# Unidade 6

## Aula 1 - A classe Rest

Nesta aula aprendemos sobre o Firebase e as facilidades que ele nos proporciona, mas não podemos fazer tudo com ele. Para realizar requisições HTTP para outros serviços, inevitavelmente precisaremos de uma camada de serviço mais robusta e genérica do que a Firebase Database. Sendo assim, vamos seguir os mesmos passos do Android. Procurar ou desenvolver uma biblioteca para chamadas REST.

Swift possui algumas bibliotecas de terceiros que são bastante utilizadas pela comunidade. Mas também possui uma API nativa do iOS para tal, então é facil implementarmos nossa própria camada.

### Atividade

Pesquise e instale ou implemente a camada Rest. Para realizar requisições na API TMDB, como no Android, utilize a api key **0d2d0307fd89b460e176ba0033dc5c46**.

### Solução

Para fins didáticos iremos implementar nossa própria classe que realizará as chamadas HTTP REST. Então siga os seguintes passos.

1. Crie um novo arquivo Swift com o nome **Rest.swift**
2. Nele, crie alguns apelidos para facilitar a legibilidade do código.

typealias JSONObject = [String:Any]  
typealias JSONResult = Result<JSONObject>

JSONObject representará nossos JSONs após serem convertidos para Dictionary sendo que este dicionário é do tipo [String:Any] ou seja, possuirá uma chave do tipo String e um valor arbitrário Any, que significa que podemos inserir qualquer tipo de objeto.

Na aula 3 da unidade 3 criamos o enum Result. Este está preparado para receber um tipo genérico T e será utilizado para manipularmos o resultado da requisição em uma camada acima (mais abstrata) do que a camada de serviço REST. O apelido JSONResult é um atalho para criarmos um item do tipo Result que receberá um objeto do tipo JSONObject. Para entender melhor, continue até o final desta atividade.

1. Depois dos apelidar alguns elementos, optamos por mover os enums criados nas aulas anteriores para este arquivo:

enum ValidationError: Error {  
 case missing(String)  
 case invalid(String, Any)  
}  
  
enum ReturnError: Error {  
 case apiError(code: Int, message:String)  
 case invalidJSON  
 case userMessage(String)  
}  
  
enum Result<T> {  
 case success(result: T)  
 case failure(error: ReturnError)  
}  
  
enum Content<T> {  
 case success(T)  
 case error(ReturnError)  
 case loading  
}

1. Na classe Rest que criaremos, precisaremos realizar algumas validações básicas de JSON, como JSON mal formatado, identificando campos faltantes e etc. Para praticar um pouco mais, crie um novo protocolo que converterá um objeto Data em um objeto JSONObject:

protocol JSONValidation {  
 static func validJson(from data: Data?) throws -> JSONObject  
}

1. Por fim vamos criar nossa classe Rest:

class Rest {  
}

1. Logo em seguida, vamos implementar nosso protocolo.

extension Rest : JSONValidation {  
 static func validJson(from data: Data?) throws -> JSONObject {  
   
 // Se o objeto `data` é nulo, disparamos um erro de `invalidJSON`  
 guard let responseData = data  
 else { throw ReturnError.invalidJSON }  
   
 // Convertemos um JSON que ainda está em um objeto `Data` para um dicionário  
 let responseJson = try JSONSerialization.jsonObject(with: responseData, options: [])  
   
 // Certificamos que o dicionário gerado corresponde ao nosso tipo `JSONObject`, caso contrario disparamos um erro de `invalidJSON`.  
 guard let jsonObject = responseJson as? JSONObject  
 else {  
 throw ReturnError.invalidJSON  
 }  
   
 return jsonObject  
 }  
}

1. Agora iremos implementar a camada de serviço em si. Na classe Rest, adicione um enum de conveniência que definirá os métodos REST que possívelmente vc irá utilizar (no nosso caso utilizaremos somente o GET).

enum HTTPMethod: String {  
 case put = "PUT"  
 case get = "GET"  
 case post = "POST"  
 case delete = "DELETE"  
 }

1. Adicione uma variável que definirá a URL base das nossas requisições:

static var baseURL = "https://api.themoviedb.org/3/%@?api\_key=0d2d0307fd89b460e176ba0033dc5c46&language=pt-BR%@"

1. Uma requisição HTTP precisa de um header especificando algumas características da requisição. Nesta API precisaremos apenas especificar que estamos conversando via JSON.

static var defaultHeader: [String:String]? = ["Content-Type": "application/json"]

1. Crie uma constante estática do objeto URLSession será ele quem irá realizar as requisições:

static private let session = URLSession.shared

1. Por fim, crie o método mais importante da nossa classe. Chamos ele de connect. Ele irá preparar e disparar as requisições HTTPs para o nosso aplicativo. Leia os comentários no código para entender melhor seu funcionamento.

internal class func connect(method: String,  
 path: String,  
 query: String? = "",  
 timeout: TimeInterval? = 30,  
 redirects: Bool? = true,  
 headers: [String:String]? = defaultHeader,  
 jsonObject: JSONObject? = nil,  
 completion: @escaping (JSONResult) -> Void) throws {  
 // Setup URL  
   
 // Query string irá concatenar na URL um filtro na nossa requisição, como por exemplo, a página de filmes que queremos.  
 var queryString = query ?? ""  
 if !queryString.isEmpty {  
 queryString = "&\(queryString)"  
 }  
   
 // Concatenamos o path e a queryString à URL base  
 let appendedUrl = String(format: baseURL, path, queryString)  
   
 // Validamos se esta manipulação realmente gerou componentes de URL  
 guard let components = URLComponents(string: appendedUrl) else {  
 throw ValidationError.invalid("baseURL", baseURL)  
 }  
  
 // Validamos se estes componentes realmente possuem uma URL  
 guard let url = components.url else {  
 throw ValidationError.invalid("path", components.path)  
 }  
  
 // Configura o Request  
 var request = URLRequest(url: url, cachePolicy: .useProtocolCachePolicy, timeoutInterval: 30)  
 request.httpMethod = method // Método HTTP Rest que passamos via parâmetros  
 request.allHTTPHeaderFields = headers // Header básico definido acima  
 if let aTimeout = timeout { request.timeoutInterval = aTimeout } // Tempo máximo que o app esperará para obter uma resposta do serviço até cancelar a requisição  
  
 // Caso estejamos realizando uma requisição POST, validamos se existem um JSON body (passado via parâmetro)  
 if let validJson = jsonObject {  
 let jsonData = try JSONSerialization.data(withJSONObject: validJson,  
 options: .init(rawValue: 0))  
 request.setValue("application/json", forHTTPHeaderField: "content-type")  
 request.httpBody = jsonData  
 }  
  
 // Após preparar o request, utilizamos estes métodos para realizar a chamada de serviço em si.  
 session.dataTask(with: request, completionHandler: { data, response, error in  
  
 OperationQueue.main.addOperation {  
   
 // Neste momento já recebemos uma resposta que poderá estar nos argumentos data, response, ou error da closure de `dataTask`  
 do {  
 // Caso haja um erro, disparamos. O bloco `catch` irá capturá-lo  
 if let responseError = error { throw responseError }  
  
 // Utilizamos nosso método protocolado para converter o resultado `data`para `JSONObject`  
 let jsonObject = try validJson(from: data)  
  
 // Neste ponto ocorreu tudo certo. Encapsulamos nosso JSONObject em um JSONResult e enviamos para a closure.  
 completion(JSONResult.success(result: jsonObject))  
 } catch let theError {  
 // Os erros ocorridos no bloco `do` são enviados para cá  
 // Convertemos o erro obtido para um ReturnError que criamos em aulas passadas  
 let error = ReturnError.apiError(code: (theError as NSError).code, message: (theError as NSError).domain)  
 // Enviamos a falha para a closure e outra classe irá tratá-la  
 completion(JSONResult.failure(error: error))  
 }  
 }  
  
 }).resume()  
  
 }

1. O método connect() é internal, ou seja, só pode ser acessado dentro da própria classe ou pelas suas subclasses (semelhante ao protected do Java). Para facilitar a legibilidade do código e expor o acesso deste método para fora da classe, você pode criar métodos específicos para cada tipo de método HTTP. Vamos criar um para o GET.

// MARK: - Public methods  
 class func get(path: String, query: String? = "", headers: [String:String]? = defaultHeader, completion:@escaping (JSONResult) -> Void) {  
 do {  
 var allHeaders = defaultHeader  
 for (key, value) in headers! {  
 allHeaders?[key] = value  
 }  
   
 try connect(method: HTTPMethod.get.rawValue, path: path, query: query, redirects: false, headers: allHeaders,  
 completion: completion)  
 } catch {  
 completion(error as! JSONResult)  
 }  
 }

Sua carga de parâmetros é menor e temos uma versão mais simplificada do método connect().

Nossa classe Rest está terminada. Veja como ela ficou por completo:

import Foundation  
  
typealias JSONObject = [String:Any]  
typealias JSONResult = Result<JSONObject>  
  
enum ValidationError: Error {  
 case missing(String)  
 case invalid(String, Any)  
}  
  
enum ReturnError: Error {  
 case apiError(code: Int, message:String)  
 case invalidJSON  
 case userMessage(String)  
}  
  
enum Result<T> {  
 case success(result: T)  
 case failure(error: ReturnError)  
}  
  
enum Content<T> {  
 case success(T)  
 case error(ReturnError)  
 case loading  
}  
  
protocol JSONValidation {  
 static func validJson(from data: Data?) throws -> JSONObject  
}  
  
class Rest {  
 static var baseURL = "https://api.themoviedb.org/3/%@?api\_key=0d2d0307fd89b460e176ba0033dc5c46&language=pt-BR%@"  
 static var defaultHeader: [String:String]? = ["Content-Type": "application/json"]  
   
 static private let session = URLSession.shared  
  
 enum HTTPMethod: String {  
 case put = "PUT"  
 case get = "GET"  
 case post = "POST"  
 case delete = "DELETE"  
 }  
  
 internal class func connect(method: String,  
 path: String,  
 query: String? = "",  
 timeout: TimeInterval? = 30,  
 redirects: Bool? = true,  
 headers: [String:String]? = defaultHeader,  
 jsonObject: JSONObject? = nil,  
 completion: @escaping (JSONResult) -> Void) throws {  
 // Setup URL  
   
 // Query string irá concatenar na URL um filtro na nossa requisição, como por exemplo, a página de filmes que queremos.  
 var queryString = query ?? ""  
 if !queryString.isEmpty {  
 queryString = "&\(queryString)"  
 }  
   
 // Concatenamos o path e a queryString à URL base  
 let appendedUrl = String(format: baseURL, path, queryString)  
   
 // Validamos se esta manipulação realmente gerou componentes de URL  
 guard let components = URLComponents(string: appendedUrl) else {  
 throw ValidationError.invalid("baseURL", baseURL)  
 }  
  
 // Validamos se estes componentes realmente possuem uma URL  
 guard let url = components.url else {  
 throw ValidationError.invalid("path", components.path)  
 }  
  
 // Configura o Request  
 var request = URLRequest(url: url, cachePolicy: .useProtocolCachePolicy, timeoutInterval: 30)  
 request.httpMethod = method // Método HTTP Rest que passamos via parâmetros  
 request.allHTTPHeaderFields = headers // Header básico definido acima  
 if let aTimeout = timeout { request.timeoutInterval = aTimeout } // Tempo máximo que o app esperará para obter uma resposta do serviço até cancelar a requisição  
  
 // Caso estejamos realizando uma requisição POST, validamos se existem um JSON body (passado via parâmetro)  
 if let validJson = jsonObject {  
 let jsonData = try JSONSerialization.data(withJSONObject: validJson,  
 options: .init(rawValue: 0))  
 request.setValue("application/json", forHTTPHeaderField: "content-type")  
 request.httpBody = jsonData  
 }  
  
 // Após preparar o request, utilizamos estes métodos para realizar a chamada de serviço em si.  
 session.dataTask(with: request, completionHandler: { data, response, error in  
  
 OperationQueue.main.addOperation {  
   
 // Neste momento já recebemos uma resposta que poderá estar nos argumentos data, response, ou error da closure de `dataTask`  
 do {  
 // Caso haja um erro, disparamos. O bloco `catch` irá capturá-lo  
 if let responseError = error { throw responseError }  
  
 // Utilizamos nosso método protocolado para converter o resultado `data`para `JSONObject`  
 let jsonObject = try validJson(from: data)  
  
 // Neste ponto ocorreu tudo certo. Encapsulamos nosso JSONObject em um JSONResult e enviamos para a closure.  
 completion(JSONResult.success(result: jsonObject))  
 } catch let theError {  
 // Os erros ocorridos no bloco `do` são enviados para cá  
 // Convertemos o erro obtido para um ReturnError que criamos em aulas passadas  
 let error = ReturnError.apiError(code: (theError as NSError).code, message: (theError as NSError).domain)  
 // Enviamos a falha para a closure e outra classe irá tratá-la  
 completion(JSONResult.failure(error: error))  
 }  
 }  
  
 }).resume()  
  
 }  
  
 // MARK: - Public methods  
  
  
 class func get(path: String, query: String? = "", headers: [String:String]? = defaultHeader, completion:@escaping (JSONResult) -> Void) {  
 do {  
 var allHeaders = defaultHeader  
 for (key, value) in headers! {  
 allHeaders?[key] = value  
 }  
   
 try connect(method: HTTPMethod.get.rawValue, path: path, query: query, redirects: false, headers: allHeaders, completion: completion)  
 } catch {  
 completion(error as! JSONResult)  
 }  
 }  
  
}  
  
extension Rest : JSONValidation {  
 static func validJson(from data: Data?) throws -> JSONObject {  
 guard let responseData = data  
 else { throw ReturnError.invalidJSON }  
   
 // Convert JSON data to Swift JSON Object  
 let responseJson = try JSONSerialization.jsonObject(with: responseData, options: [])  
   
 // Assert there's a json object with status code  
  
 guard let jsonObject = responseJson as? JSONObject  
 else {  
 throw ReturnError.invalidJSON  
 }  
   
 return jsonObject  
 }  
}

## Aula 2 - Criando a camada Service

Finalmente estamos terminando a criação das camadas da nossa arquitetura. A camada de serviço, ou o que chamaremos de Service será responsável por realizar as requisições HTTP Rest fazendo uso da classe Rest. Com o JSONResult obtido da classe Rest, a camada Service fazer a conversão destes objetos para nossos ResponseModels que por sua vez serão utilizados na camada Interactor que será implementada na unidade 7.

As classes da camada Service tem uma característica comum: elas são **estáticas**. Isto porque não são utilizadas como instância em outros objetos e tão pouco guardam estado de algo. Elas são classes que funcionam mais como “Helpers” ou facilitadores, transendo objetos de um lado, manipulando-os e mandando para outro lado da arquitetura.

Para diminuir o acoplamento, os métodos estáticos das classes da camada Service devem receber como parâmetro apenas os dados necessários para que a requisição possa ser feita, exemplo:

A classe Service para as requisições de listagem e detalhes de filmes. Vamos criar o método para obter o detalhe do filme da forma errada:

class func getMovie(\_ movie: Movie.ResponseModel, completion: @escaping (Result<MovieDetails.ResponseModel>) -> Void) {  
 //...  
}

E veja a forma correta:

class func getMovie(withId id: Int, completion: @escaping (Result<MovieDetails.ResponseModel>) -> Void) {  
 //...  
}

Veja que no exemplo correto fornecemos somente o ID do filme para a requisição que será responsável por obter os detalhes do filme com tal ID. Este (ID) é o único dado que precisamos para realizar a requisição, então fornecemos apenas ele.

Esta é só uma técnica para diminuição do acoplamento. Se você quisesse testar este método isoladamente, não precisaria de criar uma instância de Movie.ResponseModel que possui muitas propriedades, mas precisaria apenas ter um ID em mãos.

Aproveitando que demos um exemplo de como será um método da camada de Service, veja que o segundo parâmetro é um **closure**:

@escaping (Result<MovieDetails.ResponseModel>) -> Void

Este será o retorno assíncrono que podemos esperar da requisição. Como dissemos acima, a camada Service pegará um JSONResult e converterá para **ResponseModel**.

### Atividade

Crie a camada Service para as cenas **Gêneros**, **Filmes** e \*\*Detalhes de Filmes \*\*. Crie um arquivo para cada cena e utilize o método get() da classe Rest para realizar a requisição.

### Solução

Vamos começar pela cena **Gêneros**.

1. No mesmo grupo que você criou a cena de Gêneros (GenresViewController) crie um novo arquivo swift com o nome **GenreService.swift**.
2. Neste arquivo crie uma classe de mesmo nome:

class GenreService {  
}

1. Adicione o método abaixo:

class func getGenres(completion: @escaping (Result<[Genre.ResponseModel]>) -> Void) { }

Este método deverá ser chamado para obter a lista de gêneros. Você pode ver que o tipo de retorno do closure é Result<[Genre.ResponseModel]>, ou seja, ele é um Result que poderá apresentar sucesso ou falha e, em caso de sucesso, terá uma lista ([]) de gêneros (Genre.ResponseModel).

1. Se você olhou a documentação do serviço de Gêneros nas primeiras aulas, verá que o caminho para alcançarmos esta lista é **"genre/movie/list"**. Dentro do método criado chame o método get() da classe Rest fornecendo este path como parâmetro:

Rest.get(path: "genre/movie/list") { jsonResult in  
  
}

1. Dentro do closure, vamos adicionar algum tratamento para o resultado. Primeiro adicione a clausula **do catch**:

Rest.get(path: "genre/movie/list") { jsonResult in  
 do {  
   
 } catch {  
   
 }  
}

\*\*Dica:\*\* Se você leu o artigo que passamos sobre Error Handling na aula 3 da unidade 3, terá visto também a ducumentação do try catch. Para relembrar visite: https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift\_Programming\_Language/ErrorHandling.html

1. No bloco **do** desencapsule o JSONResult e faça os devidos tratamentos:

Rest.get(path: "genre/movie/list") { jsonResult in  
 do {  
 switch jsonResult {  
 case .success(let json):  
   
 guard let genreList = json["genres"] as? [JSONObject] else {  
 throw ReturnError.userMessage("Não foi possível obter os Gêneros. Por favor tente novamente mais tarde")  
 }  
   
 let genres: [Genre.ResponseModel] = genreList.map {  
 Genre.ResponseModel(with: $0)  
 }  
   
 completion(Result.success(result: genres))  
 case .failure(let error):  
 throw error  
 }  
 } catch {  
   
 }  
}

1. Exponha o erro no bloco **catch**:

Rest.get(path: "genre/movie/list") { jsonResult in  
 do {  
 switch jsonResult {  
 case .success(let json):  
   
 guard let genreList = json["genres"] as? [JSONObject] else {  
 throw ReturnError.userMessage("Não foi possível obter os Gêneros. Por favor tente novamente mais tarde")  
 }  
   
 let genres: [Genre.ResponseModel] = genreList.map {  
 Genre.ResponseModel(with: $0)  
 }  
   
 completion(Result.success(result: genres))  
 case .failure(let error):  
 throw error  
 }  
 } catch {  
 completion(Result.failure(error: error as! ReturnError))  
 }  
}

A classe completa ficará assim:

class GenreService {  
   
 class func getGenres(completion: @escaping (Result<[Genre.ResponseModel]>) -> Void) {  
   
 Rest.get(path: "genre/movie/list") { jsonResult in  
   
 do {  
 switch jsonResult {  
 case .success(let json):  
   
 guard let genreList = json["genres"] as? [JSONObject] else {  
 throw ReturnError.userMessage("Não foi possível obter os Gêneros. Por favor tente novamente mais tarde")  
 }  
   
 let genres: [Genre.ResponseModel] = genreList.map {  
 Genre.ResponseModel(with: $0)  
 }  
   
 completion(Result.success(result: genres))  
 case .failure(let error):  
 throw error  
 }  
 } catch {  
 completion(Result.failure(error: error as! ReturnError))  
 }  
 }  
 }  
}

Para os serviços de Movie faremos o mesmo procedimento. Veja como nossa classe **MovieService** ficou:

class MovieService {  
   
 class func getPopularMovies(page: Int? = 1, completion: @escaping (Result<[Movie.ResponseModel.Result]>) -> Void) {  
   
 getMovies(page: page ?? 1, path: "movie/popular", completion: completion)  
 }  
   
 class func getTopRatedMovies(page: Int? = 1, completion: @escaping (Result<[Movie.ResponseModel.Result]>) -> Void) {  
   
 getMovies(page: page ?? 1, path: "movie/top\_rated", completion: completion)  
 }  
   
 class func getUpcomingMovies(page: Int? = 1, completion: @escaping (Result<[Movie.ResponseModel.Result]>) -> Void) {  
   
 getMovies(page: page ?? 1, path: "movie/upcoming", completion: completion)  
 }  
   
 class func getMovies(page: Int? = 1, path: String, completion: @escaping (Result<[Movie.ResponseModel.Result]>) -> Void) {  
   
 var query = ""  
   
 if let page = page, page > 1 {  
 query = "page=\(page)"  
 }  
   
 Rest.get(path: path, query: query) { jsonResult in  
   
 do {  
 switch jsonResult {  
 case .success(let json):  
   
 guard let movieList = json["results"] as? [JSONObject] else {  
 throw ReturnError.userMessage("Não foi possível obter os Filmes Populares. Por favor tente novamente mais tarde")  
 }  
   
 let movies = movieList.map {  
 Movie.ResponseModel.Result(with: $0)  
 }  
   
 completion(Result.success(result: movies))  
 case .failure(let error):  
 throw error  
 }  
 } catch {  
 completion(Result.failure(error: error as! ReturnError))  
 }  
 }  
 }  
   
 class func getMovie(withId id: Int, completion: @escaping (Result<MovieDetails.ResponseModel>) -> Void) {  
 Rest.get(path: "movie/\(id)") { jsonResult in  
   
 do {  
 switch jsonResult {  
 case .success(let json):  
   
 guard let \_ = json["id"] as? Int else {  
 throw ReturnError.userMessage("Não foi possível obter os detalhes do filme. Por favor tente novamente mais tarde")  
 }  
   
 let detail = MovieDetails.ResponseModel(with: json)  
   
 completion(Result.success(result: detail))  
 case .failure(let error):  
 throw error  
 }  
 } catch {  
 completion(Result.failure(error: error as! ReturnError))  
 }  
 }  
 }  
}

Sendo assim, temos a camada Service pronta.