



## Урок 2

# Работа с сетью

Учимся работать с сетевыми запросами. Анатомия HTTP-запросов. Обзор инструментов и библиотек для работы с сетевыми запросами. Отправка запросов с помощью URLSession.

[HTTP Request](#)

[URL](#)

[AppTransportSecurity](#)

[URLSession](#)

[Alamofire](#)

[Создание клиента для сервиса openweathermap.org](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

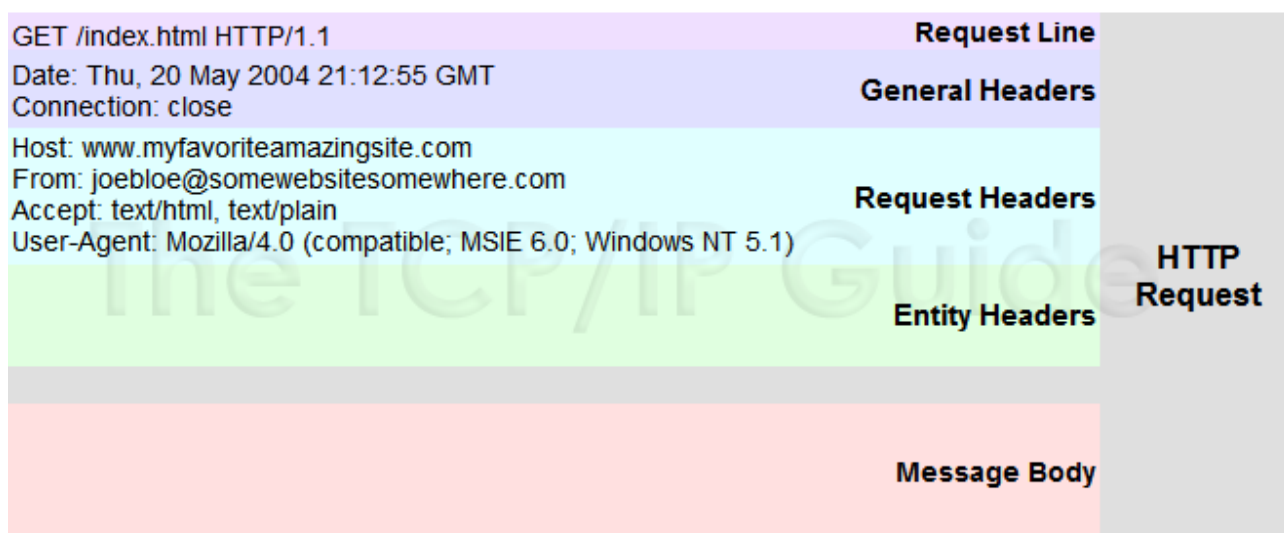
# HTTP Request

HTTP – протокол передачи данных, работающих поверх TCP. Грубо говоря, это текстовый документ, состоящий из нескольких частей.

Обычно взаимодействие с сервером состоит из операций запроса и ответа. Клиент отправляет на сервер запрос, сервер обрабатывает его и возвращает ответ. В качестве клиента может выступать браузер, мобильное или любое другое приложение. Существуют даже консольные http-клиенты.

Запросы и ответы выглядят похожим образом и состоят из множества полей с данными. Все данные можно условно разделить на три группы:

1. Стартовая строка (Request line) – содержит тип запроса и URI ресурса.
2. Заголовки (Headers).
3. Тело сообщения (Message body).



В стартовой строке содержится тип запроса. Давайте разберем, для чего он нужен. Тип запроса определяет действие, которое вы желаете совершить на сервере. Вы можете отправлять запрос на один и тот же URI, но, в зависимости от метода, действия на сервере будут совершаться разные. Если рассматривать запрос как предложение, URI – это существительное, а метод – глагол. Типы запросов:

- GET – получить. Используется для запроса данных с сервера;
- POST – создать. Используется для создания данных на сервере;
- PUT – изменить. Используется для изменения данных на сервере;
- DELETE – удалить. Используется для удаления данных на сервере.

К сожалению, эти типы всего лишь рекомендация и реальное значение различных типов запросов определяет только программист, создающий серверное приложение. Часто используют только два типа запросов: `get` – для получения данных, `post` – для всего остального, но бывает и хуже. Могут использоваться только `get` или только `post`-запросы. В любом случае строить предположения бесполезно, вы должны получать информацию о запросах, которые поддерживает сервер из документации к нему, гадать вы не должны.

Заголовки нужны, чтобы передавать различную вспомогательную информацию. Обычно в них указывается идентификатор клиента, источник запроса, вариант, в котором необходим ответ, время

отправки запроса и многое другое. Например, вы можете указывать токен для авторизации в заголовках.

Тело запроса может быть пустым либо содержать информацию, которую необходимо передать на сервер, например, там может быть массив параметров или файл для загрузки.

Отдельно стоит поговорить о параметрах. Вам часто придется передавать какие-то параметры. Например, в погодном приложении вы можете передать город, для которого необходимо получить погоду. Параметры можно передавать несколькими способами. Они могут быть переданы через URL либо через тело запроса. Довольно часто параметры передаются именно через URL, хотя это и спорное решение.

Важно понимать, что запросы и ответы содержат очень много информации, которая будет полезна для вас.

## URL

Думаю, вы много раз имели дело с URL. Его вы вводите в строке браузера для доступа к страницам. Его также называют ссылкой. Но мы так часто привыкли видеть его в этой роли, что даже в голову не приходит, что это нечто большее. URL – это указатель на местонахождение ресурса, часто в сети, но он может указывать на любые ресурсы, например, на файлы или части приложения.

Возьмем, например, URL для получения погоды – `api.openweathermap.org/data/2.5/forecast?id=524901`. Неискушенный пользователь будет рассматривать его как одну сплошную строку для получения результатов. Более опытный пользователь сможет увидеть адрес сервера **api.openweathermap.org**, и параметр **id=524901**. На самом деле URL содержит очень много информации, которая, впрочем, может быть не указана за ненадобностью.

**<схема>:[//<логин>:<пароль>@]<хост>[:<порт>][/]<URL-путь>[?<параметры>][#<якорь>]**

Вот так выглядит полный URL:

1. **<схема>**. Схема обращения к ресурсу, как правило, имеется в виду сетевая, но могут быть и другие варианты. (http, https, ftp, mailto, telnet, file). В браузере вы можете ее не указывать, тогда он попытается сам ее вычислить, но при самостоятельной отправке запросов вы должны обязательно ее указать.
2. **[//<логин>:<пароль>@]**. Логин и пароль для доступа к ресурсу. Некоторые данные могут требовать прохождения http-авторизации, в этом случае логин и пароль может быть передан прямо в URL.
3. **<хост>**. Адрес сервера или ресурса. Это может быть диск, на котором находится файл, ip-адрес или доменное имя в сети.
4. **[:<порт>]**. Многие ресурсы могут находиться на одном хосте, но на разных портах. Можно представить, что хост – многоквартирный дом, а порт – конкретная квартира. Для веб-сайтов, как правило, используется 80 порт, и он подставляется по умолчанию.
5. **<URL-путь>**. Путь – место, где находится нужный вам ресурс на сервере. Если продолжать аналогию с многоквартирным домом, это комната в конкретной квартире.
6. **[?<параметры>]**. Дополнительные параметры запроса. Вы можете передать данные, чтобы уточнить запрос, или для того чтобы сервер их записал.
7. **#<якорь>**. Чисто браузерная особенность, указывает конкретное место страницы, которое надо отобразить.

Давайте еще раз разберем URL `http://api.openweathermap.org/data/2.5/forecast?id=524901`:

1. Схема в нем не указана, но если посмотреть на сайт, откуда взят URL, можно выяснить, что используется http.
2. Логин и пароль не указаны, для доступа к ресурсу не требуется авторизация.
3. api.openweathermap.org – хост. В данном случае используется доменное имя, это адрес сервера.
4. Порт не указан, значит используется порт 80.
5. /data/2.5/forecast – конкретный путь для получения погоды за 5 дней.
6. После знака вопрос (?) указаны параметры.
7. id=524901 – единственный параметр в этом запросе. Его имя – **id**, а значение – 524901.

Пришло время поговорить о параметрах (их может быть несколько, один, ни одного). Параметры начинаются со знака **?**, разные параметры отделяются между собой знаком **&**. Имя и значение параметра отделены знаком **=**.

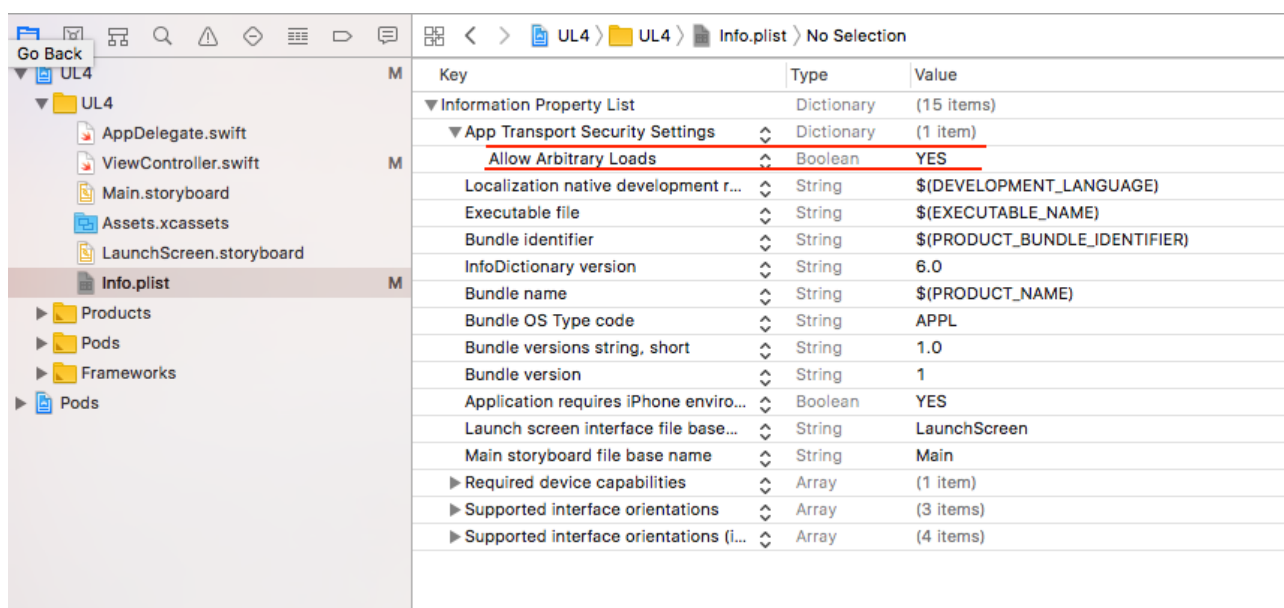
Важно понимать, что параметры не могут содержать абсолютно любые символы. Например, в них не может быть пробелов. Все символы, которые не могут быть переданы напрямую, необходимо кодировать. Например, пробел кодируется как **%20**. Как правило, кодировать самим ничего не надо. Для этого каждая библиотека имеет специальные методы.

## AppTransportSecurity

По умолчанию вы не можете отправлять незащищенные запросы в приложении. Это значит, что будут работать только HTTPS-запросы, а HTTP-запросы будут вызывать ошибку в приложении.

```
2017-09-23 10:45:56.221100+0700 UL4[714:14976] App Transport Security has
blocked a cleartext HTTP (http://) resource load since it is insecure. Temporary
exceptions can be configured via your app's Info.plist file.
```

Если вам все же надо работать с http-запросами, надо их разрешить. Для этого откройте файл **info.plist** и добавьте в него параметр **App Transport Security Settings**, а в него – **Allow Arbitrary Loads** со значением **YES**.



Key	Type	Value
Information Property List	Dictionary	(15 items)
App Transport Security Settings	Dictionary	(1 item)
Allow Arbitrary Loads	Boolean	YES
Localization native development r...	String	\$(DEVELOPMENT_LANGUAGE)
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle version	String	1
Application requires iPhone enviro...	Boolean	YES
Launch screen interface file base...	String	LaunchScreen
Main storyboard file base name	String	Main
Required device capabilities	Array	(1 item)
Supported interface orientations	Array	(3 items)
Supported interface orientations (i...	Array	(4 items)

# URLSession

Библиотека, предоставленная Apple, нужна, чтобы совершать запросы к серверу. Содержит множество компонентов.

- URLSession – сессия, в рамках которой совершаются запросы.
- URLSessionConfiguration – настройки сессии. Можно указать количество одновременных запросов на одну сессию, необходимо ли обрабатывать cookie сервера, должен ли кэшироваться ответ сервера и многое другое.
- URLSessionTask – задача, которая может быть запущена в рамках сессии.
  - URLSessionDataTask – получение информации;
  - URLSessionUploadTask – загрузка файла на сервер;
  - URLSessionDownloadTask – загрузка файла с сервера;
  - URLSessionStreamTask – постоянное соединение с сервером для обмена информацией.
- URL – URL для запроса;
- URLRequest – запрос на сервер.

URLSession – простой инструмент, который позволяет взаимодействовать с сервером. Давайте разберем несколько примеров. Например, нам необходимо получить данные от погодного сервиса. URI выглядит так:

<http://samples.openweathermap.org/data/2.5/forecast?q=Moscow,DE&appid=b1b15e88fa797225412429c1c50c122a1>.

Чтобы сделать запрос, необходима сессия, в рамках которой мы будем его совершать. Мы можем создать ее или воспользоваться сессией по умолчанию **URLSession.shared**. После этого мы создадим задачу **URLSession.shared.dataTask(with:completionHandler:)**, передав ей URL и замыкание, которое получит результат запроса.

```
// создаем URL из строки
let url = URL(string:
"http://samples.openweathermap.org/data/2.5/forecast?q=Moscow,DE&appid=b1b15e88fa797225412429c1c50c122a1")

// сессия по умолчанию
let session = URLSession.shared

// задача для запуска
let task = session.dataTask(with: url!) { (data, response, error) in
// в замыкании данные, полученные от сервера, мы преобразуем в json
    let json = try? JSONSerialization.jsonObject(with: data!, options:
JSONSerialization.ReadingOptions.allowFragments)
// выводим в консоль
    print(json)
}

// запускаем задачу
task.resume()
```

Это самый простой способ отправить запрос к серверу и получить от него данные. Но, как правило, запускать запросы на сессии по умолчанию не лучшая идея, даже если мы не будем менять настройки сессии. Лучше создать собственную сессию. Вам может показаться, что это лишний код без

причины, но в реальных приложениях сессии создаются в специальном классе, как правило одна или две.

```
// Конфигурация по умолчанию
let configuration = URLSessionConfiguration.default
// собственная сессия
let session = URLSession(configuration: configuration)

// создаем url из строки
let url = URL(string:
"http://samples.openweathermap.org/data/2.5/forecast?q=Moscow,DE&appid=b1b15e88f
a797225412429c1c50c122a1")

// задача для запуска
let task = session.dataTask(with: url!) { (data, response, error) in
// в замыкании данные, полученные от сервера, мы преобразуем в json
let json = try? JSONSerialization.jsonObject(with: data!, options:
JSONSerialization.ReadingOptions.allowFragments)
// выводим в консоль
print(json)
}
// запускаем задачу
task.resume()
```

Примеры отлично работают, но давайте поменяем URL, точнее, город, для которого мы получаем погоду. Мы запросим данные для Мюнхена и напомним название на немецком языке.

```
let url = URL(string:
"http://samples.openweathermap.org/data/2.5/forecast?q=München,DE&appid=b1b15e88
fa797225412429c1c50c122a1")
```

В результате мы получим ошибку: "ü" относится к символам, которые нельзя передавать как есть, его нужно кодировать. Но строку закодировать вы не можете. Создавать URL таким способом просто, но не безопасно.

Лучше использовать специальный конструктор `URLComponents`. Он позволяет создать url по частям, при этом параметры, которые будут созданы через специальный класс `URLQueryItem` будут автоматически кодированы и безопасны.

```
// Конфигурация по умолчанию
let configuration = URLSessionConfiguration.default

// собственная сессия
let session = URLSession(configuration: configuration)

// создаем конструктор для URL
var urlConstructor = URLComponents()
// устанавливаем схему
urlConstructor.scheme = "http"
```

```

// устанавливаем хост
urlConstructor.host = "samples.openweathermap.org"
// путь
urlConstructor.path = "/data/2.5/forecast"
// параметры для запроса
urlConstructor.queryItems = [
    URLQueryItem(name: "q", value: "München,DE"),
    URLQueryItem(name: "appid", value:
"b1b15e88fa797225412429c1c50c122a1")
]

// задача для запуска
let task = session.dataTask(with: urlConstructor.url!) { (data,
response, error) in
// в замыкании данные, полученные от сервера, мы преобразуем в json
let json = try? JSONSerialization.jsonObject(with: data!, options:
JSONSerialization.ReadingOptions.allowFragments)
// выводим в консоль
print(json)
}
// запускаем задачу
task.resume()

```

До этого мы отправляли get-запросы, но что, если необходимо отправить post-запрос? Для этого потребуется создать не просто URL для задачи, но и сам запрос. За создание запроса отвечает класс `URLRequest`. Так как мы не можем отправить post запрос на погодный сервис, отправим его на сервис, созданный для тестирования.

```

// Конфигурация по умолчанию
let configuration = URLSessionConfiguration.default

// собственная сессия
let session = URLSession(configuration: configuration)

// создаем конструктор для url
var urlConstructor = URLComponents()
// устанавливаем схему
urlConstructor.scheme = "http"
// устанавливаем хост
urlConstructor.host = "jsonplaceholder.typicode.com"
// путь
urlConstructor.path = "/posts"
// параметры для запроса
urlConstructor.queryItems = [
    URLQueryItem(name: "title", value: "foo"),
    URLQueryItem(name: "body", value: "bar"),
    URLQueryItem(name: "userId", value: "1")
]

// создаем запрос
var request = URLRequest(url: urlConstructor.url!)

```

```
// указываем метод
request.httpMethod = "POST"

// задача для запуска
let task = session.dataTask(with: request) { (data, response, error) in
    // в замыкании данные, полученные от сервера, мы преобразуем в json
    let json = try? JSONSerialization.jsonObject(with: data!, options:
JSONSerialization.ReadingOptions.allowFragments)
    // выводим в консоль
    print(json)
}
// запускаем задачу
task.resume()
```

Можно выполнить запрос и убедиться, что мы получим в ответ id созданной записи.

## Alamofire

URLSession – простой, гибкий и мощный инструмент. Но часто нам не нужна вся его мощь, а создавать запросы хочется еще проще. Для этого можно использовать библиотеку от стороннего разработчика - Alamofire.

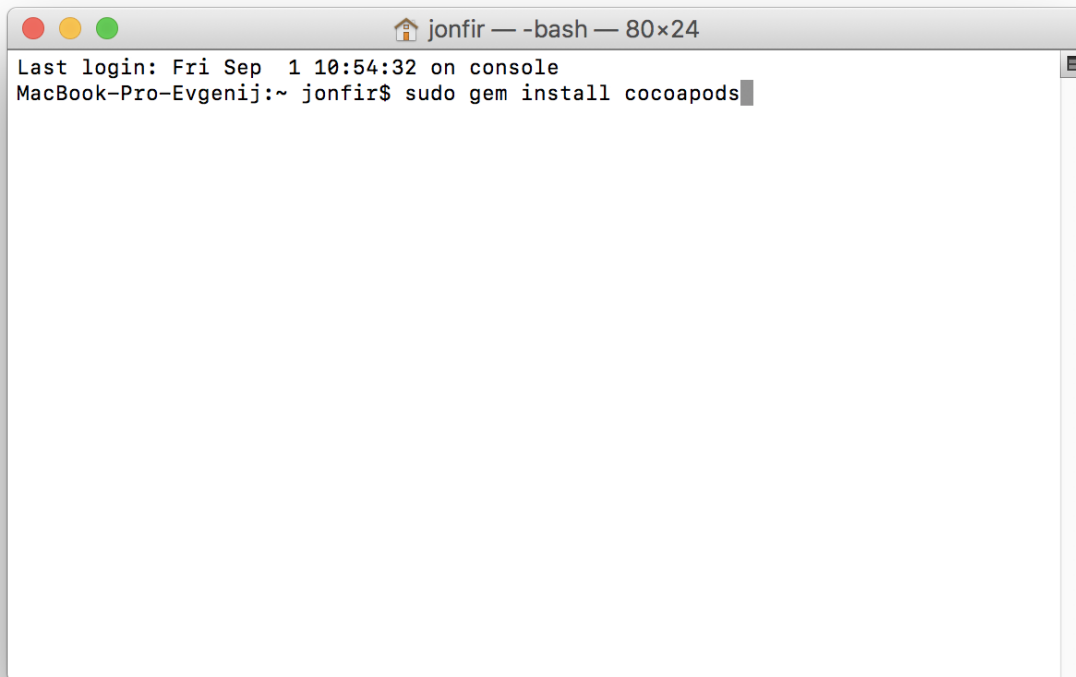
Alamofire немного упрощает работу с сетью. Чтобы использовать библиотеку, необходимо ее добавить в приложение. Ручное добавление библиотек в проект – нетривиальная задача, и мы даже не будем ее рассматривать. Есть способ получше. CocoaPods – программа, написанная на Ruby, которая берет на себя управление сторонними библиотеками, добавленными в проект.

Чтобы воспользоваться CocoaPods, ее необходимо установить на компьютер. Делается это из терминала. Откройте терминал (найти его можно в приложениях).



Вы увидите консоль ввода команд. Введите или скопируйте команду **sudo gem install cocoapods**, нажмите **Enter** и введите пароль компьютера.



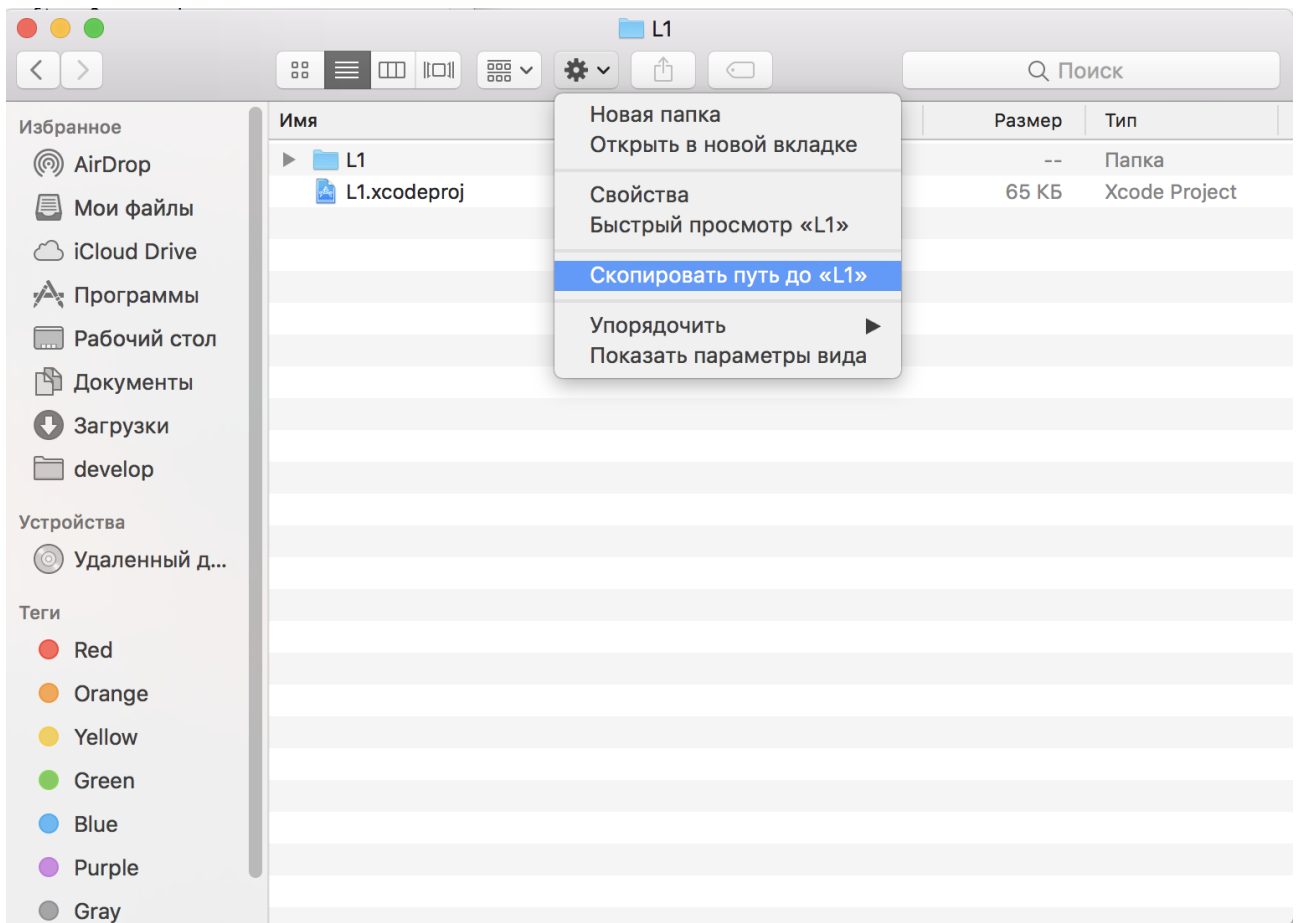


```
jonfir — -bash — 80x24
Last login: Fri Sep  1 10:54:32 on console
MacBook-Pro-Evgenij:~ jonfir$ sudo gem install cocoapods
```



```
jonfir — -bash — 80x24
Last login: Fri Sep  1 10:54:32 on console
MacBook-Pro-Evgenij:~ jonfir$ sudo gem install cocoapods
Password:
Successfully installed cocoapods-1.3.1
Parsing documentation for cocoapods-1.3.1
Done installing documentation for cocoapods after 2 seconds
1 gem installed
MacBook-Pro-Evgenij:~ jonfir$
```

Теперь можно использовать CocoaPods для управления зависимостями (библиотеками) в нашем проекте. Для этого необходимо отправиться в терминале в папку с проектом. Самый простой способ сделать это – открыть папку с проектом в Finder и через меню скопировать путь до нее. После чего перейти в терминал, набрать команду `cd`, через пробел вставить путь до папки и нажать ввод.

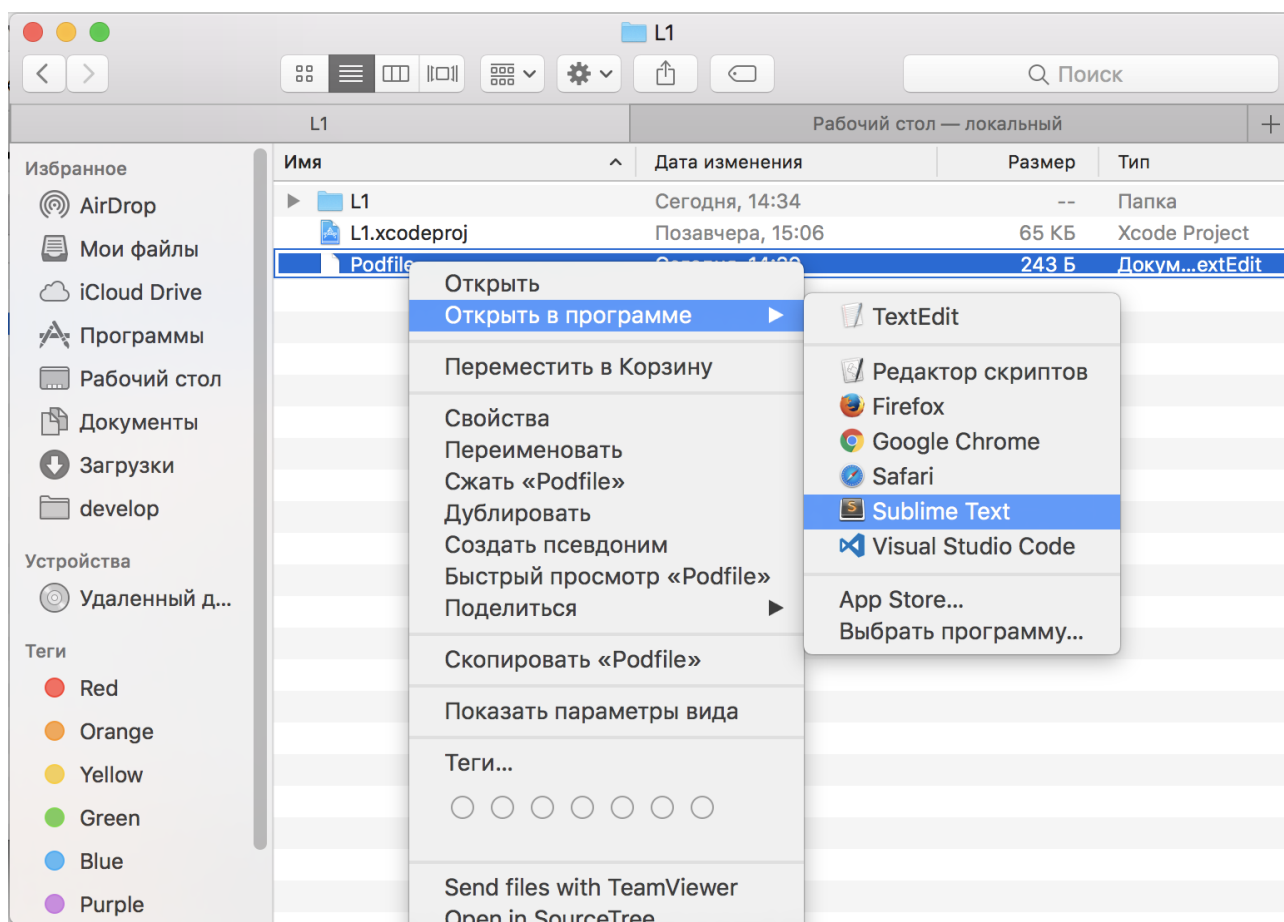
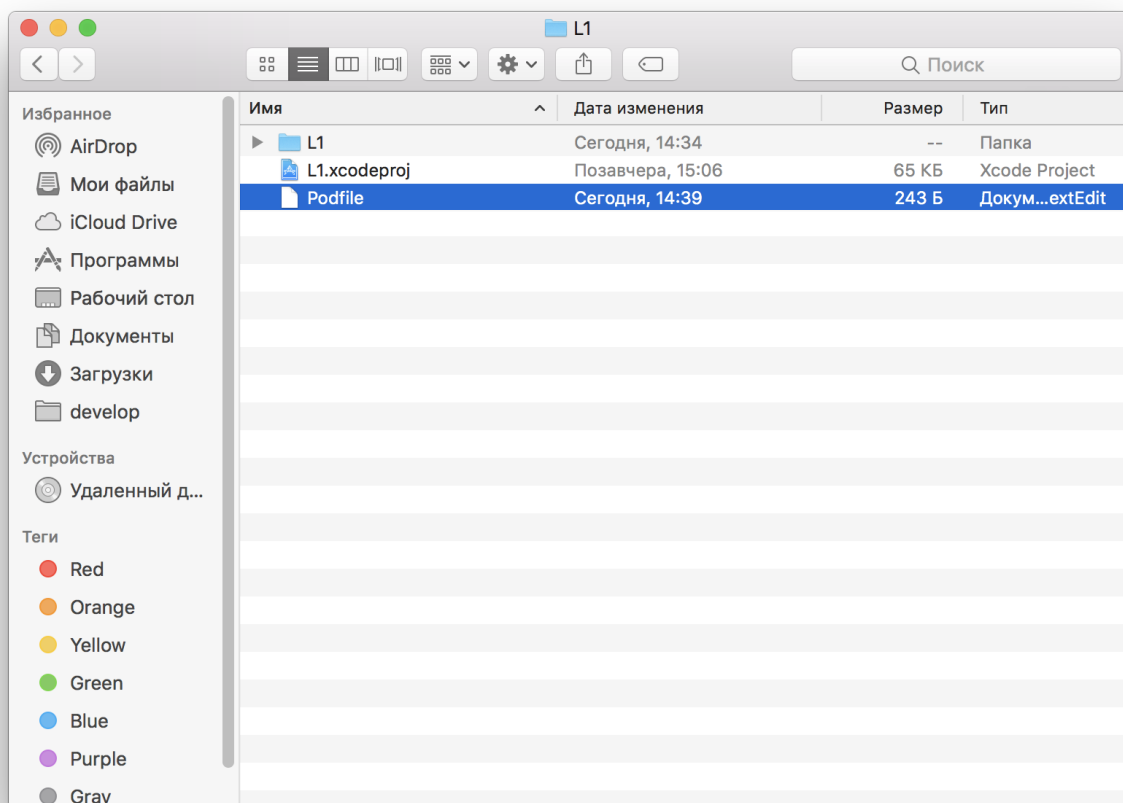


Теперь вы находитесь в нужной папке в терминале. Все зависимости, которые используются в проекте, перечисляются в специальном файле. Чтобы его создать, введите команду **pod init** и нажмите Enter.

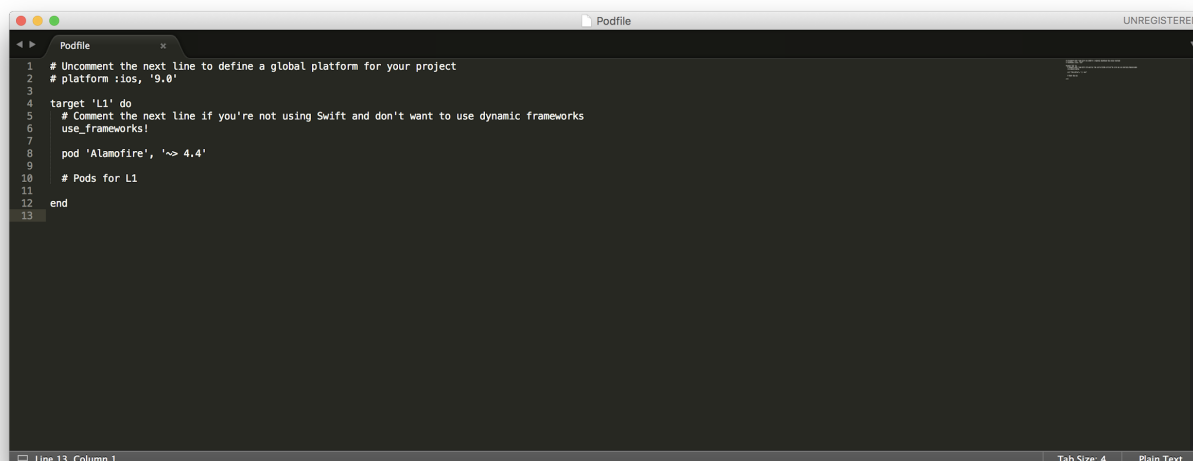
A screenshot of a macOS terminal window. The title bar shows 'L1 — -bash — 80x24'. The terminal content shows the user 'jonfir' at 'MacBook-Pro-Evgenij:L1' navigating to the directory '/Users/jonfir/develop/ios/swift2/L1' using the 'cd' command, and then running the 'pod init' command, which has just been entered and is followed by a cursor.

```
MacBook-Pro-Evgenij:L1 jonfir$ cd /Users/jonfir/develop/ios/swift2/L1
MacBook-Pro-Evgenij:L1 jonfir$ pod init
```

Теперь, когда файл создан, его можно увидеть в Finder и открыть через текстовый редактор. В качестве редактора рекомендую **sublimetext**.



Добавим в файл строку `pod 'Alamofire', '~> 4.8'`.



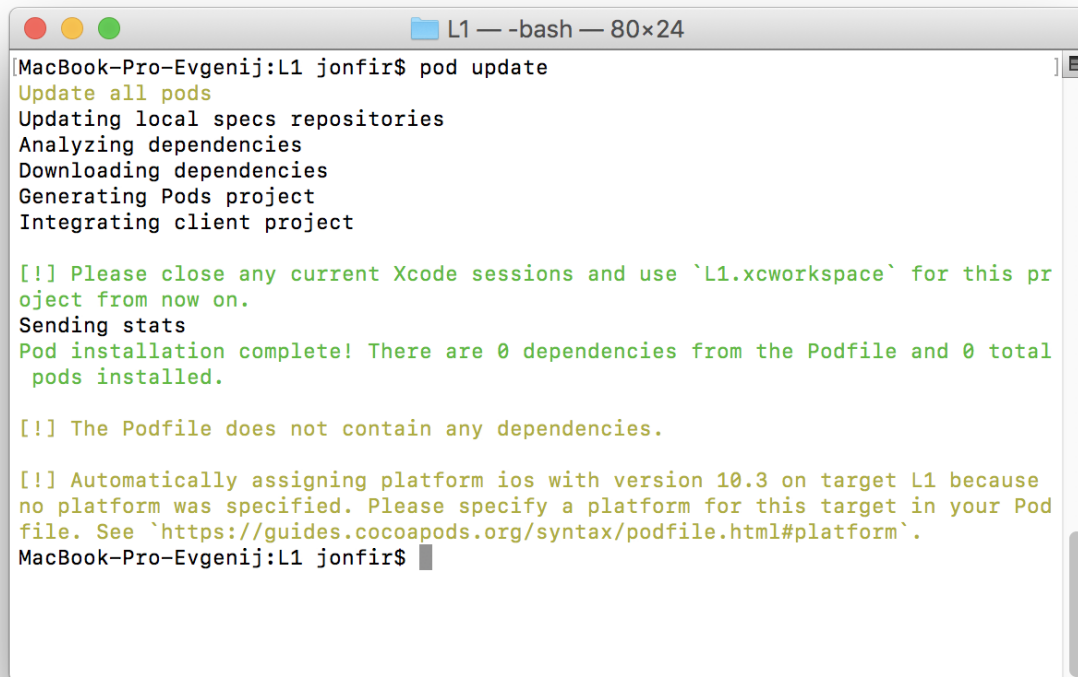
```
1 # Uncomment the next line to define a global platform for your project
2 # platform :ios, '9.0'
3
4 target 'L1' do
5   # Comment the next line if you're not using Swift and don't want to use dynamic frameworks
6   use_frameworks!
7
8   pod 'Alamofire', '~> 4.4'
9
10  # Pods for L1
11
12 end
13
```

Важно понимать, что время идет и, возможно, когда вы будете читать эту методичку, версия Alamofire будет не 4.8, а 5.6 или даже 7.0. Не поленитесь зайти на страницу библиотеки и почитать инструкцию по установке.

После того, как мы добавили в файл зависимость, необходимо провести установку библиотек. Вернемся в терминал; если вы его закрыли, проделайте операции, чтобы снова оказаться в папке проекта. После чего введите команду **pod update** и нажмите **Enter**.



```
MacBook-Pro-Evgenij:L1 jonfir$ pod update
```

A screenshot of a macOS terminal window titled "L1 — -bash — 80x24". The terminal shows the execution of the 'pod update' command. The output includes status messages like "Update all pods", "Updating local specs repositories", "Analyzing dependencies", "Downloading dependencies", "Generating Pods project", and "Integrating client project". It also contains informational messages in green text, such as "Please close any current Xcode sessions and use 'L1.xcworkspace' for this project from now on." and "Pod installation complete! There are 0 dependencies from the Podfile and 0 total pods installed." The terminal ends with a message stating that the Podfile does not contain any dependencies and a warning about automatically assigning the ios platform version 10.3. The prompt "MacBook-Pro-Evgenij:L1 jonfir\$" is visible at the bottom.

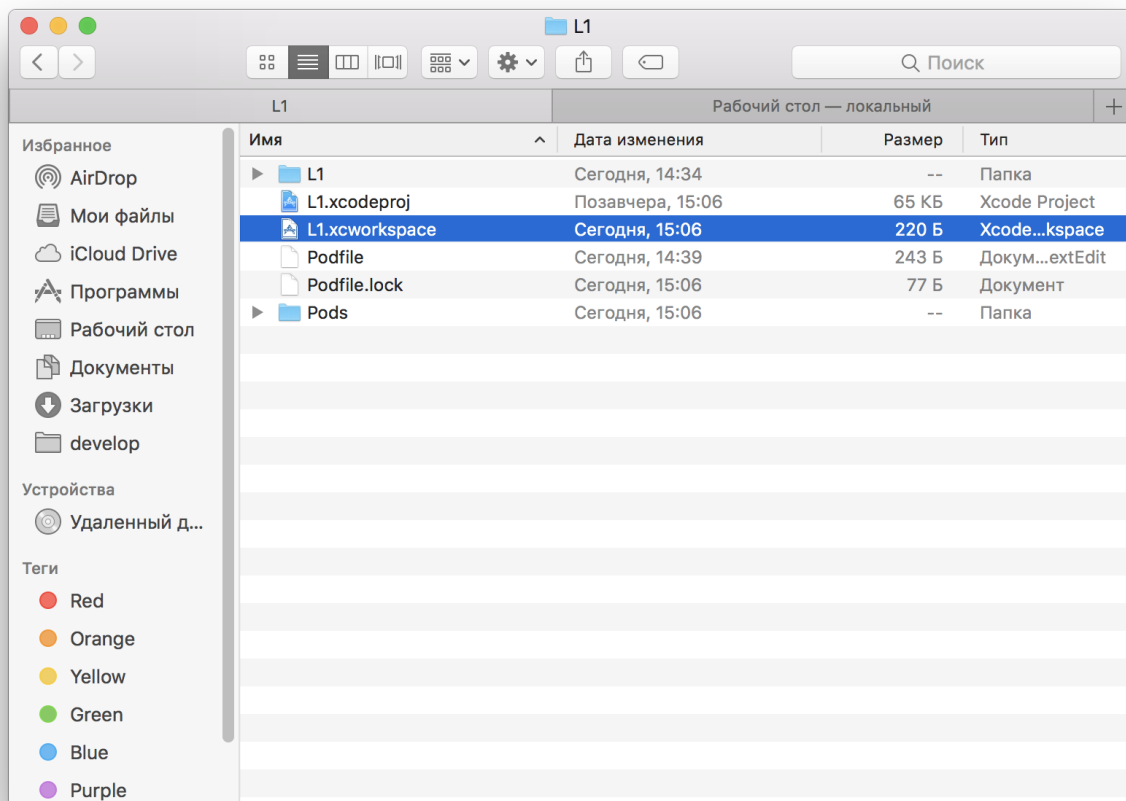
```
MacBook-Pro-Evgenij:L1 jonfir$ pod update
Update all pods
Updating local specs repositories
Analyzing dependencies
Downloading dependencies
Generating Pods project
Integrating client project

[!] Please close any current Xcode sessions and use `L1.xcworkspace` for this pr
oject from now on.
Sending stats
Pod installation complete! There are 0 dependencies from the Podfile and 0 total
pods installed.

[!] The Podfile does not contain any dependencies.

[!] Automatically assigning platform ios with version 10.3 on target L1 because
no platform was specified. Please specify a platform for this target in your Pod
file. See `https://guides.cocoapods.org/syntax/podfile.html#platform`.
MacBook-Pro-Evgenij:L1 jonfir$
```

Все, на этом терминал можно закрыть, библиотеки установлены. Обратите внимание: теперь проект необходимо открывать не как раньше, а через специальный файл: **xcworkspace**.



Теперь, когда проект подготовлен и Alamofire установлена, мы можем ее использовать.

Так как библиотека сторонняя, ее необходимо импортировать в каждом файле, где вы хотите ее использовать. Делается это добавлением строки **import Alamofire** в самом начале файла.

```
import UIKit
import Alamofire
```

Теперь перепишем примеры запросов с использованием Alamofire.

Первый пример – самый простой запрос. При использовании Alamofire мы можем не создавать URL, не получать сессию. Достаточно вызвать метод `request`, передать ему строку `url` и замыкание которое получит уже декодированный json.

```
// используем Alamofire для запроса к серверу, сразу передаем строку с URL и
// вызываем метод
// responseJSON, передавая ему замыкание.

Alamofire.request("http://samples.openweathermap.org/data/2.5/forecast?q=Moscow,
DE&appid=b1b15e88fa797225412429c1c50c122a1").responseJSON { response in

    print(response.value)

}
```

Выглядит проще, не правда ли? Кода меньше, результат тот же. Но настраивать запросы тоже можно. Для настройки своей сессии необходимо объявить статический параметр, например, в класс `SessionManager`:

```
// Добавляем расширение в класс SessionManager для хранения своего менеджера
extension SessionManager {
    static let custom: SessionManager = {
        // Конфигурация по умолчанию
        let configuration = URLSessionConfiguration.default
        // Добавляем заголовки из Alamofire
        configuration.httpAdditionalHeaders = SessionManager.defaultHTTPHeaders

        // менеджер сессии
        let sessionManager = SessionManager(configuration: configuration)
        return sessionManager
    }()
}

//...продолжаем код своей функции
// используем SessionManager.custom для запроса к серверу, сразу передаем
строку с URL и вызываем метод
// responseJSON, передавая ему замыкание.

SessionManager.custom.request("http://samples.openweathermap.org/data/2.5/foreca
st?q=Moscow,DE&appid=b1b15e88fa797225412429c1c50c122a1").responseJSON { response
in
    print(response.value)
}
```

Следующий пример – конструирование URL. С Alamofire не обязательно собирать весь URL по частям, но параметры все же необходимо из него вынести. Поэтому обрежем URL до начала параметров, вынесем их в словарь типа **Parameters** (этот тип в приложение добавляет Alamofire) и передать его в методе `request` через параметр **parameters**.

```
// параметры для запроса
let paramters: Parameters = [
    "q": "München,DE",
    "appid": "b1b15e88fa797225412429c1c50c122a1"
]

// используем SessionManager.custom для запроса к серверу, сразу передаем
строку с URL и вызываем метод
// responseJSON, передавая ему замыкание.

SessionManager.custom.request("http://samples.openweathermap.org/data/2.5/foreca
st", parameters: paramters).responseJSON { response in
    print(response.value)
}
```

Под конец разберем post-запрос. Чтобы превратить `get` в `post`, необходимо в метод **request** передать параметр **method** с параметром **.post**.



```

// параметры для запроса
let parameters: Parameters = [
    "title": "foo",
    "body": "bar",
    "userId": 1
]

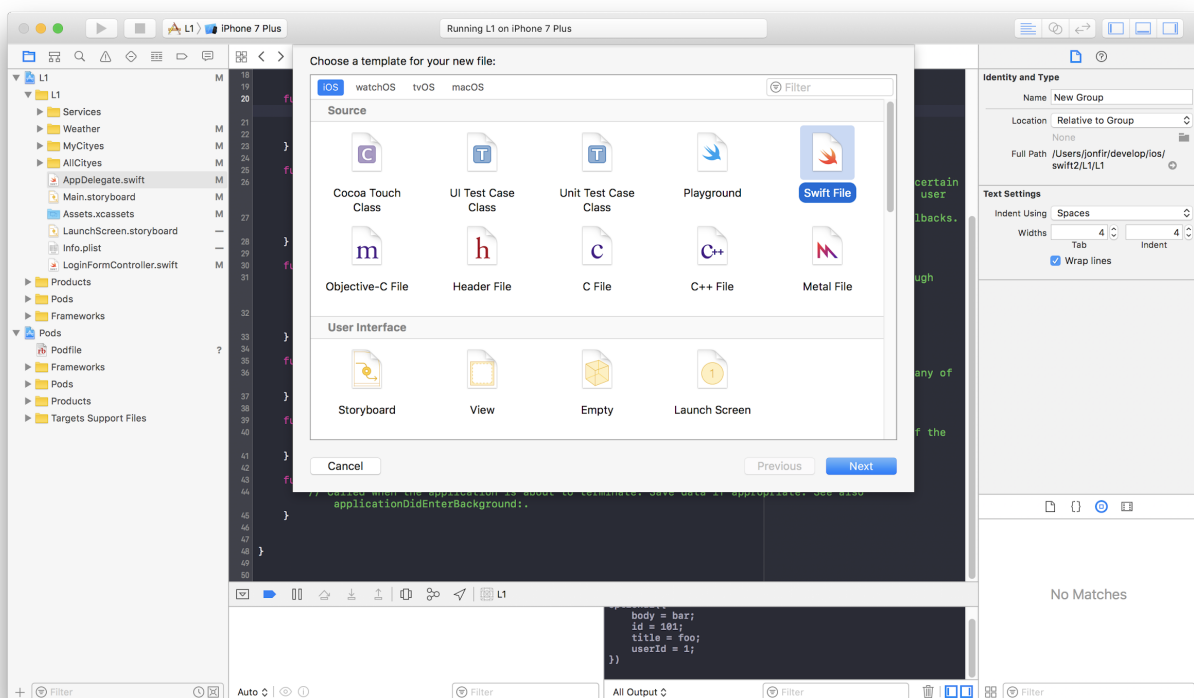
// используем SessionManager.custom для запроса к серверу, сразу передаем
// строку с URL и вызываем метод
// responseJSON, передавая ему замыкание.
SessionManager.custom.request("http://jsonplaceholder.typicode.com/posts",
method: .post, parameters: parameters).responseJSON { response in

    print(response.value)
}

```

## Создание клиента для сервиса openweathermap.org

Итак, добавим в проект класс для управления погодными данными. Для начала создадим папку **Services**, где будут находиться сервисы и создадим там один файл **WeatherService**.



Так как этот файл не шаблон, он создается пустым. Создадим в нем класс.

```
import Foundation

class WeatherService {

}
```

Теперь импортируем Alamofire.

```
import Foundation
import Alamofire

class WeatherService {

}
```

Теперь начнем создавать константы, у нас их будет две. Первая – базовый URL сервиса погоды, он же host. Если мы будем делать несколько разных запросов к сервису, эта часть будет общая, поэтому ее и следует вынести к константу. Вторая константа – ключ доступа к сервису. Он тоже общий для всех запросов.

Теперь напишем сам метод для получения погоды за пять дней. Этот метод будет получать один аргумент – city, город для которого будем получать погоду. В нем мы выполним всего один запрос к сервису, используя Alamofire. Полученные данные выведем в консоль.

```

import Foundation
import Alamofire

class WeatherService {
    // базовый URL сервиса
    let baseUrl = "http://api.openweathermap.org"
    // ключ для доступа к сервису
    let apiKey = "92cabe9523da26194b02974bfcd50b7e"

    // метод для загрузки данных, в качестве аргументов получает город
    func loadWeatherData(city: String) {

        // путь для получения погоды за 5 дней
        let path = "/data/2.5/forecast"
        // параметры, город, единицы измерения градусы, ключ для доступа к сервису
        let parameters: Parameters = [
            "q": city,
            "units": "metric",
            "appid": apiKey
        ]

        // составляем URL из базового адреса сервиса и конкретного пути к ресурсу
        let url = baseUrl+path

        // делаем запрос
        Alamofire.request(url, method: .get, parameters:
parameters).responseJSON { repsonse in
            print(repsonse.value)
        }
    }
}

```

Данные нужны для контроллера погоды **WeatherViewController**. Создадим экземпляр сервиса как свойство. В методе **viewDidLoad** обратимся к методу и получим:

```

// создаем экземпляр сервиса
let weatherService = WeatherService()

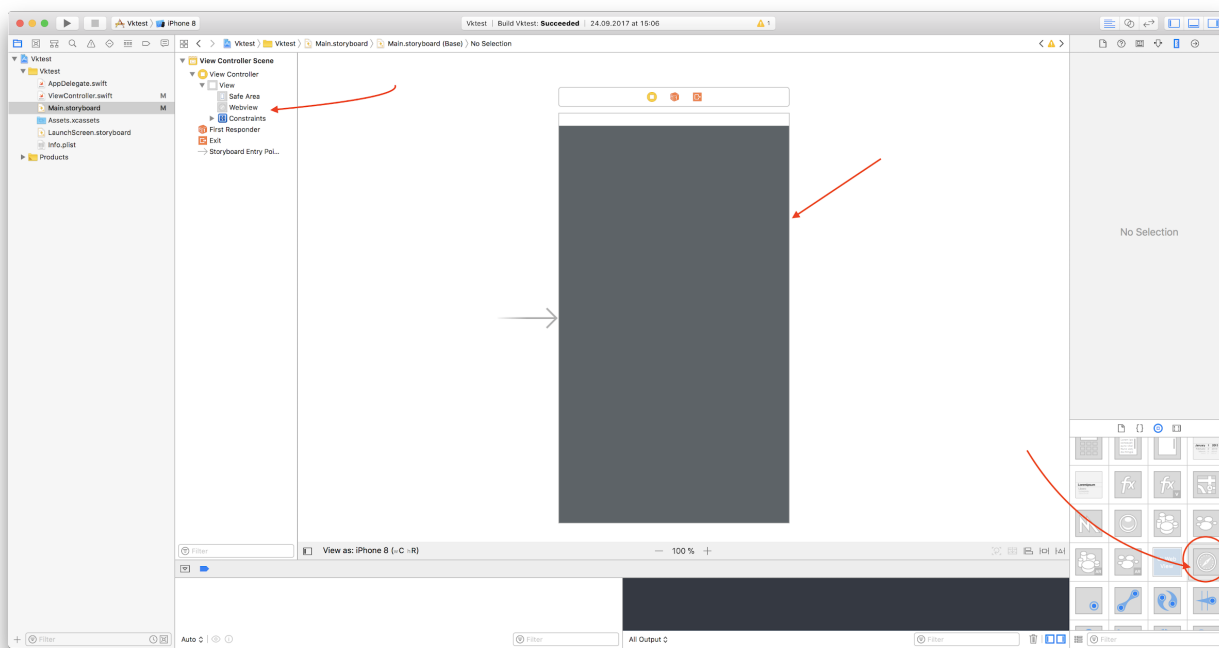
override func viewDidLoad() {
    super.viewDidLoad()

    // отправим запрос для получения погоды для Москвы
    weatherService.loadWeatherData(city: "Moscow")
}

```

# Авторизация во ВКонтакте

Для авторизации ВКонтакте используется не простая отправка пары логин/пароль на сервер. Для обеспечения безопасности приватных данных авторизация возможна только через браузер. К счастью, есть стандартный компонент, который позволяет отображать и взаимодействовать с web-страницами – `WKWebView`. Чтобы воспользоваться им, достаточно добавить на контроллер компонент **WKWebView** и растянуть его на весь экран.



Протяните от него IBOutlet в контроллер.

```
@IBOutlet weak var webview: WKWebView!
```

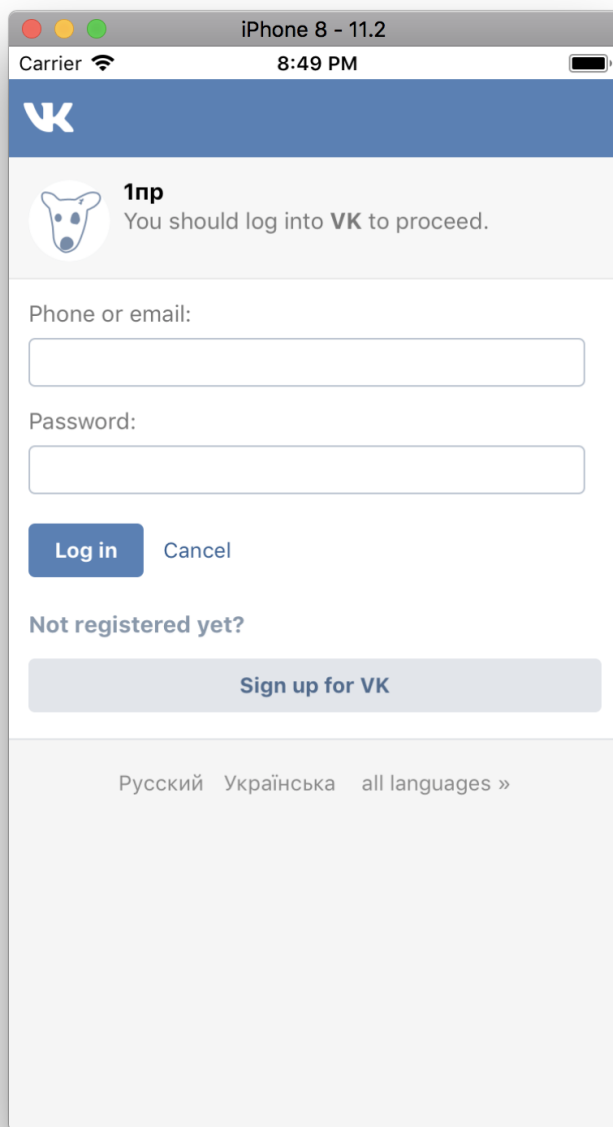
Теперь необходимо открыть в нем страницу авторизации в VK, подробная инструкция есть на [сайте](#). Технически это выглядит как передача URL в `WKWebView`.

```
var urlComponents = URLComponents()  
urlComponents.scheme = "https"  
urlComponents.host = "oauth.vk.com"  
urlComponents.path = "/authorize"  
urlComponents.queryItems = [  
    URLQueryItem(name: "client_id", value: "1234567"),  
    URLQueryItem(name: "display", value: "mobile"),  
    URLQueryItem(name: "redirect_uri", value:  
"https://oauth.vk.com/blank.html"),  
    URLQueryItem(name: "scope", value: "262150"),  
    URLQueryItem(name: "response_type", value: "token"),  
    URLQueryItem(name: "v", value: "5.68")  
]  
  
let request = URLRequest(url: urlComponents.url!)  
  
webview.load(request)
```

Обратите внимание: мы создаем URL <https://oauth.vk.com/authorize> и добавляем ему необходимые параметры. Создаем из URL запрос – URLRequest, – согласно [документации](#). После открываем страницу с помощью метода `webView.load(request)`.

Кроме стандартных параметров, вам необходимо зарегистрировать приложение в ВК на [странице](#) управления приложениями. Выберите для него тип **Standalone-приложение**. ID приложения вы найдете в разделе настройки наверху.

Если на данном этапе запустить приложение в симуляторе, мы увидим форму авторизации.



Если мы сейчас авторизуемся, увидим белый экран. Это потому, что мы никак не взаимодействуем со страницей и она живет своей жизнью. Необходимый токен содержится в URL страницы, на которую нас переадресуют после успешной авторизации. Необходимо отслеживать переходы между страницами. В этом нам поможет делегат `WKNavigationDelegate`. Установим `webView` в качестве делегата контроллер.

```

@IBOutlet weak var webView: WKWebView! {
    didSet{
        webView.navigationDelegate = self
    }
}

```

Имплементируем его контроллеру и реализуем метод, который перехватывает ответы сервера при переходе: `func webView(_ webView: WKWebView, decidePolicyFor navigationResponse: WKNavigationResponse, decisionHandler: @escaping (WKNavigationResponsePolicy) -> Void)`

```

extension ViewController: WKNavigationDelegate {
    func webView(_ webView: WKWebView, decidePolicyFor navigationResponse:
WKNavigationResponse, decisionHandler: @escaping (WKNavigationResponsePolicy) ->
Void) {

        guard let url = navigationResponse.response.url, url.path ==
"/blank.html", let fragment = url.fragment else {
            decisionHandler(.allow)
            return
        }

        let params = fragment
            .components(separatedBy: "&")
            .map { $0.components(separatedBy: "=") }
            .reduce([String: String]()) { result, param in
                var dict = result
                let key = param[0]
                let value = param[1]
                dict[key] = value
                return dict
            }

        let token = params["access_token"]

        print(token)

        decisionHandler(.cancel)
    }
}

```

Теперь мы можем отслеживать все переходы и отменять либо разрешать их по необходимости.

```

guard let url = navigationResponse.response.url, url.path == "/blank.html", let
fragment = url.fragment else {
    decisionHandler(.allow)
    return
}

```

```
}
```

Первая часть кода проверяет URL, на который было совершено перенаправление. Если это нужный нам URL (/blank.html), и в нем есть токен, приступим к его обработке, если же нет, дадим зеленый свет на переход между страницами с помощью метода **decisionHandler.allow**. Далее мы просто режем строку с параметрами на части, используя как разделители символы **&** и **=**. В результате получаем словарь с параметрами.

```
let params = fragment
    .components(separatedBy: "&")
    .map { $0.components(separatedBy: "=") }
    .reduce([String: String]() { result, param in
        var dict = result
        let key = param[0]
        let value = param[1]
        dict[key] = value
        return dict
    })
```

Токен имеет ключ "access\_token", Мы можем получить его и использовать в наших запросах к ВК.

## Практическое задание

На основе ПЗ предыдущего урока:

1. Зарегистрировать приложение в ВК;
2. Заменить форму входа на WKWebView для авторизации в ВК;
3. Сохранить токен в синглтоне Session;
4. Использовать токен в запросах к VK API;
5. Реализовать запросы к VK API;
6. Получение списка друзей;
7. Получение фотографий человека;
8. Получение групп текущего пользователя;
9. Получение групп по поисковому запросу;
10. Вывести данные в консоль.

P.S.: Парсить данные не нужно, просто вывести в консоль.

P.P.S: Использовать VKSDK нельзя, мы учимся работать с сетью, а не с фреймворком от VK.

## Дополнительные материалы

1. [http](http://)
2. [URL](http://)
3. [сервис для теста запросов](http://)
4. [погодный сервис](http://)
5. [cocoapods.org](http://cocoapods.org)
6. [Alamofire](http://)

7. [sublimetext](#)
8. [postman](#)
9. [paw](#)
10. [Авторизация в ВК](#)

## Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. <https://ru.wikipedia.org/wiki/HTTP>
2. <https://ru.wikipedia.org/wiki/URL>