Клиент-серверное приложение

Урок 7

Realm Notifications

Отслеживаем изменения в Realm. Автоматическое обновление UI.

Подписка на уведомление об изменении данных в хранилище

Как уже было сказано ранее, все, что вы получаете из Realm, будь то коллекция или один объект, является живым отражением данных в хранилище. Если меняются данные в хранилище, изменятся и объекты в приложении. Вы можете получить коллекцию из 30 элементов, которая может через несколько секунд стать пустой. Даже если рассматривать один объект, его данные могут быть изменены, более того, он может быть удален и обращение к этому объекту вызовет ошибку.

В такой ситуации становится очень сложно следить за данными. Тем более, что если данные из хранилища, отображающиеся на экране пользователя, изменились, необходимо мгновенно учесть это. Иначе пользователь будет работать с устаревшими данными.

Чтобы удержать изменения в узде, можно использовать специальные уведомления, которые предоставляет Realm.

Как работают уведомления? Имея на руках коллекцию, полученную из Realm, мы можем подписаться на уведомления о ее изменении. Как это работает? Как я уже говорил, коллекция, полученная из Realm, на самом деле не данные, а запрос к базе. Как только данные, удовлетворяющие запросу, меняются, приходит уведомление и выполняется код, связанный с этим уведомлением.

Давайте разберем процесс подписки на уведомление. Во-первых, при подписке на уведомление генерируется токен, специальная переменная, которая обозначает интерес к этому уведомлению. Пока токен существует, уведомления будут обрабатываться. Как только токен исчезнет, уведомление обрабатываться не будет. Важно хранить переменную в той области видимости, которая будет существовать, пока мы заинтересованы в получении уведомлений. Например, токен может быть свойством контроллера.

```
var token: NotificationToken?
```

Это свойство объявлено как опциональная и без начального значения, так как сам токен мы получим в момент подписки на уведомления.

Вторым шагом нам необходимо получить коллекцию из базы.

```
let dogs = realm.objects(Dog.self).filter("age > 5")
```

В примере выше коллекция содержит всех собак, чей возраст больше 5 лет, но вы могли бы получить коллекцию и без фильтра, особой разницы нет. После получения коллекции можно подписаться на получение уведомлений. Подписка добавляется в виде замыкания, переданного в метод **observe**. Результатом выполнения данного метода будет токен, который необходимо сохранить в созданном ранее свойстве.

```
self.token = dogs.observe {  (changes: RealmCollectionChange) in
            print("данные изменились")
        }
```

Если запустить данный код, мы увидим в консоли две строки «данные изменились». Возникает вопрос, почему мы изменили данные один раз, а уведомление пришло дважды. Дело в том, что

первый раз уведомление посылается при подписке на событие, а последующие разы – уже при изменении данных.

Как правило, нам мало знать, что данные изменились, важно еще и понять, что именно стало другим. Поэтому замыкание принимает в качестве аргумента **enum** с описанием изменения.

```
self.token = dogs.observe {  (changes: RealmCollectionChange) in
            switch changes {
            case .initial(let results):
                print(results)
            case let .update(results, deletions, insertions, modifications):
                print(results, deletions, insertions, modifications)
            case .error(let error):
                print(error)
            }
            print("данные изменились")
        }
```

В этом примере мы даем аргументу, переданному в замыкание, имя **change**. Как я уже сказал, это **enum** и самый простой способ узнать, какой он – написать **switch** с обработкой каждого значения. Вариантов всего три:

- .initial(let results) – срабатывает при подписке на исключение, имеет одно связанное значение с данными, которые содержатся в коллекции;
- .update(results, deletions, insertions, modifications) – срабатывает при изменении данных, имеет 4 связанных значения:
    - results – коллекция после изменения;
    - deletions – массив с числами, числа – индексы элементов, которые были удалены;
    - insertions – массив с числами, числа – индексы элементов, которые были добавлены;
    - modifications – массив с числами, числа – индексы элементов, которые были изменены.
- .error(let error) – срабатывает в случае ошибки, имеет одно связанное значение с текстом ошибки.

Имея на руках эту информацию, вы можете обновить интерфейс. Как правило, уведомления используются для изменения таблиц, но бывает, что и для элементов интерфейса.

Подписка на уведомление об изменении конкретного объекта

Кроме получения уведомлений на коллекцию, можно получать их на один объект. Процесс подписки на уведомления для одного объекта мало чем отличается от подписки на уведомление по коллекции. Для этого нам понадобится свойство для хранения токена, мы будем использовать метод **observe** и передавать ему замыкание.

Для примера создадим класс **StepCounter** с одним свойством с количеством шагов.

```
class StepCounter: Object {
    dynamic var steps = 0
}
```

Используем свойство типа токена, созданное ранее.

```
var token: NotificationToken?
```

И, собственно, сама подписка на уведомления.

```
let stepCounter = StepCounter()
        do {
            realm.beginWrite()


            realm.add(stepCounter)

            try realm.commitWrite()
        } catch {
            print(error)
        }

        token = stepCounter.observe { change in
            switch change {
            case .change(let properties):
                print(properties)
            case .error(let error):
                print("An error occurred: \(error)")
            case .deleted:
                print("The object was deleted.")
            }
        }

        do {
            realm.beginWrite()
            stepCounter.steps += 1
            try realm.commitWrite()
        } catch {
            print(error)
        }
```

Основное отличие от уведомления на коллекцию состоит в том, что в качестве аргумента в блок приходит другой **enum**. У него другие значения:

- .change(let properties) – срабатывает при изменении свойств объекта, имеет одно связанное значение с массивом кортежей. Каждый кортеж содержит три значения:
    - name – имя свойства объекта;
    - oldValue – значение свойства до изменения;
    - newValue – значение свойства после изменения.
- .deleted – срабатывает при удалении объекта из базы;
- .error(let error) – срабатывает в случае ошибки, имеет одно связанное значение с текстом ошибки.

С помощью этого типа уведомлений вы можете быть в курсе изменений конкретного объекта, например, получать информацию об изменении количества шагов, как в примере выше. При этом вы можете изменять надпись на экране.

Автоматическое обновление данных в UITableView

Самая распространенная ситуация при использовании уведомлений – обновление данных в таблице. Как мы помним, UITableView отображает данные из некоего источника, например, массива с объектами. Но мы можем указать в качестве источника данных коллекцию, полученную из **Realm**. Так как данные в коллекции постоянно меняются, мы легко можем получить ошибку выполнения программы в случае, если количество элементов уменьшится, или не отобразятся новые данные, если в коллекции появятся еще объекты.

Таблицу необходимо обновлять вручную, чтобы она считала новые данные с источника данных, но в какой момент это делать? Пытаться отслеживать изменения самостоятельно очень сложно, поэтому можно положиться на realm-уведомления.

Давайте возьмем первый пример с уведомлениями.

```
self.token = dogs.observe {  (changes: RealmCollectionChange) in
            switch changes {
            case .initial(let results):
                print(results)
            case let .update(results, deletions, insertions, modifications):
                print(results, deletions, insertions, modifications)
            case .error(let error):
                print(error)
            }
            print("данные изменились")
        }
```

Во-первых, коллекция **dogs** должна быть источником данных у UITableView.

```
func numberOfSections(in tableView: UITableView) -> Int {
        return 1
    }

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

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
-> UITableViewCell {

          let cell = tableView.dequeueReusableCell(withIdentifier: "SomeCell",
for: indexPath)

        cell.textLabel?.text = dogs?[indexPath.row].name ?? ""
```

```
        return cell
    }
```

Теперь нам надо обновлять таблицу при изменении данных.

Во-первых, изменим реакцию на получение первого уведомления. Так как до этого таблица была пустая, достаточно ее обновить целиком.

```
case .initial:
                self?.tableView.reloadData()
```

Самое интересно будет при обновлении данных в коллекции.

```
case .update(_, let deletions, let insertions, let modifications):
                self?.tableView.beginUpdates()
                  self?.tableView.insertRows(at: insertions.map({ IndexPath(row:
$0, section: 0) }),
                                           with: .automatic)
                   self?.tableView.deleteRows(at: deletions.map({ IndexPath(row:
$0, section: 0)}),
                                           with: .automatic)
                                 self?.tableView.reloadRows(at:  modifications.map({
IndexPath(row: $0, section: 0) }),
                                           with: .automatic)
                self?.tableView.endUpdates()
```

Мы не будем получать данные из коллекции, таблица и так с ней связана. Нам нужны только индексы строк, которые изменились. Все изменения находятся внутри двух методов – **self?.tableView.beginUpdates()** и **self?.tableView.endUpdates()**. Это сделано, чтобы изменения применялись не по очереди а одновременно, единой транзакцией. Каждая строка внутри транзакции – это изменение конкретных строк таблицы.

- insertRows добавляет в таблицу новые строки и перечитывает для них данные из коллекции;
- deleteRows удаляет строки из таблицы для тех элементов коллекции, которых больше нет;
- reloadRows читает из коллекции информацию для строк, данные для которых были изменены в коллекции.

Этот код достаточно универсален. И его можно использовать в любом контроллере вне зависимости от того, какие данные отображает ваша таблица.

После того, как вы написали его, вам больше не надо беспокоиться по поводу обновления данных в таблице, они будут меняться сами при изменении данных в базе.

Создание клиента для сервиса openweathermap.org

На этом занятии мы закончим разрабатывать приложение погоды. Во-первых, оно не сохраняет выбранные города, это надо исправить. Сделаем класс **City** для сохранения его в базу.

```
class City: Object {
    dynamic var name = ""
    let weathers = List<Weather>()

    override static func primaryKey() -> String? {
        return "name"
    }
}
```

Класс имеет два свойства – имя города и погода, которая была загружена для него. Теперь надо изменить контроллер **MyCityesController**, чтобы он отображал данные из Realm.

Добавим два свойства для контроллера. Одно будет хранить коллекцию городов, полученную из базы, второе – токен для уведомлений из базы.

```
var cityes: Results<City>?
 var token: NotificationToken?
```

Теперь напишем метод, чтобы получить из базы города и подписаться на уведомления о ее изменении.

```
func pairTableAndRealm() {
        guard let realm = try? Realm() else { return }
        cityes = realm.objects(City.self)
        token = cityes.observe { [weak self] (changes: RealmCollectionChange) in
            guard let tableView = self?.tableView else { return }
            switch changes {
            case .initial:
                tableView.reloadData()
            case .update(_, let deletions, let insertions, let modifications):
                tableView.beginUpdates()
                    tableView.insertRows(at: insertions.map({ IndexPath(row: $0,
section: 0) }),
                                         with: .automatic)
                     tableView.deleteRows(at: deletions.map({ IndexPath(row: $0,
section: 0)}),
                                         with: .automatic)
                  tableView.reloadRows(at: modifications.map({ IndexPath(row: $0,
section: 0) }),
                                         with: .automatic)
```

```
                tableView.endUpdates()
            case .error(let error):
                fatalError("\(error)")
            }
        }
    }
```

Этот метод мы вызовем в **viewDidLoad**.

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

        pairTableAndRealm()

    }
```

Изменим методы источника данных таблицы. Раньше они работали с массивом строк, а теперь – с коллекцией объектов, полученных из базы.

```
override func numberOfSections(in tableView: UITableView) -> Int {
        return 1
    }

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

    override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {

        //получаем ячейку из пула
        let cell = tableView.dequeueReusableCell(withIdentifier: "MyCityesCell",
for: indexPath) as! MyCityesCell
        //получаем город для конкретной строки
        let city = cityes[indexPath.row]

        //устанавливаем город в надпись ячейки
        cell.cityName.text = city.name

        return cell
    }
```

Мы выбирали города из таблицы, но логичнее предоставить пользователю возможность вводить города самому. Добавляться новые города будут через всплывающее окно. Такое окно показывается с помощью UIAlertController. Его необходимо создать, настроить в нем текстовое поле и добавить кнопки **Сохранить** и **Отмена**. Напишем метод, который будет создавать такое окно и показывать его.

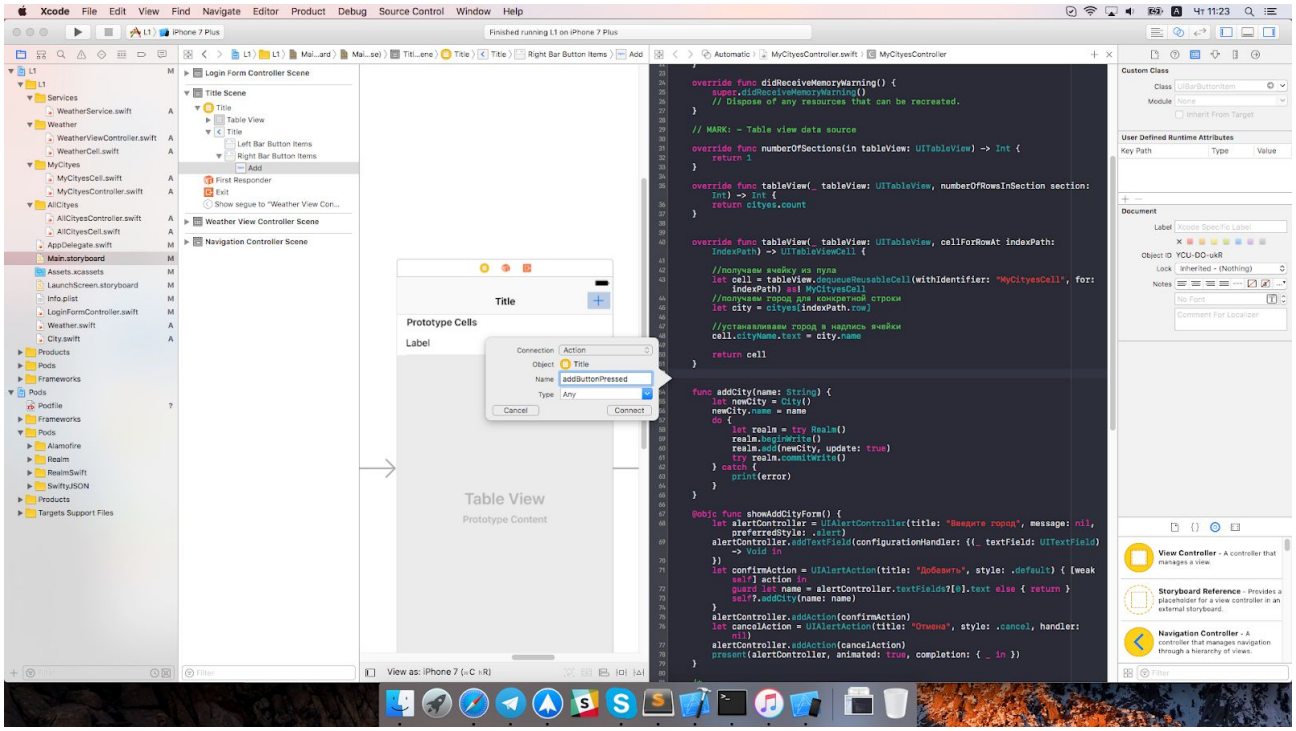
```
func showAddCityForm() {
        let alertController = UIAlertController(title: "Введите город", message:
nil, preferredStyle: .alert)
```

```
            alertController.addTextField(configurationHandler: {(_ textField:
UITextField) -> Void in
        })
         let confirmAction = UIAlertAction(title: "Добавить", style: .default) {
[weak self] action in
            guard let name = alertController.textFields?[0].text else { return }
            self?.addCity(name: name)
        }
        alertController.addAction(confirmAction)
            let cancelAction = UIAlertAction(title: "Отмена", style: .cancel,
handler: nil)
        alertController.addAction(cancelAction)
        present(alertController, animated: true, completion: { _ in })
    }
```

Когда пользователь нажимает **Добавить**, нам необходимо сохранить город в базу. Напишем специальный метод – **addCity**.

```
func addCity(name: String) {
        let newCity = City()
        newCity.name = name
        do {
            let realm = try Realm()
            realm.beginWrite()
            realm.add(newCity, update: true)
            try realm.commitWrite()
        } catch {
            print(error)
        }
    }
```

Теперь надо перейти в **main.storyboard** и удалить контроллер со всеми городами, а также добавить **@IBAction** для кнопки добавления города.



При нажатии на кнопку мы будем вызывать метод показа всплывающего окна.

```
@IBAction func addButtonPressed(_ sender: Any) {
        showAddCityForm()
    }
```

Можно запустить приложение и добавить несколько городов.

Теперь изменим метод удаления городов, так как в данный момент он абсолютно неработоспособен.

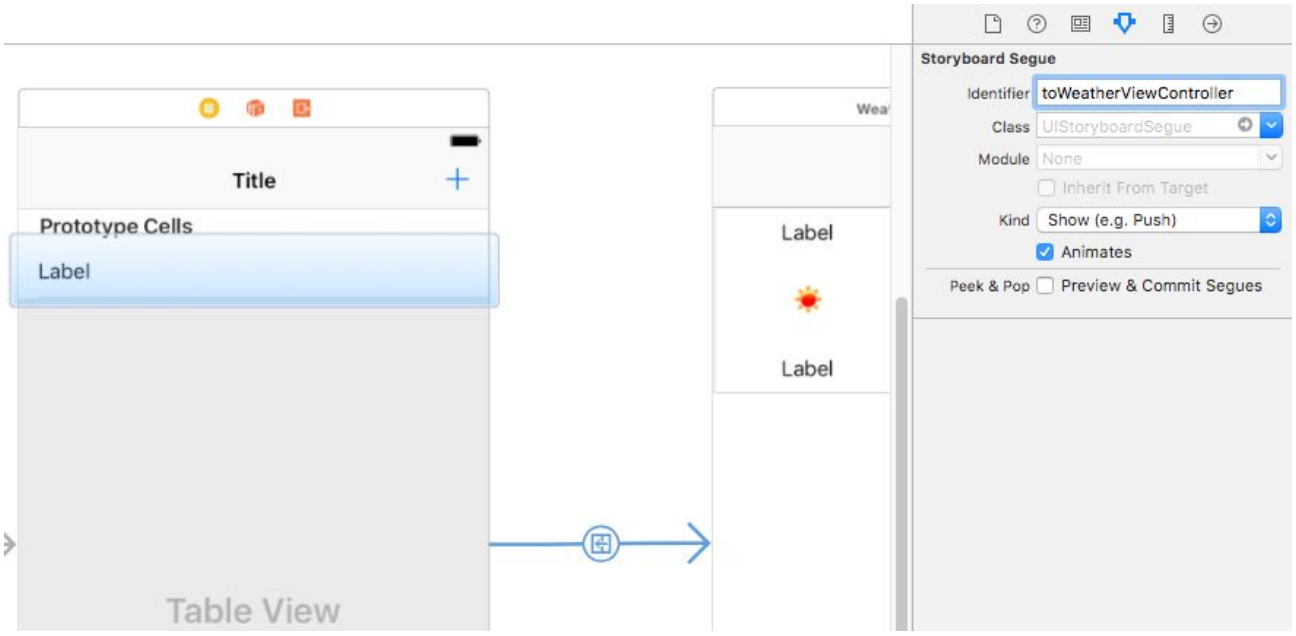
```
override func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
        let city = cityes[indexPath.row]
        if editingStyle == .delete {
            do {
                let realm = try Realm()
                realm.beginWrite()
                realm.delete(city.weathers)
                realm.delete(city)
                try realm.commitWrite()
            } catch {
                print(error)
            }
        }
    }
```

В нем вы мы получим объект города и удалим его из Realm. Обратите внимание, что ни при добавлении, ни при удалении городов мы не трогаем ни таблицу, ни коллекцию городов. Благодаря подписке на уведомления все изменяется автоматически.

Последнее, что осталось сделать с городами – удалить классы для контроллера, который мы убрали из **storyboard**. Найдите папку **AllCityes** и удалите ее.

Перейдем к модификации **WeatherViewController**. Сейчас он не содержит информации, какой город был выбран, мы загружаем данные всегда для Москвы. Давайте поправим это, передав в него выбранный город.

Откроем **storyboard** и зададим идентификатор для segue-перехода на **WeatherViewController**.



Теперь откроем **WeatherViewController** и добавим для него свойство с именем города.

```
var cityName = ""
```

Теперь вернемся на **MyCityesController** и изменим метод **prepare(for segue: UIStoryboardSegue, sender: Any?)**. Получим контроллер назначения и передадим ему имя выбранного города.

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if segue.identifier == "toWeatherViewController", let cell = sender as?
UITableViewCell {
            let ctrl = segue.destination as! WeatherViewController
            if let indexPath = tableView.indexPath(for: cell) {
                ctrl.cityName = cityes[indexPath.row].name
            }
        }
    }
```

Обратите внимание: мы передаем не сам объект города, а только его имя; по нему на втором контроллере мы сможем найти нужный город в базе.

Вернемся на **WeatherViewController** и изменим тип свойства **weathers**: он был массивом, а станет коллекцией Realm.

```
var weathers: List<Weather>!
```

Далее нам необходимо изменить класс **WeatherService**. Во-первых, изменим метод **saveWeatherData**. Будем не просто сохранять погоду, а связывать ее с городом.

```
func saveWeatherData(_ weathers: [Weather], city: String) {
// обработка исключений при работе с хранилищем
        do {
// получаем доступ к хранилищу
            let realm = try Realm()
// получаем город
                guard let city = realm.object(ofType: City.self, forPrimaryKey:
city) else { return }
// все старые погодные данные для текущего города
            let oldWeathers = city.weathers
// начинаем изменять хранилище
            realm.beginWrite()
// удаляем старые данные
            realm.delete(oldWeathers)
// кладем все объекты класса погоды в хранилище
            city.weathers.append(objectsIn: weathers)
// завершаем изменение хранилища
            try realm.commitWrite()
        } catch {
// если произошла ошибка, выводим ее в консоль
            print(error)
        }
    }
```

Удалим замыкание у метода **loadWeatherData**: оно нам больше не понадобится, о том, что новые погодные данные загружены, мы будем узнать, получая уведомления от Realm.

```
 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).responseData { [weak self] repsons in
            guard let data = repsons.value else { return }
            let weather = try! JSONDecoder().decode(WeatherResponse.self, from:
data).list
```

```
            weather.forEach { $0.city = city }

            self?.saveWeatherData(weathers, city: city)
        }

    }
```

Все готово для завершения погодного приложения. Вернемся к **WeatherViewController** и изменим загрузку погодных данных следующим образом:

```
override func viewDidLoad() {
        super.viewDidLoad()
// отправим запрос для получения погоды для Москвы
        weatherService.loadWeatherData(city: cityName)

    }
```

Мы удалили замыкание и вместо строкового литерала передаем свойство **cityName**. Напишем метод для подписки на уведомления об изменении погодных данных в городе. Обратите внимание, что методы работы с **collectionView** отличаются от методов работы с **tableView**, хотя и выполняют одни и те же действия.

```
func pairTableAndRealm() {
             guard let realm = try? Realm(), let city = realm.object(ofType:
City.self, forPrimaryKey: cityName) else { return }

        weathers = city.weathers

         token = weathers.observe { [weak self] (changes: RealmCollectionChange)
in
            guard let collectionView = self?.collectionView else { return }
            switch changes {
            case .initial:
                collectionView.reloadData()
            case .update(_, let deletions, let insertions, let modifications):
                collectionView.performBatchUpdates({
                                    collectionView.insertItems(at: insertions.map({
IndexPath(row: $0, section: 0) }))
                                     collectionView.deleteItems(at: deletions.map({
IndexPath(row: $0, section: 0)}))
                                collectionView.reloadItems(at: modifications.map({
IndexPath(row: $0, section: 0) }))
                }, completion: nil)
            case .error(let error):
                fatalError("\(error)")
            }
        }
    }
```

Добавим вызов этого метода при старте контроллера:

```
override func viewDidLoad() {
        super.viewDidLoad()
// отправим запрос для получения погоды для Москвы
        weatherService.loadWeatherData(city: cityName)
        pairTableAndRealm()
    }
```

На этом приложение готово. Мы можем добавлять и удалять города, получать погоду для нужного города и кэшировать ее в базу данных.

Практическое задание

На основе ПЗ предыдущего урока:

1. Добавить на все экраны с таблицами и коллекцией автоматическое обновление информации при изменении данных в Realm через **notifications**.

Дополнительные материалы

1. realm

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://realm.io