# Unidade 3

## Aula 1 - Convertendo dados para ViewModel e ResponseModel

Ainda não terminamos de implementar a nossa camada de model. Se você se lembrar, no curso de Android utilizamos a biblioteca Gson para realizar o parse (conversão) de objetos JSON para os nossos Models. Aqui, no curso de iOS, iremos fazer o parsing manualmente para que você pratique a linguagem.

Nas aulas futuras iremos introduzir a camada de serviço para realizarmos as chamadas REST, então considere que um objeto JSON será do tipo [String: Any] ou seja, um dicionário que aceita chaves do tipo String e valores de qualquer tipo. Para facilitar use o seguinte typealias para apelidar este tipo:

typealias JSONObject = [String:Any]

### Atividade

Criamos os ResponseModels e já vimos nesta aula como definir **initializers** para nossos objetos. Para cada ResponseModel crie um inicializador que receba via parâmetro um JSONObject. Dentro destes inicializadores, mapeie o objeto JSON nas propriedades dos models. Exemplo:

struct Genre {  
 struct ResponseModel {  
 let name: String?  
 let id: Int?  
   
 init(with json: JSONObject?) {  
 name = json?["name"] as? String  
 id = json?["id"] as? Int  
 }  
 }  
 //...  
}

Faça isto com todos os ResponseModels.

### Solução

Educador, a solução para esta aula é simples. O parse do Response Model de Gênero já está criado no exemplo. Vamos então criar um parse para os demais objetos.

#### Parsing de Movie.ResponseModel

A struct **Movie.ResponseModel** é tão simples quanto a de Gênero, porém possui mais propriedades. Então ela ficará assim:

struct Movie {  
 struct ResponseModel {  
 let page: Int?  
 let results: [Result]?  
 let total\_results: Int?  
 let total\_pages: Int?  
   
 struct Result {  
 let poster\_path: String?  
 let adult: Bool?  
 let overview: String?  
 let release\_date: String?  
 let genre\_ids: [Int]?  
 let id: Int  
 let original\_title: String?  
 let original\_language: String?  
 let title: String?  
 let backdrop\_path: String?  
 let popularity: Double?  
 let vote\_count: Int?  
 let video: Bool?  
 let vote\_average: Double?  
   
 init(with json: JSONObject?) {  
 poster\_path = json?["poster\_path"] as? String  
 adult = json?["adult"] as? Bool  
 overview = json?["overview"] as? String  
 release\_date = json?["release\_date"] as? String  
 genre\_ids = json?["genre\_ids"] as? [Int]  
 id = (json?["id"] as? Int) ?? -1  
 original\_title = json?["original\_title"] as? String  
 original\_language = json?["original\_language"] as? String  
 title = json?["title"] as? String  
 backdrop\_path = json?["backdrop\_path"] as? String  
 popularity = json?["popularity"] as? Double  
 vote\_count = json?["vote\_count"] as? Int  
 video = json?["video"] as? Bool  
 vote\_average = json?["vote\_average"] as? Double  
 }  
 }  
   
 init(with json: JSONObject?) {  
 page = json?["page"] as? Int  
 total\_results = json?["total\_results"] as? Int  
 total\_pages = json?["total\_pages"] as? Int  
   
 results = (json?["results"] as? [JSONObject])?.map {  
 Result(with: $0)  
 }  
 }   
 }  
   
 // ...  
}

Repare no mapeamento para os dados de Result:

results = (json?["results"] as? [JSONObject])?.map {  
 Result(with: $0)  
 }

O map é um método de mapeamento de arrays. O que ele faz é iterar o array inteiro e executar o bloco de código (limitado por “{ }”) para cada item do array. O elemento $0 é uma representação do item do array. Na unidade 5 aprenderemos melhor o conceito de **closure** e veremos alguns exemplos deste tipo. Se estiver muito confuso, você poderá trocar este trecho pelo seguinte código equivalente:

var array: [Result] = []  
for item in (json?["results"] as? [JSONObject]) {  
 array.append(Result(with: item))  
}  
results = array

#### Parsing de MovieDetails.ResponseModel

Este Parsing segue a mesma estrutura do anterior. Veja como fica:

struct ResponseModel {  
 typealias GenreModel = Genre.ResponseModel  
   
 let backdrop\_path: String?  
 let budget: Int?  
 let genres: [GenreModel]?  
 let id: Int  
 let original\_title: String?  
 let overview: String?  
 let vote\_average: Double?  
 let revenue: Int?  
 let title: String?  
 let release\_date: Date?  
 let runtime: Int?  
  
   
 init(with json: JSONObject?) {  
 backdrop\_path = json?["backdrop\_path"] as? String  
 budget = json?["budget"] as? Int  
 id = (json?["id"] as! Int)   
 original\_title = json?["original\_title"] as? String  
 overview = json?["overview"] as? String  
 vote\_average = json?["vote\_average"] as? Double  
 revenue = json?["revenue"] as? Int  
 title = json?["title"] as? String  
   
 runtime = json?["runtime"] as? Int  
   
 let formatter = DateFormatter()  
 formatter.dateFormat = "yyyy-MM-dd"  
 let dateString = json?["release\_date"] as? String  
   
 release\_date = formatter.date(from: dateString ?? "")  
   
 genres = (json?["genres"] as? [JSONObject])?.map {  
 GenreModel(with: $0)  
 }  
 }  
 }  
 //...  
}

O único ponto a se destacar é a formatação de String para Date:

let formatter = DateFormatter()  
 formatter.dateFormat = "yyyy-MM-dd"  
 let dateString = json?["release\_date"] as? String  
   
 release\_date = formatter.date(from: dateString ?? "")

Como você deve ter visto na referência fornecida para este Response Model, o campo release\_date representa a data de lançamento do filme e é do tipo String. Uma boa forma de armazenar este tipo de dado é diretamente em um objeto Date. Para isto você deve identificar o DateFormat da data que virá em string do serviço web (identificamos que virá no formato "yyyy-MM-dd") e utilizar o procedimento do código acima. Pesquise sobre **Date Format** para saber mais sobre os formatos de data padrões.

#### Convertendo ResponseModel em ViewModel

Analizando um pouco você irá perceber que no ViewModel de MovieDetails ficará um pouco longo. Na hora de converter os dados parseados em ResponseModel para ViewModel nos depararemos com formatações do tipo:

* Monetária: para as propriedades budget e revenue.
* Date: Para apresentar apenas o ano de release\_date na propriedade releaseDate
* Estrelas: Formataremos o valor de vote\_average, que é um número de 0 a 10 e representa a nota do filme, para mostrar apenas os valores 1, 2, 3, 4 e 5, que representarão a quantidade de estrelas que exibiremos para o filme.

Tendo isto em vista sugerimos que você, educador, crie um **initializer** para este ViewModel, parecido com os que criamos para nossos ResponseModels, mas ao invés de receber como parâmetro um JSONObject, faça com que ele receba um MovieDetails.ResponseModel.

O arquivo **MovieDetailsModels.swift** ficará assim:

import Foundation  
  
  
struct MovieDetails {  
   
 struct ResponseModel {  
 typealias GenreModel = Genre.ResponseModel  
   
 let backdrop\_path: String?  
 let budget: Int?  
 let genres: [GenreModel]?  
 let id: Int  
 let original\_title: String?  
 let overview: String?  
 let vote\_average: Double?  
 let revenue: Int?  
 let title: String?  
 let release\_date: Date?  
 let runtime: Int?  
  
   
 init(with json: JSONObject?) {  
 backdrop\_path = json?["backdrop\_path"] as? String  
 budget = json?["budget"] as? Int  
 id = (json?["id"] as! Int)   
 original\_title = json?["original\_title"] as? String  
 overview = json?["overview"] as? String  
 vote\_average = json?["vote\_average"] as? Double  
 revenue = json?["revenue"] as? Int  
 title = json?["title"] as? String  
   
 runtime = json?["runtime"] as? Int  
   
 let formatter = DateFormatter()  
 formatter.dateFormat = "yyyy-MM-dd"  
 let dateString = json?["release\_date"] as? String  
   
 release\_date = formatter.date(from: dateString ?? "")  
   
 genres = (json?["genres"] as? [JSONObject])?.map {  
 GenreModel(with: $0)  
 }  
 }  
 }  
   
 struct ViewModel {  
 let title: String  
 let originalTitle: String  
 let backDropPath: String  
 let voteAverage: Int  
 let overview: String  
 let revenue: String  
 let budget: String  
 let releaseDate: String  
 let genre: String  
 let runtime: String  
   
   
 init(with responseModel: ResponseModel) {  
 title = responseModel.title ?? ""  
 originalTitle = responseModel.original\_title ?? ""  
 backDropPath = responseModel.backdrop\_path ?? ""  
 voteAverage = Int((responseModel.vote\_average ?? 0.0)/2)  
 overview = responseModel.overview ?? ""  
   
 let formatter = NumberFormatter()  
 formatter.locale = Locale.current  
 formatter.numberStyle = .currency  
   
 if let formattedTipAmount = formatter.string(from: responseModel.revenue as NSNumber? ?? 0 as NSNumber) {  
 revenue = formattedTipAmount  
 } else {  
 revenue = ""  
 }  
   
 if let formattedTipAmount = formatter.string(from: responseModel.budget as NSNumber? ?? 0 as NSNumber) {  
 budget = formattedTipAmount  
 } else {  
 budget = ""  
 }  
   
 if let date = responseModel.release\_date {  
 let formatter = DateFormatter()  
 formatter.dateFormat = "yyyy"  
 releaseDate = formatter.string(from: date)  
 } else {  
 releaseDate = ""  
 }  
   
 genre = responseModel.genres?.first?.name ?? ""  
 runtime = "\((responseModel.runtime ?? 0)) min"  
 }  
 }   
}

Mostre aos seus alunos esta opção.

## Aula 2 - Criando layouts (Detalhes do Filme)

Vamos para a nossa primeira atividade sobre layouts. Em termos de layout, esta cena é a mais complexa. Ela apresenta diversos elementos e diversas restrições, então vamos passo-a-passo. Primeiramente vamos relembrar a aparência que ela deverá ter:



### Atividade

1. Crie o layout da cena Detalhes do filme. Ela deverá apresentar os seguintes dados:

* Cartaz do filme
* Nome brasileiro do filme
* Ano de lançamento
* Gênero
* Duração do Filme
* Sinopse
* Nome original do filme
* Investimento no filme
* Receita obtida pelo filme

Você deverá apresentar estas informações em uma Scroll View, de modo que se o conteúdo da tela extrapolar o tamanho da tela o usuário possa rolar a página para ver o restante do conteúdo.

### Solução

Para criar esta tela siga os seguintes passos.

1. Adicione uma UIViewController no **Main.storyboard**:
2. Dentro desta View Controller adicione uma ScrollView
3. Selecione a Scroll View e adicione as seguintes restrições nela
4. Adicione também uma UIView dentro da Scroll View
5. Selecione esta UIView e nomeie-a para **Content View**
6. E também adicione as seguintes restrições:
7. Para termos uma tela rolável ainda precisamos adicionar as seguintes restrições;  
   a. Selecione a **Content View** e com a tecla **Control** precionada clique sobre a **Content View**, arraste e solte em cima da **View** (que é a View raiz da View Controller)
8. Aparecerá as seguintes poções:
9. Selecione **Equal Width** e **Equal Heigth**.
10. Nunca se esqueça de atualizar os frames após adicionar restrições:
11. Selecione a **Content View** e selecione **Clear Color** na propriedade **Background Color**.
12. Insira uma **UIImageView** como subview da **ContentView**
13. Adicione as seguintes restrições a Image View
14. Na propriedade **Image** da Image View, escreva **backdrop\_sample**
15. Nos teremos o seguinte resultado por enquanto.

Ufa. O trabalho é bastante repetitivo e possui diversas maneiras para o mesmo objetivo ser alcançado. Continue adicionando as labels que faltam e as estrelas até que obtenha um resultado próximo ao proposto.

Uma dica importante é:

Para que nossas labels, imagens e botões tenham altura dinâmica, de acordo com seu texto/conteúdo basta não adicionarmos restrições de altura.

Se o tamanho da **Content View** for muito pequeno para o conteúdo, basta aumentar a altura da View Controller.

Você pode consultar o projeto no material de apoio. Basta encontrar a cena **Detalhes do Filme** no **Main.storyboard**

## Aula 3 - Manipulando erros

No aplicativo que desenvolvemos como TDP do Android focamos apenas nas funcionalidades e não demos muito atenção ao tratamento de erros. Como estamos lidando com um serviço web de terceiro, podemos nos deparar com erros comuns como **serviço indisponível**, **servidor em manutenção** ou seu celular pode estar com a internet desligada sem que você tenha percebido.

Para lidar com estes problemas conhecidos, um bom app deve conter uma estrutura mínima de tratamento de erros e providenciar as seguintes coisas ao usuário:

* Feedback: o usuário deve saber o que está acontecendo.
* Solução: se possível, devemos apresentar ao usuário uma solução para contornar o erro.
* Consistência: não adianta o usuário ser informado que ocorreu um erro se ele não souber interpretá-lo, então seja claro.

### Atividade

Identifique quais são os possíveis erros que virão do serviço web quando este já estiver implementado (implementaremos na unidade 6). Apresente uma maneira de encapsulá-los para que possamos manipulá-los facilmente.

Veja algumas referências que poderão te ajudar:

* [Error Handling - Documentação Swift](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/ErrorHandling.html)
* [Magical Error Handling in Swift](https://www.raywenderlich.com/130197/magical-error-handling-swift)

### Solução

Educador, esta é uma solução abstrata e os alunos podem apresentar diversas variações. Existem várias abordagens a serem tomadas. Aqui seguiremos uma bem simples.

Analisando os possíveis erros, identificamos:

* Internet desconectada
* Serviço não disponível (o servidor por algum motivo não está respondendo ou está fora do ar)
* Serviço não encontrado (estamos tentando acessar algo que não existe)

O erro de internet desconectada é obtido antes mesmo de realizarmos a conexão. Este erro pode ser identificado pelo código **-1009** e manipulado diretamente na camada mais baixa.

O erro de serviço não encontrado é na verdade um erro de desenvolvimento, ou seja, um bug causado pelo desenvolvedor e isto não pode chegar na mão do usuário. Então devemos nos atentar e fazer requisições a apenas serviços existentes. Não precisaremos manipulá-lo, neste caso.

Sobrou apenas o serviço indisponível.

Para facilitar mostraremos apenas mensagens genéricas quando isto acontecer. Por exemplo:

“Não foi possível obter a lista de gêneros. Tente novamente mais tarde.”

Mas as mensagens trataremos em uma atividade futura.

Então criaremos o seguinte **enum** para manipular erros de retorno de serviço:

enum ReturnError: Error {  
 case apiError(code: Int, message:String)  
 case invalidJSON  
 case userMessage(String)  
}

Com ele podemos abranger três casos:

* apiError: será usado quando obtermos um erro da API (serviço web) mas não podemos mostrar para o usuário ainda. Este erro precisará de um tratamento posteriormente.
* invalidJSON: utilizaremos este caso quando houver uma falha de comunicação e o JSON recebido vier corrompido. Pode acontecer também de haver um erro de desenvolvimento do lado do backend e o serviço estar mandando JSONs mal formatados.
* userMessage: é um erro que já está pronto para ser exibido para o usuário.

Você pode optar por querer tratar os erros de desenvolvimento mencionados acima para, talvez, obter um maior controle e identificar bugs relacionados a isto mais rapidamente.

Teriamos a seguinte enumeração:

enum ValidationError: Error {  
 case missing(String)  
 case invalid(String, Any)  
}

Apresentando os casos:

* invalid: quando o JSON que estamos tentando mandar está mau formatado
* missing: quando um campo requerido do JSON está faltando.

**Dica:** Enums, structs, classes e protocolos podem ser declarados em qualquer arquivo. Não é necessário um arquivo para cada tipo como é feito no Java.

Finalizando o tratamento de erros precisamos criar uma estrutura que sirva para manipulação do erro e do sucesso (não erro). Podemos fazer isto com um **enum também**.

enum Result<T> {  
 case success(result: T)  
 case failure(error: ReturnError)  
}

Quando uma resposta do serviço for recebida iremos convertê-la em Result seja ela um erro ou o dado que esperamos. O tipo T é o que chamamos de **Generic** e podemos fazer com que o **case success** receba qualquer tipo de dado.