Клиент-серверное приложение

Урок 4

Хранение данных

Обзор инструментов и библиотек для постоянного хранения данных в приложении. UserDefaults, Файлы. CoreData. SQLite, Realm, Keychain

NSUserDefaults

Keychain

Физические файлы в песочнице приложения

База данных

SQLite

CoreData

Realm

Создание клиента для сервиса openweathermap.org

Практическое задание

Дополнительные материалы

Используемая литература

UserDefaults

Нам часто необходимо сохранить в приложении разного рода простые данные – настройки, информацию, что пользователь уже авторизовался. Можно сохранить id пользователя, залогиненного в системе, цветовую схему или предпочтительный адрес доставки. Если вы пишете программу для чтения книг, вы бы наверняка хотели сохранять позицию чтения, настройки шрифта и цвет фона.

Для хранения таких данных создан UserDefaults. Это хранилище содержит данные в виде пары ключ\значение. Ключ – всегда строка, а значение – простой тип данных. В качестве значения может быть строка, число, булево значение. При желании вы можете сохранить в него любой объект или массив объектов, но для этого придется добавлять в класс поддержку кодирования и декодирования в двоичные данные. Кроме того, не рекомендуется сохранять в него массивы с большим количеством данных. Если у вас есть какие-либо объекты, которые необходимо хранить в приложении, например, список товаров в интернет-магазине, лучше класть это в базу данных.

Работать с UserDefaults предельно просто: достаточно получить дефолтный объект для доступа к хранилищу.

```
let userDefaults = UserDefaults.standard
```

После чего можно сохранять и получать данные.

```
// сохраним строку в хранилище
        userDefaults.set("admin", forKey: "userName")

// получим строку из хранилища
        userDefaults.string(forKey: "userName") as? String
```

В примере выше первая строка сохраняет строку по ключу **username**. Вторая строка получает данные из хранилища. Собственно, на этом все, тонкость в том, что при получении данных необходимо выбрать верный метод, основываясь на типе значения.

Данные из хранилища будут доступны даже после перезапуска приложения, но если приложение будет удалено, а затем установлено заново, они будут удалены.

Keychain

В отличие от UserDefaults, это хранилище зашифровано. Если злоумышленник получит доступ к телефону, вскроет его и доберется до устройства хранения, он не сможет прочитать данные, сохраненные в keychain. В связи с особой защищенностью, в нем можно хранить приватные данные, например, логин, пароль или ключ для доступа к серверу.

Более того, информация, сохраненная в keychain, не теряется при удалении приложения с устройства. Вы можете сохранить информацию с логином и паролем пользователя, при переустановке приложения пользователю не придется авторизоваться повторно, данные можно прочитать из хранилища.

Однако работать с keychain намного сложнее, чем с UserDefaults. У него нет удобного API, вместо этого Apple предлагает использовать обертку, которую необходимо скачать с их сайта. Есть вариант лучше: использовать библиотеку. Таких библиотек великое множество, предлагаю начать с той, что

проще – SwiftKeychainWrapper. Ее можно установить, используя CocoaPods. С ней использование keychain станет намного проще.

```
// сохраним строку в хранилище
        KeychainWrapper.standard.set("Some String", forKey: "myKey")
// получим строку из хранилища
        KeychainWrapper.standard.string(forKey: "myKey")
```

Код выше является аналогом кода из раздела UserDefaults, только с сохранением данных в keychain.

Физические файлы в песочнице приложения

Еще один способ хранения данных в iOS – файловая система. В отличие от ПК и Android, она закрыта и приложение может работать только с файлами в специальном пространстве. У каждого приложения имеется свое пространство, куда только оно может записывать данные и после читать их. Такое пространство называется песочницей.

Обычно никто не хранит переменные в файлах. То есть вы не будете сохранять их в файл, для этого есть UserDefaults, Keychain, CoreData или какая-либо база данных. В файловую систему вы сохраняете какие-либо файлы: изображения, аудио или данные, полученные из интернета, чтобы не загружать их в следующий раз заново.

База данных

База данных (БД) – совокупность взаимосвязанных, хранящихся вместе данных.

Система управления базами данных (СУБД) – комплекс программных и языковых средств, необходимых для создания баз данных, поддержания их в актуальном состоянии и организации поиска в них необходимой информации.

Если говорить простым языком, база данных – просто данные, а вот то, что помогает хранить, создавать, удалять, получать данные и управлять ими в хранилище, называется СУБД. Как правило, когда говорят «база данных SQLite, MySQL, Realm», имеют в виду именно СУБД. В приложении вы общаетесь с СУБД для взаимодействия с БД. Кроме того, бывают различные библиотеки, которые упрощают взаимодействие с СУБД, обертки над стандартными методами. Есть настолько продвинутые библиотеки, что они могут могут полностью скрыть от программиста подробности работы с СУБД. Более того, программист может поменять СУБД, но его код через библиотеку продолжит работать с ней, как будто ничего не изменилось. Как правило, такие библиотеки предоставляют методы для работы в той парадигме, в которой разрабатывается код, например, ООП. Такие библиотеки называются ORM.

Вы можете использовать в приложении на выбор несколько баз данных, например, SQLite, Realm или поискать на GitHub что-то менее популярное. Но, как правило, выбирают из этих двух.

SQLite

Это одна из самых простых СУБД. Компактная с минимумом функционала, она была создана для использования на конечных устройствах, а не серверах, как другие ее собратья. Это реляционная

база данных: данные хранятся в виде таблицы. Одна таблица представляет собой одну сущность, например, паспорт. Записи в этой таблице представляют экземпляры сущности. Таблицы могут содержать связи, так, например, таблица с данными о людях может быть связана с таблицей, хранящей данные о паспортах. Для запросов к этой СУБД используется SQL.

Ключевой особенностью использования этой базы данных является высокая гибкость. Вы можете проектировать сущности как вам угодно, создавать любые связи. Используя гибкий язык запросов, вы сможете получать любые данные.

Но за эту гибкость приходится платить: вам придется писать все запросы самостоятельно. В программе вы оперируете объектами, а в СУБД – простым набором данных. Вам придется самим писать преобразование объектов в данные, понятные СУБД. Это очень много кода и лишней работы.

CoreData

Это не база данных, как многие думают, а ORM. Она может работать поверх SQLlite, xml или хранить данные в памяти. Вы работает с ней, не задумываясь, что у нее под капотом.

CoreData – замечательный инструмент, но есть и недостатки: она не дает той гибкости, что дает чистый SQLite и работает медленнее. Для большинства приложений это не критично и CoreData выглядит как вполне разумное решение.

CoreData состоит из нескольких компонентов:

- managed object model (управляемая объектная модель) – фактически, это ваша модель (в парадигме MVC), которая содержит все сущности, их атрибуты и взаимосвязи;
- persistent store coordinator (координатор постоянного хранилища) – посредник между хранилищем данных и контекстом, в которых эти данные используются, отвечает за хранение данных и их кэширование;
- managed object contexts (контекст управляемого объекта) – используется для управления коллекциями объектов модели (в общем случае может быть несколько контекстов).

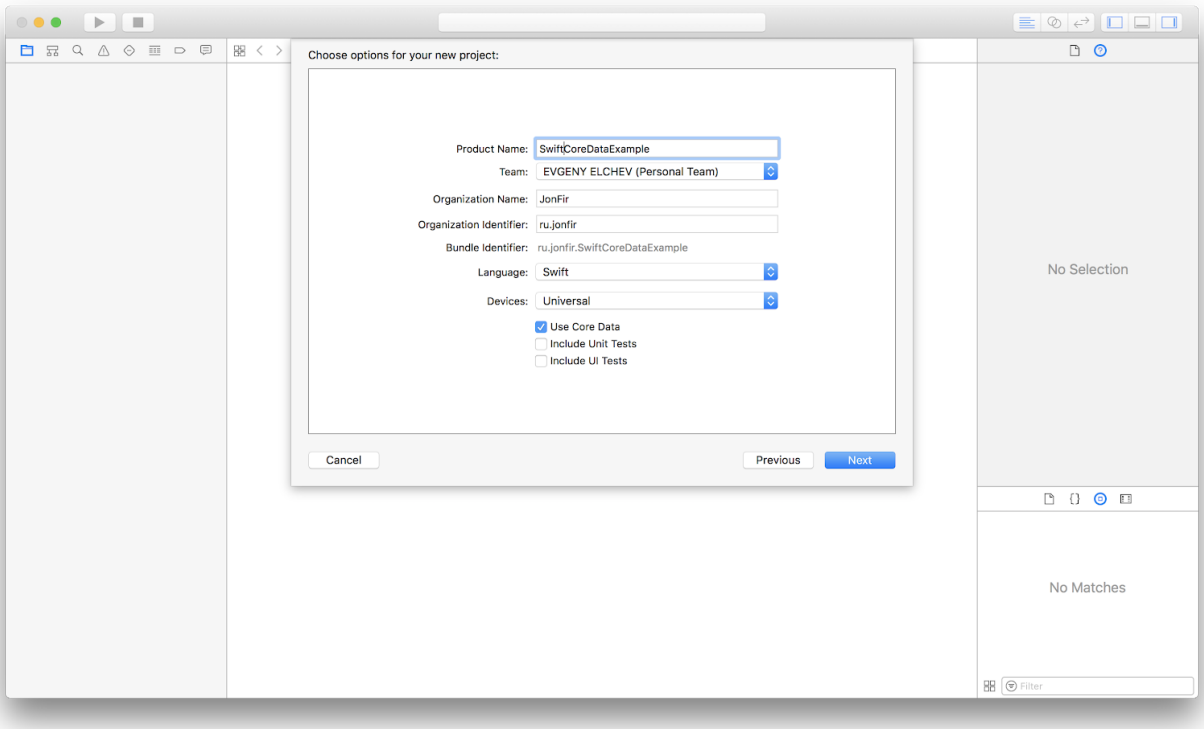
Работа с базой начинается с модели, в ней вы создаете сущности для хранения в базе. Вы описываете все поля, которые необходимо хранить. Это очень похоже на проектирование объекта, но в качестве полей нельзя использовать любые типы. Например, вы не можете хранить в свойстве массив, для этого необходимо создать еще одну сущность и связь между ними.

После того, как вы определите объектную модель, необходимо создать координатор, который будет преобразовывать данные из хранилища на основе модели в объекты. Раньше это было непросто и, как правило, все брали уже готовую настройку из интернета или использовали свои наработки. Но, начиная с iOS 10, все изменилось и настроить координатор можно в пару строк кода.

Имея координатор, можно создавать контекст. Контекст – копия вашего хранилища в памяти, таких контекстов может быть несколько. Вы можете работать с ним, не изменяя самого хранилища. После того, как вы произведете в нем необходимые изменения, вы можете объединить несколько контекстов и сохранить их в хранилище.

Мы будем изучать с вами Realm, но давайте создадим небольшой пример использования CoreData.

Для начала создадим новый проект. Выберем **Single View Application** и на шаге выбора имени для проекта поставьте галочку **Use Core Data**.

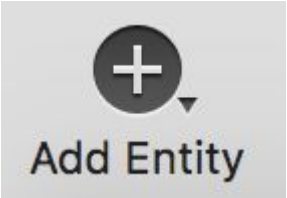


В файле **AppDelegate** будет создан код для работы с CoreData, он не самый оптимальный, но подойдет.

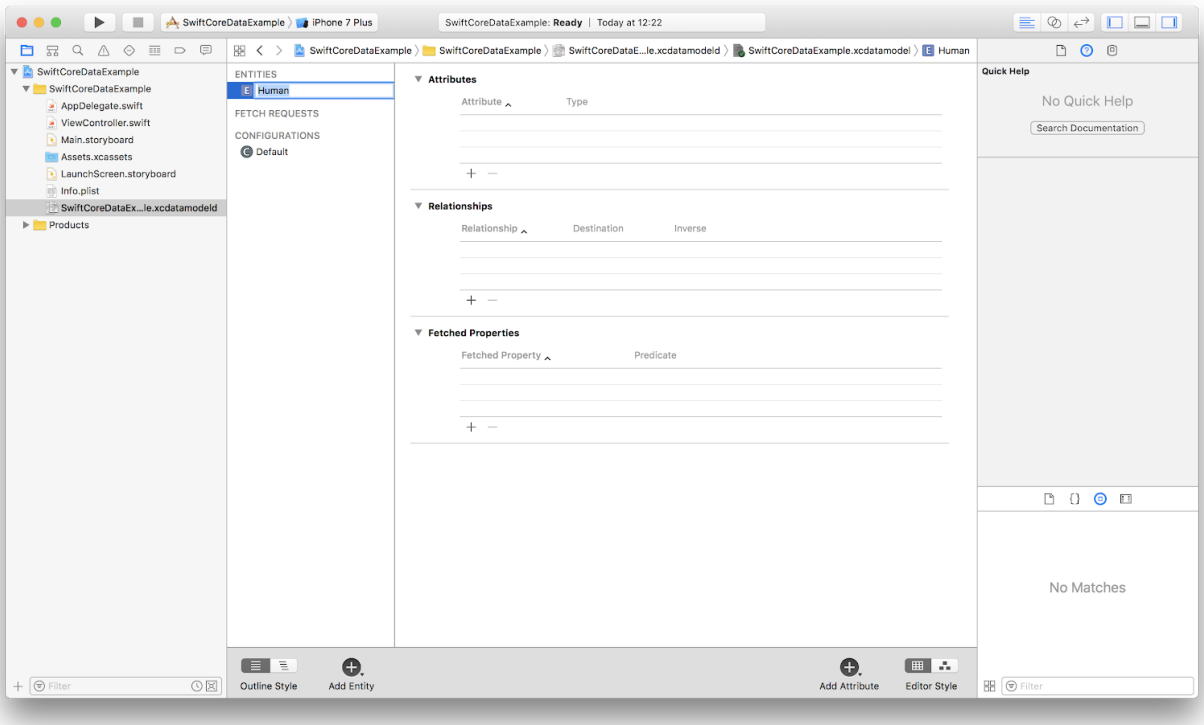
```
// MARK: - Core Data stack

    lazy var persistentContainer: NSPersistentContainer = {
        /*
         The persistent container for the application. This implementation
         creates and returns a container, having loaded the store for the
         application to it. This property is optional since there are legitimate
         error conditions that could cause the creation of the store to fail.
        */
        let container = NSPersistentContainer(name: "SwiftCoreDataExample")
          container.loadPersistentStores(completionHandler: { (storeDescription,
error) in
            if let error = error as NSError? {
                // Replace this implementation with code to handle the error
appropriately.
             // fatalError() causes the application to generate a crash log and
terminate. You should not use this function in a shipping application, although
it may be useful during development.

                /*
                 Typical reasons for an error here include:
                   * The parent directory does not exist, cannot be created, or
disallows writing.
                 * The persistent store is not accessible, due to permissions or
data protection when the device is locked.
                 * The device is out of space.
                 * The store could not be migrated to the current model version.
```

```
                    Check the error message to determine what the actual problem
was.
                 */
                fatalError("Unresolved error \(error), \(error.userInfo)")
            }
        })
        return container
    }()

// MARK: - Core Data Saving support

    func saveContext () {
        let context = persistentContainer.viewContext
        if context.hasChanges {
            do {
                try context.save()
            } catch {
// Replace this implementation with code to handle the error appropriately.
// fatalError() causes the application to generate a crash log and terminate.
You should not use this function in a shipping application, although it may be
useful during development.
                let nserror = error as NSError
                fatalError("Unresolved error \(nserror), \(nserror.userInfo)")
            }
        }
    }
```

Кроме того, для нас был создана модель, она хранится в файле **%ProjectName%.xcdatamodeld**. Мы создадим всего одну сущность – **Human**. Для этого нажмите кнопку Add Entity.



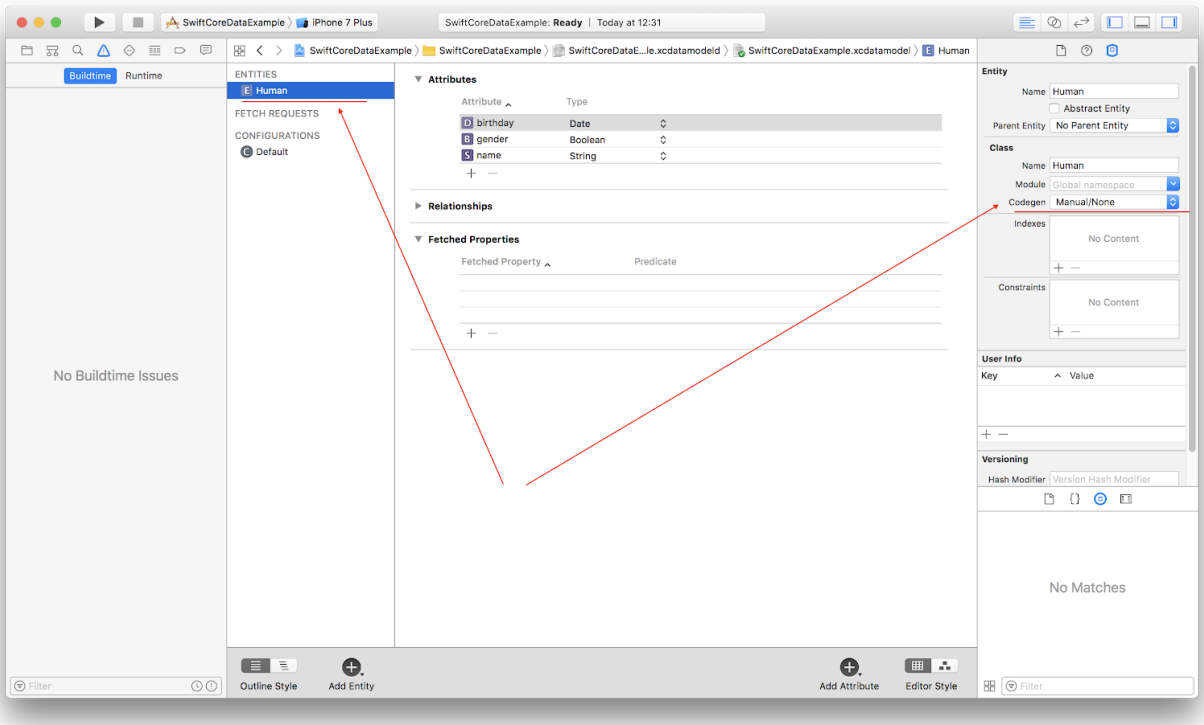
Переименуйте только что созданную сущность.



Теперь добавим несколько полей:

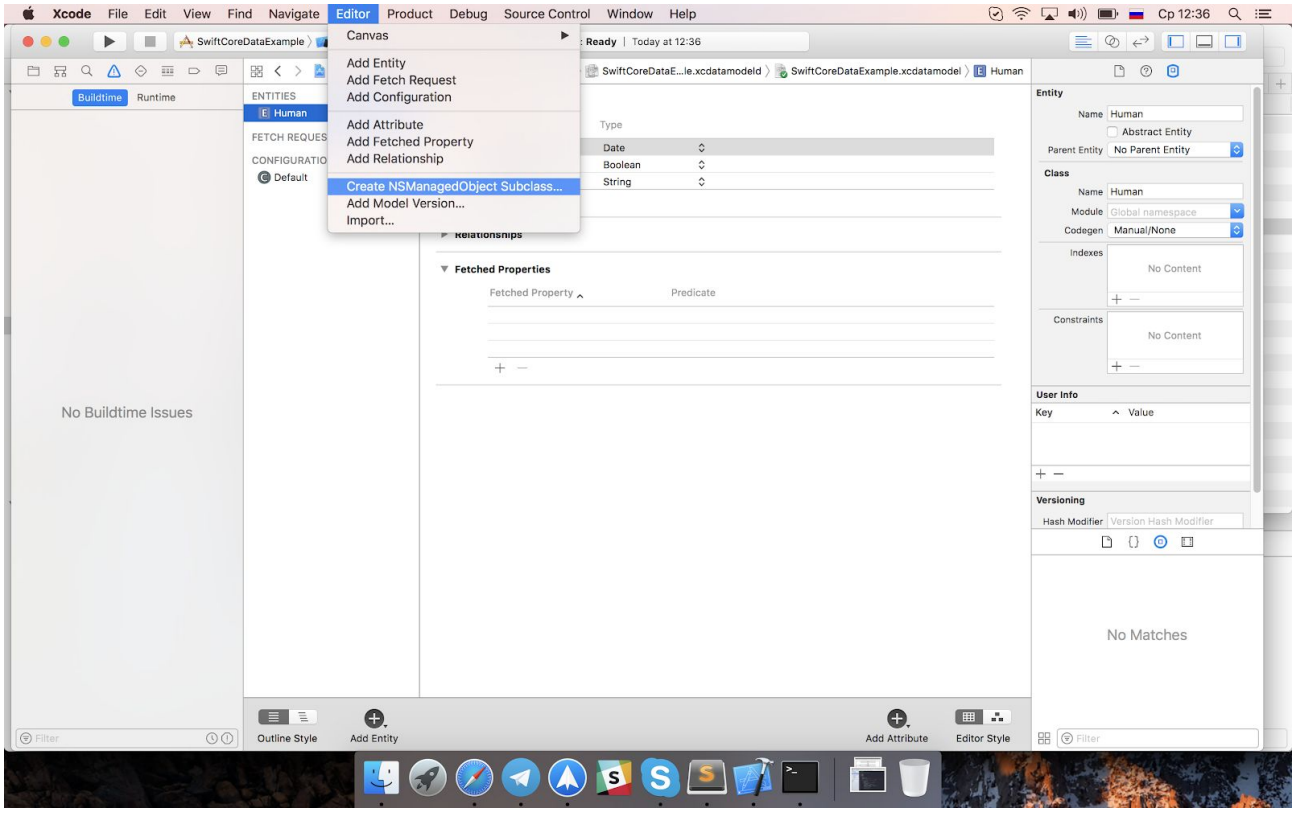
- name –string;
- gender – boolean;
- birthday – date.

Это будет простая сущность без изысков. Чтобы работать с ней, нам потребуется специальный класс, который надо сгенерировать. Прежде чем генерировать классы, надо настроить сущность. Установите ее атрибуту **Codegen** значение **Manual/None**.

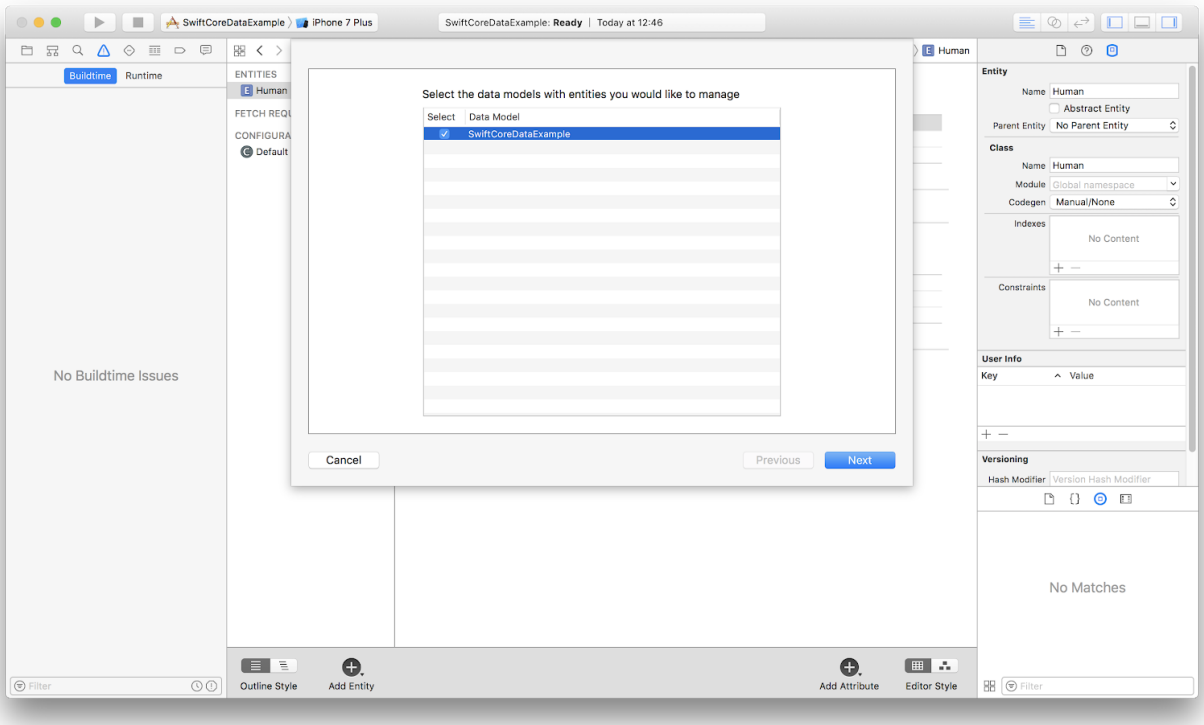


Это важный шаг, не забудьте его.

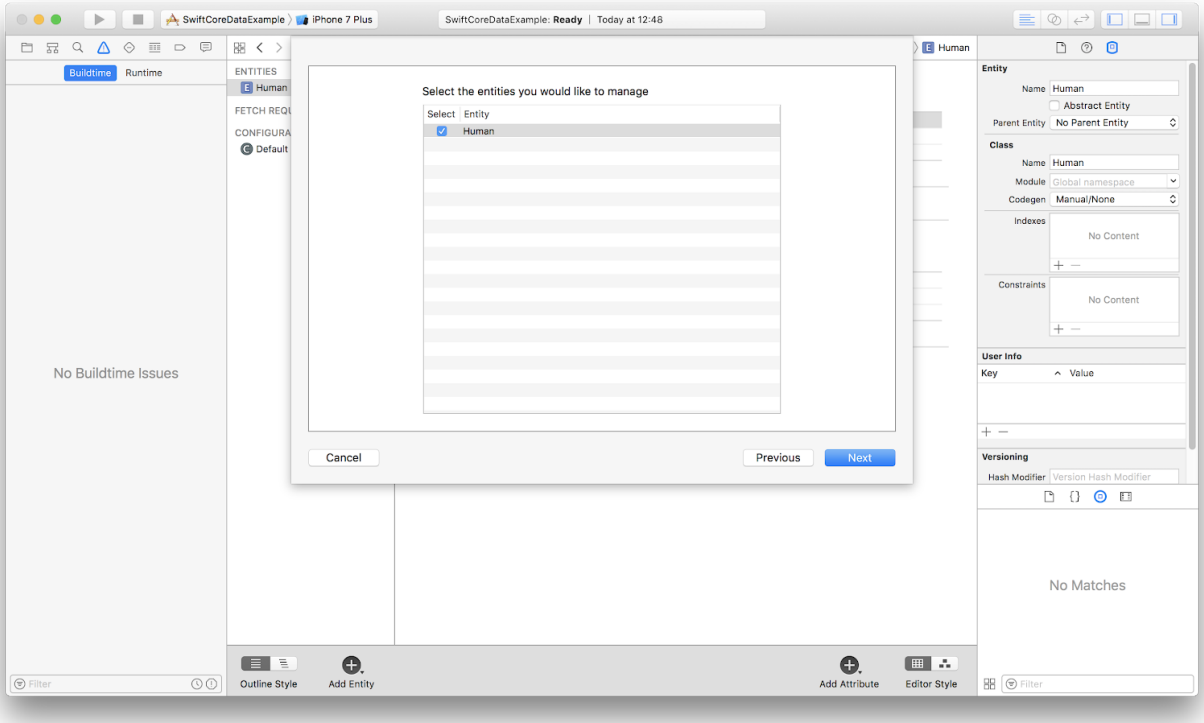
После этого выберите пункт меню **Editor -> Create NSManagedObject Subclass...**:



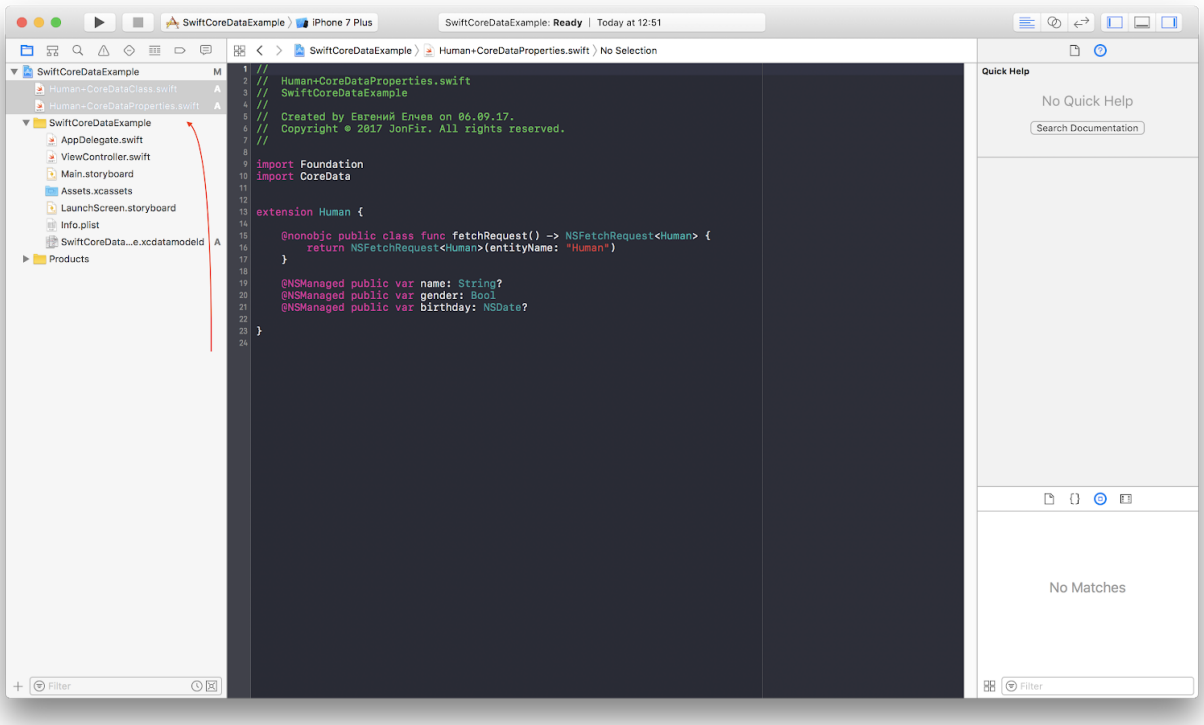
На следующем шаге выберите модели, для которых необходимо создать ManagedObject. Так как у нас одна модель, выберем ее.



И выберем сущности, сущность у нас тоже только одна.



В итоге будет сгенерировано два файла. Это объектное представление нашей сущности.



Теперь мы можем создать нового человека и сохранить его в CoreData. Для этого необходимо получить ссылку на координатор. Так как он объявлен в классе **AppDelegate**, получим ссылку на него. Имея ссылку на координатор, мы можем получить ссылку на контекст. Мы будем работать с **viewContext**, он создан для обращения к CoreData из главного потока. Теперь создадим новый экземпляр класса **Human**: используем конструктор, принимающий контекст, в котором должен быть создан объект. Объекты CoreData не просто инициализируются, одновременно они создаются в контексте и будут сохранены в хранилище. После этого мы можем изменить любые свойства объекта и вызвать метод **saveContext** у **AppDelegate**.

```
        let application =  UIApplication.shared.delegate as! AppDelegate

        let newHuman = Human(context: context)
        newHuman.name = "Jon"
        newHuman.gender = true
        newHuman.birthday = Date() as NSDate
        application.saveContext()
```

Чтобы извлечь данные из CoreData, нам потребуется ссылка на контекст. После мы используем метод **fetch** контекста и получим записи из базы данных. Нам потребуется привести типы:

```
func loadHumans() {
        let application =  UIApplication.shared.delegate as! AppDelegate
        let context = application.persistentContainer.viewContext
        let results = try! context.fetch(Human.fetchRequest()) as! [Human]
        let human = results.first!
    }
```

Конечно, это очень простые примеры. Но в целом этого хватит, чтобы сохранять данные.

Realm

Это последний инструмент, рассмотренный в нашем списке. Это не просто библиотека для работы с базой, а самостоятельная база данных. Она разработана специально для мобильных устройств. У нее много плюсов и новичкам она может показаться идеальным решением, но у нее также много минусов.

Realm не требует настройки, сущности – простые объекты, есть связи между объектами, реализованные как свойства, объекты просто сохранять, удалять, получать. У Realm очень быстрое извлечение данных. Из минусов – отсутствие каскадного удаления связанных объектов, запись только в один поток, а также быстро растущий объем файла на диске при большом количестве данных в хранилище.

Как правило, Realm – отличное решение для использования в простых приложениях с небольшим количеством данных без сложных каскадных операций.

Создание клиента для сервиса openweathermap.org

На этом уроке мы преобразуем класс для хранения погоды в Realm-класс.

Для начала добавим Realm в проект. Откроем Podfile и добавим в него строку **pod 'RealmSwift'**. Откроем терминал, перейдем папку проекта, выполним команду **pod update**. После чего откроем проект и класс Weather. Чтобы использовать Realm, его необходимо импортировать: **import RealmSwift**.

Класс Realm немногим отличается от обычного. Во-первых, он наследуется от класса **Object**. Во-вторых, обычные свойства должны быть помечены ключевым словом **@objc dynamic**. В-третьих, так как у родительского класса есть конструктор, который мы трогать не будем, надо отметить конструктор для создания из json данных как вспомогательный.

```
import Foundation
import RealmSwift

class Weather: Object, Decodable {
    @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

   <...>
}
```

Теперь объекты этого класса можно сохранять и получать из хранилища.

Ключевое слово **@objc dynamic** означает, что то, как будет будет работать переменная, определится во время выполнения программы. Если быть точным, при попытке прочитать значение этих свойств, Realm будет читать данные из хранилища.

Теперь перейдем к классу WeatherService. Добавим метод для сохранения погоды в базу.

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

// начинаем изменять хранилище
            realm.beginWrite()

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

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

Давайте разберем, что там происходит. У метода есть аргумент, массив объектов погоды, именно его мы будем сохранять в Realm.

Далее внутри метода уже знакомый вам блок **do { } catch { }**. Это необходимо потому, что метод получения экземпляра Realm и метод сохранения данных в хранилище могут сгенерировать исключение. Исключения будут созданы, если при получении доступа к хранилищу возникнет ошибка, например, структура хранилища не будет соответствовать объекту или он может не оказаться на устройстве. Как бы то ни было, ошибки могут возникнуть и необходимо быть к ним готовым.

Само взаимодействие с Realm достаточно простое. Мы получаем объект класса Realm для доступа к хранилищу. Имея эти объекты, мы начнем сеанс записи командой **realm.beginWrite()**, добавим все наши объекты в хранилище **realm.add(weathers)** и завершим сеанс записи командой **try realm.commitWrite()**.

Если попытаться изменить данные в хранилище без сеанса записи, мы увидим ошибку в консоли, а приложение упадет.

Нам осталось вызвать метод сохранения данных при получении их с сервера.

```
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

            self?.saveWeatherData(weathers)

            completion(weathers)
        }
```

На этом знакомство с Realm завершено, на следующем уроке мы изучим его более подробно.

Практическое задание

На основе ПЗ предыдущего урока:

1. Преобразовать ранее созданные объекты User, Photo, Group в объекты Realm.

Дополнительные материалы

1. CoreData
2. realm
3. База данных

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/CoreData/index.html