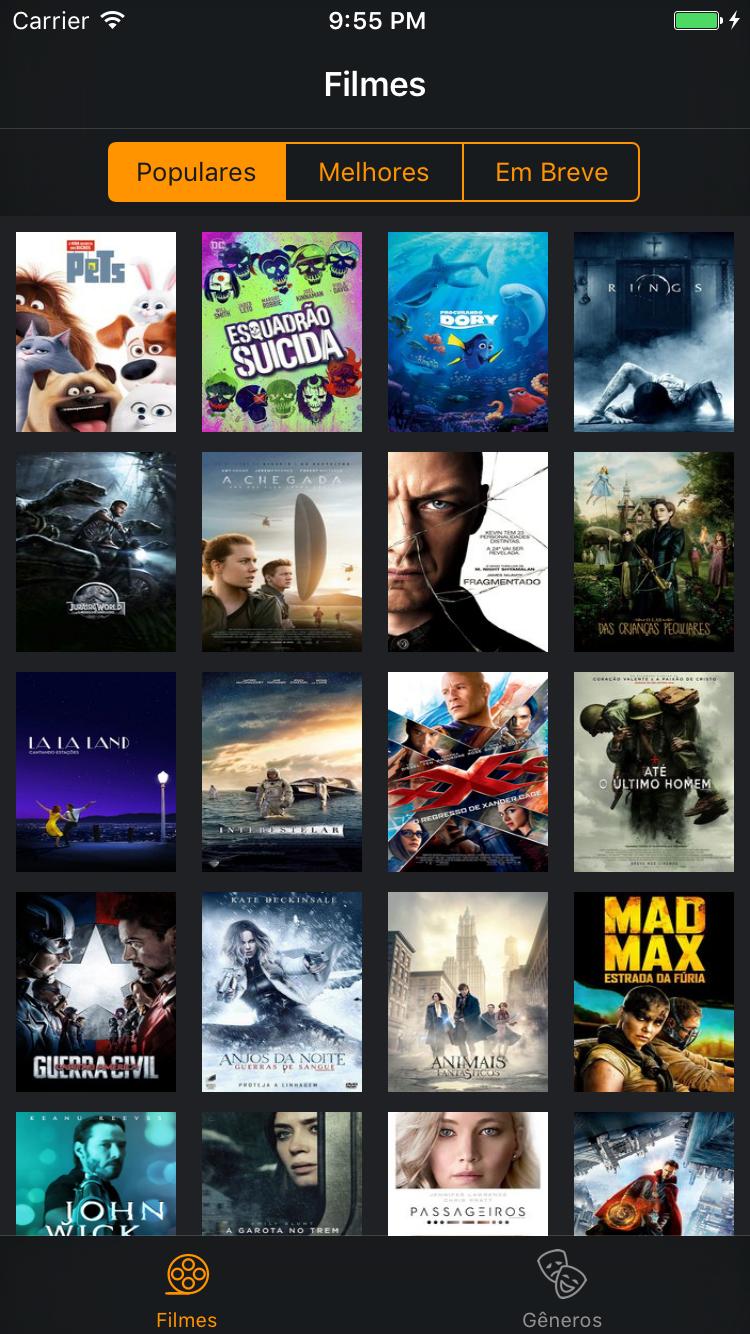
# Unidade 5

## Aula 1 - Colection Views (Filmes)

Na aula 3 da unidade 4 criamos a navegação base do aplicativo. Agora iremos criar o layout da cena de Filmes. Vamos relembrar seu visual:



Ela é composta de quatro componentes principais:

* Navigation Bar
* Barra de Filtros (Segment Control)
* Collection View
* Tab Bar

Já fizemos alguns destes componentes. Vamos riscá-los:

* Navigation Bar
* Barra de Filtros (Segmented Control)
* Collection View
* Tab Bar

Então o que precisamos fazer é a barra de filtros e a Collection View.

### Atividade

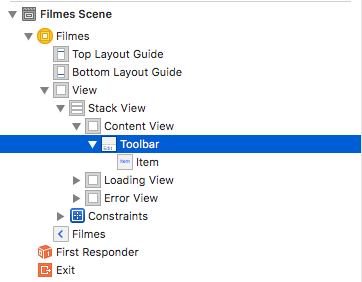
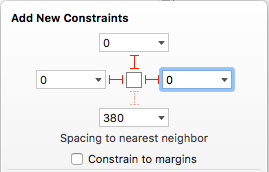
1. Crie uma barra com um Segmented Control que representará o filtro de filmes
2. Crie uma Collection View funcional. Adicione dados fictícios para testar seu funcionamento. Utilize a estrutura da View Cotroller base.

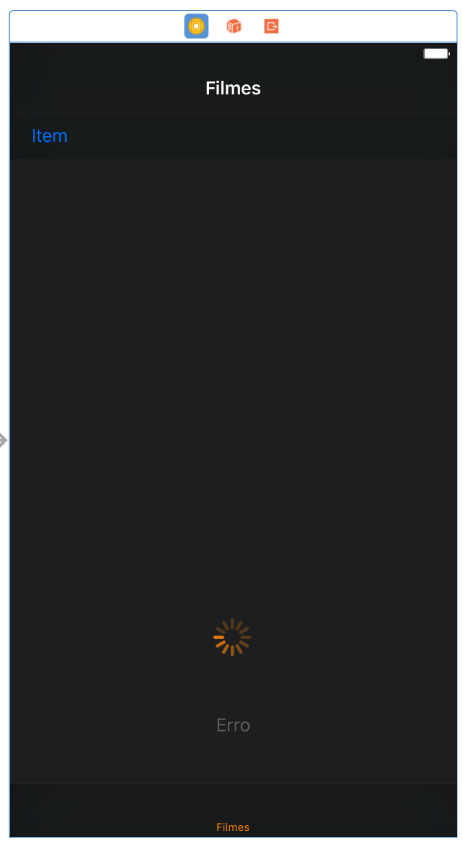
### Solução

Vamos primeiro criar a barra de filtros

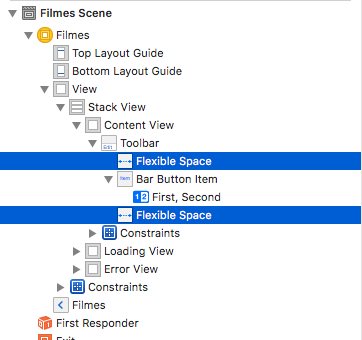
#### Barra de filtros

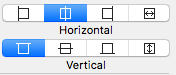
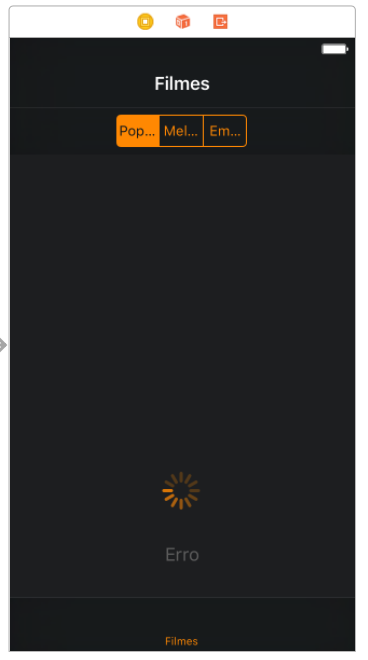
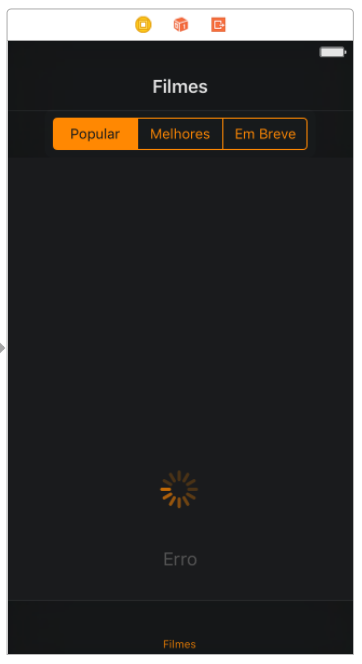
Na aula 3 da unidade 4 criamos duas cenas: a **Filmes** e a **Gêneros**. Nesta aula estamos criando o layout da cena **Filmes**, ou seja, trabalharemos no View Controller destinado a cena Filmes.

1. Na cena Filmes adicione uma **Tool Bar** como subview do que chamamos de **Content View**  
   
2. Selecione esta Tool Bar e adicione as seguintes restrições:  
     
   A Tool Bar ficará encostada no topo da Content View.



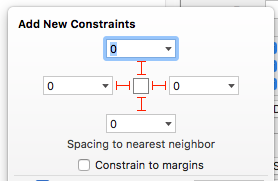
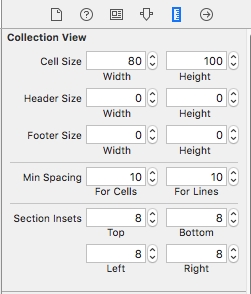
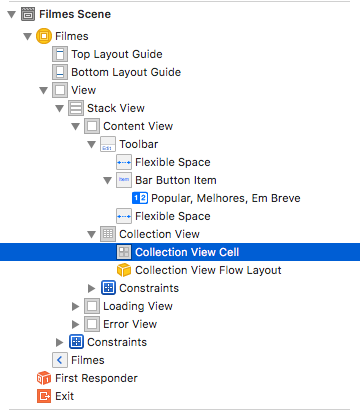
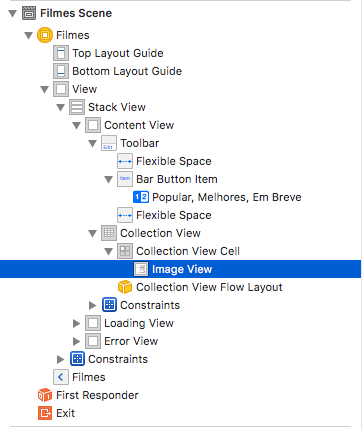
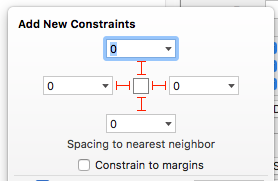
1. Remova o **Bar Button Item** da Tool Bar
2. Adicione um **Segmented Control** na Tool Bar
3. Adicione um **Flexible Space** acima e outro abaixo do **Bar Button Item** (que contém um Segmented Control). Isto fará que o Segmented Control se centralize na Tool Bar.



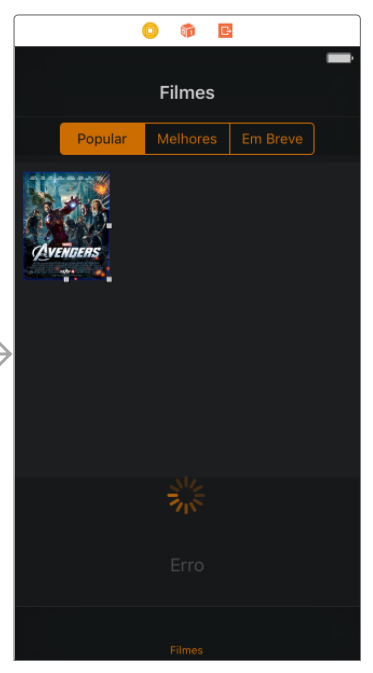
1. Agora vamos personalizar o Segmented Controller. Selecione-o e configure as seguintes propriedades:  
   **Segmented Control**
   * Segments: 3
   * Segment:
     + Segment 0
       - Title: Popular
     + Segment 1
       - Title: Melhores
     + Segment 2
       - Title: Em Breve  
         **Control**
     + Aligment:  
         
       **View**
     + Tint: #FF7F00  
       
2. Estique o Bar Button Item até que o texto contido nele fique totalmente visível  
   

#### Colection View

A criação da Collection View será dividida em duas partes: **Layout e Controller** Vamos primeiro criar seu layout:

1. Adicione uma **Collection View** como subview do que chamamos de **Content View** nesta mesta View Controller.  
   
2. Defina as seguintes restrições para a Collection View:  
     
   A collection view ficará ecostada nas bordas da Content View e com o topo encostado na Tool Bar  
   
3. Deixe a cor de background da Collection View transparente (Clear color)
4. No painel **Size Inspector** corresponda as seguintes propriedades  
   
5. Agora iremos adicionar uma imagem de exemplo na célula da colection View. Selecione a **Collection View Cell**  
   
6. Na propriedade **Identifier** escreva “MovieCollectionViewCell”
7. Adicione nela uma **Image View**  
   
8. Defina as seguintes restrições para a Image View:  
   
9. Defina a imagem **avengers-movie-poster-1** para a Image View

Veja como nosso layout ficou:



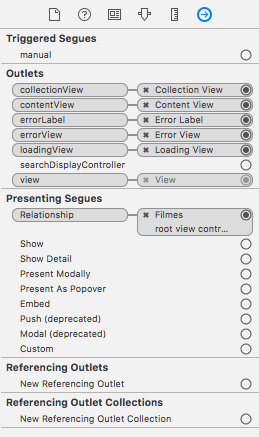
Vamos agora implementar seu View Controller básico.

1. Crie um novo arquivo chamado **MoviesViewController** que seja subclasse de UIViewController
2. Adicione os componentes básicos como fizemos na aula 1 da unidade 4.

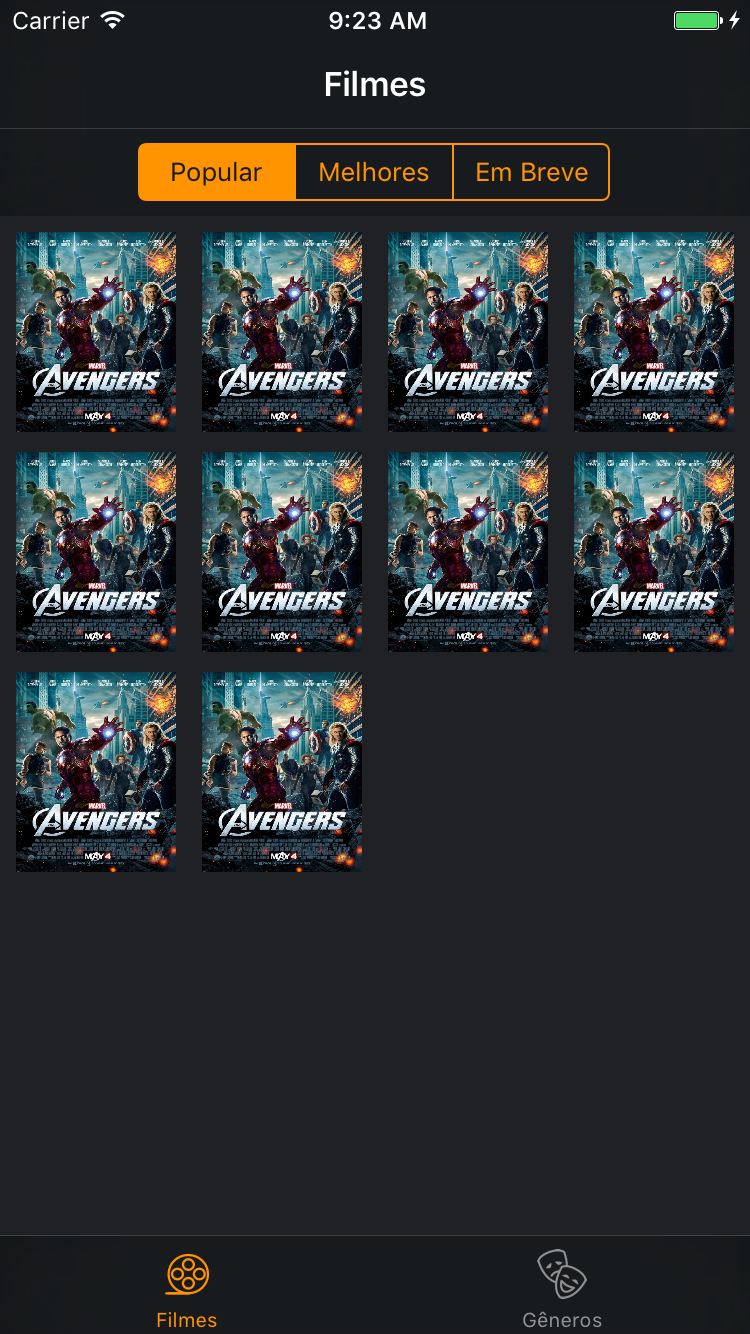
class MoviesViewController: UIViewController {  
  
 var content: Content<[Movie.ViewModel]> = .success {  
 didSet {  
 updateViews()  
 }  
 }  
  
 @IBOutlet weak var collectionView: UICollectionView!  
 @IBOutlet weak var contentView: UIView!  
 @IBOutlet weak var loadingView: UIView!  
 @IBOutlet weak var errorView: UIView!  
 @IBOutlet weak var errorLabel: UILabel!  
   
 override func viewWillAppear(\_ animated: Bool) {  
 updateViews()  
 collectionView.delegate = self  
 collectionView.dataSource = self  
 content = .success([Movie.ViewModel(id: 0, posterPath: "")])  
 }  
   
 func updateViews() {  
 switch content {  
 case .success(let list):  
 errorView.isHidden = true  
 loadingView.isHidden = true  
 contentView.isHidden = false  
 movies = list  
   
 collectionView.reloadData()  
 case .loading:  
 errorView.isHidden = true  
 loadingView.isHidden = false  
 contentView.isHidden = true  
 case .error(let error):  
 errorView.isHidden = false  
 loadingView.isHidden = true  
 contentView.isHidden = true  
   
 if case .userMessage(let message) = error {  
 errorLabel.text = message  
 }  
 }  
 view.layoutIfNeeded()  
 }  
}

1. Adicione os métodos delegates e datasource da collection View. São bastante semelhantes aos de Table Views.

extension MoviesViewController: UICollectionViewDelegate, UICollectionViewDataSource {  
   
 func collectionView(\_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {  
 let cell = collectionView.dequeueReusableCell(withReuseIdentifier: "MovieCollectionViewCell", for: indexPath)  
   
 return cell  
 }  
   
 func numberOfSections(in collectionView: UICollectionView) -> Int {  
 return 1  
 }  
   
 func collectionView(\_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {  
 return 10  
 }  
}

1. Volte na cena Filmes do Main.storyboard e vincule todos os outlets  
   

Pronto. Agora podemos testar. Veja o resultado que obtivemos:



Nas aulas seguintes iremos complementar esta tela e realizar uma listagem real de filmes, obtendo-os do backend.

## Aula 2 - Um passo a frente do MVC

Nesta aula discutiremos um pouco sobre a eficácia do MVC nos softwares da Apple. A arquitetura MVC é utilizada o tempo todo pela Apple em seus produtos e a maioria dos desenvolvedores tendem a adotar esta arquitetura por causa disso. Fazer o uso do MVC, para a Apple, permite que ela consiga explicar de maneira mais simples o uso das suas APIs, enquanto que para os desenvolvedores, usar MVC é a forma mais rápida e minimamente decente de desenvolver um aplicativo.

Temos um problema ao usar MVC em aplicativos Apple, e a comunidade apelidada de MVC como sendo Massive View Controllers, porque em uma aplicação de médio pra grande porte, nossas ViewController acabam ficando exageradamente grandes ao usarmos o Model View Controller. Isto porque existe uma linha tênue entre o que definimos como View e Controller no iOS. Facilmente podemos misturar as duas responsabilidades na classe UIViewController e suas subclasses, pois não temos um arquivo distinto que descreve Views, como no Android, apenas um componente gráfico que é basicamente um plugin para construções de View Controllers.

Acreditamos que a Apple sabe que os Massive View Controllers são um problema, então para melhorar a testabilidade de componentes MVC, a Apple mencionou **injeção de dependência** como uma boa prática na melhoria da WWDC 2016 Apps Existentes com a sessão de melhores práticas modernas . Eu interpreto isso como se dissessem: “Ei, não é bom colocar todo o código para um View Controller, injete alguns objetos de dependência que fazem certas coisas e testem interações com eles”.

### Atividade

1. Pesquise e descreva o que é injeção de dependências
2. Pesquise arquiteturas alternativas ao MVC que podem ser utilizadas com Swift. Cite suas diferenças e diga qual é a sua preferida.

### Solução

Educador, esta atividade é bastante teórica e vai exigir todo o conhecimento sobre programação orientada a objetos obtidos durante o curso todo.

Alguns esclarecimentos devem ser dados.

Esta atividade não tem o objetivo de desencorajar o uso do MVC mas sim fazer com que o aluno suba um degrau no conhecimento sobre arquiteturas de software. Conhecer MVC não foi em vão, mas é um requisito para que nós, desenvolvedores, tenhamos uma compreensão melhor de novas arquiteturas de software, pois as principais são derivadas do MVC. Separamos três arquiteturas que estão sendo bastante utilizadas no desenvolvimento iOS: MVVM, VIPER e VIP (aka Clean Swift). Antes de descrever estas arquiteturas é preciso entender o que é **injeção de dependência**.

#### Injeção de dependência

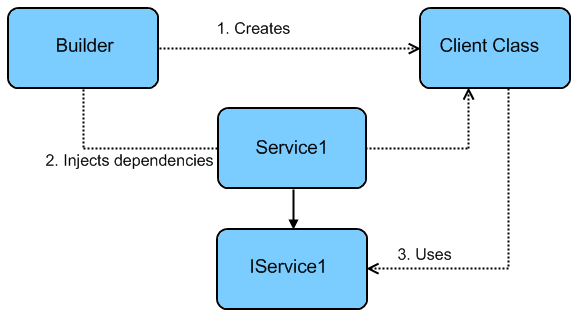
Injeção de Dependência, que é na verdade uma das formas de se fazer a inversão de controle que por sua vez, é uma técnica utilizada para diminuir o acoplamento entre classes.

Antes de tudo temos que entender que quando falamos em “injetar” uma dependência, a grosso modo, nada mais é do que passar uma classe que será utilizada para uma classe que irá consumi-la.

O padrão de injeção de dependência trabalha baseado em abstrações, sejam elas classes abstratas ou interfaces. Se pudéssemos citar um “lema”, este seria: programe para uma interface e nunca para uma implementação. E este “lema” realmente faz diferença quando queremos diminuir o acoplamento entre as classes do nosso modelo. Podemos trabalhar com a injeção de dependência de três formas: injeção por construtor (constructor injection); injeção por propriedade ou getters e setters no caso do Java ( setter injection ); e injeção por interface ( interface injection ).

O padrão de injeção de dependência é baseado em outro padrão chamado Builder. O Builder será responsável por construir para nós os objetos e armazená-los em algum lugar. É aí que entra em ação o container, que é justamente o lugar onde os objetos construídos serão armazenados. A definição mais simples para o container é que ele é um objeto que pode armazenar outros objetos dentro dele.

Para ficar mais claro e até mais fácil de entender como funciona a injeção de dependência, observe o diagrama abaixo:



O que o diagrama acima mostra é o seguinte: está sendo passada a responsabilidade de criar um objeto para o builder, indicando a ele qual é a interface que nós desejamos. Ele cria o objeto que implementa a interface e injeta esta dependência na classe Cliente.

E aí está a inversão de controle: ao invés da classe cliente saber qual classe concreta ela precisa utilizar, ela delega isto para outra classe que, além de retornar a implementação apropriada para aquela interface, ainda injeta a dependência. Com isto, quebramos o forte acoplamento e invertemos a dependência, e ao invés de depender de uma classe concreta, passamos a depender de uma abstração, passando a ter um cenário desacoplado, de fácil manutenção e evolução.

#### MVVM

A arquitetura MVVM (Model - View - View Model) assume a existência de um View Model, sendo uma camada adicional com o qual uma View Controller se comunica. Nela esta está contida a lógica de negocio para lidar com a View Controller e também pode conter objetos como por exemplo Serviços de Networking e armazenamento de dados. O View Model expõe um Model como dados formatados em suas propriedades, então a View Controller observa mudanças nessas propriedades e atualiza a sua exibição de acordo com isto.

Como nenhuma arquitetura é perfeita, o problema desta é que transferimos as mil linhas de código do View Controller para o View Model. Mas qual é a vantagem então?

A primeira é que a View Controller fica responsável apenas por exibir o conteúdo formatado pelo View Model e receber eventos do usuário e mandar para o View Model tratar. A segunda vantagem é que ela funciona muito bem com extensões reativas, que permitem observar mudanças nas propriedades do View Model e agir sobre esta mudanças de forma não imperativa. Você pode vincular a propriedade do View Model a uma view diretamente ou definir um bloco de código que é executado quando o valor da propriedade é alterado e vinculá-lo a uma View.

#### VIPER

O VIPER, que traduzindo significa Víbora mas na verdade é sigla para View - Interactor - Presenter - Entity - Router, é mais uma maneira de configurar a arquitetura do aplicativo. O VIPER divide a responsabilidade do View Controller do MVC em diferentes objetos. Um **View Controller** define os dados do **Presenter** a uma **View**, que por sua vez exibe o dado. O **Presenter** tem a simples tarefa de formatar os dados oriundos do **Interactor**, deixando prontos para serem exibidos, geralmente no formato String. O **Presenter** também pede ao **Router** que execute a navegação para um módulo diferente do app. O **Interactor** contém a lógica de negócios da aplicação na manipulação dos dados, executa solicitações de rede e salva **Entities** (modelos) para um armazenamento persistente através de um proxy. Os componentes View, Interactor, Presenter, Entity e Router de um View Controller, juntamente com ele, são chamados de módulo.

Quando comparado ao MVVM, a lógica de negócios e a apresentação dos dados foram divididos do View Model para o Interactor e o Presenter. Há também um componente adicional chamado Router (também conhecido como Wireframe), que lida com a criação de outros View Controllers, ligando suas dependências e definindo delegates que permitem a comunicação com outras cenas.

#### VIP (aka Clean Swift)

Ele recebe o nome View (Controlador) - Interactor - Presenter (VIP) de ciclo uni-derecional . VIP diverge de VIPER, mas ainda assim divide bem responsabilidades e propaga dados entre os componentes. O **View Controller** interage diretamente com um **Interactor** enviando **Request** para ele. O Interactor responde a essas solicitações enviando uma **Response** com modelo de dados para um **Presenter**. O **Presenter** formata os dados a serem exibidos, cria um **View Model** e notifica o **View Controller** de que deve atualizar sua **View** com base no **View Model**. O **View Controller** decide quando a navegação para uma cena diferente deverá acontecer chamando um método em um **Router**. O **Router** executa a configuração do próximo **View Controller** e liga os componentes, passando dados e configuração de delegates. Componentes VIP pertencentes a um único View Controller formam uma cena.

#### Comparação

É hora de uma comparação entre o MVC, MVVM, VIPER e VIP. Para algumas das métricas, MVC é uma linha de base. Tomaremos em consideração os seguintes aspectos:

**Responsabilidades** - descreve se há um objeto que desempenha todos os papéis ou se há muitos objetos com responsabilidades diferentes  
**View Controller** - descreve a função de um View Controller  
**Fluxo de dados** - descreve como os dados fluem na cena  
**Testabilidade** - avalia o quão fácil é testar componentes  
**Entrada** - avalia como é fácil para uma pessoa mergulhar em um projeto iniciado ou começar a desenvolver um app em uma determinada arquitetura  
**Colaboração** - avalia como é fácil colaborar (desenvolver em equipe) em uma cena escrita em uma certa arquitetura  
**Reutilização** - avalia se os componentes da cena podem ser reutilizados em uma cena diferente  
**Refatoração** - avalia quantos componentes são afetados durante o processo de refatoração  
**Número de arquivos** - avalia o número de arquivos no projeto  
**Linhas de código em um único arquivo** - avalia o número de linhas no arquivo único

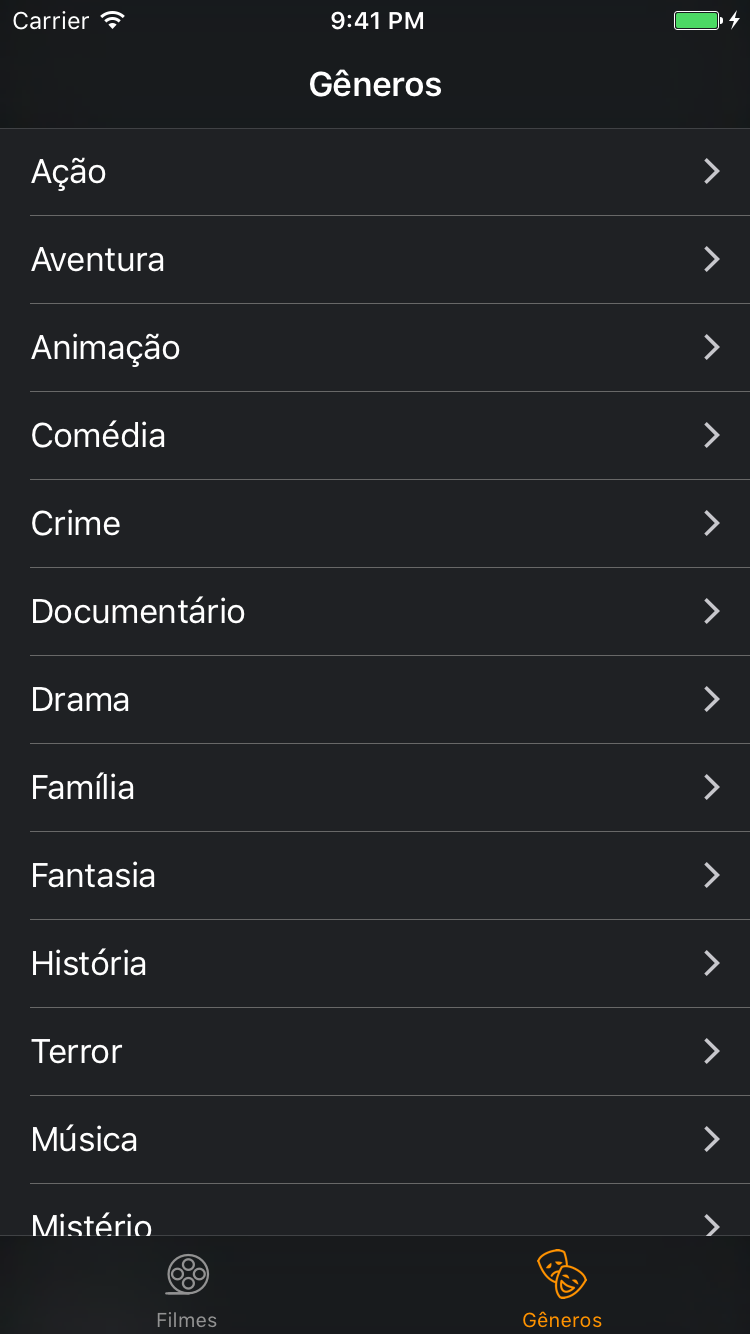
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | MVC | MVVM | VIPER | VIP |
| **Responsabilidades** | Enredado | Disperso | Disperso | Disperso |
| **View Controller** | Faz tudo | Passa ações para e vincula dados do View Model | Passa as ações e vincula os dados do Presenter | Passa ações e exibe dados do Presenter, decide quando navegar para a próxima cena |
| **Fluxo de dados** | Multidirecional | Multidirecional | Multidirecional | Unidirecional |
| **Testabilidade** | Difícil (muitas responsabilidades) | Melhor | Melhor | Melhor |
| **Entrada** | Fácil | Difícil | Mais Difícil | Difícil, mas boa documentação existente |
| **Colaboração** | Difícil | Melhor | Melhor | Melhor |
| **Reutilização** | Nada | Bastante pequena | Ok | Ok |
| **Refatoração** | Normal | Normal (afeta os limites entre View Controller e View Model) | Pior (pode afetar muitos extremos devido ao fluxo multidirecional) | Normal (afeta um limite devido ao fluxo multidirecional) |
| **Número de arquivos** | Normal | View Model adicional para cada View Controller | Muitos | Muitos |
| **Linhas de código em um único arquivo** | Muitos | Muitos | Quantidade satisfatória | Quantidade satisfatória |

#### O que iremos utilizar?

Você deve ter percebido que o MVC é a arquitetura mais rápida de ser desenvolvida. Iremos implementar aqui um pseudo VIP. Este não terá a camada Presenter pois os dados vindos dos nossos serviços web estão quase totalmente polidos (prontos para serem exibidos) e também não queremos estender muito esta aula. Na unidade 6 começaremos a implementar nossa arquitetura.

## Aula 3 - Table Views (Gêneros)

Agora que aprendemos como criar uma tabela, vamos criar a tela de Gêneros. Relembre como ela deverá ser:



### Atividade

Crie a tela de Gêneros. Você precisará criar ela em uma View Controller e utilizar Genre.ViewModel para preencher os dados da lista.

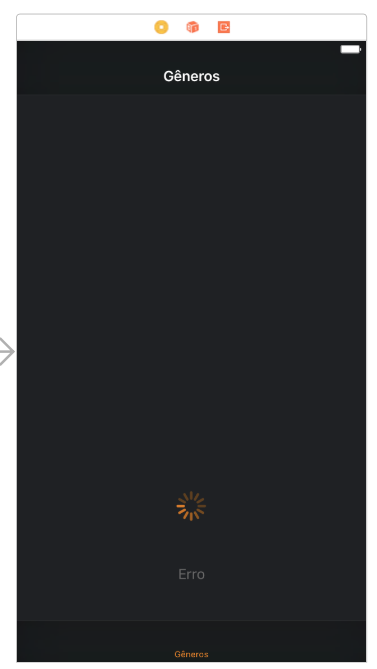
### Solução

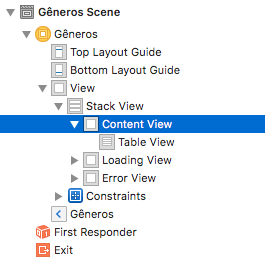
A criação de Views com tabela é simples e podemos dividir em três estapas: Composição da View, Criação do View Controller básico e implementação dos delegates e datasources.

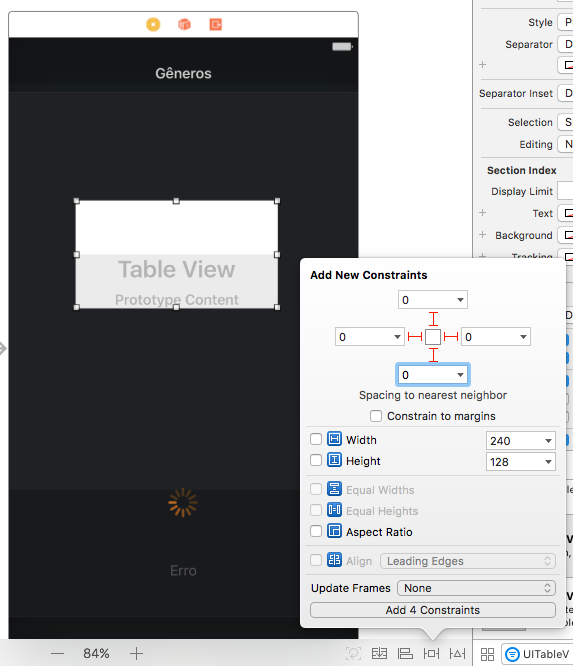
#### Composição da View

Iremos implementar a cena Gêneros na View Controller criada na aula 1 desta unidade e que já está vinculada à TabViewController. Então siga os seguintes passos.

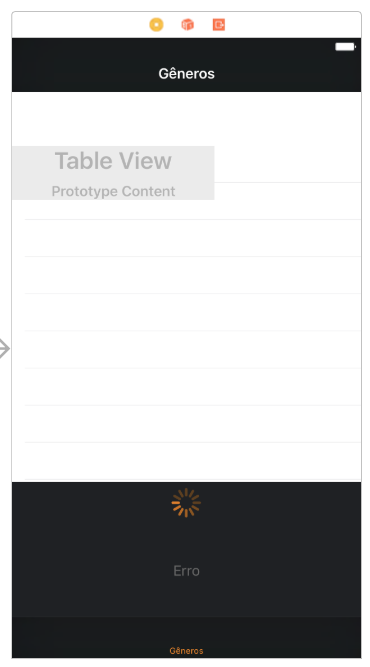
1. Abra o **Main.storyboard** e selecione a cena **Gêneros**, que até este ponto estará assim:



1. Em seu painel de navegação selecione a view que chamaos de **Content View** e insira uma UITableView nela:  
   
2. Agora selecione a Table View e adicione as seguintes restrições (auto-layout):



1. Atualize os frames e a Table View ficará assim:

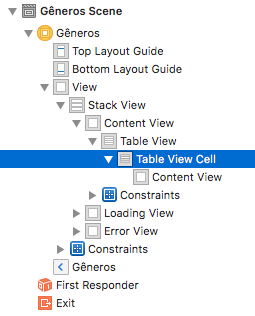


**DICA**: Para a execução desta atividade utilizamos o Xcode na versão 8.2. Talvez ele apresente um erro na exibição de Views no Storyboard como o título “TableView” e subtítulo “Prototype Content” mostrado na imagem acima que estão desalinhados.

1. Agora vamos comfigurar a aparência da TableView. Selecione a Table View e no painel de propriedades configure as seguintes propriedades:

* Background: **Clear Color**
* Separator Color: **#555555**
* Content: **Dynamic Prototypes**
* Prototype Cells: **1**
* Scroll View Style: **Black**

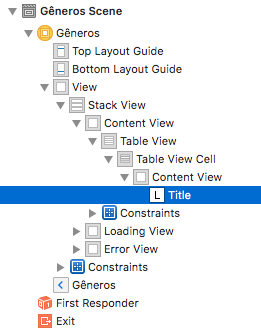
1. Agora precisaremos configurar a célula que adicionamos ao definir “Prototype Cells: **1**”. Selecione a célula criada:



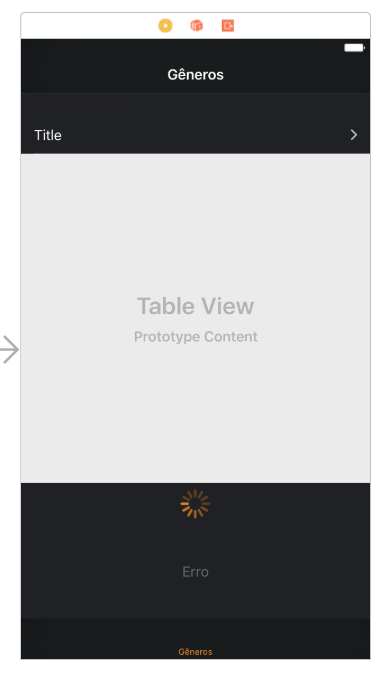
1. Vamos agora configurar a célula. Abra o painel de propriedades e modifique:

* Style: **Basic**
* Identifier: **GenreTableViewCell**
* Accessory: **Disclosure indicator**
* Background: **Clear Color**

1. Selecione Title da célula:



1. Mude o atributo **Color** para White. A tela ficará assim:



#### Criando o ViewController básico

Baseando-se na aula 1 desta unidade, vamos criar um View Controller com os atributos básicos para um bom controle de estado da View então siga os seguintes passos:

1. Crie uma nova classe com o nome **GenresViewController** que estenda a classe UIViewController.
2. Adicione o seguinte código base para nossa ViewController:

class GenresViewController: UIViewController {  
   
 var content: Content<[Genre.ViewModel]> = .loading {  
 didSet {  
 updateViews()  
 }  
 }  
   
 var genreList: [Genre.ViewModel] = []  
   
 @IBOutlet weak var tableView: UITableView!  
 @IBOutlet weak var loadingView: UIView!  
 @IBOutlet weak var contentView: UIView!  
 @IBOutlet weak var errorView: UIView!  
 @IBOutlet weak var errorLabel: UILabel!  
   
 override func viewWillAppear(\_ animated: Bool) {  
 updateViews()  
 }  
   
 func updateViews() {  
 switch content {  
 case .success(let list):  
 errorView.isHidden = true  
 loadingView.isHidden = true  
 contentView.isHidden = false  
 genreList = list  
 tableView.reloadData()  
 case .loading:  
 errorView.isHidden = true  
 loadingView.isHidden = false  
 contentView.isHidden = true  
 case .error(let error):  
 errorView.isHidden = false  
 loadingView.isHidden = true  
 contentView.isHidden = true  
  
 if case .userMessage(let message) = error {  
 errorLabel.text = message  
 }  
 }  
 }  
}

No código acima não temos muitas novidades. Atente-se para o seguinte trecho:

var content: Content<[Genre.ViewModel]> = .loading {  
 didSet {  
 updateViews()  
 }  
 }  
   
 var genreList: [Genre.ViewModel] = []

Aqui temos a propriedade content que será utilizada para controlarmos os estados da nossa View. A propriedade genreList irá armazenar o resultado que vier no case success da propriedade content, como você pode ver em updateViews().

#### Implementando os delegates e data sources.

Para implementar os delegates precisamos de três métodos básicos:

* numberOfSections(in:): que utilizamos para especificar o numero de sessões da nossa tabela.
* tableView(numberOfRowsInSection:): utilizamos para especificar o número de linhas em cada sessão.
* tableView(cellForRowAt:): que utilizamos para preencher os dados que cada célula apresentará.

Sendo assim, insira a seguinte extension na class **GenresViewController**:

extension GenresViewController: UITableViewDelegate, UITableViewDataSource {  
   
 func numberOfSections(in tableView: UITableView) -> Int {  
 return 1  
 }  
   
 func tableView(\_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {  
 return genreList.count  
 }  
   
 func tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {  
 let cell = tableView.dequeueReusableCell(withIdentifier: "GenreTableViewCell")!  
 cell.textLabel?.text = genresList[indexPath.row].name  
 return cell  
 }  
}

Ainda não podemos executar o código pois nada acontecerá e nada será exibido. As unidades seguintes serão o fechamento das funcionalidades para cada cena.