# Разработка под iOS. Взлетаем

Часть 2

# Работа с сетью

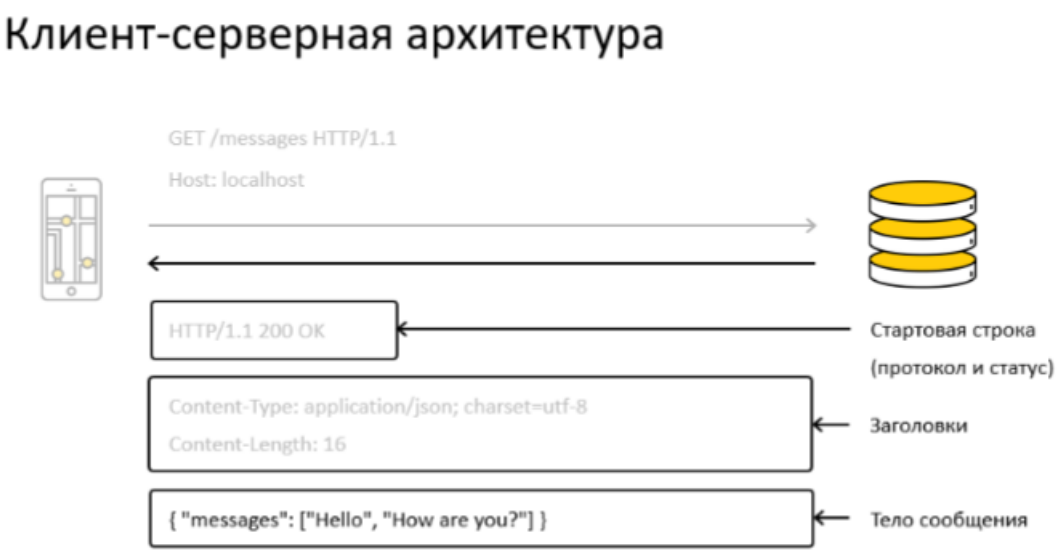
# Оглавление

# Глава 2

## 1 Разработка под iOS. Взлетаем

### 2.1. Базовые концепции сетевого взаимодействия

Всем привет! Меня зовут Вероника и я занимаюсь iOS-разработкой в Яндексе. На этой неделе я расскажу вам о взаимодействии с сетью в iOS.
HTTP - это протокол обмена, он текстовый, у него есть определенная структура (она описана в документе RFC 7230). Как мобильным разработчикам, нам не нужно знать низкоуровневые детали работы протокола HTTP, нам достаточно понимания высокоуровневой абстракции. В iOS есть реализация HTTP «из коробки». Платформа предоставляет нам класс URLSession для осуществления сетевых запросов.
Клиент-серверное взаимодействие выглядит примерно следующим образом: девайс обращается к серверу и просит, например, список сообщений, а сервер говорит: "Держи, вот твои сообщения".




Как же все это выглядит с точки зрения HTTP? Клиент, обращаясь к серверу, передает сообщение примерно следующего содержания:

- определяет метод - в данном примере это GET;
- ресурс, в нашем случае - messages;
- а также протокол и хост.

А сервер в ответ отправляет сообщение, содержащее:

- протокол и статус выполненного запроса;

- заголовки;
- тело ответа (в данном случае - JSON).

Конечно, всё намного сложнее, но главное, что нам нужно запомнить - это то, что для отправки и получения данных с помощью HTTP клиенту необходимо сделать запрос к серверу и этот запрос имеет определенный формат.
В HTTP есть ряд глаголов (их еще называют методами). Они определяют тип запроса, то есть сообщают то, что мы хотим сделать с ресурсом.
Основные HTTP глаголы следующие:

- `GET` (read) - используется для запроса содержимого по указанному ресурсу.
- `PUT` (update) - позволяет создать новую сущность или перезаписать существующую по указанному адресу.
- `POST` (create) - позволяет модифицировать существующий ресурс или создать новый.
- `DELETE` (delete) - отвечает за удаление указанного ресурса.

Существует некоторая договоренность, или, иначе говоря, архитектурный стиль, определяющий набор ограничений на создание веб-сервисов. Он называется REST.
И, говоря по-простому, основное положение идеологии REST состоит в том, что url запроса, который вы делаете - это имя существительное (оно указывает на ресурс к которому происходит обращение) и глагол (то, что хотим сделать с ресурсом).
Например, когда вы открываете главную страницу поисковой системы Яндекс, вы делаете GET запрос.
Коротко поговорим про статус-коды.
Все статус-коды разделены по зонам, и первая цифра определяет тип этой зоны.
Диапазон от 200 до 300 (верхняя граница не включается) говорит о том, что всё прошло хорошо. Обычно это 200, означающее "OK".
Последующие коды до 400-го (верхняя граница не включается) говорят о том, что было совершено перенаправление. Например, код 301 означает, что ресурс был навсегда перемещен.
Диапазон кодов с 400 по 500 (верхняя граница не включается) описывает ошибки клиента. Это могут быть сообщения о необходимости авторизации (401), обновления или истечения времени ожидания.
Пятисотые же ошибки обозначают ошибки сервера. Одна из самых распространенных - это 500, и она говорит о том, что произошла внутренняя ошибка сервера.

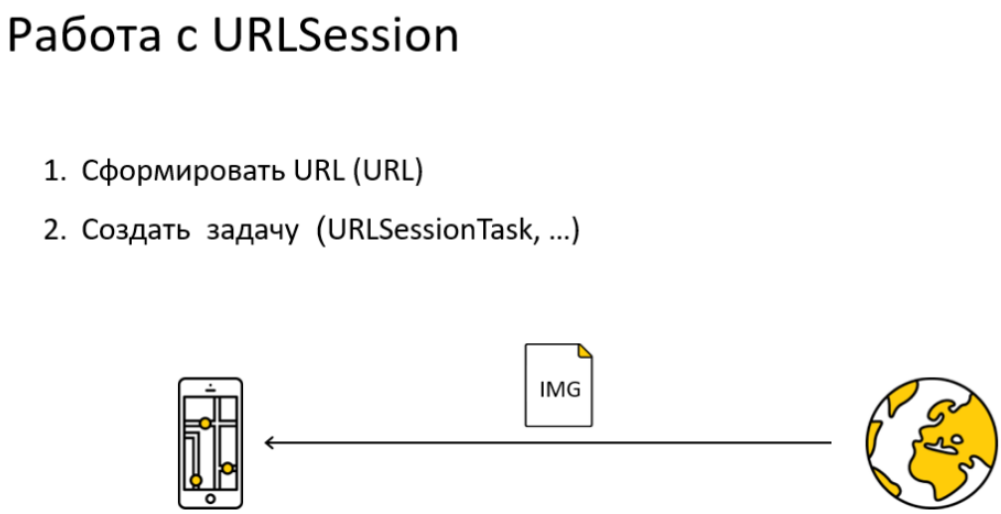
```
200..<300 - success
300..<400 - redirection
400..<500 - client errors
500+ - server errors
```

На этом краткое теоретическое вступление завершается, в следующем разделе мы приступим к взаимодействию с сетью в iOS.

## 2.2. Сетевой запрос в iOS

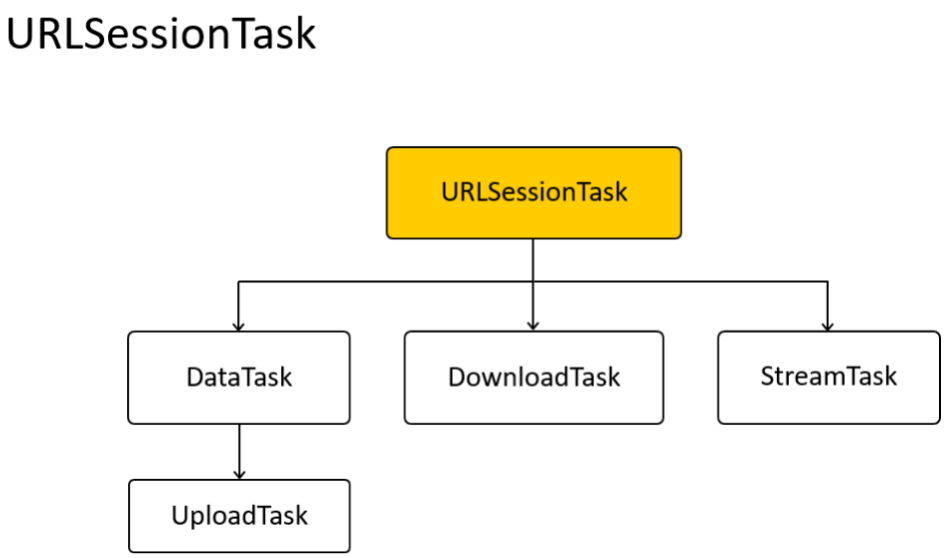
**Часть 1**

В прошлом разделе мы поговорили про клиент-серверную архитектуру и HTTP протокол.
‘ Сейчас мы посмотрим, как всё это выглядит в Swift, и научимся загружать картинки из сети.
Для создания и управления HTTP запросами в iOS используется класс URLSession. Мы можем создать свой собственный экземпляр URLSession с нестандартными настройками или использовать предоставляемый синглтон URLSession с настройками по умолчанию.
Для загрузки картинки нам будет вполне достаточно сессии с настройками по умолчанию. Таким образом, нам потребуется всего два действия:
1. Сформировать URL (URL).
2. Создать задачу (URLSessionTask).




Сформируем URL (Uniform Resource Locator) - путь к загружаемой картинке. Для этого воспользуемся стандартным конструктором класса URL.

```
let url = URL(string: "https://...images/1.png")!
```



То, что ранее мы называли запросами, в контексте URLSession называется задачами. Любая задача, используемая классом URLSession, будет являться подклассом URLSessionTask. URLSessionTask - это объект, который умеет работать с данными по url.
Существует четыре основных типа задач, которые мы можем использовать в работе:

- **DataTask** - возвращает данные непосредственно в приложение, то есть в память, в виде одного или нескольких объектов типа Data.

- **DownloadTask** - сохраняет загруженные данные из сети во временный файл, предоставляя приложению информацию о прогрессе поступления данных с сервера.

- **UploadTask** - специальный тип задач для загрузки файлов, используется для создания HTTP-запросов, для которых требуется тело запроса (например, `POST` или `PUT`).

- **StreamTask** - предоставляет интерфейс TCP/IP соединения, созданного через URLSession.

В этом курсе мы рассмотрим только DataTasks.
Для загрузки картинки из сети по сформированному url необходимо создать задачу. Воспользуемся синглтоном URLSession.

```
let task = URLSession.shared.dataTask(with: url){
    data, response, error in

    print("Done!")

    DispatchQueue.main.async{
        self.label.text = "ok"
    }
}

task.resume()
```

По завершению задачи вызывается completion block, принимающий три параметра с типами: Data, URLResponse и Error. Как их обрабатывать, мы посмотрим позднее. Задача может завершиться через несколько миллисекунд или даже через минуту, мы этого точно не знаем. И в этом основная идея этого блока кода - он выполнится сразу после завершения задачи.
Стоит отметить, что задача выполняется на своей отдельной очереди и, например, чтобы поменять элементы интерфейса, необходимо перейти на главную очередь
Важный момент также в том, что URLSession не запускает задачи сразу после их создания, и мы остаемся ответственны за запуск созданных тасков. Поэтому после создания задачи необходимо вызвать метод resume, который запустит выполнение задачи. Возможно, это выглядит странным, но на самом деле это дает нам большую гибкость: мы можем запускать, останавливать и возобновлять задачи именно тогда, когда нам это необходимо. Например, если задача выполняется очень долго и пользователь захотел остановить процесс загрузки, мы можем вызвать метод cancel у задачи, которую мы хотим отменить, и она перестанет выполняться.
На этом все, а далее мы проверим это на практике.

**Часть 2**

Только что я рассказала про базовые аспекты работы с сетью в iOS, а теперь мы попробуем это на практике. Загрузим картинку из сети и отобразим ее на контроллере.
Скопируем url из браузера и создадим объект класса URL в нашем проекте:

```
let stringURL = "https://avatars.mds.yandex.net/get-bunker/
    135516/7a83e9a35b9c1537ba8fe3a5cf1ef182838a11be/orig"
let url = URL(string: stringUrl)
```

Создание объекта класса url из строки не гарантируется, поэтому обернем инициализацию в конструкцию guard:

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

Далее необходимо создать задачу для загрузки картинки. Для этого воспользуемся синглтоном URLSession, который предоставляется платформой. Вызовем метод `dataTask(with url)` и в completion блоке для начала просто выведем сообщение в консоль:

```
print("Done!")
```

Запустим проект.
И у нас ничего не произошло. А все потому, что мы не вызвали метод resume для созданной задачи.
Исправим код:

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var imageView: UIImageView!

    override func viewDidLoad() {
        super.viewDidLoad() {
        let string = "https://avatars.mds.yandex.net/get-bunker/
            135516/7a83e9a35b9c1537ba8fe3a5cf1ef182838a11be/orig"
        guard let url = URL(string: string) else { return }
        let task = URLSession.shared.dataTask(with: url) { (data,
            response, error) in
         print("Done!")
        }
        task.resume()
```

И перезапустим проект.
Теперь мы видим сообщение в консоли, а значит наша задача действительно завершилась.
Как же получить картинку из типа Data? Очень просто: UIImage класс предоставляет инициализатор, который принимает параметр типа Data.

```
let image = UIImage(data: data)
```

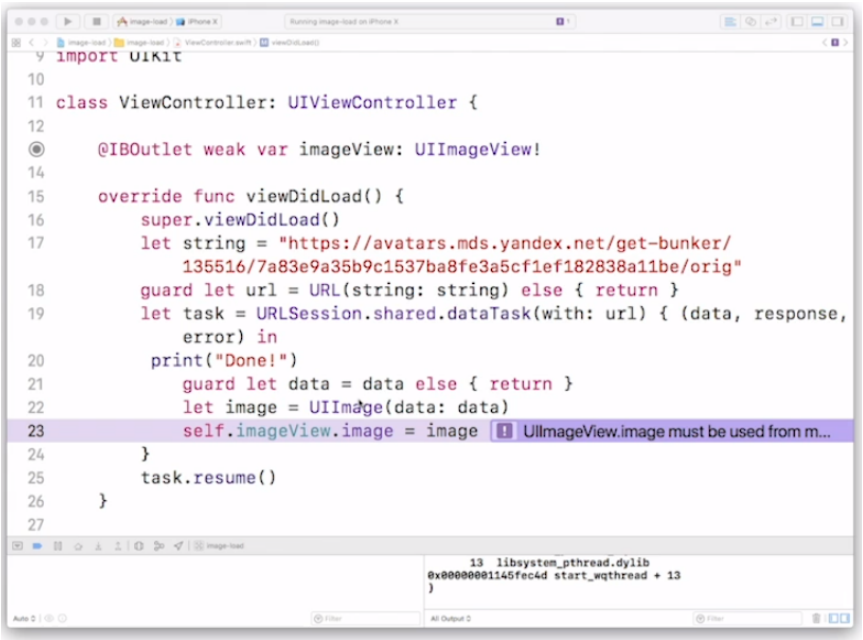
Теперь мы можем установить полученную картинку в imageView:

```
self.imageView.image = image
```

Перезапустим проект и увидим, что задача успешно завершилась, а картинка на экране не появилась.

Попробуем свернуть приложение и развернуть его. Картинка появилась.


В чем же проблема?
Вернемся в Xcode и увидим предупреждение, говорящее о том, что взаимодействовать с UI необходимо только из главного потока.




Переключиться на главный поток мы можем, например, с помощью GCD. Перенесём код работы с картинкой на главный поток.

```
DispatchQueue.main.async {
    let image = UIImage(data: data)
    self.imageView.image = image
}
```

Перезапустим проект и увидим, что картинка появляется сразу после загрузки.


В этом разделе мы научились загружать картинки из сети и отображать их в нашем приложении, а в следующем вы узнаете о том, как обрабатывать ошибки.

## 2.3. Получение данных от сервера

**Часть 1**

В прошлом разделе мы научились загружать картинки из сети, игнорируя возможные ошибки. Однако не всегда в наших приложениях все идет так, как ожидается.
Особенно много ошибок может возникать при работе с сетью. Мы уже видели диапазоны кодов ответа сервера на запрос, и немалая часть из них была связана с какой-нибудь ошибкой, произошедшей в процессе выполнения запроса.

```
200..<300 - success
300..<400 - redirection
400..<500 - client errors
500+ - server errors
```

Как же обрабатывать ошибки? Обнаружить произошедшую ошибку мы можем, используя параметры типов Error и Response в completionBlock-е выполнения задачи.

```
let task = URLSession.shared.dataTask(with: url){
    data, response, error in

    print("Done!")

    DispatchQueue.main.async {
        self.label.text = "ok"
    }
}

task.resume()
```

Важно понимать, что ошибки, возникающие при работе с сетью, могут быть разных видов.
Первый - это ошибки соединения. Мы можем их детектировать, используя объект типа Error.

```
if let error = error{
    print(error.localizedDescription)
    return
}
```

Этот объект содержит различные поля, например, localizedDescription, который мы можем обработать. Error представляет собой ошибки, вызванные библиотекой URLSession, то есть они вызваны клиентским кодом (когда внутри URLSession что-то пошло не так - например, нет соединения с интернетом).
Второй вид ошибок - это ошибки приложения, то есть те самые ошибки, которые отправляет сервер в ответ на наш запрос. В этом случаем объект типа Error будет пустым, а объект типа Response будет содержать статус-код, который мы можем правильно обработать. Например, мы могли бы перенаправить пользователя на экран авторизации, если получили в ответе код 401.

```
if let response = response as? HTTPURLResponse {
    switch response.statusCode {
        case 200..<300: break
        default:
            print("Status: \(response.statusCode)")
    }
}
```

Только что мы поговорили про обработку ошибок, возникающих при работе с сетью. Далее мы научимся получать от сервера данные в формате JSON.

**Часть 2**

Итак, что же, если одной картинки нам недостаточно и мы хотим получить данные в более сложном формате? Наиболее распространенный формат для передачи данных между клиентом и сервером - это JSON.
JSON - это человекочитаемый текст, состоящий из пар "ключ-значение"(то есть словарей) и массивов.

```
{
    "firstName": "John",
    "lastName": "Smith",
    "birthday": "23-01-1992",
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "postalCode": "10021-3100"
},
                                        "phoneNumbers":[{
                                                "type": "home",
                                                "number": "212 555-1234"
                                            },{
                                                "type": "office",
                                                "number": "646 555-4567"
                                            }
                                        ],
                                        "children": []
                                    }
```

И нам нужно научиться извлекать из этого текста объекты.
С появлением Swift 4 для поддержки кодирования и декодирования в iOS используются два протокола: это Encodable (для кодирования) и Decodable (для декодирования). Также есть протокол, объединяющий эти функции, и называется он Codable.

```
public protocol Encodable { ... }

public protocol Decodable { ... }

public typealias Codable = Decodable  Encodable
```

Не только классы (class), но и структуры (struct) и перечисления (enum) могут реализовать эти протоколы.
А теперь разберемся подробнее, для чего все-таки нужны эти протоколы.
Протокол Encodable говорит, что тип, реализующий данный протокол, способен кодировать себя во внешнюю репрезентацию. Encodable содержит всего одну функцию: `encode(to:)`, которая позволяет закодировать объект к заданному кодировщику.

```
public protocol Encodable {
    public func encode(to encoder: Encoder) throws
}
```

Протокол Decodable говорит нам, что тип, его реализующий, может декодировать себя из внешней репрезентации. И этот протокол также содержит всего один метод: `init(from:)`, который создает новый экземпляр данного типа, декодируя из заданного декодировщика.

```
public protocol Decodable {
    public init(from decoder: Decoder) throws
}
```

Протокол Codable же объединяет в себе функционал двух протоколов и говорит нам о том, что реализующий его тип может как преобразовывать себя во внешнюю репрезентацию, так и быть созданным из нее, и, соотвественно, содержит обе функции: `encode(to:)` и `init(from:)`.
Таким образом, чтобы кодировать и декодировать наш собственный объект (пользовательский тип), нам нужно сделать его Codable.
Самый простой способ реализовать протокол Codable - это объявить свойства объекта с использованием тех типов данных, которые уже реализуют протокол Codable. Такими типами данных являются: `String, Int, Double, Date, Data, URL и Bool`. Словари и массивы являются Codable объектами, если содержат Codable типы.
А теперь посмотрим, как использовать протокол Codable для кодирования и декодирования пользовательских типов данных.
Как мы уже узнали, самый распространенный формат для передачи данных между клиентом и сервером - JSON. В iOS мы можем воспользоваться объектом JSONEncoder для преобразования Codable объекта в тип Data.

```
open class JSONEncoder { ... }
```

```
struct Message: Codable {
    let title: String
    let date: Date
}
let message = Message(title: "My message date: Date())
let jsonData: Data = try JSONEncoder().encode(message)
```

Метод encode(_:) JSONEncoder'a вернет нам кодированную в JSON репрезентацию Codable-объекта. Всего пара строк - и готово. Выглядит просто, не правда ли?
Научиться декодировать Codable объекты, также просто, как и кодировать.
В противовес JSONEncoder'у существует класс JSONDecoder для декодирования данных в формате JSON в наш пользовательский Codable тип. Функция JSONDecoder'a decode(_:from:) вернет нам значение указанного Codable типа, декодированное из JSON-объекта. Несложно выглядит, правда?
Далее мы посмотрим, как все это работает на практике. Мы сможем получить список членов экипажа, которые сейчас находятся на космической станции.
А теперь, когда мы умеем получать объект из JSON, мы можем получить из сети более сложные данные, нежели картинка.
Получать данные будем из открытого API -

```
http://api.open-notify.org/astros.json.
```

Url возвращает краткую информацию о людях, которые сейчас находятся в космосе. Ответ представлен в виде JSON следующего формата:

```
{
    "message ": "success",
    "number": NUMBER_OF_PEOPLE_IN_SPACE,
    "people": [
        {
            "name": NAME,
            "craft": SPACECRAFT_NAME
        },
    ...
    ]
}
```

Нас интересуют два поля: поле number, в котором передается количество людей, находящихся сейчас в космосе, и поле people, которое является массивом и содержит имена людей и соотвествующие им названия космических станций.
Посмотрим ответ сервера на GET запрос для получения списка людей, находящихся в космосе. Перейдём по ссылке:

```
http://api.open-notify.org/astros.json
```

и увидим ответ сервера:

```
{
    "people": [
        {
            "craft": "ISS",
            "name": "Alexey Ovchinin"
        },
        {
            "craft": "ISS",
            "name": "Nick Hague"
        },
        {
            "craft": "ISS",
            "name": "Christina Koch"
        }
    ],
    "number": 3,
    "message": "success"
}
```

В самом простом варианте мы могли бы попробовать получить только количество людей, игнорируя вложенные объекты. Эта информация представлена в поле "number". Попробуем это сделать.
Напишем тип данных, который будет отражать структуру получаемого от сервера ответа. Назовем ее `SpaceInfo`. Для начала получим информацию только о количестве людей в космосе, игнорируя поле "people". Так как мы используем только те типы данных, которые уже соответствуют протоколу Decodable (в нашем случае Int и String), то созданная структура автоматически будет соответствовать протоколу Decodable.
Нам осталось только написать GET запрос для получения данных с сервера. Сохраняем адрес в строку, получаем из строки объект класса URL, и далее пишем уже известную нам конструкцию для создания DataTask'а для совершения GET запроса.

```
imporf Foundation
import PlaygroundSupport

PlaygroundPage.current.needsIndefiniteExecution = true

struct SpaceInfo: Decodable {
    let message: String
    let number: Int
}

func load() {
    let stringUrl = "http://api.open-notify.org/astros.json"
```

```
    guard let url = URL(string: stringUrl) else { return }
    let task = URLSession.shared.dataTask(with: url) { (data, response,
        error) in

    }
}
```

В целях демонстрации мы будем игнорировать сценарии, когда у нас что-то не получилось.

```
guard error == nil else { return }
guard let data = data else { return }
```

Теперь нам необходимо декодировать ответ, полученный от сервера. Он находится в поле data.
Для декодирования JSON, мы будем использовать специальный объект JSONDecoder, о котором я рассказывала ранее.

```
let spaceInfo = try? JSONDecoder().decode(SpaceInfo.self, from: data)
```

Написанная строка кода позволяет получить объект указанного типа из сериализованного JSON. Так как декодирование может закончиться ошибкой, например, когда указанный тип не соответствует формату JSON, обернем конструкцию в guard и напечатаем в консоль об ошибке в случае неудачи:

```
print("Error: can't parse SpaceInfo")
```

Теперь у нас уже есть успешно декодированный объект, поэтому просто выведем полученную информацию в консоль.

```
print("There are currently \(astrosObject.number) humans in space")
```

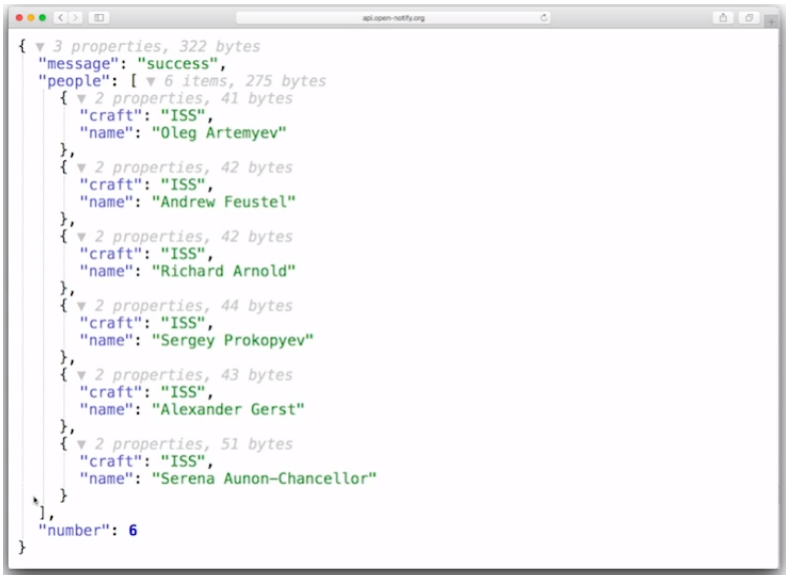
Напишем resume(), чтобы созданная задача запустилась:

```
task.resume()
```

и вызовем функцию load.
Выполним написанный код.
Код выполнился, и в консоли мы видим сообщение:

```
"There are currently 6 humans in space"
```

Мы смогли получить количество людей, находящихся в космосе.
А что, если теперь мы хотим получить список космических станций? Снова посмотрим на JSON в браузере.




Космические станции отдаются внутри списка поля people. Поле "people"содержит массив объектов с полями "name"и "craft". Модифицируем написанную ранее структуру, чтобы получить данные о космических станциях. Добавим поле "people". Оно будет массивом объектов некоторого типа, назовем его Astronaut.

```
let people: [Astronaut]
```

Теперь определим структуру для типа Astronaut. Как мы видели из структуры JSON, вложенный объект имеет два строковых поля - name и craft. Важно, чтобы новая структура соответствовала протоколу Decodable, иначе структура верхнего уровня SpaceInfo перестанет соответствовать этому протоколу. Так как мы используем внутри новой структуры поля типов String, она автоматически реализует протокол Decodable.

```
struct Astronaut: Decodable {
    let craft: String
    let name: String
}
```

А теперь, когда мы завершили описание структуры Astronaut, мы можем напечатать список космических станций в консоль. Добавим еще один print для вывода информации о космических станциях. Сначала соберем все уникальные названия космических станций, например, с помощью сета.

```
let uniqueSpacecrafts = Set(spaceInfo.people.map({ $0.craft }))
```

А затем выведем информацию, используя print.

```
print("Spacecrafts: \(uniqueSpacecrafts.joined(separator: " ,")) ")
```

Выполним наш новый код и увидим в консоли, что мы получили правильный результат:

```
There are currently 6 humans in space
Spacecpafts: ISS
```

Вот так просто можно работать с форматом JSON для получения различной информации от сервера. В следующем разделе я расскажу о том, чем отличаются пользовательские данные от анонимных и для чего нужна авторизация в приложении.

## 2.4. Пользовательские данные и авторизация в приложении

Далее мы разработаем приложение, которое позволяет получить список фотографий с вашего Яндекс.Диска. Вы уже научились получать информацию с помощью GET запросов и парсить JSON. Однако, чтобы получить доступ, например, к вашим личным фотографиям на Яндекс.Диске, этого недостаточно. И это хорошо, иначе бы каждый разработчик, умеющий делать GET запросы к API, смог бы получить любые данные из сети.

Для получения доступа к пользовательским данным необходимо пройти авторизацию.

Каждый API по-разному обеспечивает процесс авторизации, но сегодня мы научимся работать с авторизацией независимо от ее реализации в API. Важно понимать разницу между анонимными и пользовательскими данными.


Для доступа к анонимным данным обычно не требуется авторизация. То есть такие данные может получить любой желающий. Например, анонимными данными является тот список астронавтов, который мы получали в прошлом видео. А вот пользовательские данные, к примеру, список фотографий в облаке или любимые фильмы на кинопоиске - это персональные пользовательские данные. Иногда может быть так, что API предоставляет только пользовательские данные (и, соответственно, не работает без авторизации).

И если процесс получения анонимных данных достаточно стандартен - обычный GET запрос, то с получением доступа к пользовательским данным все немного сложнее - нам нужно научиться подтверждать, что у нас есть права на совершение запрашиваемого действия.

Как сервер может понять, что доступ запрашивает авторизованный пользователь? Он делает это с помощью аутентификации.

Посмотрим на приложение, которое мы сделаем далее.

Приложение умеет показывать окно авторизации и список загруженных с вашего Яндекс.Диска фотографий.

Для авторизации в приложении вводим свой логин и пароль вашего аккаунта в Яндексе и затем разрешаем приложению доступ к Яндекс.Диску.

На следующем экране отображается список фотографий, полученных с Яндекс.Диска.

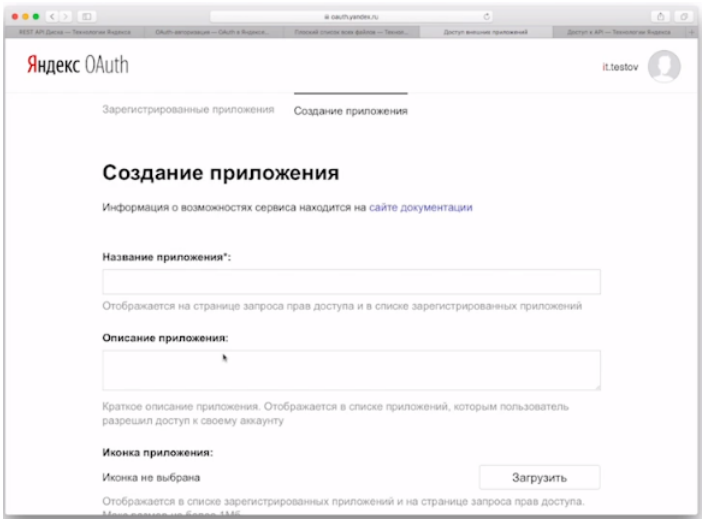
Перейдём к реализации приложения.
Как я уже говорила, мы будем использовать API Яндекс.Диска. Для начала посмотрим документацию. Для того, чтобы начать работать с API диска, необходимо получить OAuth token.
Для авторизации через Яндекс.Паспорт используется протокол OAuth 2.0. После авторизации мы сможем работать с сервисами Яндекса от имени авторизованного пользователя.
Посмотрим на один из методов, который предоставляет Яндекс.Диск.

```
https://cloud-api.yandex.net/v1/disk/resources/files
    ? [limit=<количество файлов в списке>]     & [media_type=<тип запрашиваемых
файлов>]     & [offset=<смещение относительно начала списка>]     &
[fields=<свойства, которые нужно включить в ответ>]     & [preview_size=<размер
превью>]     & [preview_crop=<признак обрезки превью>]
```

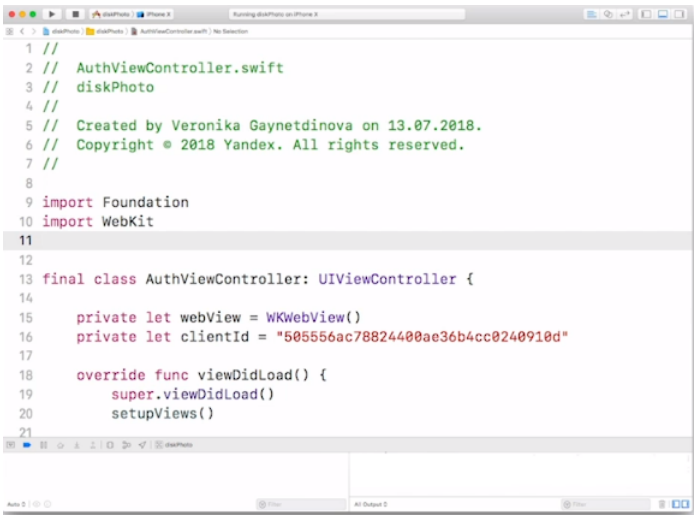
В нашем демо-приложении мы воспользуемся запросом на получение плоского списка всех файлов. Данный метод возвращает список всех файлов в алфавитном порядке, не учитывая структуру каталогов.
Как видно из описания, GET запрос может принимать несколько параметров.
Чтобы получить только изображения, хранящиеся на диске нам нужно будет указать параметр "media_type=image": И если мы сейчас попробуем выполнить этот запрос, то получим ошибку сервера 401, что означает "пользователь не авторизован".
Если в поле message у вас отображается текст в неправильной кодировке, вы можете изменить её в настройках браузера.

```
{
    "message":"Не авторизован.",
    "description":"Unauthorized",
    "error":"UnauthorizedError"
}
```

Для успешного выполнения запроса необходимо в заголовках отправить пользовательский токен с ключом "Authorization".
Получение токена подробно описано в документации. Первое, что нужно сделать - Зарегистрировать приложение на Яндекс.OAuth. Сделать это можно на странице создания приложений.
Нажимаем кнопку "Зарегистрировать новое приложение".
Для каждого приложения обязательно указать только название и доступы. Но чем больше информации о приложении вы предоставите, тем легче будет понять пользователям, кому именно они предоставляют доступ к своему аккаунту.
Назовем наше приложение.
Укажем доступы к Яндекс.Диску.




И установим callback URL, чтобы в дальнейшем перехватить отправленный в ответ токен.
Теперь, когда мы успешно зарегистрировали приложение, мы можм приступить к реализации.
В прошлом видео мы успешно прошли регистрацию приложения, теперь мы можем реализовать получение токена. Чтобы это сделать, необходимо открыть страницу Яндекс.OAuth в приложении. Сделать это мы можем с помощью элемента web-view.
WebView - это специальный объект, который позволяет встраивать веб-содержимое в ваши iOS приложения. Приложение должно направить на Яндекс.OAuth по следующему адресу:



```
1  //
2  //  AuthViewController.swift
3  //  diskPhoto
4  //
5  //  Created by Veronika Gaynetdinova on 13.07.2018.
6  //  Copyright © 2018 Yandex. All rights reserved.
7  //
8
9  import Foundation
10 import WebKit
11
12
13 final class AuthViewController: UIViewController {
14
15     private let webView = WKWebView()
16     private let clientId = "505556ac78824400ae36b4cc0240910d"
17
18     override func viewDidLoad() {
19         super.viewDidLoad()
20         setupViews()
21
```

В подготовленном проекте у нас уже есть контроллер с webView, который будет отвечать за авторизацию. Мы будем показывать его, когда необходимо получить пользовательский токен, и после получения токена будем отдавать его делегату.
Напишем для этого протокол. В протоколе будет один метод, передающий полученный токен:

```
protocol AuthViewControllerDelegate: class {}
func handleTokenChanged(token: String)
```

Далее создадим свойство delegate внутри контроллера авторизации:

```
weak var delegate: AuthViewControllerDelegate?
```

Вернемся в наш основной контроллер, в котором мы будем использовать полученные токены.
В целях демонстрации мы будем хранить токен прямо в главном контроллере в строковом поле token.

```
private var token: String =
```

И в случае, если токен пустой, будем запрашивать новый. Напишем это в функцию updateData:

```
guard !token.isEmpty else {}
```

Чтобы получить новый токен, нам нужно открыть контроллер с webView:

```
guard !token.isEmpty else {
let requestTokenViewController = AuthViewController()
present(requestTokenViewController, animated: false, completion: nil)
}
```

После получения токена внутри контроллера с авторизацией, мы будем пробрасывать полученный токен в метод делегата, поэтому установим свойство delegate:

```
guard !token.isEmpty else {
let requestTokenViewController = AuthViewController()
requestTokenViewController.delegate = self
present(requestTokenViewController, animated: false, completion: nil)
return
}
```

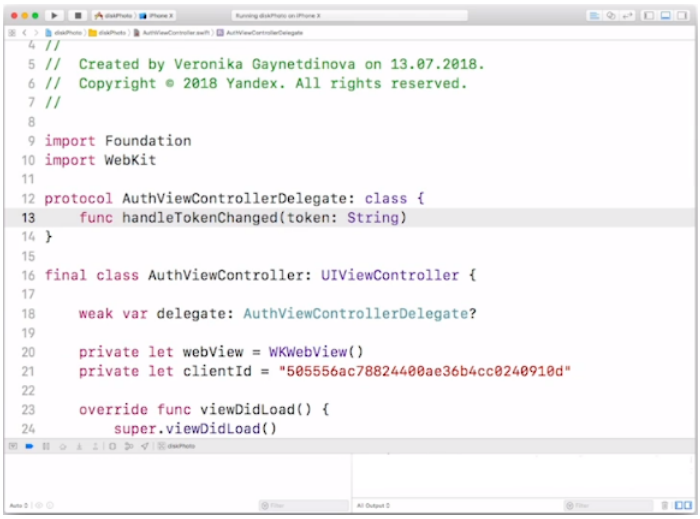
И затем реализуем протокол делегирования:

```
extension ViewController: AuthViewControllerDelegate {
    func handleTokenChanged(token: String) {}
}
```

Сохраним токен и обновим данные:

```
extension ViewController: AuthViewControllerDelegate {
    func handleTokenChanged(token: String) {
        self.token = token
        updateData()
}
}
```

Переходим обратно в контроллер авторизации.




Как я уже сказала, webView уже размещено на этом контроллере. А еще у нас здесь есть clientId, который мы скопировали из браузера со страницы зарегистрированного приложения.
Напишем запрос для webView, направляющий по адресу для получения токена:

```
private var tokenGetRequest: URLRequest? { }
```

Так как нам потребуется указать параметры в адресе (`response_type` и `client_id`), воспользуемся для создания URL специальным классом - URLComponents:

```
private var tokenGetRequest: URLRequest? {
    guard var urlComponents = URLComponents(string: "https://oauth.yandex.ru/authorize")
else { return nil }
}
```

Установим необходимые параметры с помощью поля `queryItems` у объекта URLComponents:

```
urlComponents.queryItems = [ response_type, client_id ]
```

Таким образом мы указали, что в результирующем URL должно быть два параметра: первый - response_type со значением token и второй - client_id, куда мы установили идентификатор зарегистрированного приложения.
Теперь мы можем сформировать URL:

```
guard let url = urlComponents.url else { return nil }
```

А на основе полученного URL мы можем создать объект класса URLRequest, чтобы в дальнейшем использовать его в webView:

```
return URLRequest(url: url)
```

Теперь, когда у нас есть правильный URLRequest, мы можем запросить у пользователя доступ к его Яндекс.Диску. Для этого во viewDidLoad контроллера, отвечающего за авторизацию, мы можем установить этот реквест в webView:

```
guard let request = tokenGetRequest else { return }
webView.load(request)
```

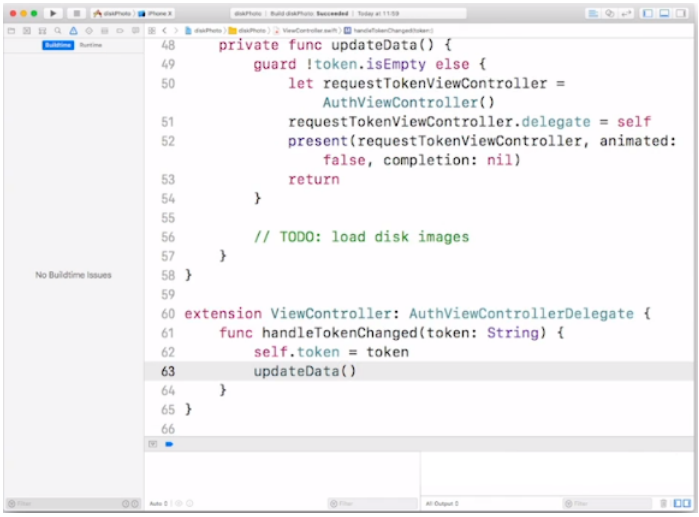
После успешного получения доступов в редиректе нам будет отправлен токен. Чтобы перехватить редирект, мы можем реализовать протокол делегата для webView.
В подготовленном проекте это уже сделано, поэтому я просто установлю свойство navigationDelegate у webView:

```
webView.navigationDelegate = self
```

И внутри реализованного метода для navigationDelegate я добавлю пробрасывание токена делегату:

```
if let token = token {}
```

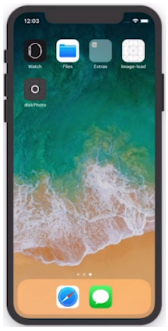
Уже сейчас мы можем запустить приложение и проверить работу авторизации. Возвращаемся в основной контроллер приложения:



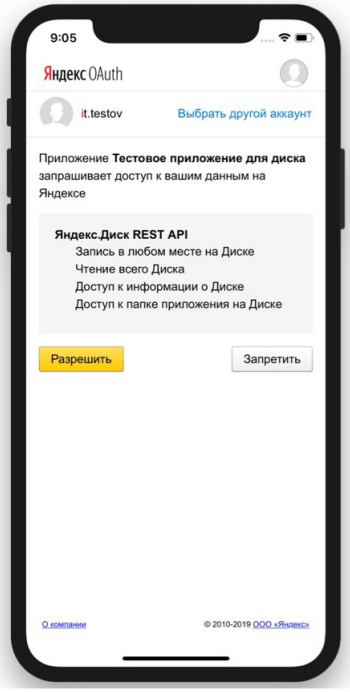

Чтобы увидеть полученный токен, добавим print в метод получения токена от контроллера авторизации:

```
print("New token: \(token)")
```

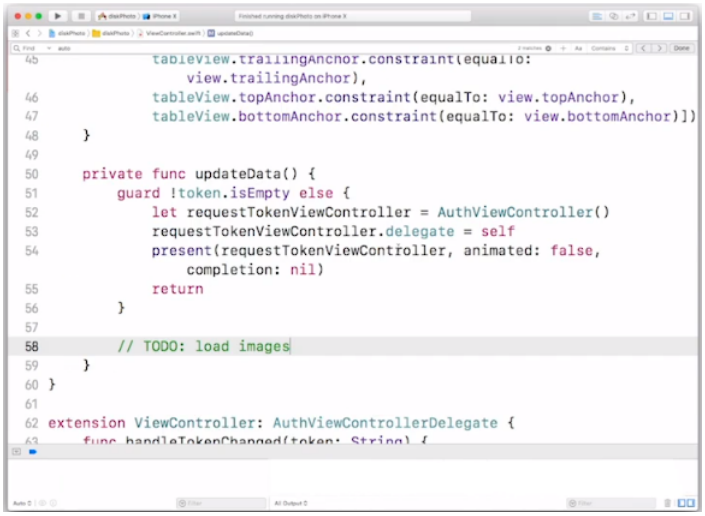
Запускаем приложение.



Разрешаем приложению работать со своим Яндекс.Диском.




И видим: в консоль вывелась строка с полученным токеном.
Теперь мы можем приступить к запросу данных из API Яндекс.Диска.
Вернемся в документацию и посмотрим описание метода для получения плоского списка файлов. Плоский список не учитывает структуру каталогов, поэтому данный метод очень удобен, если нужно получить с диска все файлы какого-то определенного формата. Запрос списка всех файлов следует отправлять с помощью метода GET.
Скопируем URL из документации и напишем в нашем iOS-проекте GET-запрос для этого ресурса. Вернемся в функцию UpdateData:



Т.к. мы будем использовать пареметр в запросе, для более удобного формирования URL снова воспользуемся классом URLComponents:

```
var components = URLComponents(string:
"https://cloud-api.yandex.net/v1/disk/resources/files")
```

Установим параметр для указания типа запрашиваемых файлов с диска:

```
components?.queryItems = [URLQueryItem(name: "media_type", value: "image")]
```

Так мы указали, что хотим получать только изображения. Теперь мы можем получить объект класса URL:

```
guard let url = components?.url else { return }
```

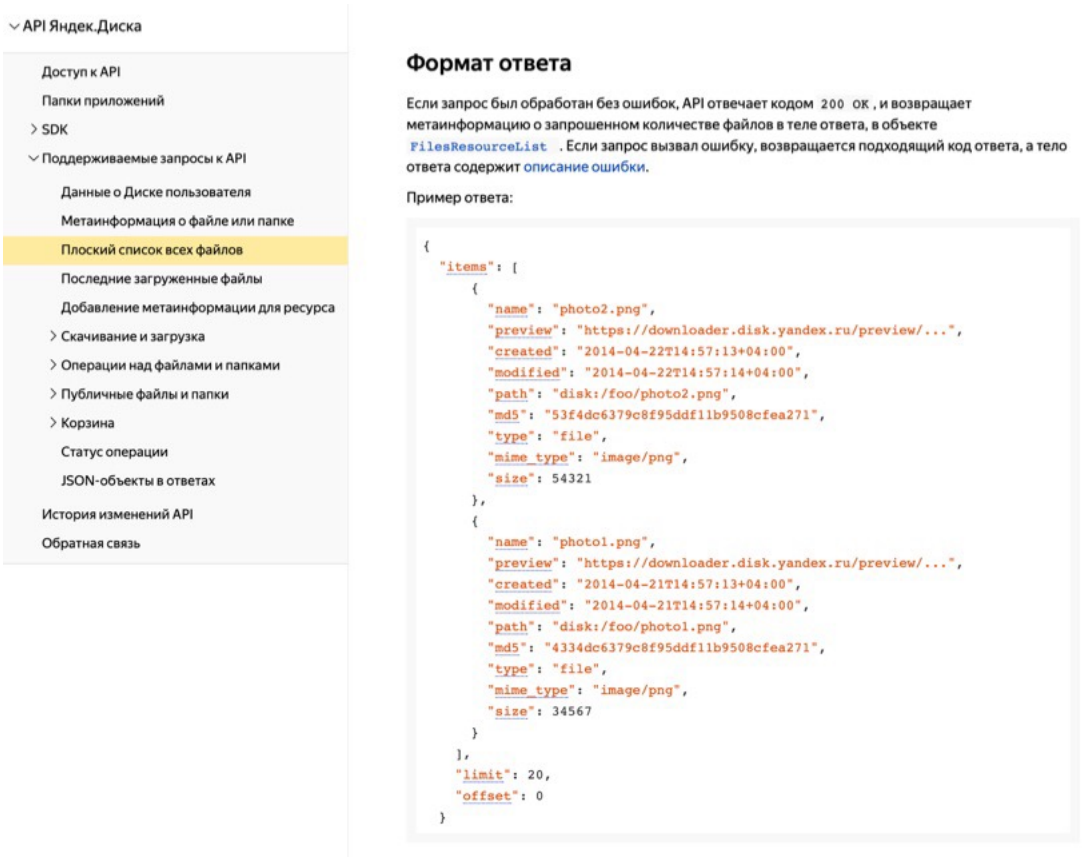
Если бы нам не нужно было указывать заголовки для совершаемого запроса, мы могли бы воспользоваться этим URL и сразу создать data task. Однако нам необходимо указать заголовок Authorization, в котором мы должны отправить токен, чтобы API диска разрешило нам получить пользовательские данные.
Поэтому создаем объект класса URLRequest и устанавливаем необходимый нам заголовок с токеном:

```
var request = URLRequest(url: url)
request.setValue("OAuth \(token)", forHTTPHeaderField: "Authorization")
```

Теперь мы можем воспользоваться полученным запросом для создания data task:

```
let task = URLSession.shared.dataTask(with: request) { [weak self]
(data, response, error) in

}
task.resume()
```

А сейчас нужно научиться извлекать объекты из ответа сервера. Посмотрим еще раз в документацию, чтобы изучить фотмат ответа.
Если запрос был обработан без ошибок, API отвечает кодом 200 и возвращает в теле ответа информацию о запрашиваемых объектах. Пример такого ответа мы можем увидеть в документации:




Напишем тип данных, повторяющий структуру ответа сервера на успешный запрос. Сначала опишем структуру файла, который возвращается внутри массива items:

```
import Foundation
struct DiskFile: Codable { }
```

Т.к. нам не нужны все возвращаемые поля, мы могли бы сохранять только 3 из них: имя файла, ссылку на превью и размер.

```
struct DiskFile: Codable {
    let name: String?
    let preview: String?
    let size: Int64?
}
```

Теперь опишем структуру верхнего уровня, которая содержит массив items. Назовем ее DiskResponse.

```
struct DiskResponse: Codable{
    let items: [DiskFile]?
}
```

Остальные поля нам сейчас не важны, поэтому не будем добавлять их в структуру.
Вернемся к запросу, который был написан ранее для получения данных с Яндекс.Диска. Напишем код для получения объекта из ответа сервера.

```
guard let sself = self, let data = data else { return }
guard let newFiles = try?
    JSONDecoder().decode(DiskResponse.self, from: data) else { return }
```

Добавим print для вывода информации о количестве полученных объектов.

```
print("Received: \(newFiles.items?.count ?? 0) files")
```

Запустим проект и проверим, что все работает. Сейчас мы увидели, что в консоли напечаталось количество полученных файлов. Следующая задача - научиться выводить информацию о полученых файлах на экран.
Для начала сохраним информацию о файлах на диске в переменную контроллера. Объявляем переменную:

```
private var FilesData: DiskResponse?
```

И устанавливаем ее значение после обработки ответа сервера:

```
self.filesData = newFiles
```

Чтобы отобразить информацию о файлах на экране, воспользуемся таблицей. В подготовленном проекте UITableView уже размещено на главном контроллере, поэтому нам осталось только наполнить таблицу данными. Для этого реализуем протокол DataSource для таблицы. Напомню, что обязательно нужно реализовать 2 метода: один, который возввращает ячейку для таблицы, и другой, который возвращает количество ячеек. Напишем реализацию для этих функций. Количество ячеек будет соответствовать количеству полученных файлов.
Реализуем второй метод. Для начала будем использовать системные ячейки. В качестве текста установим в системную ячейку название файла. Сначала получим текущий файл из модели. И затем установим название файла в textLabel ячейки.

```
extension ViewController: UITableViewDataSource {

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
-> UITableViewCell {
        let cell = UITableViewCell()
        guard let items = filesData?.items, items.count > indexPath.row else {
            return cell
        }
        let currentFile = items[indexPath.row]
        cell.textLabel?.text = currentFile.name
        return cell
    }

    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int)
-> Int {
        return filesData?.items?.count ?? 0
    }
}
```

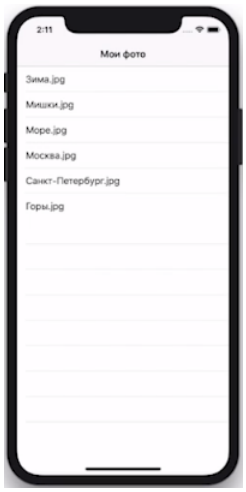
Установим свойство DataSource у таблицы.

```
tableView.dataSource = self
```

Также важно добавить обновление таблицы после получения новых данных о файлах. Вернемся в функцию получения данных из API диска и добавим необходимые строки кода:

```
DispatchQueue.main.async { [weak self] in
    self?.tableView.reloadData()
}
```

Запустим проект. Мы видим, что запрос снова выполнился успешно.




А в таблице появился список, состоящий из названий полученных файлов.
Теперь осталось отобразить превью загруженных картинок и размер файлов. Сделаем это с помощью кастомной ячейки. Сейчас мы не будем подробно рассматривать создание UI компонентов: в подготовленном проекте нужная нам ячейка уже создана. Поэтому зарегистрируем новую ячейку в таблице. Вынесем идентификатор для ячейки в константу, чтобы затем использовать ее в нескольких местах кода:

```
private let fileCellIdentifier = "FileTableViewCell"
```

И затем зарегистрируем новый класс ячеек в таблице:

```
tableView.register(FileTableViewCell.self, for CellReuseIdentifier:
fileCellIdentifier)
```

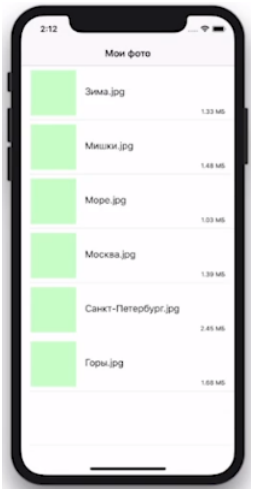
Вернемся в функцию создания ячейки и заменим строку инициализации:

```
let cell = tableView.dequeueReusableCell(withIdentifier: fileCellIdentifier, for:
indexPath
```

И затем передадим данные о файле в ячейку:

```
if let fileCell = cell as? FileTableViewCell {
fileCell.bindModel(currentFile)
}
```

Если сейчас мы запустим проект, то увидим, что картинки не будут отображены в таблице. Проверим это.



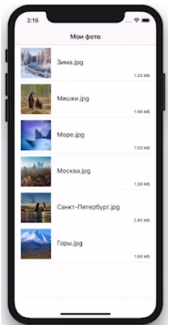

Дело в том, что сама ячейка не умеет загружать картинки по URL: она делегирует это другому объекту. Установим свойство delegate для ячейки:

```
fileCell.delegate = self
```

А затем реализуем асинхронную загрузку картинок для ячеек. Напишем расширение для viewController с реализацией функции делегата ячейки. Необходимо реализовать только одну функцию, она называется loadImage. Т.к. каринки хранятся на диске, для их загрузки также потребуется отправлять токен авторизации. Поэтому снова создаем запрос, в котором указываем заголовок с токеном. Далее пишем уже знакомый нам код для загрузки картинки из сети. И возвращаем в completion block полученную картинку, не забыв переключиться на главную очередь.

```
extension ViewController: FileTableViewCellDelegate {

    func loadImage(stringUrl: String, completion: @escaping ((UIImage?) -> Void))
{
        guard let url = URL(string: stringUrl) else { return }
        var request = URLRequest(url: url)
        request.setValue("OAuth \(token)", forHTTPHeaderField: "Authorization")

        let task = URLSession.shared.dataTask(with: request) { (data, response,
error) in
            guard let data = data else { return }

            DispatchQueue.main.async {
                completion(UIImage(data: data))
            }
        }
        task.resume()
    }
}
```

Снова запустим проект и увидим, что картинки в ячейках теперь отображаются корректно.




В последних трех разделах я рассказывала о получении доступа к пользовательским данным, и мы разработали приложение дл получения доступа к фотографиям с Яндекс.Диска. В следующем разделе я расскажу о том, как отправлять данные с устройства для сохранения их на сервер.

## 2.5. Отправка данных на сервер

Вы уже умеете получать данные из API, но что если нам нужно сохранить какие-то данные на сервер? Например, загрузить новую фотографию на Яндекс.Диск или поменять название уже существующей? Вспомним про HTTP-методы, о которых мы говорили в самом начале.

```
GET - read
PUT - update
POST - create
DELETE - delete
```

Для получения данных из API мы пользовались GET запросами, а для сохранения новых данных или изменения существующих будем использовать метод POST.
Как правило, POST запрос отличается от GET и DELETE запросов тем, что у него есть тело запроса. POST запрос выглядит следующим образом:

```
POST http://localhost/api/message
Host: localhost
Content-Type: application/json

{ "message": "hello!" }
```

Указывается метод, то есть POST, ресурс, к которому происходит обращение, проставляются необходимые заголовки и передается тело запроса. Телом, например, может быть JSON.
Таким образом, для того, чтобы сделать POST-запрос, одного url нам уже недостаточно, нам необходимо также передать тело запроса. Для этих целей мы будем пользоваться классом URLRequest. Использование этого класса мы уже затрагивали ранее.
Таким образом, для совершения POST запроса, мы будем придерживаться следующего плана:

1. Сформируем URL (URL).

2. Создадим HTTP-запрос (то есть URLRequest).

3. Создадим задачу (URLSessionTask).

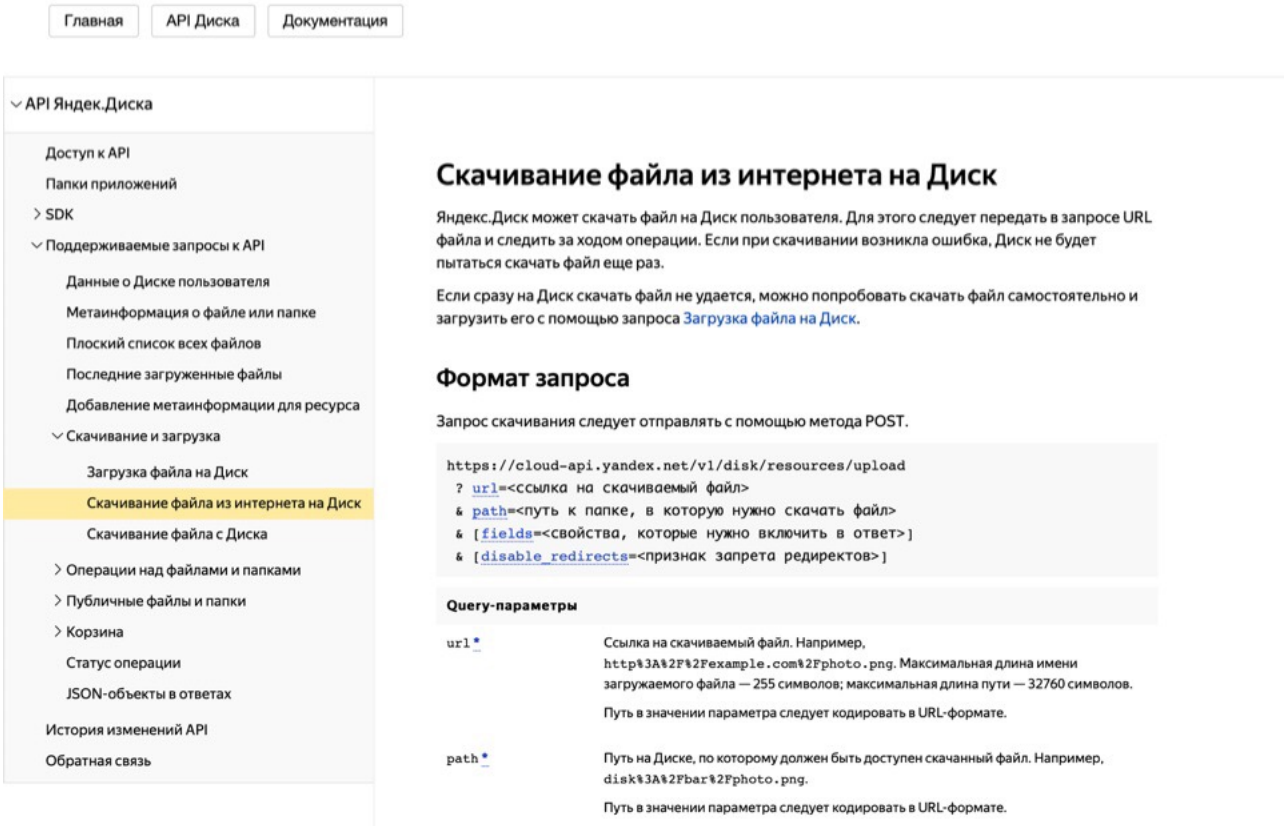
Создать URLRequest для POST запроса мы можем, например, следующим образом:

```
var request = URLRequest(url: url)
request.httpMethod = "POST"
request.setValue("application/json",
forHTTPHeaderField: "Content-Type")
request.httpBody = try! JSONSerialization.data(
withJSONObject: ["message": "Hello"])
```

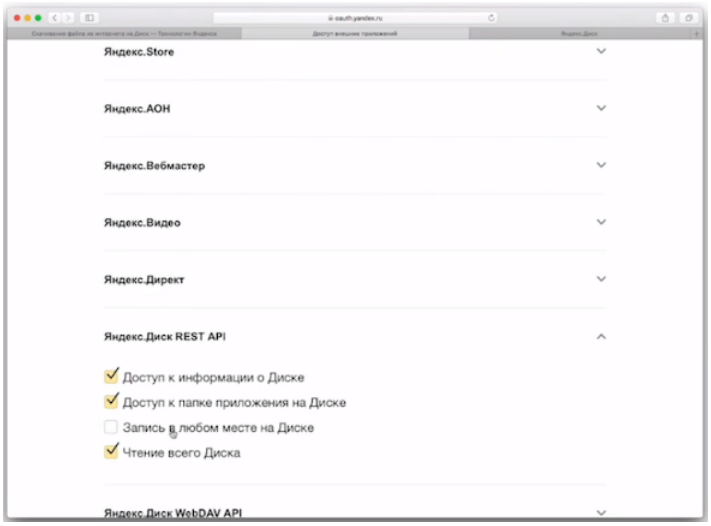
Инициализируем URLRequest и затем указываем необходимый нам HTTP метод, то есть POST, а также устанавливаем тело запроса. В данном случае используется JSON. Далее полученный запрос мы можем использовать для создания dataTask-ов.

```
let url = URL(string: "http://localhost:8080/message")!
var request = URLRequest(url: url)

let task = session.dataTask(with: request) {
    data, response, error in

if error != nil {
print("ok")
}
}
task.resume()
```

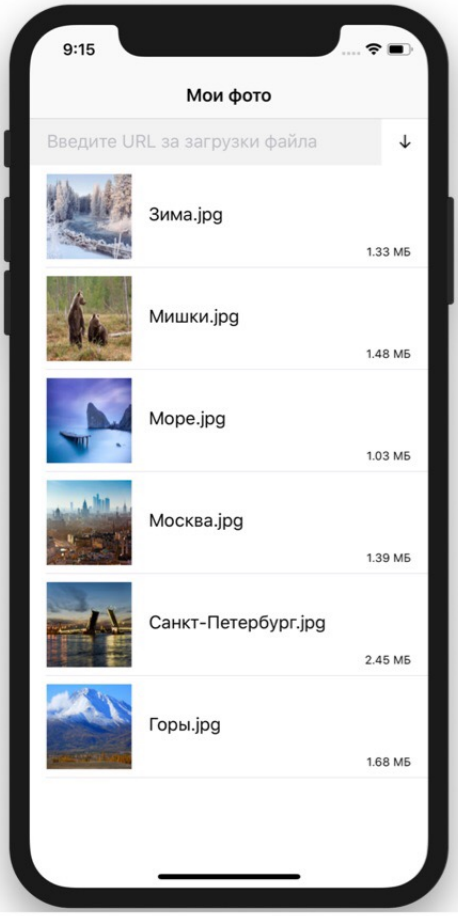
А теперь попробуем выполнить POST запрос, используя уже знакомое нам API Яндекс.Диска. Воспользуемся проектом, который был написан нами ранее. В этом проекте у нас уже есть авторизация и сохранение токена в контроллер.
Обратимся к документации API Яндекс.Диска. Яндекс.Диск может скачать файл из интернета на Диск пользователя. Для этого необходимо передать в запросе URL файла и далее следить за статусом операции.
Если запрос завершится кодом в диапазоне от 200 до 300, это означает, что файл успешно загружен на диск.
Посмотрим на формат запроса и увидим, что есть два обязательных параметра.




Это url, то есть путь к файлу, который мы хотим скачать, и path - путь к папке, куда мы хотим сохранить новый файл. Этот метод API, как и предыдущий, требует, чтобы пользователь был авторизован. Но, кроме того, пользователь должен разрешить операцию записи в любом месте на своем Диске.
Для этого откроем страницу с информацией о зарегистрированном приложении и поменяем настройки.
Нажмем кнопку "редактировать":
Перейдем к доступам для Яндекс.Диска и отметим недостающий пункт.




Сохраним настройки. Теперь, когда мы перезапустим iOS-приложение, во время авторизации произойдет повторный запрос разрешений для работы с Диском.
Вернемся в наше приложение.
UI элементы для реализации нового функционала приложения уже размещены на контроллере. Посмотрим на них. Для этого запустим приложение.
Добавилось два элемента: текстовое поле для ввода URL и кнопка для скачивания указанного файла на диск.




Нам остается только реализовать функционал этой кнопки, написав запрос для скачивания файла на Диск пользователя.
Переходим к функции, в которой нужно реализовать загрузку файла. Эта функция вызывается сразу после нажатия на кнопку. Для получения url воспользуемся классом URLComponents, так как нам необходимо установить параметры.
Создаем объект и указываем обязательные для данного запроса параметры. В качестве названий загружаемых файлов будем использовать слово "item":

```
var components = URLComponents(string: "https:/cloud-api.yandex.net/v1/disk
/resources/upload")
components?.queryItems = [
    URLQueryItem(name: "url", value: fileUrl)
    URLQueryItem(name: "path", value: "item"
])
```

Получим объект класса URL и создадим на его основе URLRequest.

```
guard let url = components?.url else { return }
var request = URLRequest(url: url)
```

Установим http метод запроса:

```
request.httpMethod = "POST"
```

И добавим токен авторизации в заголовки:

```
request.setValue("OAuth = \(token)", forHTTPHeaderField: "Authorization"
```

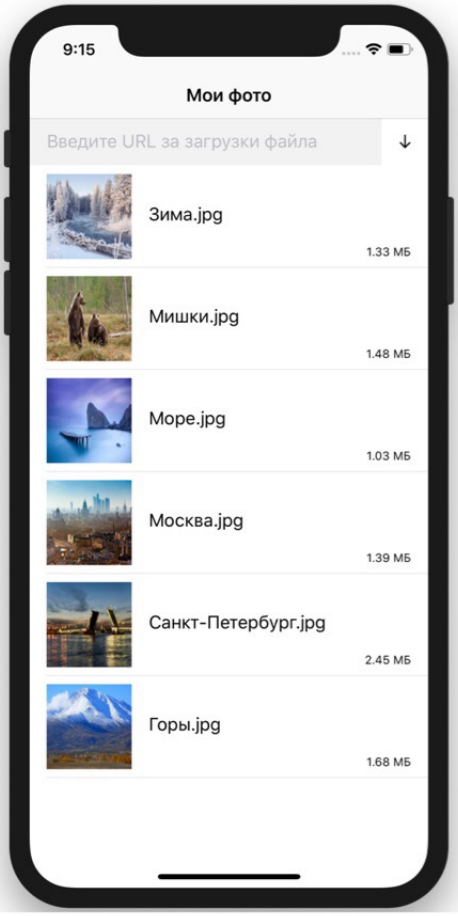
Создаем dataTask для выполнения post запроса:

```
let task = URLSession.shared.dataTask(with: request) { [weak self] (data, response,
error) in

}
.resume()
```

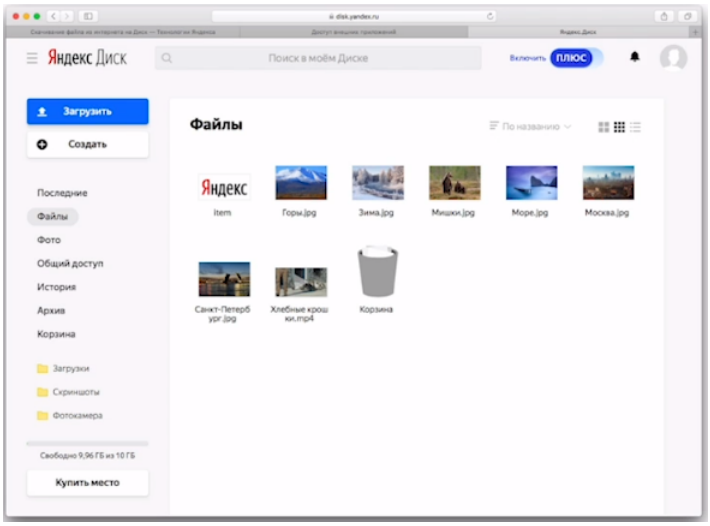
Обработаем полученный от сервера ответ:

```
if let response = response as? HTTPURLResponse {
    switch response.statusCode {
        case 200..<300:
        print("Success")
        default:
        print("Status: \(response.statusCode)")
    }
}
```

Запускаем приложение.



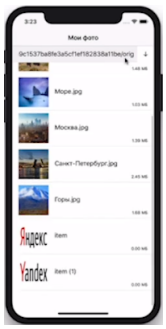

Перейдем в браузер и скопируем URL открытого изображения. Вернемся в приложение и попробуем загрузить картинку по URL. Нажимаем кнопку и видим, что сообщение об успешном завершении запроса напечаталось, однако новая картинка не появилась.
Проверим, работает ли загрузка файлов. Посмотрим на содержимое Диска в браузере. Действительно, новый файл появился. Однако в приложении изображения оперативно не обновляются.




Исправим это. Добавим вызов функции для обновления данных контроллера, после успешного выполнения запроса на скачивание файла.

```
self?.updateData()
```

Перезапускаем приложение. Снова идем в браузер и копируем на этот раз URL другой картинки. Возвращаемся в приложение и загружаем картинку. Теперь список картинок становится актуальным сразу после успешного скачивания нового изображения.




На этом у меня все. На протяжении недели мы с вами изучили, как взаимодействовать с сетью из iOS приложения, а на следующей неделе вы узнаете, как работать с базой данных в iOS.