06. Работа с сетью

Alamofire

Alamofire – это свободно распространяемая, библиотека с открытым исходным кодом, для работы с сетью.

В **Alamofire** доступно несколько основных методов:

- **upload**: позволяет передавать файлы на сервер при помощи методов `multipart`, `stream`, `file` или `data`
- **download**: позволяет загружать данные, а так же возобновлять ранее приостановленные загрузки
- **request**: позволяет выполнять любой запрос по протоколу HTTP, не связанный с передачей файлов

Все эти методы являются глобальными и для их использования не надо создавать экземпляр класса **Alamofire**.

1. Добавляем в проект Alamofire через cacaopods.
2. Переходим в класс `ViewController` и импортируем фреймворк `import Alamofire`
3. Переходим в метод `sendRequest` и удаляем кусок кода относящийся к `URLSession`

Выполнение запроса при помощи метода `request`:

```
@IBAction func sendRequest(_ sender: Any) {
```

```
    let jsonUrlString = "https://swiftbook.ru//wp-content/
uploads/api/api_courses"

    guard let url = URL(string: jsonUrlString) else { return }

    request(url).responseJSON { (response) in
        print(response)
    }
}
```

У класса Alamofire есть метод request, который в качестве параметров принимает адрес, по которому следует послать запрос. Метод `responseJSON` говорит о том, что ответ от сервера нам нужен в JSON формате. Далее в клоужере мы получаем ответ от сервера и выводим его на консоль.

Так же, как и в URLSession, если мы не указываем тип запроса, то по умолчанию выполняется GET запрос. Если вам необходимо задать явно тип запроса, то его можно передать в параметр `method` метода `request`:

```
request(url, method: .get).responseJSON { (response) in
    print(response)
}
```

Полный запрос со всеми параметрами:

```
request(url: URLConvertible, method: HTTPMethod, parameters:
Parameters?, encoding: ParameterEncoding, headers: HTTPHeaders?)
```

- *URLConvertible* – это протокол с одной функцией, которая возвращает URL. Данный протокол реализован для таких типов данных, как `String`, `URL` и `URLComponents`, соответственно в качестве параметра мы можем передавать любой из вышеперечисленных типов:

```
public protocol URLConvertible {
    func asURL() throws -> URL
```

```
}
```

- *HTTPMethod* – это enum, со всеми возможными типами запросов:

```
public enum HTTPMethod: String {
    case options = "OPTIONS"
    case get     = "GET"
    case head    = "HEAD"
    case post    = "POST"
    case put     = "PUT"
    case patch   = "PATCH"
    case delete  = "DELETE"
    case trace   = "TRACE"
    case connect = "CONNECT"
}
```

- *Parameters?* – это словарь, через параметры которого данные передаются на сервер для создания, изменения или удаления объектов:

```
public typealias Parameters = [String: Any]
```

- *ParameterEncoding* – протокол, определяющий кодировку передаваемых параметров. Данный проток реализуют методы `URLEncoding`, `JSONEncoding` и `PropertyListEncoding`:

```
public protocol ParameterEncoding {
    func encode(_ urlRequest: URLRequestConvertible, with
parameters: Parameters?) throws -> URLRequest
}
```

- *HTTPHeaders?* – словарь с типом `[String: String]`, который может использоваться для авторизации

```
public typealias HTTPHeaders = [String: String]
```

В итоге на выходе мы получаем объект класса `DataRequest`, в котором содержится сам запрос и мы можем его сохранить, передать в качестве параметра в другой метод или отправить.

Обработка ответа

Для определения статуса запроса используется параметр `statusCode`. Ответ так же можно обрабатывать вручную. В этом случае можно воспользоваться соответствующими методами:

- `responseJSON.response?.statusCode` – возвращает статус код ответа
- `responseJSON.result.value` – возвращает значение, в случае, если ответ пришел без ошибки.
- `responseJSON.result.error` – возвращает ошибку

```
request(url, method: .get).responseJSON { (responseJSON) in

    guard let statusCode = responseJSON.response?.statusCode else
{ return }

    print("statusCode: ", statusCode)

    if (200..<300).contains(statusCode) {
        let value = responseJSON.result.value
        print("value: ", value ?? "nil")
    } else {
        print("error")
    }
}
```

Настройка запроса

Класс `DataRequest` содержит 4 метода:

- `validate(statusCode: _ )`
- `validate(contentType: _ )`
- `validate(клоужер для ручной валидации)`
- `validate()`

Рассмотрим пример с использованием метода `validate()`:

```
public func validate() -> Self {
    return validate(statusCode:
self.acceptableStatusCodes).validate(contentType:
self.acceptableContentTypes)
}
```

Видим, что он состоит из двух других валидаций:

- `self.acceptableStatusCodes` – возвращает массив статус кодов с типом `Int` в диапазоне от 200 до 299
- `self.acceptableContentTypes` – возвращает массив допустимых заголовков с типом `String`

У `DataResponse` есть параметр `result`, который может сказать нам, пришел ответ с ошибкой или с результатом.

Итак, применим валидацию для запроса:

```
request(url, method: .get).validate().responseJSON
{ (responseJSON) in

    switch responseJSON.result {
    case .success(let value):
        print(value)
    case .failure(let error):
        print(error)
```

```
    }
}
```

Если у нас не будет вылидации запроса (validate()), то result всегда будет равен .success, за исключением ошибки из-за отсутствия интернета, соответсвенно что бы минимизировать количество ошибок используйте запрос с валидацией.

Обработка результата ответа

Так как мы работаем с сетью, то для выполнения запроса и получения ответа требуется какое-то время, поэтому одним из параметров, который принимает метод `responseJSON` является замыкание, которое выполнится после получения ответа. Это замыкание в качестве параметра принимает объект `ResponseData`, полученный от сервера. Ответ от сервера может быть в виде какого то объекта или массива словарей.

```
request(url, method: .get).responseJSON { (responseJSON) in

    switch responseJSON.result {
    case .success(let value):
        print("value", value)
    case .failure(let error):
        print(error)
    }
```

Объект `responseJSON`, с которым мы работаем содержит ответ сервера на наш запрос. В кейсе `.success` мы присваиваем значение ответа с типом `Any` константе `value` и выводим это значение на консоль. В кейсе `.failure` выводим на консоль ошибку.

Так как наш ответ имеет тип `Any`, то для дальнейшей работы с ним, его необходимо привести к нужному формату данных, т.е. это должен быть массив словарей с типом `[String: Any]`:

Присвоим полученные данные элементам структуры:

```
request(url, method: .get).responseJSON { (responseJSON) in

    switch responseJSON.result {
    case .success(let value):

        guard let arrayOfItems = value as? Array<[String: Any]>
else { return }

        for field in arrayOfItems {
            let course = Course(id: field["id"] as? Int,
                                name: field["name"] as? String,
                                imageUrl: field["imageUrl"] as?
String,
                                numberOfLessons:
field["number_of_lessons"] as? Int,
                                numberOfTests:
field["number_of_tests"] as? Int)

            self.courses.append(course)
        }

        DispatchQueue.main.async {
            self.tableView.reloadData()
        }

    case .failure(let error):
        print(error)
    }
}
```

Мы взяли данные из `response.result.value` и пробуем перевести их в массив словарей, с ключом типа `String` и значением типа `Any`. Формат такой именно потому, что данные приходят именно так. Если по какой-либо причине

данные пришли в другом формате, то на консоль выйдет сообщение с ошибкой. Затем цикл перебирает элементы по одному, на основании данных из них создает объекты Item и помещает эти объекты в массив (свойство для хранения элементов структуры).

В **Alamofire** так же, как и в **URLSession** запросы выполняются в фоновом режиме, поэтому для обновления интерфейса необходимо запустить метод `reloadData()` в основном потоке.

Создание инициализатора в структуре данных:

```
init?(json: [String: Any]) {

    guard
        let id = json["id"] as? Int,
        let name = json["name"] as? String,
        let imageUrl = json["imageUrl"] as? String,
        let numberOfLessons = json["number_of_lessons"] as? Int,
        let numberOfTests = json["number_of_tests"] as? Int
    else { return nil }

    self.id = id
    self.name = name
    self.imageUrl = imageUrl
    self.numberOfLessons = numberOfLessons
    self.numberOfTests = numberOfTests
}
```

Данный инициализатор мы делаем опциональным и в том случае, если сервер нам что то не вернет, то мы получим `nil`. Соответственно сами свойства структуры уже не обязательно делать опциональными. Сделав свойства структуры не опциональными, отменяем извлечение опционала при конфигурации ячейки:

```
cell.numberOfLessons.text = "Number of lessons: \
(course.numberOfLessons)"
```

```
cell.numberOfTests.text = "Number of tests: \
(course.numberOfTests)"
```

Поменяем запрос в соответствии с новыми условиями:

```
for field in arrayOfItems {
    guard let course = Course(json: field) else { return }
    self.courses.append(course)
}
```

Можно еще больше оптимизировать данный код, если мы добавим в нашей структуре метод обработки массива:

```
static func getArray(from jsonArray: Any) -> [Course]? {

    guard let jsonArray = jsonArray as? Array<[String: Any]> else
{ return nil }
    var courses: [Course] = []

    for jsonObject in jsonArray {
        if let course = Course(json: jsonObject) {
            courses.append(course)
        }
    }
    return courses
}
```

Оптимизируем запрос:

```
switch responseJSON.result {
case .success(let value):
    self.courses = Course.getArray(from: value)!
case .failure(let error):
    print(error)
}
```

Можно оптимизировать обработку массива, используя метод `compactMap`:

```
static func getArray(from jsonArray: Any) -> [Course]? {
    guard let jsonArray = jsonArray as? Array<[String: Any]> else
{ return nil }
```

```
    return jsonArray.compactMap { Course(json: $0) }
}
```

POST запросы

Рассмотрим примеры POST запросов. В качестве сервиса для тестирования таких запросов мы снова будем использовать сервис JSON Placeholder.

Для начала создадим свойство с параметрами, которые мы хотим отправить на сервер:

```
let userData: [String: Any] = [
    "name": "Network Requests",
    "lesson": "POST Request with Alamofire",
    "numberOfLessons": 18,
    "numberOfTests": 10
]
```

Поменяем ссылку для создания запроса

```
let jsonUrlString = "https://jsonplaceholder.typicode.com/posts"
```

Создадим запрос с типом метода и параметрами:

```
request(url, method: .post, parameters:
userData).validate().responseJSON { responseJSON in

}
```

В принципе этот запрос уже сам по себе отправляет данные на сервер, и нам остается только проверить успешность выполнения запроса:

```
guard let statusCode = responseJSON.response?.statusCode else
{ return }
print("statusCode: ", statusCode)
```

Выведем на консоль результат, в случае успешной выгрузки:

```
if (200..<300).contains(statusCode) {
```

```
    let value = responseJSON.result.value
    print("value: ", value ?? "nil")
} else {
    print("error")
}
```

Хранение данных

UserDefaults

Класс `UserDefaults` позволяет хранить в памяти устройства небольшие объемы данных. `UserDefaults` позволяет пользователю повторно не вводить свое имя, выбранную им валюту для отображения цен, или количество максимально набранных очков в игре.

Пример сохранение имени пользователя:

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

    guard firstNameTextField.text?.isEmpty == false else {
        wrongFormatAlert()
        return
    }
    guard secondNameTextField.text?.isEmpty == false else {
        wrongFormatAlert()
        return
    }

    if let _ = Double(firstNameTextField.text!) {
        wrongFormatAlert()
    } else if let _ = Double(secondNameTextField.text!) {
        wrongFormatAlert()
    } else {
        self.userNameLabel.text = firstNameTextField.text! + " "
+ secondNameTextField.text!
```

```
        UserDefaults.standard.set(userNameLabel.text, forKey:
"UserName") // Сохранение данных в UserDefaults
    }
}
```

Восстановление сохраненных данных:

```
override func viewDidLoad() {
    super.viewDidLoad()

    if let userName = UserDefaults.standard.value(forKey:
"UserName") {
        userNameLabel.isHidden = false
        userNameLabel.text = userName as? String
    }
}
```

CoreData

Подготовка приложения:

Создаем приложение с использованием **Core Data**

Переходим в класс `AppDelegate`

В методе `didFinishLaunchingWithOptions` создаем новое окно, делаем его ключевым и в качестве стартового вью контроллера назначаем `UINavigationController`

```
window = UIWindow(frame: UIScreen.main.bounds)
window?.makeKeyAndVisible()

window?.rootViewController =
UINavigationController(rootViewController: ViewController())
```

Переходим в класс `ViewController` и в методе `viewDidLoad` работаем над внешним видом приложения:

```
// Цвет заливки вью контроллера
```

```
view.backgroundColor = .white


// Цвет навигейшин бара
navigationController?.navigationBar.barTintColor =
UIColor(displayP3Red: 21/255, green: 101/255, blue: 192/255,
alpha: 1)


// Добавляем кнопку "Добавить" в навигейшин бар
navigationItem.rightBarButtonItem = UIBarButtonItem(title:
"Добавить", style: .plain, target: self, action:
#selector(addItem)) // Вызов метода для кнопки
```

Объявляем метод `addItem`, что бы компилятор не ругался:

```
// Действие при нажатии на кнопку "Добавить"
@objc func addItem(_ sender: AnyObject) {
}
```

Снова переходим в метод `viewDidLoad`:

```
// Цвет текста для кнопки
navigationController?.navigationBar.tintColor = .white
```

Меняем суперкласс для `ViewController` на `UITableViewController`

Добавляем новое свойство класса, для идентификации ячейки:

```
var cellId = "Cell" // Идентификатор ячейки
```

Переходим во `viewDidLoad` и присваиваем ячейку для TableView:

```
// Присваиваем ячейку для TableView с иднетификатором "Cell"
tableView.register(UITableViewCell.self, forCellReuseIdentifier:
cellId)
```

В свойствах класса создаем массив для хранения данных:

```
var itemsArray = [String]() // Массив для хранения записей
```

Имплементируем методы Table View Data Source:

```
//MARK: Table View Data Source

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

override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier:
cellId, for: indexPath)
    let item = itemsArray[indexPath.row]
    cell.textLabel?.text = item
    return cell
}
```

Переходим в метод `addItem`:

```
// Создание алёрт контроллер
let alert = UIAlertController(title: "Новая задача", message:
"Пожалуйста заполните поле", preferredStyle: .alert)


// Создание кнопки для сохранения новых значений
let saveAction = UIAlertAction(title: "Сохранить",
style: .default) { action in

    // Проверяем не является ли текстовое поле пустым
    guard alert.textFields?.first?.text?.isEmpty == false else {
        print("The text field is empty") // Выводим сообщение на
консоль, если поле не заполнено
        return
    }

    // Добавляем в массив новую задачу из текстового поля
    self.itemsArray.append((alert.textFields?.first?.text)!)

    // Обновляем таблицу
    self.tableView.reloadData()
}
```

```
// Создаем кнопку для отмены ввода новой задачи
let cancelAction = UIAlertAction(title: "Отмена",
style: .destructive, handler: nil)


alert.addTextField(configurationHandler: nil) // Присваиваем
алёрту текстовое поле
alert.addAction(saveAction) // Присваиваем алёрту кнопку для
сохранения результата
alert.addAction(cancelAction) // Присваиваем алерут кнопку для
отмены ввода новой задачи

present(alert, animated: true, completion: nil) // Вызываем алёрт
контроллер
```