Урок 6

Realm

Детальный обзор Realm. Сохранение, извлечение, удаление, фильтрация данных

Realm

Модель

Связи

Запись

Чтение

Создание клиента для сервиса openweathermap.org

Практическое задание

Дополнительные материалы

Используемая литература

Realm

На прошлом уроке мы уже познакомились с Realm: записали данные в хранилище, но не извлекали их. На этом занятии подробнее изучим базу данных.

Сама база данных хранится в файловой системе телефона или (в случае с симулятором) на ПК. В рамках курса мы посмотрим на этот файл и научимся проверять, какие данные хранятся в нем.

В приложении для доступа к хранилищу мы используем объект класса **realm**. Именно через него мы получаем доступ к хранилищу. Как уже было сказано ранее таких объектов может быть создано сколько угодно.

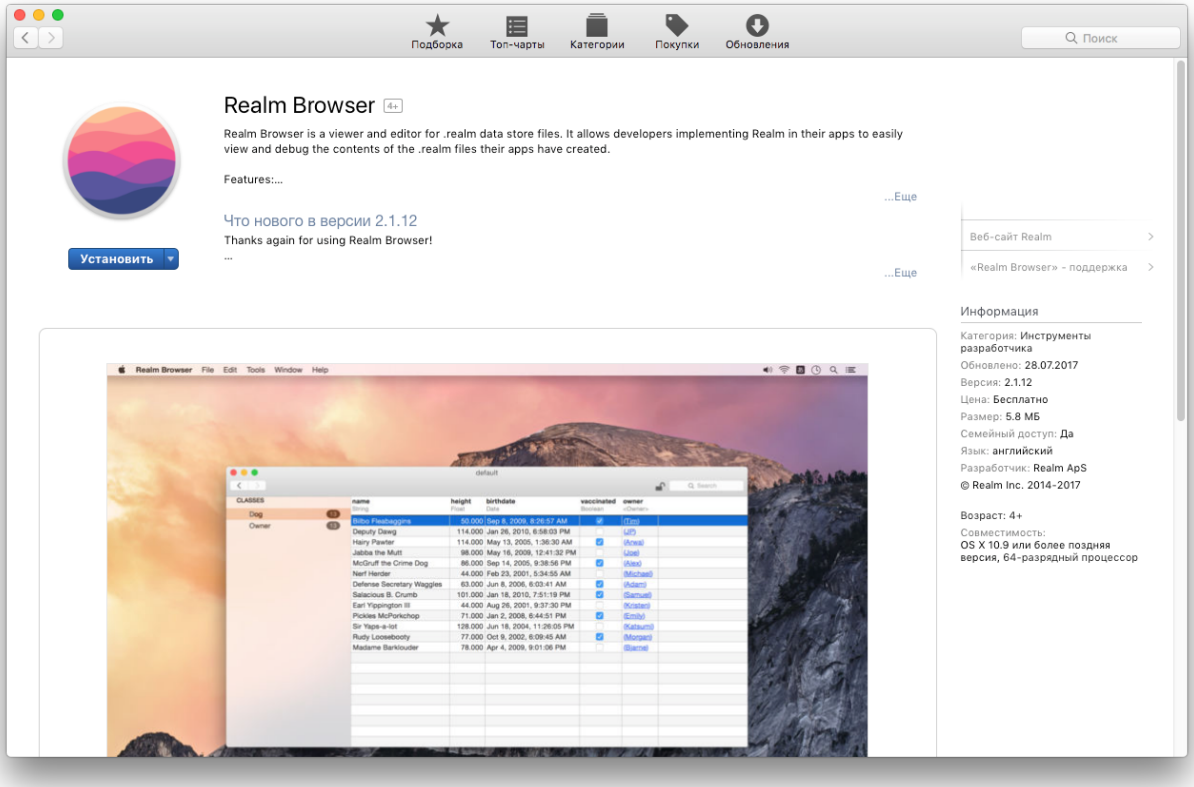
Давайте напишем простой пример. Создадим класс **TestEntyty** с одним свойством **name**.

```
class TestEntyty: Object {
    dynamic var name = ""
}
```

Создадим объект и запишем его в базу.

```
        let testEntyty = TestEntyty()
        testEntyty.name = "Иван"

        do {
            let realm = try Realm()
            realm.beginWrite()
            realm.add(testEntyty)
            try realm.commitWrite()

        } catch {
            print(error)
        }
```

Когда мы выполним этот код в первый раз, будет создан файл на диске и в нем – сущность с одним свойством **name**, которое равно **Иван**. Этот файл можно просмотреть, открыв в специальной программе **Realm Browser** (скачайте ее из App Store).



Но для того, чтобы открыть файл, необходимо его найти. Он находится в недрах вашего ПК, где-то в папке, созданной для симулятора. Чтобы узнать его местонахождение, можно в коде вывести в консоль путь к нему.

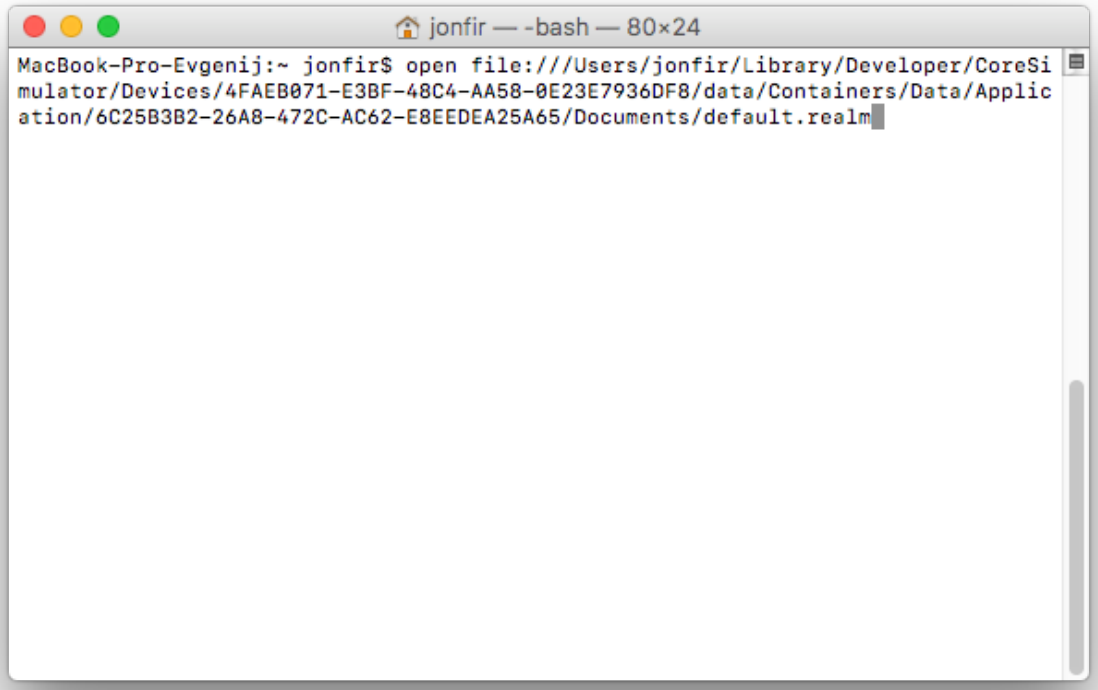
```
print(realm.configuration.fileURL)
```

```
Optional(file:///Users/jonfir/Library/Developer/CoreSimulator/Devices/4FAEB071-E3BF-48C4-AA58-0E23E7936DF8/data/Containers/Data/Application/6C25B3B2-26A8-472C-AC62-E8EEDEA25A65/Documents/default.realm)
```

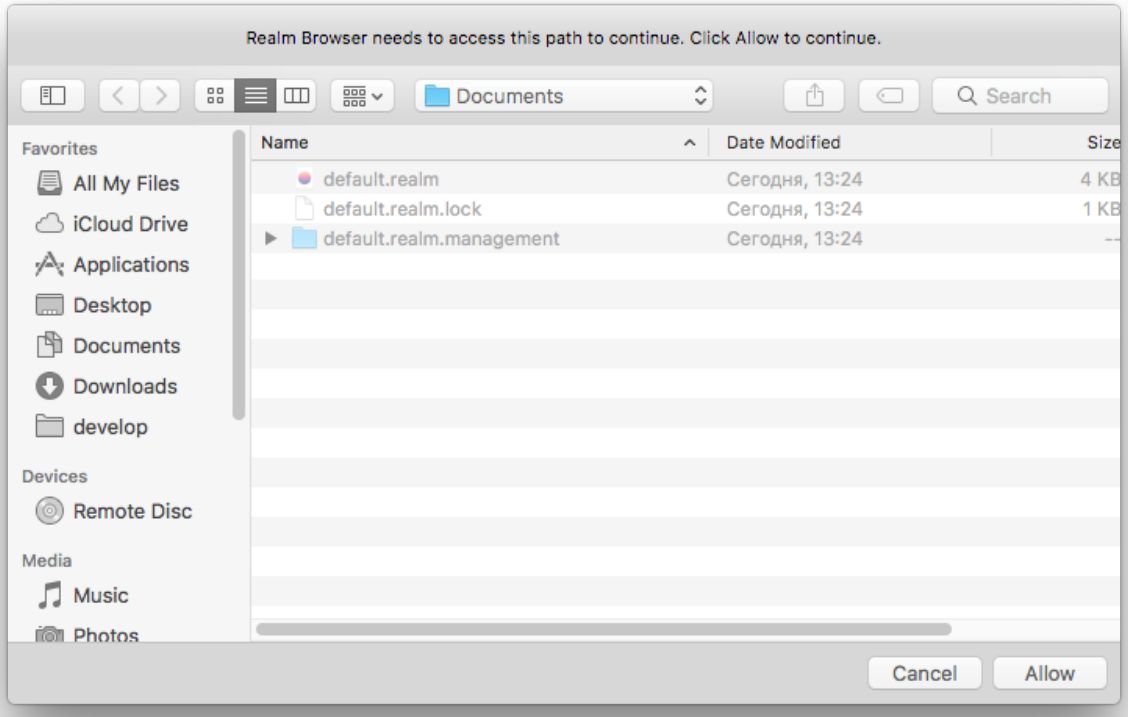
Эту строку необходимо скопировать. Обратите внимание, что значение находится внутри Optional, это копировать не надо.

```
file:///Users/jonfir/Library/Developer/CoreSimulator/Devices/4FAEB071-E3BF-48C4-AA58-0E23E7936DF8/data/Containers/Data/Application/6C25B3B2-26A8-472C-AC62-E8EEDEA25A65/Documents/default.realm
```

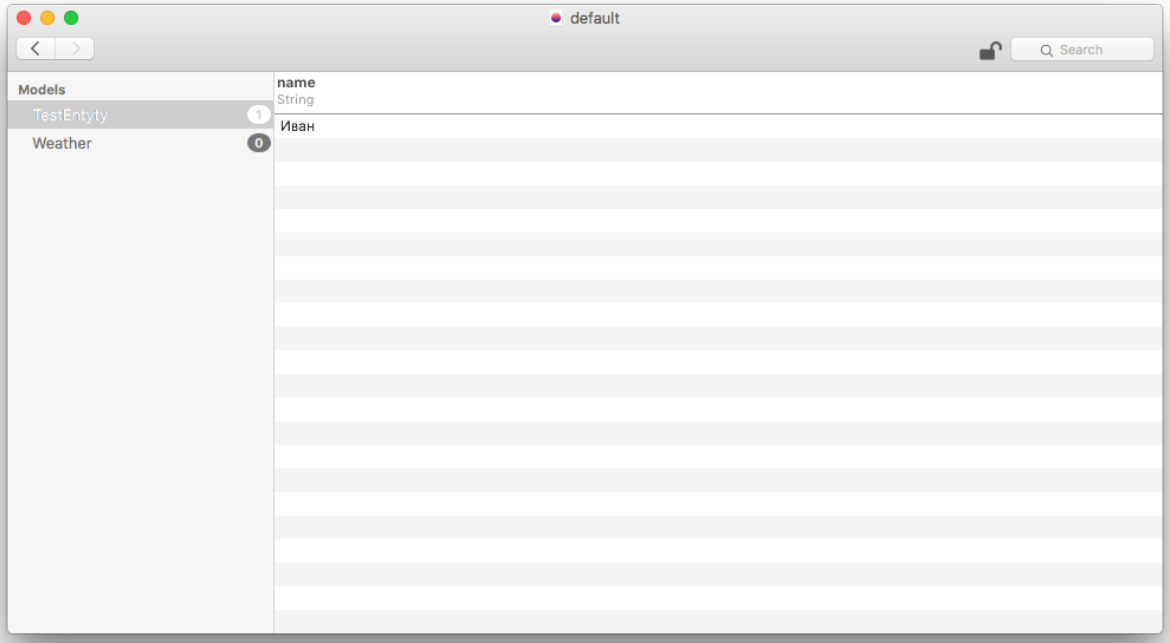
Теперь откроем терминал, напишем в нем **open** и вставим скопированную строку.



Выполняем команду: откроется realm browser и запросит подтверждение, что файл необходимо открыть.



Нажимаем **Allow** и наблюдаем базу данных.



Если у вас несколько записей с полем **Иван**, вы выполнили код сохранения несколько раз.

Сделаем пример интереснее: добавим больше полей нашей тестовой сущности.

```
class TestEntyty: Object {
    @objc dynamic var name = ""
    @objc dynamic var age = 0
    @objc dynamic var gender = true
    @objc dynamic var petName = ""
}
```

И заполним их для новой сущности – **Петр**.

```
        let testEntyty = TestEntyty()
        testEntyty.name = "Петр"
        testEntyty.age = 18
        testEntyty.gender = true
        testEntyty.petName = "Пушок"
        do {
            let realm = try Realm()
            print(realm.configuration.fileURL)
            realm.beginWrite()
            realm.add(testEntyty)
            try realm.commitWrite()

        } catch {
            print(error)
        }
```

Выполним этот код и получим ошибку.

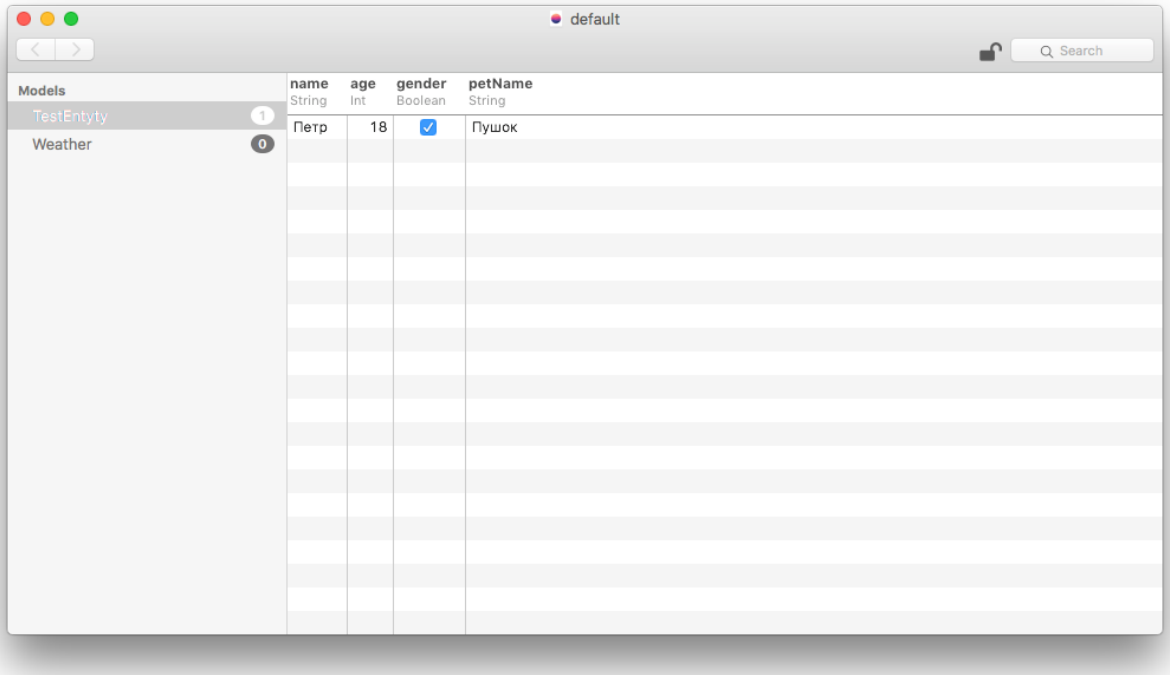
```
Error Domain=io.realm Code=10 "Migration is required due to the following errors:
- Property 'TestEntyty.age' has been added.
- Property 'TestEntyty.gender' has been added.
- Property 'TestEntyty.petName' has been added."
UserInfo={NSLocalizedDescription=Migration is required due to the following errors:
- Property 'TestEntyty.age' has been added.
- Property 'TestEntyty.gender' has been added.
- Property 'TestEntyty.petName' has been added., Error Code=10}
```

Ошибка означает, что сущность изменилась: теперь она содержит больше полей, но **realm** не знает, что делать с теми записями, которые находятся в базе, ведь у них нет этих полей. По-хорошему, необходимо написать миграцию, специальную инструкцию, как подготовить старые данные к новой форме. Но пока мы сделаем по-другому. Мы включаем специальный режим **realm**, в котором он, если не может изменить базу, будет ее просто удалять и создавать заново. Этот режим хорош для экспериментов.

```
Realm.Configuration.defaultConfiguration =
Realm.Configuration(deleteRealmIfMigrationNeeded: true)
```

Теперь закроем Realm Browser и запустим приложение еще раз. Если его не закрыть, удалить файл не удастся, и мы снова получим ошибку, но уже другую.

Когда код выполнится, мы повторим действия для открытия файла и увидим новую запись в базе.



Новая сущность имеет куда больше полей. Вы можете изменить их через браузер, но, как правило, это не требуется. Смотреть базу через браузер необходимо, только если вы не можете понять, проходит запись или нет.

Давайте еще немного поговорим про конфигурацию. **Realm** имеет много различных параметров, но, как правило, они не изменяются. В этом примере мы установили параметру **deleteRealmIfMigrationNeeded** значение **true**. В реальном приложение этот параметр никогда не трогают, так как при малейшей ошибке база данных будет удалена и создана заново, при этом данные пропадут, что недопустимо.

Конфигурация – это экземпляр класса **Realm.Configuration** и его необходимо передать **realm**. Можно установить его глобально, для всех **realm**, что вы создадите. Именно так мы и поступили в примере. Но можно сконфигурировать один конкретный экземпляр.

```
let config = Realm.Configuration(deleteRealmIfMigrationNeeded: true)
let realm = try Realm(configuration: config)
```

В примере выше мы установили настройку «удалять базу при ошибках». Но база будет удаляться только в случае, если мы обратимся к ней через этот экземпляр **realm**.

Модель

В качестве модели для **realm** используется обычный объект. При этом все свойства объекта становятся свойствами модели. На основе модели создаются записи в хранилище. Поэтому нельзя просто так менять свойства классов в **realm**: при изменении свойств меняется модель данных в хранилище и надо делать миграцию уже хранящихся там данных.

Классы для сохранения в приложении могут использоваться так же, как и обычные. В них могут быть методы и вычисляемые свойства, они не сохраняются в модели и не влияют на хранилище.

Простые модели мы уже научились создавать. Самая простая модель – это класс, унаследованный от Object с описанием свойств.

Но свойства не могут быть любыми. Есть всего 10 типов свойств, которые мы можем использовать. К тому же некоторые из них не могут быть опциональными, а некоторые опциональные свойства задаются не так, как у обычных классов. Самое простое при проектировании класса – воспользоваться шпаргалкой.

Type	Non-optional	Optional
Bool	`@objc dynamic var value = false`	`let value = RealmOptional<Bool>()`
Int	`@objc dynamic var value = 0`	`let value = RealmOptional<Int>()`
Float	`@objc dynamic var value: Float = 0.0`	`let value = RealmOptional<Float>()`
Double	`@objc dynamic var value: Double = 0.0`	`let value = RealmOptional<Double>()`
String	`@objc dynamic var value = ""`	`@objc dynamic var value: String? = nil`
Data	`@objc dynamic var value = Data()`	`@objc dynamic var value: Data? = nil`
Date	`@objc dynamic var value = Date()`	`@objc dynamic var value: Date? = nil`
Object	n/a: must be optional	`@objc dynamic var value: Class?`
List	`let value = List<Class>()`	n/a: must be non-optional
LinkingObjects	`let value = LinkingObjects(fromType: Class.self, property: "property")`	n/a: must be non-optional

В ней перечислены все типы и пример описания.

Кроме обычных свойств модель может содержать ключи. Ключи бывают разных типов. Основной тип ключей – **Primary key**, так называемый главный ключ сущности. Этим типом отмечается какое-либо свойство, после чего оно становится уникальным идентификатором модели. Значение в этих свойствах не могут повторяться. Другими словами, если пометить поле **name** как **Primary key**, **realm** не даст вам добавить в базу два разных объекта с именем **Иван**, с этого момента имя сущности в хранилище должно быть уникальным. Конечно бессмысленно делать уникальным имя, оно часто повторяется, как правило, ключом является специальное свойство – **id**.

Давайте добавим **id** в наш пример.

```
class TestEntyty: Object {

    @objc dynamic var name = ""
    @objc dynamic var age = 0
    @objc dynamic var gender = true
    @objc dynamic var petName = ""

    @objc dynamic var id = 0

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

Мы добавили свойство **id** типа **int** и переопределили функцию **primaryKey**. Данная функция возвращает строку с именем атрибута, который и будет **Primary key**.

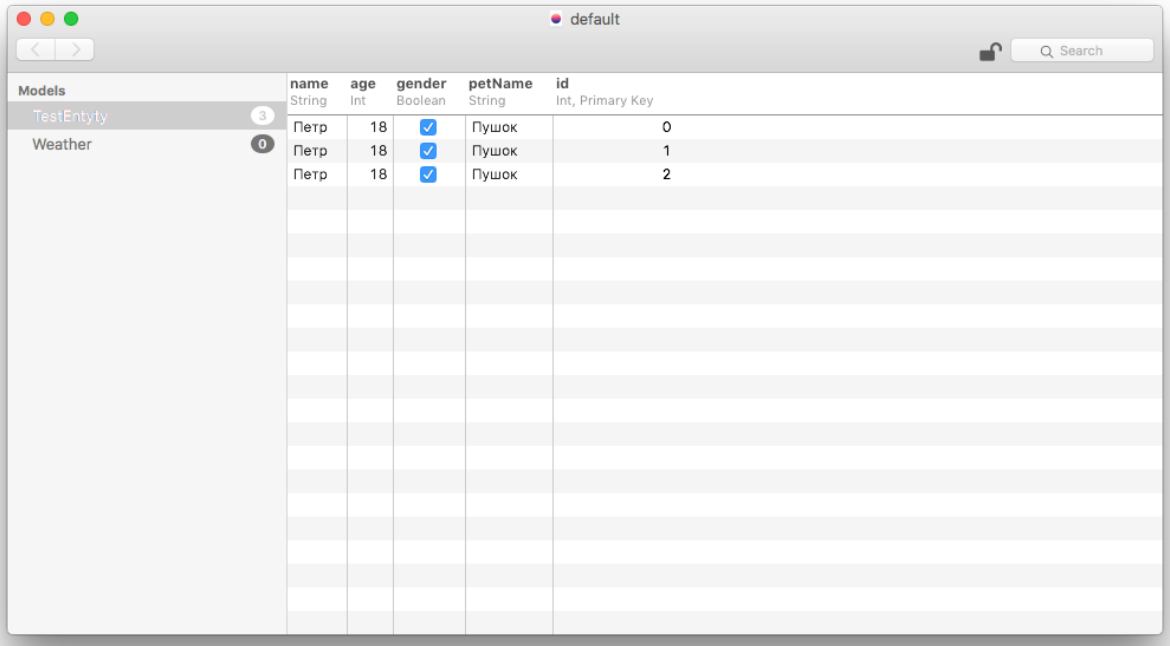
Изменим теперь пример для добавления в базу.

```
let testEntyty = TestEntyty()
        testEntyty.id = 0
        testEntyty.name = "Петр"
        testEntyty.age = 18
        testEntyty.gender = true
        testEntyty.petName = "Пушок"

        do {
            let realm = try Realm()
            realm.beginWrite()
            realm.add(testEntyty)
            try realm.commitWrite()

        } catch {
            print(error)
        }
```

Запустим этот пример несколько раз и получим ошибку.

```
Terminating app due to uncaught exception 'RLMException', reason: 'Attempting to
create an object of type 'TestEntyty' with an existing primary key value '0'.'
```

Realm сообщает, что мы пытаемся сохранить объект с уже имеющимся в базе ключом 0. Если изменить значение **id** на другое число, например 1, сохранение вновь пройдет.



Очень часто при получении данных из хранилища нам не нужны все данные, бывает, что мы ищем определенные значения. Например, мы часто ищем записи с определенным именем и именем животного. В таких случаях имеет смысл проиндексировать эти свойства. Добавление индексов осуществляется переопределением метода **indexedProperties**. Этот метод возвращает массив строк, где каждая строка – имя свойства, которое будет индексировано. На официальном сайте есть предупреждение, что индексы замедляют запись объектов в базу, но ускоряют выборку. Добавляйте записи только при необходимости.

```
class TestEntyty: Object {

    @objc dynamic var name = ""
    @objc dynamic var age = 0
    @objc dynamic var gender = true
    @objc dynamic var petName = ""

    @objc dynamic var id = 0

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

    override static func indexedProperties() -> [String] {
        return ["name"]
    }
}
```

Последнее, что необходимо знать о свойствах – их можно игнорировать в модели. Другими словами, у вашего класса могут быть свойства, которые не надо сохранять в хранилище. Например, если мы не желаем сохранять в базу пол, достаточно переопределить метод **ignoredProperties**. Этот метод возвращает массив строк, где каждая строка – имя свойства.

```
class TestEntyty: Object {

    @objc dynamic var name = ""
    @objc dynamic var age = 0
    @objc dynamic var gender = true
    @objc dynamic var petName = ""

    @objc dynamic var id = 0

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

    override static func indexedProperties() -> [String] {
        return ["name"]
    }

    override static func ignoredProperties() -> [String] {
        return ["gender"]
    }
}
```

Связи

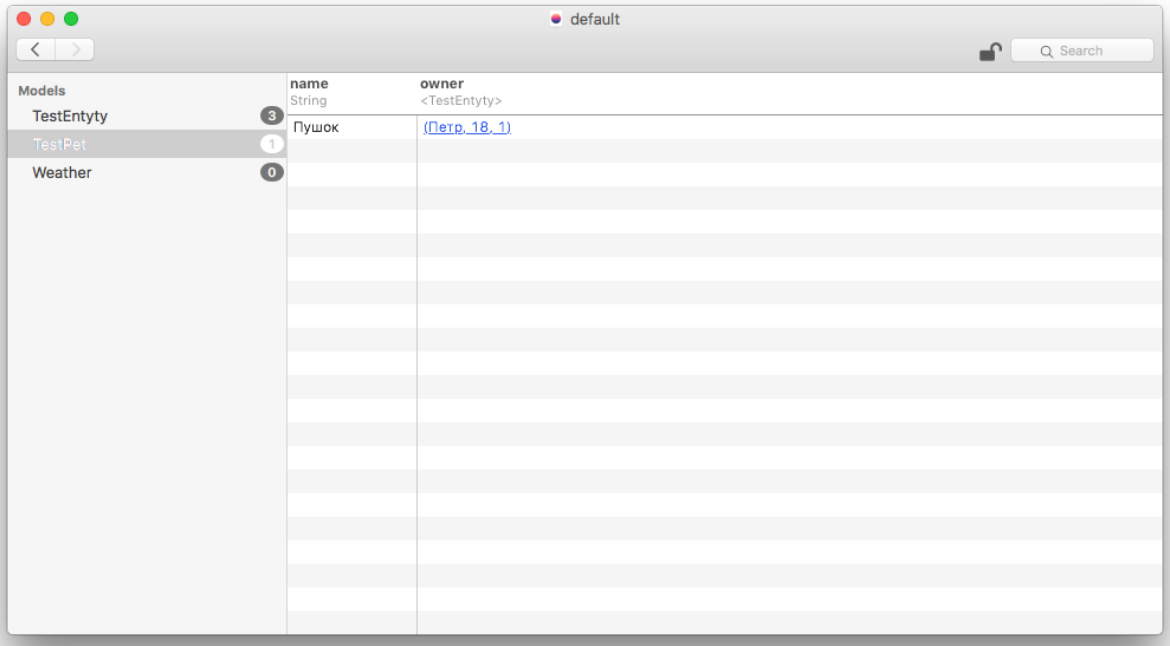
Сущности в хранилище могут быть связаны друг с другом через свойства. Изменим наш тестовый класс. Уберем из него свойство **petName** и создадим отдельный класс **TestPet**. Он будет олицетворять домашнего питомца. Теперь мы добавим самую простую связь – **многие к одному**. У нашего пета будет хозяин.

```
class TestEntyty: Object {

    @objc dynamic var name = ""
    @objc dynamic var age = 0
    @objc dynamic var gender = true

    @objc dynamic var id = 0

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

    override static func indexedProperties() -> [String] {
        return ["name"]
    }

    override static func ignoredProperties() -> [String] {
        return ["gender"]
    }
}
```

```
class TestPet: Object {
    @objc dynamic var name = ""
    @objc dynamic var owner: TestEntyty?
}
```

Теперь создадим объекты обоих классов. Объекту класса **TestPet** в свойство **owner** мы запишем объект класса **TestEntyty**.

```
let testEntyty = TestEntyty()
        testEntyty.id = 1
        testEntyty.name = "Петр"
        testEntyty.age = 18
        testEntyty.gender = true

        let pet = TestPet()
        pet.name = "Пушок"
        pet.owner = testEntyty

        do {
            let realm = try Realm()
            print(realm.configuration.fileURL)
            realm.beginWrite()
            realm.add(testEntyty)
            realm.add(pet)
            try realm.commitWrite()

        } catch {
            print(error)
        }
```



Теперь объекты связаны. Если мы извлекаем из базы объект домашнего животного, через свойство **owner** мы можем получить и тестовый объект.

Название связи **многие к одному** происходит из того факта, что многие объекты могут быть связаны с одним и тем же.

Realm поддерживает еще один тип записи – **многие ко многим**. В этом варианте многие объекты могут быть связаны со многими объектами.

Снова изменим примеры.

```
class TestEntyty: Object {

    @objc dynamic var name = ""
    @objc dynamic var age = 0
    @objc dynamic var gender = true

    @objc dynamic var id = 0

    let pets = List<TestPet>()

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

    override static func indexedProperties() -> [String] {
        return ["name"]
    }

    override static func ignoredProperties() -> [String] {
        return ["gender"]
```

```
    }
}

class TestPet: Object {
    @objc dynamic var name = ""
    let owners = LinkingObjects(fromType: TestEntyty.self, property: "pets")
}
```

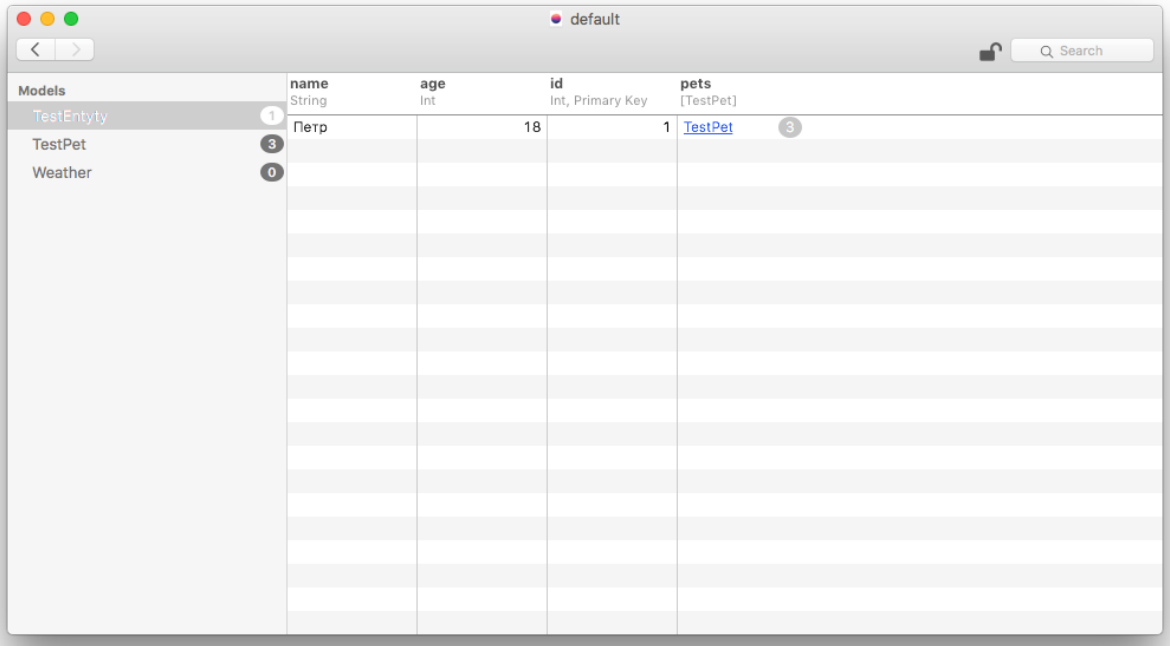
В данном варианте у класса **TestEntyty** есть свойство **let pets = List<TestPet>()**. Это коллекция, которая содержит связанные объекты. Это свойство и есть связь.

У класса **TestPet** есть свойство **let owners = LinkingObjects(fromType: TestEntyty.self, property: "pets")**, обеспечивающее обратную связь. Через него мы можем получить все объекты, содержащие этого питомца.

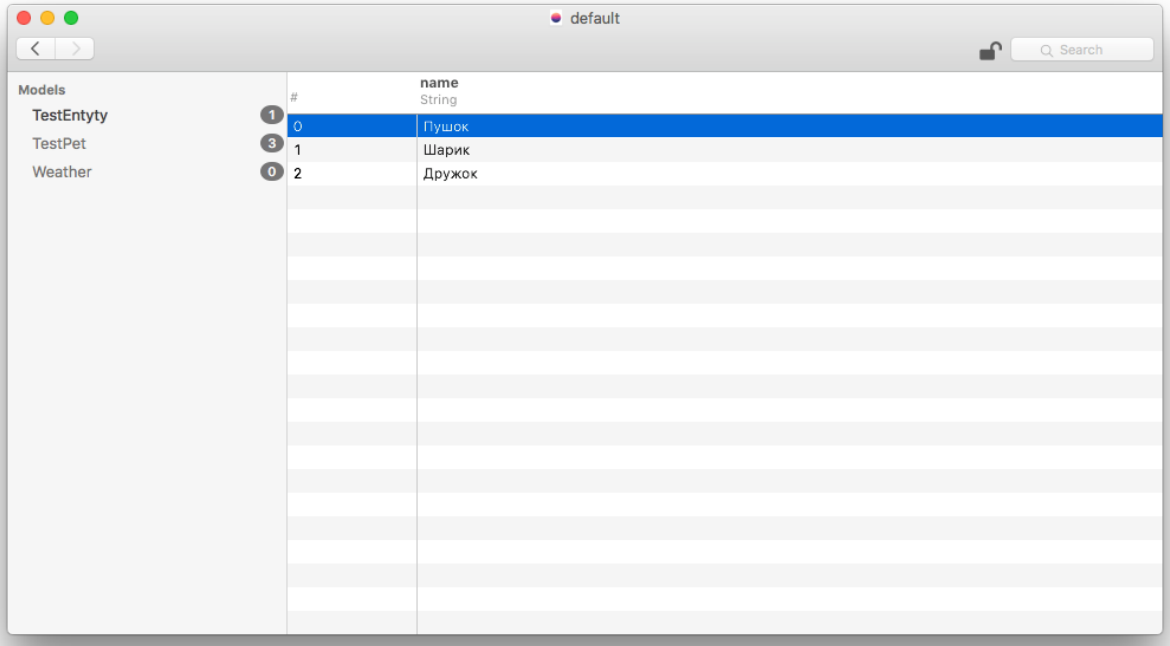
Изменим пример записи объектов в базу.

```
let testEntyty = TestEntyty()
        testEntyty.id = 1
        testEntyty.name = "Петр"
        testEntyty.age = 18
        testEntyty.gender = true

        let pet1 = TestPet()
        pet1.name = "Пушок"

        let pet2 = TestPet()
        pet2.name = "Шарик"

        let pet3 = TestPet()
        pet3.name = "Дружок"

        testEntyty.pets.append(pet1)
        testEntyty.pets.append(pet2)
        testEntyty.pets.append(pet3)

        do {
            let realm = try Realm()
            print(realm.configuration.fileURL)
            realm.beginWrite()
            realm.add(testEntyty)
            try realm.commitWrite()

        } catch {
            print(error)
        }
```

Теперь, если посмотреть в браузер, мы увидим эту связь и количество домашних животных.



Если нажмем на запись, увидим всех животных, принадлежащих ей.



Связи – очень полезный инструмент, который используется повсеместно. Как правило, все данные будут так или иначе связаны. Если рассматривать VK: у пользователей есть друзья – это связанные записи. У пользователей есть группы –это связанные записи. У пользователей есть фото –это тоже связанные записи.

Запись

Мы уже разбирали и записывали объекты в базу, так что этот раздел будет коротким. Кроме простого добавления данных в **realm**, существует еще два – запись с обновлением и запись коллекции.

Для начала разберем несколько способов создать объект для записи в базу. Классический способ – создать объект так же, как любой другой в Swift.

```
class Dog: Object {
    @objc dynamic var name = ""
    @objc dynamic var age = 0
}
```

```
var myDog = Dog()
        myDog.name = "Rex"
        myDog.age = 10
```

Этот способ рекомендуется как основной, но существуют и другие. Объект можно создать, используя конструктор на основе словаря. В этом варианте ключи являются именами свойств, а значения – значениями.

```
let myOtherDog = Dog(value: ["name" : "Pluto", "age": 3])
```

Можно пойти дальше и создать объект из массива.

```
let myThirdDog = Dog(value: ["Fido", 5])
```

В этом случае значения в массиве должны быть в той же последовательности, что и свойства в описании класса.

Вас могут удивить все эти возможности, особенно создание из массива, чьи значения имеют разный тип. Это противоречит философии Swift, но дело в том, что **realm** написан на Objective-C, и эти возможности уходят корнями именно туда.

После того, как объект создан, его необходимо добавить в хранилище. Самый простой способ мы разбирали: достаточно открыть сессию на запись и вызвать метод **add** у экземпляра **realm**. В результате объект будет записан.

```
do {
            let realm = try Realm()
            print(realm.configuration.fileURL)
            realm.beginWrite()
            realm.add(myDog)
            try realm.commitWrite()

        } catch {
            print(error)
        }
```

По одному объекты сохранять не очень удобно, можно сохранять их массив. Для этого достаточно передать массив в метод **add**.

```
do {
            let realm = try Realm()
            print(realm.configuration.fileURL)
            realm.beginWrite()
            realm.add([myDog, myOtherDog, myThirdDog])
            try realm.commitWrite()

        } catch {
            print(error)
        }
```

Также при сохранении объектов, содержащих связи с другими, связанные объекты автоматически сохраняются в базу.

Если объект имеет **Primary key**, мы можем не просто его записать в базу, а обновить уже имеющийся, в случае, если объект с таким же значением ключа существует. Для этого есть вариант метода **add** с аргументом **update**. Этот метод работает даже с массивами.

```
var myDog = Dog()
        myDog.name = "Rex"
        myDog.age = 10

        let myOtherDog = Dog(value: ["name" : "Pluto", "age": 3])
        let myThirdDog = Dog(value: ["Fido", 5])
        do {
            let realm = try Realm()
            print(realm.configuration.fileURL)
            realm.beginWrite()
            realm.add([myDog, myOtherDog, myThirdDog], update: true)
            try realm.commitWrite()

        } catch {
            print(error)
        }
```

Больше способов записать объект в хранилище нет. Только через метод **add** внутри сессии на запись. Если же вы создадите объект, но не запишете его в базу, он пропадает, как любой обычный объект Swift.

Кроме записи, вы, возможно, пожелаете изменить ряд свойств объекта, уже находящегося в хранилище. Для этого достаточно присвоить свойствам новые значения, но сделать это необходимо внутри сессии на запись, в противном случае вы получите ошибку.

```
do {
            let realm = try Realm()
            print(realm.configuration.fileURL)
            realm.beginWrite()
```

```
            myDog.name = "Пушок"
            try realm.commitWrite()

        } catch {
            print(error)
        }
```

Чтение

Чтобы получить все объекты определенного класса из базы, достаточно одной строчки.

```
let dogs = realm.objects(Dog.self)
```

В ответ вам придет специальная структура типа **Results**. **Results** очень похож на массив, объекты из него можно получать по индексу, его можно итерировать.

Основное отличие заключается в том, что **Results** не содержит данных. Фактически, это еще не исполненный запрос к базе. Исполняется он в момент, когда вы попытаетесь прочитать свойства конкретного объекта.

Очень важно понимать это свойство результатов выборки данных из базы. Мысль, что вы уже получили данные, может сыграть с вами злую шутку. Дело в том, что коллекция на самом деле всего лишь запрос к базе и количество ее элементов может изменить в любой момент.

```
let dogs = realm.objects(Dog.self)
        print(dogs.count)//0

        do {
            let realm = try Realm()
            realm.beginWrite()
            let newDog = Dog()
            newDog.name = "Пушок"
            newDog.age = 13
            realm.add(newDog)
            try realm.commitWrite()
        } catch {
            print(error)
        }

        print(dogs.count)//1
```

В примере выше мы получили из базы все объекты типа **Dog** и вывели в консоль количество этих элементов; в этом моменте их количество было 0. Затем мы создали собаку и добавили ее в базу. После чего мы вновь вывели в консоль количество элементов в коллекции **dogs**, и теперь в ней уже был один элемент. Самое важное: мы не трогали коллекцию и не проводили над ней манипуляций.

Часто нам нужны не все объекты определенного класса, а лишь их часть. В таком случае мы проводим фильтрацию по одному или нескольким полям.

```
var tanDogs = realm.objects(Dog.self).filter("color = 'tan' AND name BEGINSWITH 'B'")
```

В примере выше мы получаем всех собак цвета **tan** и именами, начинающимися с **B**.

Рассмотрим список доступных условий:

- ==, <=, <, >=, >, !=, и BETWEEN поддерживает типы Int, Int8, Int16, Int32, Int64, Float, Double and Date;
- ==, != сравнение для любых типов, которые можно сравнить;
- ==, !=, BEGINSWITH, CONTAINS, и ENDSWITH для строк (String);
- LIKE для строк, похож на CONTAINS, но вы можете использовать символы * и ? при сравнении. ? соответствует одному любому символу, * соответствует последовательности любых символов в любом количестве;
- Выражения можно объединить, используя операторы **AND**, **OR**, **NOT**;
- **IN** проверяет свойство на вхождение в коллекцию **name IN {‘Lisa’, ‘Spike’, ‘Hachi’}**.

Также данные можно сортировать. Здесь все предельно просто: указываете метод **sorted** и имя свойства, по которому необходимо отсортировать данные.

```
var tanDogs = realm.objects(Dog.self).sorted(byKeyPath: "age")
```

Создание клиента для сервиса openweathermap.org

На этом уроке мы будем доводить до ума работу с **realm** в приложении. Во-первых, мы расширим модель **Weather**, добавив в нее город.

```
import Foundation
import RealmSwift

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

    convenience init(json: JSON) {
        self.init()

        self.date = json["dt"].doubleValue
```

```
        self.temp = json["main"]["temp"].doubleValue
        self.pressure = json["main"]["pressure"].doubleValue
        self.humidity = json["main"]["humidity"].intValue
        self.weatherName = json["weather"][0]["main"].stringValue
        self.weatherIcon = json["weather"][0]["icon"].stringValue
        self.windSpeed = json["wind"]["speed"].doubleValue
        self.windDegrees = json["wind"]["deg"].doubleValue
        self.city = city

    }
}
```

После получения погоды от сервера и создания объектов мы будем указывать город, который мы получили как аргумент для запроса.

```
let weather = try! JSONDecoder().decode(WeatherResponse.self, from: data).list
weather.forEach { $0.city = city }
```

Метод для сохранения данных в базу мы тоже передадим **город**. Используя название города, получим все записи погоды для текущего города и удалим их: в текущей реализации при каждом запросе в сеть мы сохраняем полученные данные в базу и в ней копятся повторяющиеся данные. Теперь, перед тем как сохранить данные в базу, мы будем удалять из нее старые.

```
// сохранение погодных данных в realm
    func saveWeatherData(_ weathers: [Weather], city: String) {
// обработка исключений при работе с хранилищем
        do {
// получаем доступ к хранилищу
            let realm = try Realm()

// все старые погодные данные для текущего города
              let oldWeathers = realm.objects(Weather.self).filter("city == %@",
city)

// начинаем изменять хранилище
            realm.beginWrite()

// удаляем старые данные
            realm.delete(oldWeathers)

// кладем все объекты класса погоды в хранилище
            realm.add(weathers)

// завершаем изменение хранилища
            try realm.commitWrite()
        } catch {
// если произошла ошибка, выводим ее в консоль
            print(error)
        }
    }
```

Еще одно изменение в методе **loadWeatherData** будет касаться замыкания. Сейчас оно принимает в качестве аргумента погодные данные, но нам не надо их передавать, учитывая, что они находятся в базе.

```
// метод для загрузки данных, в качестве аргументов получает город
    func loadWeatherData(city: String, completion: @escaping () -> 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 { [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)

            completion()
        }
    }
```

Нам осталось переписать контроллер WeatherViewController для загрузки данных из базы. Мы напишем метод для получения данных.

```
func loadData() {
        do {
            let realm = try Realm()

                let weathers = realm.objects(Weather.self).filter("city == %@",
"Moscow")

            self.weathers = Array(weathers)

        } catch {
// если произошла ошибка, выводим ее в консоль
            print(error)
        }
    }
```

И вызовем его в замыкании, которое передается в метод загрузки данных из интернета.

```
// отправим запрос для получения погоды для Москвы
        weatherService.loadWeatherData(city: "Moscow") { [weak self] in

            self?.loadData()

            self?.collectionView?.reloadData()
        }
```

Практическое задание

На основе ПЗ предыдущего урока:

1. Организовать сохранение данных, полученных от VK API в Realm;
2. Все экраны теперь должны получать данные не от запросов к серверу, а из Realm;

Необходимо учесть, что данные на сервере все же меняются и их копию в Realm надо регулярно обновлять.

Дополнительные материалы

1. realm

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. realm.io