Урок 4

FileManager

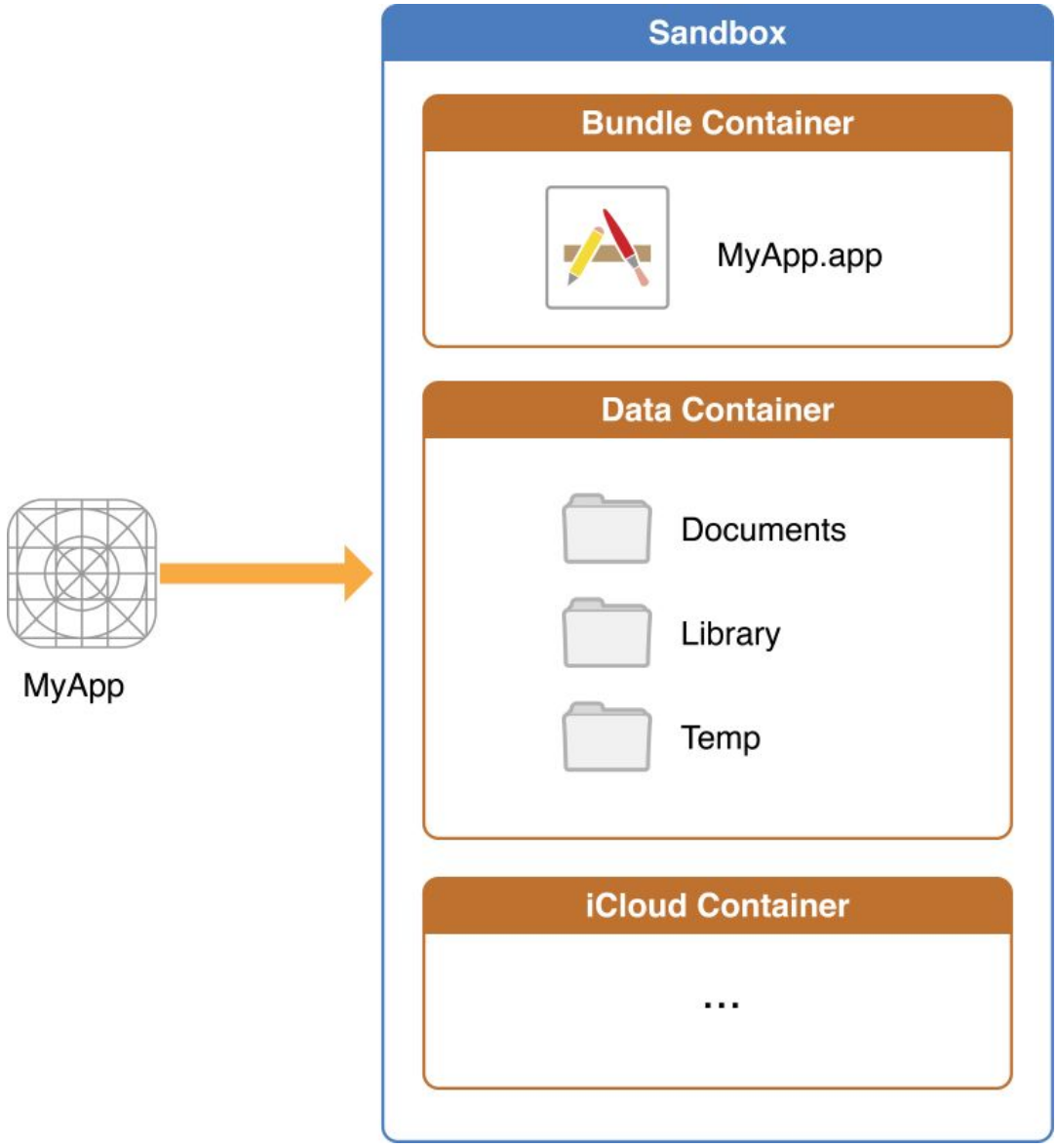
Разбираемся с файловой системой. Рассматриваем стандартные директории для сохранения файлов. Учимся сохранять изображения и другие документы.

Файловая система

Как и на компьютере, в смартфоне есть файловая система – место, где хранятся файлы. Главное отличие мобильной iOS от операционных систем персональных компьютеров заключается в том, что приложение не имеет доступа ко всем файлам. Ему предоставлен отдельный участок – «песочница», где оно может хранить данные. «Песочницы» делятся на контейнеры, а они, в свою очередь, на папки.

Существует три контейнера:

1. **BundleContainer** хранит исполняемый файл приложения, а также его ресурсы (xib-файлы, asset-каталог и прочее).
2. **DataContainer** хранит вспомогательные файлы приложения и пользователя.
3. **iCloudContainer** хранит данные, синхронизируемые с облаком.



В этом уроке рассмотрим работу с **DataContainer**, в котором хранятся пользовательские файлы. Этот контейнер делится на стандартные папки, которыми мы обязаны пользоваться.

1. **Documents** – содержит необходимые пользователю файлы и документы. Все, что лежит в этой папке, доступно пользователю и резервируется в iCloud.
2. **Library** – хранит данные, доступ к которым пользователю не нужен. Это файлы баз данных, кэш или другая информация, необходимая приложению. Здесь есть подпапки специального назначения.
3. **Tmp** – содержит временные файлы, которые не нужно сохранять после закрытия приложения. К примеру, файл, хранящийся на сервере, который необходимо скачать, изменить и отправить обратно. Перед завершением приложения файлы в этом каталоге нужно удалить. Но операционная система может и сама очищать временные файлы, когда приложение закрыто.

Вы можете создавать свои папки, но лучше использовать стандартные.

В папке **Documents** должны храниться файлы, необходимые пользователю. В графическом редакторе это могут быть изображения, в записной книжке – текстовые файлы с заметками, в аудиоредакторе – звуковые файлы, с которыми работает пользователь.

В папке **Tmp** хранятся файлы, необходимые во время текущего пользовательского сеанса. Например, если приложение записывает и отправляет аудиофайлы на сервер, на время записи и отправки их следует размещать в этой папке. Как только файл становится не нужен или работа с приложением завершается, его следует удалить. Папка Tmp периодически очищается операционной системой. Здесь не стоит хранить данные, необходимые в длительной перспективе.

В папке **Library/Caches** можно хранить кэш данных для ускорения работы приложения. Это может быть кэш базы данных или часто загружаемых из сети изображенный – например, аватар пользователя и его друзей в приложении социальной сети. Эта папка тоже может быть очищена операционной системой, если на устройстве недостаточно памяти. Приложение должно нормально работать без данных в этой папке, а при необходимости создавать их заново.

FileManager

Для работы с файловой системой существует специальный класс – **FileManager**. Он прост в использовании и позволяет создавать и удалять директории и файлы в них, а также получать информацию о существовании файла и его атрибуты.

Чтобы работать с файлом, необходимо получить путь к нему. Путь – это URL, уже знакомый нам по теме работы с сетью. URL, как правило, строится от системной части до каталога, в котором находится или будет расположен файл. Затем добавляется пользовательская часть URL, которая включает имя файла и подкаталоги, если они необходимы.

Получить стандартные пути можно так:

```
let cachesDirectory = FileManager.default.urls(for: .cachesDirectory, in:
.userDomainMask).first
let documentsDirectory = FileManager.default.urls(for: .documentDirectory, in:
.userDomainMask).first
let tmpDirectory = FileManager.default.temporaryDirectory

print(cachesDirectory)
print(documentsDirectory)
print(tmpDirectory)
```

```
Optional(file:///Users/jonfir/Library/Developer/CoreSimulator/Devices/5F6CA5C2-E57D-40C4-93C7-2D184D7725A0/data/Containers/Data/Application/74FDFB22-F2BB-40D7-9F1B-1FBBFE36CC18/Library/Caches/)

Optional(file:///Users/jonfir/Library/Developer/CoreSimulator/Devices/5F6CA5C2-E57D-40C4-93C7-2D184D7725A0/data/Containers/Data/Application/74FDFB22-F2BB-40D7-9F1B-1FBBFE36CC18/Documents/)
file:///Users/jonfir/Library/Developer/CoreSimulator/Devices/5F6CA5C2-E57D-40C4-93C7-2D184D7725A0/data/Containers/Data/Application/74FDFB22-F2BB-40D7-9F1B-1FBBFE36CC18/tmp/
```

Для некоторых каталогов есть альтернативные способы получить путь – правда, в формате строки, а не URL.

```
let documentsDirectory = NSHomeDirectory()
let tmpDirectory = NSTemporaryDirectory()

print(documentsDirectory)
print(tmpDirectory)
```

```
/Users/jonfir/Library/Developer/CoreSimulator/Devices/5F6CA5C2-E57D-40C4-93C7-2D184D7725A0/data/Containers/Data/Application/36BA3B19-E789-4C76-835A-EFCE320B8510

/Users/jonfir/Library/Developer/CoreSimulator/Devices/5F6CA5C2-E57D-40C4-93C7-2D184D7725A0/data/Containers/Data/Application/36BA3B19-E789-4C76-835A-EFCE320B8510/tmp/
```

Поработаем с папкой для временных файлов. Создадим URL документа: добавим к URL директории имя файла.

```
let tmpDirectory = FileManager.default.temporaryDirectory
let testFile = tmpDirectory.appendingPathComponent("TestFile.txt").path
```

Проведем с файлом ряд действий: проверим, существует ли он; создадим его; проверим еще раз; убедимся в его наличии.

```
var fileExists = FileManager.default.fileExists(atPath: testFile)
print("fileExists \(fileExists)")
let data = "Hello world!".data(using: .utf8)
FileManager.default.createFile(atPath: testFile, contents: data, attributes: nil)
fileExists = FileManager.default.fileExists(atPath: testFile)
print("fileExists \(fileExists)")
```

В выводе будет **false** и **true**, так как сначала файла не было, а потом мы его создали. Используя путь до файла, можно найти его, открыть и убедиться, что он содержит текст «Hello world!».

Мы записываем двоичные данные, поэтому данный способ годится для любых форматов. Сохраним изображение.

```
let testFile = tmpDirectory.appendingPathComponent("image.png").path
let data = UIImagePNGRepresentation(UIImage(named: "tree")!)
FileManager.default.createFile(atPath: testFile, contents: data, attributes:
nil)
```

Читаются файлы по-разному в зависимости от их типа. Текстовый файл можно прочитать так:

```
let tmpDirectory = FileManager.default.temporaryDirectory
let testFile = tmpDirectory.appendingPathComponent("TestFile.txt").path
do {
    let text = try String(contentsOfFile: testFile)
    print(text)
} catch {
    print(error)
}
```

Изображение читаем следующим образом:

```
let tmpDirectory = FileManager.default.temporaryDirectory
let testFile = tmpDirectory.appendingPathComponent("image.png").path
let image = UIImage(contentsOfFile: testFile)
```

Удаляем все файлы одинаково:

```
let tmpDirectory = FileManager.default.temporaryDirectory
let testFile = tmpDirectory.appendingPathComponent("image.png").path
do {
    try FileManager.default.removeItem(atPath: testFile)
} catch {
    print(error)
}
```

Внимание: удаление выполняется в блоке **do/catch** и может сгенерировать исключение. Если файла не существует или его нельзя удалить, получим ошибку.

Кэширование изображений

Чаще всего разработчики сталкиваются с задачей кэширования изображений из интернета. Оно позволяет не тратить время и ресурсы на повторную загрузку.

В зависимости от частоты использования данных можно выбрать разное время их кэширования. Если изображение понадобится только единожды, кэшировать его нет смысла. Если несколько раз, стоит сохранить его на пару часов. При постоянном использовании сохранять изображение надо навсегда.

Процесс кэширования состоит из операций запроса изображения, проверки наличия в кэше и, если файл найден, проверки актуальности кэша. Если файл отсутствует, его необходимо загрузить и поместить в кэш. Желательно выполнять все эти операции, не блокируя главный поток.

Для этого можно создать класс, где за каждый шаг будут отвечать конкретные методы. Они будут вызываться по цепочке. Чтобы запустить эти операции в глобальной очереди, можно использовать GCD.

Напишем сервис для загрузки, кэширования изображений, а также установки их в UITableView и UICollectionView с защитой от гонки состояний.

```
import Foundation
import Alamofire

class PhotoService {

    private let cacheLifeTime: TimeInterval = 30 * 24 * 60 * 60
    private static let pathName: String = {

        let pathName = "images"

        guard let cachesDirectory = FileManager.default.urls(for:
.cachesDirectory, in: .userDomainMask).first else { return pathName }
        let url = cachesDirectory.appendingPathComponent(pathName, isDirectory:
true)

        if !FileManager.default.fileExists(atPath: url.path) {
            try? FileManager.default.createDirectory(at: url,
withIntermediateDirectories: true, attributes: nil)
        }

        return pathName
    }()

    private func getFilePath(url: String) -> String? {

        guard let cachesDirectory = FileManager.default.urls(for:
.cachesDirectory, in: .userDomainMask).first else { return nil }

        let hasheName = String(describing: url.hashValue)
        return cachesDirectory.appendingPathComponent(PhotoService.pathName +
"/" + hasheName).path
    }

    private func saveImageToChache(url: String, image: UIImage) {
        guard let fileName = getFilePath(url: url) else { return }
        let data = UIImagePNGRepresentation(image)
        FileManager.default.createFile(atPath: fileName, contents: data,
attributes: nil)
    }

    private func getImageFromChache(url: String) -> UIImage? {
        guard
            let fileName = getFilePath(url: url),
            let info = try? FileManager.default.attributesOfItem(atPath:
fileName),
            let modificationDate = info[FileAttributeKey.modificationDate] as?
```

```
Date
            else { return nil }

        let lifeTime = Date().timeIntervalSince(modificationDate)

        guard
          lifeTime <= cacheLifeTime,
            let image = UIImage(contentsOfFile: fileName) else { return nil }


        images[url] = image
        return image
    }


    private var images = [String: UIImage]()

    private func loadPhoto(atIndexpath indexPath: IndexPath, byUrl url: String)
{
        Alamofire.request(url).responseData(queue: DispatchQueue.global()) {
[weak self] response in
            guard
                let data = response.data,
                let image = UIImage(data: data) else { return }

            self?.images[url] = image
            self?.saveImageToChache(url: url, image: image)
            DispatchQueue.main.async {
                self?.container.reloadRow(atIndexpath: indexPath)
            }

        }
    }

    func photo(atIndexpath indexPath: IndexPath, byUrl url: String) -> UIImage?
{
        var image: UIImage?
        if let photo = images[url] {
            image = photo
        } else if let photo = getImageFromChache(url: url) {
            image = photo
        } else {
            loadPhoto(atIndexpath: indexPath, byUrl: url)
        }
        return image
    }

    private let container: DataReloadable

    init(container: UITableView) {
        self.container = Table(table: container)
    }
```

```
    init(container: UICollectionView) {
        self.container = Collection(collection: container)
    }


}

fileprivate protocol DataReloadable {
    func reloadRow(atIndexpath indexPath: IndexPath)
}

extension PhotoService {

    private class Table: DataReloadable {
        let table: UITableView

        init(table: UITableView) {
            self.table = table
        }

        func reloadRow(atIndexpath indexPath: IndexPath) {
            table.reloadRows(at: [indexPath], with: .none)
        }

    }

    private class Collection: DataReloadable {
        let collection: UICollectionView

        init(collection: UICollectionView) {
            self.collection = collection
        }

        func reloadRow(atIndexpath indexPath: IndexPath) {
            collection.reloadItems(at: [indexPath])
        }
    }
}
```

Начнем его разбор.

```
private let cacheLifeTime: TimeInterval = 30 * 24 * 60 * 60
```

Для начала мы задали константу, она будет определять время в секундах в течении которого кэш считается актуальным. Обратите внимание на стиль которым она задана. Это умножение чисел - дней, часов, минут и секунд. Конечно можно было сразу написать результат этого умножения, но такой вариант сложно прочитать, а наш очень легко читается.

```
private static let pathName: String = {
```

```
        let pathName = "images"

        guard let cachesDirectory = FileManager.default.urls(for:
.cachesDirectory, in: .userDomainMask).first else { return pathName }
        let url = cachesDirectory.appendingPathComponent(pathName, isDirectory:
true)

        if !FileManager.default.fileExists(atPath: url.path) {
            try? FileManager.default.createDirectory(at: url,
withIntermediateDirectories: true, attributes: nil)
        }

        return pathName
    }()
```

Следующая конструкция это статическое свойство, имя папки в которой будут сохраняться наши изображения. Свойство инициируется с помощью замыкания. Помимо этого в замыкании происходит проверка, существует ли папка? Если не существует, она будет создана.

```
private func getFilePath(url: String) -> String? {

        guard let cachesDirectory = FileManager.default.urls(for:
.cachesDirectory, in: .userDomainMask).first else { return nil }

        let hasheName = String(describing: url.hashValue)
        return cachesDirectory.appendingPathComponent(PhotoService.pathName +
"/" + hasheName).path
    }
```

Метод getFilePath получает на вход URL изображения и возвращает на его основе путь к файлу для сохранения или загрузки. Обратите внимание, имя для файла мы получаем на основе его URL, от которого берется хэш. Сделано это потому что нам нужно некое уникальное имя, которое однозначно идентифицирует изображение, лучше всего подходит URL. А хэш гарантирует, что в имени не будет недопустимых символов.

```
    private func saveImageToChache(url: String, image: UIImage) {
        guard let fileName = getFilePath(url: url) else { return }
        let data = UIImagePNGRepresentation(image)
        FileManager.default.createFile(atPath: fileName, contents: data,
attributes: nil)
    }
```

Метод saveImageToChache сохраняет изображение в файловой системе.

```
private func getImageFromChache(url: String) -> UIImage? {
        guard
            let fileName = getFilePath(url: url),
            let info = try? FileManager.default.attributesOfItem(atPath:
fileName),
```

```
            let modificationDate = info[FileAttributeKey.modificationDate] as?
Date
            else { return nil }

        let lifeTime = Date().timeIntervalSince(modificationDate)

        guard
            lifeTime <= cacheLifeTime,
            let image = UIImage(contentsOfFile: fileName) else { return nil }


        images[url] = image
        return image
    }
```

getImageFromCache загружает изображение из файловой системы. При этом он проводит ряд проверок. Первым делом мы пытаемся получить атрибуты изображения FileManager.default.attributesOfItem. Этот метод вернет всю техническую информацию о файле, если он существует. Нас интересует дата последнего изменения. Если со времени изменения файла прошло больше секунд, чем указано в нашем свойстве cacheLifeTime, файл считается устаревшем и мы не будем заново загружать его из сети.

```
private var images = [String: UIImage]()
```

Свойство images это словарь в котором будут храниться загруженные и извлеченные из файловой системы изображения. Это кэш в оперативной памяти с минимальным временем доступа, специально для таблиц и коллекций, где очень важна скорость получения изображений.

```
private func loadPhoto(atIndexpath indexPath: IndexPath, byUrl url: String) {
        Alamofire.request(url).responseData(queue: DispatchQueue.global()) {
[weak self] response in
            guard
                let data = response.data,
                let image = UIImage(data: data) else { return }

            self?.images[url] = image
            self?.saveImageToChache(url: url, image: image)
            DispatchQueue.main.async {
                self?.container.reloadRow(atIndexpath: indexPath)
            }

        }
    }
```

Метод loadPhoto загружает фото из сети. Это обычный Alamofire-запрос на получение данных, он проходит в глобальной очереди, загружает изображение, сохраняет его на диске и в словаре images. Кроме того, после окончания загрузки он обновляет строку в таблице, чтобы отобразить загруженное изображение.

```
func photo(atIndexpath indexPath: IndexPath, byUrl url: String) -> UIImage? {
```

```
        var image: UIImage?
        if let photo = images[url] {
            image = photo
        } else if let photo = getImageFromChache(url: url) {
            image = photo
        } else {
            loadPhoto(atIndexpath: indexPath, byUrl: url)
        }
        return image
    }
```

Метод photo предоставляет изображение по URL. При этом мы ищем изображение сначала в кэше оперативной памяти, потом в файловой системе; если его нигде нет, загружаем из сети. IndexPath требуется, чтобы установить загруженное изображение в нужной строке, а не в ячейке, которая в момент, когда загрузка завершится, может быть использована для совершенно другой строки.

Теперь пришло время поговорить об установке изображения в ячейку таблицы. Как вы помните, ячейка != строка. Они переиспользуются и запросто может возникнуть такая ситуация, что мы начнем загружать изображение, когда ячейка используется для одной строки, а закончим, когда она будет задействована для другой; изображение будет не соответствовать строке. Чтобы предотвратить эту проблему, мы не будем создавать замыкание, устанавливающее ячейке изображение. Мы в конце загрузки обновим ячейку по indexpath. При обновлении нужная строка возьмет изображение из кэша; даже если в этот момент будет использована другая ячейка, ничего страшного не произойдет.

Для перезагрузки необходима ссылка на объект таблицы или коллекции, и здесь нас подстерегает первая сложность: у таблицы и коллекции разные классы и методы обновления элемента. Для унифицирования обновления мы создадим протокол и два объекта, в которые обернем коллекцию и таблицу.

```
fileprivate protocol DataReloadable {
    func reloadRow(atIndexpath indexPath: IndexPath)
}
```

DataReloadable – простой протокол декларирующий всего один метод – reloadRow. Именно его мы будем вызывать по окончанию загрузки.

```
private class Table: DataReloadable {
        let table: UITableView

        init(table: UITableView) {
            self.table = table
        }

        func reloadRow(atIndexpath indexPath: IndexPath) {
            table.reloadRows(at: [indexPath], with: .none)
        }

    }
```

Table – первый класс, имплементирующий протокол DataReloadable. В него будет заворачиваться UITableView. Класс достаточно простой, у него есть свойство для хранения ссылки на таблицу и конструктор, принимающий эту таблицу. Самым важным является реализация метода reloadRow, он вызывает обновление строки в таблице.

```
    private class Collection: DataReloadable {
        let collection: UICollectionView

        init(collection: UICollectionView) {
            self.collection = collection
        }

        func reloadRow(atIndexpath indexPath: IndexPath) {
            collection.reloadItems(at: [indexPath])
        }
    }
```

Класс Collection аналогичен классу Table, с той лишь разницей, что он хранит ссылку на экземпляр UICollectioView и обновляет ее элементы.

```
    private let container: DataReloadable

    init(container: UITableView) {
        self.container = Table(table: container)
    }

    init(container: UICollectionView) {
        self.container = Collection(collection: container)
    }
```

Теперь определяем свойство container для хранения обертки и два конструктора для установки таблицы или коллекции, в зависимости от того, что сейчас используется.

Создание клиента для сервиса openweathermap.org

В нашем погодном приложении иконки добавлены в каталог ресурсов. Начнем загружать их из интернета и кэшировать. Сначала добавим класс для кэширования и загрузки изображений в проект. Создадим файл **PhotoService** и поместим в него код, приведенный ранее.

Далее добавим вычисляемое свойство в класс **Weather** – оно будет возвращать URL.

```
var url: String {
        return "http://openweathermap.org/img/w/\(weatherIcon).png"
    }
```

Перейдем в класс **WeatherViewController** и добавим свойство для хранения сервиса.

```
var photoService: PhotoService?
```

В конце изменим метод подготовки ячейки.

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

        guard let weather = weathers?[indexPath.row] else { return cell }
        dateFormatter.dateFormat = "dd.MM.yyyy HH.mm"

        let date = Date(timeIntervalSince1970: weather.date)
        let stringDate = dateFormatter.string(from: date)

        cell.setWeather(text: "\(weather.temp) C")
        cell.setTime(text: stringDate)

        cell.icon.image = photoService?.photo(atIndexpath: indexPath, byUrl:
weather.url)

        return cell
    }
```

В результате получаем:

- загрузку изображений, которая не снижает отзывчивость приложения;
- кэш, который снижает нагрузку на канал связи и уменьшает время получения изображения;
- защиту от неопределенности, какой строке принадлежит загруженное изображение.

Практическое задание

1. Реализовать кэширование изображений в приложении.

Дополнительные материалы

1. Filesystem

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://developer.apple.com/documentation