PASO A PASO 10:
Networking (con Alamofire)

Vamos a crear una aplicación que muestre en una tabla información de distintos productos. Particularmente, la información se va a obtener de un JSON leído de una API remota [1]. Asimismo, estructuraremos la aplicación siguiendo los fundamentos de MVC.

[1] https://api.myjson.com/bins/1411cr

1. Comenzamos creando un proyecto, definiendo la estructura de carpetas y agregando la siguiente librería:
   a. Alamofire.

2. Creamos las tres clases que vamos a usar, en las carpetas asociadas:
   a. ProductsTableViewController, en el grupo 'viewcontrollers'.
   b. ProductService, en el grupo 'service'.
   c. ProductDAO, en el grupo 'dao'.
   d. Product, en el grupo 'model'.

3. Implementación del MODEL:
   a. La única precaución que debemos tener al armar el modelo, es que defina un constructor que nos permita inicializar instancias a partir de un diccionario. De esta forma, cuando obtengamos un JSON, vamos a poder crear el objeto correspondiente y configurar sus atributos.

4. Implementación del DAO [***anexo 1***]:
   a. Definimos el método que se va a encargar de hacer el request al servidor, obtener el JSON, parsearlo a un array de objetos y devolverlo a quien corresponda. Este método debe recibir un **closure** (bloque) como parámetro, que se invocará cuando se haya finalizado el procesamiento. Particularmente, ese closure va a recibir el resultado de la operación como parámetro; por lo tanto el closure recibirá un parámetro de tipo "`[Product]`". Recordar que en la definición del tipo de bloque debemos incluir la palabra clave "`@escaping`".

```
func getProductsFromAPI(daoCompleted:
                        @escaping (([Product]) -> Void)) -> Void
```

   b. Para hacer el request invocamos el método "`request(url)`" de Alamofire (donde url es un parámetro de tipo String con la url del endpoint), y sobre el resultado invocamos el método "`responseJSON`" que toma como parámetro un closure. Este closure recibe como parámetro, el resultado (o "response") del request realizado.

```
Alamofire.request("https://myapi.com/request").responseJSON {
      (response) in
      // instructions go here
}
```

c. Dentro del bloque que se pasa como parámetro al método "`responseJSON`" vamos a obtener el JSON leído del servidor, y luego vamos a parsearlo tal como hacíamos con los archivos .plist: se obtuvo un JSON (en `response.value`) que es un array de diccionarios, se itera el array y cada diccionario se usa para instanciar un nuevo objeto de tipo `Product`.

```
if let rootArray = response.value as? [ [String:AnyObject] ] {
    var productsResultArray: [Product] = []
    for aDictionary in rootArray {
        let aProductObject = Product(dictionary: aDictionary)
        productsResultArray.append(aProductObject)
    }
}
```

d. Finalmente, cuando se termina todo el procesamiento, se invoca el closure recibido como parámetro ("`daoCompleted`") y se le pasa el array construido como parámetro.

```
daoCompleted(productsResultArray)
```

5. Implementación del Service [***anexo 2***]:
   a. Definimos el método que se va a encargar de instanciar un objeto DAO e invocar su método "`getProductsFromAPI`". Al igual que en el caso anterior, este método debe recibir un **closure** (bloque) como parámetro, que se invocará cuando se haya finalizado el procesamiento; y el closure, a su vez, recibirá un parámetro de tipo "`[Product]`". Recordar que en la definición del tipo de bloque debemos incluir la palabra clave "`@escaping`".

```
func getProducts(serviceCompleted:
                 @escaping (([Product]) -> Void)) -> Void
```

   b. Simplemente instanciamos un objeto de tipo `ProductDAO` y luego invocamos el método creado en el paso anterior. Dentro del closure pasado como parámetro al DAO, se invoca el closure recibido como parámetro en el Service ("`serviceCompleted`"); pasándole como parámetro el valor recibido desde el DAO.

```
let productDAO = ProductDAO()
productDAO.getProductsFromAPI(daoCompleted: { (products) in
      serviceCompleted(products)
})
```

6. Implementación del ViewController [***anexo 3***]:

a. En el viewDidLoad, vamos a iniciar el proceso de request al servidor. Instanciamos un objeto de tipo `ProductService` e invocamos el método creado en el paso anterior. Dentro del closure pasado como parámetro al Service, se recibe el array de productos ("`[Product]`") que se necesita para configurar la tabla; simplemente lo guardamos en una variable local y luego recargamos la tabla.

```
let productService = ProductService()
productService.getProducts(serviceCompleted: { (products) in
      self.products = products
      self.tableView.reloadData()
})
```

b. Finalmente, se completa la implementación del `TableViewController`.

Anexo 1 - DAO

```
import Foundation
import Alamofire

class ProductDAO {

  func getProductsFromAPI(daoCompleted: @escaping (([Product]) -> Void)) -> Void {

    Alamofire.request("https://api.myjson.com/bins/1411cr").responseJSON {
      (response) in
      // Desde ACA comienza la definicion de un bloque asincronico.
      // En este caso particular, es un bloque que recibe un parametro
      // llamado "response".
      // El parametro recibido va a contener la respuesta del servidor.

      // 'response.value' contiene el JSON leido del servidor,
      // pero por defecto es un AnyObject,
      // por eso es que hay que hacer el 'if let ... as? [[String:AnyObject]]'

      if let rootArray = response.value as? [ [String:AnyObject] ] {

        // ACA 'rootArray' es el JSON, entendido como un Array de Diccionarios;
        // y cada diccionario representa un objeto Product.

        // Se define el array que va a agrupar todos los objetos parseados.
        var productsResultArray: [Product] = []

        // Se itera el array, y se procesa cada diccionario individualmente.
        for aDictionary in rootArray {

          // ACA 'aDictionary' es un diccionario
          // que representa a un objeto Product.

          let aProductObject = Product(dictionary: aDictionary)
          productsResultArray.append(aProductObject)

        }

        // ACA termine de procesar todos los diccionarios del JSON.
        // Como ya tengo parseados todos los objetos,
```

```
        // puedo invocar el closure para avisar que se termino el
        // procesamiento del DAO, y paso como parametro el resultado.
        daoCompleted(productsResultArray)
      }

      // ACA termina la definicion del bloque asincronico.
    }
  }
}
```

Anexo 2 - Service

```
import Foundation

class ProductService {

  func getProducts(serviceCompleted: @escaping ( ([Product]) -> Void ) ) -> Void {

    // Se instancia un objeto de tipo ProductDAO
    let productDAO = ProductDAO()

    // Se invoca el método y se pasa un bloque como parámetro
    productDAO.getProductsFromAPI(daoCompleted: { (products) in
      // ACA comienza la definicion del bloque asincronico.
      // En este caso particular, es un bloque que recibe
      // un parametro llamado "products", que es el array
      // de productos leido y parseado por el DAO.

      // Simplemente se invoca el closure recibido como parametro (del metodo)
      // pasando como parametro el array recibido como parametro (del closure)
      serviceCompleted(products)

      // ACA termina la definicion del bloque asincronico.
    })
  }
}
```

Anexo 3 - ViewController

```
import UIKit

class ProductsTableViewController: UITableViewController {

  var products:[Product] = []

  override func viewDidLoad() {
```

```
    super.viewDidLoad()

    // Se instancia un objeto de tipo ProductService
    let productService = ProductService()

    // Se invoca el método y se pasa un bloque como parámetro
    productService.getProducts(serviceCompleted: { (products) in
      // ACA comienza la definicion del bloque asincronico.
      // En este caso particular, es un bloque que recibe
      // un parametro llamado "products", que es el array
      // de productos obtenido por el service.

      self.products = products
      self.tableView.reloadData()

      // ACA termina la definicion del bloque asincronico.
    })
  }

  override func numberOfSections(in tableView: UITableView) -> Int {
    return 1
  }

  override func tableView(_ tableView: UITableView,
                          numberOfRowsInSection section: Int) -> Int {
    return products.count
  }

  override func tableView(_ tableView: UITableView,
                          cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "productCell",
                                             for: indexPath)
    if let productCell = cell as? ProductTableViewCell {
      productCell.setup(news: products[indexPath.row])
    }
    return cell
  }
}
```