



Урок 3

Mapping json

Преобразуем json-ответ сервера в объекты приложения.
Анатомия json-формата. JSONSerialization.

[Ответ сервера](#)

[Анатомия json-формата](#)

[JSONSerialization](#)

[Codable](#)

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

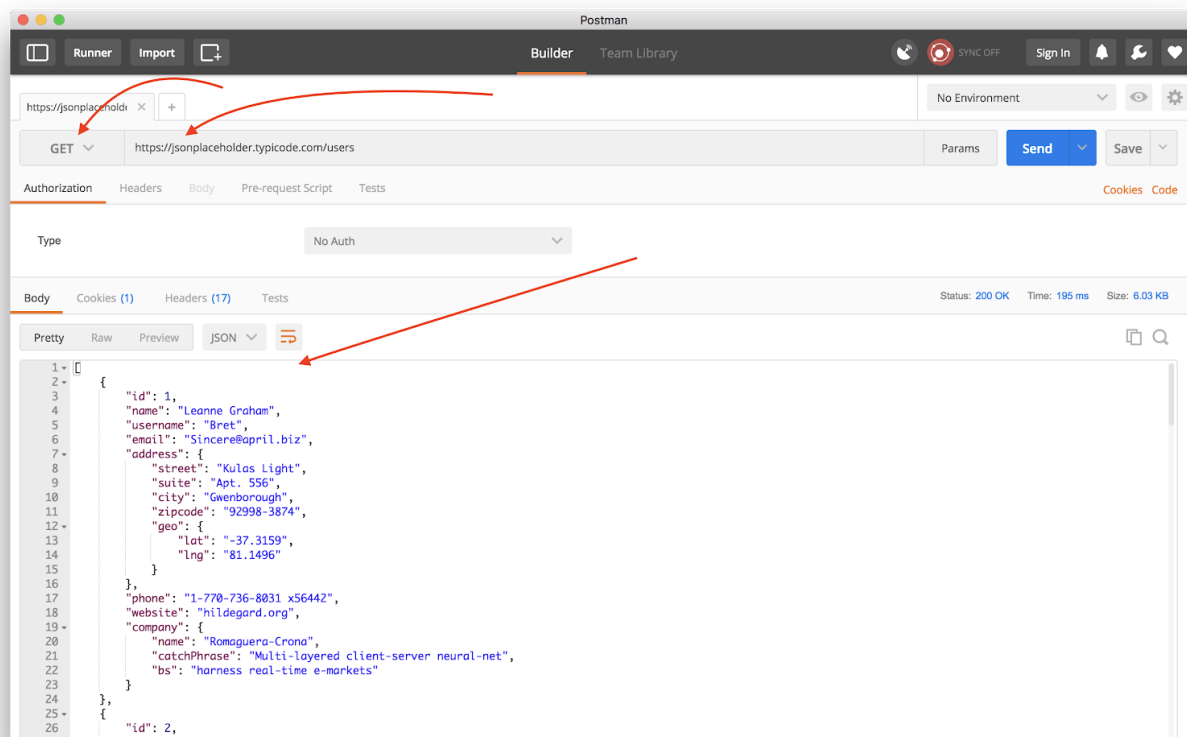
[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Ответ сервера

При запросе на сервер мы, как правило, ждем от него ответ. Если сервер имеет хорошую документацию, мы можем изучить ответ заранее. Но далеко не всегда мы имеем хорошую документацию и нам приходится выяснять структуру ответа самостоятельно. Чтобы изучить ответ сервера, я рекомендую использовать [postman](#). Это отличная бесплатная программа. Например, есть URL <https://jsonplaceholder.typicode.com/users> и мы должны выяснить, в каком виде он возвращает данные. Откроем postman, добавим туда URL, выберем тип запроса (в данном случае get) и выполним его.



В ответ пришел json. Прежде чем разбирать json, поговорим о том, что именно нам может вернуть сервер.

В первую очередь от сервера мы получаем двоичные данные, каким бы ни было содержимое. Далее мы можем эти данные раскодировать. Если мы запрашивали файл, его можно либо сохранить на диск, либо использовать в приложении. Так, если приходит изображение, мы можем сразу создать объект `UIImage` и отобразить его на экране, его даже не нужно сохранять. Правда, если при следующем запуске приложения снова понадобится это изображение, его придется качать заново.

Если мы запрашивали не файл, скорее всего, пришел текст. Это может быть html-код страницы, но мобильным приложениям редко приходится работать с html-данными. Это, скорее, исключительный случай, когда вы хотите взаимодействовать с сайтом, который для этого не предназначен.

Еще один вариант – xml. Это текстовый формат представления структур данных. Он достаточно мощный, может описывать различные коллекции данных, объекты и прочее. Но, к сожалению, он очень объемный, содержит много лишней информации и в последние годы его уже не используют.

Чаще всего вы будете работать с json. Это современный, простой и легкий формат, в котором тоже можно описать любую структуру данных.

Работая с URLSession, вы всегда будете получать двоичные данные и дальше вам придется преобразовывать их в подходящий тип, строку, изображение, аудио. Если вы используете Alamofire, у вас есть специальные парсеры. Вам достаточно выбрать нужный и вы получите декодированные данные.

В этом уроке рассмотрим json-формат и научимся с ним работать.

Анатомия json-формата

Итак, мы получили данные от сервера, в которых содержится json. Для начала нам надо преобразовать эти данные в json-объект. Если мы используем URLSession, то воспользуемся методом `JSONSerialization.jsonObject(with: data, options: JSONSerialization.ReadingOptions.mutableContainers)`. Передаем ему данные, получаем то, что хотели – объект json. Если используется Alamofire, достаточно указать тип ответа `responseJSON` и мы получим сразу декодированный объект.

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

    guard let data = data,
          let json = try? JSONSerialization.jsonObject(with:
data, options: JSONSerialization.ReadingOptions.mutableContainers)
    else {

        return

    }

}
```

```
Alamofire.request(url).responseJSON { response in
    guard let json = response.value else {
        return
    }
}
```

С полученным json-объектом уже можно работать, преобразовывать в необходимые структуры данных. Чтобы это сделать, необходимо понять, что собой представляет json-структура. Json-структура – пары **ключ-значение**, как словарь, с той лишь разницей, что в качестве ключа всегда текстовая строка, а вот значения могут быть различных типов. Именно поэтому нельзя сконвертировать json в словарь. Также json может содержать массивы. Фактически, это структура из словарей и массивов различной степени вложенности, т.е. находящихся одни в других. Автоматически прочитать такую структуру невозможно. Нужны инструкции от программиста.

Типы, которые могут быть в json:

- Объект – пары **ключ:значение** в фигурных скобках «{ }»;
- Массив заключается в квадратные скобки «[]»;
- Число;
- Литералы true, false и null;
- Строка.

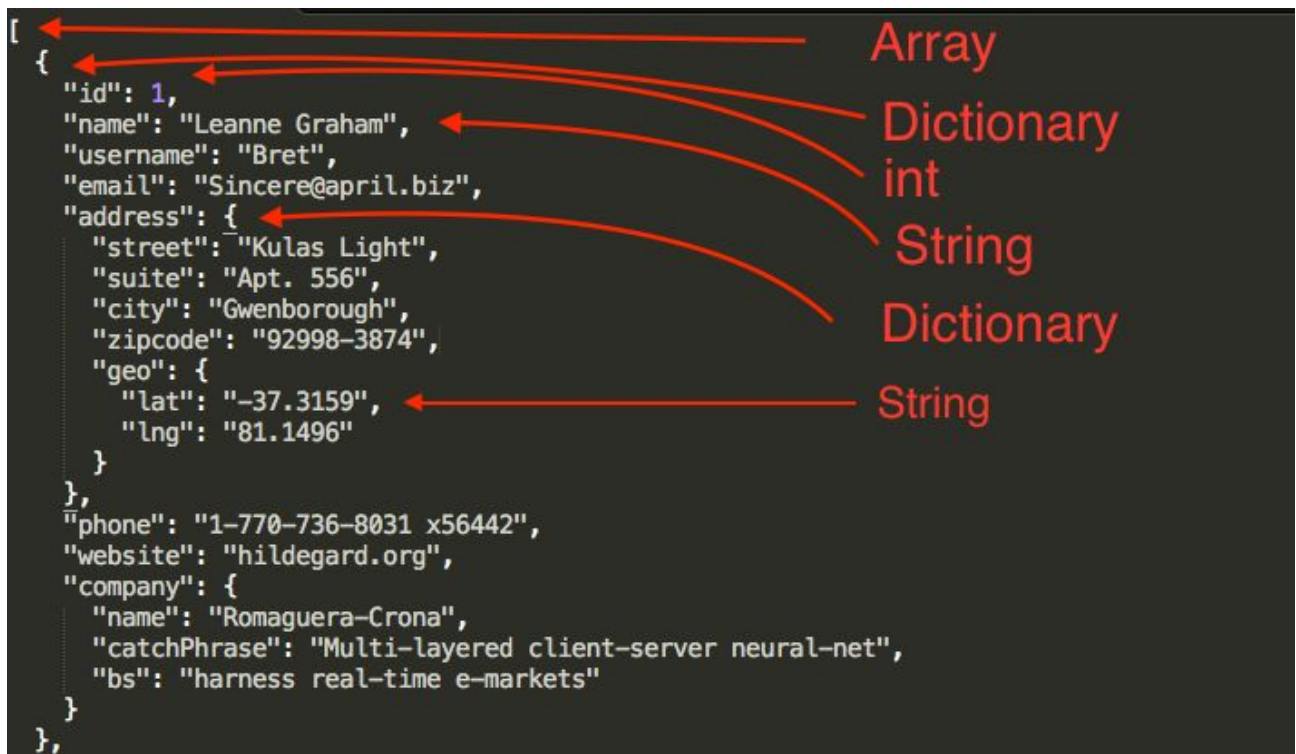
Аналоги swift-структур данных:

- Объект – Dictionary;
- Массив – Array;
- Число – Int, Double;
- Литералы true, false и null. – Bool, nil;
- Строка – String.

Как мы видим, у всех данных есть аналоги. Сложность состоит в том, что массивы и словари могут содержать разные типы, а в swift такое невозможно.

Как вы понимаете, каждый ответ сервера уникален. Разные URL возвращают разный json. Нужно его исследовать, определить, как его парсить и написать инструкцию, метод для преобразования, парсинга json.

Давайте исследуем json, полученный по указанному ранее URL.



Что мы видим? Во-первых, все находится внутри квадратных скобок. Вспоминаем, что квадратные скобки – это массив, значит, верхний уровень json – массив объектов, его можно перебирать в цикле. На втором уровне мы видим фигурные скобки, значит на втором уровне у нас словарь, или, другими словами, пока мы видим, что у нас есть массив, элементы которого являются словарями.

Теперь приступим к анализу словаря:

- id – целое число, значит Int;
- name – строка;
- username – строка;
- email – строка;
- адрес – вложенный словарь:
 - street – строка;
 - suite – строка;
 - city – строка;
 - zipcode – строка;
 - geo – вложенный словарь:
 - lat – выглядит как double, но так как находится в кавычках, это тоже строка;

- lng – строка.
- phone – строка;
- website – строка;
- company – вложенный словарь:
 - name – строка;
 - catchPhrase – строка;
 - bs – строка.

Формат данных очень важен, ведь мы будем приводить типы в момент парсинга. Дело в том, что json-объект – Any и его нужно вскрывать уровень за уровнем, приводя типы для каждой переменной. Если мы ошибемся хотя бы в одном типе, получим ошибку.

JSONSerialization

Давайте напишем функцию для парсинга этого json.

```
let url = URL(string: "https://jsonplaceholder.typicode.com/users")!

URLSession.shared.dataTask(with: url) { data, response, error in

    guard let data = data,
          let json = try? JSONSerialization.jsonObject(with: data,
options: JSONSerialization.ReadingOptions.mutableContainers) else {
        return
    }
    let array = json as! [Any]
    for userJson in array {
        let userJson = userJson as! [String: Any]
        let id = userJson["id"] as! Int
        let name = userJson["name"] as! String
        let username = userJson["username"] as! String
        let email = userJson["email"] as! String
        let addressJson = userJson["address"] as! [String: Any]
        let street = addressJson["street"] as! String
        let suite = addressJson["suite"] as! String
        let city = addressJson["city"] as! String
        let zipcode = addressJson["zipcode"] as! String
        let geoJson = addressJson["geo"] as! [String: Any]
        let lat = geoJson["lat"] as! String
        let lng = geoJson["lng"] as! String
        let phone = userJson["phone"] as! String
        let website = userJson["website"] as! String
        let companyJson = userJson["company"] as! [String: Any]
        let companyName = companyJson["name"] as! String
        let catchPhrase = companyJson["catchPhrase"] as! String
        let bs = companyJson["bs"] as! String

        print(id, name, username, email, street, suite, city, zipcode,
lat, lng, phone, website, companyName, catchPhrase, bs)

    }
}.resume()
```

Обратите внимание, как мы постепенно приводим типы для каждого уровня и обрабатываем переменные соответствующим образом.

```
data -> let json = try? JSONSerialization.jsonObject(with: data, options: JSONSerialization.ReadingOptions.mutableContainers)
[ -> let array = json as! [Any]
  -> for userJson in array
  { -> let userJson = userJson as! [String: Any]
    "id": 1, -> let id = userJson["id"] as! Int
    "name": "Leanne Graham", -> let name = userJson["name"] as! String
    "username": "Bret", -> let username = userJson["username"] as! String
    "email": "Sincere@april.biz", -> let email = userJson["email"] as! String
    "address": { -> let addressJson = userJson["address"] as! [String: Any]
      "street": "Kulas Light", -> let street = addressJson["street"] as! String
      "suite": "Apt. 556", -> let suite = addressJson["suite"] as! String
      "city": "Gwenborough", -> let city = addressJson["city"] as! String
      "zipcode": "92998-3874", -> let zipcode = addressJson["zipcode"] as! String
      "geo": { -> let geoJson = addressJson["geo"] as! [String: Any]
        "lat": "-37.3159", -> let lat = geoJson["lat"] as! String
        "lng": "81.1496" -> let lng = geoJson["lng"] as! String
      }
    },
    "phone": "1-770-736-8031 x56442", -> let phone = userJson["phone"] as! String
    "website": "hildegard.org", -> let website = userJson["website"] as! String
    "company": { -> let companyJson = userJson["company"] as! [String: Any]
      "name": "Romaguera-Crona", -> let name = companyJson["name"] as! String
      "catchPhrase": "Multi-layered client-server neural-net", -> let catchPhrase = companyJson["catchPhrase"] as! String
      "bs": "harness real-time e-markets" -> let bs = companyJson["bs"] as! String
    }
  }
]
```

Возможно, это изображение лучше объяснит, как следует приводить тип для каждой строки.

После того как мы распарсили json, встает вопрос, что делать с полученными данными. Конечно, мы можем вывести их в консоль, но вряд ли это понадобится в реальном приложении. Положить эти переменные в массив или словарь мы не можем, так как каждая переменная имеет свой тип. И, фактически, каждый словарь в массиве – это пользователь. Давайте создадим структуру User, которая будет представлять все эти данные. Если быть точным, нам нужны 4 структуры – User, Address, Geo, Company. Каждую из структур мы снабдим соответствующим конструктором для парсинга его под json.

```
struct User {
    let id: Int
    var name = ""
    var username = ""
    var email = ""
    var address: Address?
    var phone = ""
    var website = ""
    var company: Company?

    init(id: Int) {
        self.id = id
    }

    // вспомогательный init для парсинга json
    init(json: [String: Any] ) {
        let id = json["id"] as! Int
        self.init(id: id)

        self.name = json["name"] as! String
        self.username = json["username"] as! String
    }
}
```

```

        self.email = json["email"] as! String
        self.phone = json["phone"] as! String
        self.website = json["website"] as! String

        let addressJson = json["address"] as! [String: Any]
        self.address = Address(json: addressJson)

        let companyJson = json["company"] as! [String: Any]
        self.company = Company(json: companyJson)
    }
}

struct Address {
    struct Geo {
        var lat = ""
        var lng = ""
    }

    var street = ""
    var suite = ""
    var city = ""
    var zipcode = ""
    var geo: Geo?

    init(json: [String: Any] ) {
        self.suite = json["suite"] as! String
        self.city = json["city"] as! String
        self.zipcode = json["zipcode"] as! String
        let geoJson = json["geo"] as! [String: Any]
        let lat = geoJson["lat"] as! String
        let lng = geoJson["lng"] as! String
        self.geo = Geo(lat: lat, lng: lng)
    }
}

struct Company {
    var name = ""
    var catchPhrase = ""
    var bs = ""

    init(json: [String: Any] ) {
        self.name = json["name"] as! String
        self.catchPhrase = json["catchPhrase"] as! String
        self.bs = json["bs"] as! String
    }
}

```

Важно понимать, что использование принудительного приведения типов “!” допустимо только в учебных целях. В реальном приложении очень легко получить ошибку, если ответ от сервера по каким либо причинам будет отличаться от того, что вы ожидаете.

Осталось только начать парсить объекты.

```

let url = URL(string: "https://jsonplaceholder.typicode.com/users")!

URLSession.shared.dataTask(with: url) { data, response, error in

    guard let data = data,
          let json = try? JSONSerialization.jsonObject(with: data,
options: JSONSerialization.ReadingOptions.mutableContainers) else {
        return
    }

    let array = json as! [[String: Any]]
    let users = array.map { User(json: $0) }

}.resume()

```

В конце мы получаем массив структур User, с которыми уже можем работать.

Codable

Это относительно новый инструмент, появившийся в Swift 4. Он позволяет достаточно удобно преобразовывать json в объекты нашего языка.

Давайте перепишем пример выше на Codable. Во-первых, нам надо написать такие же структуры, как мы видим в ответе сервера.

```

struct User: Codable {
    let id: Int
    let name: String
    let username: String
    let email: String
    let address: Address
    let phone: String
    let website: String
    let company: Company
}

struct Address: Codable {
    let street: String
    let suite: String
    let city: String
    let zipcode: String
    let geo: Geo
}

struct Geo: Codable {
    let latitude: String
    let longitude: String
}

```



```

enum CodingKeys: String, CodingKey {
    case latitude = "lat"
    case longitude = "lng"
}

struct Company: Codable {
    let name: String
    let catchPhrase: String
    let bs: String
}

```

Первая структура – User. Ее свойства в точности повторяют названия и типы полей в json. Также она имплементирует протокол Codable. Это все необходимое, чтобы использовать ее для парсинга. Давайте посмотрим пример до конца. Поле address имеет тип Address, это структура, чьи поля совпадают с вложенным объектом в json. И она точно так же имплементирует протокол Codable. У нее есть свойство geo, которое также является структурой типа Codable. Аналогично обстоят дела со свойством company. На каждый вложенный объект в json мы должны создавать свой тип, имплементирующий Codable. С одной стороны, это не всегда удобно, с другой стороны, это все что нам необходимо сделать.

Отдельного внимания заслуживает структура Geo. Ее поля не совпадают по названию со свойствами в json. Такая структура не распарсится, но, к счастью, есть инструмент для уточнения свойств. Это перечисление CodingKeys. Достаточно создать такой enum, вложенный в структуру, указать, что его тип String, он имплементирует протокол CodingKey и перечислить все поля, указав их именование в json.

Приступим к парсингу.

```

Alamofire.request("https://jsonplaceholder.typicode.com/users").responseData {
    response in
        guard let data = response.value else { return }

        do {
            let users = try JSONDecoder().decode([User].self, from: data)
            print(users)
        } catch {
            print(error)
        }
    }
}

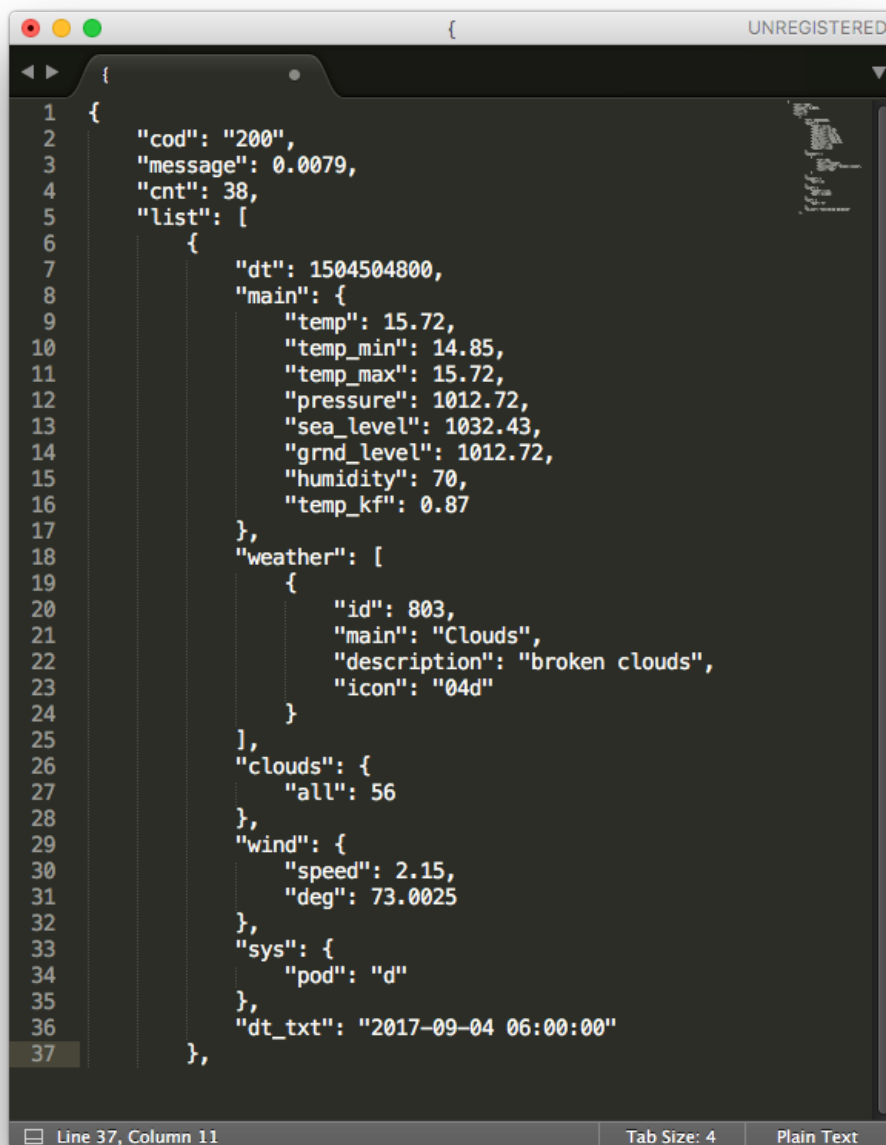
```

Ключевой здесь является строка **try JSONDecoder().decode([User].self, from: data)**. Создаем экземпляр класса JSONDecoder и вызываем его метод decode, которому передаем тип извлекаемой структуры и данные, которые необходимо парсить. Обратите внимание: мы передали не просто User.self, а [User].self. Дело в том, что в json здесь содержится не один пользователь, а их массив, следовательно, тип извлекаемого объекта – массив структур User. Так как метод decode может генерировать исключения, он заключен в блок do/catch. Если мы ошиблись с именованием или типом полей в структурах, будет сгенерировано исключение, и в ошибке мы получим точные сведения, где она произошла.

Создание клиента для сервиса openweathermap.org

На прошлом занятии мы остановились на том, что загрузили погодные данные из сети. На этом занятии создадим структуру для хранения погоды, распарсим полученный json и отобразим данные на экране.

Для начала посмотрим, как выглядит ответ сервера.



```
{
  "cod": "200",
  "message": 0.0079,
  "cnt": 38,
  "list": [
    {
      "dt": 1504504800,
      "main": {
        "temp": 15.72,
        "temp_min": 14.85,
        "temp_max": 15.72,
        "pressure": 1012.72,
        "sea_level": 1032.43,
        "grnd_level": 1012.72,
        "humidity": 70,
        "temp_kf": 0.87
      },
      "weather": [
        {
          "id": 803,
          "main": "Clouds",
          "description": "broken clouds",
          "icon": "04d"
        }
      ],
      "clouds": {
        "all": 56
      },
      "wind": {
        "speed": 2.15,
        "deg": 73.0025
      },
      "sys": {
        "pod": "d"
      },
      "dt_txt": "2017-09-04 06:00:00"
    }
  ],
}
```

На первом уровне у нас словарь. В нем лежат вспомогательные данные:

- cod – код ответа, 200 – без ошибок;
- message – время ответа сервера;

- cnt – сокращение от count и означает количество элементов;
- list – массив с элементами, именно его будем парсить.

Сами объекты состоят из элементов:

- dt – дата в формате timestamp;
- main – словарь с основными данными:
 - tmp – температура в градусах Цельсия;
 - temp_min – минимальная температура;
 - temp_max – максимальная температура;
 - pressure – давление;
 - sea_level – уровень моря;
 - grnd_level – уровень земли;
 - humidity – влажность;
 - temp_kf – температурный коэффициент.
- weather – массив словарей с данными о погоде:
 - id – идентификатор погоды;
 - main – какая погода (облачно, ясно, гроза и т.д.);
 - description – описание;
 - icon – id иконки.
- clouds – словарь с данными о облаках:
 - all – плотность или другие данные об облачности.
- wind – словарь с данными о ветре:
 - speed – скорость в метрах в секунду;
 - deg – направление в градусах.
- sys – словарь с системными данными;
- dt_txt – текстовое представление даты и времени.

Не все данные из массива нам нужны. Подумаем, что нам может быть полезным: dt, tmp, pressure, humidity, weather – main, weather – icon, wind – speed, wind – deg. Создадим нужный класс. Для начала нам нужен новый файл **Weather.swift**. Прописываем в нем класс **Weather** с необходимыми полями.

```
class Weather {
    var date = 0.0
    var temp = 0.0
    var pressure = 0.0
    var humidity = 0
    var weatherName = ""
    var weatherIcon = ""
    var windSpeed = 0.0
    var windDegrees = 0.0
}
```

Возможно, у вас возникнет вопрос, почему мы сделали класс, а не структуру, и не использовали вложенные структуры для свойств weather и wind. Этот класс будет храниться в realm, поэтому нужен именно класс без вложенных структур.

Будем парсить json с помощью Codable. Есть сложность: в json-ответе погода состоит из вложенных объектов, а извлекать надо объект без вложенных свойств. Для этого будем использовать так называемый ручной режим извлечения. К сожалению, он требует так же много кода, как и JSONSerialization.

Создадим структуру WeatherResponse, она будет представлять верхнеуровневый объект json-ответа. Имплементируем протокол Decodable. Почему не Codable? Codable – два протокола. Decodable служит для извлечения данных и Encodable – для упаковки. Так как мы будем писать извлечение вручную, потребуется также вручную писать и упаковку, а она нам не нужна, так что имплементируем только один протокол и не будем писать лишний код.

```
class WeatherResponse: Decodable {
    let list: [Weather]
}
```

Теперь определим ключи для свойств погоды.

```
class Weather: Object, Decodable {
    dynamic var date = 0.0
    dynamic var temp = 0.0
    dynamic var pressure = 0.0
    dynamic var humidity = 0
    dynamic var weatherName = ""
    dynamic var weatherIcon = ""
    dynamic var windSpeed = 0.0
    dynamic var windDegrees = 0.0
    dynamic var city = ""

    enum CodingKeys: String, CodingKey {
        case date = "dt"
        case main
        case weather
        case wind
    }

    enum MainKeys: String, CodingKey {
        case temp
        case pressure
        case humidity
    }

    enum WeatherKeys: String, CodingKey {
        case main
        case icon
    }

    enum WindKeys: String, CodingKey {
        case speed
        case deg
    }
}
```

Мы создали перечисления для каждого уровня объектов в ответе. Теперь мы можем использовать их для ручного извлечения. Для этого необходим конструктор, определенный в протоколе.

```
convenience required init(from decoder: Decoder) throws {
    self.init()
}
```

В него передается decoder, из которого мы можем извлекать данные. Эти данные извлекаются в виде контейнеров, из которых уже можно забирать необходимые данные. Давайте для начала извлечем главный контейнер и дату, которая находится на самом высоком уровне.

```
let values = try decoder.container(keyedBy: CodingKeys.self)
self.date = try values.decode(Double.self, forKey: .date)
```

Контейнер извлекается с помощью метода container, при этом мы передаем ему перечисление с ключами, соответствующее данному уровню. Из контейнера мы можем извлечь значение, находящееся на этом же уровне вложенности. Когда извлекаем дату, указываем тип поля (Double) и значение перечисления для него.

Теперь извлечем следующий уровень.

```
let mainValues = try values.nestedContainer(keyedBy: MainKeys.self, forKey:
.main)
self.temp = try mainValues.decode(Double.self, forKey: .temp)
self.pressure = try mainValues.decode(Double.self, forKey: .pressure)
self.humidity = try mainValues.decode(Int.self, forKey: .humidity)
```

Используя контейнер предыдущего уровня, мы извлекаем нижестоящий контейнер методом nestedContainer и уже знакомым нам способом получаем из него значения, присваивая их необходимым свойствам.

Приступим к следующему уровню.

```
var weatherValues = try values.nestedUnkeyedContainer(forKey: .weather)
let firstWeatherValues = try weatherValues.nestedContainer(keyedBy:
WeatherKeys.self)
self.weatherName = try firstWeatherValues.decode(String.self, forKey: .main)
self.weatherIcon = try firstWeatherValues.decode(String.self, forKey: .icon)
```

Здесь есть определенные сложности: необходимые нам данные хранятся в массиве. Поэтому мы будем извлекать контейнер, не распаковывая его, с помощью метода nestedUnkeyedContainer. После можно извлекать элементы контейнера с помощью метода nestedContainer. Этот метод вызывается столько раз, сколько элементов в массиве. Необходим только первый, так что мы вызовем его один раз. Далее будем работать по уже известной схеме: извлечем необходимые свойства.

Полностью конструктор выглядит вот так:

```
convenience required init(from decoder: Decoder) throws {
    self.init()
    let values = try decoder.container(keyedBy: CodingKeys.self)
    self.date = try values.decode(Double.self, forKey: .date)

    let mainValues = try values.nestedContainer(keyedBy: MainKeys.self,
forKey: .main)
    self.temp = try mainValues.decode(Double.self, forKey: .temp)
    self.pressure = try mainValues.decode(Double.self, forKey: .pressure)
    self.humidity = try mainValues.decode(Int.self, forKey: .humidity)

    var weatherValues = try values.nestedUnkeyedContainer(forKey: .weather)
    let firstWeatherValues = try weatherValues.nestedContainer(keyedBy:
```

```

WeatherKeys.self)
    self.weatherName = try firstWeatherValues.decode(String.self, forKey:
.main)
    self.weatherIcon = try firstWeatherValues.decode(String.self, forKey:
.icon)

    let windValues = try values.nestedContainer(keyedBy: WindKeys.self,
forKey: .wind)
    self.windSpeed = try windValues.decode(Double.self, forKey: .speed)
    self.windDegrees = try windValues.decode(Double.self, forKey: .deg)

}

```

Теперь перейдем в класс `WeatherService` и изменим метод загрузки данных из интернета, чтобы парсить json.

```

// делаем запрос
Alamofire.request(url, method: .get, parameters: parameters).responseData {
reponse in
    guard let data = reponse.value else { return }
    let json = JSON(data: data)

    let weather = try! JSONDecoder().decode(WeatherResponse.self, from:
data).list
    print(weather)
}

```

Получим массив объектов `Weather`. Теперь их надо показать на экране. Ответ `Alamofire`, как и `URLSession`, обрабатывается асинхронно, параллельно основному потоку, так что, чтобы вытащить из **responseData**-замыкания данные, мы будем использовать замыкание. Замыкание мы передадим в метод получения данных **loadWeatherData**. После того, как получим данные, вызовем замыкание и передадим ему полученные данные.

```

func loadWeatherData(city: String, completion: @escaping ([Weather]) -> Void ){

    // 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).responseData { repsonse in
    guard let data = repsonse.value else { return }
    let weather = try! JSONDecoder().decode(WeatherResponse.self, from:
data).list

    completion(weather)
}

}
```

Теперь перейдем к WeatherViewController. Добавим свойство для хранения погодных данных.

```
// массив с погодой
var weathers = [Weather]()
```

Добавим в вызов метода замыкание и отобразим в нем полученные данные.

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

    // отправим запрос для получения погоды в Москве
    weatherService.loadWeatherData(city: "Moscow") { [weak self] weathers in
        // сохраняем полученные данные в массиве, чтобы коллекция могла получить
к ним доступ
        self?.weathers = weathers
        // коллекция должна прочитать новые данные
        self?.collectionView?.reloadData()
    }
}
```

Осталось научить коллекцию отображать полученные данные. Установим количество ячеек равным количеству элементов в массиве.

```
override func collectionView(_ collectionView: UICollectionView,
numberOfItemsInSection section: Int) -> Int {
    return weathers.count
}
```

Так как мы будем работать с классом Date, представляющим дату, и необходимо получать понятное ее представление, нужен специальный класс DateFormatter для такого преобразования. Определим его свойством, так как он нужен в одном экземпляре.

```
let dateFormatter = DateFormatter()
```

Изменяем метод подготовки ячейки.

```
override func collectionView(_ collectionView: UICollectionView, cellForItemAt
indexPath: IndexPath) -> UICollectionViewCell {
    let cell = collectionView.dequeueReusableCell(withReuseIdentifier:
```

```

"WeatherCell", for: indexPath) as! WeatherCell

    let weather = weathers[indexPath.row]

    cell.weather.text = "\(weather.temp) C"
    dateFormatter.dateFormat = "dd.MM.yyyy HH.mm"
    let date = Date(timeIntervalSince1970: weather.date)
    let stringDate = dateFormatter.string(from: date)
    cell.time.text = stringDate

    cell.icon.image = UIImage(named: weather.weatherIcon)
    return cell
}

```

Сначала получаем ячейку: это было реализовано. Затем получаем объект погоды для текущей ячейки. Устанавливаем градусы, используя свойство `temp`. Установим формат, в котором будем выводить дату. Мы используем шаблон из букв `dd/MM/yyyy/HH/mm` – он предназначен для дня/месяца/года/часа/минут. Полный список шаблонов можно посмотреть на сайте nsdateformatter.com. Также там можно проверить составленный шаблон. Используя свойство `date`, мы создаем экземпляр класса **Date**. Используя полученную дату и экземпляр класса `DateFormatter`, получаем строку из даты и устанавливаем ее для ячейки. Последним шагом мы используем свойство `weatherIcon`, чтобы достать иконку из набора, который добавили в проект на прошлом уроке.

Стоит уточнить, что этот код «с душком», как любят говорить программисты, то есть, он нуждается в рефакторинге. Обычно код конфигурации ячейки выносится или в отдельный метод контроллера, или же в саму ячейку. Рассмотрим второй вариант.

Откройте класс ячейки и для начала создайте статическое свойство для хранения **dateFormatter**. Так как экземпляр `dateFormatter` одинаков для всех строк, не стоит создавать его каждый раз заново, в статическом свойстве он будет создан один раз. Задаем значение этого свойства в замыкании. Это позволит создать объект форматтера, и сразу же сконфигурировать его.

```

static let dateFormatter: DateFormatter = {
    let df = DateFormatter()
    df.dateFormat = "dd.MM.yyyy HH.mm"
    return df
}()

```

Теперь создадим метод **configure** и перенесем в него код конфигурации из контроллера.

```

func configure(withWeather weather: Weather) {
    let date = Date(timeIntervalSince1970: weather.date)
    let stringDate = WeatherCell.dateFormatter.string(from: date)

    self.weather.text = String(weather.temp)
    time.text = stringDate
    icon.image = UIImage(named: weather.weatherIcon)
}

```

Осталось только вызвать метод конфигурирования в контроллере.

```

override func collectionView(_ collectionView: UICollectionView, cellForItemAt
indexPath: IndexPath) -> UICollectionViewCell {

```



```
        let cell = collectionView.dequeueReusableCell(withReuseIdentifier:
"WeatherCell", for: indexPath) as! WeatherCell

        cell.configure(withWeather: weathers[indexPath.row])

        return cell
    }
```

Практическое задание

На основе ПЗ предыдущего урока:

1. Спроектировать и написать классы User, Photo, Group для представления соответствующих данных приложения;
2. Изменить методы получения данных от VK API. Преобразовать json-ответ сервера в спроектированные классы.
3. На ранее созданных экранах выполнить необходимые запросы к API и отобразить полученные данные.

Дополнительные материалы

1. [Postman](#)

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>