano dele é \ \(c.ano)")

}

}

O resultado será:

Meu carro é um [Civic] e o ano dele é 2014

Meu carro é um [Civic] e o ano dele é 2014

Para se comparar objetos em **Swift** usamos o protocolo **Equatable** e sobrescrevemos a função “==”, a qual é responsável por comparar objetos e retornar **true** caso sejam iguais. Veja que o símbolo “==” é uma função.

Carro.swift

class Carro : CombustivelFlex, Equatable {

...

```
static func == (lhs: Carro, rhs: Carro) -> Bool {

    return lhs.nome == rhs.nome && lhs.ano == rhs.ano

}

}
```

Importante: veja que utilizamos outro protocolo, o **Equatable**, que é o que contém a função de comparação “==”. Foi necessário sobrescrever essa função para que ela comparasse objetos.

Veja a implementação de um método para comparar objetos.

ViewController.swift

```
class ViewController: UIViewController {

    override func viewDidLoad() {

        super.viewDidLoad()

        testeObjCarro()

    }
```

```

override func didReceiveMemoryWarning() {

    super.didReceiveMemoryWarning()

    // Dispose of any resources that can be recreated.

}

```

```

func testeObjCarro() {

    let c1 = Carro(nome: "Civic", ano: 2014)

    let c2 = Carro(nome: "Civic", ano: 2014)

    let b = c1 == c2

    print("Objetos iguais? \(b)")

}

}

```

Importante: como podem observar, é necessário comparar cada uma das propriedades para saber se o objeto é realmente igual.

Ao compilar o resultado será:

Objetos iguais? true

xCode

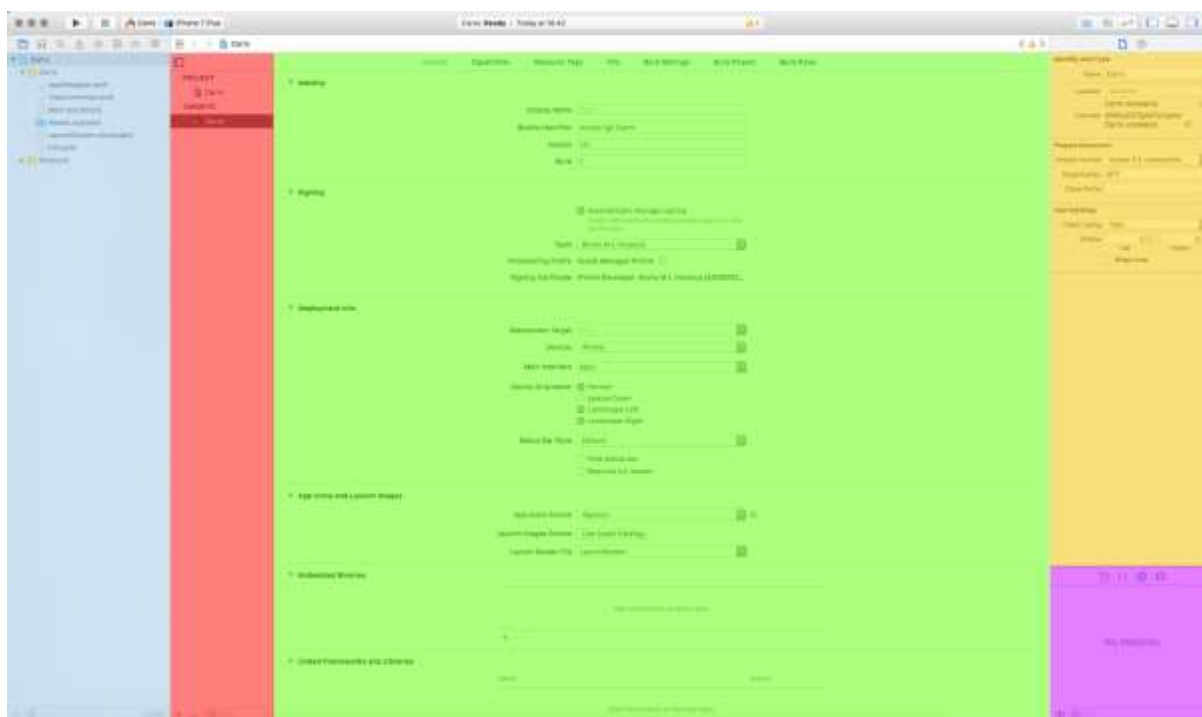
Até o momento vimos conceitos importantes sobre Swift, classes, métodos etc.

Vamos agora entender melhor a IDE de desenvolvimento para posteriormente criamos nosso primeiro app.

O xCode é a IDE oficial da Apple para desenvolvimento de apps para iOS, MacOS, tvOS e watchOS, em resumo é a única IDE da Apple.

Vamos entender um pouco mais sobre as janelas do xCode.

Figura 26 - xCode



Azul: essa área é onde é exibida a estrutura de arquivo do projeto, bem como outras janelas, como debug, erros de compilação, hierarquia de classes e várias outras. Ao clicar em cada um dos ícones na parte logo acima desta área, aparece uma janela diferente, veremos cada uma delas mais adiante.

Vermelho: essa área é onde exibe os targets do projeto. Targets podem ser outros aplicativos (ter um projeto genérico e vários outros targets que são projetos customizados, pode ser também versões do seu app como; lite, pró etc.) ou uma lib que você venha a usar no seu projeto.

Verde: área de conteúdo onde você irá codificar, onde aparecerão as telas visuais de sua storyboard (veremos em detalhes posteriormente).

Amarelo: área destinada a interagir com os componentes visuais que forem selecionados na área verde. Isso ficará mais claro posteriormente.

Roxo: área destinada a exibir os componentes que podemos usar como textbox, labels, tableview, etc.

Dica: neste link (<https://help.apple.com/xcode/mac/8.0/>) é possível ver toda a documentação do xCode.

Figura 27 – Guias de explorer (lado esquerdo do xCode)



Nome da Guia	Descrição
1 – Project Navigator	Exibe a estrutura física das pastas e arquivos do projeto
2 – Symbol Navigator	Exibe a estrutura de cada classe / métodos do projeto
3 – Find Navigator	Nesta guia podemos pesquisar por textos nos arquivos do projeto, bem como pesquisar onde está sendo usado método x etc.

4 – Issue Navigator	Exibe erros, warnings do projeto
5 – Test Navigator	Exibe o resultado de teste unitários criados no projeto bem como erros e warnings desses testes
6 – Debug Navigator	Exibe o debug, a ordem de threads iniciadas do projeto
7 – Breakpoint Navigator	Exibe os breakpoints incluídos no projeto bem como pode configurar propriedades dos breakpoints também
8 – Report Navigator	Exibe relatórios de como está compilação atual, console de debug etc.

A figura 27 exibe os botões que nos permitem alterar as propriedades das views e componentes de tela que usarmos.

Figura 28 – Guias para alterar os componentes (lado direito do xCode)



Nome da Guia	Descrição
1 - File Inspector	Permite visualizar as informações do arquivo como localização no projeto ou no sistema operacional real
2 – Quick Help Inspector	Rápida ajuda sobre o componente
3 – Identity Inspector	Permite customizar o nome da classe do componente selecionado e seus metadados

4 – Attributes Inspector	Permite customizar os atributos do componente selecionado
5 – Size Inspector	Permite customizar o tamanho de um componente, assim como seu alinhamento na tela. Nesta guia podemos configurar também as propriedades do Autosizing e Autolayout
6 – Connections Inspector	Permite visualizar os outlets e actions de um componente, ou seja, ligações com a classe.

A figura 28 exibe os botões que nos permitem alterar, incluir e deletar arquivos do projeto, enfim, nos permite mexer em toda a estrutura lógica e física do projeto, nos permitem também debugar código, inspecionar hierarquia de classes etc.

Capítulo 3. Navegação entre telas, Storyboards, UINavigationController

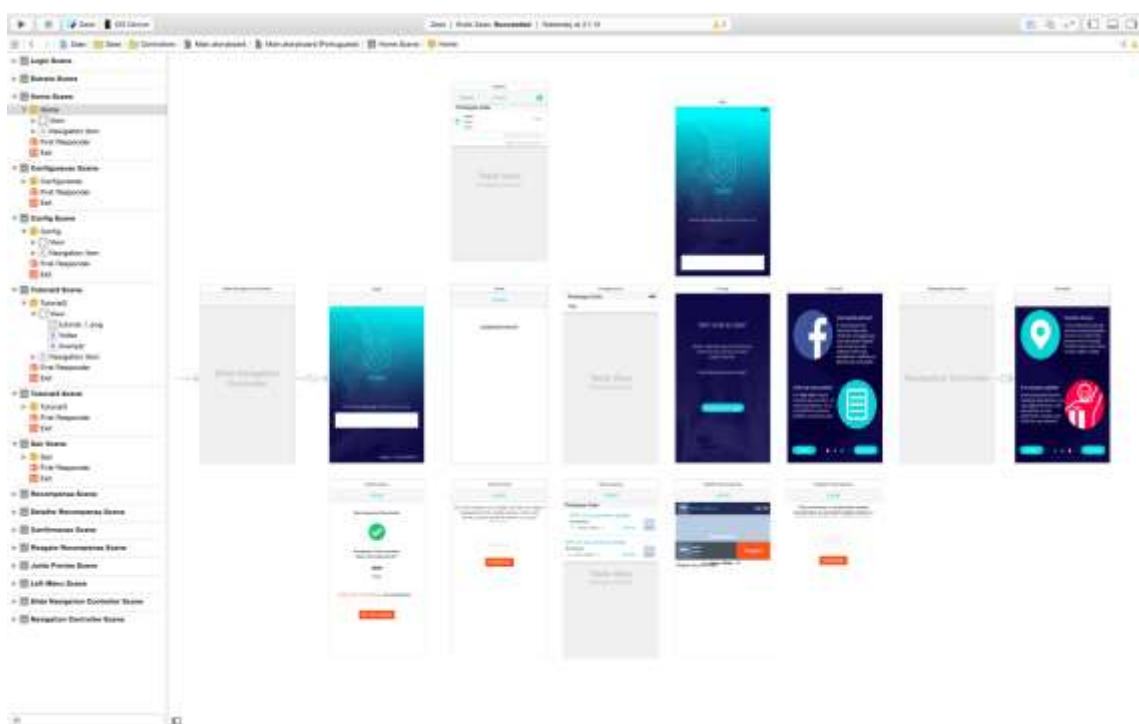
Neste capítulo vamos aprender as melhores práticas para navegar entre telas (controllers) e como passar parâmetros de uma para outra.

Storyboards

O arquivo **Main.storyboard** é um arquivo muito importante para o nosso projeto, pois é nele que iremos criar as telas do app.

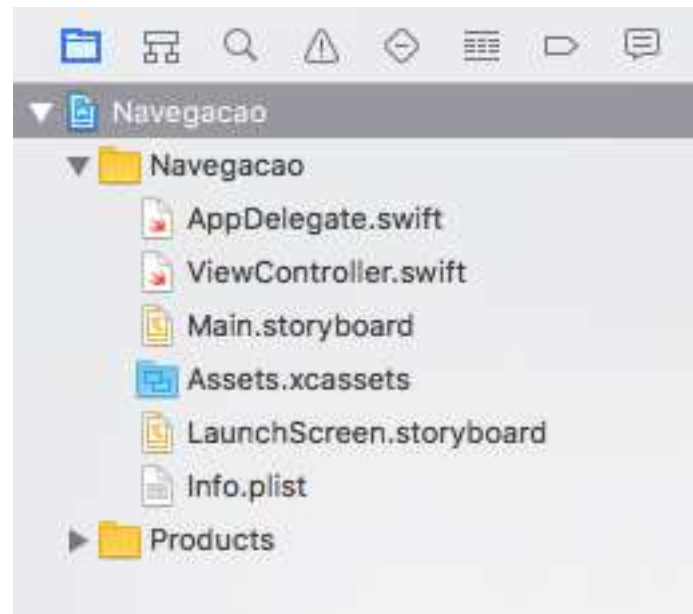
Importante: em versões do xCode anteriores a 4.0 eram utilizados arquivos .xib (ainda são utilizados, mas em menor proporção) que eram a representação gráfica da interface da tela de um app, mas após o iOS 5 surgiram as storyboards, uma forma simples e muito prática para visualizar estas telas. Com ela podemos editar toda interface gráfica do app e também visualizar todo o fluxo do app com facilidade.

Figura 29 – Storyboards



O template escolhido para a criação do projeto cria automaticamente um arquivos de **Storyboard** e alguns outros arquivos. Veremos o que cada um deles significa.

Figura – 30 – Estrutura de arquivos



Conforme mostra a figura 30, vamos enumerar e explicar cada um dos arquivos que o xCode cria para nós.

Arquivo ou pasta	Descrição
AppDelegate.swift	Utilizada para iniciar o aplicativo, recebe eventos do sistema operacional para isso.
Info.plist	Responsável pelas configurações do app, como o nome, ícone, código da versão, orientações, em resumo é o arquivo de configuração do projeto.
Main.storyboard	Storyboard inicial do projeto onde conterà as views (telas) que forem usadas pelo app.

LaunchScreen.storyboard	Arquivos de screen de inicialização do app.
ViewController.swift	Arquivo de cabeçalho e implementação da classe criada automaticamente.
Assets.xcassets	Pasta que contém as imagens do projeto, organizadas em 1x, @2x e @3x.
Products	Pasta que contém o app compilado (arquivo .app).

AppDelegate.swift

```
import UIKit
```

```
@UIApplicationMain
```

```
class AppDelegate: UIResponder, UIApplicationDelegate {
```

```
    var window: UIWindow?
```

```
    func application(_ application: UIApplication,  
didFinishLaunchingWithOptions launchOptions: [UIApplicationLaunchOptionsKey:  
Any]?) -> Bool {
```

```
        // O método acima é chamado quando a aplicação está iniciando, o  
dictionary informado no parâmetro contém informações sobre a inicialização,  
se existir.
```

```
        return true
```

```
}
```

```
func applicationWillResignActive(_ application: UIApplication) {
```

```
// Chamado para indicar que a aplicação vai mover do estado ativo
```

para o

inativo, que pode acontecer se o usuário estiver recebendo uma
ligação.

```
}
```

```
func applicationDidEnterBackground(_ application: UIApplication) {
```

```
// Chamado para indicar que a aplicação entrou em background,
```

que pode

```
acontecer se o usuário apertar o botão home.
```

```
}
```

```
func applicationWillEnterForeground(_ application: UIApplication) {
```

```
// Chamado quando a aplicação está voltando do background,
```

antes de ser

```
exibida ao usuário.
```

```
}
```

```
func applicationDidBecomeActive(_ application: UIApplication) {
```

```
// Chamado para indicar que a aplicação moveu do estado  
inativo para ativo,  
  
isso pode acontecer depois de que usuário finaliza uma ligação  
  
}
```

```
func applicationWillTerminate(_ application: UIApplication) {  
  
    // Chamado quando a aplicação será finalizada e liberada a  
    memória. Utilize  
  
    esse método para apagar todos os recursos que estiver  
    utilizando.  
  
}  
  
}
```

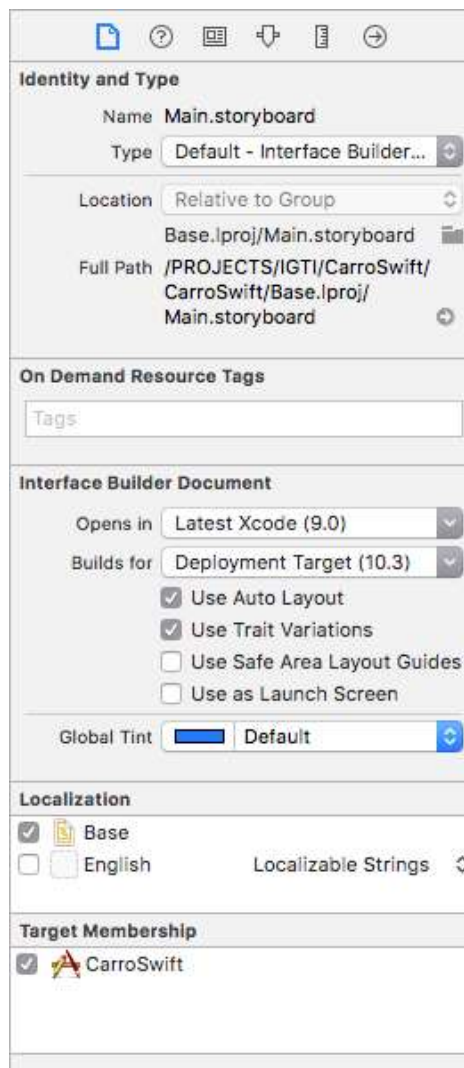
Guias

Vamos agora ver um pouco sobre as principais guias que utilizaremos sempre que estivermos desenvolvendo.

Todas as guias são dinâmicas, ou seja, dependendo do arquivo selecionado elas serão diferentes.

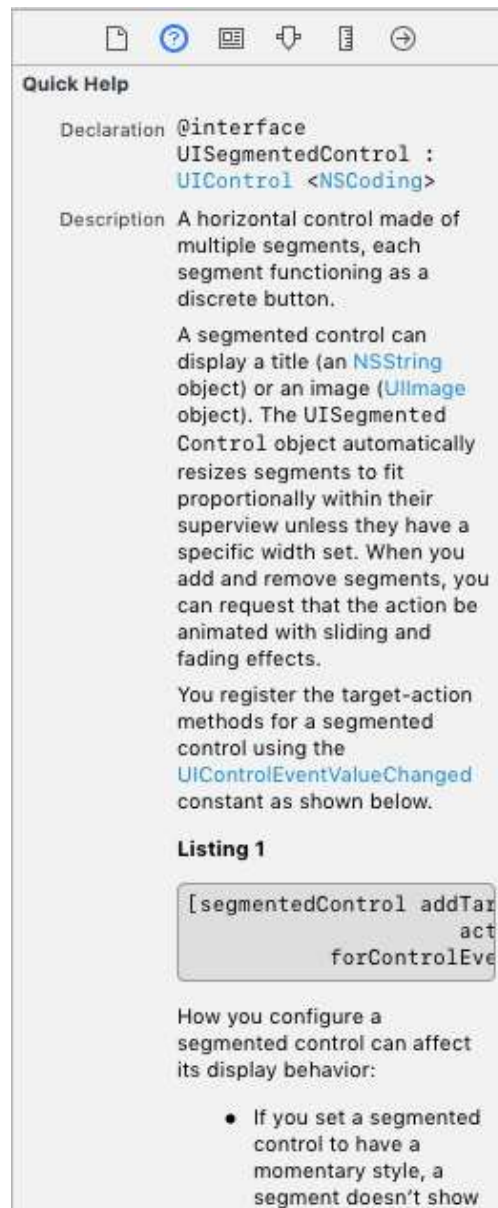
A primeira é a **File Inspector**, que é exibida na figura 3.1.1, e exibe configurações do arquivo selecionado, bem como se utilizar **Auto Layout**.

Figura 3.1.1 – Guia File Inspector



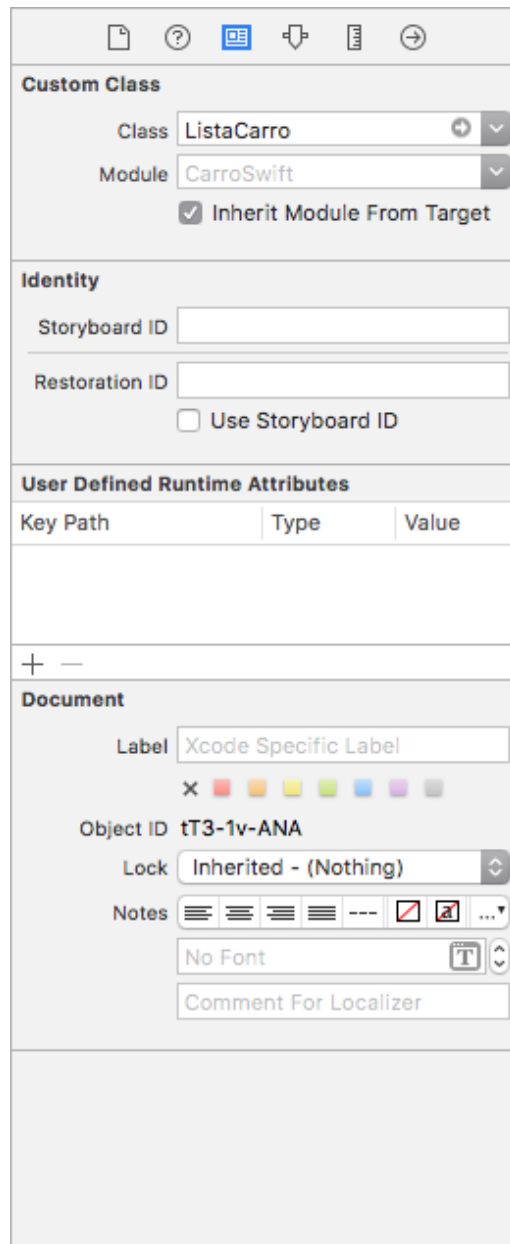
A segunda guia é a **Quick Help Inspector**, que é exibida na figura 3.1.2 e é a guia de ajuda, nela são exibidas informações sobre documentação da Apple (que é instalada juntamente com xCode) de cada componente selecionado.

Figura 3.1.2 – Guia Quick Help Inspector



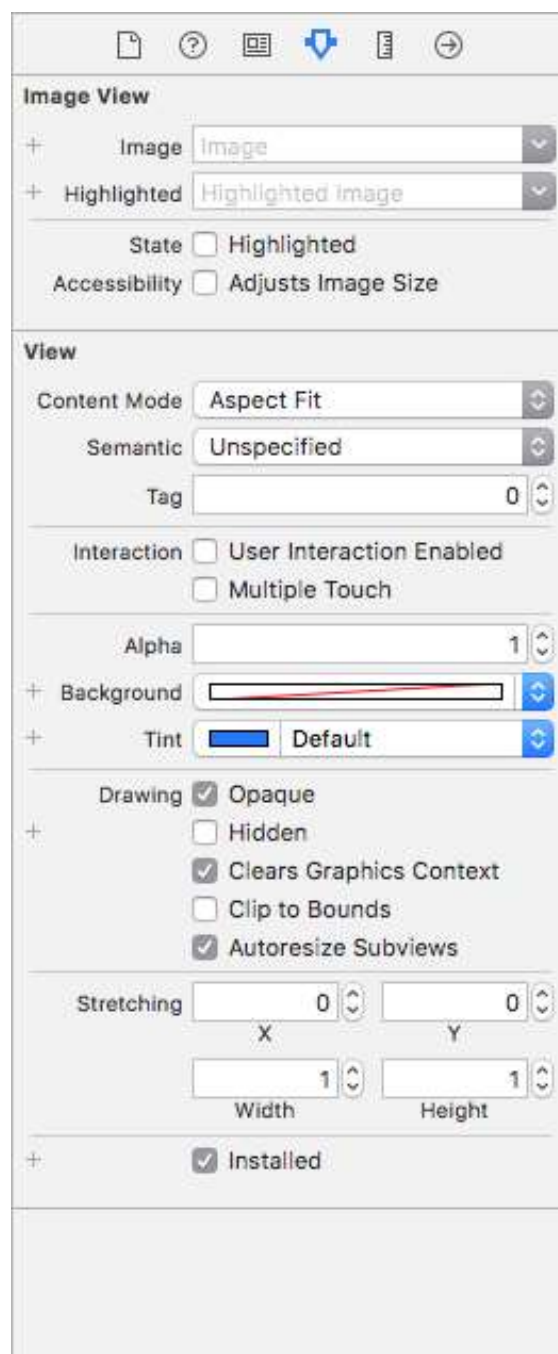
A terceira guia é **Identity Inspector**, que é exibida na figura 3.1.3. Esta guia exibe informações sobre qual classe está vinculada a um componente ou a uma **viewController**.

Figura 3.1.3 – Guia Identity Inspector



A quarta guia é a **Attributes Inspector**, que é exibida na figura 3.1.4. Essa guia exibe configurações de propriedades do componente selecionado (TableView, TextField, TextView etc.), e nela podemos mudar propriedades (background, cor de texto, nome da imagem etc.) de qualquer componente visual que utilizarmos.

Figura 3.1.4 – Guia Attributes Inspector



A quinta guia é a **Size Inspector**, que é exibida na figura 3.1.5. Essa guia exibe informações sobre **AutoLayout** aplicado nos componentes ou **AutoSizing**, depende de qual opção o desenvolvedor escolheu. Exibe também espaçamentos entre células, no caso de uma **Tableview**, e várias outras informações.

Figura 3.1.5 – Guia Size Inspector

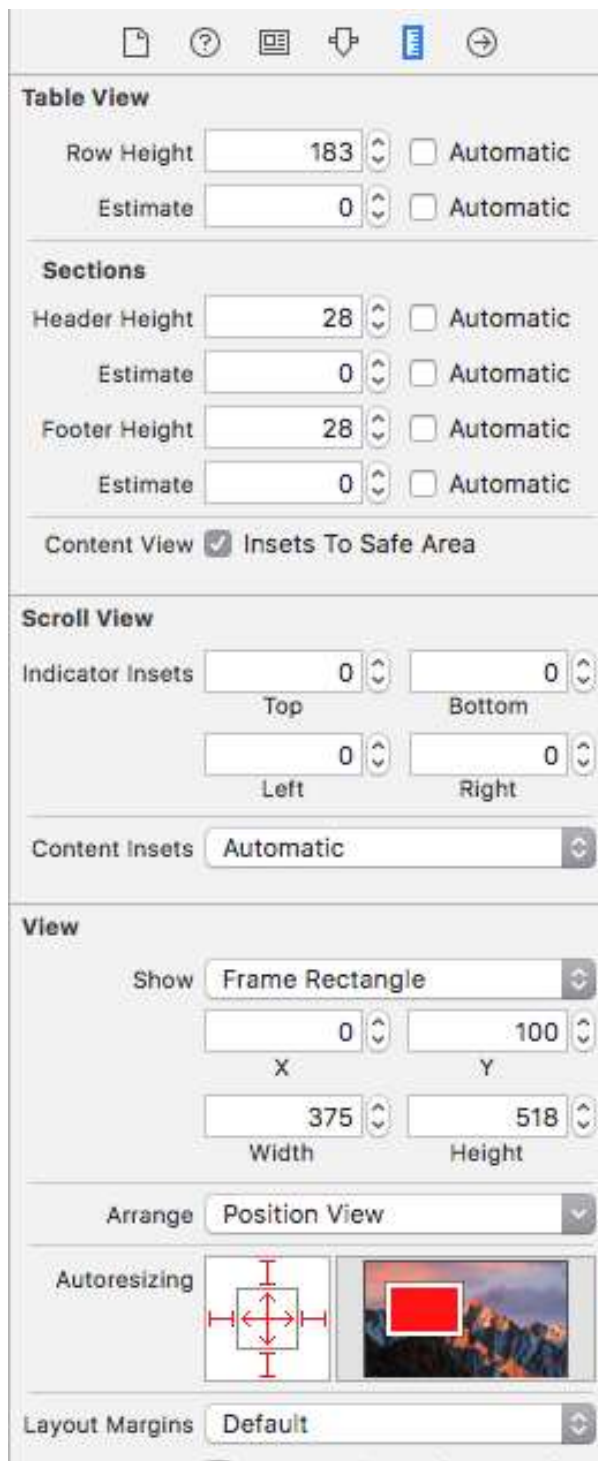


Table View

Row Height ☐ Automatic

Estimate ☐ Automatic

Sections

Header Height ☐ Automatic

Estimate ☐ Automatic

Footer Height ☐ Automatic

Estimate ☐ Automatic

Content View ☒ Insets To Safe Area

Scroll View

Indicator Insets
Top Bottom

Left Right

Content Insets



View

Show

X Y

Width Height

Arrange

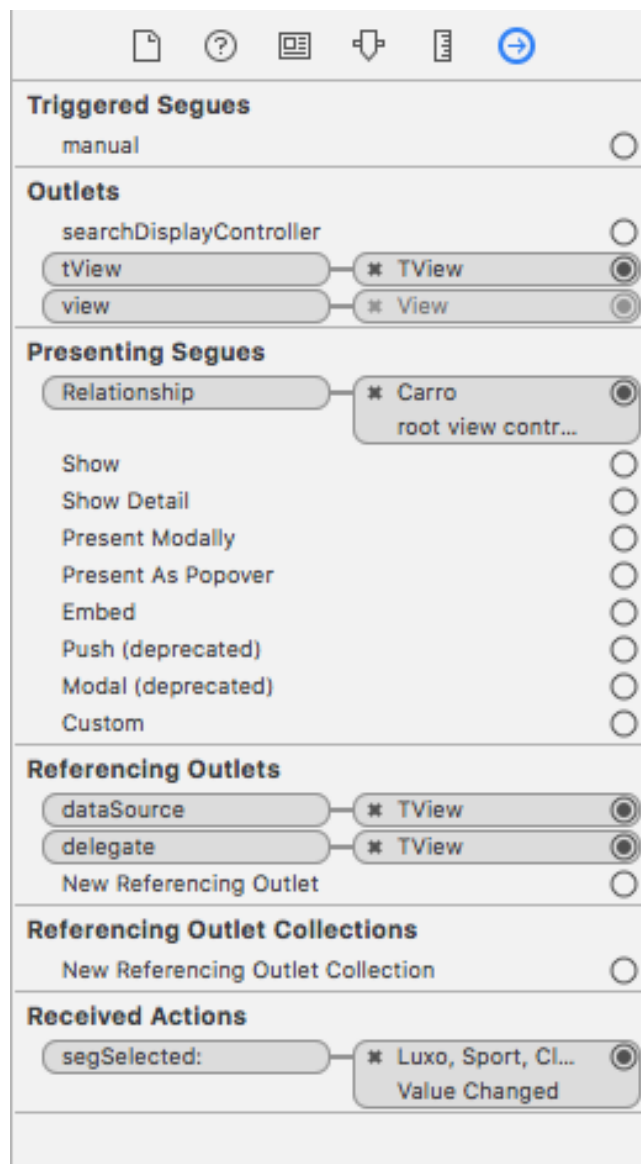
Autoresizing  

Layout Margins

A sexta e última guia é a **Connections Inspector**, que é exibida na figura 3.1.6. Essa guia exibe as conexões entre componentes visuais e classe

previamente selecionada na guia **Identity Inspector**. Essas conexões podem ser variáveis, métodos, DataSource, Delegates etc.

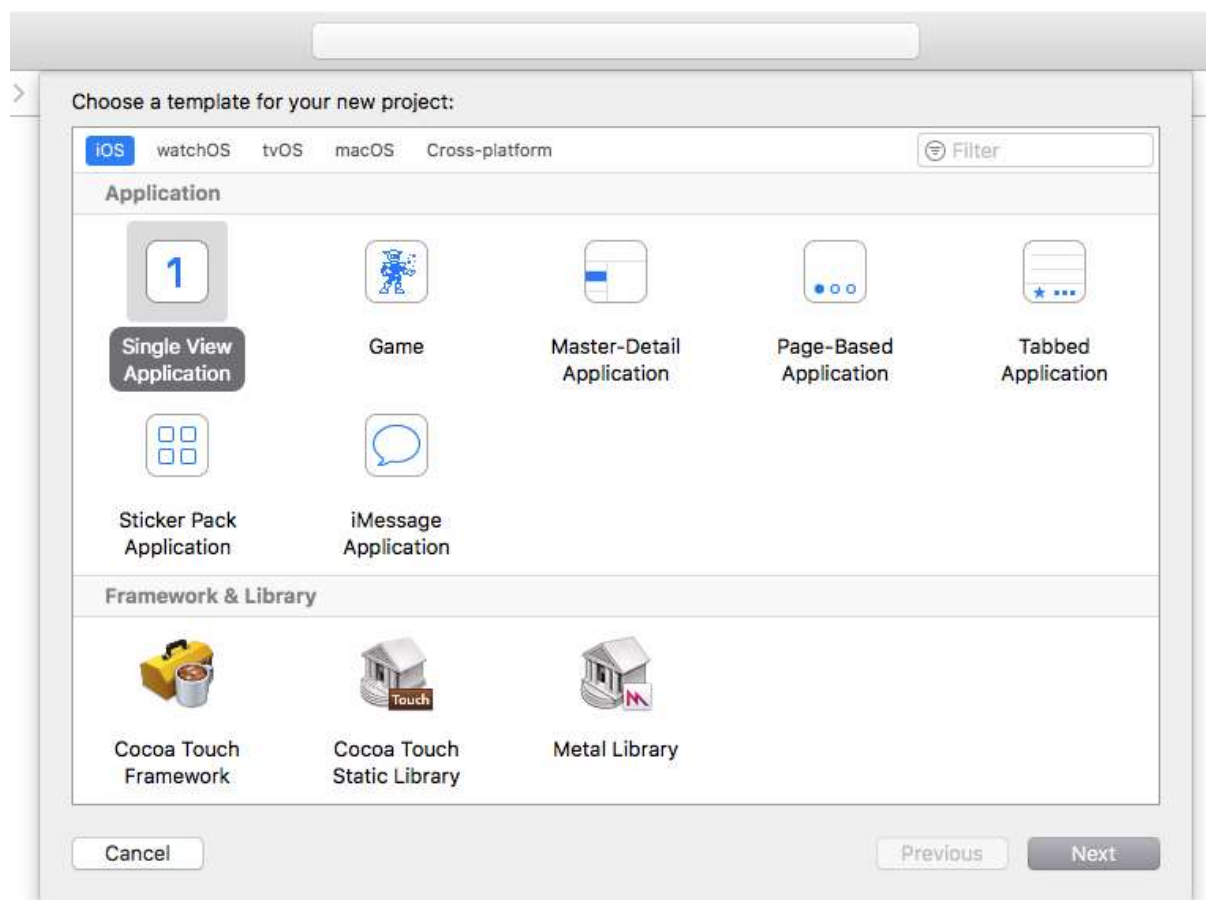
Figura 3.1.6 – Guia Connections Inspector



Navegação entre telas

Vamos começar criando um novo projeto no xCode. Escolha a opção **Create a New xCode Project**, logo após aparecerá uma tela para escolha do template, veja na figura 29.

Figura 29 – Criando novo projeto



Na próxima tela dê o nome de **Navegacao** para o seu projeto, conforme mostra a figura 30. Escolha um lugar para salvar e pronto. Seu projeto deve estar igual ao da figura 31.

Figura 30 – Configurando criação de projeto

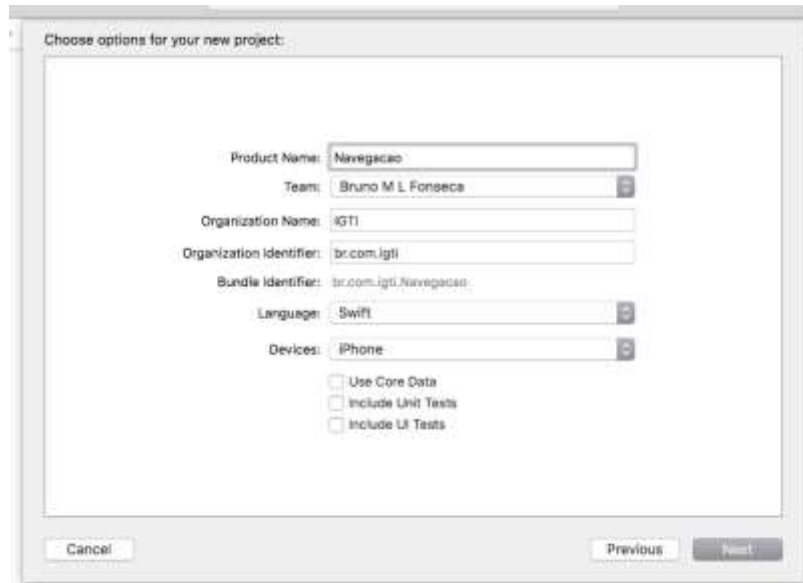
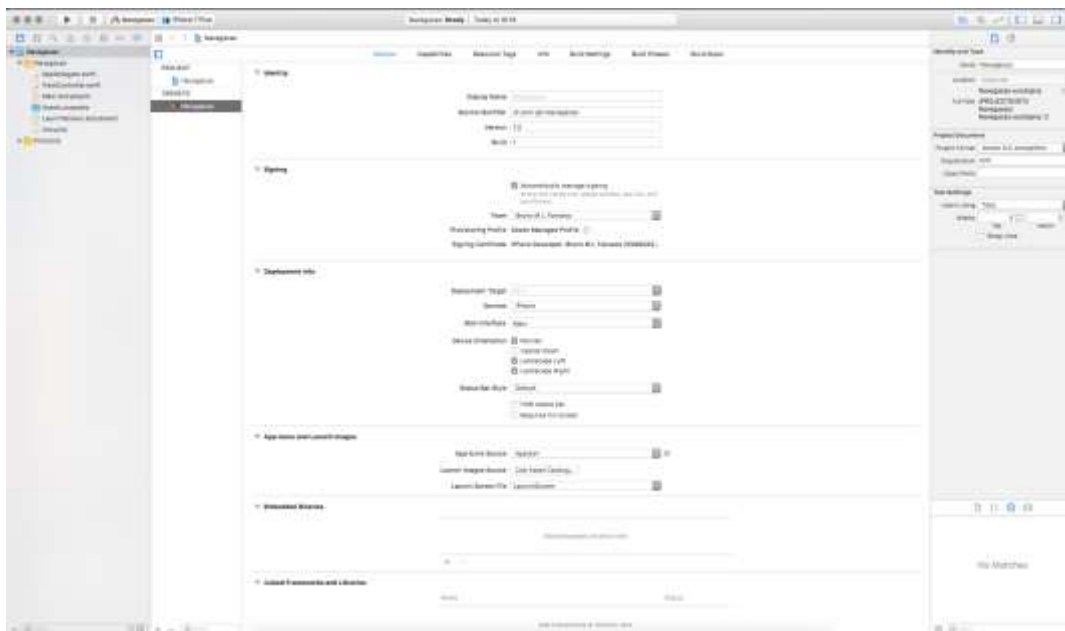
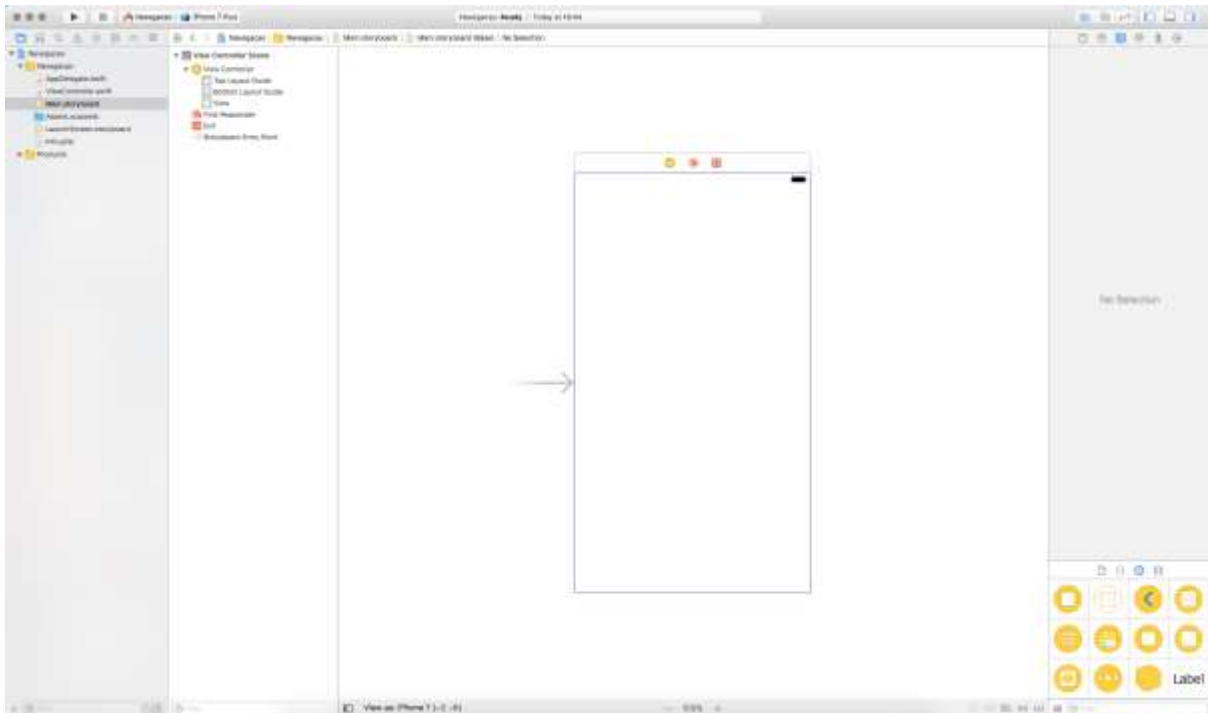


Figura 31 – Novo projeto criado



Agora selecione o arquivo **Main.storyboard**, sua tela deverá estar igual à figura 32.

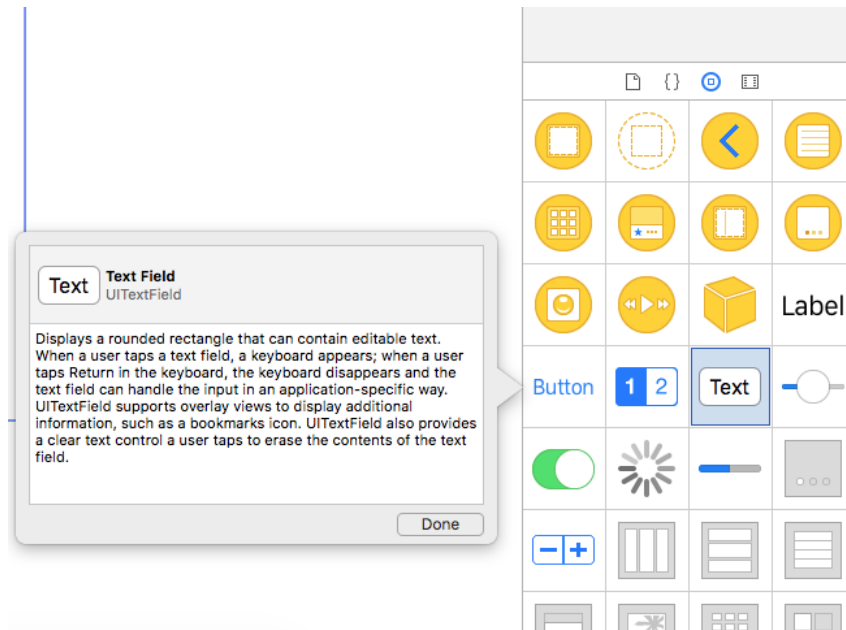
Figura 32 – Selecionando uma storyboard



Vamos aproveitar a estrutura já criada pelo template que escolhemos.

Como podem ver, essa é a nossa view, se compilarem o projeto **command + R** poderão ver o app rodando, mas antes vamos incluir alguns componentes visuais nessa tela.

Figura 33 – Componentes visuais



Na figura 33 vocês veem alguns componentes visuais que podemos usar (será uma tela de login), vamos escolher um campo de texto e arrastá-lo para a nossa view. Você pode mudar o tamanho dele e colocar na posição que achar melhor.

Incluímos dois campos de textos e um botão (**button** na aba de componentes visuais), conforme figura 34.

Figura 34 – Componentes visuais incluídos no projeto



Figura 35 – Propriedades de um componente visual



Vamos alterar o nome do botão que adicionamos. Selecione o botão; você verá que a aba de detalhe das propriedade de componente visual irá mudar para as propriedades do componente selecionado, no nosso caso, um botão.

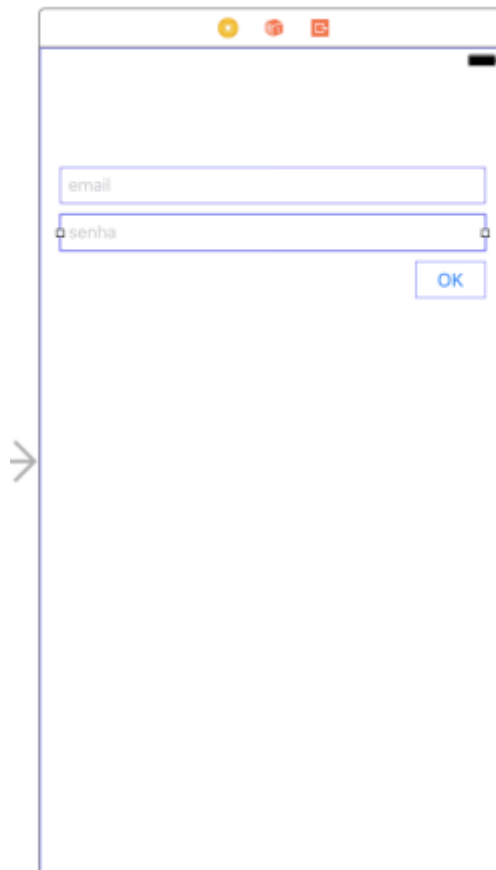
Altere o campo **Default Title**, (está escrito **button** ele). Vamos colocar **OK** como título do botão.

Sua tela deverá parecer como a figura 34.

Agora, vamos selecionar cada um dos campos texto que criamos e alterar a propriedade **Placeholder Text**.

Sua view deverá estar como a figura 36.

Figura 36 – Componentes visuais preenchidos



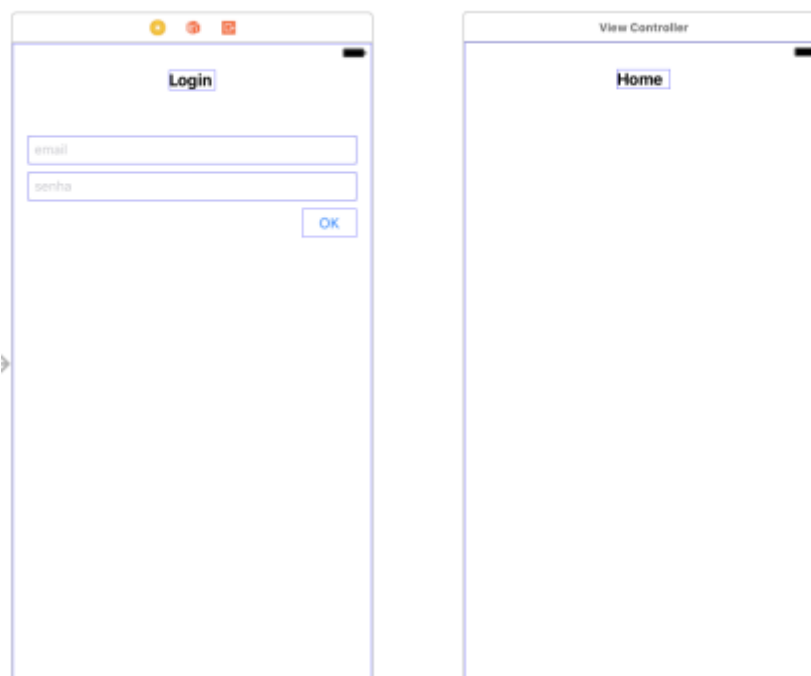
Bom, vamos agora criar uma outra view, arrastando o componente de view conforme figura 37.

Figura 37 – Incluindo nova view ao projeto



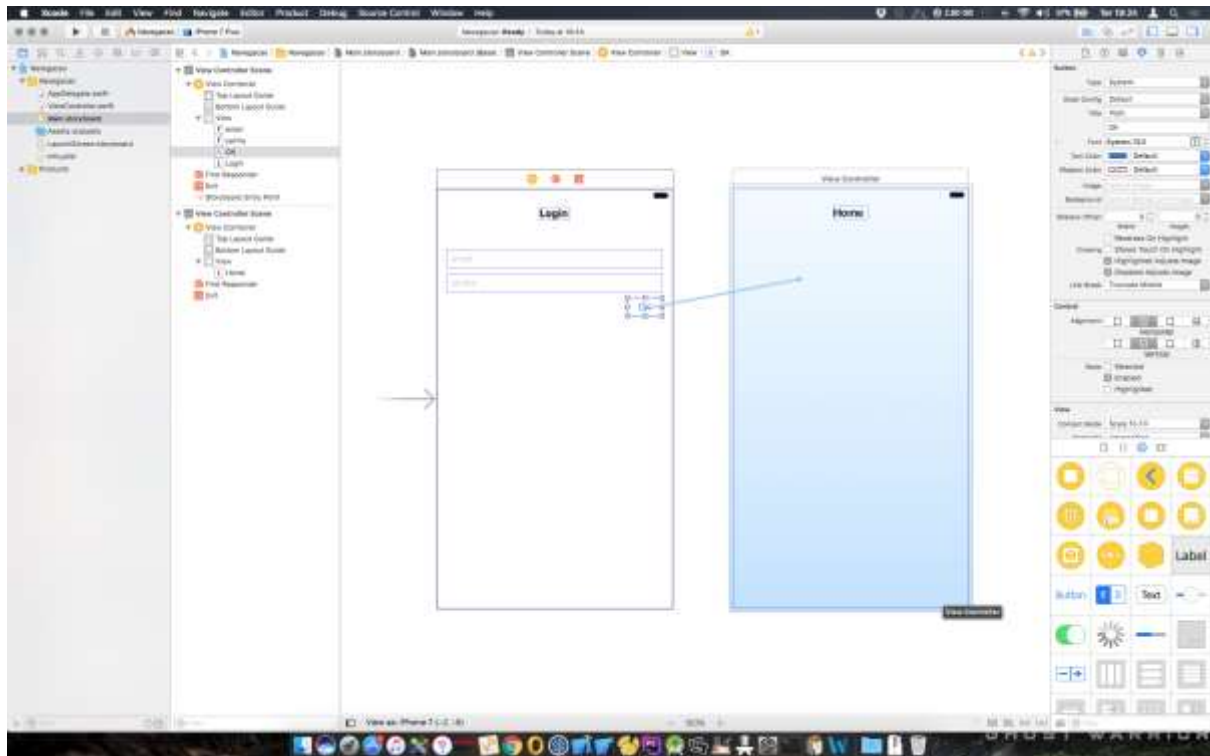
Inclua também dois labels, um para a tela de login e outro para a tela home, recém criada. Veja figura 38.

Figura 38 – Adicionando nova view



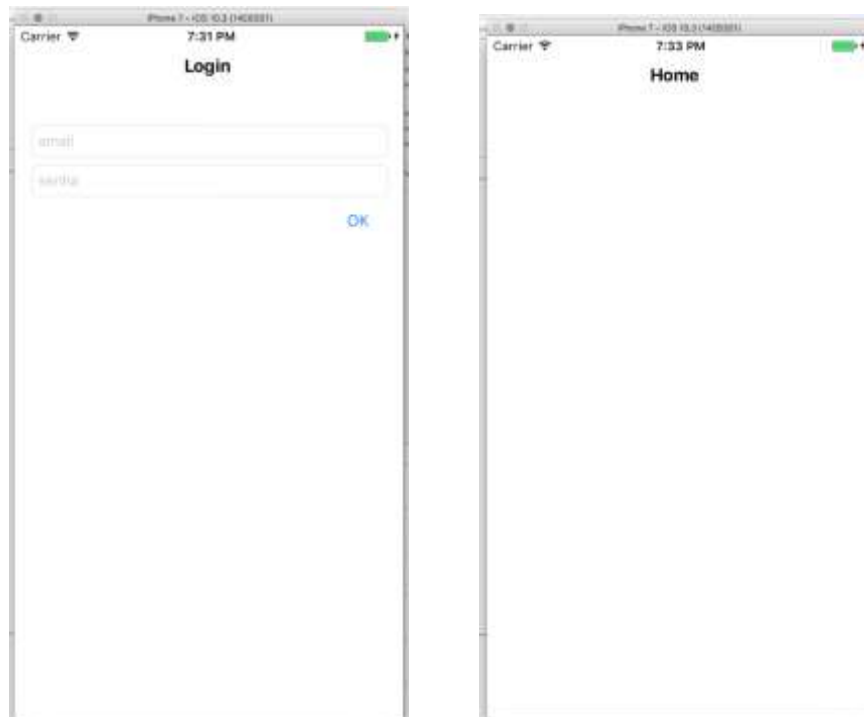
Agora vamos clicar com o botão direito do mouse em cima do botão e arrastá-lo para a nova tela criada, conforme mostra a figura 39. Ao soltar o botão do mouse irá aparecer um menu de contexto; escolha a opção **Present Modally** e pronto, agora vamos compilar o projeto **command + R**.

Figura 39 – Ligando componente visual a outra tela



Depois de compilar o projeto, ele deverá estar conforme a figura 40. Ao clicar no botão será direcionado para a tela home.

Figura 40 – Projeto compilado, clicando no botão “OK” será direcionado para a tela Home

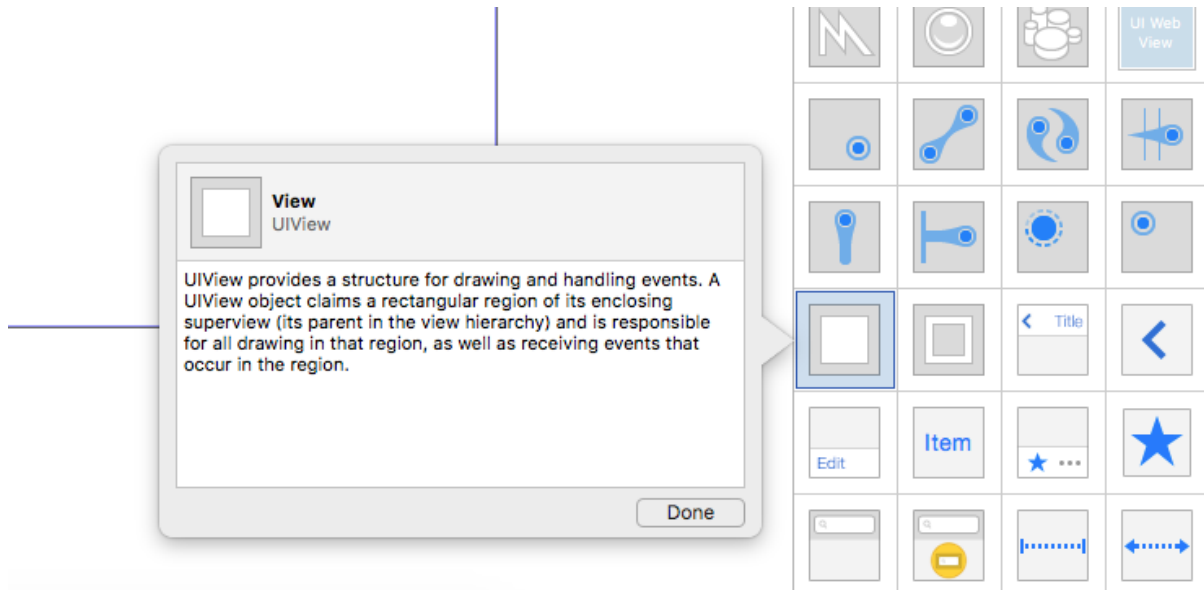


Então, vimos o quanto é simples fazer a transição de uma tela para outra, mas repare que não passamos nenhum parâmetro para a tela home. Outro detalhe é que, depois de clicarmos no botão “**OK**”, ficamos presos na tela home.

Vamos modificar isso.

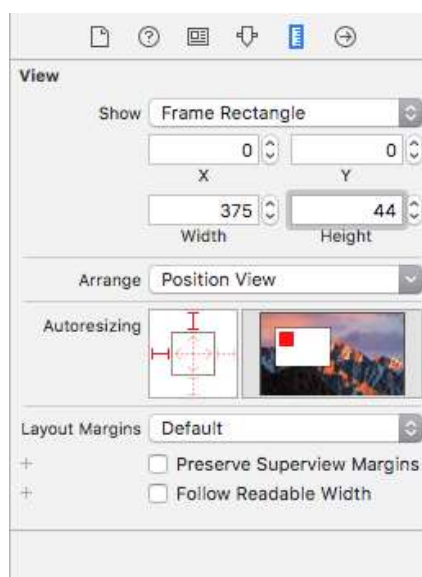
Vamos agora incluir uma nova view dentro da tela home, então vá na aba de componentes visuais e escolha uma view, conforme figura 41.

Figura 41 – Incluindo uma nova view



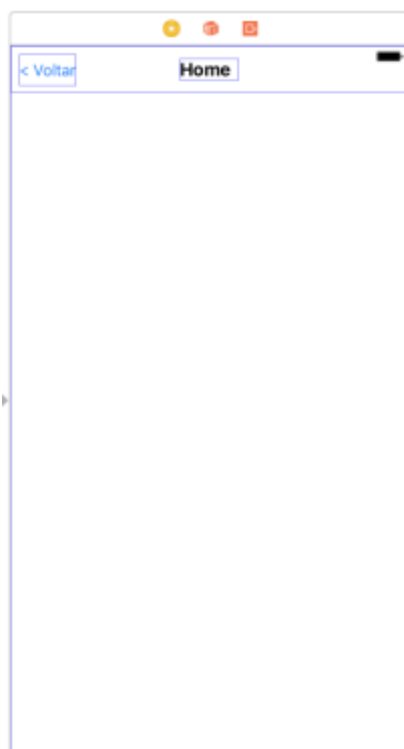
Coloque a view totalmente no topo da nossa tela e deixe com o **height**, conforme figura 42. Para chegar nesta aba deverá selecionar a view e ir na aba do lado direito. Caso tenha dúvidas, verifique no capítulo 2 as descrições de cada aba do xCode.

Figura 42 – Alterando propriedades da view



Feito isso, adicione um botão e organize os componentes visuais conforme figura 43.

Figura 43 – Organização dos componentes visuais



Até agora apenas incluímos os componentes visuais necessários para nossa tela, lembrando que tudo que fizemos poderia ser feito via código.

Agora vamos ligar o botão ao nosso código, mas para fazermos isso precisamos criar um novo arquivo que será o arquivo de controller da nossa viewController “**Home**”. Clique com o botão direito em cima da pasta “**Navegação**” e escolha a opção **New File**. Escolha a opção Cocoa Touch Class (figura 44) e clique no botão **Next**, em seguida dê um nome para o arquivo, veja figura 45.

Figura 44 – Criação de arquivo para tela listagem

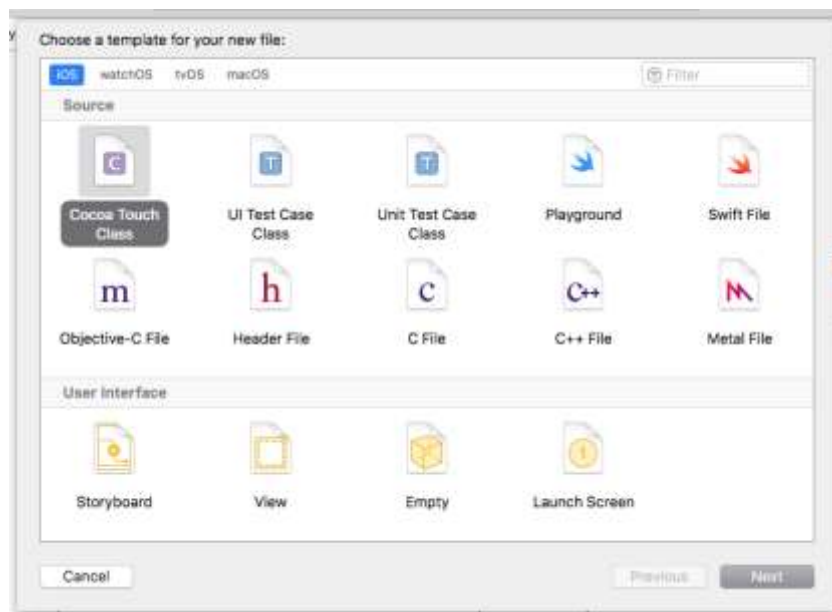
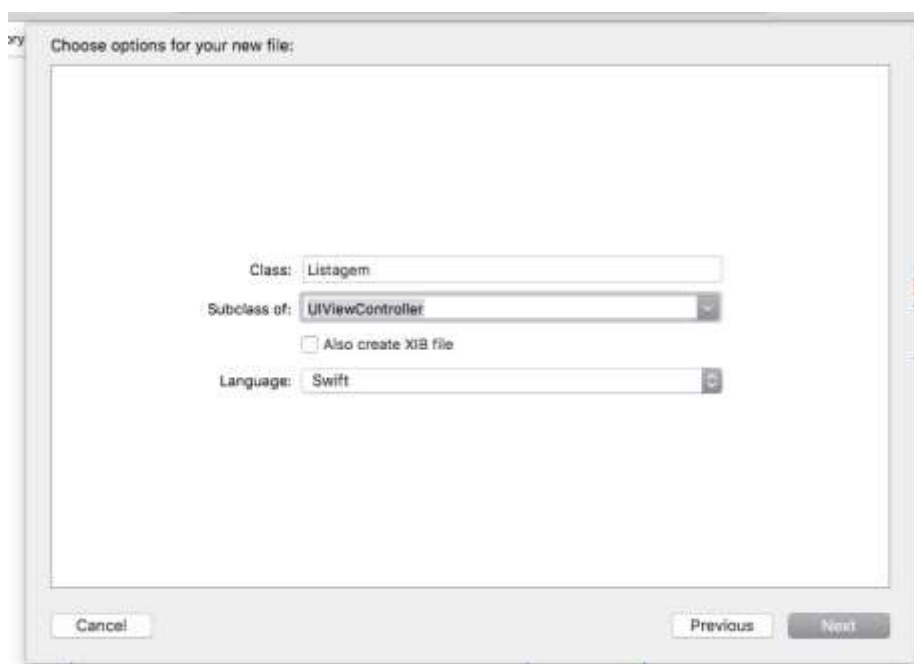


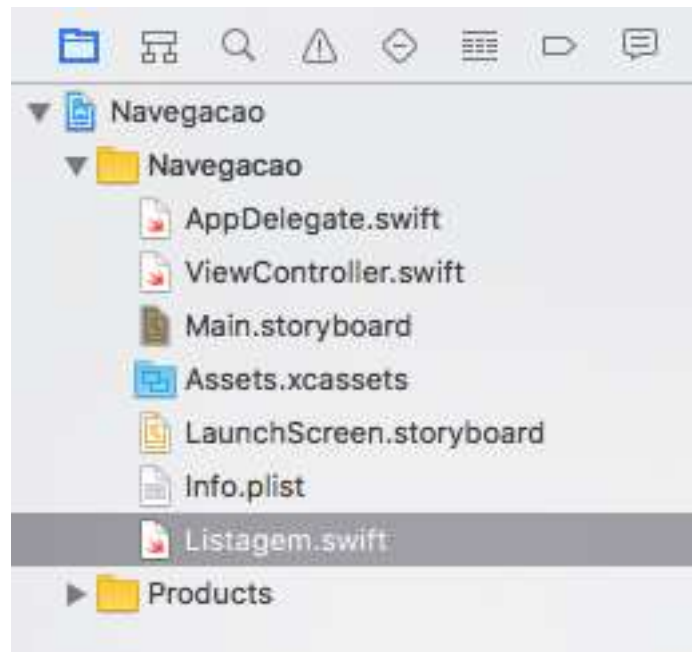
Figura 45 – Dando nome ao arquivo para a tela de listagem



Repare que o nome da nossa classe será **Listagem**, a subclass será **UIViewController** e a linguagem será **Swift**. Clique em next e salve o arquivo na pasta raiz do projeto (ela já estará selecionada).

Sua estrutura de arquivos deverá estar conforme figura 46.

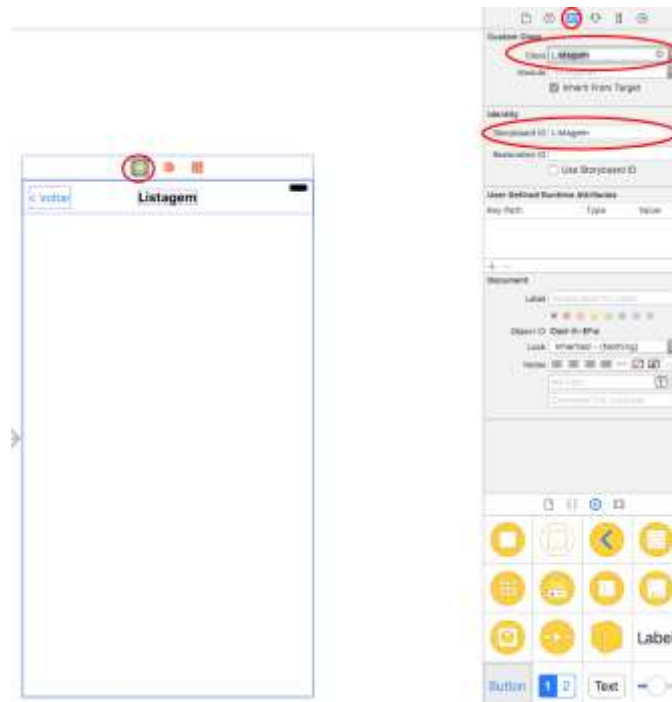
Figura 46 – Estrutura de arquivos



Agora que criamos a classe responsável pela nossa tela de Listagem, vamos linká-las. Veja os campos em vermelho.

Primeiro você deve selecionar na tela o primeiro ícone que é o objeto da nossa tela, feito isso vá até a aba **Identity Inspector** e no campo **Class** coloque o nome do arquivo que acabamos de criar, vamos aproveitar e colocar no campo **Storyboard ID** um ID para podermos chamar nossa tela de onde quisermos. Repare que pode ser o nome que quiser, inclusive o mesmo nome do nosso arquivo, mas é importante que seja um nome que tenha a ver com a tela, que no nosso caso é uma listagem.

Figura 47 – Linkando classe recém criada a sua viewController no storyboard

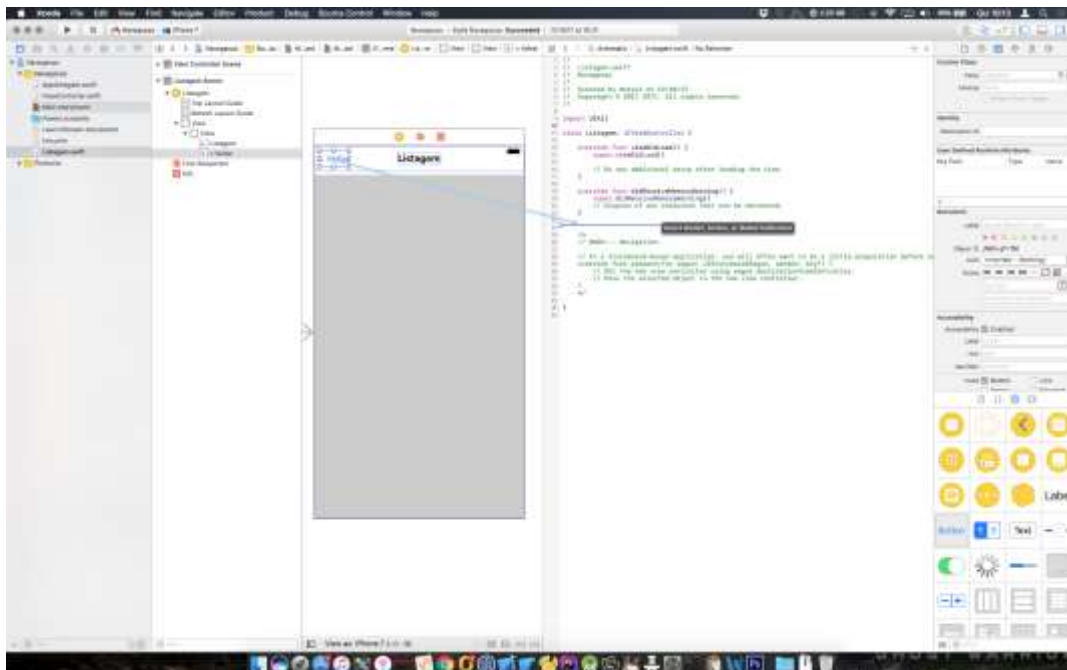


Feito isso, o que precisamos fazer agora é ligar o botão a um método (que iremos criar) na nossa classe.

Conforme figura 48, vamos acessar a aba **Assitant Editor**, ela nos dará visão da nossa **viewController** e da classe que vinculamos a ela.

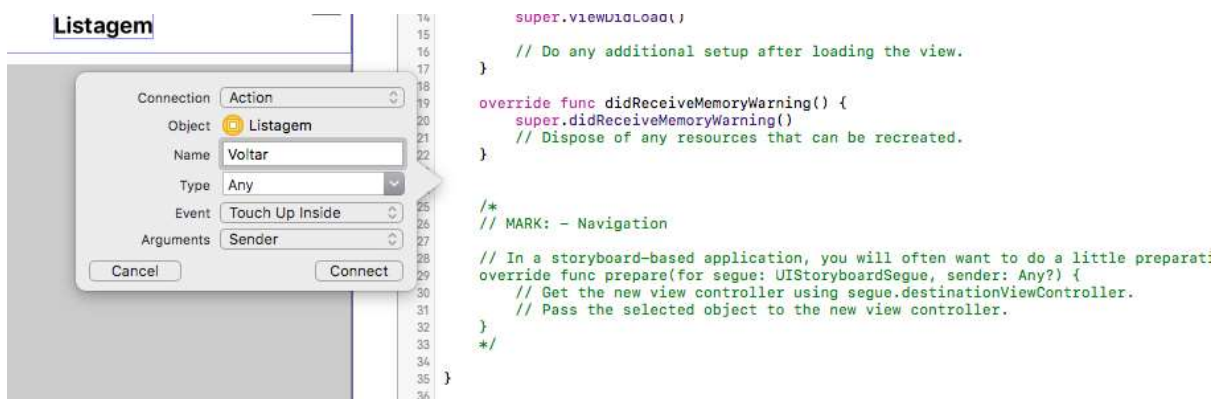
Vamos clicar com o botão direito do mouse sobre o botão “voltar” e arrastar até o código como mostrado na figura 48.

Figura 48 – Ligando botão voltar ao método



Ao soltarmos o botão do mouse veremos o menu mostrado na figura 49.

Figura 49 – Criando e vinculando componente visual a um método.



Observe os campos que iremos alterar; o campo **Connection** deve estar setado para **Action**, isso significa que vamos criar um método, o campo **Name** deve conter o nome do seu método, no nosso caso, “**Voltar**”; quanto aos outros campos, pode-se deixar o valor padrão que veio neles. Clique no botão **Connect** e será

criado um método conforme abaixo. Dentro desse método vamos colocar o comando para voltar.

```
@IBAction func Voltar(_ sender: Any) {  
  
    self.dismiss(animated: true, completion: nil)  
  
}
```

Vamos falar um pouco do código que usamos.

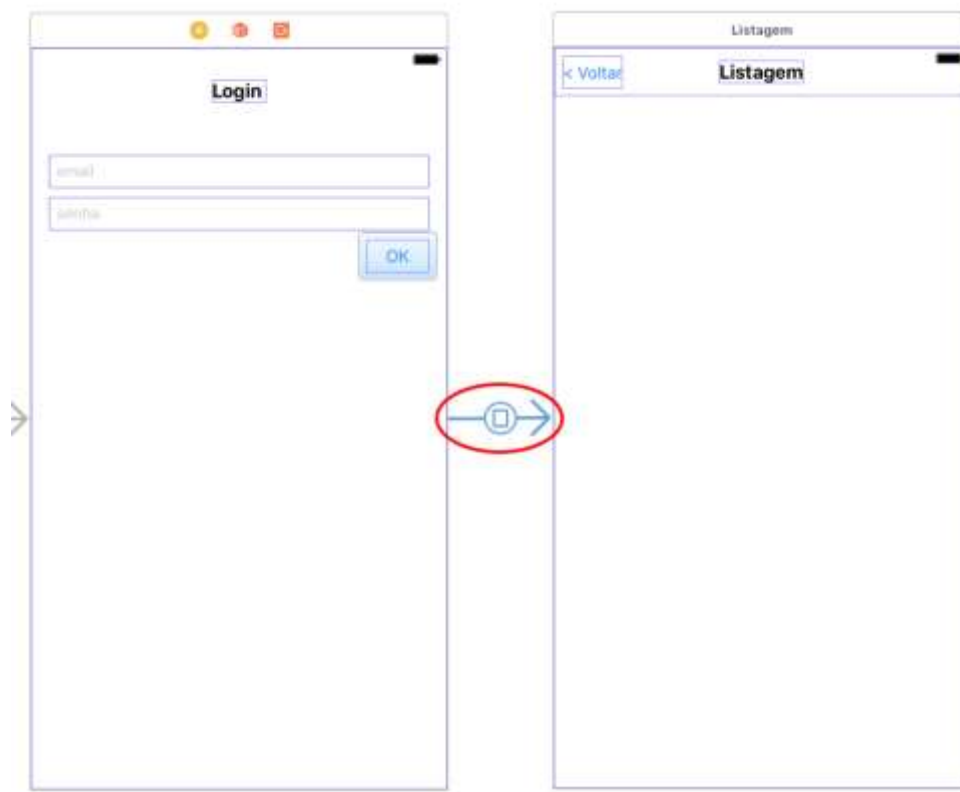
A palavra **@IBAction** significa que esse nosso método está ligado diretamente a um componente visual, que no nosso caso é o botão voltar. Nesse nosso método é passado um parâmetro que não usamos ainda, mas é a referência ao objeto chamador, ou seja, **sender** é o nosso botão voltar. Por fim, usamos um comando para dar um **dismiss** na tela que chamamos passando dois parâmetros **animated**, significa que nossa tela vai sumir fazendo uma animação e **completion** como nil, pois não usamos esse parâmetro.

Ao compilarmos (**command + R**) no nosso app, podemos ir até nossa tela e clicar no botão voltar, veremos uma animação ao contrário da animação da chamada da tela.

Agora vamos passar parâmetro para esta nossa tela de listagem.

Para isso vamos para a nossa tela de **Login**, mas antes vamos apagar o segue criado, apenas selecione clicando em cima dele e aperte delete no teclado.

Figura 50 – Apagando o segue criado para chamar a tela de listagem



Feito isso, vamos agora ligar o botão “**OK**” ao um método na nossa classe **ViewController**. Faremos da mesma forma com que ligamos o botão “**Voltar**”.

Faça como teste, o resultado será o mostrado na figura 51.

ViewController.swift

```
@IBAction func Logar(_ sender: Any) {  
  
    ...  
  
    let vc = storyboard?.instantiateViewController(withIdentifier:
```

"Listagem") as! Listagem

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

    }

    ...
```

Vamos comentar um pouco o código acima.

Criamos uma constante com **let** chamada **vc**, ela recebe o resultado de um cast que fazemos, identificamos a ViewController que queremos chamar com o método **instantiateViewController()** da classe **storyboard** no parâmetro **withIdentifier**, este parâmetro busca pelo campo **Storyboard ID** que incluímos na nossa viewController anteriormente e fazemos cast para a classe **Listagem**.

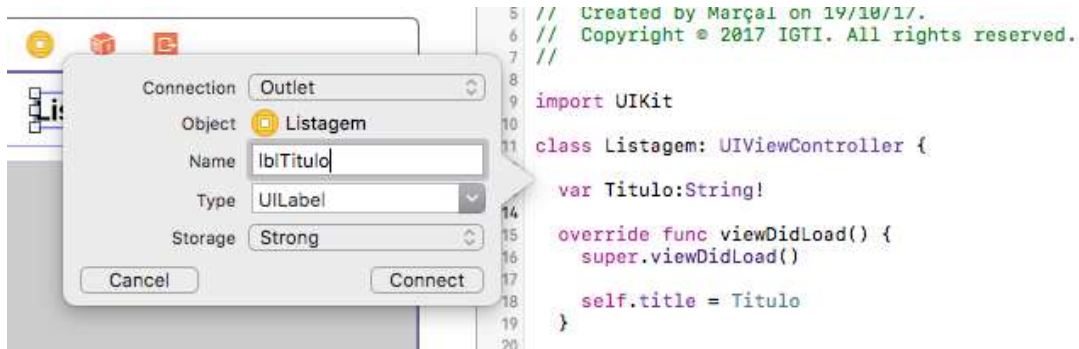
O que fizemos até agora foi mudar a forma de chamarmos nossa tela, agora vamos passar o parâmetro.

Vamos passar o título da tela seguinte que será **Listagem 1**, pois queremos que esse título seja dinâmico, para isso será necessário criamos uma variável na classe **Listagem** para receber esse valor.

Agora precisaremos ligar o nosso label que será o campo que receberá o parâmetro dinamicamente. Veja que a figura 51 é praticamente a mesma coisa quando criamos um método, mas agora vamos criar um objeto para referenciar o nosso label.

Os campos que devem atentar agora são **Connection**, que deve estar setado para **Outlet**, e **Name**, que será o nome para a variável que será criada.

Figura 51 – Criação de variável ligada a um componente visual



Feito isso, o resultado será o seguinte:

```
class Listagem: UIViewController {
```

```
    var Titulo:String!
```

```
    @IBOutlet var lblTitulo: UILabel!
```

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

```
        super.viewDidLoad()
```

```
        lblTitulo.text = Titulo
```

```
    }
```

```
    override func didReceiveMemoryWarning() {
```

```
super.didReceiveMemoryWarning()

// Dispose of any resources that can be recreated.

}

@IBAction func Voltar(_ sender: Any) {

    self.dismiss(animated: true, completion: nil)

}

}
```

Veja que criamos uma variável **Titulo** que receberá o valor da tela de Login e criamos um **@IBOutlet**, isso significa que esta variável está ligada diretamente a um componente visual, no nosso caso é o label título.

Repare que dentro do método **viewDidLoad()** setamos o nosso label **lblTitulo** para receber a variável **Titulo**.

Ao compilarmos veremos como a figura 52.

Figura 52 – Tela com título dinâmico



Obs.: reparem que o título ficou muito próximo à barra de status, deixo isso para vocês adequarem o visual. Dica: precisam apenas posicionar melhor os componentes visuais.

A Classe UINavigationController

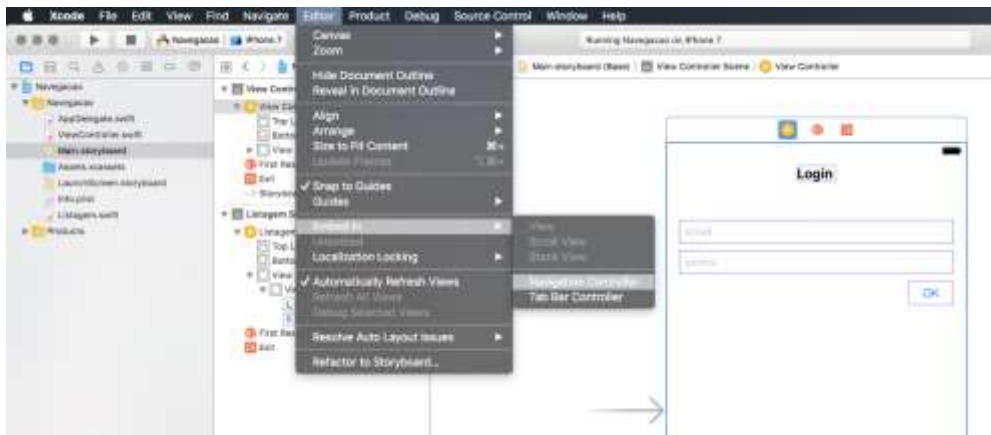
Nesta seção vamos falar da classe **UINavigationController**, que é uma das mais utilizadas no desenvolvimento para iOS, pois facilita bastante a navegação.

A Apple adotou um padrão conhecido como navigation bar, que é a barra de navegação que encontramos no topo dos aplicativos para **iOS**. Para adicionar uma

barra de navegação é só adicionar uma **UINavigationController** como controller principal, isso pode ser observado na figura 53.

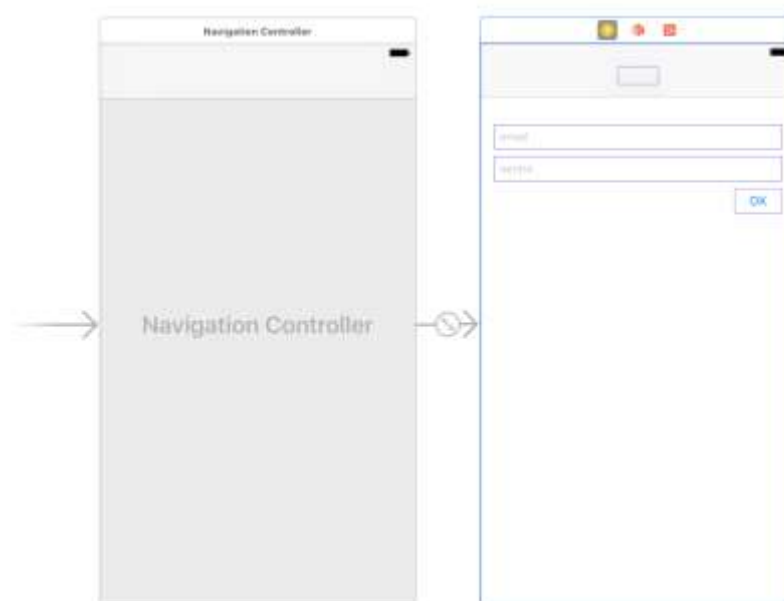
Selecione a tela de Login, vá ao menu **Editor->Embed In->Navigation Controller**.

Figura 53 – Incluindo uma UINavigationController no projeto



Feito isso, suas views ficarão iguais à figura 54.

Figura 54 – UINavigationController criada



Agora vamos trocar nossa classe **ViewController.swift** para chamar a nossa tela Home utilizando um controle de navegação, ou seja, navigation controller.

Feito o indicado na figura 54, teremos que alterar nossa classe.

ViewController.swift

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {

        super.viewDidLoad()

        // Do any additional setup after loading the view, typically from a
nib.

    }

    override func didReceiveMemoryWarning() {

        super.didReceiveMemoryWarning()

        // Dispose of any resources that can be recreated.

    }
```

```

@IBAction func Logar(_ sender: Any) {

    let vc = storyboard?.instantiateViewController(withIdentifier:
>Listagem") as! Listagem

    vc.Titulo = "Listagem 1"

    self.navigationController?.pushViewController(vc, animated:
true)

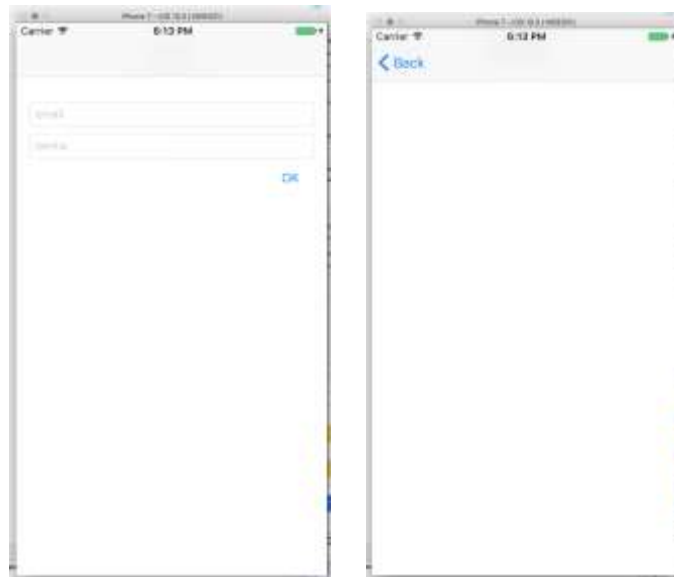
}

}

```

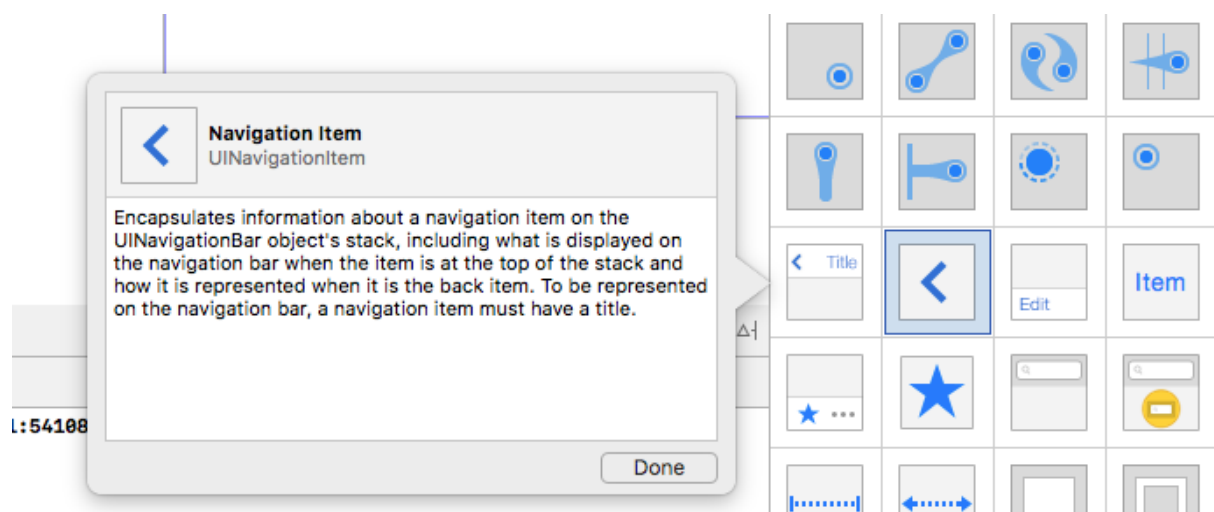
Reparem que utilizamos **self.navigationController** e o método **pushViewController()**. Dessa forma, teremos automaticamente uma barra de navegação superior adicionada, veja figura 55. Como podem reparar, a barra de navegação que construímos manualmente não aparece mais, ela está lá, mas a navigation bar está por cima. Reparem também que foi incluído um botão “back” automaticamente. Vamos melhorar um pouco e colocar respectivos títulos na navigation bar da tela **Login** e **Listagem**.

Figura 55 – Controle de navegação




Vamos usar o componente chamado **Navigation Item** para incluir um item de título na nossa navigation bar. Clique com o botão esquerdo do mouse, arraste e solte em cima da navigation bar.

Figura 56 – Incluindo Navigation Item



A tela de Login ficará como a figura 57.

Figura 57 – Tela de login com Navigation Item

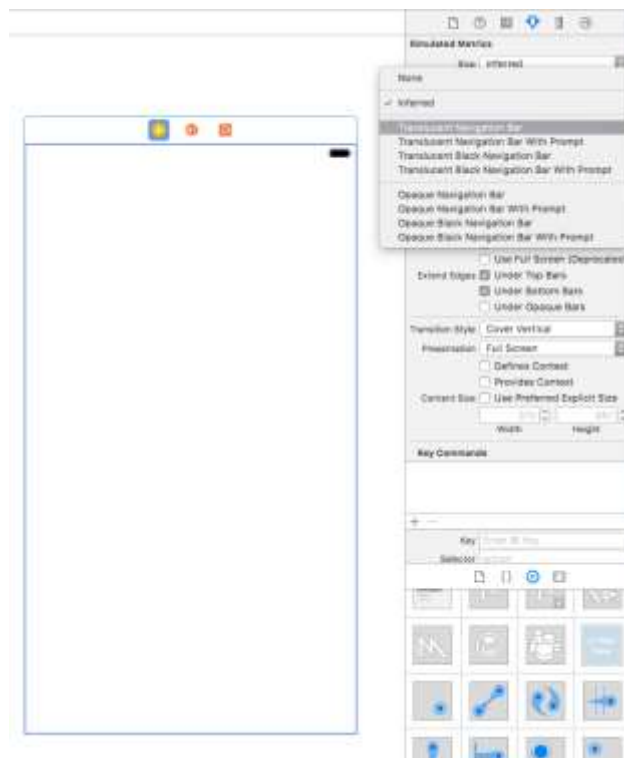


The image shows a mobile application screen with a white background. At the top, there is a light gray navigation bar. Inside the navigation bar, the word "Login" is centered in a dark gray font. Below the navigation bar, there are two text input fields. The first field is labeled "email" and the second is labeled "senha". To the right of the "senha" field, there is a blue button with the text "OK" in white. The entire screen is framed by a thin gray border.

Agora vamos incluir um Navigation Item na tela Home, mas antes precisaremos fazer com que a viewController na storyboard nos exiba visualmente a navigation bar.

Selecione a **viewController** na storyboard, depois acesse a aba **Attributes Inspector**, clique no item **Top Bar** e escolha a opção **Translucent Navigation Bar**, veja detalhes na figura 58.

Figura 58 – Incluindo métrica Navigation Bar para aparecer na storyboard



Feito isso, siga o mesmo processo para incluir o **Navigation Item**, sua viewController deveria ficar como a da figura 59.

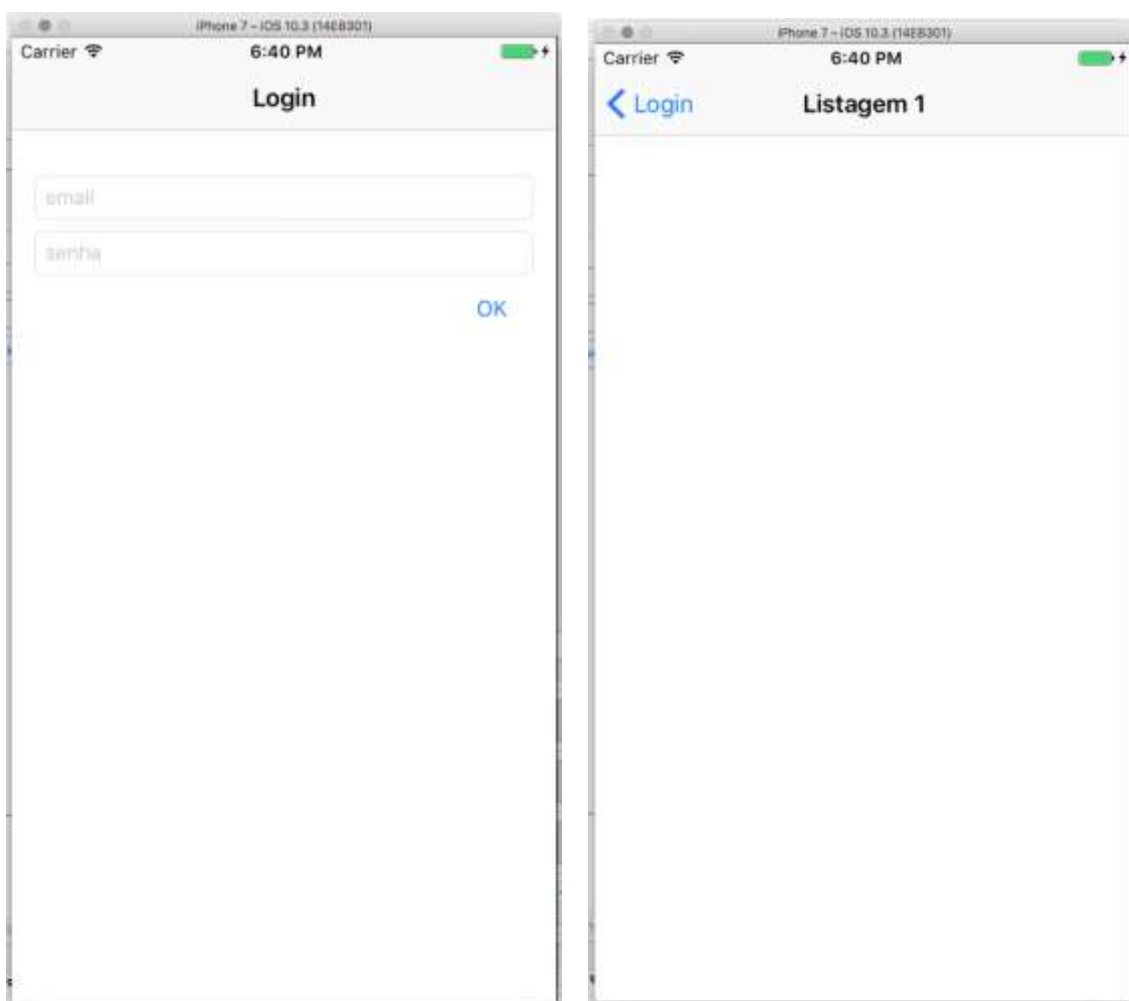
Figura 59 – Tela Listagem com navigation bar incluída



Importante: como usaremos controle de navegação, não precisamos mais do botão voltar que incluímos anteriormente, nem do label para o título, vamos removê-los. Lembre-se também de retirar o código referente a esses campos na classe **Listagem.swift**.

Feito isso, basta compilar. Veja o resultado na figura 60.

Figura 60 – Títulos da Navigation bar incluídos



Importante: reparem que o botão “back” muda automaticamente de acordo com o título da tela anterior, no nosso caso, **Login**.

Como retiramos o botão voltar e campos que não usamos mais da tela **Listagem**, veja como ficou a classe **Listagem.swift**.

Listagem.swift

```
import UIKit

class Listagem: UIViewController {

    var Titulo:String!

    override func viewDidLoad() {

        super.viewDidLoad()

        self.title = Titulo

    }

    ...

}
```

Importante: Veja que para atribuirmos automaticamente o título (do parâmetro que passamos para a tela) usamos **self.title**.

Exceptions

Até o **Swift 1.1**, o tratamento de erros era feito passando um objeto do tipo **NSError** como parâmetro dos métodos, mas a partir do **Swift 2.0** foi criada a sintaxe **try/let/catch**. Uma exceção precisa ser um **enum** que implementa o protocolo **ErrorType**.

“O tratamento de erros é a arte de falhar com graça”

Autor desconhecido

O bom tratamento de erros melhora a experiência tanto para usuário final quanto para desenvolvedores, facilitando a identificação de problemas, suas causas e gravidade associada. O tratamento de erros permite que sistemas falhem de forma adequada para não frustrar ou prejudicar os usuários.

Importante: os exemplos abaixo são apenas ilustrativos, não iremos criar arquivos ou incluir em nosso projeto.

Vejamos um exemplo simples.

Velocidade.swift

```
enum VelocidadeException: Error {  
  
    case rapido  
  
    case muito_rapido  
  
}
```

Os métodos que precisam lançar exceções devem utilizar a palavra reservada **throws** em sua declaração, conforme mostrado abaixo:

Carro.swift

```
class Carro {  
  
    ...  
  
    func acelerarComVelocidade(_ velocidade: Int, distancia: Int) throws {  
  
        print("Acelerar para \ (velocidade) km/h e Distância \ (distancia)  
metros")  
  
        if (velocidade > 120) {  
  
            throw VelocidadeException.rapido  
  
        }  
  
    }  
  
}
```

Ao chamar um método que pode lançar uma exceção, devemos utilizar a sintaxe **try/let/catch** conforme mostrado a seguir. É possível tratar o erro específico informando o tipo desejado (ex.: **VelocidadeException.rapido**) ou apenas declarar a cláusula **catch** sem nenhum tipo para capturar todos os erros.

ViewController.swift

```
func testeMetodoAcelerar() {  
  
    let c = Carro(nome: "Civic", ano: 2014)  
  
    c.acelerarComVelocidade(100, distancia: 500)  
  
    do {  
  
        try c.acelerarComVelocidade(121, distancia: 500)  
  
    } catch VelocidadeException.rapido {  
  
        print("Erro, você esta andando rápido demais")  
  
    } catch VelocidadeException.muito_rapido {  
  
        print("Erro, você está andando muito rápido")  
  
    } catch {  
  
        print("Erro genérico")  
  
    }  
  
}  
  
testeMetodoAcelerar()
```

Ao executar serão exibidas as mensagens abaixo:

Acelerar para: 100km/h e Distancia: 500 metros

Acelerar para: 121km/h e Distancia: 500 metros

Erro, você está andando rápido demais

Sempre que um método lançar uma exceção, é recomendado fazer o tratamento de erros adequado com o **try/let/catch**. Podemos usar uma artimanha bem bacana e que simplifica nosso código, veja a execução abaixo. Repare que tentamos dividir por zero e caiu em nosso **catch**, a outra divisão deu certo e a última retornou nil ao invés de dar crash no app.

Figura 61 - Exceptions

```
16 func testeDividir() {
17     do {
18         let s = try dividir(a: 1, b: 8)
19         print("Resultado: \(s)")
20     } catch {
21         print("Erro ao dividir por Zero")
22     }
23
24     let s2 = try? dividir(a: 10, b: 2)
25     print("Resultado Optional: \(s2)")
26
27     let s3 = try? dividir(a: 10, b: 0)
28     print("Resultado: \(s3)")
29 }
30 testeDividir()
31
```

```
"Erro ao dividir por Zero"
5
"Resultado Optional: Optional(5)"
nil
"Resultado: nil"
```

O código abaixo pode ser executado no arquivo **.playground** que criamos no início da apostila.

```
enum VelocidadeException: Error {

    case rapido

    case muito_rapido

}
```

```
func dividir(a:Int, b:Int) throws -> Int {

    if (b == 0) {

        throw VelocidadeException.muito_rapido

    }
```

```

        return a / b

    }

func testeDividir() {

    do {

        let s = try dividir(a: 1, b: 0)

        print("Resultado: \(s)")

    } catch {

        print("Erro ao dividir por Zero")

    }

    let s2 = try? dividir(a: 10, b: 2)

    print("Resultado Optional: \(s2)")

    let s3 = try? dividir(a: 10, b: 0)

    print("Resultado: \(s3)")

}

testeDividir()

```


Ao usar **try!** não é preciso fazer **try/catch**. Mas devemos usar quando tivermos certeza que o método **dividir()** não irá gerar erro, pois, caso aconteça, o app será fechado ocasionando um crash. Como não queremos isso em nosso código, podemos usar **try?** pois dessa forma não precisamos fazer **try/catch** e caso o método `dividir()` lance uma exceção será retornado **nil** e o app não irá fechar.

Portanto, deve-se utilizar os **opcionais** (“?”, “!”) para evitar a condição de erro onde você espera um valor mas nenhum valor é fornecido. Como programador inteligente, você pode manipular esse recurso para devolver o **nil** intencionalmente em uma condição de erro.

Declaração **guard** é uma maneira rápida de afirmar que uma condição é verdadeira, ou seja, se a condição abaixo for verdadeira, a execução segue normalmente; caso seja falsa, resultará em um erro e parará a execução instantaneamente.

guard

```
let image = info[UIImagePickerControllerOriginalImage] as? UIImage

else {

    return

}
```

Dica: ao invés de usar todo o tratamento de erro que estamos acostumados com o comando **try/catch**, devemos nos atentar para o mecanismos de tratamento de erros simplificado que **Swift** nos proporciona, como **guard**, **throws**, **try** etc.

Obs.: você pode aprofundar um pouco mais sobre tratamento de erros em Swift.

Capítulo 4. O aplicativo Carro

A partir de agora iremos dar início a criação do nosso primeiro app, será um app de listagem de carros e uma tela de detalhe. Também teremos uma tela onde iremos utilizar um componente chamado **Webview** (veremos posteriormente com mais detalhes).

Já vimos vários conceitos até aqui, agora vamos aplicá-los em um app de verdade.

Importante: para quem tem um Macbook, iMac ou Macmini, pode criar o projeto guiado por essa apostila. Na área de downloads existe o projeto completo para baixar, mas recomendo que criem o projeto do zero para exercitar.

Para quem não possui um Mac existe um vídeo na área de downloads que é a demonstração da ferramenta (xCode) e nele eu crio todo o projeto passo a passo para poderem acompanhar com facilidade.

Criando o projeto

Vamos começar criando um novo projeto chamado Carro.

File-New->New Project->Single View Application dê o nome de **CarroSwift**, você pode escolher o nome que quiser.

Vamos agora organizar nossos arquivos conforme fizemos em capítulos anteriores. Lembrem-se de quando criarem um grupo devem vinculá-lo a uma pasta física dentro do projeto.

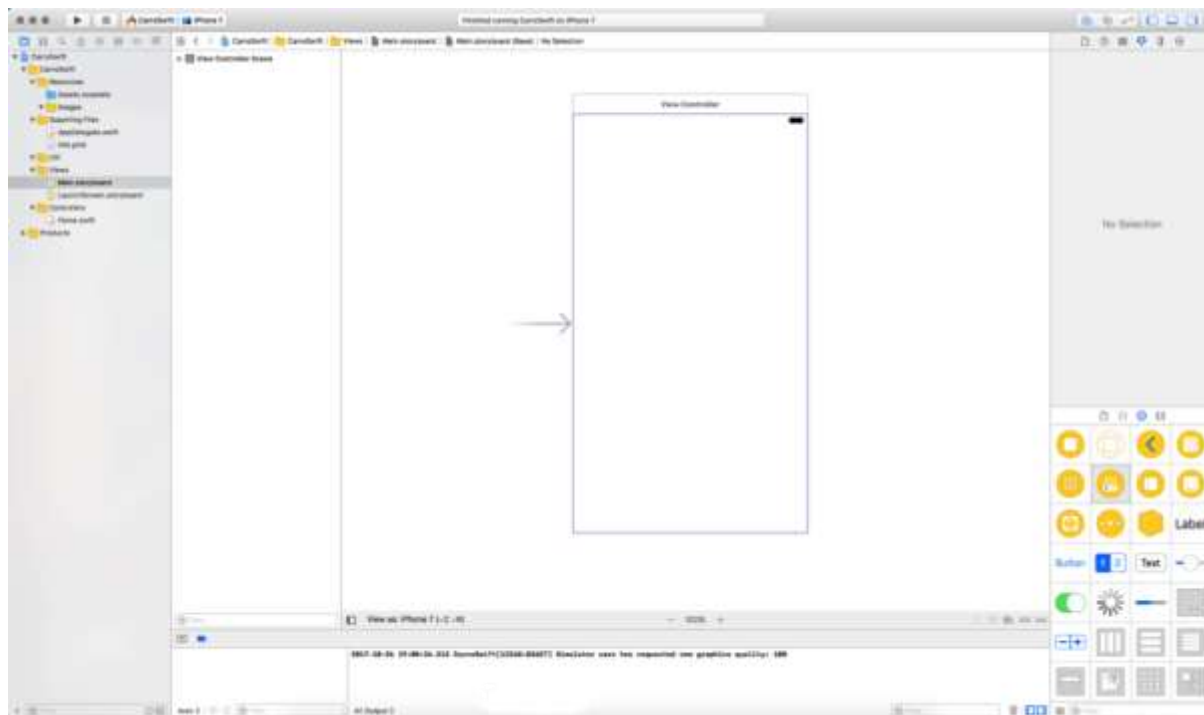
Reparem na figura 62 que eu já troquei o arquivo inicial **ViewController.swift** para **Home.swift** e fiz as devidas ligações (vincular a nova classe **Home.swift** à controller) no arquivo **Main.storyboard**.

Figura 62 – Estrutura inicial do projeto CarroSwift



A figura 63 mostra o estado inicial da nossa **Main.storyboard**.

Figura 63 – Storyboard inicial

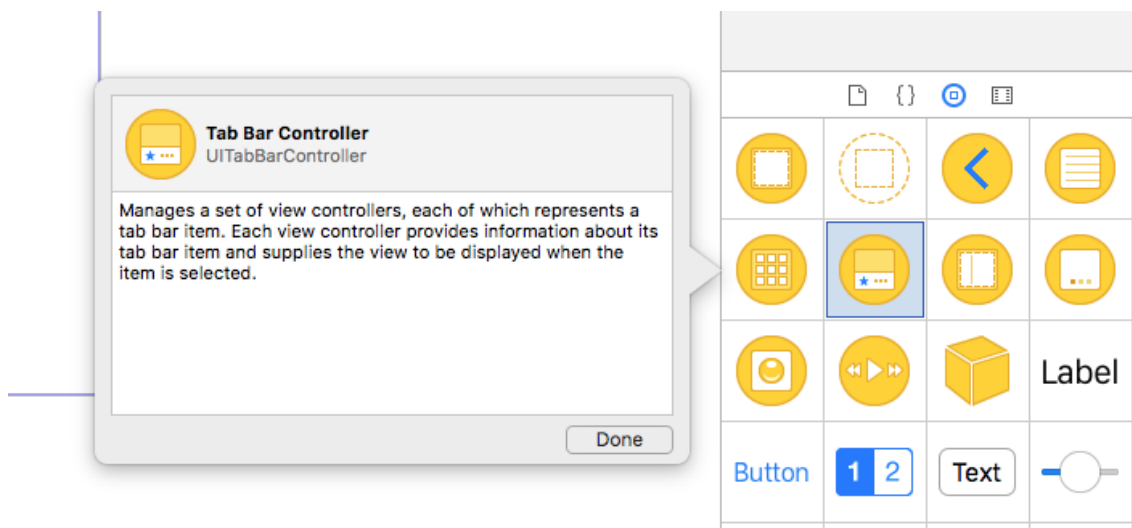


Incrementando nosso projeto

Vamos apagar essa view controller e vamos incluir uma **Tab Bar Controller**, veja na figura 64.

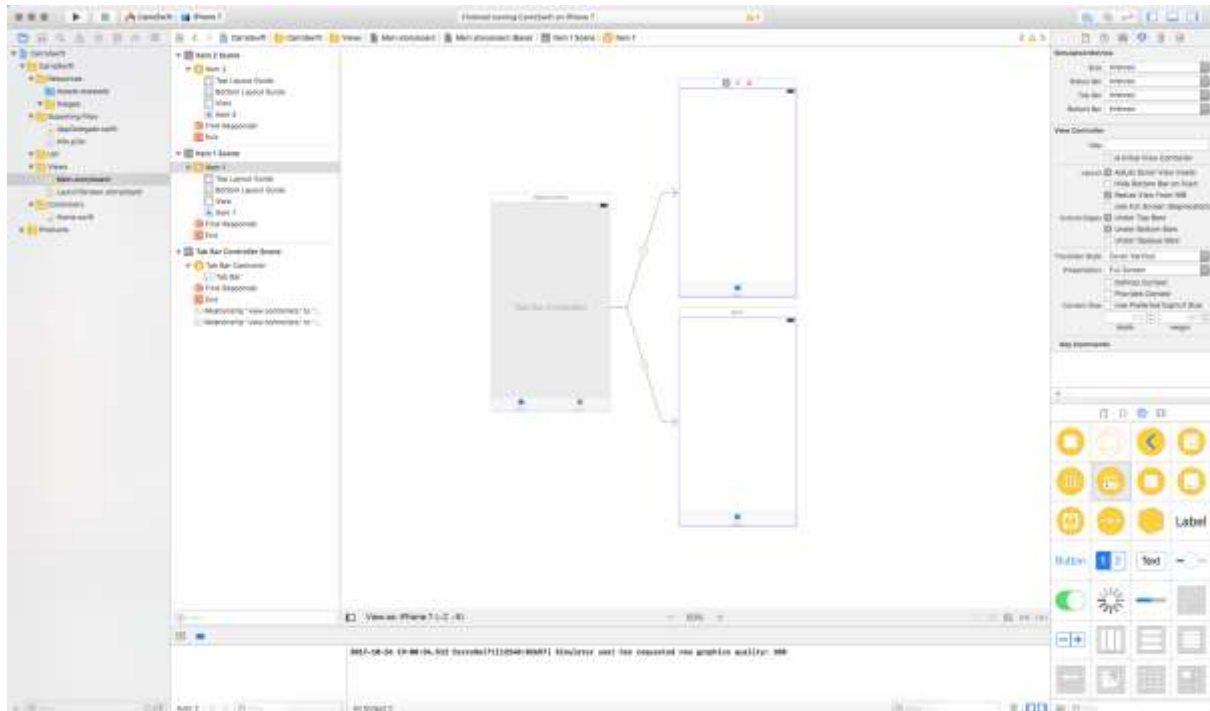
A **TabBarController** é um componente que contém três **ViewControllers**, inicialmente uma do tipo **TabBarController** e outras que são **ViewController**.

Figura 64 – Incluindo TabBarController no projeto



A figura 65 mostra como está nosso projeto com a **TabBarController** incluída.

Figura 65 – TabBarController incluída

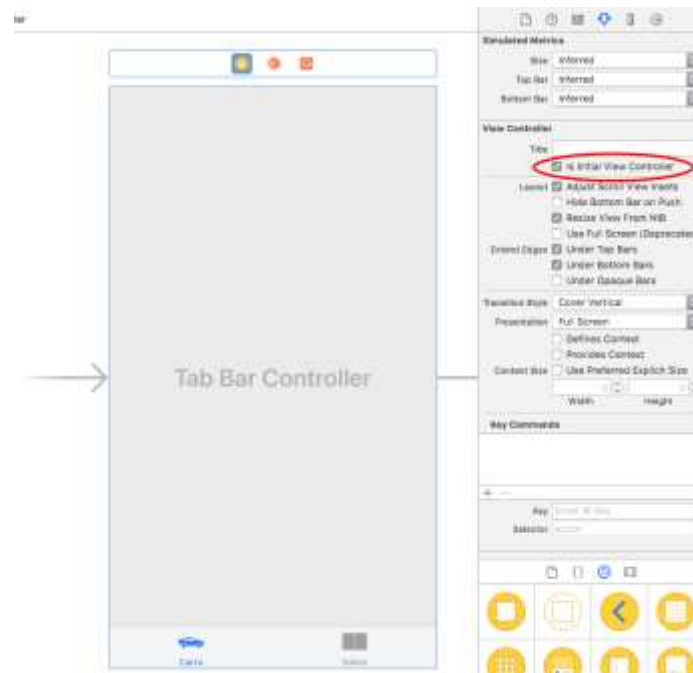


Reparem que quando apagamos a **viewController** (na storyboard), não temos mais uma **viewController** default da storyboard, isso é, uma **viewController** que será sempre a primeira a ser exibida quando chamar a storyboard.

Vamos então setar a nossa **viewController TabBarController** como a **viewController** inicial.

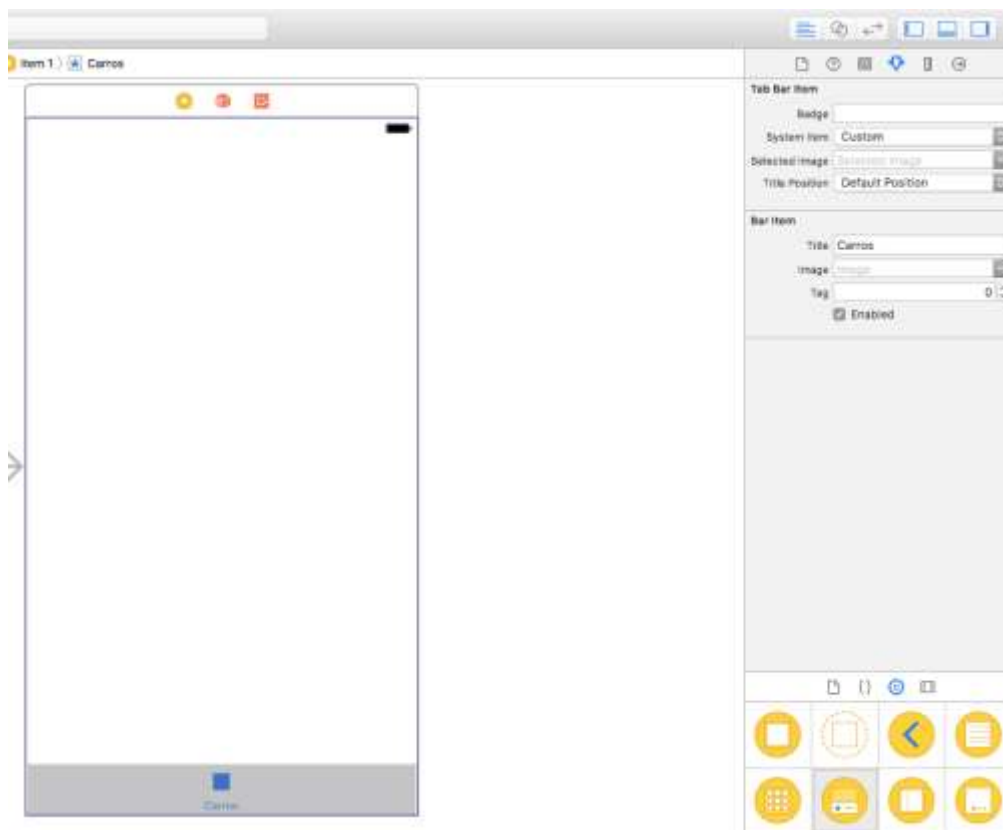
Selecione a view **TabBarController** e vá até a aba lateral chamada **Attributes Inspector**, que é a aba que nos mostra as propriedades de cada componente que selecionamos na área de conteúdo que desenvolvemos, e marque o check **“Is Inicial View Controller”**. Você pode reparar que apareceu uma “seta” apontando para a view, veja figura 66.

Figura 66 – Setando view inicial



Agora, clique duas vezes em cada item para editá-lo ou vá na aba **Attributes Inspector** e coloque a primeira view com o nome de **Carros** e a segunda com o nome de **Sobre**. Vamos também alterar a imagem para uma já predefinida pela IDE, como mostra a figura 67.

Figura 67 – Alterando itens e imagem da tab bar



São seis imagens, três para cada item. As imagens para os projetos são de baixa resolução, alta resolução e altíssima resolução.

As imagens são:

Imagem	Descrição
tab_carros.png	Imagem de baixa resolução
tab_carros@2x.png	Imagem de alta resolução
tab_carros@3x.png	Imagem de altíssima resolução

Para adicionar ao projeto basta selecionar **Assets.xcassets** e arrastar as três imagens, o mesmo vale para as imagens **tab_sobre**.

Feito isso, o resultado deverá ser igual ao da figura 68.

Figura 68 – Incluindo imagens para a TabBar

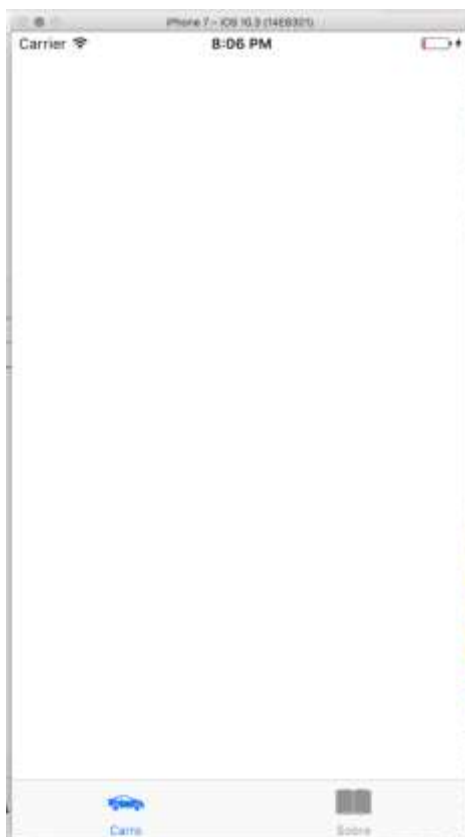


Importante: as imagens são divididas em 1x, 2x, 3x como podem ver pela figura 68. O iOS que define qual imagem usar, em resumo 1x para iPhone 3G e 3GS, 2x para iPhones com tela de retina, a partir do iPhone 4, e 3x para iPhones “plus”.

Ao compilar o projeto, verá algo igual a figura 69.

As imagens são respectivamente o **@2x** o dobro da primeira, **@3x** o triplo da primeira. **Ex.:** se temos uma imagem 100x100 o **@2x** dela será 200x200 e o **@3x** será 300x300.

Figura 69 – Imagens da tabbar definidas

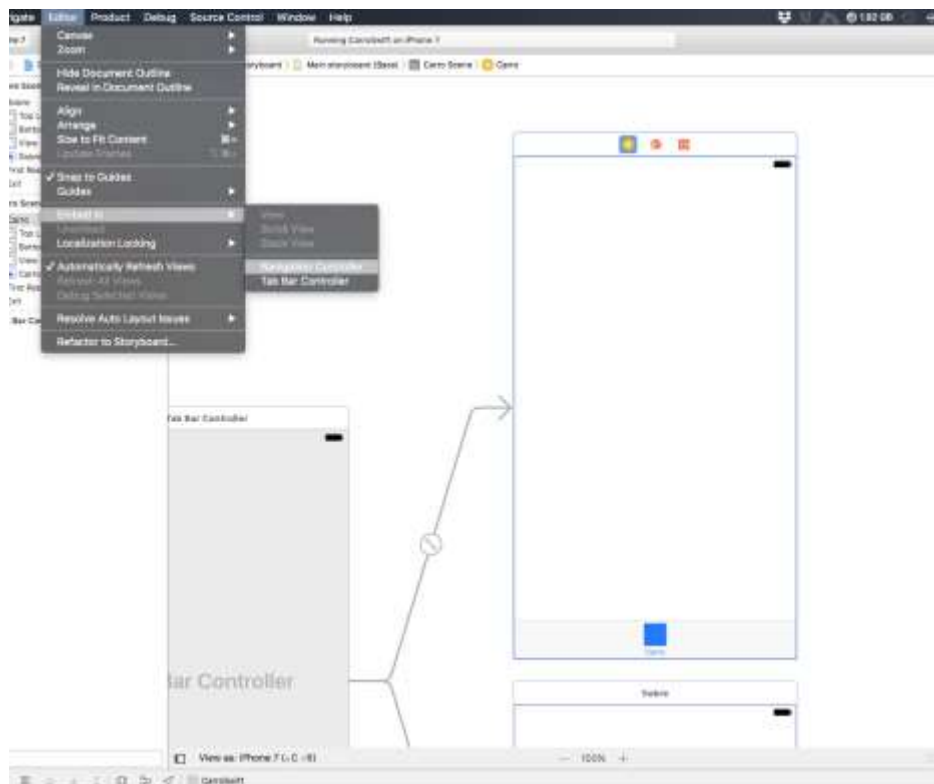


Vamos incluir agora uma **Navigation Controller** para facilitar o controle de navegação do nosso app.

Obs.: lembre-se que já incluímos **Navigation Controller** no capítulo 3.

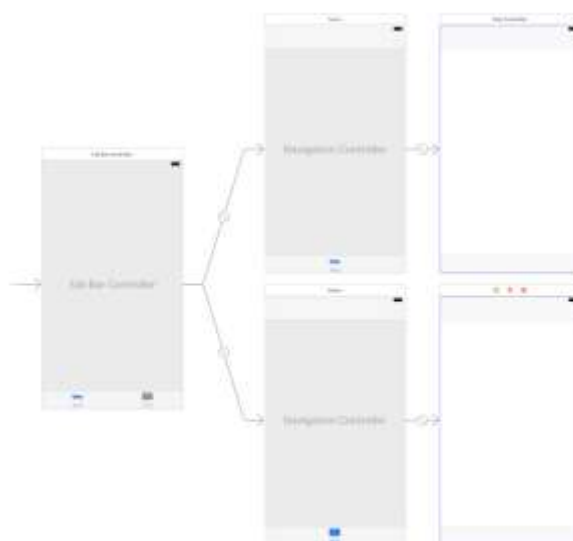
Primeiro iremos selecionar a **viewController** de carro. Feito isso, faça como na figura 70.

Figura 70 – Incluindo Navigation Controller no projeto Carro



Nosso projeto deverá estar conforme a figura 71.

Figura 71 – Navigation Controller incluída



Agora inclua outra **Navigation Controller** na viewController de **Sobre**.

Importante: como a **Navigation Controller** nos dá uma forma automática de lidar com a navegação entre telas, incluímos em cada uma das viewsControllers pelo fato de que elas estão inseridas em uma **TabBar**, ou seja, possuem um menu na parte inferior dessas telas conforme mostra a figura 69. Então, quando chamarmos cada nova viewController a partir da tela **Carro** ou **Sobre** estaremos usando uma instância da **Navigation Controller** e isso nos dará controle de navegação sem perder o **TabBar** inferior.

Podemos aproveitar agora e já incluir itens de título para nossas viewsControllers, já fizemos isso anteriormente. Nossa tela deverá ser como a figura 72

Figura 72 – Inclusão de Navigation Item

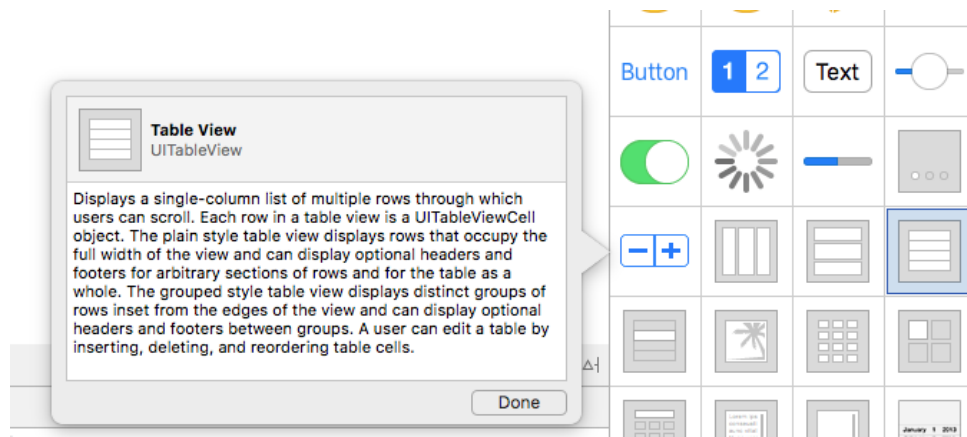


TableviewController

Falaremos um pouco sobre um dos componentes mais famosos e usados do iOS as Tableviews (que são nada mais nada menos que listas), é um componente bastante versátil.

Vamos começar incluindo na viewController **Carro**, veja figura 73.

Figura – 73 – Incluindo Tableview



Ao incluir a tableview na viewController acerte as extremidades até ficar conforme figura 74.

Figura 74 – Tableview incluída



Importante: repare que a tableview está um pouco diferente da que vocês incluíram, pois nesta já aparece as células para podemos alterar visualmente.

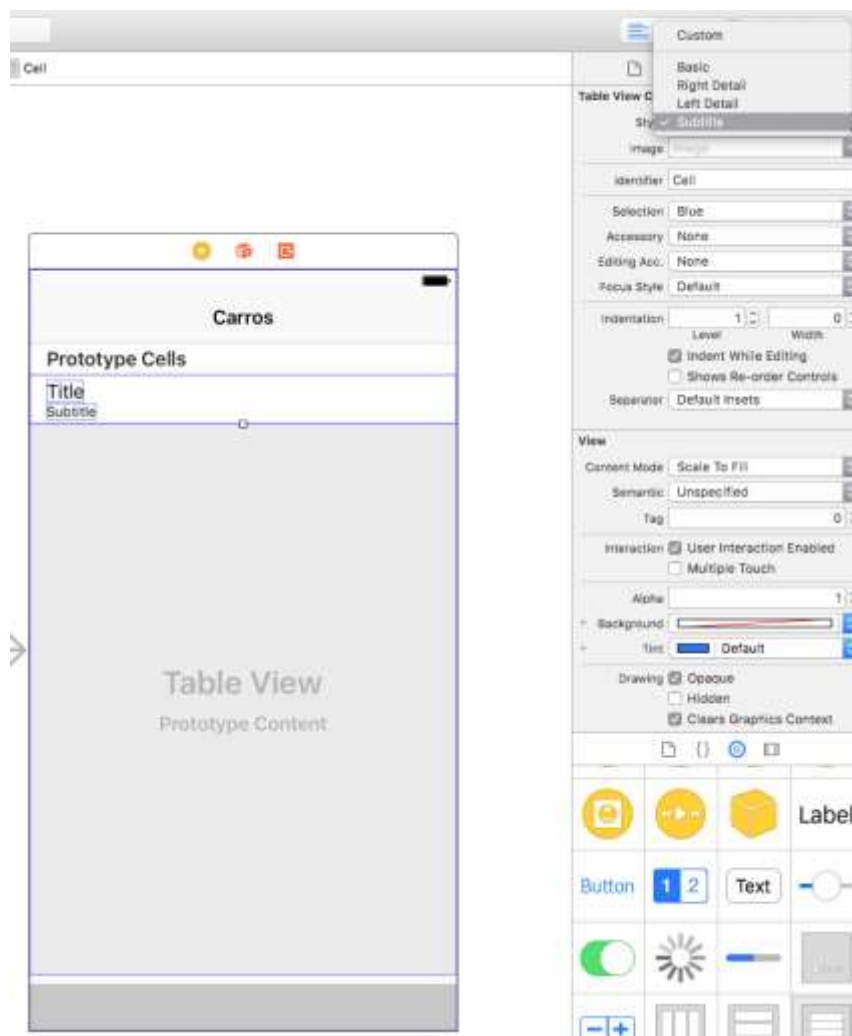
Dica: selecione a tableview e depois vá até a aba **Attributes Inspector** e altere o campo **Content** para **Static Cells** e depois volte para **Dynamic Prototype**

Observe que aparecerá as células de sua tableview. Como precisaremos apenas de uma, vamos apagar as restantes.

Feito isso, vamos começar a configura nossa tableview.

Selecione a célula que sobrou e altere o campo **Style** para **Subtitle**. No campo **Identifier** coloque o texto “**Cell**”, veja figura 75.

Figura 75 – Configurando a tableview



Dica: o campo **Identifier** identifica este tipo de célula para quando formos preenchê-la dinamicamente.

Ao compilar teremos o que é exibido na figura 76.

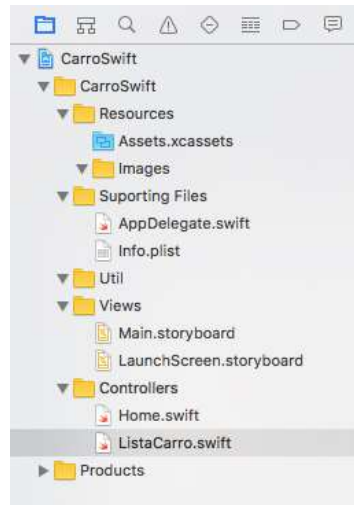
Figura 76 – Tableview incluída



Agora vamos popular de forma rápida nossa **viewController**, vamos utilizar **Array** e **Dictionary**.

Primeiramente vamos criar uma nova classe para nossa **viewController** de carros, criei uma classe chamada **ListaCarro.swift** que será do tipo **Cocoa Touch Class** dentro da pasta **Controllers**.

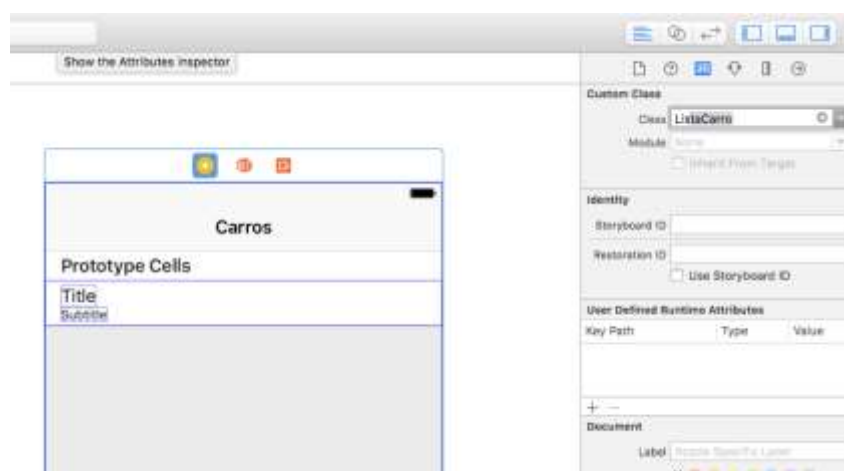
Figura 77 – Classe de listagem de carro incluído



Importante: vamos aproveitar e já vincular nossa viewController visual (storyboard) a classe que acabamos de criar.

Selecione a **viewController** Carros e acesse a aba **Identity Inspector**, veja na figura 78.

Figura 78 – Vinculando classe a uma viewController

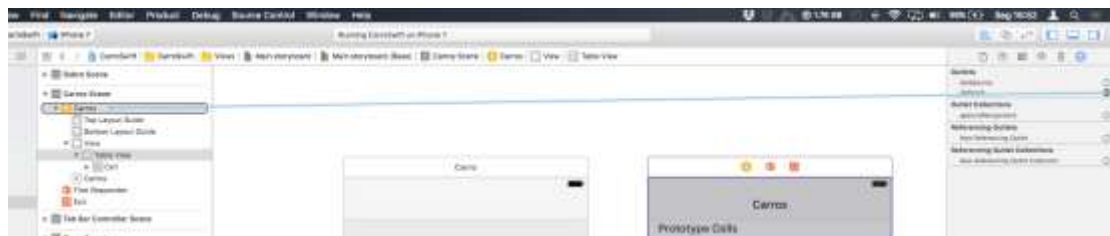


Feito isso, nossa **viewController** está vinculada à classe **ListaCarro.swift**.

Como escolhemos uma célula que não será customizada, vamos simplesmente ligar agora o delegate do nosso componente, pois é este delegate que faz com que os métodos delegados de **UITableView** sejam chamados.

Selecione a tableView na última aba Connections Inspector, clique no sinal de “+” dos campos **delegate** e **dataSource** e arraste até o objeto pai da **viewController**, veja figura 79.

Figura 79 – Ligando visualmente o delegate e o dataSource da tableView



Obs.: faça isso com ambos os campos, um de cada vez.

Agora vamos criar nosso código para popular a tableView.

Como estamos usando um componente visual, tableView e ela responde view delegates precisaremos implementar os métodos obrigatórios destes delegates, para isso vamos alterar a classe que criamos, veja abaixo:

Importante: precisaremos usar um protocolo que é o **UITableViewDataSource**, para saber quais métodos ele possui é só segurar o botão command e com o botão esquerdo do mouse clicar no nome **UITableViewDataSource**.

Figura 80 – Utilizando protocolos da tableview

```

483 public protocol UITableViewDataSource : NSObjectProtocol {
484
485
486     @available(iOS 2.0, *)
487     public func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int
488
489
490     // Row display. Implementers should *always* try to reuse cells by setting each cell's reuseIdentifier and
491     // Cell gets various attributes set automatically based on table (separators) and data source (accessory v
492
493     @available(iOS 2.0, *)
494     public func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell
495
496

```

A figura 80 exibe dois protocolos obrigatórios para a utilização de um tableview, ou seja, precisaremos implementar esses métodos em nossa classe **ListaCarro.swift**.

Será necessário também ligarmos nosso tableview direto à classe, veja na figura 81, daremos o nome de **tView**.

Obs.: Selecione o botão **Assistant Editor**, figura 81, no canto direito da tela, feito isso sua tela se parecerá com a figura 81.1 (certifique-se que esteja na classe correta, às vezes o **xCode** vem com outra classe selecionada, mas é só clicar no botão **Automatic->Automatic** logo acima e escolher a classe que vinculamos à nossa classe).

Feito isso, selecione o tableview e clique com o botão direito do mouse, arraste até a parte de código como mostra a figura 81 e solte, irá aparecer uma tela para dar nome ao seu tableview, veja detalhes juntamente na imagem 81.

Figura 81 – Assistant Editor

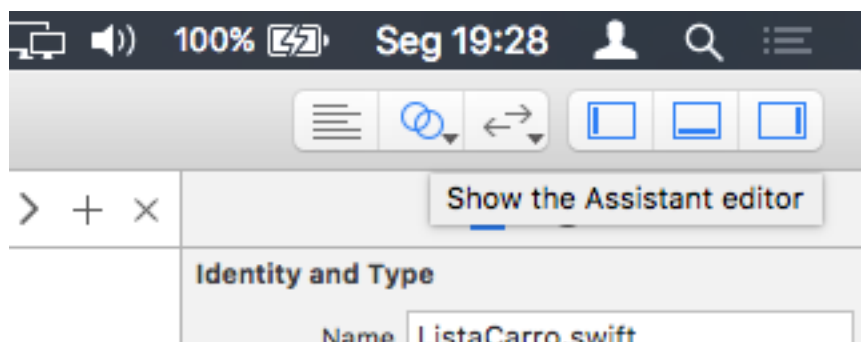
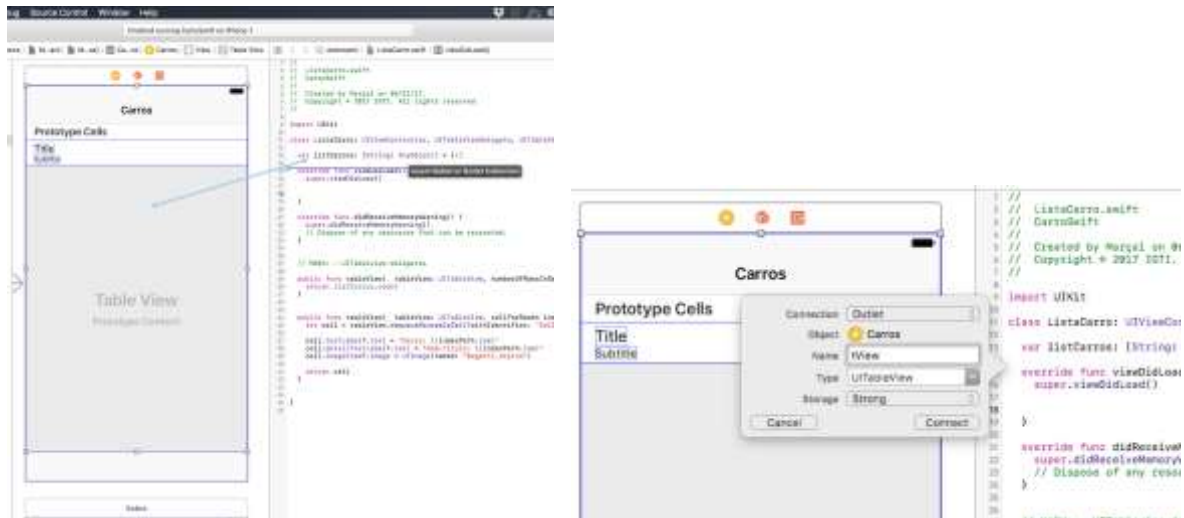


Figura 81.1 – Ligando tableview a sua respectiva classe e criando @IBOutlet para a tableview



ListaCarro.swift

import UIKit

class ListaCarro: UIViewController, UITableViewDataSource {

...

override func viewDidLoad() {

super.viewDidLoad()

tView.register(UITableViewCell.self, forCellReuseIdentifier:

"Cell")

}

// MARK: - UITableView delegates

```
public func tableView(_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
```

```
    return 10
```

```
}
```

```
public func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
```

```
    let cell = UITableViewCell(style: UITableViewCellStyle.subtitle,
reuseIdentifier: "Cell")
```

```
    cell.textLabel?.text = "Carro: \(indexPath.row)"
```

```
    cell.detailTextLabel?.text = "Sub-título: \(indexPath.row)"
```

```
    cell.imageView?.image = UIImage(named: "Bugatti_Veyron")
```

```
    return cell
```

```
}
```

```
}
```

Vamos analisar o código acima.

No método **viewDidLoad()** registramos nosso tableview, veja que utilizamos o nome da célula que demos quando criamos o tableview.

Depois disso, temos a implementação dos métodos obrigatórios do protocolo **UITableViewDataSource**.

O primeiro método retorna o número de linhas que teremos e o segundo é chamado quando cada célula aparecer, pois é nele que preenchemos informações dentro das células.

Neste método instanciamos a célula, mais uma vez utilizando o nome da célula que demos na criação do tableview.

Feito isso usamos as propriedades da célula, **textLabel**, **detailTextLabel** e **imageView**.

Obs.: repare que adicionamos uma imagem ao nosso projeto da mesma forma que adicionamos as imagens da tabBar.

Ao compilar o resultado será o mostrado o mesmo da figura 82.

Figura 82 – Tableview preenchida



Agora vamos melhorar um pouco nossa tableview, vamos criar um array e um dictionary e vamos preencher dinamicamente.

Agora, vamos criar um **Array** e um **Dictionary** para incrementar nossa tableview.

ListaCarro.swift

```
import UIKit
```

```
class ListaCarro: UIViewController, UITableViewDataSource {

    var listCarros = Array<Any>()

    @IBOutlet var tableView: UITableView!

    override func viewDidLoad() {

        super.viewDidLoad()

        tableView.register(UITableViewCell.self, forCellReuseIdentifier:
"Cell")

        self.populaCarros()

    }

    override func didReceiveMemoryWarning() {

        super.didReceiveMemoryWarning()

        // Dispose of any resources that can be recreated.

    }

    func populaCarros() {

        let dic1 = ["Titulo": "Carro 1", "Subtitulo": "Velocidade 200km"]
```

```
let dic2 = ["Titulo":"Carro 2", "Subtitulo":"Velocidade 190km"]
```

```
let dic3 = ["Titulo":"Carro 3", "Subtitulo":"Velocidade 75km"]
```

```
let dic4 = ["Titulo":"Carro 4", "Subtitulo":"Velocidade 74km"]
```

```
let dic5 = ["Titulo":"Carro 5", "Subtitulo":"Velocidade 29km"]
```

```
listCarros = [dic1, dic2, dic3, dic4, dic5]
```

```
}
```

```
// MARK: - UITableView delegates
```

```
public func tableView(_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
```

```
    return listCarros.count
```

```
}
```

```
public func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
```

```
    let cell = UITableViewCell(style: .subtitle, reuseIdentifier: "Cell")
```

```
    let dic = listCarros[indexPath.row] as? [String:Any]
```

```

        cell.textLabel?.text = dic!["Título"] as? String

        cell.detailTextLabel?.text = dic!["Subtítulo"] as? String

        cell.imageView?.image = UIImage(named: "Bugatti_Veyron")

        return cell
    }
}

```

Veja que criamos um array **listCarros** do tipo **Array<Any>**, isso significa que é um array que suporta qualquer tipo, criamos também um método chamado **populaCarros()** para facilitar nossa implementação, pois dentro deles criamos vários **Dicionários** e dentro de cada dicionário **Título** e **Subtítulo**.

Feito isso, substituímos no método delegado **numberOfRowsInSection** o count do nosso array de carros. E, por fim, substituímos as propriedades **textLabel** e **detailTextLabel** da tableview para buscar valor dentro do nosso dicionário.

Note que antes precisamos fazer um **cast** do **listCarros** em cada índice (**indexPath.row**) para então poder setar os valores com segurança.

Cocoapods

Agora vamos buscar os dados dos carros em um servidor (como se tivéssemos acessando um serviço web), não usaremos a forma nativa, vamos usar uma lib de terceiros chamada **Alamofire**¹ e **SwiftyJSON**² para fazer essa

¹ HTTP networking library written in swift - <https://github.com/Alamofire/Alamofire>

requisição, é confiável e de código aberto, logo, se precisar fazer alguma alteração no futuro poderemos fazer tranquilamente.

Para isso vamos utilizar o **cocoapods**³, que é um gerenciador de dependências para iOS.

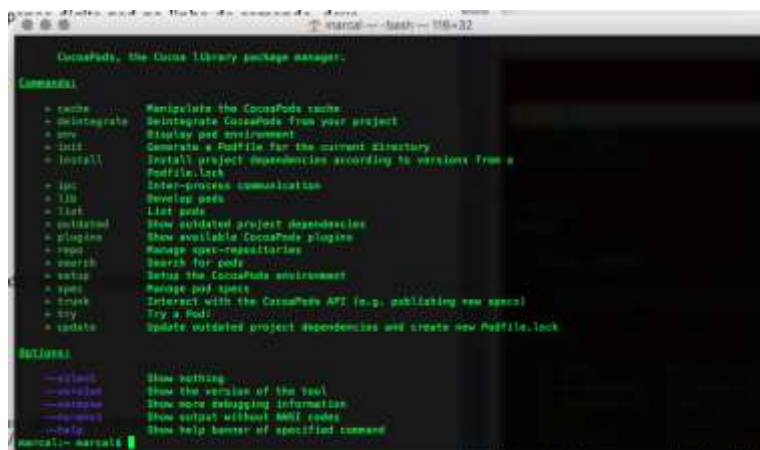
Vamos fazer a instalação do **cocoapods** e depois adicioná-lo ao projeto.

O cocoapods é uma gem (gerenciador de pacotes), vá ao terminal do Mac e digite o comando abaixo:

sudo gem⁴ install cocoapods

Aguarde o fim da instalação e depois apenas digite pod na linha de comando. Ela deve ficar parecida com a figura 83.

Figura 83 – Instalação cocoapods



² Serializa facilmente JSON para ser usado em swift - <https://github.com/SwiftyJSON/SwiftyJSON>

³ Gerenciador de depedências para Swift - <https://cocoapods.org/>

⁴ gem – RubyGem (gerenciador de pacotes para ruby)

Agora que instalamos, vamos adicionar ao nosso projeto, pelo terminal vá até a pasta que está o seu projeto, caso seja uma hierarquia de pastas muito grande pode apenas arrastar a pasta **CarroSwift** do xCode para o terminal e soltar, ficará como abaixo, o comando “**cd**” é para entrar na pasta.

```
cd /PROJECTS/IGTI/CarroSwift
```

Importante: use o comando **ls** para saber se está na pasta que contém o arquivo **CarroSwift.xcodeproj**.

Feito isso é só adicionar o cocoapods com o comando abaixo:

```
pod init
```

Será criado um arquivo chamado **Podfile**. Este arquivo é onde iremos colocar as libs que queremos usar em nosso projeto, no caso, **Alamofire**.

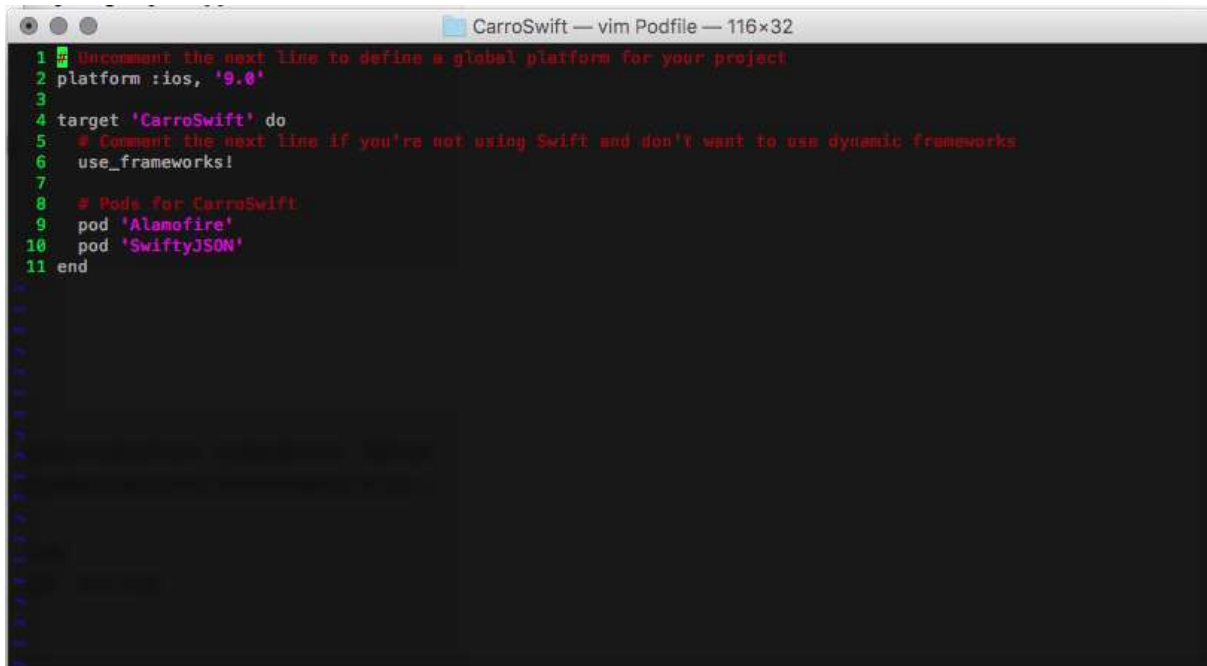
Edite o arquivo **Podfile** e inclua o **Alamofire** como mostra a imagem 84.

Linha a ser incluída é:

```
pod 'Alamofire'
```

```
pod 'SwiftyJSON'
```

Figura 84 – Editando arquivo Podfile



```

1 # Uncomment the next line to define a global platform for your project
2 platform :ios, '9.0'
3
4 target 'CarroSwift' do
5 # Comment the next line if you're not using Swift and don't want to use dynamic frameworks
6 use_frameworks!
7
8 # Pods for CarroSwift
9 pod 'Alamofire'
10 pod 'SwiftJSON'
11 end

```

Feito isso, salve o arquivo e execute o comando abaixo:

pod install

Importante: caso já tenha o cocoapods instalado em sua máquina poderá ser necessário atualizar o repositório do próprio **cocoapods**, utilize o comando abaixo:

pod repo update

Figura 85 – Alamofire instalado

```

CarroSwift --bash-- 116x32
marcal@CarroSwift:~$ vim Podfile
marcal@CarroSwift:~$ pod install
Analyzing dependencies
Downloading dependencies
Installing Alamofire (4.4.0)
Generating Pods project
Integrating client project

[[! Please close any current Xcode sessions and use "CarroSwift.xcworkspace" for this project from now on.
Sending state
Pod installation complete! There is 1 dependency from the Podfile and 1 total pod installed.
marcal@CarroSwift:~$ cd
marcal@CarroSwift:~$ d
total 16
drwxr-xr-x 11 marcal wheel 3748 24 Out 19:51 CarroSwift
drwxr-xr-x  5 marcal wheel 1788  6 Nov 18:45 CarroSwift.xcodeproj
drwxr-xr-x  3 marcal wheel 1028  9 Nov 19:26 CarroSwift.xcworkspace
-rw-r--r--  1 marcal wheel 2748  9 Nov 19:26 Podfile
-rw-r--r--  1 marcal wheel 2888  9 Nov 19:26 Podfile.lock
drwxr-xr-x  8 marcal wheel 2728  9 Nov 19:26 Pods
marcal@CarroSwift:~$ open CarroSwift.xcworkspace/
marcal@CarroSwift:~$ pod install
Analyzing dependencies
Downloading dependencies
Using Alamofire (4.4.0)
Installing SwiftyJSON (3.1.4)
Generating Pods project
Integrating client project
Sending state
Pod installation complete! There are 2 dependencies from the Podfile and 2 total pods installed.

```

Conforme mostra a figura 85, vemos que foi instalado o **Alamofire** e o **SwiftyJSON**.

Se mandarmos listar os arquivos e pastas do nosso diretorio veremos o que é exibido na figura 86.

Figura 86 – Listando arquivos do projeto

```

CarroSwift --bash-- 116x32
marcal@CarroSwift:~$ d
total 16
drwxr-xr-x 11 marcal wheel 3748 24 Out 19:51 CarroSwift
drwxr-xr-x  5 marcal wheel 1788  6 Nov 18:45 CarroSwift.xcodeproj
drwxr-xr-x  3 marcal wheel 1028  9 Nov 19:26 CarroSwift.xcworkspace
-rw-r--r--  1 marcal wheel 2748  9 Nov 19:26 Podfile
-rw-r--r--  1 marcal wheel 2888  9 Nov 19:26 Podfile.lock
drwxr-xr-x  8 marcal wheel 2728  9 Nov 19:26 Pods
marcal@CarroSwift:~$

```

Foram adicionados os seguintes arquivos e pastas:

CarroSwift.xcworkspace

Podfile

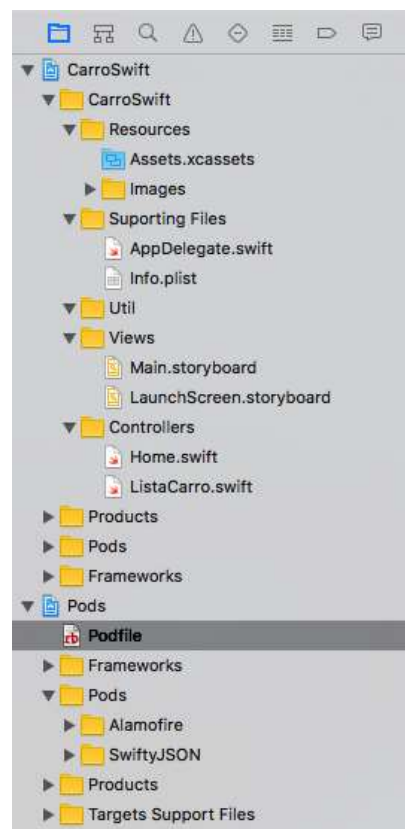
Podfile.lock

Pods

Importante: depois de adicionar o cocoapods em seu projeto, será necessário abrir o projeto sempre pelo arquivo **CarroSwift.xcworkspace**.

A figura 87 exibe como ficará a estrutura do nosso projeto depois de adicionar o **cocoapods**.

Figura 87 – Estrutura do projeto depois de adicionar cocoapods



Foram adicionadas algumas pastas e o projeto **Pods**, que contém as libs que adicionamos ao projeto.

Importante: veja que dentro da pasta **Pod** no projeto **Pods** foi incluída uma pasta para o **Alamofire** e outra para **SwiftyJSON**.

Para incluir mais **Pods** ao projeto é só incluir mais uma linha no arquivo **Podfile** e rodar o comando de instalação. Como não é nosso objetivo, vamos seguir em frente.

A **pod** ou **lib Alamofire** é robusta e mantida por muitos desenvolvedores ao redor do mundo, é bastante seguro utilizá-la em projetos.

Obs.: No cocoapods existem várias libs que podemos usar, como para download e cache de imagem, animação, menus etc. Podem procurar no site do cocoapods e ver quais projetos estão lá. <http://cocoapods.org>.

TableviewController – Parte 2

Agora que vimos sobre cocoapods (o gerenciador de dependência usado no desenvolvimento iOS) vamos voltar à nossa tableview.

Vimos como preencher nossa tableview manualmente e de forma dinâmica usando Array e Dictionary, agora vamos buscar essas informações de um **JSON** que está em algum servidor, ou seja, vamos usar o **Alamofire** e **SwiftyJSON** que incluímos via **cocapods**.

ListaCarro.swift

```
import UIKit
```

```
import Alamofire

import SwiftyJSON

...

func populaCarros() {

    Alamofire.request("http://brunomarc.al.com/servicos/files/carros_luxo.js
on").responseJSON { (responseData) -> Void in

        if((responseData.result.value) != nil) {

            let json = JSON(responseData.result.value!)

            if let resData = json["carros"]["carro"].arrayObject {

                self.listCarros = resData as! [[String:AnyObject]]

            }

            if self.listCarros.count > 0 {

                self.tView.reloadData()

            }

        }

    }

}
```

O **Alamofire** funciona utilizando blocos, veremos posteriormente.

Feito isso, setamos o **JSON** para uma variável e convertemos em um **arrayObject** que é um tipo do **SwiftyJSON** e finalmente setamos para o nosso array **listCarros**.

Importante: se rodarem a **URL** do **JSON** em um navegador poderemos ver o retorno, vide figura 88.

```

1  # Importamos el módulo requests para hacer peticiones HTTP
2  import requests
3
4  # Definimos la URL de la API de OpenWeatherMap
5  url = "https://api.openweathermap.org/data/2.5/weather?q={ciudad}&appid={clave_api}&units={unidades}"
6
7  # Definimos la clave API de OpenWeatherMap (reemplazar con la clave real)
8  clave_api = "1234567890abcdef1234567890abcdef"
9
10 # Definimos la ciudad que queremos consultar (reemplazar con el nombre de la ciudad)
11 ciudad = "Madrid"
12
13 # Definimos las unidades de medida (reemplazar con las unidades deseadas: "metric" para métricas, "imperial" para imperiales)
14 unidades = "metric"
15
16 # Realizamos la petición GET a la API
17 response = requests.get(url.format(ciudad=ciudad, clave_api=clave_api, unidades=unidades))
18
19 # Verificamos si la petición fue exitosa
20 if response.status_code == 200:
21     # Extraemos los datos de la respuesta
22     data = response.json()
23
24     # Mostramos los datos de la ciudad
25     print("Datos de la ciudad de " + ciudad + ":")
26     print("Nombre: " + data["name"] + ", Latitud: " + str(data["lat"]) + ", Longitud: " + str(data["lon"]))
27
28     # Mostramos los datos del tiempo
29     print("Condiciones del tiempo:")
30     print("Temperatura: " + str(data["main"]["temp"]) + " °C, Viento: " + str(data["wind"]["speed"]) + " m/s, Humedad: " + str(data["humidity"]) + "%")
31
32 else:
33     # Si la petición no fue exitosa, mostramos el código de estado y el mensaje de error
34     print("Error al consultar la API. Código de estado: " + str(response.status_code) + ", Mensaje: " + response.text)
35
36 # Cerramos la conexión con la API
37 response.close()

```

Projeto de Aplicações para iOS – Página 152 de 310

ListaCarro.swift

```
import UIKit
```

```
import Alamofire
```

```
import SwiftyJSON
```

```
...
```

```
// MARK: - UITableView delegates
```

```
public func tableView(_ tableView: UITableView, numberOfRowsInSection: Int) -> Int {
```

```
    return listCarros.count
```

```
}
```

```
public func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
```

```
    let cell = UITableViewCell(style: .subtitle, reuseIdentifier: "Cell")
```

```
    var dic = self.listCarros[indexPath.row]
```

```

cell.textLabel?.text = dic["nome"] as? String

cell.detailTextLabel?.text = dic["url_info"] as? String


let url = URL(string: dic["url_foto"] as! String)

let data = try? Data(contentsOf: url!)

cell.imageView?.image = UIImage(data: data!)


return cell
}

```

Veja como ficará nosso método delegado do **Tableview**.

Definimos um dicionário “**dic**” para receber cada índice do array **listCarros** passando o índice de cada célula da tableview com a propriedade **indexPath.row**

Setamos os labels para receber os valores dos respectivos campos, **nome** e **url_info**.

A nossa maior alteração está no campo imagem, setamos uma variável **url**, utilizando o método **URL()** que converterá a nossa string em **URL** e que será usado posteriormente para fazer o download da imagem. Veja que usamos a notação **try?**, por isso precisamos garantir que nossa **URL** seja realmente uma imagem.

Com o método **Data()** fazemos o download propriamente dito e depois utilizamos um construtor da classe **UIImage** para ler os bytes da imagem que fizemos o download para nosso componente **imageView**.

Ao executarmos o projeto, veremos um retorno como o da figura 89.

Figura 89 – Buscando dados de um arquivo JSON online



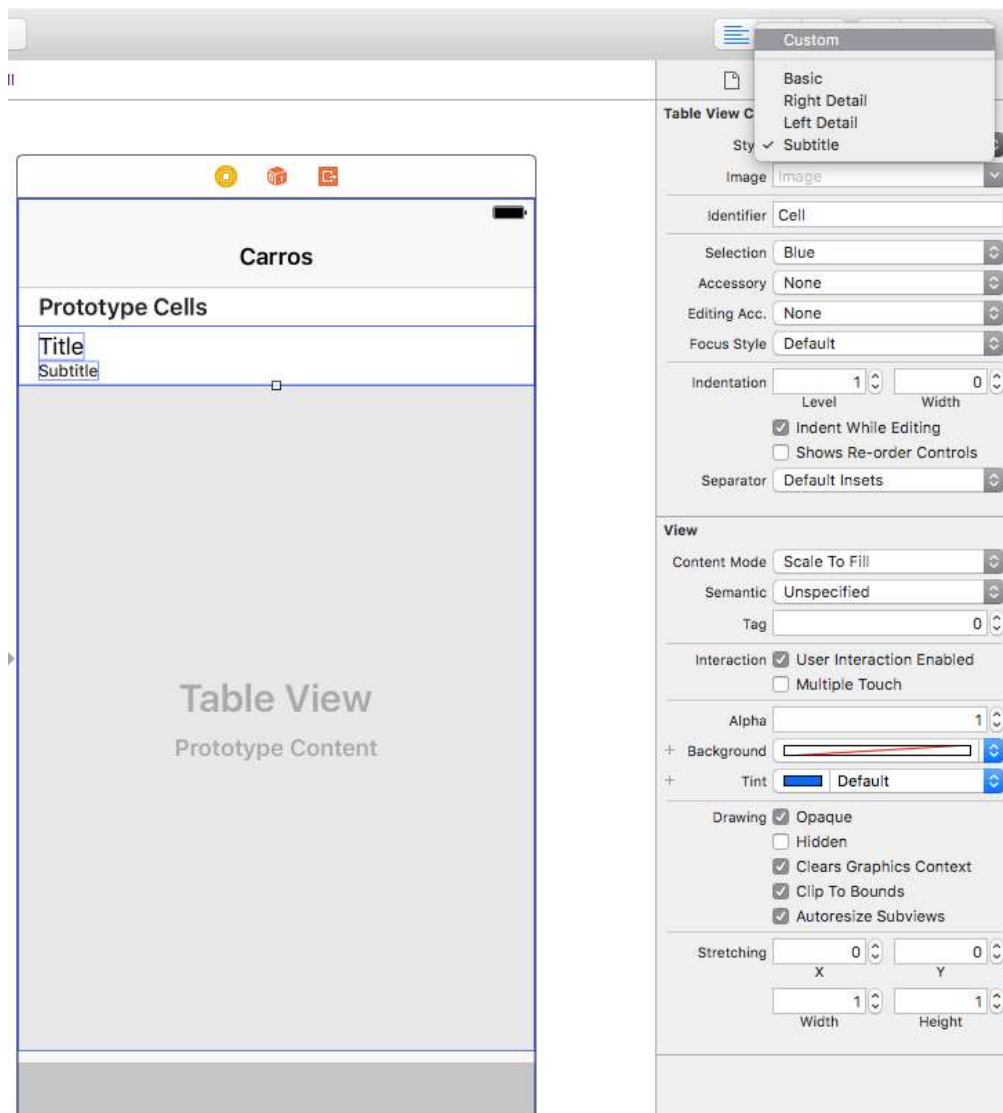
Como podemos notar, nosso código ficou legal, mas não é usual, visto que se tivermos muitos carros as imagens demorarão para aparecer. Vamos melhorar um pouco a forma como buscamos essas imagens, vamos criar um novo arquivo chamado **DownloadImageView.swift** dentro da pasta **Util** e vamos fazer o download das imagens de forma assíncrona. Para isso, teremos também que alterar nossa célula da tableview, pois ela está estática, e vamos fazer uma célula customizada, pois iremos aproveitar e adicionar os campos que temos no nosso JSON.

Dica: no **cocoapods** também existem libs que fazem o download de imagem de forma assíncrona (**SDWebImage** é um exemplo), vale pesquisar sobre isso.

Agora vamos começar por tornar nossa célula customizada, a figura 90 ilustra o que devemos fazer.

Selecione a célula da tableview e defina o campo **Style** com “**Custom**”.

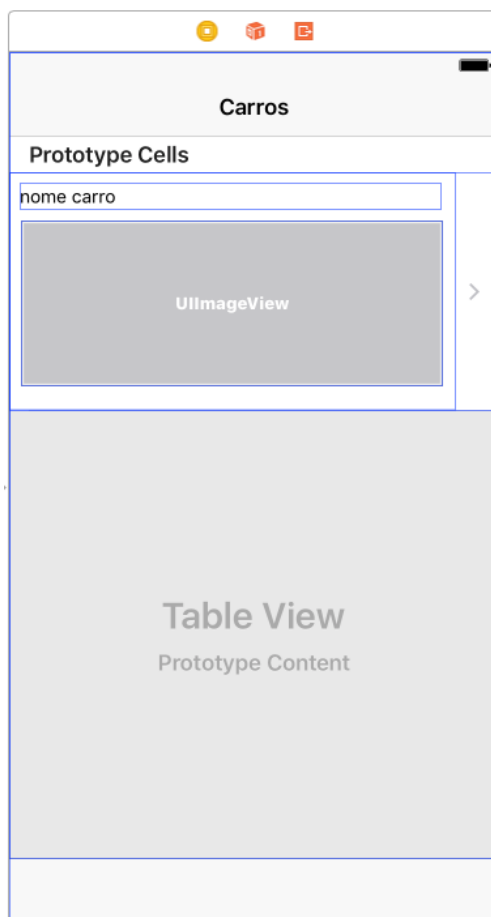
Figura 90 – Alterando tableview para célula customizada



Feito isso, a célula ficará vazia, vamos agora incluir os componentes que precisaremos e aumentar o height da célula.

Inclua os componentes conforme imagem 91, como também será possível clicar em nossa célula vamos setar uma propriedade da tableview chamada **Accessory** que nos indica que teremos uma outra tela de detalhes.

Figura 91 – Incluindo campos dá célula customizada



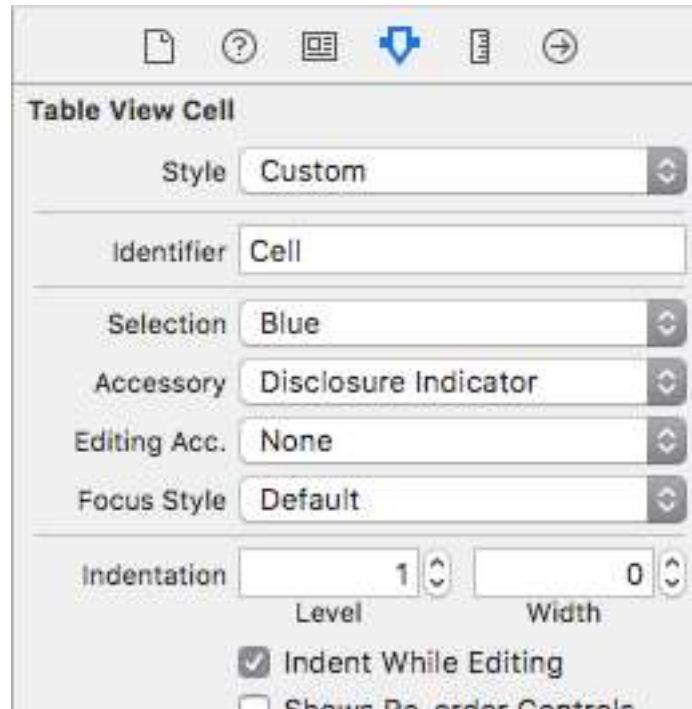
Veja abaixo instruções para incluir os componentes corretamente.

Dica: para deixar os campos dessa forma precisará primeiro aumentar o tamanho da célula e depois acertar os campos, isso pode ser feito manualmente direto na storyboard.

Accessory

Selecione a célula e vá à aba **Attributes Inspector** e altere o campo **Accessory** para **Disclosure Indicator**, veja figura 92.

Figura 92 – Incluindo Accessory indicator

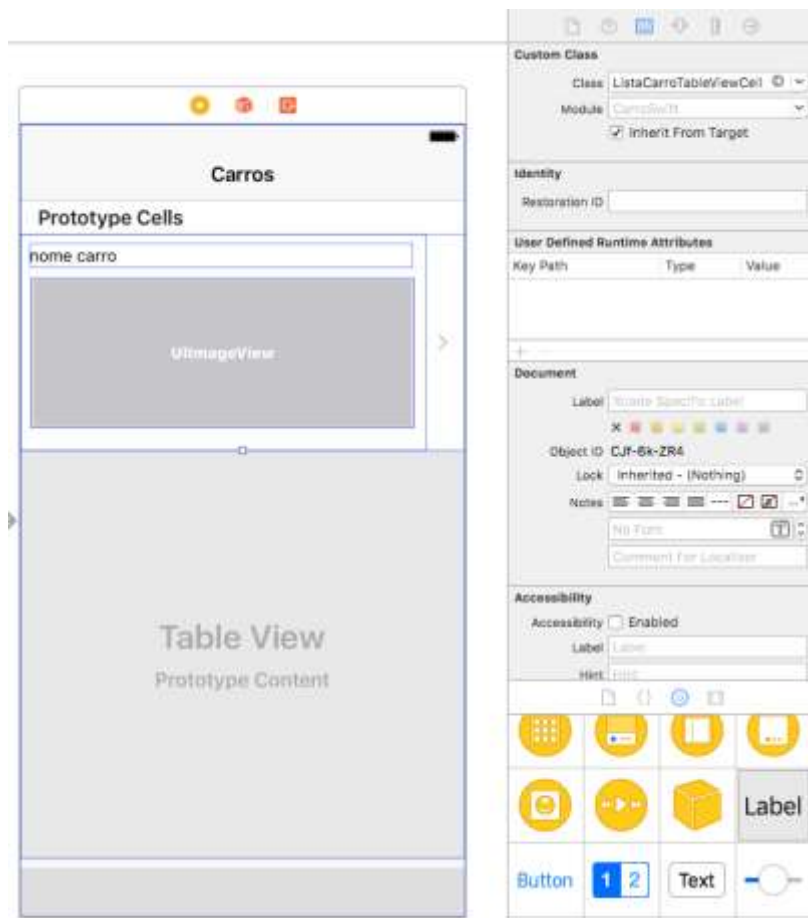


Importante: como nossa célula será customizada, não poderemos usar a classe **UITableViewCell**, precisaremos criar nossa classe para então a customizarmos.

Vamos então criar um novo arquivo dentro da pasta **Controllers** chamado **ListaCarroTableViewCell.swift**, esse arquivo será do tipo **UITableViewCell**.

Feito isso, agora precisamos dizer que nossa célula estará vinculada ao arquivo que acabamos de criar, então vamos selecionar a célula e vamos à aba **Identity Inspector**, e no campo **Class** vamos colocar a nossa classe recém criada. A figura 93 ilustra essa ação.

Figura 93 – Vinculando célula a uma classe



Importante: ao fazermos essa vinculação será possível ligarmos visualmente os componentes que adicionamos em nossa célula à nossa classe.

Selecione a **viewController** e clique no botão **Assistant Editor**, assim verá a storyboard juntamente com o código.

Selecione a imagem dentro da nossa célula e vá em **Automatic** e escolha a nossa classe de célula, veja figura 94.

Agora é só ligar os componentes visuais e dar nomes a eles, **txtNomeCarro** e **imgCarro**.

ListCarroTableViewCell.swift

```
import UIKit

class ListaCarroTableViewCell: UITableViewCell {

    @IBOutlet var txtNomeCarro: UILabel!

    @IBOutlet var imgCarro: DownloadImageView!

    override func awakeFromNib() {

        super.awakeFromNib()

    }

    override func setSelected(_ selected: Bool, animated: Bool) {

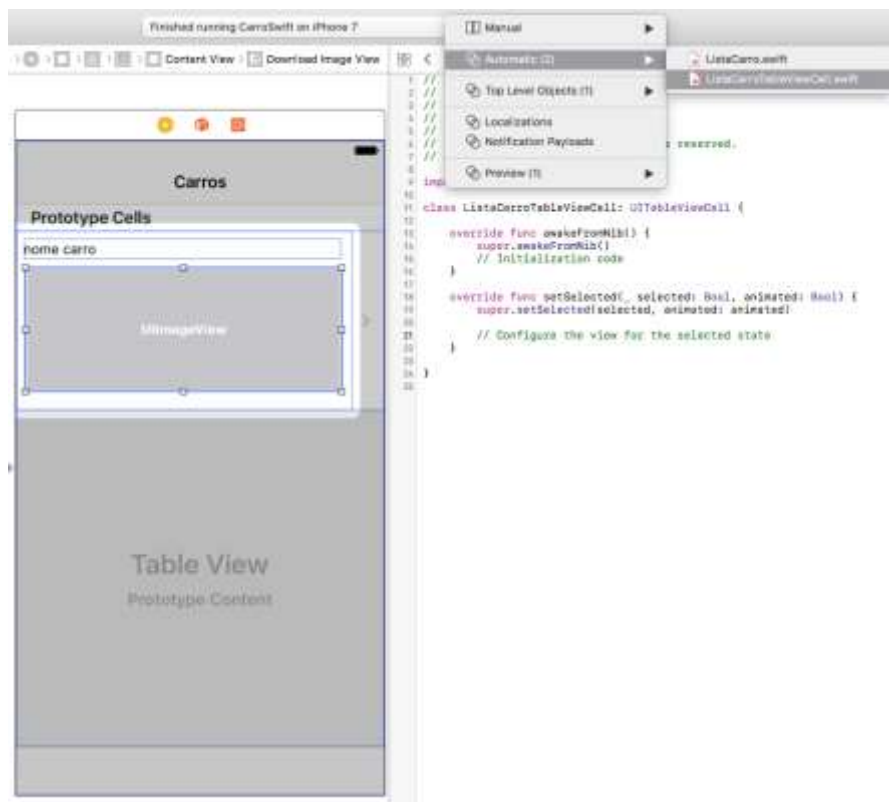
        super.setSelected(selected, animated: animated)

    }

}
```

Obs.: veja que quando ligamos nossa imagem ela já é vinculada ao novo tipo, no caso, **DownloadImageView**.

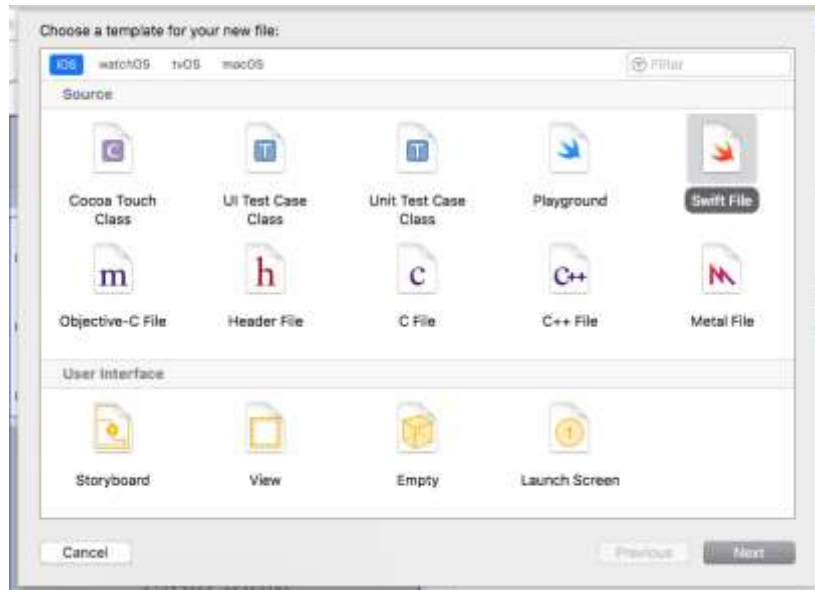
Figura 94 – Vinculando componente visual à nossa classe



Agora precisamos criar nossa classe **DownloadImageView**, crie utilizando o template **Swift File** como mostra figura 95.

Feito isso, podemos agora ligar os componentes e setar os nomes.

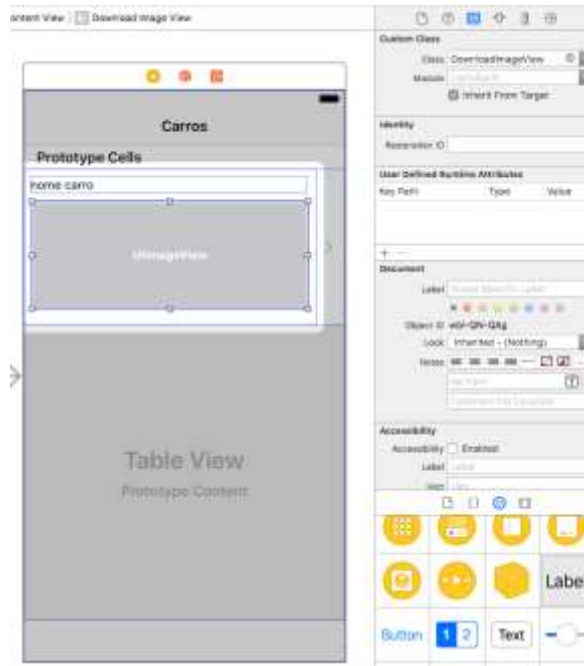
Figura 95 – Criando arquivo com tempate Swift File



Importante: nosso componente de imagem por padrão é do tipo **UIImageView**, mas como vamos alterar seu comportamento precisaremos modificar para o nosso novo tipo **DownloadImageView**.

Igual foi feito com a célula, selecione a imagem e vá até a aba **Identity Inspector** e no campo **Class** coloque a classe **DownloadImageView**, veja figura 96.

Figura 96 – Definindo nova classe para o componente de imagem



Abaixo temos o código completo da classe **DownloadImageView**, vamos dar uma olhada melhor nele.

DownloadImageView.swift

```
import UIKit
```

```
class DownloadImageView : UIImageView {
```

```
// Para exibir a animação durante o download
```

```
var progress: UIActivityIndicatorView!
```

```
let queue = OperationQueue()
```

```
let mainQueue = OperationQueue.main
```

```
required init(coder aDecoder: NSCoder){
```

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

```
    createProgress()
```

```
}
```

```
// Construtor
```

```
override init(frame: CGRect) {
```

```
    super.init(frame: frame)
```

```
    createProgress()
```

```
}
```

```
func createProgress(){
```

```
    progress = UIActivityIndicatorView(activityIndicatorStyle:
UIActivityIndicatorViewStyle.gray)
```

```
    addSubview(progress)
```

```
}
```

```
override func layoutSubviews() {
```

```

        progress.center = convert(self.center,from: self.superview)
    }

```

```

func setUrl(url: String) {

    setUrl(url: url, cache: true)

}

```

```

func setUrl(url: String, cache: Bool){

    self.image = nil

    progress.startAnimating()

    queue.addOperation({self.downloadImg(url: url, cache: true)})

}

```

```

func replace(_ s: String, string: String, withString: String) -> String {

    return s.replacingOccurrences(of: string, with: withString,
options: NSString.CompareOptions.literal, range: nil)

}

```

```

func downloadImg(url: String, cache: Bool) {

    var data: NSData!

    if(!cache) {

```

```

        data = NSData(contentsOf: URL(string: url)!)

    } else {

        var path = replace(url, string: "/", withString: "_")

        path = replace(path, string: "\\", withString: "_")

        path = replace(path, string: ":", withString: "_")

        path = NSHomeDirectory() + "/Documents/" + path

        // Se o arquivo existir no cache

        let exists = FileManager.default.fileExists(atPath: path)

        if (exists) {

            data = NSData(contentsOf: URL(string: url)!)

        } else {

            data = NSData(contentsOf: URL(string: url)!)

            data.write(toFile: path, atomically: true)

        }

    }

    mainQueue.addOperation({self.showImg(data: data)})

}

```

```

func showImg(data: NSData) {

    if(data.length > 0) {

        self.image = UIImage(data: data as Data)

    }

}

```

```
    }  
  
    progress.stopAnimating()  
  
    }  
  
}
```

Nossa classe **DownloadImageView** é usada para baixar imagens em outra thread, reparem que é feito isso utilizando a classe **OperationQueue**. No método **downloadImg()** fazemos o download da imagem propriamente dito, criamos um cache também, ou seja, quando formos acessar a imagem novamente estará baixada no aparelho, não precisamos fazer download novamente e deixando o app mais rápido. Reparem que uma **flag** indicando se vai fazer cache ou não pode ser passada por parâmetro, isso deixa a classe ainda mais útil.

Utilizamos o método **setUrl()** para enviar a URL da imagem que será baixada, isso simplifica em muito a utilização dessa classe em qualquer outro app.

Por fim, utilizamos o método **showImg()** para exibir a imagem baixada, mas somente na thread principal (que é acessada via **OperationQueue.main**), a Apple recomenda atualizar as views ou qualquer componente visual somente na thread principal do app.

Ao compilar verá a tela conforme figura 97.

Figura 97 – Implementado download de imagem em segundo plano



Agora vamos melhorar nossa tela, pois incluiremos mais 2 categorias de carros diferentes. Ela deverá ficar conforme a figura 98.

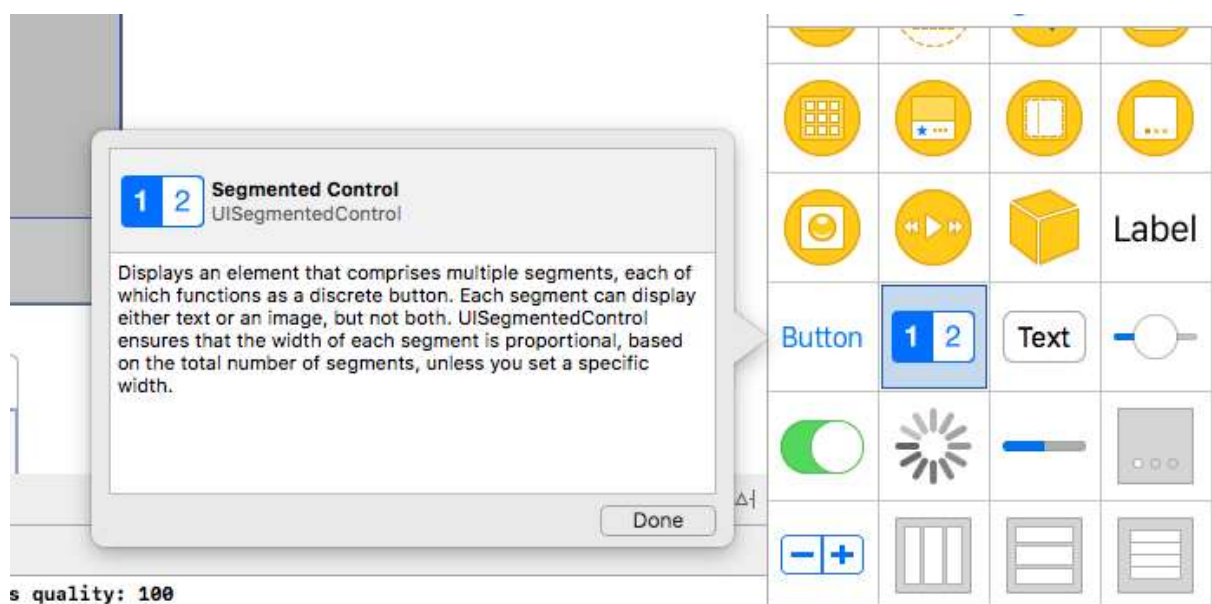
Figura 98 – Inclusão de novas categorias de carros



Importante: logo de início já vemos que incluímos um novo componente visual, ele é um **SegmentedControl**. É bastante usado para simular abas, claro que podemos criar abas de vários forma, mas esta é uma forma bem simples e bastante eficaz.

Diminua um pouco a tableview e depois selecione o componente SegmentedControl conforme mostra a figura 99.

Figura 99 – Incluindo Segmeted Control



Podemos adicionar itens ao **SegmentedControl** tanto de forma visual quanto programaticamente, mas como iremos usar da forma mais simples possível vamos fazer da forma visual.

A figura 100 exibe as propriedades do **SegmentedControl** que precisaremos alterar. Ao selecionar o componente na aba Attributes Inspector irá aparecer as propriedades do componente.

O campo **Segments** indica a quantidade de itens que o **segmentedControl** conterà, aumente para 3 e o campo **Segment** indica qual o item, então selecione

um item no campo **Segment** e altere o campo **Title** dele e verá a alteração aparecer visualmente no componente.

Os itens do segmented serão **Luxo, Sport, Clássicos**. Repare que o primeiro item vem selecionado, deixe dessa forma pois os carros de luxo serão os primeiros a serem exibidos.

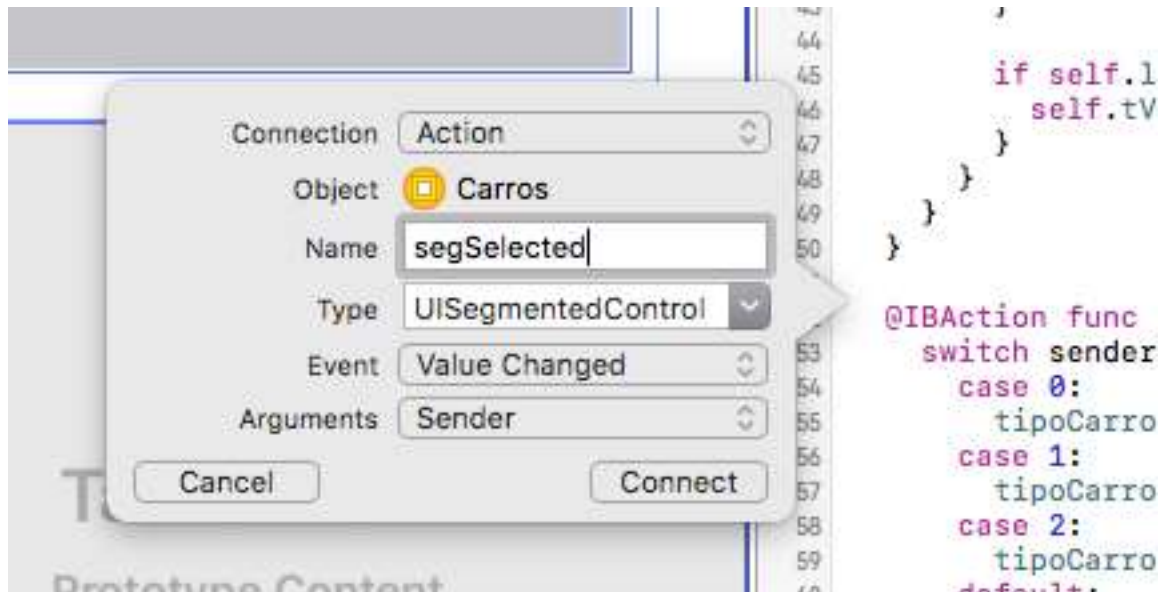
Figura 100 – Propriedades do SegmentedControl



Agora precisamos criar um **@IBAction** para nosso componente, então, selecione o componente e vá no botão **Assistente Editor** (que exibe o componente visual juntamente com o código da classe vinculada a **viewController**).

Feito isso, clique com o botão direito do mouse sobre o componente e arraste para a parte do código e solte, lembrando que devemos criar um **Action**, veja figura 101.

Figura 101 – Criando @IBAction para segmentedControl



Abaixo veja o código do **@IBAction** que acabamos de criar.

```
@IBAction func segSelected(_ sender: UISegmentedControl) {

    switch sender.selectedSegmentIndex {

        case 0:

            tipoCarro = "luxo"

        case 1:

            tipoCarro = "esportivos"

        case 2:

            tipoCarro = "classicos"

        default:
```

```

        break;
    }

    self.populaCarros()
}

```

Obs.: criamos uma variável no contexto da classe chamada **tipoCarro** para poder concatenar seu valor juntamente com a nossa **URL**.

Este método será chamado toda vez que clicar em algum item do **segmentedControl** passando o índice de cada item e após a seleção de qual item chamaremos nosso método **populaCarros()**.

Agora precisaremos alterar o nosso método **populaCarros()**, veja abaixo como ficará:

```

func populaCarros() {

    let url =
"http://brunomarcas.com/servicos/files/carros_{tipo}.json".replacingOccurrences(of:
"{tipo}", with: tipoCarro)

    Alamofire.request(url).responseJSON { (responseData) -> Void in

        if((responseData.result.value) != nil) {

            let json = JSON(responseData.result.value!)

            if let resData = json["carros"]["carro"].arrayObject {

                self.listCarros = resData as! [[String:AnyObject]]
            }
        }
    }
}

```

```

    }

    if self.listCarros.count > 0 {

        self.tView.reloadData()

    }

}

}

}

```

Importante: veja que criamos uma variável local chamada **url** com uma definição **{tipo}** que irá ser trocada de acordo com a seleção do **segmentedControl**.

A saber temos três URLs possíveis:

http://brunomarcas.com/servicos/files/carros_luxo.json

http://brunomarcas.com/servicos/files/carros_esportivos.json

http://brunomarcas.com/servicos/files/carros_classicos.json

Importante: conforme falado anteriormente, criamos uma variável tipoCarro que recebe o valor “**luxo**”, pois será o primeiro tipo que vamos exibir.

...

```

class ListaCarro: UIViewController, UITableViewDelegate,

```

```
UITableViewDataSource {

    var listCarros = [[String:AnyObject]]()

    var tipoCarro:String = "luxo"

    ...
}
```

Ao compilar nosso projeto poderemos selecionar os itens do segmentedControl e veremos os carros serem carregados conforme o tipo, veja figura 102.

Figura 102 – Exibindo carros tipos diferentes de carros



Tela de detalhe dos carros

Agora vamos criar uma tela de detalhe para os carros, com isso vamos ver outros conceitos importantes como passagem de parâmetros para outra **viewController**, exibição de mapas etc.

O primeiro passo que faremos será deixar nossa Tableview clicável e faremos isso usando mais método delegate da tableview.

Eis o método que iremos usar:

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

Ele envia uma instância do tableview que estamos usando e o indexPath que indica a célula que for clicada.

Agora veja o que precisaremos criar:

1. Adicionar **viewController** no arquivo **Main.storyboard**
2. Criar, dentro da pasta **Controllers** um arquivo do tipo **Cocoa Touch Class** e de o nome de **DetailViewController.swift**.
3. Ligar nossa **viewController** à classe **DetailViewController** e atribuir um **StoryboardID** para ela, geralmente usamos o mesmo nome da classe.
4. Incluir componentes (view, imagem, label, textfield 2 botões) visuais e criar os **IBOutlets** e **IBActions** necessários.

Nossa tela deve se parecer com a figura 103.

Figura 103 – Componentes visuais tela de detalhe



Obs.: Tudo isso já foi feito anteriormente, basta seguir como fizemos.

Importante: incluímos todos esses componentes dentro da view que adicionamos, por que fizemos isso? Será respondido posteriormente.

Incluímos um novo componente **UITextView**, geralmente usado quando queremos exibir texto com várias linhas, o funcionamento dele é muito parecido com o **UITextField**.

Reparem também que nossa imagem é do tipo **DownloadImageView**, ou seja, usamos a classe que fizemos para fazer o download e exibir as imagens e vamos usá-la novamente só para exibir, como baixarmos as imagens e gravamos no cache poderemos usar as imagens deste cache deixando assim o app mais rápido.

Com as ligações todas feitas, o código abaixo exhibe como deve estar nossa classe de detalhe com seus respectivos **IBOutlets** e **IBActions** criados.

DetailViewController.swift

```
import UIKit

class DetailViewController: UIViewController {

    @IBOutlet var titleLabel: UILabel!

    @IBOutlet var imgCarro: DownloadImageView!

    @IBOutlet var descriptionTextView: UITextView!

    @IBOutlet var btMapa: UIButton!

    @IBOutlet var btVideo: UIButton!

    override func viewDidLoad() {

        super.viewDidLoad()

        // Do any additional setup after loading the view.

    }

    override func didReceiveMemoryWarning() {

        super.didReceiveMemoryWarning()

        // Dispose of any resources that can be recreated.

    }

}
```

```
}
```

```
@IBAction func showMap(_ sender: UIButton) {
```

```
}
```

```
@IBAction func showVideo(_ sender: UIButton) {
```

```
}
```

```
}
```

Reparem que faltou um pequeno detalhe, como iremos receber os valores que virão da outra tela? Para isso precisarmos criar outra variável.

Vamos, então, criar nossa variável do tipo carro, ela será parecida com a `listCarros` que criamos só que não será um array, pois só receberemos os dados de um carro.

```
var detalheCarro = [String:AnyObject]()
```

Em resumo, nossa variável **detalheCarro** é um dicionário.

Vamos agora chamar nossa tela de detalhes.

ListaCarro.swift

```
...

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

        let detail = self.storyboard?.instantiateViewController(withIdentifier:
"DetailViewController") as! DetailViewController

        detail.detalheCarro = self.listCarros[indexPath.row]

        navigationController?.pushViewController(detail, animated: true)

    }

    ...
```

Veja que criamos um constante **detail** que recebe uma instância de **viewController**, ao final fazemos o **cast** para a **DetailViewController**, dessa forma podemos passar os dados para a variável que criamos, veja que usamos **indexPath.row**, ou seja, passamos apenas as informações da célula que foi clicada.

E por fim chamamos nossa tela usando o método **pushViewController**.

Agora só falta fazer com que nossos campos da tela de detalhe recebam os dados enviados.

DetailViewController.swift

```
import UIKit
```

```
class DetailViewController: UIViewController {
```

```
    @IBOutlet var titleLabel: UILabel!
```

```
    @IBOutlet var imgCarro: DownloadImageView!
```

```
    @IBOutlet var descriptionTextView: UITextView!
```

```
    @IBOutlet var btMapa: UIButton!
```

```
    @IBOutlet var btVideo: UIButton!
```

```
    var detalheCarro = [String:AnyObject]()
```

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

```
        super.viewDidLoad()
```

```
        self.populaCampos()
```

```
    }
```

```
    override func didReceiveMemoryWarning() {
```

```
        super.didReceiveMemoryWarning()
```

```
        // Dispose of any resources that can be recreated.
```

```
}
```

```
func populaCampos() {  
  
    titleLabel.text = detalheCarro["nome"] as? String  
  
    descriptionTextView.text = detalheCarro["desc"] as? String  
  
    imgCarro.setUrl(url: (detalheCarro["url_foto"] as? String)!)  
  
}
```

```
@IBAction func showMap(_ sender: UIButton) {  
  
  
  
}
```

```
@IBAction func showVideo(_ sender: UIButton) {  
  
  
  
}  
  
}
```

Analisando o código acima, criamos um método chamado **populaCampos()** e atribuímos aos nossos componentes visuais os dados que passamos para a variável **detalheCarro**.

Importante: veja que para a imagem usamos uma notação um pouco diferente, usamos o operador “?”, pois esse valor pode vir nulo e usamos o operador “!” para fazer o **unwrapping** da variável.

Por fim, dentro do método **viewDidLoad()**, chamamos o nosso método **populaCampos()**.

Ao compilarmos nosso projeto, nossa viewController deve estar igual à da figura 104.

Figura 104 – Campos preenchidos na tela de detalhe do carro



Na nossa URL de carros “**esportivos**” temos alguns campos diferentes no JSON, vejam eles na figura 105.

Figura 105 – Novos campos no JSON

```
- carro: {
  {
    nome: "Ferrari FF",
    desc: "A Ferrari FF acaba de ser revelada. Se trata do primeiro modelo da marca a ter tração integral, daquele que gosta de percorrer caminhos mais difíceis que exigem tração integral. Este modelo revoluciona a área de freios de cerâmica da Brembo.",
    url_info: "http://www.ferrari.com/English/GT_Sport%20Cars/CurrentRange/FF/Pages/FF.aspx",
    url_foto: "http://brunomarcial.com/servicos/files/images/Ferrari_FF.png",
    url_video: "http://www.youtube.com/watch?v=IqxOYqK2SVg",
    latitude: "44.532218",
    longitude: "10.864019"
  }
}
```

Vamos usar estes campos para exibir um vídeo promocional do carro e vamos usar a latitude e longitude para localizar no mapa onde é a fábrica deste carro, no caso a Ferrari.

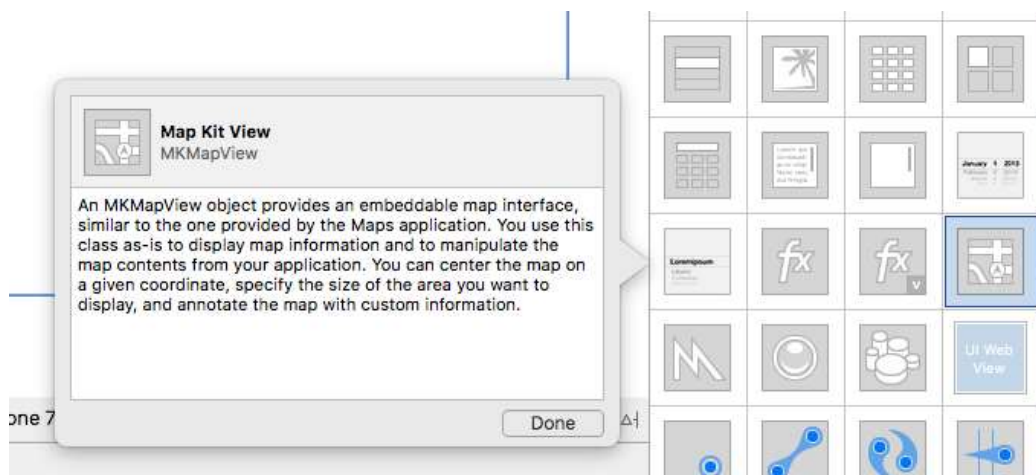
Usando Mapkit e UIWebView

Neste capítulo vamos conhecer um pouco sobre o **MapKit** (lib de mapas da Apple) e sobre o **MPVideoPlayer** (lib de vídeos da Apple).

Vamos começar pelo **Mapkit**.

Vamos começar criando uma **viewController** no arquivo **Main.storyboard** e criando também uma classe dentro da pasta **Controllers** chamada **MapViewController.swift**. Feito isso, crie um **@IBOutlet** para o mapa chamado **mapView**. Veja o componente na figura 106.

Figura 106 – Componente MapView



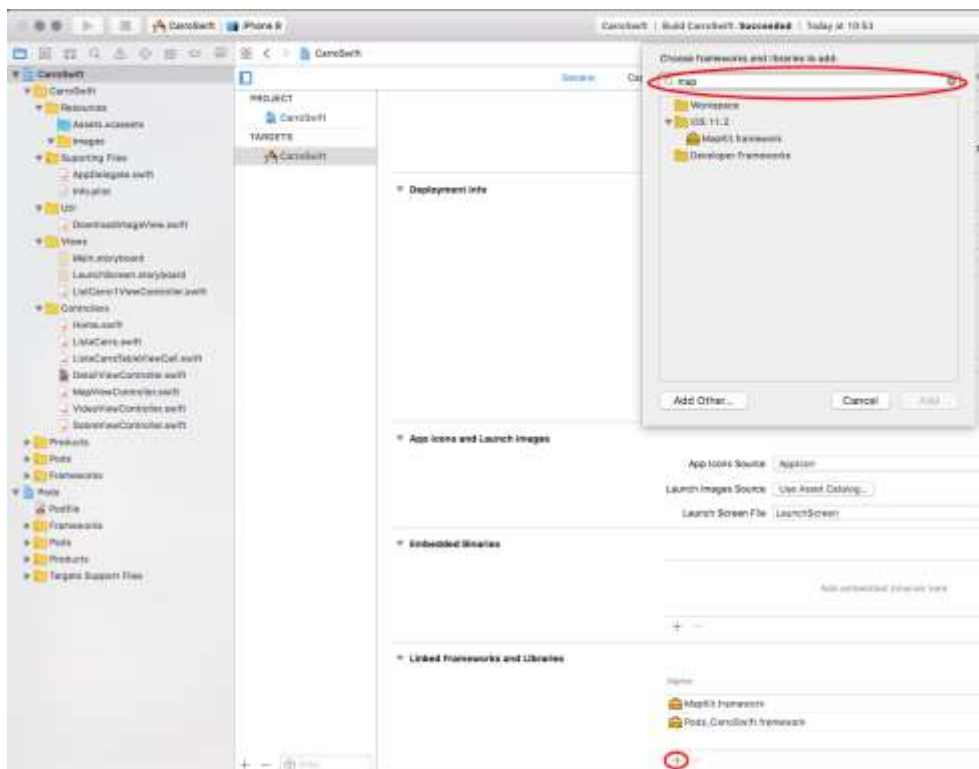
Depois de adicionar o componente de mapa, sua viewController deve estar parecendo com a figura 107.

Figura 107 – ViewController com MapView incluído



Agora vamos incluir o framework do Mapkit em nosso projeto, veja figura 108.

Figura 108 – Incluindo framework Mapkit



Selecione o projeto e clique no sinal de “+” na seção **Linked Frameworks and Libraries**. Feito isso irá aparecer uma janela para você escolher o framework que quer incluir, apenas digite “map” no campo de busca que irá aparecer o **MapKit.framework**, selecione-o e clique no botão “Add”.

Veja na figura 109 que o framework foi adicionado ao projeto.

Figura 109 – MapKit incluído no projeto



Agora estamos prontos para começar nosso desenvolvimento, ou seja, chamar a **viewController** de mapa e passar os parâmetros necessários e exibir no mapa, **latitude**, **longitude** e um **título** para o ponto que será o nome do carro.

Dica: Um pequeno detalhe é que precisaremos criar um **@IBAction** para o botão do mapa na **DetailViewController.swift**.

DetailViewController.swift

```
@IBAction func showMap(_ sender: UIButton) {  
  
    let map = self.storyboard?.instantiateViewController(withIdentifier:  
"MapViewController") as! MapViewController  
  
    map.lat = self.detalheCarro["latitude"] as? String  
  
    map.lon = self.detalheCarro["longitude"] as? String  
  
    map.titulo = self.detalheCarro["nome"] as? String  
  
    navigationController?.pushViewController(map, animated: true)  
  
}
```

Alteramos apenas o método **showMap()** para passar lat, lon e título para o mapa, agora na classe **MapViewController.swift** precisaremos criar esses parâmetros.

O que fazemos no método **showMap()** é instanciar uma variável com a **viewController** de mapa, passar os parâmetros e chamar o método **pushViewController** para então chamar a tela de mapa.

MapViewController.swift

```
import UIKit
```

```
import MapKit
```

```
class MapViewController: UIViewController {
```

```
    @IBOutlet var mapView: MKMapView!
```

```
    let locationManager = CLLocationManager()
```

```
    var lat: String?
```

```
    var lon: String?
```

```
    var titulo: String?
```

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

```
        super.viewDidLoad()
```

```

        self.exibeMapa()
    }

    override func didReceiveMemoryWarning() {

        super.didReceiveMemoryWarning()

        // Dispose of any resources that can be recreated.
    }

    func exibMapa() {

        self.locationManager.desiredAccuracy =
kCLLocationAccuracyBest

        self.locationManager.requestWhenInUseAuthorization()

        self.locationManager.startUpdatingLocation()

        self.mapView.showsUserLocation = true

        let Lat = (lat! as NSString).doubleValue

        let Lon = (lon! as NSString).doubleValue

        let center = CLLocationCoordinate2D(latitude: Lat, longitude:
Lon)

        let span = MKCoordinateSpan(latitudeDelta: 0.01,
longitudeDelta: 0.01)

        let location = MKCoordinateRegion(center: center, span: span)

```

```

        self.mapView.setRegion(location, animated: true)

        let pin = MKPointAnnotation()

        pin.coordinate = CLLocationCoordinate2DMake(Lat, Lon)

        pin.title = titulo

        self.mapView.addAnnotation(pin)

    }

}

```

Veja que tivemos que importar o **Mapkit** para nossa classe, logo após criamos o **@IBOutlet** e uma variável **locationManager** que indicará a região do mapa que será exibida, como o mapa é grande demais, vamos aproximar o mapa e exibiremos o ponto mais próximo possível, veja na figura X.

Criamos também três variáveis, **lat**, **lon** e **titulo** que são os parâmetros que recebemos da tela de detalhe.

Enfim, criamos um método chamado **exibeMapa()** onde setaremos a região a ser exibida e criaremos o incluiremos um **pin** no mapa.

Veja que na classe de mapa criamos as propriedades **lat**, **lon** e **titulo** para receber está no MapKit, ou seja precisamos importar o MapKit em nossa classe, vejam que fizemos isso também.

Importante: o método **requestWhenInUseAuthorization()** é para solicitar a permissão do usuário para nosso app acessar sua localização. A Apple é extremamente preocupada com a segurança do usuário, logo, qualquer recurso que seu app precisar usar como: localização, acesso à câmera, acesso ao microfone,

acesso à lista de música, acesso aos contatos etc. seu app sempre precisará da permissão do usuário.

No início do método setamos a **locationManager** com algumas configurações iniciais, dizemos também que o mapa poderá exibir a localização do usuário.

Criamos outras variáveis Lat, Lon para então fazermos o cast para **doubleValue**, que o método **CLLocationCoordinate2D()** espera receber e setamos esse resultado para uma variável **center**, que é onde centralizaremos a visualização da região do mapa, usamos mais alguns métodos do Mapkit **MKCoordinateSpan()** e **MKCoordinateRegion()**.

Enfim, criamos uma variável pin do tipo **MKPointAnnotation()** que será o nosso ponto no mapa, usamos suas propriedades para setar as coordenadas, ou seja, **lat**, **lon** e **titulo** usando o método **CLLocationCoordinate2DMake()** passando **Lat** e **Lon** indicamos onde será o ponto do mapa e por fim setamos com o método **addAnnotation()** o ponto ao nosso mapa.

Importante: no simulador devemos setar o botão **simulate location**, veja figura 110, para que seja possível simular uma localização, pois sabemos que o simulador não possui GPS.

Figura 110 – Simulando location no simulador



Ao compilarmos nosso projeto veremos o exibido nas figuras 111 e 112.

Figura 111 – Ponto e região do mapa setados



Figura 112 – Ponto da fábrica da Ferrari no mapa e localização atual do usuário setada para o Rio de Janeiro



Dica: para afastarmos o mapa no simulador, segure o botão option e clique com o mouse no mapa fazendo o gesto de pinça, dessa forma poderá ampliar e diminuir a visualização do mapa.

Agora vamos rodar uma url do Youtube, reparem na figura 105 que existe um campo chamado **url_video**, vamos exibir esse vídeo clicando no botão vídeo que está na nossa **viewController** detalhe.

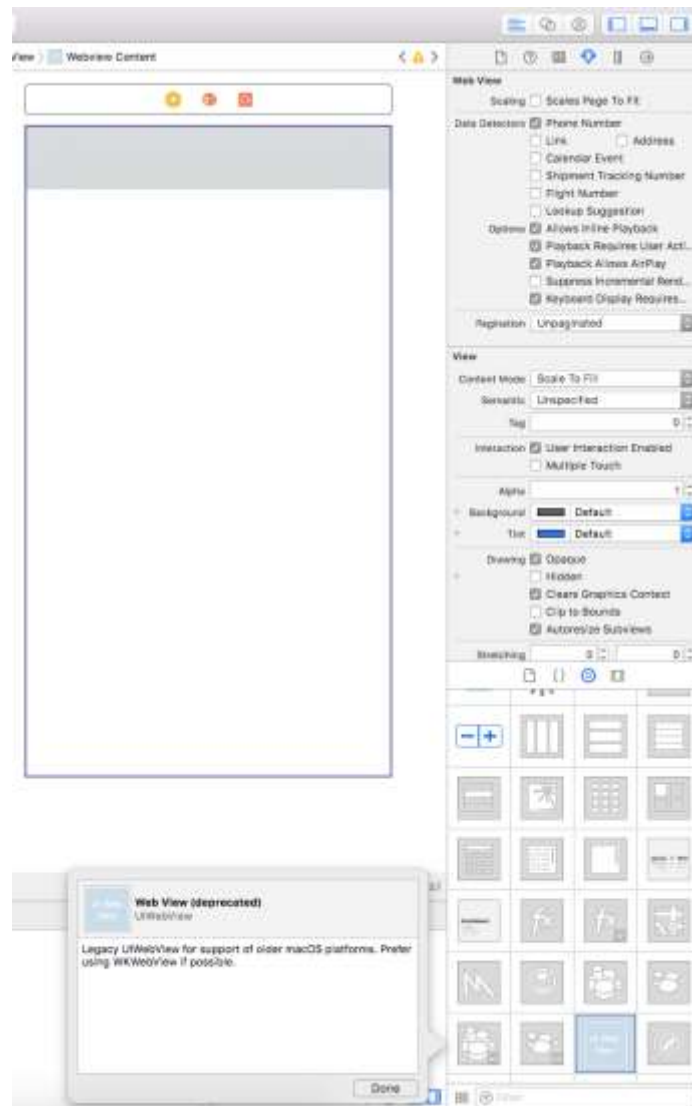
Para isso vamos usar uma forma simples e muito usual para exibir vídeos, vamos usar uma **UIWebView**, dessa forma já teremos controlar **play**, **pause** e **stop** automaticamente.

Vamos começar criando um novo **viewController** na **Main.storyboard**, e criaremos uma classe para esse **viewController** chamada **VideoViewController.swift**.

Lembre-se que devemos fazer as devidas ligações, como vincular no **viewController** a classe que usaremos (já fizemos isso em capítulos anteriores), criar **@IBOutlets**, dar um **StoryboardID** para a **viewController** etc. tudo que vimos anteriormente.

A figura 113 exibe o componente **Webview** e como ficará nossa **viewController** para o vídeo.

Figura 113 – Componente WebView



DetailViewController.swift

```
@IBAction func showVideo(_ sender: UIButton) {

    let video = self.storyboard?.instantiateViewController(withIdentifier:
"VideoViewController") as! VideoViewController

    video.urlID = self.detalheCarro["url_video"]?.components(separatedBy:
```

```
"=")[1]
```

```
navigationController?.pushViewController(video, animated: true)

}
```

Na classe de detalhe do veículo criamos um método chamado **showVideo()** e é nele que iremos instanciar nossa **viewController** e fazer a chamada utilizando o método **pushViewController**. Reparem que passamos um parâmetro que é a URL, mas não é ela completa, veja que usamos o método **components()** para quebrar nossa string e retornar apenas o ID do vídeo no Youtube, pois para exibir URLs do Youtube em apps a url usada é um pouco diferente.

VideoViewController.swift

```
import UIKit

class VideoViewController: UIViewController {

    var urlID: String?

    @IBOutlet var webViewContent: UIWebView!

    override func viewDidLoad() {

        super.viewDidLoad()
```

```

        self.playVideo()

    }

    ...

    func playVideo() {

        let URLcompleta = "https://www.youtube.com/embed/" +
self.urlID!

        let baseURL = URL(string: URLcompleta)!

        self.webviewContent.allowsInlineMediaPlayback = true

        self.webviewContent.loadHTMLString("<iframe width=\"100%\"
height=\"100%\" src=\"\(URLcompleta)\\" allowfullscreen></iframe>", baseURL:
baseURL)

    }

}

```

Vejam como ficou o código na **VideoViewController.swift**.

Criamos uma variável para receber o parâmetro da tela detalhe, **urlID**, criamos um **@IBOutlet** para a **UIWebView** que criamos e por fim criamos um método chamado **playVideo()** que efetivamente roda o vídeo.

Veja que fazemos uma concatenação da **urlID**, recebida por parâmetro com a url base para rodar links do Youtube em apps e setamos para a variável **URLcompleta**.

Feito isso precisamos utilizar a método **URL()** para retornar uma URL utilizável para nosso componente (webview).

Por fim usamos o método **loadHTMLString()** do webview para adicionarmos um **<iframe>** que é um pedaço de código **HTML** para então rodar nosso vídeo com extrema facilidade.

No nosso **iframe** passamos a url de forma simples e indicamos que ele será fullscreen (**allowfullscreen**) e passamos para o componente nossa URL modificada com o método **URL()**. Ao compilar teremos o resultado exibido na figura 113.1.

Figura 113.1 – URL do Youtube rodando no webview



Agora vamos implementar o item “Sobre” na nossa tabbar. Já temos a **viewController** criar, faltando apenas a classe para vinculação, então, vamos criá-la.

Nossa tela de sobre conterá um webview direcionando para a página do IGTI, apenas isso.

SobreViewController.swift

```
import UIKit
```

```
class SobreViewController: UIViewController {
```

```
    @IBOutlet var webViewContent: UIWebView!
```

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

```
        super.viewDidLoad()
```

```
        self.showURL()
```

```
    }
```

```
    ...
```

```
    func showURL() {
```

```
        let url = "http://igti.com.br"
```

```
        let urlRequest = URLRequest(url: URL(string: url)!)
```

```
        self.webViewContent.allowsInlineMediaPlayback = true
```

```
        self.webViewContent.loadRequest(urlRequest as URLRequest)
```

```
    }
```

```
}
```

Criamos um método **showURL()**, é bastante parecido com o **playVideo()**, a única diferença é que usamos o método **loadRequest()** passando a **URLRequest** que criamos. Ao compilarmos teremos o resultado igual ao da figura 114.

Figura 114 – Usando Webview para exibir endereço de site



Com isso terminamos nosso app, com ele aprendemos de forma simples as funcionalidades mais comuns para criação de um app, navegação de telas, passagem de parâmetros entre telas, mapKit, webview etc. Vamos agora nos aprofundar um pouco em alguns conceitos e ver mais detalhes sobre o mundo Apple. Com isso você será capaz de fazer alterações no app que criamos caso sinta necessidade.

Capítulo 5. Entendendo o uso de Blocos / Closures

Blocos / Closures

Um bloco / closure é um trecho de código que pode ser executado de forma independente ou até passado como parâmetro para outra função e está disponível desde o iOS 4, inclusive a Apple recomenda bastante o uso de bloco / closure.

A ideia é poder criar funções “customizáveis”, ou seja, um pedaço de código que espera outro pedaço de código. Existem duas formas de fazer isso, no mundo C podemos passar diretamente uma função como parâmetro para ser executada dentro de outra função, isso não é um bloco é o que chamado de “callback”. Em **Swift**, podemos passar uma função como parâmetro ou mesmo fazer uma função retornar uma função.

Vamos à sintaxe.

```
{ (parâmetros) -> tipo retorno }
```

Vamos agora criar um arquivo **.playground** (já fizemos anteriormente), vamos chamá-lo de **MyPlayground.playground**.

No trecho de código abaixo podemos ver a utilização de blocos.

MyPlayground.playground

```
import UIKit
```

```
let imprimeData = { () -> Void in
```

```
    let date = NSDate()
```

```
print("Data: ", date)

}

imprimeData()
```

Criamos uma variável **imprimeData()** que recebe um bloco, reparem na sintaxe.

Este nosso bloco não recebe nenhum parâmetro e também não retorna nenhum parâmetro. A execução pode ser vista na figura 115, que é o nosso arquivo **.playground**.

Figura 115 - Blocos



Uma função criada por um bloco pode receber parâmetros e também retornar resultado, como qualquer outra função, veja abaixo:

```
let soma = { (a: Int, b: Int) -> Int in

    return a + b
```



```

}

let resultado = soma(3,4)

print("Soma: ", resultado)

```

Nosso clousre **soma()** recebe dois parâmetros e faz a soma entre eles já retornando o resultado para a variável soma, veja na figura 116.

Figura 116 – Criando bloco soma()



Até agora simplesmente criamos nossos blocos dentro do arquivo .playground, mas e se quisermos colocá-los dentro de uma função, apenas colocaremos o código já criado dentro de uma função e a chamamos. Veja abaixo:

```

func executaSoma() {

    let soma = { (a: Int, b: Int) -> Int in

        return a + b

    }
}

```

```
let resultado = soma(3,4)

print("Soma: ", resultado)

}

executaSoma()
```

Vamos agora aprofundar um pouco mais nossos exemplos.

```
import UIKit

func soma(x: Int, y: Int) -> Int {

    return x + y

}

func calculadora(calculo: (Int, Int) -> (Int), a: Int, b: Int) {

    let resultado = calculo(a, b)

    print("Resultado: ", resultado)

}

calculadora(calculo: soma(x:y:), a: 10, b: 20)
```

No exemplo acima, definimos uma função de **soma()**, que recebe dois inteiros como parâmetros e retorna um inteiro. Depois definimos uma função

genérica chamada **calculadora()** que recebe como parâmetro uma função com a assinatura **(Int, Int) -> Int** que significa uma função que receba dois inteiros e retorne um inteiro.

Ao executar passamos a função **soma()**, números 10 e 20 e internamente atribuímos a função **soma()** a uma variável chamada **calculo** e executamos os dois inteiros, que, obviamente, serão somados. E a resposta no final será 30.

Nota: então, simplificando, estamos dizendo que nossa função **calculadora()** recebe outra função chamada **soma()** para fazer a soma, propriamente dita, dos números 10 e 20 que passamos como parâmetros.

Podemos então criar outras funções que façam o restante das contas básicas, veja abaixo:

```
import UIKit
```

```
func soma(x: Int, y: Int) -> Int {  
  
    return x + y  
  
}
```

```
func subtracao(x: Int, y: Int) -> Int {  
  
    return x - y  
  
}
```

```
func multiplicacao(x: Int, y: Int) -> Int {
```

```

    return x * y
}

```

```

func divisao(x: Int, y: Int) -> Int {

    return x / y
}

```

```

func calculadora(calculo: (Int, Int) -> (Int), a: Int, b: Int) {

    let resultado = calculo(a, b)

    print("Resultado: ", resultado)

}

```

```

calculadora(calculo: soma(x:y:), a: 10, b: 20)

```

```

calculadora(calculo: subtracao(x:y:), a: 192, b: 39)

```

```

calculadora(calculo: multiplicacao(x:y:), a: 54, b: 100)

```

```

calculadora(calculo: divisao(x:y:), a: 1038934, b: 1293)

```

Com o código acima, teremos o resultado abaixo:

Resultado: 30

Resultado: 153

Resultado: 5400

Resultado: 803

Fazendo a pequena alteração abaixo podemos falar qual o tipo de operação que estamos realizando.

```
import UIKit
```

```
func soma(x: Int, y: Int) -> Int {  
  
    return x + y  
  
}
```

```
func subtracao(x: Int, y: Int) -> Int {  
  
    return x - y  
  
}
```

```
func multiplicacao(x: Int, y: Int) -> Int {  
  
    return x * y  
  
}
```

```
func divisao(x: Int, y: Int) -> Int {  
  
    return x / y  
  
}
```

```
func calculadora(tipo: String, calculo: (Int, Int) -> Int, a: Int, b: Int) {  
  
    let resultado = calculo(a, b)  
  
    print("\(tipo): \(resultado)")  
  
}  
  
calculadora(tipo: "soma", calculo: soma(x:y:), a: 10, b: 20)  
  
calculadora(tipo: "subtracao", calculo: subtracao(x:y:), a: 192, b: 39)  
  
calculadora(tipo: "multiplicacao", calculo: multiplicacao(x:y:), a: 54, b: 100)  
  
calculadora(tipo: "divisao", calculo: divisao(x:y:), a: 1038934, b: 1293)
```

O resultado será:

soma: 30

subtracao: 153

multiplicacao: 5400

divisao: 803

Usando Closures para animações

Blocos são frequentemente usados para passar trechos de códigos como argumento para métodos, e esse recurso pode ser utilizado para criar animações.

Nota: a partir do iOS 4 a Apple recomenda o uso de blocos para criar animações, pois deixa o código mais simples e de fácil entendimento.

Vamos criar um novo projeto chamado **Animacao** do tipo **Single View Application**.

Para criar uma animação a classe **UIView** contém um método chamado **animate**, que na verdade é um bloco, vamos usá-lo para nossa animação.

Para nosso teste de animação incluiremos uma imagem de bola para que possamos animá-la, lembre-se que temos que adicionar a imagem ao nosso projeto.

Feito isso, incluiremos em nossa **viewController** dois componentes: uma imagem e um botão, nossa tela ficará igual a tela da figura 117.

Figura 117 – Animação



Vamos apenas adicionar o código abaixo na nossa **viewController** criada pelo projeto.

ViewController.swift

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet var bola: UIImageView!

    override func viewDidLoad() {

        super.viewDidLoad()

    }

    override func didReceiveMemoryWarning() {

        super.didReceiveMemoryWarning()

    }

    @IBAction func animate(_ sender: Any) {

        self.showAnimation()

    }

}
```



```
func showAnimation() {

    UIView.animate(withDuration: 1, delay: 0, options:
UIViewAnimationOptions.transitionCrossDissolve, animations: {

        self.bola.frame = CGRect(x: self.bola.frame.origin.x +
300, y: self.bola.frame.origin.y, width: self.bola.frame.size.width, height:
self.bola.frame.size.height)

    }) { (valid) in

        if valid && self.bola.frame.origin.x > 400 {

            self.bola.frame = CGRect(x: 16, y:
self.bola.frame.origin.y, width: self.bola.frame.size.width, height:
self.bola.frame.size.height)

        }

    }

}
```

Vejam que criamos um método chamado **showAnimation()** para iniciar nossa animação. Nele usamos o método **animate()** da **UIView**.

Esse método recebe alguns parâmetros, vamos ver:

withDuration: é a quantidade de tempo (em segundos) que nossa animação deverá durar.

delay: é se vamos incluir um tempo de demora em nossa animação.

options: o tipo de animação que usaremos (nesse caso é irrelevante).

animations: é a animação em si, aqui colocamos nosso código que precisa ser animado.

completion: qual código executaremos após o término da nossa animação.

Veja que dentro do primeiro bloco **animations** setamos a nossa imagem para outra posição, apenas pegamos a posição original e somamos 300, repare que os outros parâmetros continuam com o mesmo valor, pois buscamos o valor da própria imagem.

Já no bloco **completion** (que é o complemento depois da animação, caso precisemos) incluímos um código para validar se o método **animate()** retornou corretamente e se a posição “x” da bola é maior que 400, pois se for voltaremos com a bola para a posição inicial.

Criamos também um **@IBAction** para nosso botão que chamará “**chutar**”. Ao clicar neste botão, será chamado o método **showAnimation()** que criamos, e movimentará a bola.

Capítulo 6. Persistência

Antes de usarmos banco de dados em nossos apps devemos avaliar outras opções que existem no mercado, pois podem nos levar a soluções mais simples.

Classe UserDefaults

Essa é uma classe nativa do iOS e funciona como uma estrutura de chave e valor onde podemos salvar e recuperar qualquer valor de forma rápida.

Nos permite salvar números, booleanos, strings, arrays, dicionários etc.

Esta é uma classe que armazena dados de configuração do app, dados de login etc. Não é indicado para armazenar grandes quantidades de dados, dados complexos para isso temos o **SQLite** (bando de dados relacional embarcado) que veremos posteriormente.

Vamos criar um novo projeto chamado **Persistencia**, escolha o **Single View Application**.

Vamos adicionar o código abaixo no método **viewDidLoad()** da classe **ViewController.swift**.

ViewController.swift

```
let prefs = UserDefaults.standard
```

```
prefs.setValue("conteúdo salvo em UserDefaults", forKey: "chave")
```

```
prefs.synchronize()
```

```
print("prefs: ", prefs.string(forKey: "chave"))
```

O resultado será:

prefs: conteúdo salvo em UserDefaults

O método **standard** retorna a instância compartilhada de **UserDefaults**, depois disso podemos utilizar os métodos para salvar os dados sempre informando a chave e o valor.

Importante: como o **UserDefaults** utiliza um banco de dados interno, podemos recuperar essas informações quando quisermos, até mesmo se o usuário fechar o app e para garantir isso precisamos usar o método **synchronize()** para sincronizar o dado com o banco de dados interno, como se fosse um **commit**.

Para recuperar o valor usamos o método **string(forKey)**, mas vamos criar uma classe que irá encapsular e facilitar nosso trabalho com **UserDefaults**.

Prefs.swift

```
import Foundation
```

```
class Prefs {
```

```
class func setObject(_ value: Any, key: String) {  
  
    let config = UserDefaults.standard  
  
    config.setValue(value, forKey: key)  
  
    config.synchronize()  
  
}
```

```
class func getString(_ key: String) -> String {  
  
    let config = UserDefaults.standard  
  
    if let ret = config.string(forKey: key) {  
  
        return ret  
  
    } else {  
  
        return ""  
  
    }  
  
}
```

```
class func getInt(_ key: String) -> Int {  
  
    let config = UserDefaults.standard  
  
    return config.integer(forKey: key)  
  
}
```

```
class func getBool(_ key: String) -> Bool {
```

```
        let config = UserDefaults.standard

        return config.bool(forKey: key)
    }

class func getFloat(_ key: String) -> Float {

    let config = UserDefaults.standard

    return config.float(forKey: key)
}

class func getDouble(_ key: String) -> Double {

    let config = UserDefaults.standard

    return config.double(forKey: key)
}

class func getURL(_ key: String) -> URL? {

    let config = UserDefaults.standard

    return config.url(forKey: key)
}

class func getData(_ key: String) -> Data? {

    let config = UserDefaults.standard
```

```

        return config.data(forKey: key)
    }

    class func getArray(_ key: String) -> [Any]? {

        let config = UserDefaults.standard

        return config.array(forKey: key)!
    }

    class func getStringArray(_ key: String) -> [String]? {

        let config = UserDefaults.standard

        return config.stringArray(forKey: key)
    }

    class func getDictionary(_ key: String) -> [String: Any] {

        let config = UserDefaults.standard

        return config.dictionary(forKey: key)!
    }
}

```

Importante: reparem que nossa classe possui apenas um método para gravar valor e vários para recuperar o valor, isso pelo fato de usarmos o método

setValue() que serve para qualquer tipo de dado, mas para retornar precisamos retornar o tipo específico. Outro detalhe é a palavra **class** na declaração da função, isso indica que é uma função / método de classe / estático, ou seja, a **Prefs** não precisa ser instanciada, podemos usar as funções diretamente como é mostrado nos exemplos abaixo.

Agora que criamos nossa classe de encapsulamento do UserDefaults, vamos usá-la, veja abaixo:

ViewController.swift

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {

        super.viewDidLoad()

        Prefs.setObject("Conteúdo salvo em UserDefaults", key:
"chave")

        print("class Prefs.: ", Prefs.getString("chave"))

        let arr = [1, 2, 3, 4, 5]
```



```

        Prefs.setObject(arr, key: "numbers")

        print("numbers: ", Prefs.getArray("numbers")!)
    }

    override func didReceiveMemoryWarning() {

        super.didReceiveMemoryWarning()

        // Dispose of any resources that can be recreated.
    }
}

```

Veja que guardamos e imprimimos conteúdo do tipo **String** e **Array**, veja que nossa classe, **Prefs**, simplificou bastante o uso e o resultado é exibido abaixo:

```

class Prefs.: Conteúdo salvo em UserDefaults

numbers: [1, 2, 3, 4, 5]

```

Escrita e leitura de arquivos

Arquivos também podem ser utilizados para salvar informações e já fizemos isso na nossa classe **DownloadImageView**, nós salvamos as imagens dos carros.

Para ler e salvar arquivos, podemos utilizar a classe **Data**. O método **writeToFile()** escreve o conteúdo do **Data** no arquivo especificado e o método **dataWithContentsOfFile()** é utilizado para ler o arquivo e retornar um **Data**.

Importante: no iOS existe uma pasta específica para criar os arquivos de nossa aplicação, caso seja feito em outra pasta seu app corre o risco de ser reprovado pela Apple, pois isso faz parte das boas práticas de desenvolvimento que ela recomenda.

Para isso usaremos uma classe auxiliar chamada **StringUtils**, é uma classe com manipulações simples de string, não entraremos em detalhe sobre essa classe. Percebe-se que as conversões existentes são simples e de fácil entendimento.

ViewController.swift

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {

        super.viewDidLoad()

        Prefs.setObject("Conteúdo salvo em UserDefaults", key:
"chave")

        print("class Prefs.: ", Prefs.getString("chave"))

        let arr = [1, 2, 3, 4, 5]

        Prefs.setObject(arr, key: "numbers")
    }
}
```

```
print("numbers: ", Prefs.getArray("numbers"))!
```

```
let str = "Salvar string em arquivo"
```

```
self.writeFile(str)
```

```
print(readFile("arquivo"))
```

```
}
```

```
override func didReceiveMemoryWarning() {
```

```
    super.didReceiveMemoryWarning()
```

```
    // Dispose of any resources that can be recreated.
```

```
}
```

```
func getPath(_ fileName: String) -> String {
```

```
    let path = NSHomeDirectory() + "/Documents/" + fileName +  
    ".txt"
```

```
    print(path)
```

```
    return path
```

```
}
```

```
func writeFile(_ valor:String) {
```

```
    let path = getPath("arquivo")
```

```

        let nsdata = StringUtils.toNSData(valor)

        try? nsdata.write(to: URL(fileURLWithPath: path), options:
[.atomic])
    }

func readFile(_ chave: String) -> String {

    let path = getPath("arquivo")

    let nsdata = try? Data(contentsOf: URL(fileURLWithPath: path))

    let s = StringUtils.toString(nsdata)

    if let s = s {

        return s

    } else {

        return ""

    }

}
}

```

Foram criados dois métodos para ler e para gravar arquivo, vamos a eles.

func getPath(_ fileName: String) -> String: recebe como parâmetro o nome do arquivo que será criado para gravar o conteúdo, com base neste nome retorna o caminho completo do arquivo, lembrando que devemos sempre salvar arquivos na pasta Documents que é específica para isso.

Utiliza a classe **NSHomeDirectory()** para retornar o caminho padrão do app e concatena com o nome passado por parâmetro e por fim retorna o caminho completo do arquivo.

func writeFile(_ valor:String): recebe como parâmetro o valor / conteúdo que será gravado.

Busca o caminho utilizando o método **getPath()**, utiliza a classe **StringUtils** para converter nosso conteúdo que está em **String** para **Data** e grava o conteúdo no arquivo.

func readFile(_ chave: String) -> String: recebe como parâmetro o nome do arquivo ou chave, utiliza o método **getPath()** para retornar o caminho do arquivo, usa o método **Data()** para retornar um **Data** na variável **nsdata** e por fim utilizamos a classe **StringUtils** para converter o **Data** em **String**.

Ao rodarmos nosso código veremos o resultado abaixo:

class Prefs.: Conteúdo salvo em UserDefaults

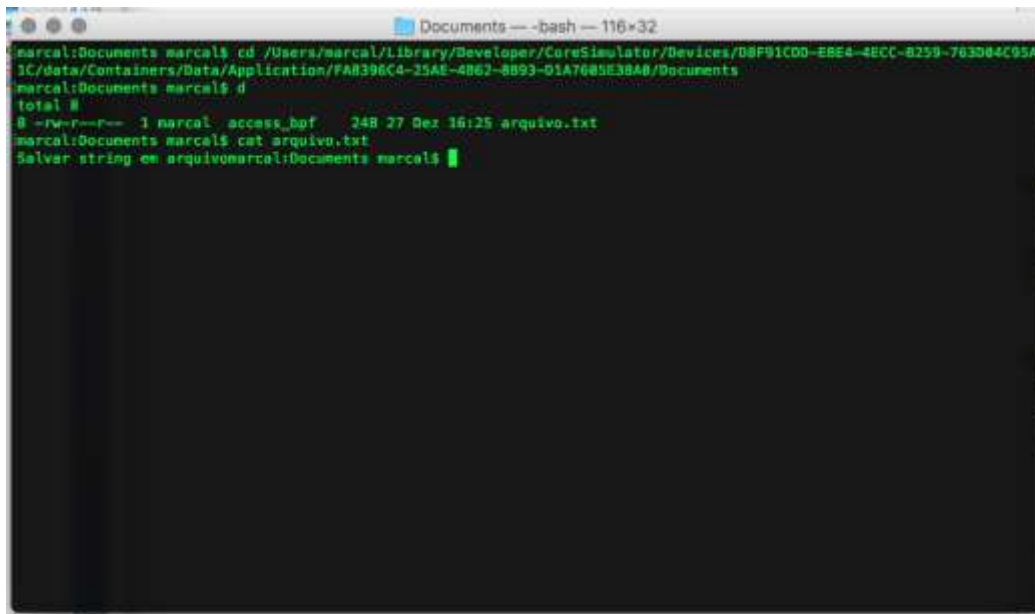
numbers: [1, 2, 3, 4, 5]

/Users/marcal/Library/Developer/CoreSimulator/Devices/D8F91CDD-EBE4-4ECC-B259-763D04C95A1C/data/Containers/Data/Application/FA8396C4-25AE-4B62-8B93-D1A7605E38A0/Documents/arquivo.txt

Salvar string em arquivo

Vejam que mandamos imprimir o caminho do arquivo, se pegarmos o caminho e acessarmos via terminal poderemos ver nosso arquivo gravado lá, veja na figura 118.

Figura 118 – Arquivo gravado



```

marcal:Documents marcal$ cd /Users/marcal/Library/Developer/CoreSimulator/Devices/D8F91CDD-EBE4-4BCC-B259-763D84C93A1C/data/Containers/Data/Application/FAB396C4-25AE-4862-8B93-01A7685E38A8/Documents
marcal:Documents marcal$ d
total 4
-rw-r--r--  1 marcal  access.hpf    24B  27 Dez 16:25 arquivo.txt
marcal:Documents marcal$ cat arquivo.txt
Salvar string em arquivomarcal:Documents marcal$
  
```

Importante: este caminho que temos é do app no simulador, quando usarmos em um device o caminho será diferente, mas a ideia segue a mesma lógica.

Banco de dados SQLite

O SQLite é um banco de dados simples porém muito poderoso e de código-fonte aberto. É construído em C para aproveitar o máximo de performance e por isso é muito utilizado em dispositivos móveis para saber mais sobre o SQLite pode acessar o site <http://www.sqlite.org>.

Não pensem que pelo **SQLite** ser um banco mais simples ele só pode ser manipulado por linha de comando, isso é possível mas no mercado tem várias ferramentas para gerenciarmos bancos de dados **SQLite**.

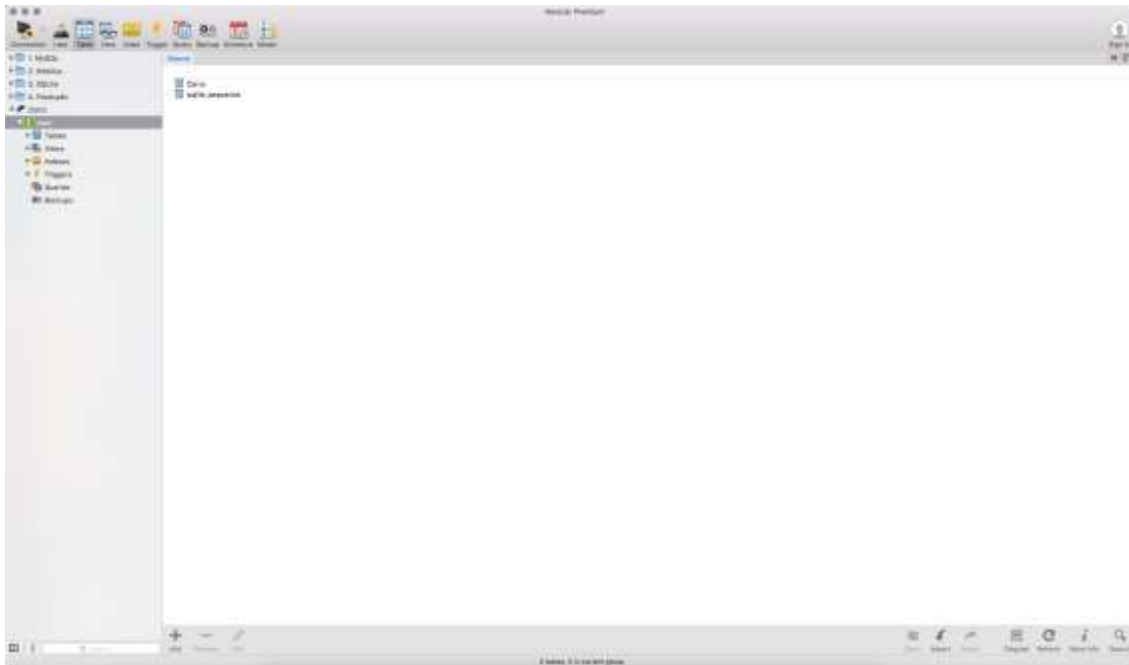
O **Navicat** é uma das melhores e mais completas ferramentas que existem no mercado, e ele também pode ser utilizado com vários outros bancos de dados (**MySQL**, **SQL server**, **PostgreSQL** etc.).

A licença do **Navicat** é bastante cara, mas temos outras soluções no mercado como o **Lita**, **SQLite Database Browser** etc. fazendo uma busca rápida pelo Google é possível encontrar várias ferramentas, veja a que melhor atende às suas necessidades.

Usarei o **Navicat** pois ele possui algumas coisas boas que irei mostrar.

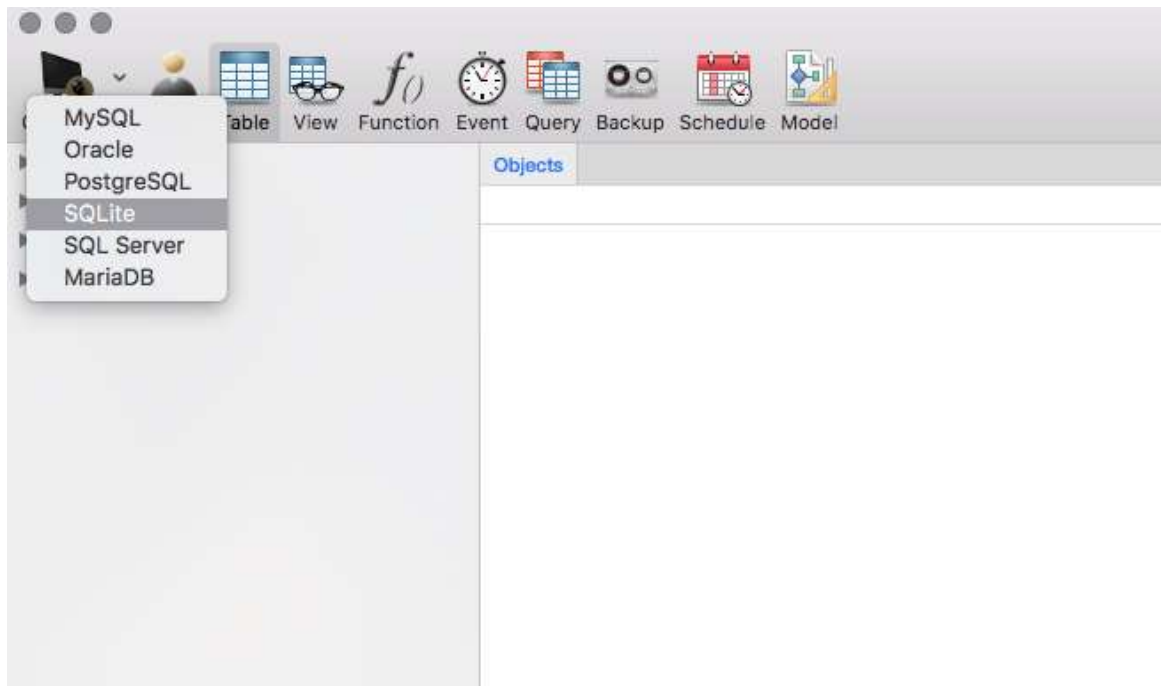
A figura 119 exhibe o **Navicat**, com o banco de dados **Carro**, aberto.

Figura 119 - Navicat



Podemos criar o banco de dados utilizando o **Navicat** ou podemos criar direto no projeto, veremos isso posteriormente, agora veremos como criar um banco de dados **SQLite** com o **Navicat**.

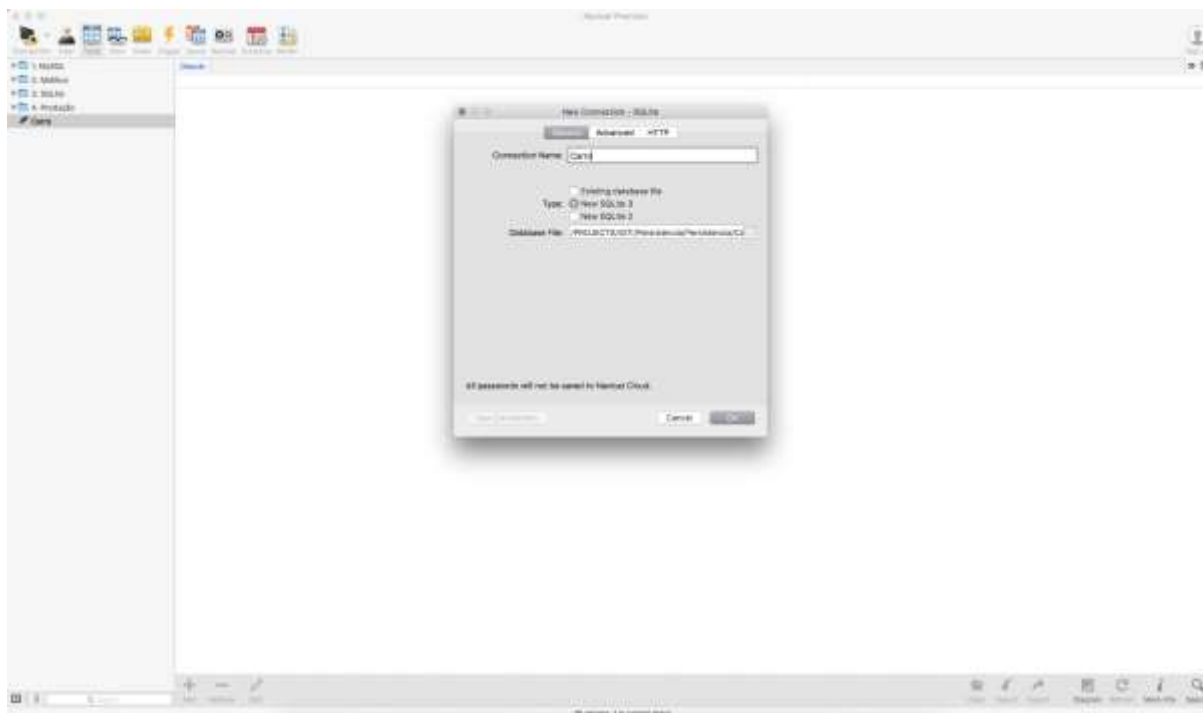
Figura 120 – Criando banco de dados SQLite



Conforme mostrado na figura, selecione o tipo de banco de dados SQLite.

Escolha o nome do banco e onde irá salvá-lo (figura 121).

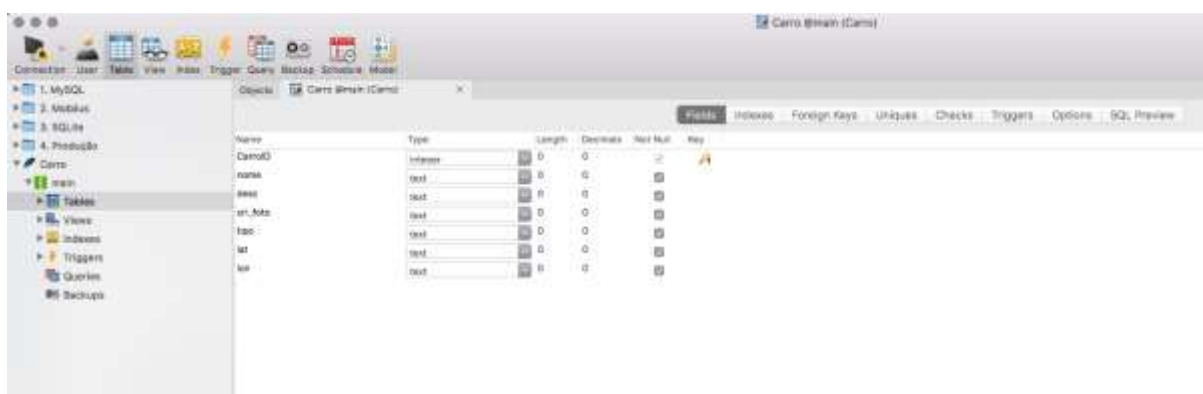
Figura 121 – Salvando banco de dados



Feito isso, o banco estará criado.

Agora precisamos criar nossa tabela carro, veja na figura 122 quais os campos e o nome que daremos a nossa tabela.

Figura 122 – Criando tabela carro



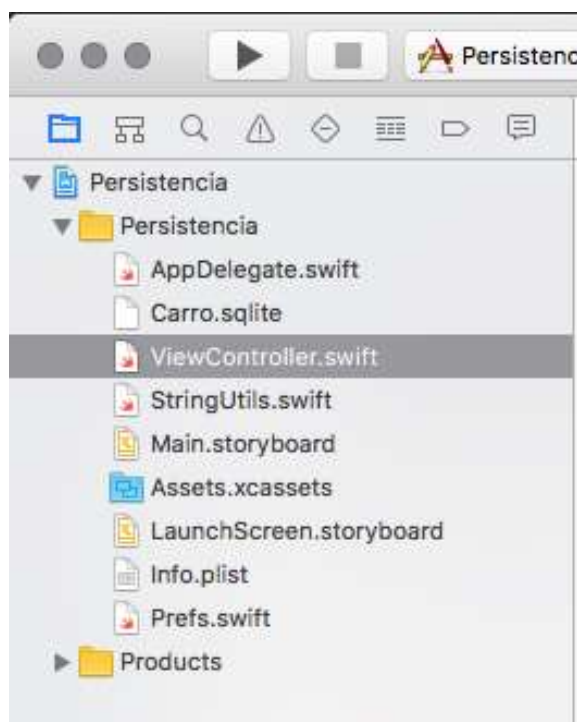
Importante: esta apostila não tratará da criação a manutenção de banco de dados **SQLite**, foi citado apenas para uso no projeto de persistência, caso queiram se aprofundar em **SQLite**, o que é importante, existe muito material na internet bem como no site oficial.

Importante lembrar também que outras ferramentas para gestão de bancos SQLite funcionam da mesma forma que o **Navicat**, tornando o aprendizado aqui útil para diversas ferramentas.

Abaixo veja como está a estrutura de arquivos do projeto.

Dica: caso crie seu banco de dados em alguma ferramenta, precisará incluir o arquivo **.sqlite** no projeto como mostra a figura 123, se fizer via código, como mostraremos posteriormente, não precisar fazer a inclusão, pois o arquivo já será criado no lugar correto.

Figura 123 – Adicionado banco de dados SQLite

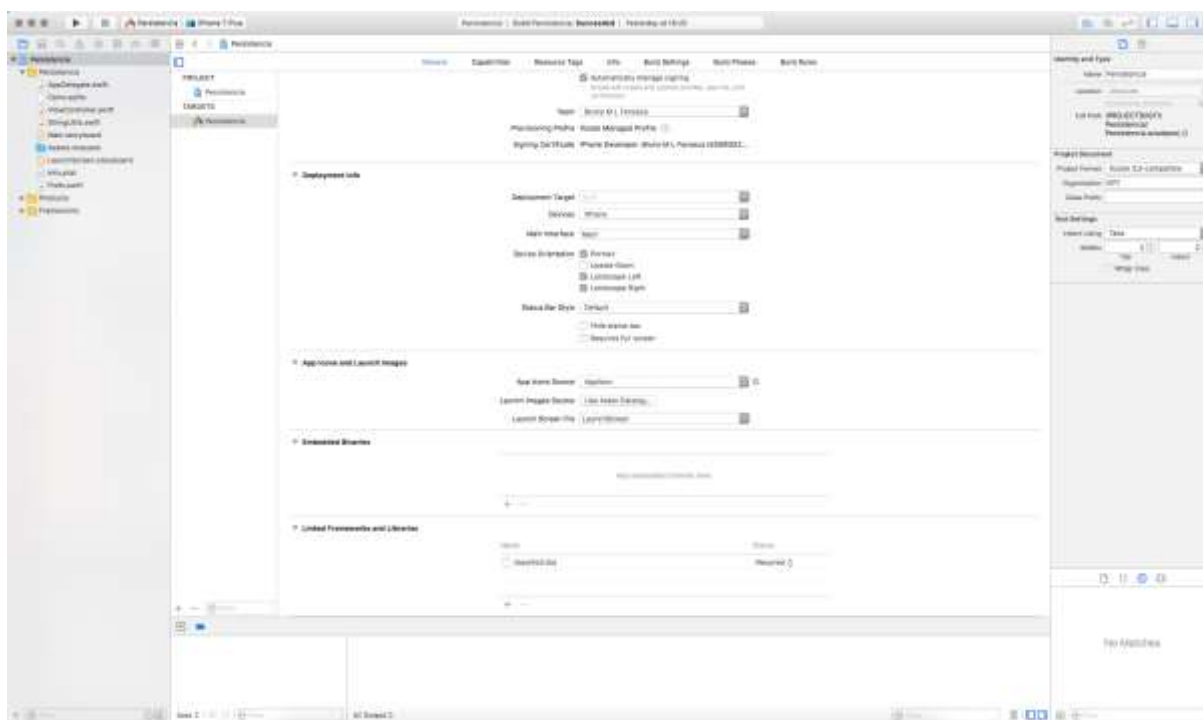


Dica: a utilização de uma ferramenta para a manutenção do banco de dados é muito útil, pois poderemos abrir arquivo de banco **.sqlite** e vermos o que foi incluído, excluído, alterado dependendo do que fizemos em nossa aplicação.

Agora, precisaremos adicionar a lib para utilizarmos o SQLite em nosso projeto de persistência, veja a figura 124.

Selecione o projeto e vá à aba **General**, role toda para baixo, encontre a seção **Linked Frameworks and Libraries** clique no botão “+”, aparecerá outra tela onde será possível buscar por libs e frameworks, então digite “sqlite” aparecerá duas libs, escolha a **libsqlite3.tdb**. Agora precisamos criar um arquivo de bridge.

Figura 124 – Adicionando lib SQLite ao projeto

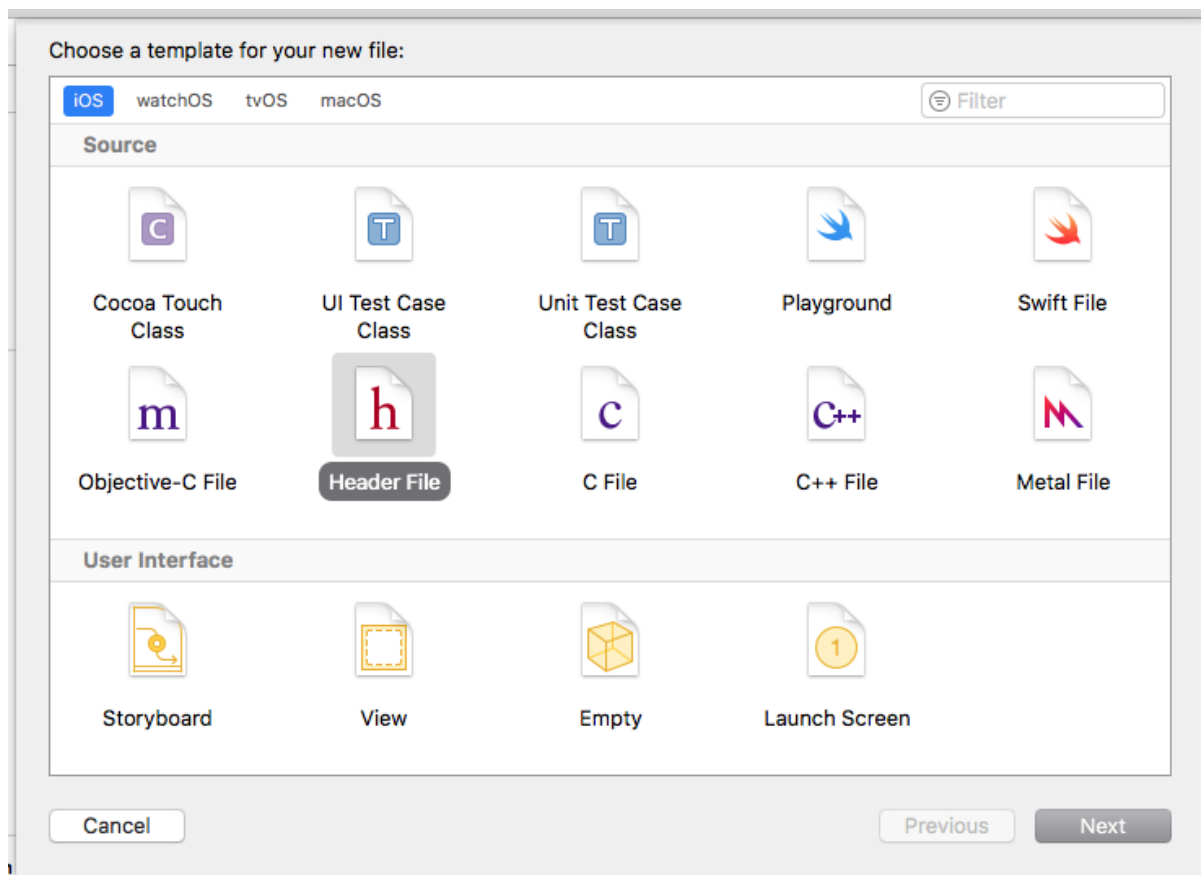


Importante: a lib do **SQLite** é escrita em **Obj-C**, por isso precisamos de um arquivo de **bridge** para fazer essa ligação e podermos usar **libs Obj-C** em projetos **Swift**.

Ainda existem muitas libs escritas em **Obj-C** que ainda não foram migradas para **Swift**, e pra ser sincero não sei quando serão. Por isso o arquivo de **bridge** é uma forma de se comunicar com essas libs e ao mesmo tempo deixar o projeto organizado caso seja necessário usar **Obj-C**.

Vamos incluir um novo arquivo no projeto, escolha o template **Header File** como mostra figura 125, de o nome de **Persistencia-Bridging-Header.h**.

Figura 125 – Criando arquivo de bridge



Feito isso, agora, precisamos indicar em nosso projeto onde está o arquivo de bridge. Selecione o projeto e vá na aba **Build Settings** e no campo de busca digite “**bridge**”, encontre a seção **Objective-C Bridging Header**, clique duas vezes e inclua o caminho abaixo:

`$(SRCROOT)/$(PROJECT_NAME)/Persistencia-Bridging-Header.h`

Notas: estamos usando duas variáveis de ambiente do xCode para retornar o caminho e o nome do projeto, dessa forma só precisamos incluir o nome do arquivo de bridge, conforme figura 126.

Figura 126 – Adicionando caminho do arquivo de bridge



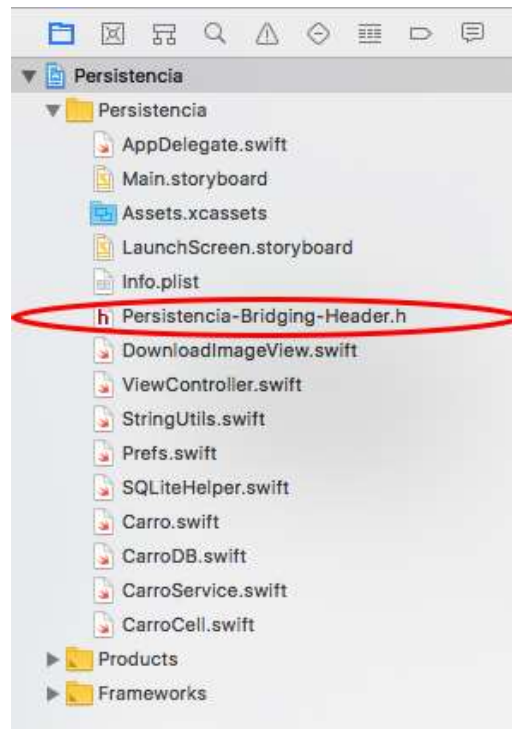
Veja como deverá ficar nosso arquivo de bridge configurado na figura 127.

Figura 127 – Caminho do arquivo de bridge adicionado



Agora devemos selecionar o arquivo de bridge e incluir o **#import** para usarmos a lib do SQLite, veja abaixo.

Figura 127.1 – Estrutura do projeto contendo arquivo de bridge



Criando classe utilitária para SQLite

Para facilitar o trabalho com SQLite, que diga-se de passagem, é muito trabalhoso, criei uma classe para encapsular o trabalho com SQLite, veja o bloco de código a seguir:

SQLiteHelper.swift

```
import Foundation
```

```
class SQLiteHelper: NSObject {
```

```
// sqlite3 *db;
```

```
var db: OpaquePointer? = nil
```

```
// Construtor
```

```
init(database: String) {
```

```
    super.init()
```

```
    self.db = open(database)
```

```
}
```

```
// Caminho do banco de dados
```

```
func getFilePath(_ nome: String) -> String {
```

```
    // Caminho com o arquivo
```

```
    let path = NSHomeDirectory() + "/Documents/" + nome
```

```
    print("Database: \(path)")
```

```
    return path
```

```
}
```

```
// Abre o banco de dados
```



```
func open(_ database: String) -> OpaquePointer? {

    var db: OpaquePointer? = nil;

    let path = getFilePath(database)

    let cPath = StringUtils.toCString(path)

    let result = sqlite3_open(cPath, &db);

    if(result != SQLITE_OK) {

        print("Não foi possível abrir o banco de dados SQLite
\\(result)")

        return nil

    }

    return db

}

// Executa o SQL

func execSql(_ sql: String) -> CInt {

    return self.execSql(sql, params: nil)

}
```

```
func execSql(_ sql: String, params: Array<AnyObject>!) -> CInt {

    var result:CInt = 0

    // Statement

    let stmt = query(sql, params: params)

    // Step

    result = sqlite3_step(stmt)

    if (result != SQLITE_OK && result != SQLITE_DONE) {

        sqlite3_finalize(stmt)

        let msg = "Erro ao executar SQL\n\(sql)\nError:
\\(lastSQLError())"

        print(msg)

        return -1

    }

    // Se for insert recupera o id

    if (sql.uppercased().hasPrefix("INSERT")) {

        let rid = sqlite3_last_insert_rowid(self.db)

        result = CInt(rid)

    } else {
```

```
        result = 1

    }

    // Fecha o statement

    sqlite3_finalize(stmt)

    return result

}

// Executa o SQL e retorna o statement

func query(_ sql:String) -> OpaquePointer {

    return query(sql, params: nil)

}

// Executa o SQL e retorna o statement

func query(_ sql:String, params: Array<AnyObject>!) -> OpaquePointer
{

    var stmt:OpaquePointer? = nil

    let cSql = StringUtils.toCString(sql)
```

```
// Prepare

let result = sqlite3_prepare_v2(self.db, cSql, -1, &stmt, nil)

if (result != SQLITE_OK) {

    sqlite3_finalize(stmt)

    let msg = "Erro ao preparar SQL\n\\(sql)\\nError:
\\(lastSQLiteError())"

    print("SQLite ERROR \\(msg)")

} else {

    print("SQL [\\(sql)], params: \\(params)")

}

// Bind Values (?, ?, ?)

if(params != nil) {

    bindParams(stmt!, params:params)

}

return stmt!

}

// Faz o bind dos parametros (?, ?, ?) de um SQL
```

```
func bindParams(_ stmt:OpaquePointer, params: Array<Any>!) {

    if(params != nil) {

        let size = params.count

        for i:Int in 1...size {

            let value = params[i-1]

            if(value is Int) {

                let number:CInt = toCInt(value as! Int)

                sqlite3_bind_int(stmt, toCInt(i), number)

                // println("bind int \(i) -> \(value)")

            } else {

                let text: String = value as! String

                let ns = text as NSString

                sqlite3_bind_text(stmt, Int32(i),

ns.utf8String, -1, nil)

            }

        }

    }

}
```

```
    }  
}  
  
// Retorna true se existe a próxima linha da consulta  
  
func nextRow(_ stmt:OpaquePointer) -> Bool {  
    let result = sqlite3_step(stmt)  
  
    let next: Bool = result == SQLITE_ROW  
  
    return next  
}  
  
// Fecha o banco de dados  
  
func close() {  
    sqlite3_close(self.db)  
}  
  
func closeStatement(_ stmt:OpaquePointer) {  
    // Fecha o statement  
    sqlite3_finalize(stmt)  
}  
  
// Retorna o último erro de SQL
```

```
func lastSQLException() -> String {

    var err:UnsafePointer<Int8>? = nil

    err = sqlite3_errmsg(self.db)

    if(err != nil) {

        let s = NSString(utf8String: err!)

        return s! as String

    }

    return ""

}

// Lê uma coluna do tipo Int

func getInt(_ stmt:OpaquePointer, index:CInt) -> Int {

    let val = sqlite3_column_int(stmt, index)

    return Int(val)

}

// Lê uma coluna do tipo Double

func getDouble(_ stmt:OpaquePointer, index:CInt) -> Double {

    let val = sqlite3_column_double(stmt, index)
```

```
        return Double(val)

    }

    // Lê uma coluna do tipo Float

    func getFloat(_ stmt:OpaquePointer, index:CInt) -> Float {

        let val = sqlite3_column_double(stmt, index)

        return Float(val)

    }

    // Lê uma coluna do tipo String

    func getString(_ stmt:OpaquePointer, index:CInt) -> String {

        let s = sqlite3_column_text(stmt, index)

        if let s = s {

            return String(cString: s)

        }

        return ""

    }

    // Converte Int (swift) para CInt(C)
```



```
func toCInt(_ swiftInt : Int) -> CInt {

    let number : NSNumber = swiftInt as NSNumber

    let pos: CInt = number.int32Value

    return pos

}

}
```

Vamos comentar um pouco sobre a classe **SQLiteHelper**.

Métodos:

getFilePath(String): recebe como parâmetro o nome do banco de dados, com isso podemos buscar o caminho dele dentro do simulador ou de um device físico.

open(String): recebe o caminho do banco recuperado pelo método **getFilePath(String)** abre o banco de dados e retorna uma instância do banco.

createTable(): cria a tabela **Carro** com os devidos campos, e setando o campo ID como **primary key**, ou seja, é o ID único que controla a tabela.

save(Carro): recebe como uma instância da classe **Carro** e faz o insert dos carros recebidos no banco de dados.

execSql(String): recebe o SQL a ser executado, este método facilita o uso de qualquer comando SQL.

execSql(String, Array<AnyObject>): recebe uma string que é o comando SQL a ser executado e um array de parâmetros que são os valores do campo.

query(String): recebe o SQL (comando **SELECT**) a ser executado e permite fazer a consulta em qualquer tabela que exista no banco.

query(String, Array<AnyObject>): recebe o SQL a ser executado e um Array com campos a serem utilizados juntamente com o SQL.

Importante: depois desta introdução aos métodos da classe **SQLiteHelper** eu recomendo dar uma olhada posteriormente nos outros métodos que usei para encapsular o código do **SQLite** para melhorar o entendimento.

Só de ver a fonte não dá para entender muito bem o que está sendo feito, então vamos praticar que é a melhor forma para entendermos o código acima.

Agora vamos usar a classe **SQLiteHelper** em um projeto de persistência.

Praticando com SQLite

O que vamos fazer no projeto de **Persistência** é chamar a **URL** de algum tipo de carro e vamos incluir no banco de dados. Feito isso, vamos realizar a consulta ao banco de dados e exibir em um tableview, para isso precisaremos incluir um tableview na nossa storyboard e criar os devidos **Outlets** e **Actions**. Não podemos nos esquecer de implementarmos os protocolos da tableview para que ela possa de fato exibir carros, e, por fim, nossa tableview terá uma célula customizada e logo precisaremos criar uma classe para nossa célula.

Vamos lá.

Importante: para darmos continuidade precisaremos criar mais dois arquivos que são o **CarroDB.swift** e **CarroService.swift** classes que conterão nossas regras para a inclusão e leitura dos carros, bem como acesso a URL de carros que está em um servidor.

CarroDB.swift

```
import Foundation

class CarroDB {

    var db: SQLiteHelper

    init() {

        self.db = SQLiteHelper(database: "Carro.sqlite")

    }

    // Cria a tabela carros (apenas se não existe)

    func createTables() {

        let sql = "create table if not exists Carro (id integer primary key
autoincrement,nome text, desc text, url_foto text, url_info text, tipo text);"

        let result = db.execSQL(sql)

        print("createTables executado com sucesso \(result)")

    }

    func getCarrosByTipo(_ tipo: String) -> Array<Carro> {
```

```
var carros : Array<Carro> = []

let stmt = db.query("SELECT * FROM Carro where tipo = ?",
params:[tipo as AnyObject])

while (db.nextRow(stmt)) {

    let c = Carro()

    c.id = db.getInt(stmt, index: 0)

    c.nome = db.getString(stmt, index: 1)

    c.desc = db.getString(stmt, index: 2)

    c.url_foto = db.getString(stmt, index: 3)

    c.url_info = db.getString(stmt, index: 4)

    c.tipo = db.getString(stmt, index: 5)

    carros.append(c)

}

db.closeStatement(stmt)
```

```

        return carros

    }

    // Salva um novo carro ou atualiza se já existe id

    func save(_ carro: Carro) {

        if(carro.id == 0) {

            // Insert

            let sql = "insert or replace into Carro (nome, desc,
url_foto, url_info, tipo) VALUES (?, ?, ?, ?, ?);"

            let params = [carro.nome, carro.desc, carro.url_foto,
carro.url_info, carro.tipo]

            let id = db.execSQL(sql, params:params as
Array<AnyObject>!)

            print("Carro \(carro.nome), id: \(id) salvo com sucesso.")

            carro.id = Int(id)

        } else {

```

```
// Update
```

```
let sql = "update Carro set nome = ?, desc = ?, url_foto =  
?, url_info = ?, tipo = ?) where id = ? VALUES (?, ?, ?, ?, ?);"
```

```
let params = [carro.nome, carro.desc, carro.url_foto,  
carro.url_info, carro.tipo, carro.id] as [Any]
```

```
let id = db.execSQL(sql, params:params as  
Array<AnyObject>)
```

```
print("Carro \ \(carro.nome), id: \ \(id) \ \(carro.id) atualizado  
com sucesso.")  
  
}  
  
}
```

```
// Deleta o carro
```

```
func delete(_ carro: Carro) {  
  
    let sql = "delete from carro where id = ?"  
  
    let result = db.execSQL(sql, params: [carro.id as AnyObject])  
  
    print("delete carro com sucesso \ \(result)")  
  
}
```

```
// Deleta todos os carros do tipo informado
```

```
func deleteCarrosTipo(_ tipo: String) {  
  
    let sql = "delete from Carro where tipo = ?"  
  
    let result = db.execSQL(sql, params: [tipo as AnyObject])  
  
    print("delete carros com sucesso \ \(result)")  
  
}  
  
func close() {  
  
    // Fecha o banco de dados  
  
    self.db.close()  
  
}  
}
```

CarroService.swift

```
import Foundation
```

```
class CarroService {
```

```
    class func getCarrosByTipo(_ tipo: String, cache: Bool, callback:
```

```
@escaping (_ carros:Array<Carro>?, _ error:Error?) -> Void) {

    var db = CarroDB()

    // Busca os carros do banco de dados

    let carros : Array<Carro> = cache ? db.getCarrosByTipo(tipo) : []

    // Se existir no banco de dados retorna

    if(carros.count > 0) {

        db.close()

        for c in carros {

            print(c.nome)

            print(c.url_foto)

        }

        // Retorna os carros pela função de retorno

        callback(carros, nil)

        print("Retornando carros \ (tipo) do banco")

        return

    }

}
```



```
// Cria a URL e Request
```

```
let http = URLSession.shared
```

```
let url =
```

```
URL(string:"http://brunomarcas.com/servicos/files/carros_" + tipo + ".json")!
```

```
let request = URLRequest(url: url)
```

```
// Faz a requisicao HTTP
```

```
let task = http.dataTask(with: request, completionHandler:
{(data, response, error) -> Void in
```

```
    if(error != nil) {
```

```
        // Chama o callback de erro
```

```
        callback(nil, error!)
```

```
    } else {
```

```
        // Faz parser de JSON. Cria lista de carros.
```

```
let carros = CarroService.parserJSON(data!)
```

```
if(carros.count > 0) {
```

```
    db = CarroDB()
```

```
    db.deleteCarrosTipo(tipo)
```

```

        for c in carros {

            // Salva o tipo do carro

            c.tipo = tipo

            // Salva o carro no banco

            db.save(c)

        }

        db.close()

    }

DispatchQueue.main.sync(execute: {

    callback(carros, nil)

})

}

})

task.resume()

}

// Parser JSON

class func parserJSON(_ data: Data) -> Array<Carro> {

```

```

var carros : Array<Carro> = []

// Faz a leitura do JSON, converte para dictionary

do {

    let dict = try JSONSerialization.jsonObject(with: data,
options: JSONSerialization.ReadingOptions.mutableContainers) as! NSDictionary

    // Dictionary para todos os carros

    let jsonCarros: NSDictionary = dict["carros"] as!

NSDictionary

    let arrayCarros: NSArray = jsonCarros["carro"] as!

NSArray

    // Array de carros

    for obj in arrayCarros {

        let dict = obj as! NSDictionary

        let carro = Carro()

        carro.nome = dict["nome"] as! String

        carro.desc = dict["desc"] as! String

        carro.url_info = dict["url_info"] as! String

        carro.url_foto = dict["url_foto"] as! String
    }
}

```

```

        carros.append(carro)

    }

    } catch let error as NSError {

        print("Erro ao ler JSON \(error)")

    }

    // Retorna a lista de carros

    return carros

}

}

```

Importante: no código das classes **CarroDB** e **CarroService** possui comentários explicativos sobre cada passo, portanto fica mais fácil o entendimento.

Como temos muita coisa para fazer, vamos por partes. No arquivo **AppDelegate.swift** vamos incluir o código abaixo dentro do método **didFinishLaunchingWithOptions()**.

```

// SQLite

print("Criando banco de dados...")

let db = CarroDB()

db.createTables()

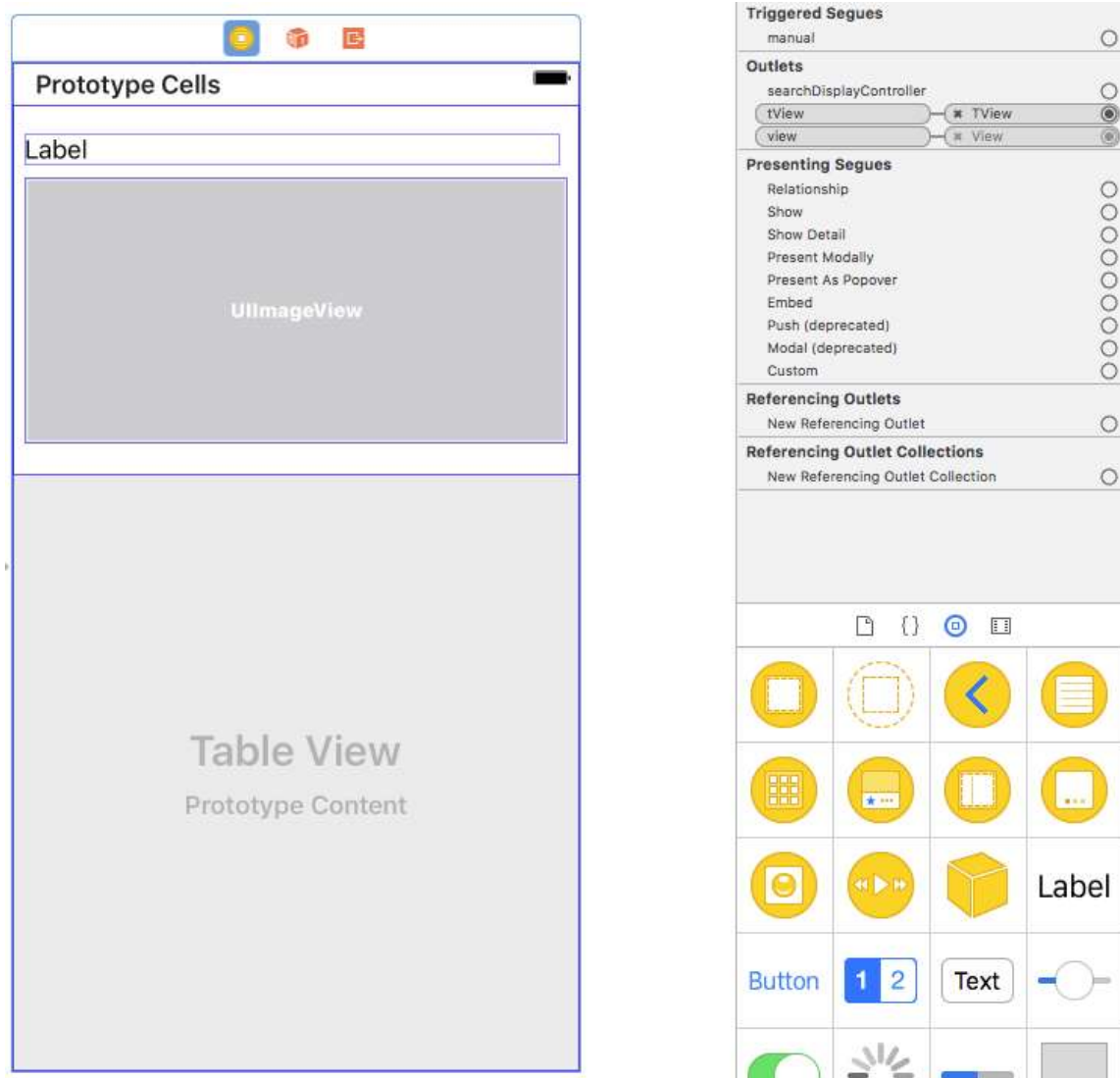
print("Banco de dados criado com sucesso.")

```

Vejam que instanciamos um objeto do tipo **CarroDB()** e chamamos o método **createTables()** para criar nosso banco de dados e a tabela **Carro**.

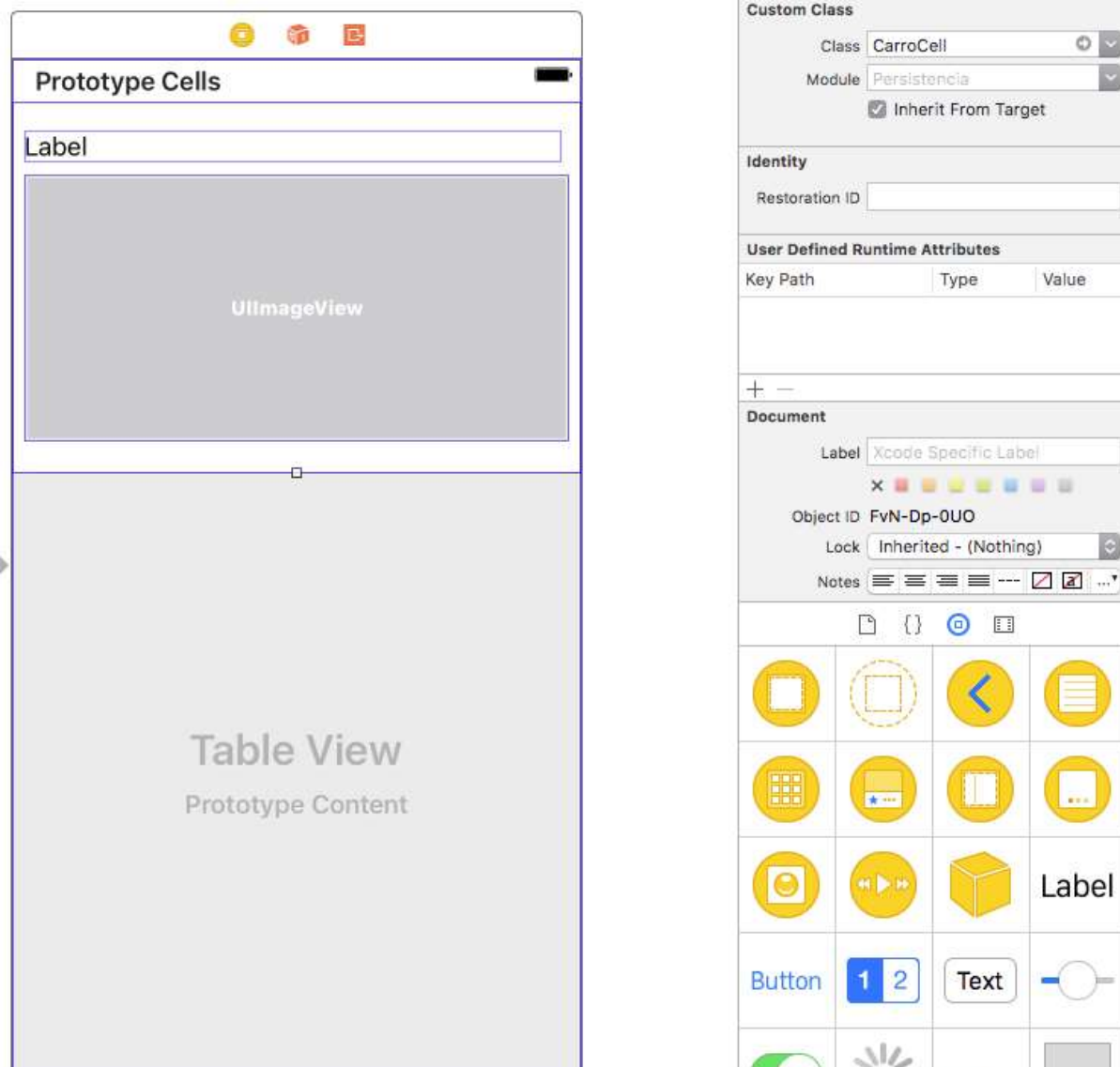
Agora vamos incluir nossa **tableview** na **viewController** que existe no arquivo **Main.storyboard** e criar a célula customizada. A figura 128 mostra como ficará nosso tableview e célula customizada, veja que o tableview foi ligado a um Outlet chamado **tView**.

Figura 128 – Criação de tableview e IBOutlets necessários



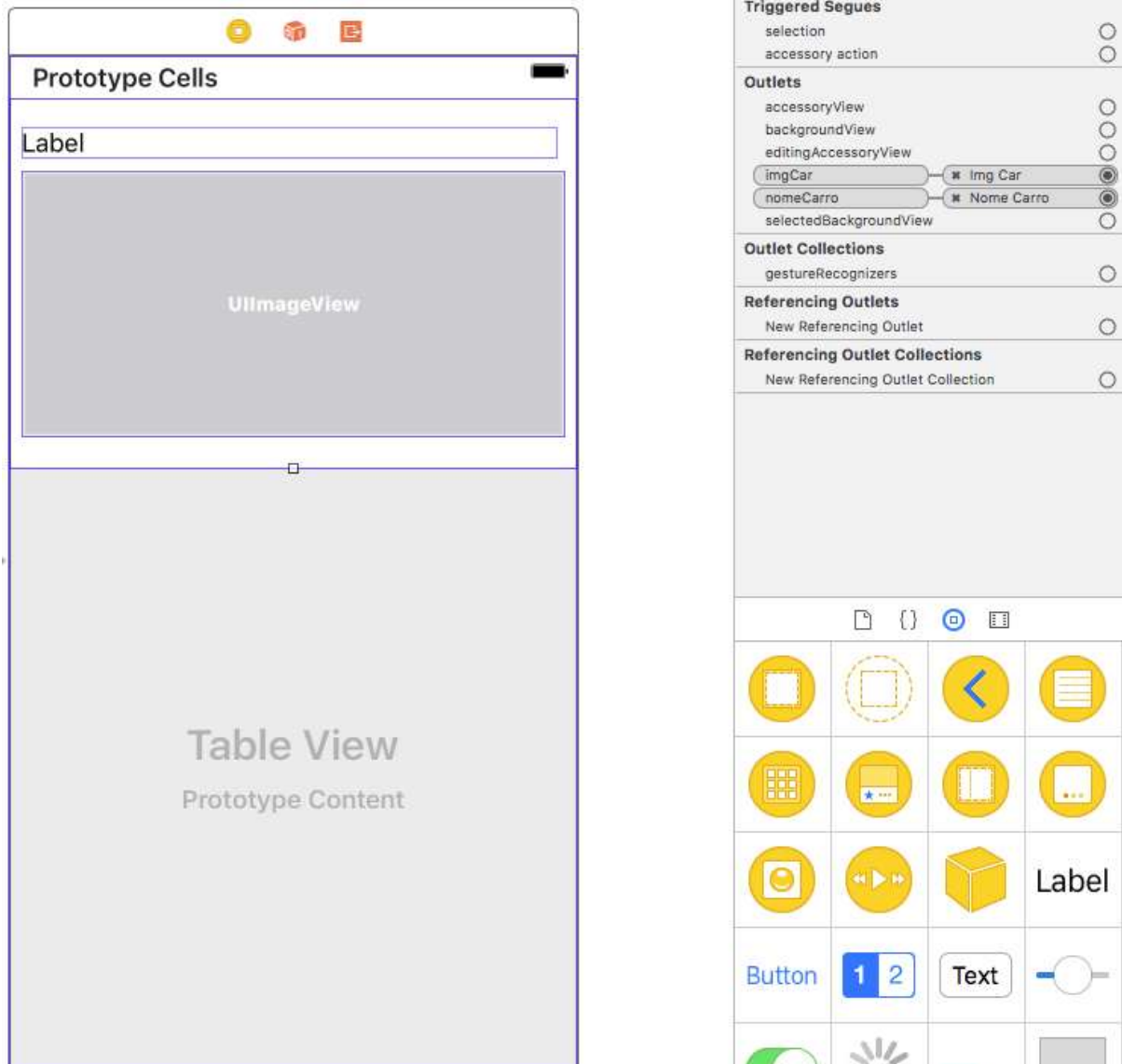
Ao selecionarmos nossa célula, veremos que criamos uma classe para ela **CarroCell.swift** e já fizemos a vinculação como mostra a figura 129.

Figura 129 – Vinculação de célula CarroCell



Agora, veja na figura 130 os **IBOutlets** que criamos para a nossa célula.

Figura 130 – Criação de IBOutlet para cada componente visual da célula



Importante: a classe **CarroCell.swift** deverá ser criada, pois é a classe onde serão feitas as ligações para os campos da célula.

Outro detalhe é o campo imagem que deve ser do tipo **DownloadImageView**, lembre-se que usamos essa classe para fazer download das imagens dos carros, usaremos da mesma forma neste projeto. Você deverá adicionar o arquivo **DownloadImageView.swift** no projeto **Persistencia**.

Criado nosso tableview, vamos agora utilizar as classes criadas **CarroDB**, **CarroService** e **SQLiteHelper**.

Como nossa **viewController** já está ligada ao arquivo **ViewController.swift** vamos deixar assim e utilizá-lo.

ViewController.swift

```
import UIKit

class ViewController: UIViewController, UITableViewDelegate,
UITableViewDataSource {

    var carro: Array<Carro> = []

    var type = "classicos"

    @IBOutlet var tableView: UITableView!

    override func viewDidLoad() {

        super.viewDidLoad()

    }

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



```
super.viewDidAppear(animated)
```

```
// Busca carros
```

```
self.getCars()
```

```
self.tableView.register(CarroCell.self, forCellReuseIdentifier: "Cell")
```

```
}
```

```
override func didReceiveMemoryWarning() {
```

```
super.didReceiveMemoryWarning()
```

```
// Dispose of any resources that can be recreated.
```

```
}
```

```
func getCars() {
```

```
let funcaoRetorno = { (_ carros:Array<Carro>?, error:Error?) ->
```

Void in

```
if let error = error {
```

```
let alert = UIAlertController(title: "ERROR",
```

```
message: error.localizedDescription, preferredStyle: UIAlertControllerStyle.alert)
```

```
alert.addAction(UIAlertAction(title: "OK", style:
```

```
UIAlertActionStyle.default, handler: nil))
```

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

```
} else if let carro = carros {
```

```
    self.carro = carro
```

```
    self.tableView.reloadData()
```

```
}
```

```
}
```

```
CarroService.getCarrosByTipo(type, cache:true, callback:
```

```
funcaoRetorno)
```

```
}
```

```
// MARK: - UITableView delegates
```

```
func tableView(_ tableView: UITableView, numberOfRowsInSectionSection: Int) -> Int {
```

```
    return self.carro.count
```

```
}
```

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
```

```
    let cell = tableView.dequeueReusableCell(withIdentifier:
```

```
"carroCell", for: indexPath as IndexPath) as! CarroCell
```

```
let linha = indexPath.row
```

```
let carro = self.carro[linha]
```

```
cell.nomeCarro.text = carro.nome
```

```
cell.imgCar.setUrl(url: carro.url_foto, cache: true)
```

```
return cell
```

```
}
```

```
}
```

Vamos por partes:

No método **viewDidAppear()** chamamos o método **getCars()** e registramos nossa célula com o método **register()**.

```
func getCars() {
```

```
let funcaoRetorno = { (_ carros:Array<Carro>?, error:Error?) -> Void in
```

```
if let error = error {
```

```
let alert = UIAlertController(title: "ERROR", message:
error.localizedDescription, preferredStyle: UIAlertControllerStyle.alert)
```

```

        alert.addAction(UIAlertAction(title: "OK", style:
UIAlertActionStyle.default, handler: nil))

        self.present(alert, animated: true, completion: nil)

    } else if let carro = carros {

        self.carro = carro

        self.tableView.reloadData()

    }

}

```

```

        CarroService.getCarrosByTipo(type, cache:true, callback:
funcaoRetorno)

    }

```

Reparem que o método **getCars()** tem uma variável que recebe uma função, ou seja, um bloco que faz o tratamento de erro exibindo um **alert** ou, caso a variável **carro** esteja preenchida, fará um **reload** no **tableview**. E por fim chama o método **getCarrosByTipo()** que está na classe **CarroService** passando a variável / bloco **funcaoRetorno** que criamos inicialmente.

O método **getCarrosByTipo()** da classe **CarroService** recebe os parâmetros **tipo**, que é o tipo de carro que vamos buscar; **cache**, que indica se vamos fazer cache das imagens dos carros; **callback**, que é o nosso bloco **funcaoRetorno**; e **Error**, que indica se ocorreu erro.

Na linha 18 buscamos os carros do banco de dados utilizando o método **getCarrosByTipo()** da classe **CarroDB**, esta vai realizar um comando select no banco de dados para ver se existe algum registro; caso não, é feita uma requisição **HTTP** para a nossa URL de carros; **esportivos**, **luxo** ou **classicos**.

Importante: vejam que não usamos **Alamofire** neste projeto, fizemos uma requisição a serviços externo de forma nativa usando o método **dataTask()**, que é um closure, da classe **URLSession** para fornecer outro exemplo de acesso a serviços externos.

Se der erro na requisição chamamos nosso **callback**, que é um bloco, caso contrário realizamos o parser com o método **parserJSON()** da classe **CarroService** e salvamos no banco de dados com o método **save**.

Por fim, temos a implementação de protocolos da tableview que já estamos familiarizados.

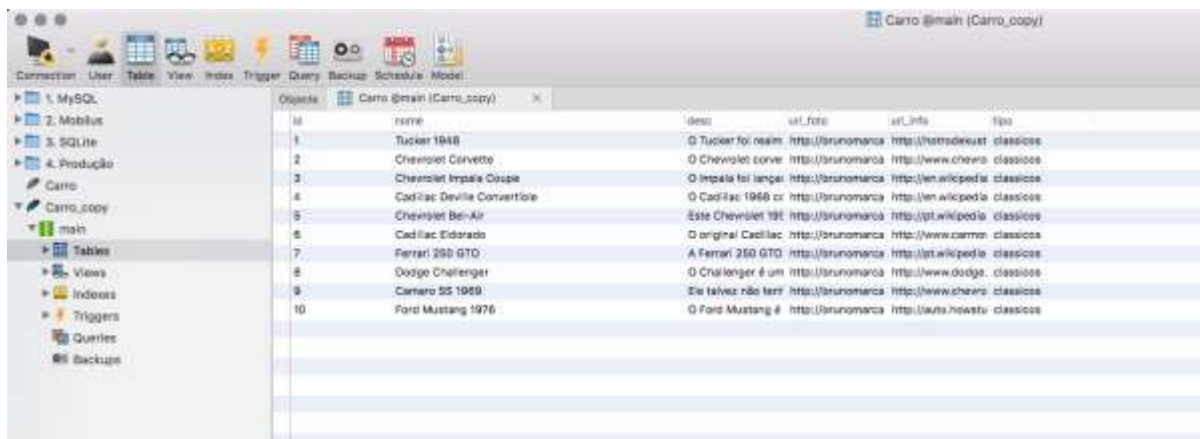
Ao executarmos, teremos o que é exibido na figura 131.

Figura 131 – Compilando projeto de Persistência



Quando abrimos o banco de dados em alguma ferramenta de administração, poderemos ver os dados dos veículos na tabela **Carro** como mostra a figura 132.

Figura 132 – Carros populados no banco de dados



id	nome	desc	url_foto	url_info	tipo
1	Tucker 1948	O Tucker foi o primeiro	http://brunomarcia	http://brunomarcia	classicos
2	Chevrolet Corvette	O Chevrolet corve	http://brunomarcia	http://www.chev	classicos
3	Chevrolet Impala Coupe	O Impala foi lançado	http://brunomarcia	http://en.wikipedia	classicos
4	Cadillac Deville Convertible	O Cadillac 1966 co	http://brunomarcia	http://en.wikipedia	classicos
5	Chevrolet Bel-Air	Este Chevrolet 196	http://brunomarcia	http://pt.wikipedia	classicos
6	Cadillac Eldorado	O original Cadillac	http://brunomarcia	http://www.cadmo	classicos
7	Ferrari 250 GTD	A Ferrari 250 GTD	http://brunomarcia	http://pt.wikipedia	classicos
8	Dodge Challenger	O Challenger é um	http://brunomarcia	http://www.dodge	classicos
9	Camaro SS 1969	Ele talvez não ten	http://brunomarcia	http://www.chev	classicos
10	Ford Mustang 1976	O Ford Mustang é	http://brunomarcia	http://auto.howsto	classicos

Importante: caso precise popular registros iniciais ao seu projeto, pode criar um script e colocar dentro do seu código ou pode abrir o arquivo **.sqlite** em algum gerenciador e fazer as devidas inclusões, administrações em tabelas etc.

Dica: o iOS também possui um **ORM** para manutenção de banco de dados **SQLite** chamado **CoreData** que também é bastante útil.

Capítulo 7. APIs proprietárias

A Apple possui várias APIs / SDKs que ela oferece para o uso dos desenvolvedores, veja algumas delas: acesso à câmera, acesso aos contatos, acesso às imagens da câmera, acesso aos mapas, acesso à iCloud etc. Apple faz isso para facilitar o trabalho dos desenvolvedores. Mas para usar essas APIs é necessária a permissão do usuário, ou seja, todo app que precise acessar algum dado sensível do usuário, como as APIs que citei acima, precisará pedir permissão para ele.

Vamos agora criar um projeto para acessar a galeria de fotos, ou seja, as fotos que tiramos e vão direto para o “rolo da câmera”.

Este projeto será como se fosse uma tela de perfil de usuário, contendo foto do usuário, nome e e-mail, como mostra a figura 133.

Figura 133 – Tela de perfil do usuário



Não precisaremos criar nenhuma classe extra para esse projeto, vamos usar a própria **ViewController.swift** já criada com o projeto.

ViewController.swift

```
import UIKit
```

```
class ViewController: UIViewController {
```

```
    @IBOutlet weak var imgPerfil: UIImageView!
```

```
    @IBOutlet var userName: UILabel!
```

```
    @IBOutlet var email: UILabel!
```

```
// MARK: - Properties
```

```
let imagePicker = UIImagePickerController()
```

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

```
    super.viewDidLoad()
```

```
    // Do any additional setup after loading the view, typically from a nib.
```



```

        self.customizeControls()

    }

    override func didReceiveMemoryWarning() {

        super.didReceiveMemoryWarning()

        // Dispose of any resources that can be recreated.

    }

    // MARK: - Methods

    func customizeControls() {

        self.imgPerfil.layer.masksToBounds = true

        self.imgPerfil.layer.cornerRadius =
self.imgPerfil.frame.size.width / 2

    }

    @IBAction func showCamera(_ sender: Any) {

        guard UIImagePickerController.isSourceTypeAvailable(.camera)

    else {

        let alert = UIAlertController(title: "Alert", message: "Este
dispositivo não tem câmera", preferredStyle: UIAlertControllerStyle.alert)

```

```

        alert.addAction(UIAlertAction(title: "OK", style:
UIAlertActionStyle.default, handler: nil))

        self.present(alert, animated: true, completion: nil)

        return
    }

    imagePicker.sourceType = .camera

    imagePicker.delegate = self

    present(imagePicker, animated: true)
}

@IBAction func showPhotoLibrary(_ sender: Any) {

    guard
UIImagePickerController.isSourceTypeAvailable(.photoLibrary) else {

        let alert = UIAlertController(title: "Alert", message: "Não
foi possível abrir o rolo da câmera", preferredStyle: UIAlertControllerStyle.alert)

        alert.addAction(UIAlertAction(title: "OK", style:
UIAlertActionStyle.default, handler: nil))

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

```

```

        return

    }

    imagePicker.sourceType = .photoLibrary

    imagePicker.delegate = self

    present(imagePicker, animated: true)

}

}

extension ViewController: UIImagePickerControllerDelegate,
UINavigationControllerDelegate {

    func imagePickerController(_ picker: UIImagePickerController,
didFinishPickingMediaWithInfo info: [String : Any]) {

        defer {

            picker.dismiss(animated: true)

        }

        guard let image = info[UIImagePickerControllerOriginalImage]
as? UIImage else {

            return

        }
    }
}

```

```
        imgPerfil.contentMode = .scaleAspectFill

        imgPerfil.image = image
    }

    func imagePickerControllerDidCancel(_ picker:
UIImagePickerController) {

        defer {

            picker.dismiss(animated: true)

        }

        print("cancelamento")

    }

}
```

Vamos agora comentar o código acima.

Criamos **Outlets** para os componentes visuais que incluímos na storyboard.

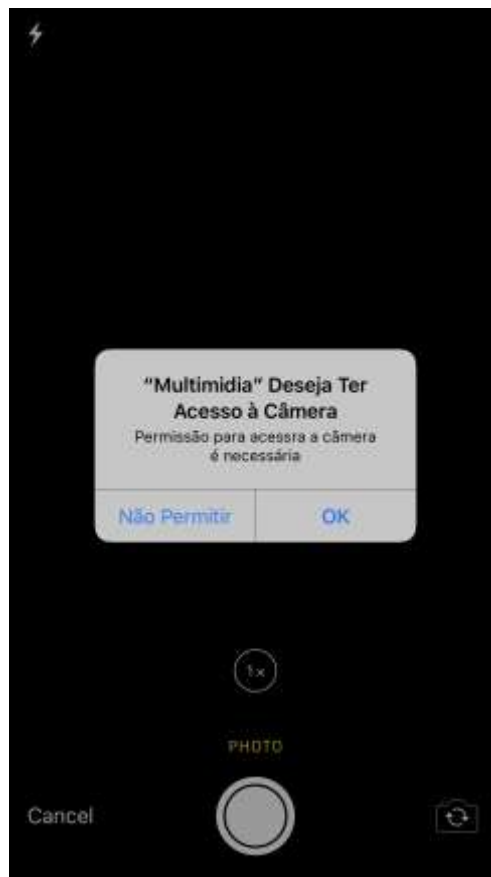
Criamos um método chamado **customizeControls()** para customizar a imagem de perfil, pois ela será redonda, usamos as propriedades **layer.maskToBounds** e **layer.cornerRadius** para setar o raio que o **UIImageView** deverá seguir, ao rodar nossa imagem será um círculo.

Criamos **Actions** também **showCamera()** e **showPhotoLibrary()** para acessar a câmera e o rolo da câmera (biblioteca de fotos) respectivamente.

```
@IBAction func showCamera(_ sender: Any)
```

Importante: Ao abrir o app pela primeira vez será pedida ao usuário a permissão para acessar a câmera, como mostra figura 134.

Figura 134 – Pendido acesso à câmera



Para exibir essa mensagem precisamos incluir uma chave no arquivo **info.plist** (selecione o projeto no **xCode** e vá na aba **Info**) do nosso projeto, a chave é exibida na figura 135.

A chave é **Privacy - Camera Usage Description** e deve ser do tipo **String** e conter uma mensagem de o porquê seu app precisa de acessar as fotos do usuário.

Figura 135 – Inclusão de chave para acesso à câmera.

Key	Type	Value
Required device capabilities	Array	(1 item)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Main storyboard file base name	String	Main
Bundle version	String	1
Launch screen interface file base name	String	LaunchScreen
Executable file	String	\$(EXECUTABLE_NAME)
Application requires iPhone environm...	Boolean	YES
Bundle versions string, short	String	1.0
Supported interface orientations	Array	(3 items)
Bundle OS Type code	String	APPL
Privacy - Camera Usage Descrip...	String	Permissão para acessar a câmera é necessária
Localization native development region	String	\$(DEVELOPMENT_LANGUAGE)
Supported interface orientations (iPad)	Array	(4 items)
Bundle name	String	\$(PRODUCT_NAME)

Criamos esse método para ser chamado quando o usuário clicar no botão “**camera**”. Usamos uma nova declaração condicional do **Swift, guard** que requer a execução de uma condição para sair do bloco atual. Neste caso a condição é: `UIImagePickerController.isSourceTypeAvailable(.camera)`

Todas as declarações criadas dentro de um bloco **guard** estão disponíveis para o resto da função ou bloco. Usando **return** para sair do bloco ou parar a execução quando for necessário.

Feito isso setamos o **sourceType** do **imagePicker** que criamos, indicando que queremos abrir a câmera e setamos o delegate do **imagePicher** para **self**.

Por fim usamos o método nativo **present()** para abrir a **viewController** da câmera (esta **viewController** é implementada pela classe **UIImagePickerController**).

`@IBAction func showPhotoLibrary(_ sender: Any)`

O método acima é para chamar o rolo da câmera e funciona da mesma forma que o método **showCamera()**, apenas trocamos o **sourceType**.

Importante: podemos unificar ambos os métodos, pois são muito comuns, mas deixarei isso para vocês fazerem como exercício.

Note que nossa implementação dos delegates foi um pouco diferente, usamos uma **Extension**, vamos entender melhor.

```
extension ViewController: UIImagePickerControllerDelegate,  
UINavigationControllerDelegate {  
  
}
```

Extension como vimos anteriormente serve para estendermos uma classe, no caso estendemos a classe **ViewController**, dessa forma o uso de delegates fica organizado no código.

Podemos estender também tipos como **String**, **Date**, **Int** etc. incluindo métodos de acordo com a necessidade do app que estivermos criando.

```
func imagePickerController(_ picker: UIImagePickerController,  
didFinishPickingMediaWithInfo info: [String : Any])
```

O método acima é usado quando o usuário termina de escolher uma foto, ou seja, clica no botão “**Usar Foto**”. O código é bastante parecido com **showCamera()** e **showPhotoLibrary()**, com uma diferença, pois alteramos a propriedade **contentMode** da nossa imagem setando para **.scaleAspectFill**, ou

seja, definindo para a imagem ocupar todo o espaço do componente **UIImageView**, mas sem distorcer a imagem.

```
func imagePickerControllerDidCancel(_ picker: UIImagePickerController)
```

Método acima é chamado quando o usuário clica no botão “**Cancel**” como mostra a figura 134.

Capítulo 8. Apple TV

No dia 9 de setembro de 2015, em São Francisco, Califórnia, a Apple anunciou sua nova **Apple TV** (4ª geração), apostando em uma nova forma de consumir conteúdo através de aparelhos de TV.

Mas o que é uma Apple TV, muitas pessoas não sabem, então, vamos a uma breve explicação.

Apple TV é um set-top box (aparelho que se conecta a uma TV “caixinha preta”), veja na figura 133 a Apple TV de 4ª geração.

Figura 133 – Apple TV – 4ª geração



Mas o que ela faz? Simples, ela transforma qualquer TV com uma entrada HDMI em uma smartTV.

O sistema operacional da Apple TV é uma derivação do **iOS**, o **tvOS**, ou seja, a Apple pegou o iOS e o transformou em um sistema operacional para sua nova Apple TV, já estão percebendo onde quero chegar?

No evento de lançamento do iPhone 6s e 6s Plus, a Apple também lançou sua nova Apple TV (4ª geração) e deixou o mercado louco, pois juntamente ela lançou também o SDK para desenvolvimento.

Em resumo, podemos fazer aplicativos para a Apple TV usando **Objc-C** ou **Swift**, tudo que estamos aprendendo poderá ser usado para este novo mercado, sem falar que o desenvolvedor iOS fica mais valorizado no mercado pelo fato de atuar em vários ambientes, alguns apps já estenderam suas funcionalidades para Apple TV, como: Netflix, Prime Video da Amazon, Facebook, Globo Play etc.

O tvOS vem com a Siri, que é a assistente virtual, ou seja, tudo pode ser feito, também, através de comandos de voz.

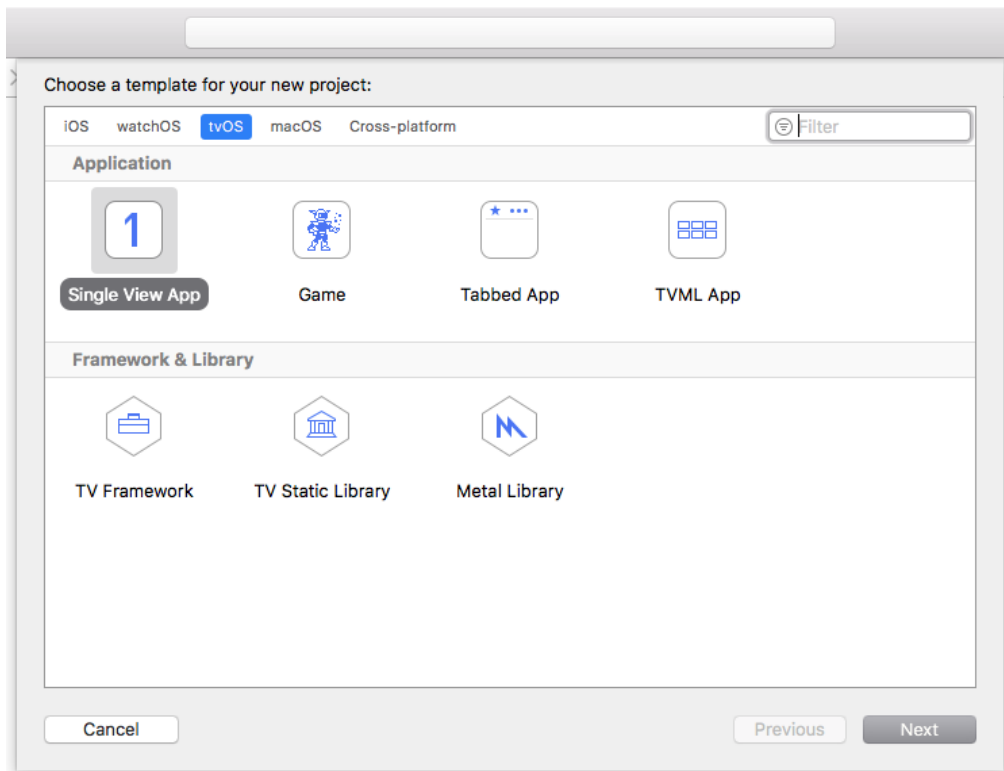
Desenvolvendo para Apple TV

Agora vamos começar nosso projeto da Apple TV, lembrando que tudo que vimos nos capítulos anteriores vale também para Apple TV, pois o ambiente de desenvolvimento é exatamente o mesmo, mudando apenas o tipo de projeto.

Vamos criar um projeto do tipo **tvOS**. Vamos em **File -> New -> New Project** e escolha **tvOS e Single View Application**.

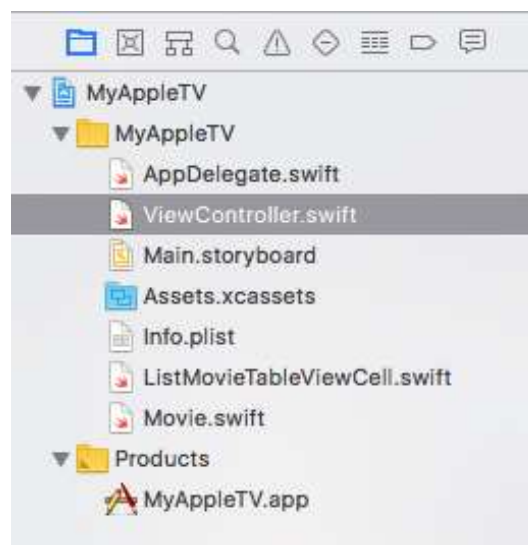
Nesse projeto vamos exibir uma tableview listando alguns filmes e, ao clicar, exibiremos um alert com o título do filme clicado. A ideia é ser bem simples mesmo para mostrar que o desenvolvimento é praticamente o mesmo tanto para apps quanto para Apple TV, bem como AppleWatch também.

Figura 134 – Criando projeto para Apple TV



Feito isso, escolha onde quer salvar seu projeto e pronto. A figura 135 mostra a estrutura de arquivos deste projeto.

Figura 135 – Estrutura de arquivos projeto AppleTV



Vamos agora aos detalhes de cada arquivo criado.

Main.storyboard

Figura 136 – Main.storyboard



Movie.swift

```
import Foundation
```

```
class Movie: NSObject {

    var movieTitle: String = ""

    var imageName: String = ""

    var desc: String = ""
```

```
init(title: String, image: String, desc: String) {  
  
    self.movieTitle = title  
  
    self.imageName = image  
  
    self.desc = desc  
  
}  
  
}
```

O arquivo acima é a nossa classe de modelo para os filmes que iremos exibir na tableview, veja que criamos properties e um construtor para a classe, pois esta é bem simples.

ListMovieTableViewCell.swift

```
import UIKit  
  
class ListMovieTableViewCell: UITableViewCell {  
  
    @IBOutlet var imgCell: UIImageView!  
  
    @IBOutlet var movieTitle: UILabel!  
  
    @IBOutlet var desc: UITextView!
```

```
override func awakeFromNib() {  
  
    super.awakeFromNib()  
  
    // Initialization code  
  
}  
  
override func setSelected(_ selected: Bool, animated: Bool) {  
  
    super.setSelected(selected, animated: animated)  
  
    // Configure the view for the selected state  
  
}  
}
```

A classe acima é a célula do nosso tableview, visto que criamos uma célula customizada para exibir os filmes, veja que nela incluímos os **Outlets**.

ViewController.swift

```
import UIKit  
  
class ViewController: UIViewController, UITableViewDelegate,
```

```
UITableViewDataSource {
```

```
    var list: Array<Movie> = []
```

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

```
        super.viewDidLoad()
```

```
        self.getMovies()
```

```
    }
```

```
    override func didReceiveMemoryWarning() {
```

```
        super.didReceiveMemoryWarning()
```

```
        // Dispose of any resources that can be recreated.
```

```
    }
```

```
    func getMovies() {
```

```
        let m1 = Movie(title: "The Five-Years Engagement", image:
"poster1", desc: "The Five-Year Engagement (Cinco Anos de Noivado (título no
Brasil) ou Espera Aí... Que Já Casamos (título em Portugal)) é um filme de comédia
romântica escrito, dirigido e produzido por Nicholas Stoller e estrelado por Jason
Segel e Emily Blunt como um casal cujo relacionamento se torna tenso quando...")
```

```
        let m2 = Movie(title: "DeadPool", image: "poster2", desc:
```

"Deadpool é um filme de ação e comédia americano dirigido por Tim Miller e distribuído pela 20th Century Fox que tem como protagonista o icônico personagem da Marvel que dá nome ao longa, sendo o oitavo título da franquia X-Men. A obra é estrelada por Ryan Reynolds no papel titular...")

```
let m3 = Movie(title: "The Terminator", image: "poster3", desc:
"The Terminator (no Brasil, O Exterminador do Futuro; em Portugal, O Exterminador Implacável) é um filme de ficção científica e suspense estadunidense de 1984, dirigido por James Cameron, sendo o primeiro da franquia Terminator. The Terminator foi aclamado pela crítica especializada...")
```

```
let m4 = Movie(title: "Homem Formiga", image: "poster4", desc:
"Ant-Man (Homem-Formiga (título no Brasil) ou O Homem-Formiga (título em Portugal)) é um filme americano de super-herói de 2015 baseado nos personagens da Marvel Comics de mesmo nome: Scott Lang e Hank Pym. É o décimo segundo filme do Universo Cinematográfico Marvel e o primeiro filme solo...")
```

```
list = [m1, m2, m3, m4]
```

```
}
```

```
// MARK: - UITableView delegates
```

```
func tableView(_ tableView: UITableView, numberOfRowsInSectionSection
section: Int) -> Int {

    return self.list.count

}
```



```
func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {

    let cell = tableView.dequeueReusableCell(withIdentifier: "Cell",
for: indexPath as IndexPath) as! ListMovieTableViewCell

    let movie = self.list[indexPath.row]

    cell.movieTitle.text = movie.movieTitle

    cell.imgCell.image = UIImage(named: movie.imageName)

    cell.desc.text = movie.desc

    return cell

}
```

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

    let movie = self.list[indexPath.row]

    let alert = UIAlertController(title: "Alert", message: "Filme
escolhido \ \(movie.movieTitle)", preferredStyle: UIAlertControllerStyle.alert)

    alert.addAction(UIAlertAction(title: "OK", style:
UIAlertActionStyle.default, handler: nil))

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

```
}  
  
}
```

Por fim temos a classe acima que é nossa controller, ela está vinculada diretamente à nossa **viewController** na **Main.storyboard**.

Nela instanciamos nossa classe **Movie** e criamos um **Array** que é a nossa lista de filmes e criamos um método chamado **getMovies()** para recuperar os filmes. Neste método incluímos manualmente itens ao nosso **Array** de filmes para que seja exibido na tableview.

Como estamos utilizando um tableview precisamos implementar os métodos delegates que iremos utilizar, são eles:

Retorna o número de linhas que a seção vai conter

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section:  
Int) -> Int
```

Responsável por cada célula é neste método que preenchemos dados na célula

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath:  
IndexPath) -> UITableViewCell
```

Responsável pelo click em uma célula

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

Ao compilar verá o resultado como mostra a figura 136.1

Figura 136.1 – App sendo compilado no simulador da Apple TV



Importante: Como a interação com Apple TV é feita via controle remoto precisamos clicar em **shift + command + R** para exibir o controler remoto.

E sempre que formos usar o simulador temos que usar através do controle, segure a tecla option para poder clicar e rolar a viewController.

Capítulo 9. Certificados e Provisionamentos

Nesse capítulo vamos entender melhor os certificados e provisionamentos que cercam os apps iOS e entendê-los.

Criando certificado

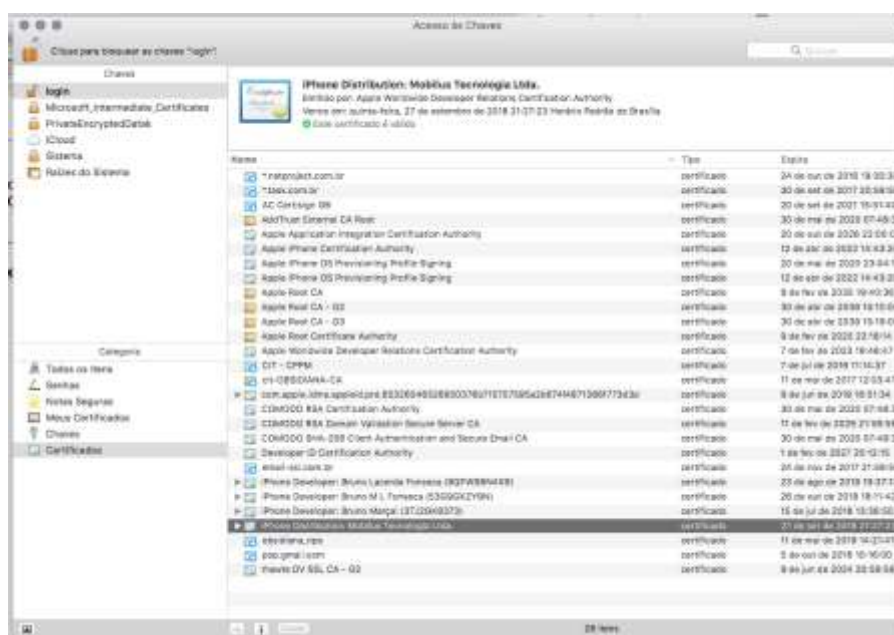
Vamos começar com o certificado, ou seja, ele é a autorização para seu Mac poder desenvolver app para iOS, watchOS, tvOS ou macOS.

A primeira coisa que precisamos fazer é gerar um arquivo de certificado via Acesso Chaves (Keychain).

Em seu Mac abra o aplicativo **Acesso às chaves**, **Keychain** se seu Mac estiver em inglês. A figura 137 exhibe a tela do **Acesso às chaves**.

Importante: para a criação de certificados e provisionamentos é necessário que já tenha feito o registro como desenvolvedor da Apple e pago o valor que escolheu.

Figura 137 – Programa acesso as chaves

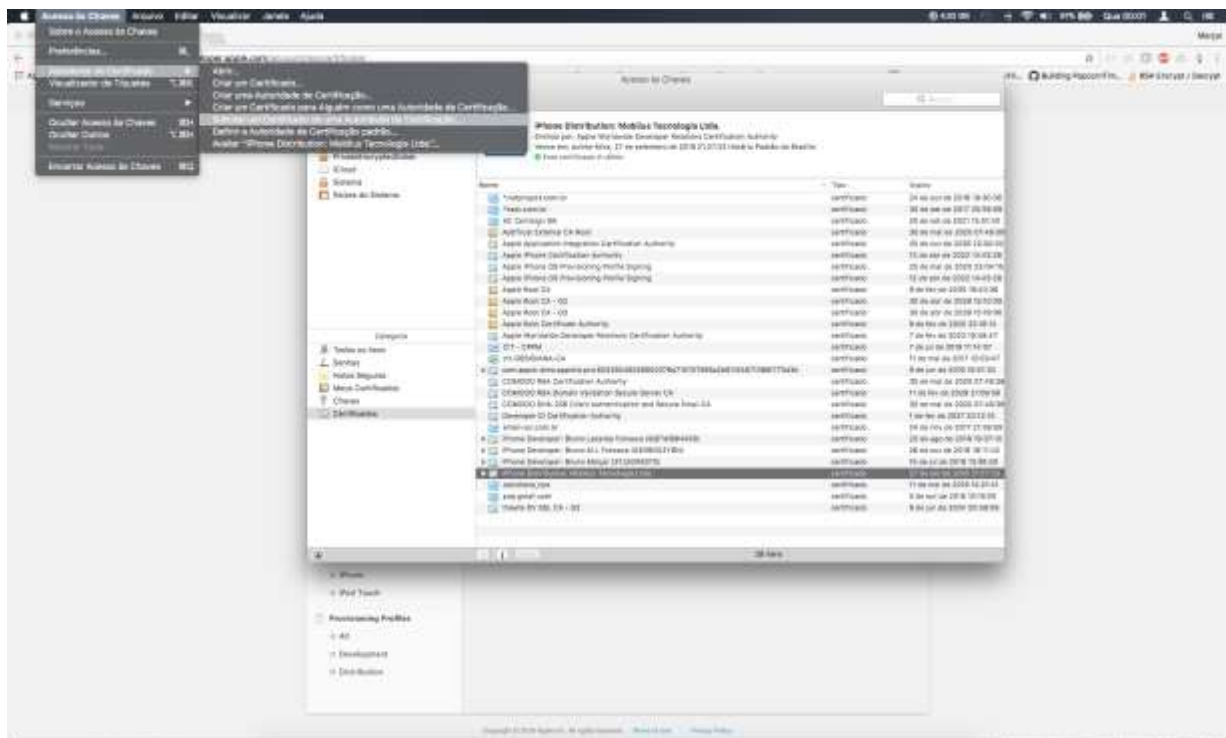


Agora vamos usá-lo para criar um arquivo de certificado que precisamos utilizar no site da Apple para gerar nosso certificado.

A figura 138 exibe como devemos criar este arquivo.

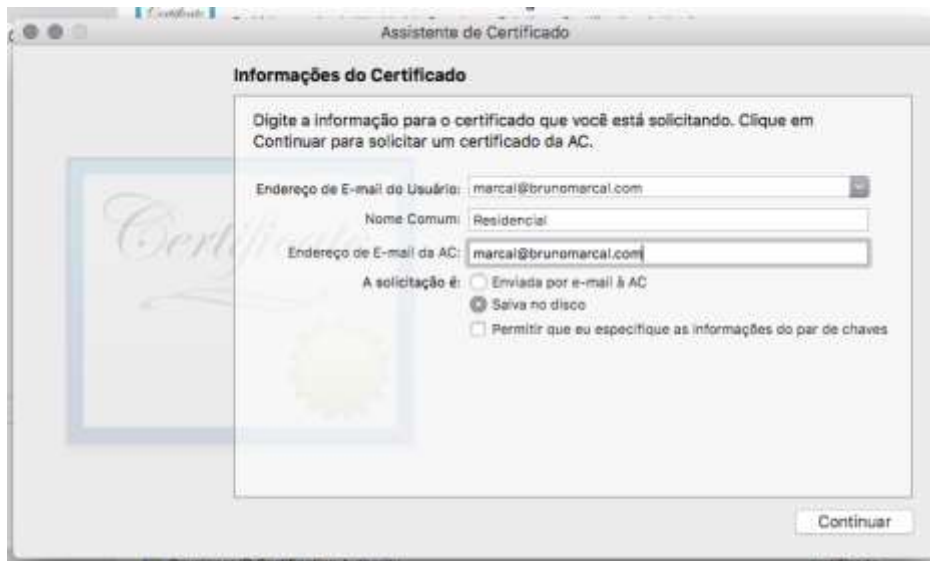
Acesse as **Chaves->Assistente de certificado->Solicitar um Certificado de uma Autoridade de Certificação.**

Figura 138 – Criando arquivo de certificado no Mac



Feito isso, aparecerá a tela exibida na figura 139.

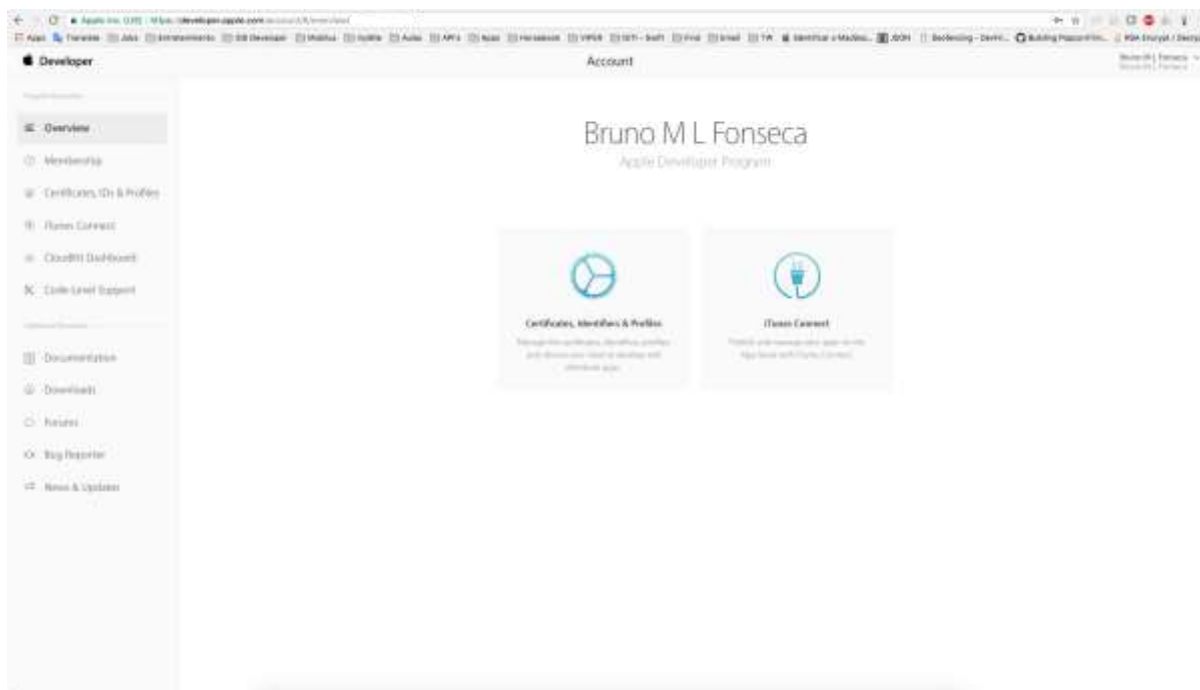
Figura 139 – Tela de informação para criação de certificado



Coloque seu e-mail nos campos **Endereço de E-mail do Usuário** e **Endereço de E-mail AC** e no campo “**A solicitação é:**” marque a opção **salvar no disco**, feito isso vai aparecer uma tela para salvar o arquivo, salve-o. Será salvo um arquivo com a extensão **.certSigningRequest**.

Agora vamos ao site da Apple <https://developer.apple.com/> e clique na opção Account e entre com seu **AppleID** que usou para cadastrar sua conta de desenvolvedor junto a Apple. A figura 140 exibe a tela inicial do site após o login.

Figura 140 – Site para desenvolvedor Apple



Acesse o item **Certificates, IDs, & Profiles** e veremos uma tela conforme figura 141.

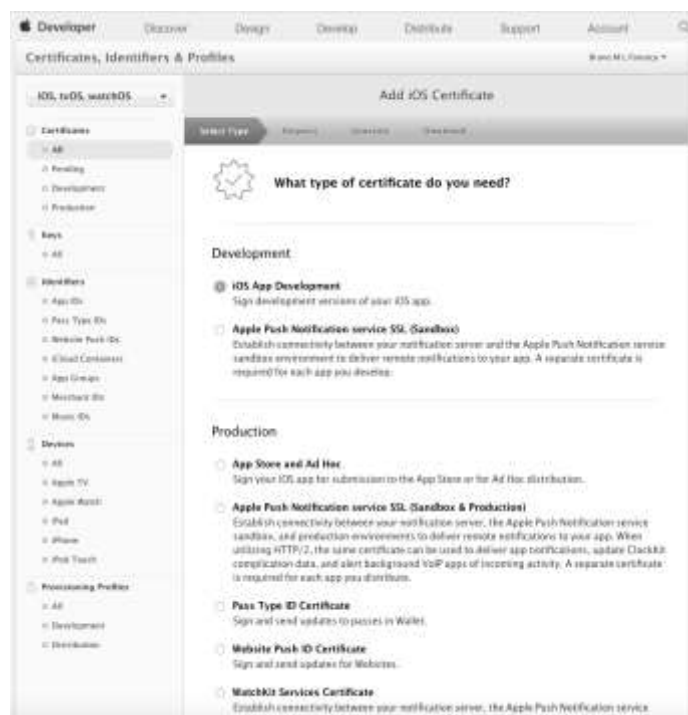
Figura 141 – Certificados, IDs e Profiles



Clique no botão “+” para adicionar um novo certificado. Clique no item **iOS App Development**, esse item é para criar um certificado de desenvolvimento.

Importante: a criação de certificado de produção segue o mesmo padrão só que o item a ser escolhido deve ser **App Store and Ad Hoc**.

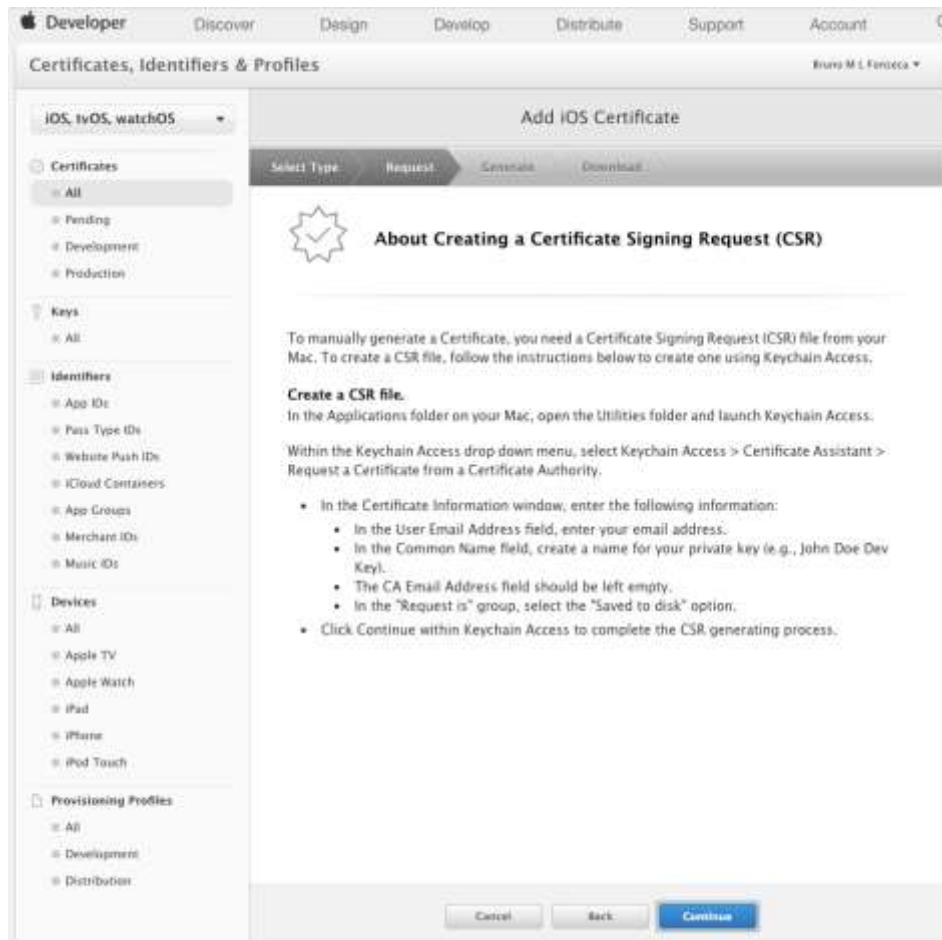
Figura 142 – Escolhendo tipo de certificado a ser criado



Feito isso, clique no botão “**Continuar**” no final da página.

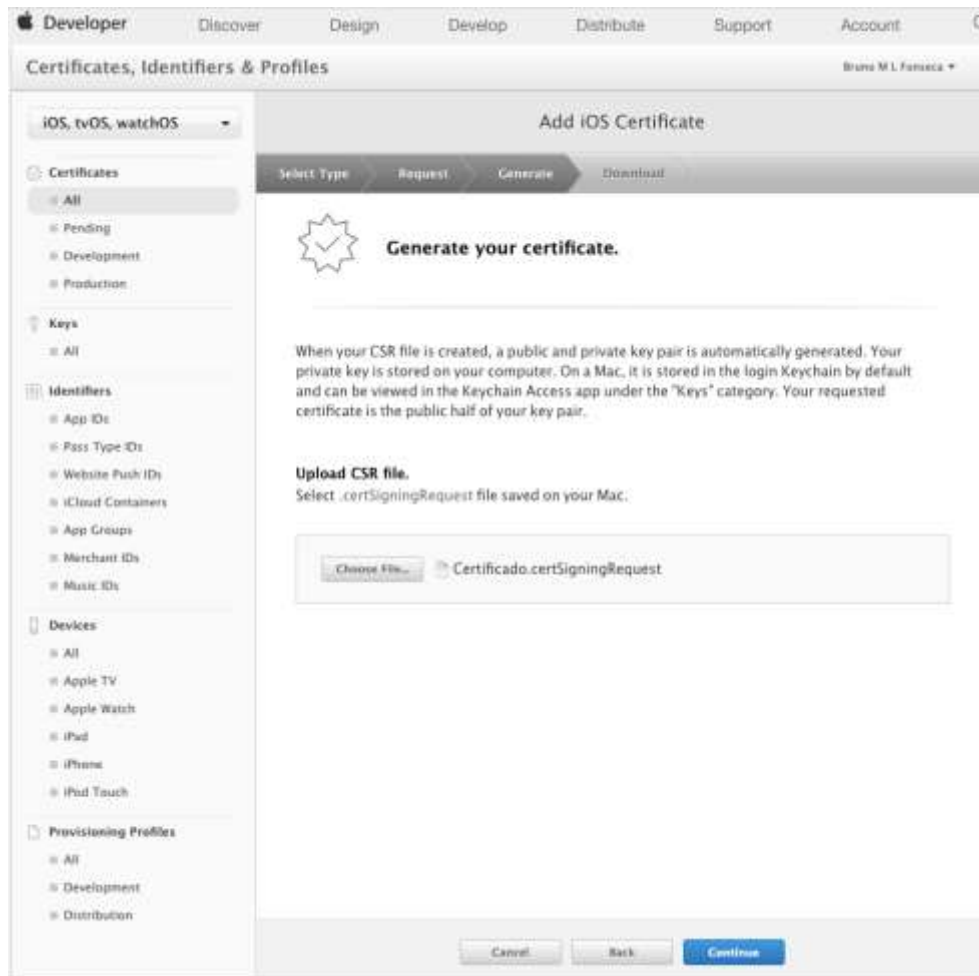
Será exibida uma tela conforme a figura 143. É só clicar em “**Continuar**” e escolhe o arquivo de certificado que criamos, veja figura 144.

Figura 143 – Tela informativa sobre arquivo de certificado



Escolha o arquivo de certificado que criamos e clique em **“Continuar”**, veja figura 144.

Figura 144 – Seleção do arquivo de certificado criado



Feito isso, o site da Apple irá gerar o certificado com base no arquivo **.certSigningRequest** e nos apresentará uma tela, como a figura 145 exibe, informando o sucesso e com um botão para download do certificado. Agora é só fazer o download e clicar duas vezes que o certificado será instalado no seu Mac e aparecerá no **Acesso chaves**, como mostra a figura 146.

Figura 145 – Tela para download do certificado

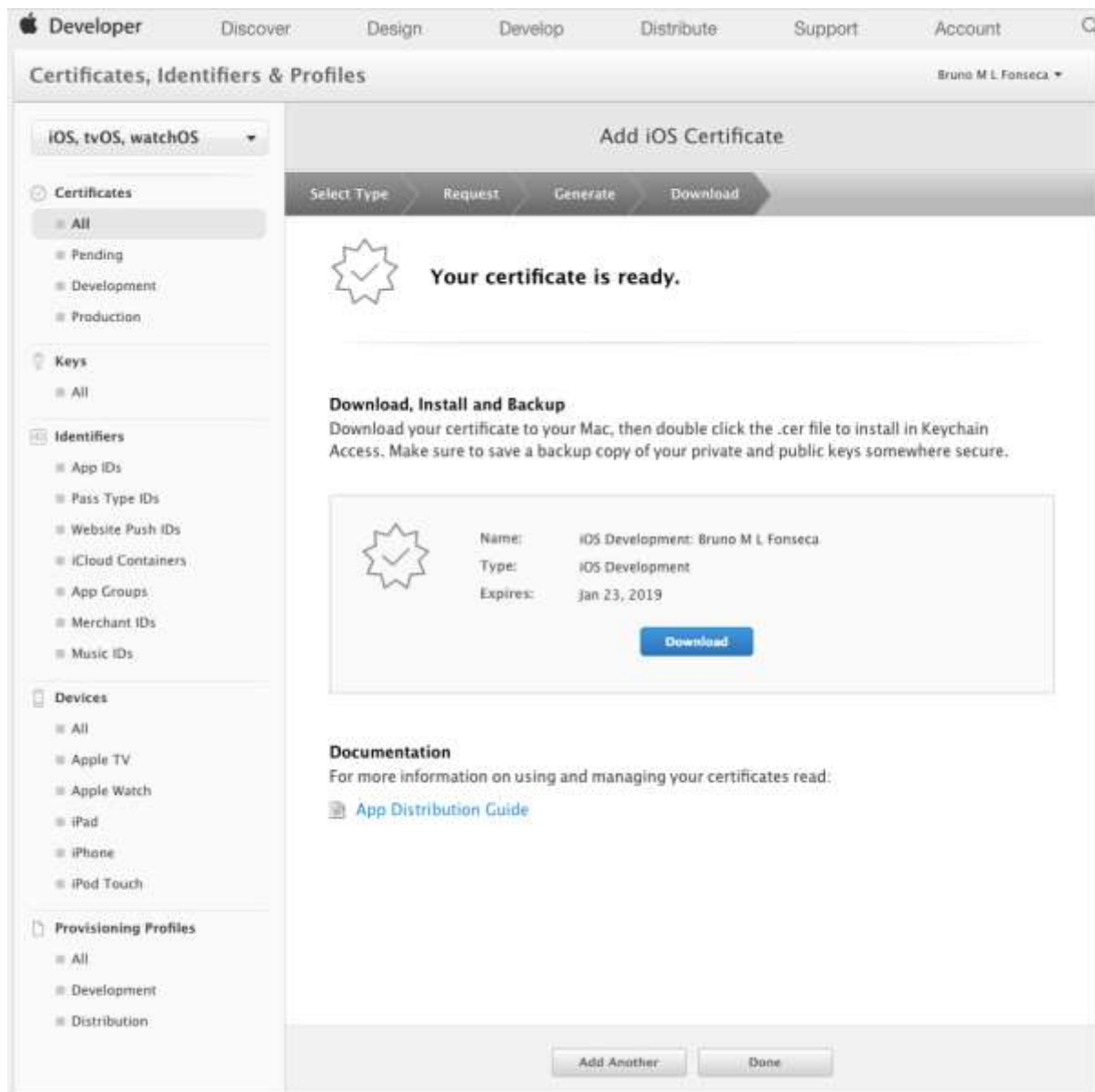
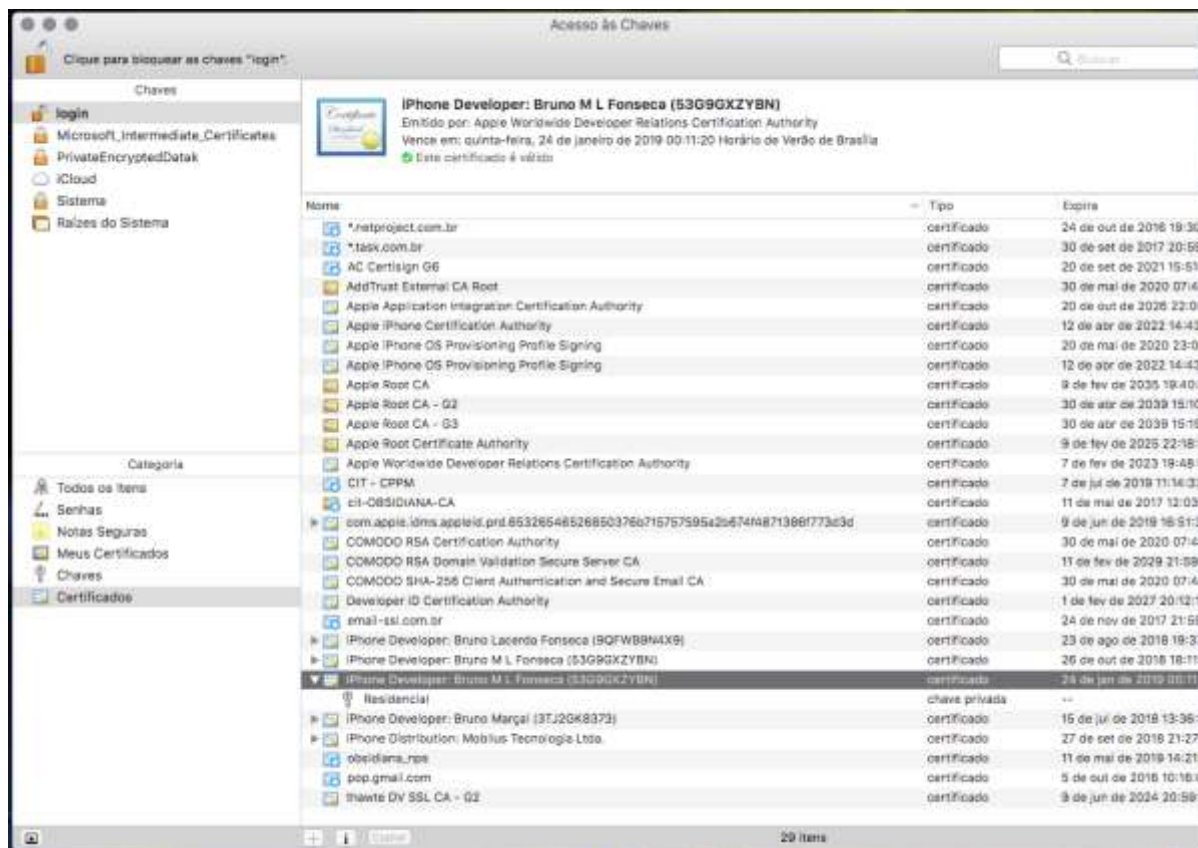


Figura 146 – Certificado instalado



Agora que seu certificado está instalado precisamos criar provisionamentos para o app, mas antes precisamos fazer uma série de passos, vamos lá.

Abra o **xCode** e clique em “**command + ,**”, aparecerá a tela conforme a figura 147.

Figura 147 – Tela de preferencias do xCode



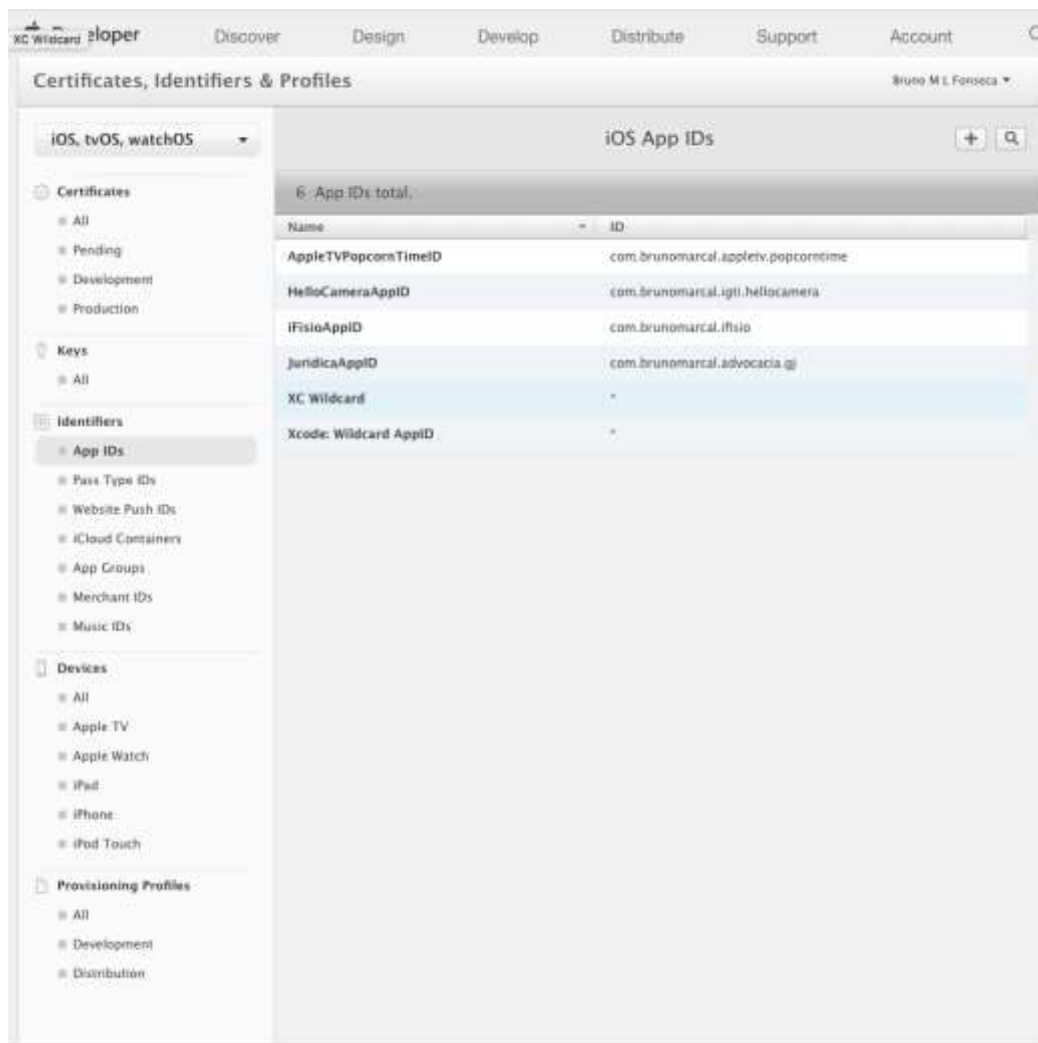
Na tela de preferências você deverá escolher o item “**Account**” e vincular seu **AppleID**.

Criando AppID

Agora vamos criar o **AppID** que é o **bundle** que usamos no app, no nosso caso foi **com.brunomarcas.multimidia**, portando precisamos criar um **AppID** com este bundle.

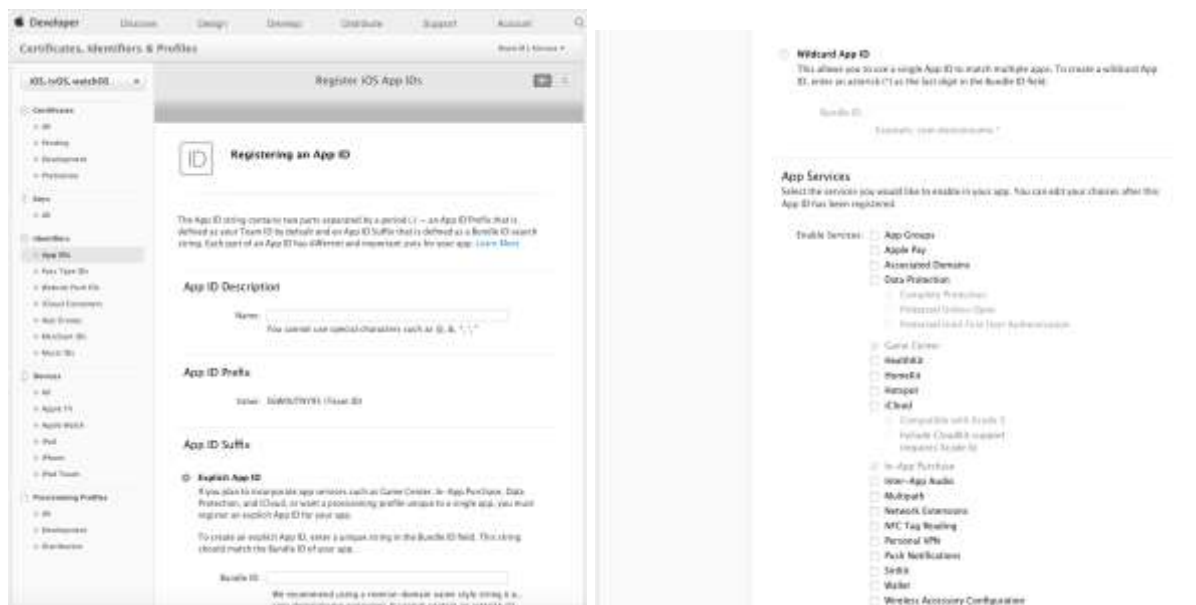
Clique no botão de “+” e conforme mostra a figura 149.

Figura 149 – Criando AppID



A tela para a criação de AppID irá aparecer, veja figura 150.

Figura 150 – Tela de criação de AppID



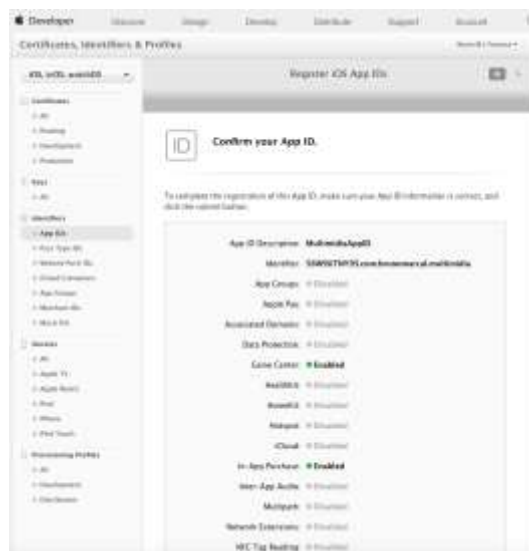
Como podem observar, a criação de um **AppID** é um pouco complexa, mas vamos ao importante.

Devemos preencher o campo **Name** com um nome para identificar nosso **AppID**, pode ser **MultimidiaAppID** e o campo **Bundle ID** preenchemos com o nosso bundle, no caso, **com.brunomarcas.multimidia**, clique no botão **“Continue”**.

Importante: é no **AppID** que estão os serviços que um app pode oferecer, como **iCloud**, **Push Notification**, **Apple Pay**, **Data Protection** etc. por isso **AppID** é baseado no bundle para ser único.

Vai ser exibido, como mostra a figura 151, uma tela de confirmação da criação do **AppID**, assim você pode conferir quais serviços ativou etc.

Figura 151 – Confirmação de criação de AppID



Feito isso, basta clicar no botão **“Register”** para criá-lo.

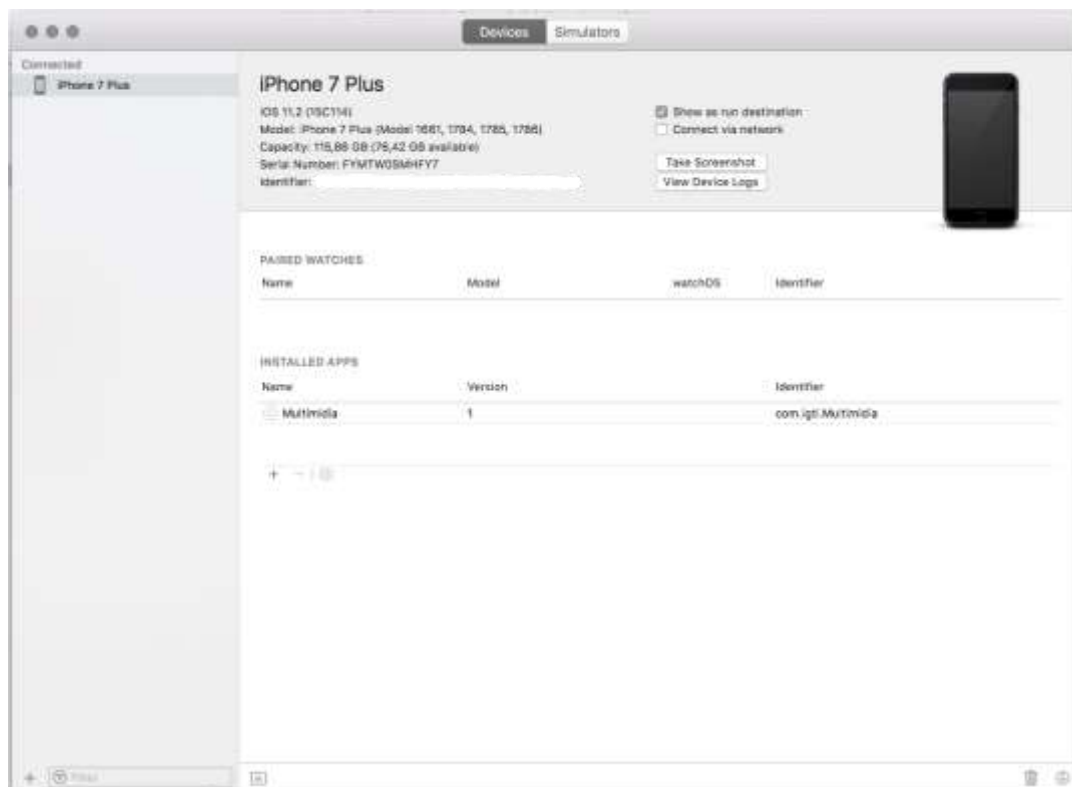
Incluindo Devices

Agora precisaremos incluir devices, ou seja, **UDIDs** de aparelhos Apple que iremos usar (iPhone, iPad, Apple TV etc.)

Importante: **UDID** é um número único da Apple que identifica o device, pode ser encontrado no **iTunes** ou no **xCode**.

No xCode vá em **Window->Device and Simulators** e irá aparecer a tela conforme figura 152, lembre-se que o device deve estar conectado ao Mac para poder visualizar o **UDID**, o campo **Identifier** é o **UDID**.

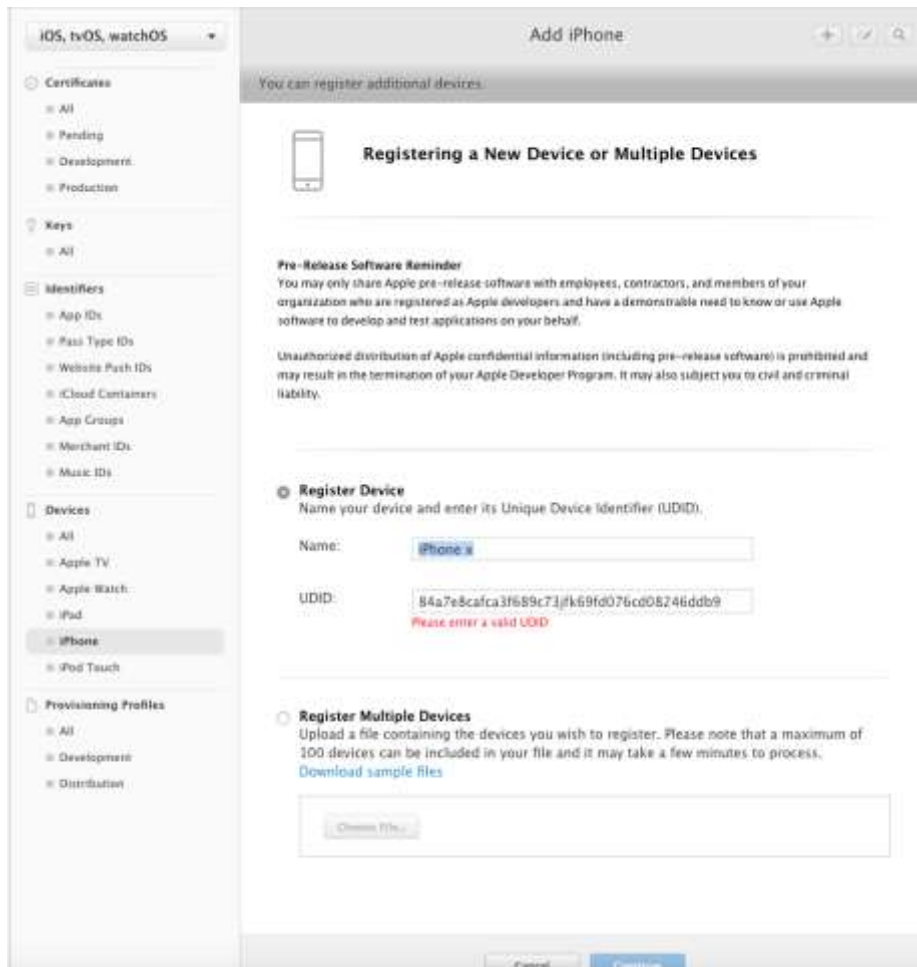
Figura 152 – Identificado UDID (Identifier) do device



Agora podemos incluir o device no site, veja figura 153, damos um nome e incluimos o **UDID** no campo. Notem que incluí um **UDID** incorreto, existe uma verificação que a Apple faz para validar esse número, portando deve ser de um device original.

Depois clique no botão “**Continue**” para incluir o device.

Figura 153 – Incluindo device



IOS, tvOS, watchOS

- Certificates
 - All
 - Pending
 - Development
 - Production
- Keys
 - All
- Identifiers
 - App IDs
 - Pass Type IDs
 - Website Push IDs
 - iCloud Containers
 - App Groups
 - Merchant IDs
 - Music IDs
- Devices
 - All
 - Apple TV
 - Apple Watch
 - iPad
 - iPhone**
 - iPod Touch
- Provisioning Profiles
 - All
 - Development
 - Distribution

Add iPhone

You can register additional devices.

Registering a New Device or Multiple Devices

Pre-Release Software Reminder
You may only share Apple pre-release software with employees, contractors, and members of your organization who are registered as Apple developers and have a demonstrable need to know or use Apple software to develop and test applications on your behalf.

Unauthorized distribution of Apple confidential information (including pre-release software) is prohibited and may result in the termination of your Apple Developer Program. It may also subject you to civil and criminal liability.

Register Device
Name your device and enter its Unique Device Identifier (UDID).

Name:

UDID:
Please enter a valid UDID

Register Multiple Devices
Upload a file containing the devices you wish to register. Please note that a maximum of 100 devices can be included in your file and it may take a few minutes to process.
[Download sample files](#)

Após incluído o device, ele irá aparecer no site, veja figura 154.

Figura 154 – Device incluído



Criando provisionamento

Agora poderemos criar o nosso provisionamento, que é a autorização para enviarmos nosso app para o device e essa autorização.

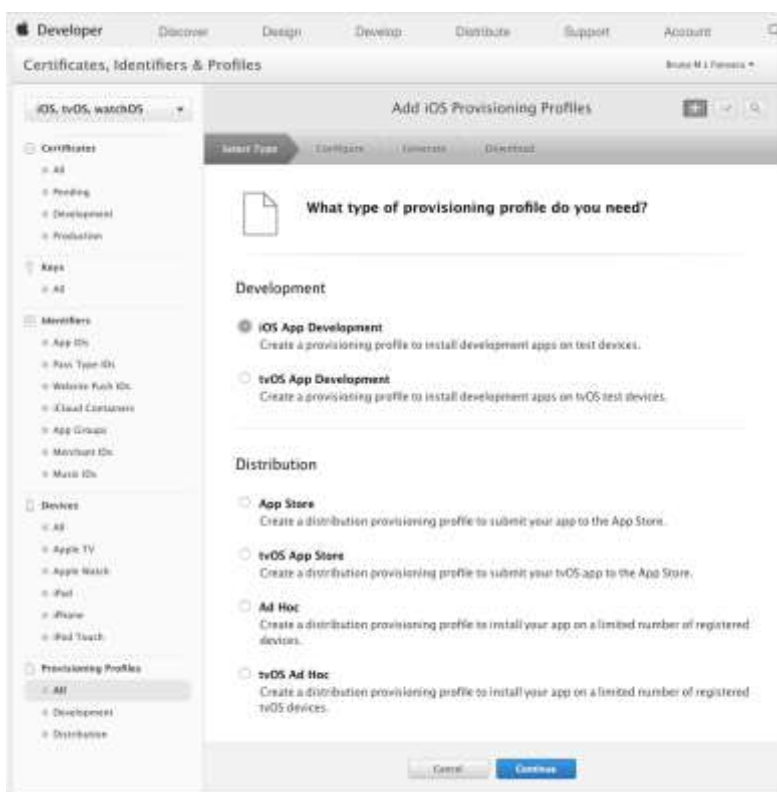
Figura 155 – Criando provisionamento



Acesse o item **All** da seção **Provisioning Profile**, agora clique no botão “+”, verá uma tela conforme figura 156.

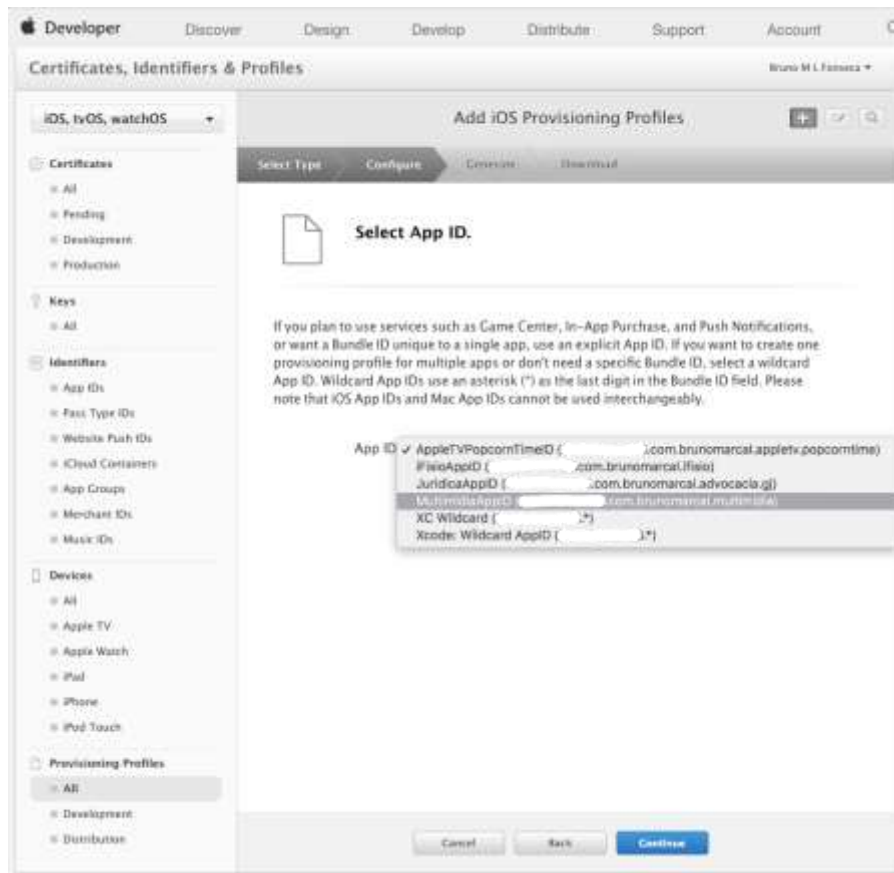
Selecione o item **iOS App Development** e clique no botão “Continue”.

Figura 156 – Criando provisionamento



Feito isso irá aparecer uma tela conforme figura 157 para escolher o **AppID** criado anteriormente.

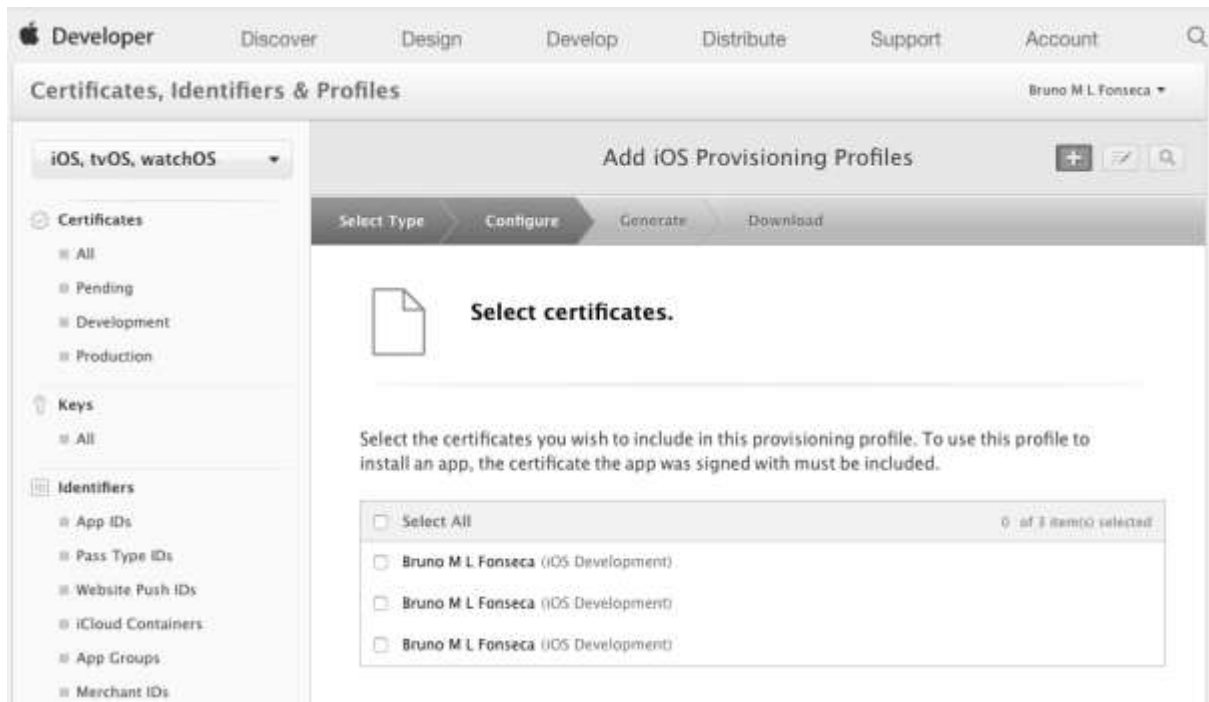
Figura 157 – Escolhendo AppID para criação de provisionamento



Escolha o AppID que criamos e clique no botão **“Continue”**.

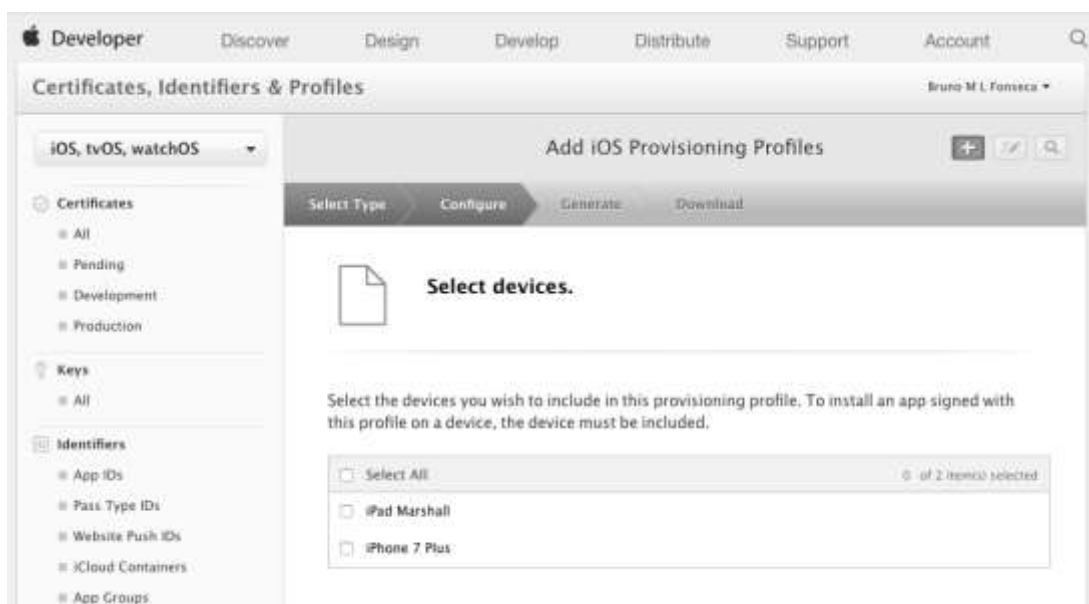
Após isso, aparecerá uma tela para escolher qual certificado irá usar, escolha o certificado que criamos e clique em **“Continue”**.

Figura 158 – Escolha de certificado para criação de provisionamento



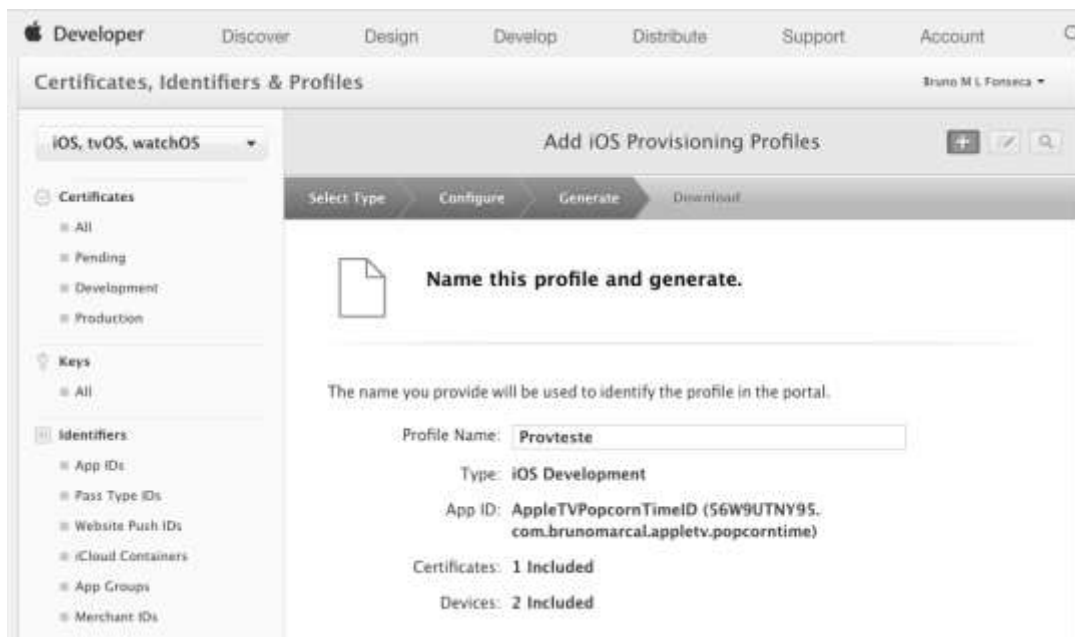
Feito isso, temos mais uma tela para escolha dos devices que poderão rodar o app em específico.

Figura 159 – Escolha de devices para provisionamento



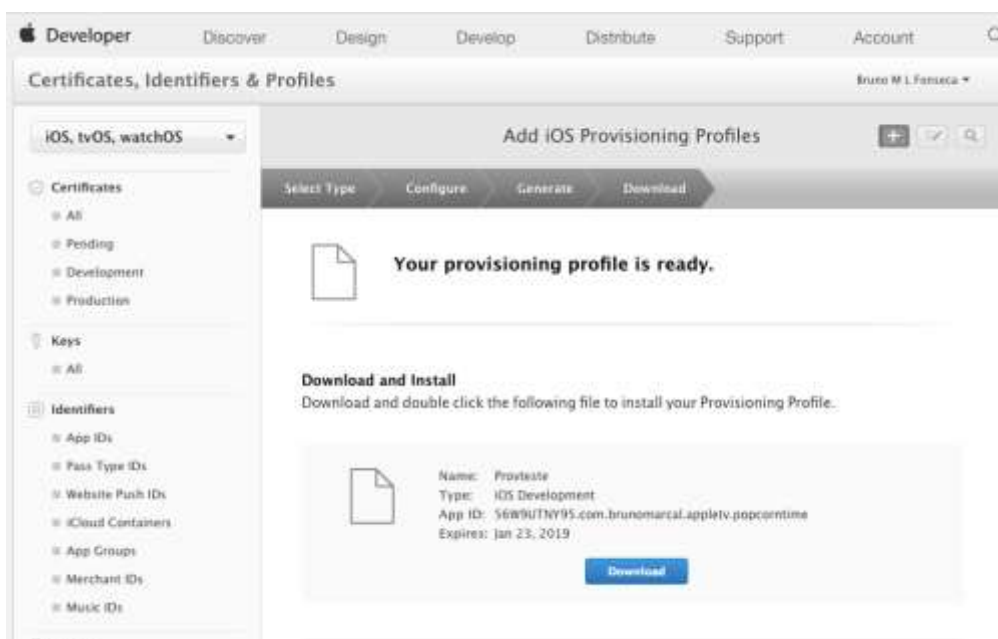
Após a tela de devices temos uma tela para confirmar algumas informações e colocar o nome do provisionamento no campo **Profile Name**, veja figura 160. Dê o nome de **Provteste** e clique em “**Continue**”.

Figura 160 – Nomeando o provisionamento



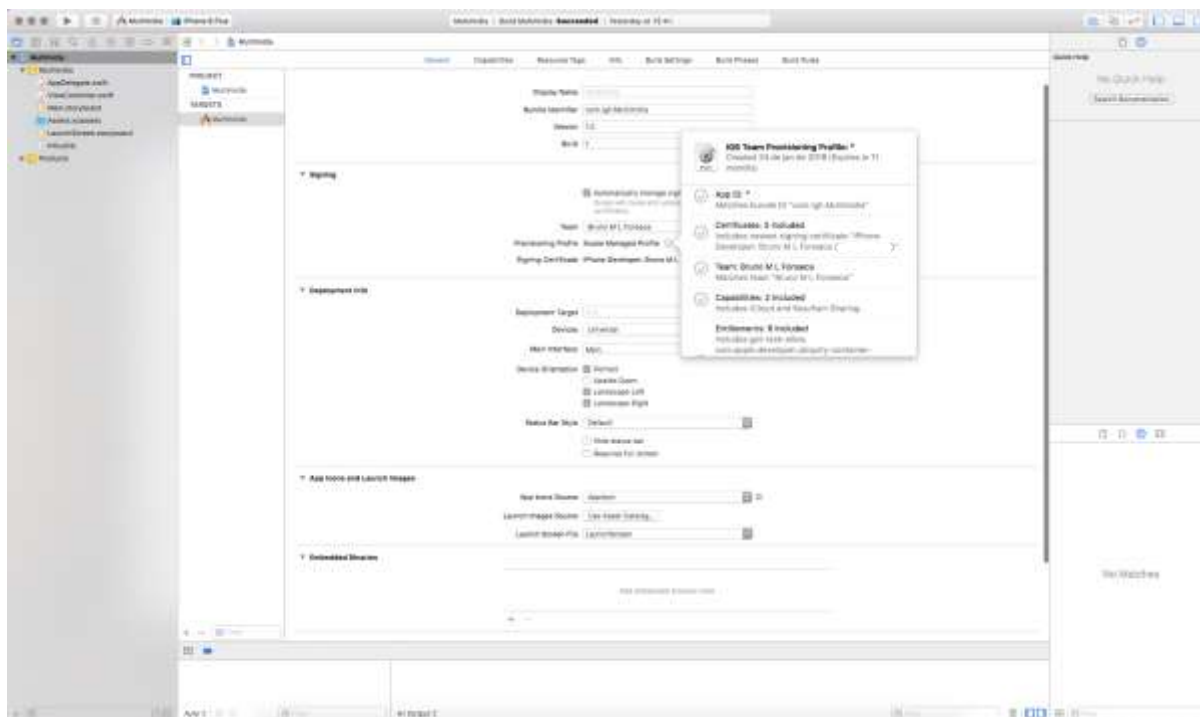
Feito isso, temos a última tela, ufa... para o download do provisionamento.

Figura 161 – Tela de download do provisionamento



Faça o download e clique duas vezes no arquivo para instalar o provisionamento, ele poderá ser visto no xCode, como mostra figura 162.

Figura 162 – Provisionamento instalado



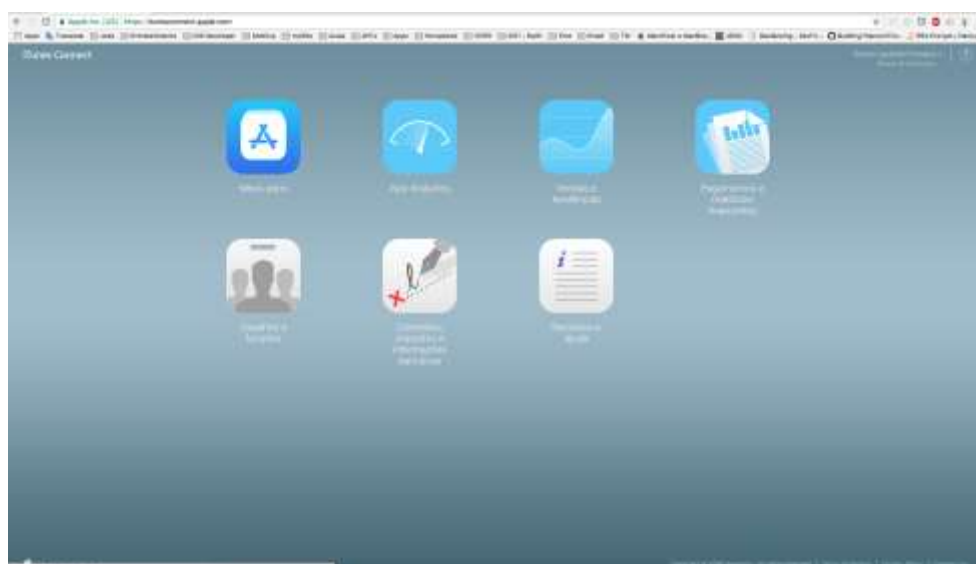
Importante: os provisionamentos podem ser criados para serem usados por vários apps, mas na hora de publicar o app na loja deverá ter seu próprio provisionamento, pois são únicos, devido ao **bundle** e serviços oferecidos via **AppID**.

iTunes Connect

No **iTunes Connect** é um site onde prepararmos os metadados do nosso aplicativo, Apple precisa disso para melhor gerir nossos apps. A figura 163 exhibe a tela inicial do **iTunes Connect**.

Importante: só conseguirá acesso a essa tela depois de fazer o cadastro como desenvolvedor da Apple e pagar o valor do tipo de conta que escolher.

Figura 163 – Tela inicial iTunes Connect



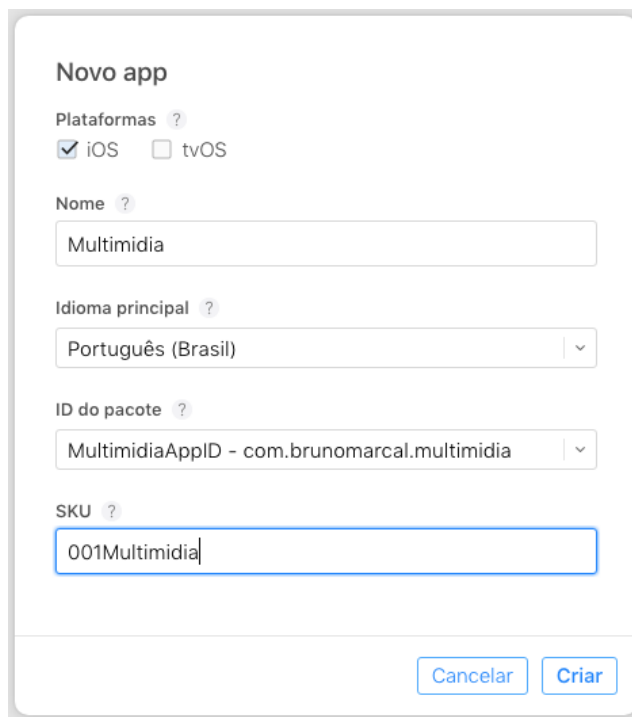
Clicando no ícone **Meus apps** veremos a tela exibida na figura 164.

Figura 164 – Criando metadados para o app



Clique em **Novo app** para abrir a tela de informações iniciais sobre o app, é exibida na figura 165.

Figura 165 – Tela inicial para criação dos metadados do app



Novo app

Plataformas ?
☒ iOS ☐ tvOS

Nome ?

Idioma principal ?

ID do pacote ?

SKU ?

[Cancelar](#) [Criar](#)

Importante: no campo ID do pacote só aparecerá o bundle da sua app se estiver tudo cadastrado corretamente, ou seja, **AppID** e **Provisinamento** criado.

Já o campo **SKU** é um ID que você mesmo pode escolher para seu app, um ID para seu próprio controle.

Após clicar no botão **“Criar”**, a tela conforme figura 166 será exibida.

Figura 166 – Tela de detalhe dos metadados do app

The screenshot shows the iTunes Connect interface for the app 'Sublinha'. The top navigation bar includes 'App Store', 'Receitas', 'Análises', and 'Métricas'. The main header displays the app name 'Sublinha' and its version '1.0.0'. Below this, the 'Informações do app' section provides details about the app, including its genre 'Livraria, referência e pesquisa' and its availability 'Disponível para download'. The 'Informações de conteúdo' section lists the app's content rating as 'Livre' and its age restriction as 'Todos os públicos'. The 'Informações de suporte' section lists the app's support email as 'suporte@sublinha.com' and its support URL as 'http://www.sublinha.com'. The 'Informações de contato' section lists the app's contact email as 'contato@sublinha.com' and its contact URL as 'http://www.sublinha.com'. The 'Informações de desenvolvimento' section lists the app's developer as 'Sublinha, Lda' and its developer website as 'http://www.sublinha.com'.

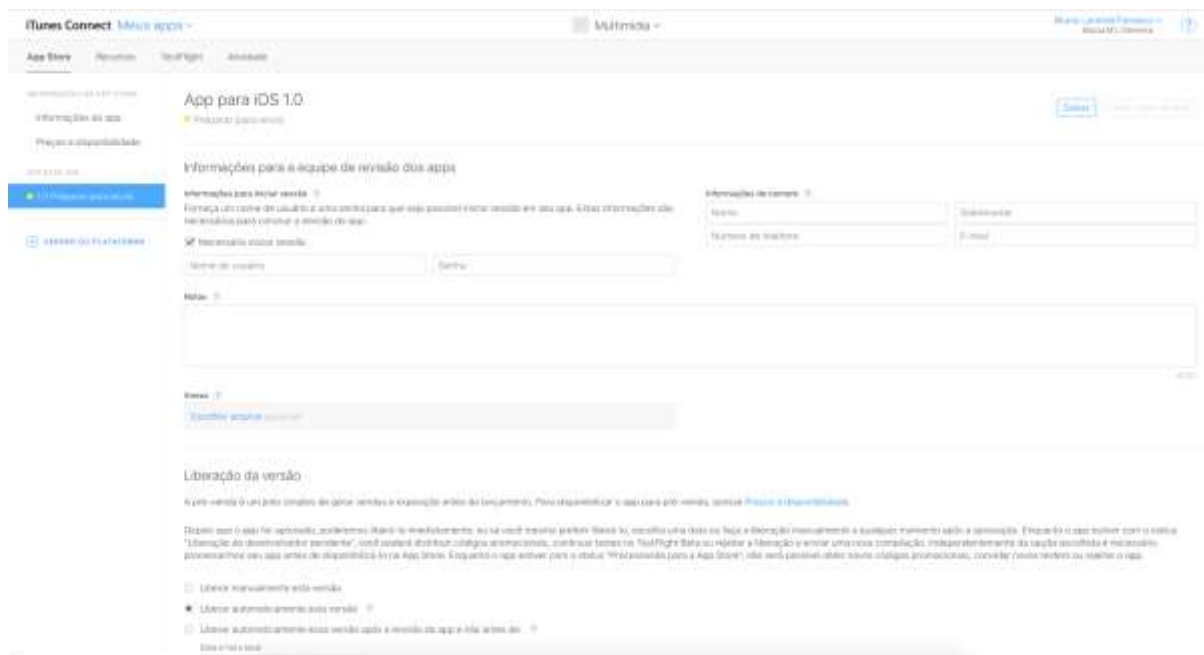
Figura 167 – Tela de screenshot do app e informações que aparecerão na App Store

[illegible]

A figura 167 exibe campos para descrição do app, screenshots de telas (para cada modelo de iPhone), palavras-chave para auxiliar na pesquisa etc.

Os campos como descrição e screenshot serão exibidos na **App Store** quando os usuários buscarem por apps.

Figura 168 – Tela que exibe informações de como testar



The screenshot shows the iTunes Connect interface for an app named 'App para iOS 1.0'. The page is titled 'Informações para a equipe de revisão dos apps'. It includes a sidebar with navigation links like 'App Store', 'Recursos', 'Suporte', and 'Atividade'. The main content area has a 'Salvar' button in the top right. Below the app name, there's a section for 'Informações para a equipe de revisão dos apps' with a text area for the app description. There are also checkboxes for 'Necessário avaliar versão' and 'Liberar automaticamente esta versão'. The bottom of the page has a 'Enviar para revisão' button.

A figura 168 exibe campos para auxiliar os testers da Apple a como testar seu app, informar usuário e senha, caso precise, e outros dados que sejam importantes no momento do teste.

Após preencher as informações pertinentes a seu app é só clicar no botão **“Salvar”** e depois em **“Enviar para revisão”** e aguardar o tempo de aprovação da Apple.

Importante: se esses dados forem necessários e não forem informados, o tester da Apple irá reprovar seu app, por isso é muito importante o preenchimento correto.

Detalhe: pode ser preenchido em inglês ou em português.

Deem uma olhada mais aprofundada nos campos, verão que são de fácil entendimento.

Existem mais campos como ícone do app, interação com **Apple Watch** etc., mas não comentaremos sobre esses outros campos.

Após criar seu app, a tela de lista de apps exibirá o app recentemente criado como mostra a figura 169.

Figura 169 – App Multimídia incluído



Importante: o iTunes Connect é bastante explicativo, publicar seu app é simples e fácil.

Referências

Apple Inc. Documentação para desenvolvedores.

LECHETA, Ricardo. *Desenvolvendo para iPhone e iPad*. 5ª ed. Novatec, 2017.

MARK, Dave; LAMARCHE, Jeff. *Dominando o desenvolvimento no iPhone*. 3ª ed. Alta Books, 2014.