Разработка под iOS. Взлетаем

Часть 3

Работа с базой данных

Оглавление

Глава 2

1 Разработка под iOS. Взлетаем

3.1. NSCoding

Здравствуйте! Меня зовут Константин, и я занимаюсь разработкой мобильных приложений в компании Яндекс. На этой неделе мы рассмотрим некоторые из Persistance stores, которые есть в iOS.

Persistance store - это хранилище, которое позволяет сохранять данные между запусками приложения. В iOS SDK для этого существуют следующие компоненты:

- NSUserDefaults
- Keychain
- NSCoding
- Core Data

Давайте поговорим про NSCoding.

NSCoding - это механизм, обеспечивающий легкое кодирование и раскодирование объекта с помощью одноименного протокола. Закодированные объекты, например, могут быть сохранены на диск. Многие системные классы уже реализуют данный протокол.

Для соответствия протоколу класс должен реализовать 2 метода:

```
func encode(with aCoder: NSCoder)
init?(coder aDecoder: NSCoder)
```

Метод encode вызывается классом, который кодирует объект. Туда он передает сам себя. Метод может вызываться неоднократно для одноименного объекта.

init вызывается в момент декодинга объекта. Например, реализация может выглядеть вот так:

```
class Person: NSCoding {

    var name: String
    var lastName: String

    init(name: String, lastName: String) {
        self.name = name
        self.lastName = lastName
    }

    func encode(with aCoder: NSCoder) {
        aCoder.encode(name, forKey: "name")
        aCoder.encode(lastName, forKey: "lastName")
    }
```

```
    required convenience init?(coder aDecoder: NSCoder) {
        guard let name = aDecoder.decodeObject(forKey: "name") as? String,
        let lastName = aDecoder.decodeObject(forKey: "lastName") as? String
        else {
            return nil
        }
        self.init(name: name, lastName: lastName)
    }
}
```

Следуя принципам ООП, объект, который кодируется или декодируется, ответственен за то, чтобы закодировать или декодировать те данные, которые необходимы ему для полного восстановления своего состояния. Соответственно, если класс содержит ссылку на объект другого класса, то весь граф объектов должен поддержать протокол NSCoding. Например, это может выглядеть вот так:

```
class Person: NSObject, NSCoding {

    var friends: Array<Person>

    func encode(with aCoder: NSCoder) {
        aCoder.encode(friends, forKey: "friends")
    }

    required convenience init?(coder aDecoder: NSCoder) {
        let friends = aDecoder.decodeObject(forKey: "friends ") as? Array<Person>
    }

}
```

Теперь рассмотрим NSCoder. NSCoder - это абстрактный класс. Он предоставляет интерфейс для перевода объектов между памятью и другим форматом.

Самые распространенные реализации данного класса - это NSKeyedArchiver и NSKeyedUnarchiver. С их помощью можно переводить объект в байтовое представление и обратно. Это позволит потом сохранить объект в файл или передать его по сети.

Кодирование может выглядеть вот так:

```
let person: Person = ...

do {

    let data = try NSKeyedArchiver.archivedData(withRootObject: person,
requiringSecureCoding: false)

    let path = ...
    FileManager.default.createFile(atPath: path, contents: data, attributes: nil)

} catch {

    print(error)

}
```

Протокол NSSecureCoding отличается от NSCoding тем, что он более устойчив к подмене объектов.

```
static var supportsSecureCoding: Bool
```

Использование NSCoding небезопасно, потому что для проверки типа декодируемого объекта его надо создать и, если это часть коллекции, то добавить в граф объектов. NSSecureCoding проверяет класс объекта до его создания.

Теперь поговорим о миграции. Преимущество миграции объектов, закодированных с помощью NSCoding, состоит в их обратной совместимости. Вы можете читать данные в любой последовательности или игнорировать ненужные значения. Нам необходимо описывать всю миграцию самим, если она потребуется.

И в заключение. Реализация протокола содержит много booler played кода, но это же и дает прозрачность того, как работает данный механизм. Также поддержка протокола всем графом объектов может быть довольно сложной. В следующем разделе мы поговорим о Core Data.

3.2. NSManagedObjectModel

Теперь мы рассмотрим фреймворк Core Data. Разберем, для чего он нужен, из чего состоит и как с ним работать.

Core Data - это фреймворк, который позволяет управлять графом объектов в приложении. Он предоставляет решения общих задач для работы с объектами, связанных с жизненным циклом, управлением зависимостями и хранением.

Давайте начнем с модели данных, или, если вам близка терминология баз данных, со схемы данных. Модель представлена классом NSManagedObjectModel. Она описывает сущности и отношения между ними. Чем общирнее модель, тем больше Core Data подойдет для вашего приложения.

Модель хранится в *.xcdatamodel и при сборке проекта компилируется в файл с расширением .momd. Модель позволяет описывать абстрактные сущности. Это такие сущности, объекты которых никогда не будут созданы, но могут быть унаследованы другими сущностями.

Сущности могут наследоваться друг от друга. Наследование работает так же, как и для обычных классов. Далее мы создадим свою модель данных.

Сейчас мы создадим свою модель данных, познакомимся с инструменттом для создания моделей и более подробно разберем, из чего состоит модель данных и что мы моем в ней задать.

Чтобы было более понятно, что происходит, начнем с пустого проекта. Для начала необходимо создать саму модель данных.

Откроем меню "Файл" , New file и выберем Data Model.

Давайте создадим нашу первую сущность. Для этого нажмем на кнопку Add Entity и назовем ее Person. Имя сущности в модели и имя класса, который будет ее представлять в коде, могут быть разными. По умолчанию они совпадают, но их можно поменять тут же, в инспекторе объектов.

Обычно, если вы работаете с Objective-C, то стоит использовать префикс MO для названий ваших классов. Здесь же есть возможность указать, что данная сущность будет абстрактной.

Обратим внимание на поле CodeGen. Это свойство описывает то, надо ли для нас сгенерировать класс данной сущности и если надо, то как. Manual означает, что мы сами должны сгенерировать класс; Class Definition означает, что класс будет сгенерирован автоматически; Category также означает, что класс будет сгенерирован автоматически и все свойства будут добавлены в него через Extension. Пока оставим значение по умолчанию - Class Definition и вернемся к этому немного позже.

Создадим атрибуты для нашей сущности. Для этого нажмем на кнопку Add Attribute и зададим название - firstName. Название атрибута не может быть таким же, как имена методов или свойств у классов NSObject и NSManagedObject. Например, нельзя задать атрибут с названием Description.

Теперь зададим тип атрибута: String. Core Data поддерживает множество нативных типов, например String, Int, Date и так далее. Также есть тип BinaryData, в котором можно хранить картинки, видео или другие ресурсы. Но лучше так не делать и хранить ресурсы в отдельных файлах на диске. Это поможет сэкономить память в базе данных и ускорить ее работу.

Для некоторых типов данных есть возможность задать дополнительные параметры. Например, для типа String это минимальный и максимальный размер строки, а также значение по умолчанию. Набор дополнительных полей зависит от тип данных.

Давайте зададим для атрибута firstName ограничение, что он не может быть меньше, чем 1 символ.

В Core Data есть поддержка Optional, но их лучше избегать, особенно при работе с числами. Использование Optional приводит к усложнению запросов и индексов. Пустое значение Optional представлено в SQL в виде NULL. NULL в SQL обозначает, что значение не задано. Оно не равно нулю или пустой строке. Также NULL не равен самому себе.

По умолчанию все атрибуты Optional. Уберем этот признак для нашего атрибута.

Еще один пункт, на который стоит обратить внимание - это Transient. Транзиентные атрибуты - это такие атрибуты, значения которых не сохраняются в хранилище, но Core Data продолжает отслеживать изменения в таких атрибутах. В основном они используются для вычисляемых или производных значений.

Создадим еще один атрибут и назовем его lastName. Зададим ему тип String и сделаем его обязательным - не Optional.

Создадим атрибут и назовем его birthDate. Зададим ему тип Date. Обратим внимание, что свойства у атрибутов в инспекторе объектов поменялись, т.к. они зависят от типа атрибута. Выберем, что дата рождения не может быть меньше, чем 1 января 1900 года.

Создадим еще одну сущность и назовем ее Employee. Перейдем в инспектор объектов и укажем, что она будет наследоваться от Person.

Создадим атрибут для Employee и назовем его position. Зададим ему тип String и сделаем его обязательным.

Создадим еще одну сущность и назовем ее Organization.

Создадим атрибут для организации и назовем его name. Зададим ему тип String и сделаем его обязательным.

Сущносити могут быть связанными друг с другом. Создадим отношения между ними. В графе Relatoinships добавим поле employees. Затем зададим, с какой сущностью будет данное отношение, выбрав в Destination Employee.

Отношения бывают один к одному, один ко многим или многие ко многим. Укажем для данного отношения "один ко многим".

Также можно настроить правила, которые будут действовать для отношений в случае удаления объекта данной сущности. Возможны следующие варианты поведения:

- Nullify. В данном случае при удалении объекта Department все объекты, которые на него ссылались, обнулят свои ссылки, т.е. их значение будет nil.
- Cascade. При удалении объекта Department удаляются все сотрудники, которые на него ссылались.
- Deny. Объект можно будет удалить тогда, когда на него никто не будет ссылаться.
- No action. Если мы удалим объект Department, то все объекты Employee, которые на него ссылались, сохранят свои ссылки на несуществующий объект.

Теперь вернемся к Employee и добавим отношение department, выбрав в destination Organization.

Обратим внимание на inverse. Inverse - это двустороннее отношение, которое имеет обратную связь. Например, если мы добавим сотрудника в отдел, то он автоматически будет добавлен в список сотрудников отдела. Рекомендовано всегда создавать двустороннюю связь.

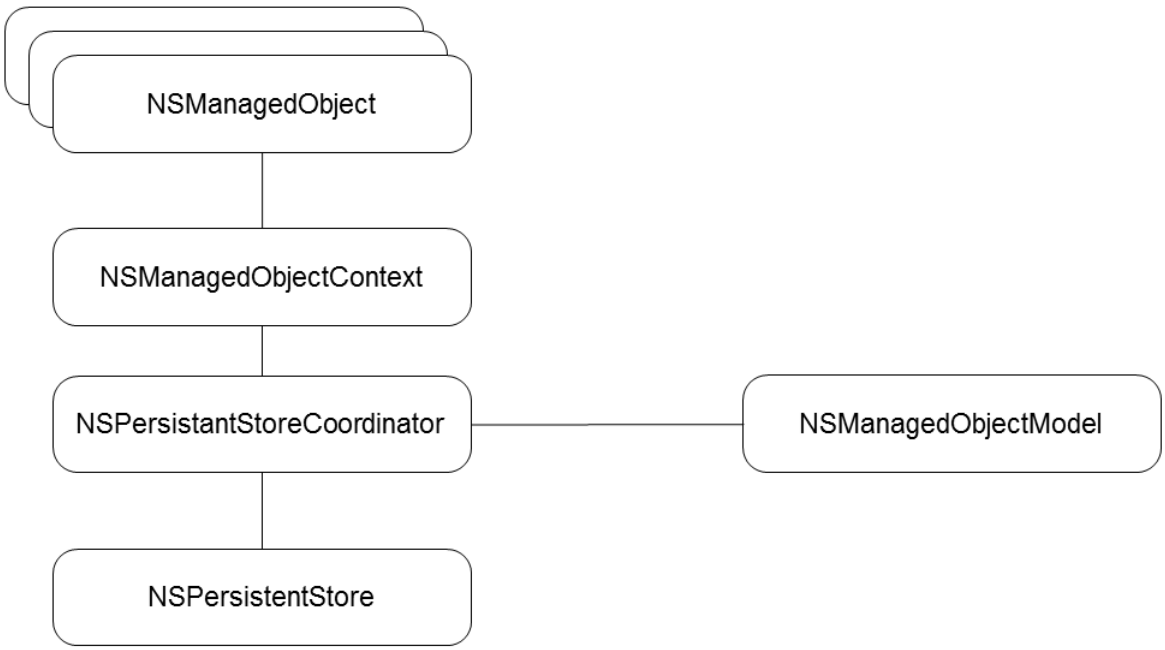
Fetched property - это такие свойства, которые представляют собой слабую связь один ко многим, являются своего рода шаблонами для получения данных. Например, можем создать у Department свойство, которое будет возвращать недавно нанятых сотрудников. Fetched requests отличаются от отношений тем, что они могут получать не все объекты заданного типа, а дополнительно иметь определенные фильтры.

Отлично! Мы с вами немного познакомились с моделью данных и создали свою собственную. В следующем разделе вы узнаете, из чего состоит Core Data.

3.3. Стэк Core Data

В этом разделе мы узнаем, из чего состоит Core Data и как ее компоненты взаимодействуют между собой.

Стек Core Data состоит из NSManagedObjectModel, NSManagedObjectContext, NSManagedObject, NSPersistantStoreCoordinator и NSPersistentStore.



Рассмотрим каждый компонент более подробно.

NSManagedObjectModel - класс, описывает модель данных, ту, что мы создавали с вами ранее. Она первой загружается при инициализации всего стека. На основании модели данных происходит маппинг объектов в хранилище данных. После того, как модель инициализирована, создается NSPersistant StoreCoordinator.

NSPersistantStoreCoordinator - это класс, который загружает и создает сущности, описанные в модели данных из хранилища. Координатор находится в середине стека Core Data. Если NSManagedObjectModel

описывает структуру данных, то координатор достает данные из хранилища и передает их в нужный NSManagedObjectContext. Он может работать с несколькими хранилищами.

NSPersistentStore - это абстрактный класс, который обеспечивает хранение данных. Core Data поддерживает 4 вида хранения данных:

- SQLite;
- Binary;
- In-memory;
- XML (не поддерживается на iOS)

Все хранилища делятся на атомарные (представлены классом NSAtomicStore), которые стоит использовать, если вы хотите интегрировать свой формат файла для работы с Core Data. Данный тип направлен на то, чтобы хранить данные, которые могут быть выражены в памяти. Атомарные хранилища предпочитают простоту вместо производительности. Пример реализации такого хранилища - Binary и XML. И инкрементальные (представлены классом NSIncrementalStore), которые стоит использовать, чтобы загружать и сохранять данные инкрементально, что позволит управлять большими объемами данных. Пример реализации такого вида хранилищ - SQLite.

NSManagedObjectContext - это класс, с которым вы будете взаимодействовать больше всего. Он служит для получения, сохранения и изменения объектов из хранилища. Когда вы получаете объекты из координатора, контекст сохраняет их копии, и они формируют граф на основе отношений. Можно менять объекты внутри контекста, создавать и удалять. Данные в хранилище не поменяются, пока вы не сохраните изменения в контексте, которые потом попадут в координатор. Когда происходит сохранение, контекст проверяет, что все объекты в валидном состоянии, и записывает их. Все объекты должны быть зарегистрированы в контексте. Контекст отслеживает изменения всех зарегистрированных в нем объектов и их связей. Важно отметить, что контекст является потокозависимым, и нельзя создавать контекст в одном потоке и передавать в другой.

Контекст также может обладать операциями redo и undo. Возможность данных операций используется за счет undoManager'a. Если вам не нужны эти операции, то следует убрать undoManager, т.к. его наличие - это потенциальный performance hit.

NSManagedObject - это базовый класс для реализации сущностей в Core Data. Объекты именно этого класса возвращает Core Data по вашему запросу. Полезно сделать subclass для каждой из ваших сущностей, описанных в модели. Мы уже видели, что существует возможность использовать автогенерацию данных классов на основе модели данных.

При создании subclass'a нужно учитывать следующее:

- нельзя переопределять эти методы:

```
primitiveValueForKey:
setPrimitiveValue:forKey:
isEqual:
hash
superclass
class
self
zone
func isProxy() -> Bool
func isKind(of aClass: AnyClass) -> Bool
func isMember(of aClass: AnyClass) -> Bool
func conforms(to aProtocol: Protocol) -> Bool
```

```
func responds(to aSelector: Selector!) -> Bool
managedObjectContext
entity
objectID
isInserted
isUpdated
isDeleted
isFault
description;
```

- нельзя использовать KVC, т.к. Core Data использует их сама;
- нельзя вызывать dealloc/deinit; вместо этого следует использовать метод

```
func didTurnIntoFault()
```

- нельзя использовать init, вместо этого следует использовать

```
func awakeFromInsert()
func awakeFromFetch()
func validateForUpdate() throws
```

Но обязательно следует вызывать super. Это обусловлено тем, что объект не всегда создается со всеми значениями, а может быть пустышкой - fault. Также может не удаляться сразу, что обусловлено механизмом Faulting, о котором мы поговорим позднее.

Каждый NSManagedObject состоит из 2 базовых элементов:

- NSEntityDescription - содержит название, атрибуты и связи сущности;
- NSManagedObjectContext - контекст, которому принадлежит сущность.

Создание объекта NSManagedObject не гарантирует его появления в хранилище: нужно явно вызвать сохранение для контекста, к которому объект относится.

Мы с вами узнали, из чего состоит Core Data; теперь давайте попробуем создать его.

Сейчас мы научимся создавать стек Core Data. Сначала давайте немного приведем в порядок наш проект.

Для начала удалим наш стандартный ViewController.

Создадим новый ViewController и назовем его OrganizationsViewController. Напишем в нем следующий код:

```
import UIKit

class OrganizationsViewController: UIViewController {

}
```

Отлично! Теперь мы можем начать подготавливать наш UI. Но для начала соберем проект. Для этого откроем storyboard и найдем там ViewController.

Сменим класс на OrganizationsViewController. Добавим NavigationController. Для этого выберем в меню Editor -> Embedded in -> NavigationController. Теперь добавим Outlet и назовем его tableView.

Вернемся к коду нашего ViewController'а и напишем в нем следующее:

```
UIViewController {

    @IBOutlet weak var tableView: UITableView!

    override func viewDidLoad {
        super.viewDidLoad()
        tableView.refreshControl = UIRefreshControl()

        tableView.refreshControl?.beginRefreshing
    }
}
```

Пройдем шаг за шагом по коду. Мы добавили в таблицу UIRefreshControl, который отрисовывает спиннер, и запустили его. Давайте запустим наш проект.

Мы увидим, что на экране у нас есть таблица, и в ней - вращающийся спиннер. Отлично! Наш UI готов.

Теперь перейдем к созданию стека Core Data. Откроем файл AppDelegate и внутри одноименного класса напишем следующий код:

```
func createContainer(completion: @escaping
(NSPersistentContainer) -> ()) { let container = NSPersistentContainer(name:
"Model")
    container.loadPersistentStores(completionHandler: { _,
    error in
        guard error == nil else {
            fatalError("Failed to load store")
        }
        DispatchQueue.main.async { completion(container) }
    })
}
```

Теперь давайте пройдем шаг за шагом по данному коду. Первым шагом мы создаем контейнер с типом NSPersistentContainer и передаем ему название нашей модели - "Model": NSPersistentContainer - это такой helper class из фреймворка Core Data, который появился в iOS 10 и позволяет быстро создавать весь стек.

Следующим шагом мы вызовем метод loadPersistentStores. Данный метод попробует открыть файл базы данных с таким же именем, как и имя модели, и, если файл не найден, то сгенерирует его для нас.

Теперь добавим импорт фреймворка Core Data:

```
import Core Data
```

и соберем проект.

Отлично! Теперь наш стек создан. Давайте взглянем поближе на то, что есть в классе NSPersistent Container.

```
open class NSPersistentContainer : NSObject {

    open class func DefaultDirectoryURL() -> URL

    open var name: String { get }
    open var viewContext: NSManagedObjectContext { get }
    open var managedObjectModel: NSManagedObjectModel { get }
    open var persistentStoreCoordinator: NSPersistentStoreCoordinator { get }
    open var persistentStoreDescriptions: [NSPersistentStoreDescription]
    ...
}
```

И мы увидим, что здесь уже есть все, что нам нужно: весь проинициализированный стек. viewContext, созданный для нас NSPersistentContainer и привязанный к главному потоку приложения. Тут же лежат NSManagedObjectModel и NSPersistentStoreCoordinator.

Для загрузки и сохранения данных нашему viewController потребуется контекст. Давайте добавим ему его. Откроем класс OrganizationsViewController и напишем. Для начала добавим импорт фреймворка Core Data:

```
import Core Data
```

Теперь напишем:

```
var context: NSManagedObjectContext! {
    didSet {
        tableView.refreshControl?.endRefreshing()
    }
}
```

Мы определили propertly context и добавили остановку спиннера, как только контекст будет задан. Теперь нужно передать нашему контроллеру сам контекст. Вернемся в AppDelegate и добавим в свойства контейнер:

```
var container: NSPersistentContainer!
```

А в методе didFinishLaunchingWithOptions напишем следующее:

```
createContainer { container in
    self.container = container

    if let nc = self.window?.rootViewController as?
    UINavigationController,
    let vc = nc.topViewController as?
    OrganizationsViewController
```

```
    {
        vc.context = container.viewContext
    }
}
```

Для начала мы вызовем helper-функцию createContainer, которая создаст весь стек Core Data и вернет нам сам контейнер. Следующим шагом мы сохраним к себе контейнер, далее получим наш OrganizationsViewController, который находится внутри UINavigationController'а, и передалим в него viewContext из контейнера.

Стоит отметить, что не стоит использовать такое получение контроллера в реальном проекте. Давайте запустим наш проект и увидим, что спиннер исчез.

Поздравляю! Мы успешно проинициализировали стек Core Data и передали контекст нашему viewController'у. Далее мы поговорим про класс NSObjects и про его кодогенерацию.

В случае включения автогенерации классов они будут сгенерированы для нас при сборке проекта и лежать в Derived Data. Также есть возможность сгенерировать классы руками. Давайте сделаем это.

Откроем нашу модель данных, перейдем в меню Editor, create NSManagedObject subclass. Выберем нашу модель и нажмем Next. Выберем, например, класс Organization и нажмем Next и Create. Как мы видим, для нас создался класс Organization.

Давайте посмотрим на него повнимательнее. Обратим внимание на то, что все свойства и методы помечены ключевым словом NSManaged. Это значит, что Core Data обеспечивает хранение и реализацию данных свойств и методов в субклассах NSManagedObject. В случае, если вы пишете субкласс своими руками, вам следует добавить NSManaged к каждому свойству, которое совпадает с атрибутом в модели данных. NSManaged - это такой же признак, как https://staff.yandex-team.ru/dynamic в Objective-C. Однако он доступен только при поддержке Core Data.

Теперь обратим внимание на методы. Как мы видим, Core Data предоставляет нам методы для работы с коллекцией отношений, в нашем случае - сотрудников. Мы ими воспользуемся в последующих уроках.

В следующем разделе мы научимся сохранять данные в Core Data.

3.4. Context. Сохранение данных

В этом разделе мы научимся сохранять данные в Core Data. Для начала давайте откроем storyboard и добавим кнопку Bar Button Item.

Зададим ей system item Add и сделаетм Action в OrganizatonsViewController, назвав метод addOrganization. С подготовкой UI мы закончили. Теперь перейдем непосредственно к его созданию.

Как мы уже говорили ранее, для создания объекта модели нам потребуются 2 вещи: описание класса из нашей модели и NSManagedObjectContext. Описание можно получить из класса NSEntityDescription, а контекст будет отслеживать изменения в нашем объекте, в том числе его создание. Давайте в методе AddOrganization напишем следующий код:

```
guard let context = context else { return }

guard let organization =
NSEntityDescription.insertNewObject(forEntityName: "Organization", into: context)
as? Organization else
{
    return
}
```

Теперь разберем его. Мы добавили новый объект класса Organization в NSManagedObjectContext, чтобы он мог отслеживать в нем изменения. После создания объект имеет пустые поля и является некоей болванкой. Если сразу попытаться сохранить его, то произойдет ошибка, т.к. объект не пройдет валидацию. Помните, мы делали name обязательным полем? Поэтому давайте зададим name.

```
organization.name = "Yandex"
```

Теперь наша организация готова к сохранению. Сохранение всех изменений, произошедших в контексте, происходит с помощью метода save:

```
context.save()
```

Поскольку данный метод помечен как throws, добавим do catch:

```
do {
    try context.save()
} catch {
    print(error)
}
```

Поскольку большинство вызовов Core Data помечено throws, то нам везде необходимо производить проверки и корректно реагировать на ошибки. Для того, чтобы не растягивать наш урок, я буду писать try!, чего в настоящем проекте делать не следует.

Теперь, если запустить приложение и нажать на +, то будет создана организация и сохранена в базу данных. Далее мы поговорим о NSManagedObjectContext, о том, что происходит при изменении, сохранении данных, а также немного коснемся многопоточности.

Как мы уже говорили ранее, каждый контекст отслеживает все изменения в данных, которые к нему относятся. Изменения внутри контекста случаются, когда мы добавляем, удаляем или изменяем объекты.

Различают 2 вида изменений.

Первый - между вызовами

```
func save() throws
```

Для того, чтобы сохранить данные контекста в хранилище, Core Data должна знать, какие объекты добавлялись, удалялись или обновлялись. Core Data также отслеживает изменения в атрибутах. Все это помогает ей понимать, какие данные надо сохранить, а также решать конфликты, которые возникают при сохранении.

При сохранении данных контекст отправляет уведомление

```
NSManagerObjectContextDidSave
```

Оно содержит информацию обо всех сохраненных изменениях, сделанных с момента последнего вызова save.

Второй вариант - между вызовами

```
func processPendingChanges()
```

Метод добавляет изменения в граф объектов. При добавлении изменения контекст отправляет уведомление

```
NSManagedObjectContextObjectsDidChange
```

Оно содержит информацию обо всех изменениях, совершенных с момента вызова processPendingChanges. Обычно нам не надо вызывать данный метод самостоятельно: Core Data вызывает его для нас.

Оба эти уведомления обычно широко используются для синхронизации данных между разными контекстами, а также NSFetchedResultController, который используется для отслеживания изменений. С FetchedResultController мы познакомимся чуть позднее.

Также для отслеживания изменений любой NSManagedObject имеет свой уникальный идентификатор, который представлен классом NSManagedObjectID. Идентификатор есть только у тех объектов, которые отслеживаются контекстом. Идентификатор уникален между контекстами. Для того, чтобы получить объект по его идентификатору, у NSManagerObjectContext существуют следующие методы:

```
func registeredObject(for objectID: NSManagedObjectID) -> NSManagedObject?

func object(with objectID: NSManagedObjectID) -> NSManagedObject

func existingObject(with objectID: NSManagedObjectID) throws -> NSManagedObject
```

- ```
  registeredObject(for objectID:)
  ```

  Данный метод ищет объект только внутри контекста и не делает обращения к хранилищу, за счет чего он достаточно быстрый. Метод вернет nil, если объект не найден.

- ```
  object(with objectID:)
  ```

  Очень быстрый метод. Не делает вообще никаких проверок; если объекта с таким ID внутри контекста нет, то создает пустой объект.

- ```
  existingObject(with objectID:)
  ```
  ```

Делает поиск внутри контекста, и, если не находит объекта в нем, то делает FetchRequest в базу данных. В случае, если объект не найден, произойдет ошибка.

Теперь давайте немного обсудим многопоточность. Как мы уже говорили ранее, контекст всегда потокозависим. Когда контекст создается, есть возможность указать, какому потоку он будет принадлежать:

```
init(concurrencyType ct: NSManagedObjectContextConcurrencyType)
```

Есть 2 очереди, с которыми можно создать контекст:

- privateQueueConcurrencyType - контекст создает свой поток и выполняется в нем
- mainQueueConcurrencyType - все свои действия контекст будет выполнять в главном потоке приложения.

Для выполнения операций, связанных с многопоточностью, у контекста есть 2 метода:

```
func perform(_ block: @escaping () -> Void)
func performAndWait(_ block: () -> Void)
```

Эти методы вызывают переданный им блок на том потоке, который привязан к контексту. Стоит отметить, что если вы используете контекст на главном потоке, то методы можно вызывать явно, без использования perform и performAndWait.

Объекты класса NSManagedObject не предназначены для передачи между потоками. В случае, если вам необходимо передать NSManagedObject из одного потока в другой, вы должны передать между потоками NSManagedObjectID. В случае, если у вас несколько контекстов, то при их синхронизации могут возникнуть конфликты. Но об этом мы поговорим позже.

В этом разделе я постарался немного рассказать о том, какие методы используются для отслеживания изменений, и о работе с потоками. Обе эти темы довольно обширны, и если вам интересно узнать больше, то стоит обратиться к документации. В следующем разделе мы познакомимся с механизмами загрузки данных из Core Data.

3.5. NSFetchedResultController

Теперь, когда мы научились сохранять данные, настало время научиться получать их обратно. В этом разделе мы поговорим про NSFetchedRequest, а также немного обсудим его достоинства и недостатки. Для начала нам с вами необходимо проделать немного подготовительной работы.

Откроем storyboard и выберем UITableView. Укажем ей в качестве Data Source и Delegate наш View controller. Теперь в инспекторе объектов в поле Prototype Cells изменим значение на 1, тем самым добавив один шаблон ячейки в таблицу. Выберем ячейку и выставим в меню Style - Subtitle. Обязательно зададим CellIdentifier - OrganizationCellIdentifier.

Теперь откроем OrganizationsViewController и добавим константу:

```
fileprivate let CellIdentifier = "OrganizationCellIdentifier"
```

А также создадим список организаций. Пока он будет пустым:

```
fileprivate var organizations = [Organization]()
```

Теперь необходимо реализовать протокол UITableViewDataSource. Для этого создадим extension и добавим следующий код:

```
extension OrganizationsViewController: UITableViewDataSource {

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

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
-> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier,
for: IndexPath)
        return cell
    }
}
```

Используем наш массив организаций, чтобы вернуть количество ячеек:

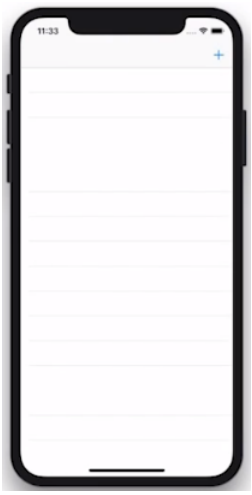
```
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int
{
    return organizations.count
}
```

Добавим название организации и количество сотрудников в нее для отображения в таблице, написав следующий код:

```
let organization = organizations[indexPath.row]

cell.textLabel?.text = organization.name
cell.detailTextLabel.text = "Employees \(organization.employees?.count ?? 0)"
```

Запустим наш проект и убедимся, что мы все сделали правильно:



Мы по-прежнему видим пустой список. Теперь, когда подготовительная часть готова, можно переходить к загрузке данных. Для получения данных используется клас NSFetchedRequest, который обладает обширным набором параметров, о которых я расскажу чуть позже. Сейчас давайте создадим функцию и перенесем туда endRefreshing():

```
func fetchData() {
    tableView.refreshControl?.endRefreshing()
}
```

Добавим вызов новой функции в didSet контекста:

```
fetchData()
```

Теперь внутри функции напишем следующий код:

```
let request = NSFetchRequest <Organization>(entityName: "Organization")
```

При создании FetchRequest необходимо указать тип сущности, которую мы будем запрашивать. Сделать это можно при помощи соответствующего initializer'a - entityName.

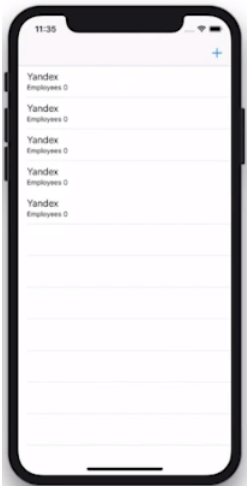
Простой запрос создан, осталось его выполнить. Это можно сделать с помощью метода Fetch у контекста. Присвоим результат вызова нашей переменной organizations:

```
organizations = try! context.fetch(request)
```

Осталось только обновить данные в таблице, и все будет готово:

```
tableView.reloadData()
```

Запустим приложение и увидим наши организации:



Пока у нас нет обновления списка при добавлении новой организации, поэтому добавленная организация появится только после перезапуска приложения.

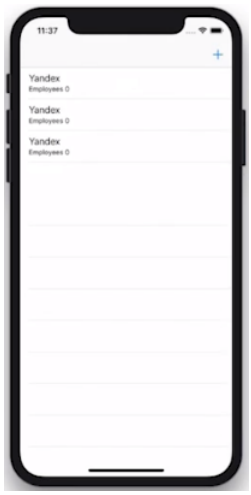
Теперь рассмотрим внимательно, что же происходит "под капотом" нашего вызова fetch и какие недостатки несет именно такое обращение к Core Data. Каждый раз, когда вы выполняете fetchRequest, Core Data проходит по всему своему стеку, чтобы получить данные, вплоть до файловой системы. fetchRequest делает полный цикл через контекст, persistentStoreCoordinator, persistentStore, SQLite и назад в обратном порядке. fetchRequest - очень мощный механизм запроса данных, он подразумевает

большой объем работы. Выполнение fetchRequest - очень дорогая операция. Позже мы рассмотрим и другие способы получения данных, а пока стоит использовать fetchRequest с осторожностью.

Теперь вернемся назад, к нашему примеру, и посмотрим, какими еще свойствами обладает NSFetchRequest. Например, можно ограничить количество возвращаемых значений, задав fetchLimit. Стоит отметить, что в случае большого количества данных всегда рекомендуется использовать fetchLimit. Зададим fetchLimit равным 3:

```
request.fetchLimit = 3
```

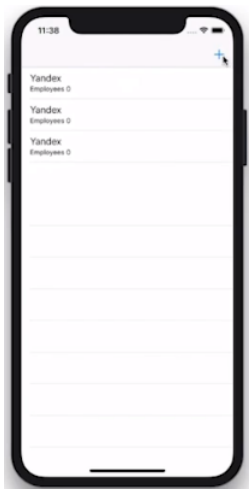
Запустим прект и увидим, что теперь мы получили всего 3 организации:



Также есть свойство fetchOffset, которое позволит задавать смещение выдачи относительно начала. Для проверки следующего свойства давайте поменяем имя у создаваемой организации, например, Яндекс.Такси:

```
organization.name = "Yandex Taxi"
```

Запустим и нажмем "добавить":



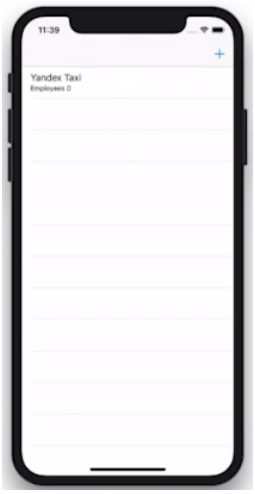
Теперь вернемся в код и создадим NSPredicate:

```
let predicate = NSPredicate(format: "name like '*Taxi'")
```

Добавим его в fetchRequest:

```
request.predicate = predicate
```

Запустив приложение, можно увидеть, что в списке организаций будут только организации, заканчивающиеся на Taxi:



NSPredicate обладают огромным набором возможностей и своим собственным синтаксисом, разбор которого выходит за рамки нашего урока. Также можно сгруппировать несколько NSPredicate в один. Еще одно интересное свойство NSFetchRequest, которое стоит отметить - это sortDescriptors. В нем можно указать список объектов класса NSSortDescriptor для указания сортировки данных. Создадим sortDescriptor и в качестве имени укажем name нашей организации:

```
let sortDescriptor = NSSortDescriptor(key: "name", ascending: true)
```

Параметр ascending позволяет задавать сортировку по возрастанию. Теперь можем добавить сортировку в request:

```
request.sortDescriptors = [ sortDescriptor ]
```

Стоит заметить, что значения predicate и sortDescriptors будут обработаны уже в SQLite, преобразовавшись в соответствующий запрос.

Вот мы и познакомились с одним из мощнейших механизмов для получения данных. Далее мы рассмотрим более гибкий механизм - NSFetchedResultsController, для чего он нужен и как его использовать.

NSFetchedResultsController очень удобно использовать с UITableView и UICollectionView благодаря его методам delegate, которые с легкостью позволяют отслеживать добавление, удаление и изменение данных. Также главным отличием от обычного NSFetchedRequest является то, что NSFetchedResultsController уведомляет нас об изменениях, произошедших в контексте. Достигается это с помощью того, что контроллер отслеживает изменения в переданном ему контексте, слушая натификации, о которых мы говорили ранее.

Также NSFetchedResultsController может дополнительно кэшировать данные.

Далее мы улучшим наш контроллер, добавив в него NSFetchedResultsController, что позволит нам отслеживать добавление новых элементов.

Теперь пришло время заменить вызов context.fetch() на работу с NSFetchedResultsController'ом. Для начала давайте создадим property:

```
private var fetchedResultsController: NSFetchedResultsController<Organization>?
```

Давайте создадим метод, который будет создавать и конфигурировать наш resultsController. Создадим NSFetchedRequest вместе с sortDescriptor, как мы это делали ранее. Пришло время создать сам resultsController. Для этого напишем следующее:

```
func setupFetchedResultsController(for context: NSManagedObjectContext) {
    let sortDescriptor = NSSortDescriptor(key: "name", ascending: true)
    let request = NSFetchRequest<Organization>(entityName: "Organization")
    request.sortDescriptors = [ sortDescriptor ]

    fetchedResultsController = NSFetchedResultsController(fetchRequest: request,
managedObjectContext: context, sectionNameKeyPath: nil, cacheName: nil)
    fetchedResultsController?.delegate = self
}
```

Разберем более внимательно, что тут происходит. При создании мы передаем контекст, на котором будут отслеживаться изменения и на котором будет выполняться сам request. Стоит отметить, что наличие хотя бы одного sortDescriptor является обязательным. sectionNameKeyPath используется для разбиения данных на секции. В нашем случае оставим его пустым. cacheName - имя файла для кэширования данных. Также оставим его пустым, что будет означать отсутствие кэширования.

Добавим вызов создания контроллера в didSet контекста:

```
setupFetchedResultsController(for: context)
```

Давайте добавим соответствие делегату. Для этого сделаем extension:

```
extension OrganizationsViewController: NSFetchedResultsControllerDelegate {

}
```

Отлично! Мы создали NSFetchedResultsController. Теперь нужно позвать обновление данных. Для этого в методе fetchData уберем старый код создания FetchRequest и заменим context.fetch() на fetch данных через resultsController:

```
try! fetchedResultsController?.performFetch()
```

Следующим шагом удалим наш массив организаций, а вместо него будем использовать данные из fetchResultsController'а. Для этого исправим методы data source таблицы. Сделать это будет очень просто, т.к. fetchResultsController содержит все необходимые для этого данные.

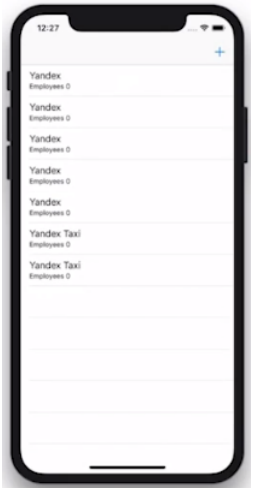
Для исправления метода numberOfRowsInSection воспользуемся numberOfObjects у секции, которая представлена классом NSFetchedResultsSectionInfo:

```
guard let sections = fetchedResultsController?.section else {
    return 0
}
return sections[section].numberOfObjects
```

В методе cellForRowAtIndexPath воспользуемся методом ObjectsAtIndex, чтобы получить организацию:

```
guard let organization = fetchedResultsController?.object(at: indexPath) else {
    return cell
}
```

Давайте запустим проект и удостоверимся, что мы сделали все правильно:



Должен отобразиться список организаций. Теперь реализуем методы NSFetchedResultsController Delegate. Для обновления списка будем использовать методы UITableView: insertRows, reloadRows и deleteRows. Сначала реализуем метод controllerWillChangeContent, который будет вызываться перед обновлением таблицы:

```
func ControllerWillChangeContent(_ controller:
NSFetchedResultsController<NSFetchRequestResult>) {

}
```

Вызовем в нем метод beginUpdate, который предшествует операциям изменения таблицы:

```
tableView.beginUpdates()
```

Затем реализуем метод controllerDidChangeContent, который вызывается после окончания всех изменений:

```
func ControllerDidChangeContent(_ controller:
NSFetchedResultsController<NSFetchRequestResult>) {

}
```

Вызовем в нем метод endUpdates, который подтверждает все изменения в таблице и запускает их анимацию:

```
tableView.endUpdates()
```

Теперь реализуем метод controllerDidChangeAnObjectAtIndexPathForTypeNewIndexPath. Этот метод будет вызываться для каждой добавленной, удаленной или измененной записи, где в поле type будет передаваться тип произошедшего изменения:

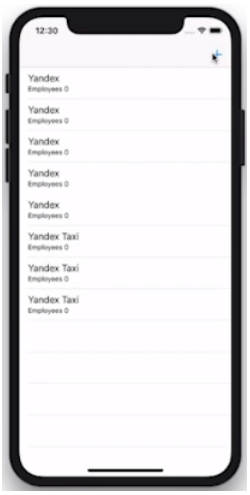
```
func controller(_ controller:
NSFetchedResultsController<NSFetchRequestResult>, didChange anObject: Any, at
indexPath: IndexPath?, for type: NSFetchedResultsChangeType, newIndexPath:
IndexPath? {

}
```

Нам остается только написать switch и вызвать соответствующие методы на таблице. Давайте сделаем это, написав следующий код:

```
switch type {
    case .insert:
        tableView.insertRows(at: [newIndexPath!], with: .automatic)
    case .update:
        tableView.reloadRows(at: [indexPath!], with: .automatic)
    case .move:
        tableView.deleteRows(at: [indexPath!], with: .automatic)
        tableView.insertRows(at: [newIndexPath!], with: .automatic)
    case .delete:
        tableView.deleteRows(at: [indexPath!], with: .automatic)
}
```

Теперь запустим приложение и нажмем на кнопку "добавить" :



Мы увидим, что NSFetchedResultsController получил уведомление о добавлении нового объекта и передал нам эти данные. Поэтому теперь мы видим добавление организации. В этом разделе мы продолжили знакомство с NSFetchedResultsController'ом и использовали его в нашем приложении. В следующем разделе мы поговорим об отношениях в Core Data и добавим сотрудников нашим организациям.

3.6. Отношения. Работа с графом объектов

В этом разделе мы добавим сотрудников в организации. Для начала создадим новый viewController и назовем его EmployeesViewController:

```
import UIKit
import CoreData

class EmployeesViewController: UIViewController {

}
```

Теперь давайте добавим организацию, для которой мы будем просматривать список сотрудников:

```
var organization: Organization!
```

Воспользуемся свойством title у viewController'а для отображения названий организаций в заголовке, написав во viewDidLoad следующее:

```
override func viewDidLoad() {
    super.viewDidLoad
    title = organization.name
}
```

Откроем storyboard и добавим новый viewController:

Зададим ему класс EmployeesViewController, перед этим предварительно собрав проект. Теперь добавим segue к нашему новому viewController'у с типом show. Зададим ему идентификатор - employees. Добавим table view и зададим ей constraint'ы. Выставим data source'ом и делегатом таблицы наш viewController.

Зададим prototype cell. Выставим стиль subtitle. Укажем identifier - EmployeeCellIdentifier. Добавим Navigation Item:

Положим в него Bar Button Item и зададим System Item - Add. Сделаем Action: addEmployee:

```
@IBAction func addEmployee(_ sender: Any {
}
```

Сделаем outlet для tableView:

```
@IBOutlet weak var tableView: UITableView!
```

Откроем EmployeesViewController и напишем следующее:

```
fileprivate let cellIdentifier = "EmployeeCellIdentifier"
private var fetchedResultsController = NSFetchedResultController<Employee>!
```

Создадим новый метод для инициализации fetchedResultsController'а:

```
func setupFetchedResultsController(for context: NSManagedObjectContext) {

}
```

И добавим его вызов во viewDidLoad:

```
setupFetchedResultsController(for organization.managedObjectContext!)
```

Затем напишем:

```
try! fetchedResultsController?.performFetch()
```

Реализуем метод настройки fetchedResultsController уже знакомым нам кодом с тем условием, что теперь сущность будет Employee:

```
let sortDescriptor = NSSortDescriptor(key: "lastName", ascending: true)
let request = NSFetchRequest<Employee>(entityName: "Employee")
request.sortDescriptors = [ sortDescriptor ]

fetchedResultsController = NSFetchedResultsController(fetchRequest: request,
managedObjectContext: context, sectionNameKeyPath: nil, cacheName: nil)
fetchedResultsController?.delegate = self
```

Стоит уделить внимание тому, что теперь нам нужны не все сотрудники, а только конкретной организации. Поэтому добавим предикат:

```
let predicate = NSPredicate(format: "department = %@", organization)
```

```
request.predicate = predicate
```

Добавим extension и реализуем делегат NSFetchedControllerDelegate, где повторим все то же самое, что мы сделали для списка организаций:

```
extension EmployeesViewController: NSFetchedResultsControllerDelegate {
    func controllerWillChangeContent(_ controller:
        NSFetchedResultsController<NSFetchRequestResult>) {
        tableView.beginUpdates()
    }

    func controllerDidChangeContent(_ controller:
        NSFetchedResultsController<NSFetchRequestResult>) {
        tableView.endUpdates()
    }

    func controller(_ controller:
        NSFetchedResultsController<NSFetchRequestResult>, didChange anObject:
Any, at indexPath: IndexPath?, for type: NSFetchedResultsChangeType, newIndexPath:
IndexPath?) {
        switch type {
            case .insert:
                tableView.insertRows(at: [newIndexPath!], with: .automatic)
            case .update:
                tableView.reloadRows(at: [indexPath!], with: .automatic)
            case .move:
                tableView.deleteRows(at: [indexPath!], with: .automatic)
                tableView.insertRows(at: [newIndexPath!], with: .automatic)
            case .delete:
                tableView.deleteRows(at: [indexPath!], with: .automatic)
        }
    }
}
```

Осталось только реализовать табличный data source. Давайте сделаем это, написав следующий код:

```
extension EmployeesViewController: UITableViewDataSource {
    func numberOfSections(in tableView: UITableView) -> Int {
        guard let sections = fetchedResultsController.sections else {
            return 0
        }
        return sections.count
    }

    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int)
-> Int {
        guard let sections = fetchedResultsController.sections else {
            return 0
        }
        return sections[section].numberOfObjects
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
-> UITableViewCell {
        let cell = UITableView.dequeueReusableCell(withIdentifier: cellIdentifier,
for: IndexPath)
        guard let employee = fetchedResultsController?.object(at: IndexPath) else {
            return cell
        }
        cell.textLabel?.text = employee.lastName
        cell.detailTextLabel?.text = employee.position ??
        return cell
    }
}
```

Вернемся к методу addEmployee. Для начала объявим переменную context, которую возьмем из нашей организации:

```
guard let context = organization.managedObjectContext else {
    return
}
```

После чего добавим в контекст новый объект класса Employee:

```
let employee = NSEntityDescription.insertNewObject(forEntityName: "Employee", into:
context) as! Employee
```

И заполним необходимые поля:

```
employee.firstName = "Konstantin"
employee.lastName = "Snegov"
employee.birthDate = Date()
employee.position = "iOS Developer"
```

Теперь добавим сотрудника в организацию:

```
organization.addToEmployees(employee)
```

Этот метод не является частью Core Data, а был сгенерирован для нас при сборке проекта. Если в этом месте не вызвать сохранение контекста, то можно увидеть, как сотрудник добавляется, но при перезапуске его уже не будет. Поэтому добавим сохранение:

```
try! context.save()
```

Откроем OrganizationsViewController и добавим следующий код:

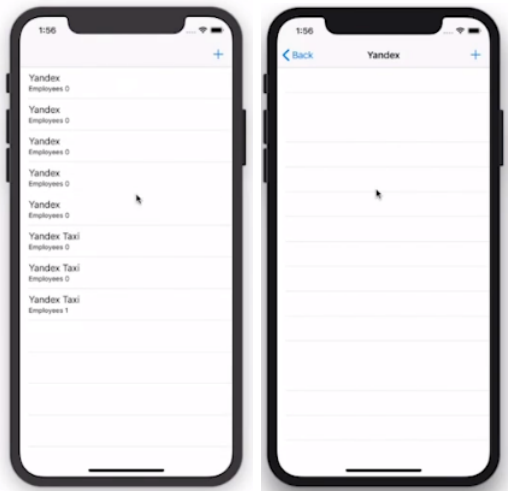
```
fileprivate let employeesSegueIdentifier = "employees"
```

Реализуем метод prepare(for segue:, написав в нем следующее:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any) {
    if segue.identifier == employeesSegueIdentifier,
    let vc = segue.destination as? EmployeesViewController,
    let indexPath = tableView.indexPathForSelectedRow {

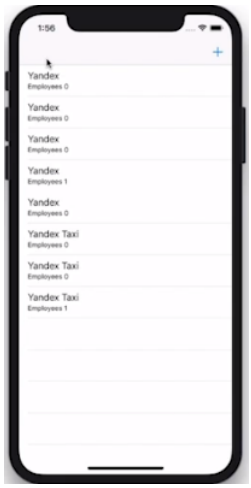
        vc.organization = fetchedResultController?.object(at: indexPath)
    }
}
```

Сначала проверяется идентификатор segue, затем мы получаем viewController для отображения списка сотрудников. Потом получаем indexPath для выбранной ячейки, и вуаля - мы передали организацию во viewController! Давайте запустим наше приложение и попробуем выбрать организацию:



Мы увидим, что наш переход сработал и в заголовке показывается навание организации. Нажмем на +:

И увидим, что появился сотрудник. Если вернуться назад, то также мы увидим, что количество сотрудников у организации поменялось:



В следующем разделе мы подробнее поговорим про memory management объектов в Core Data и про отношения объектов между собой.

3.7. Управление памятью

В этом видео мы погорим про memory management и узнаем, кто держит наши объекты и как они связаны между собой.

Так кто же держит наши NSManagedObject? Их держит NSManagedObjectContext, внутри которого они зарегистрированы. По умолчанию эта связь - слабая, а это значит, что не стоит полагаться на то, что объект, загруженный в context, будет там всегда. Исключения составляют inserted, deleted

и updated объекты, для которых еще не был вызван save(). Если у контекста есть undo manager, он тоже может держать сильные ссылки на объекты.

Поведение контекста может быть изменено установкой retainsRegisteredObjects равным true.

```
var retainsRegisteredObjects: Bool
```

Теперь жизненный цикл managed объектов будет зависить от жизни контекста. Это удобно в случаях, когда можно держать в памяти небольшой объем данных, используя ее как кэш.

Давайте рассмотрим отношения между несколькими NSManagedObject'ами. Каждый объект хранит сильную ссылку на объект, с которым он связан. Отношения между объектами могут порождать retain cycle'ы.

Для того, чтобы разорвать такой цикл, когда вы закончили работать с объектом, достаточно вызвать метод refreshObject:mergeChanges:

```
func refresh(_ object: NSManagedObject,
mergeChanges flag: Bool)
```

Он превратит его в fault. О механизме faulting'а мы поговорим далее.

Отношения в Core Data могут быть двусторонними. В нашем случае если организация имеет ссылку на сотрудников, то сотрудник всегда имеет ссылку на организацию. Исключения составляют fetched property, которые представляют слабую одностороннюю ссылку на объекты. Их рассмотрение выходит за рамки нашего курса. Всегда рекомендуется создавать двусторонние отношения, т.к. Core Data использует этот механизм для проверки целостности графа объектов.

В этом разделе мы поговорили о том, кто держит наши объекты и отношения между ними. Далее мы познакомимся с механизмом Faulting и узнаем, как им пользоваться.

Faulting - это механизм, позволяющий уменьшить размер занимаемой памяти при работе с Core Data. Достигается это тем, что вместо полноценных объектов используются легковесные объекты - faults, которые не содержат никаких данных, кроме ID.

Полноценный объект

Person	
objectID	123
firstName	Иван
lastName	Иванов

Fault

Person	
objectID	123
firstName	null
lastName	null

Это позволяет контексту не занимать много места, но при этом поддерживать уникальность объектов. Fault означает, что объект не был полностью загружен из хранилища. Такой объект будет обладать нужным типом, но его свойства не будут проинициализированны. По умолчанию все связи объектов будут fault.

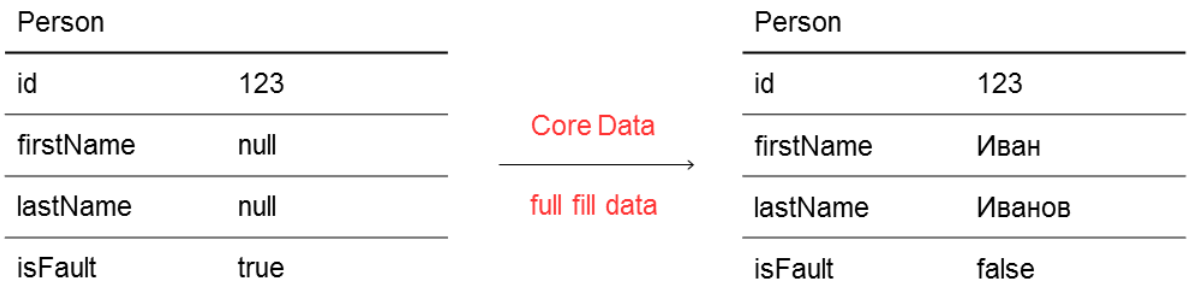
Для того, чтобы понять, является ли NSManagedObject fault'ом, у него есть соответствующее свойство:

```
var isFault: Bool { get }
```

Как же нам сделать полноценный объект из fault? Для этого нам не нужно выполнять fetch requests. Если при обращении к property Core Data понимает, что объект - fault, то она сама загрузит

данные для нашего объекта и проинициализирует его всеми значениями. Этот процесс называется "firing fault":

```
let person: Person =...                    print (person.firstName)
```



Когда происходит firing fault, Core Data не всегда идет в хранилище за данными. Она может попытаться начитать их из кэша. Если данные были найдены в кэше, то инициализация значений будет очень быстрой. В случае же, если данных не было в кэше, то будет создан и выполнен fetch request, результаты которого будут закэшированы. Поэтому firing fault подряд для нескольких значений может быть неэффективным, но об этом - чуть позже.

Также стоит отметить, что есть методы, обращение к которым не приводит к срабатыванию firing fault:

```
isEqual:
hash
superclass
class
self
zone
func isProxy() -> Bool
func isKind(of aClass: AnyClass) -> Bool
func isMember(of aClass: AnyClass) -> Bool
func conforms(to aProtocol: Protocol) -> Bool
func responds(to aSelector: Selector!) -> Bool
description
managedObjectContext
entity
objectID
isInserted
isUpdated
isDeleted
isFault.
```

Например, к ним относится метод description, если он не был переопределен. Также не приводит к firing fault'у запрос количества объектов для связей. Стоит отметить, что нет механизма, который бы загружал объет частями: в случае, если firing fault произошел, будут загружены все свойства объекта, кроме его связей. Связи загружаются только тогда, когда происходит обращение к элементам списка.

Для того, чтобы сделать обратное преобразование из полноценного объекта в fault и тем самым уменьшить объем памяти, стоит вызвать метод refreshObject:mergeChanges:

```
func refresh(_ object: NSManagedObject, mergeChanges flag: Bool)
```

Когда NSManagedObjectObject наполняется значениями, то вызывается метод awakeFromFetch:

```
func awakeFromFetch()
```

Когда он преобразовывается в fault, вызываются методы willTurnIntoFault и didTurnIntoFault:

```
func willTurnIntoFault()
func didTurnIntoFault()
```

В NSFetchRequest есть параметр fetchBatchSize, который позволяет задать количество загружаемых целиком объектов, а остальные вернуть как fault:

```
var fetchBatchSize: Int { get set }
```

Параметр обычно используют вместе с fetchLimit, чтобы загружать данные частями:

```
var fetchLimit: Int { get set }
```

В этом разделе мы познакомились с механизмом faulting и узнали, как он работает. В следующем разделе мы поговорим про валидацию данных.

3.8. Валидация и версионирование данных

В этом разделе мы поговорим про валидацию данных внутри NSManagedObject, рассмотрим, как мы можем ее расширить и когда она вызывается.

Основные правила валидации в Core Data описываются схемой данных. Например, мы можем задавать минимальные и максимальные значения, задавать проверку строк через регулярные выражения. Все эти проверки довольно тривиальные и позволяют проверять только одно свойство.

Более сложные проверки подразделяют на два типа: те, что накладываются на одно свойство, и те, что позволяют провалидировать консистентность всего объекта.

Для валидации одного свойства следует использовать методы протокола NSKeyValueCoding:

```
func validateValue(_ value:
AutoreleasingUnsafeMutablePointer<AnyObject?>,
forKey key: String) throws
```

Для валидации всего объекта, следует переопределить методы validateForInsert, validateForUpdate, validateForDelete:

```
func validateForDelete() throws
func validateForInsert() throws
func validateForUpdate() throws
```

Давайте рассмотрим каждый из этих подходов.

В протоколе NSKeyValueCoding есть метод

```
validateValue:forKey:error:
```

```
func validateValue(_ value:
AutoreleasingUnsafeMutablePointer<AnyObject?>,
forKey key: String) throws
```

Он позволяет провалидировать данные для задаваемого значения. NSManagedObject уже содержит реализацию данного метода, и нам не стоит его переопределять; в нем и просходит валидация значений, описанная в схеме.

Вместо этого стоит реализовать метод

```
validate<Key>:error:
```

где key - название вашего свойства для валидации:

```
func validate<Key>(value:
AutoreleasingUnsafeMutablePointer<AnyObject?>) throws
```

Если мы реализуем такой метод, то нам не нужно будет вызывать его самим, Core Data это сделает за нас, когда будет вызван метод

```
validateValue:forKey:error:
```

Как видно в примере:

```
func validateFirstName(value:
AutoreleasingUnsafeMutablePointer<AnyObject?>) throws {

    let string = value?.pointee as! String

    if string.count > 0 { // Your validation
        return
    }

    let error: NSError = ...
    throw error

}
```

Входящий параметр - это указатель, который преобразовывается в строку. В случае, если преобразование прошло успешно и валидация пройдена, мы просто выйдем из метода, а в случае, если нет, выбросим ошибку. Стоит описывать ошибки валидации очень детально, чтобы потом было понятно, что пошло не так и почему.

Как понятно из названия этих методов:

```
func validateForDelete() throws
func validateForInsert() throws
func validateForUpdate() throws
```

они будут вызваны при добавлении, изменении и удалении объекта. Вы можете переопределить любой из них в зависимости от необходимости.

При реализации своего метода стоит обязательно вызвать super. После вызова super все свойства объекта будут провалидированы и мы можем проверить консистентность всего объекта целиком:

```
override func validateForInsert() throws {
    try super.validateForInsert()
```

```
    try validateConsistency()
}

override func validateForUpdate() throws {
    try super.validateForUpdate()
    try validateConsistency()
}
```

Например, мы можем проверить, что для занятия должности в компании человек должен иметь определенный стаж работы.

Вам не нужно запускать валидацию данных самостоятельно, она запускается самостоятельно при сохранении контекста:

```
context.save()
```

В случае, если валидация не пройдена, вся операция сохранения контекста прерывается и ни одно из изменений не будет сохранено в хранилище. Разрешение ошибок и пересохранение данных остается за вами.

В этом разделе мы познакомились с механизмом валидации данных и узнали, как добавить свои проверки. Далее мы поговорим о версионировании модели данных.

Теперь мы поговорим об изменении схемы данных, ее версионировании и миграции данных.

До сих пор у нас в приложении была только одна версия схемы модели данных, и она не менялась. Представим, что мы выпустили наше приложение в AppStore и поняли, что для добавления новой функциональности нам необходимо изменить схему данных.

Вы можете открыть хранилище Core Data только с той версией модели данных, с которой оно создавалось. При создании стека Core Data проверяет текущую версию модели и версию модели, использованной в хранилище. Поэтому в случае измения модели мы должны изменить данные и в хранилище, чтобы они соответсвовали новой версии модели данных.

Для миграции хранилища необходимы старая и новая модели данных, а также при необходимости надо предоставить описание того, как должен быть произведен маппинг данных из старой версии в новую. Маппинг можно не описывать в случае, если добавились:

- новые сущности;
- необязательные атрибуты;
- атрибуты со значениями по умолчанию.

В случае отсутсвия маппинга миграция будет называться lightweight.

Для создания lightweight миграции не нужно ничего делать, если вы используете NSPersistentContainer. Но если вы добавляете хранилище вручную, то достаточно передать соответствующее описание:

```
let description = NSPersistentStoreDescription()
description.shouldInferMappingModelAutomatically = true
description.shouldMigrateStoreAutomatically = true
```

Теперь давайте создадим маппинг самостоятельно.

Создадим новую версию модели. Для этого перейдем в меню Editor -> Add model version, зададим название модели и нажмем Finish. Чтобы найти новую версию схемы данных, обратим внимание на Project Navigator и развернем схему данных. Теперь видно, что у нас есть две схемы. Выбрав новую схему и открыв инспектор объектов, в разделе Model Version можно задать текущую версию модели. Выберем новую версию.

Добавим в сущность Employee атрибут salary с типом Double. Уберем признак Optional, а так же значение по умолчанию. Зададим размер минимальной заработной платы в размере 100.

Если мы попробуем запустить наше приложение, то оно упадет с ошибкой валидации, потому как схема данных изменилась и Core Data не смогла произвести легкую миграцию существующих данных самостоятельно.

Откроем сущность Person и переименуем firstName в name. Теперь соберем приложение, чтобы сгенерировались новые классы наших моделей. Сборка завершится с ошибкой, т.к. нам надо в коде переименовать name.

Давайте сделаем это. Откроем EmployeesViewController, в методе addEmployee заменим firstName на name, и добавим salary = 300:

```
employee.name = "Konstantin"
employee.lastName = "Snegov"
employee.birthDate = Date()
employee.position = "iOS Developer"
employee.salary = 300
```

А так же в DataSource таблицы напишем следующий код:

```
cell.textLabel?.text = employee.lastName! + employee.name!
cell.detailTextLabel?.text = "\(employee.salary) \(employee.position ??  )"
```

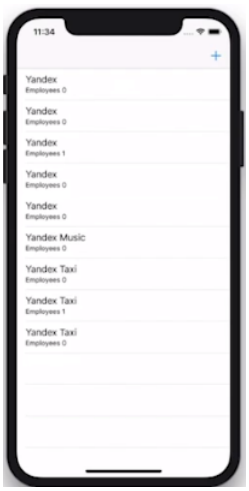
Теперь все готово, чтобы создать наш первый маппинг.

Для этого создадим новый файл: File -> New File -> Mapping Model. Укажем изначальную версию модели и текущую версию. Зададим название и нажмем Create. Мы видим, что для нас сгенерировался маппинг существующих значений и нам необходимо вписать только недостающие. Давайте сделаем это. Выберем Employee и для name напишем следующее:

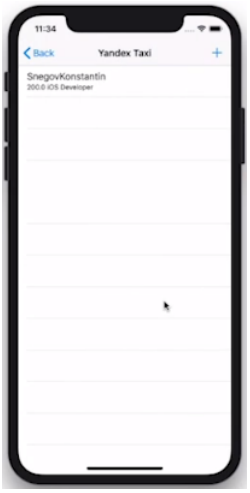
```
$source.firstName
```

Это означает, что из модели источника мы возьмем поле firstName. А для salary напишем константу 200.

Запустим наше приложение:



Как мы видим, теперь приложение запускается, и если мы перейдем к списку сотрудников, то увидим, что для всех существующих сотрудников по прежнему выводится имя и зарплата по умолчанию:



Теперь давайте разберем, что же произошло при запуске приложения? При загрузке стека Core Data NSPersistanceStoreCoordinator загрузил текущую версию схемы данных и проверил версию данных, используемую хранилищем. Поскольку они не совпадают, и lightweight миграция не была возможна, был произведен поиск соотвествующей модели маппинга. После чего была произведена миграция хранилища на новую версию схемы.

Версий моделей данных может быть много. Но для того, что бы работала автоматическая миграция, необходимо предоставить маппинги из всех предыдущих схем в текущую. Например, если в хранилище версия 1, а текущая схема имеет версию 3, то необходимо предоставить маппинг из версии 1 в версию 3 и из версии 2 в версию 3.

Также возможно реализовать свою миграцию данных, но это выходит за рамки нашего курса. Если вы не можете выразить миграцию терминами маппинга, то вам необходимо написать политику миграции самостоятельно, использовав класс NSEntityMigrationPolicy. Его рассмотрение выходит за рамки нашего урока.

Вы познакомились с тем, как создавать версии модели данных и переключать их. В следующем разделе мы рассмотрим работу с несколькими контекстами.

3.9. Многопоточность

В этом разделе мы рассмотрим работу с несколькими контекстами, поговорим, для чего они нужны, научимся создавать их и передавать данные из одного контекста в другой.

Как мы уже говорили ранее, контекст создается вместе с очередью, на которой он будет выполнять свои операции. Есть две очереди, с которыми можно создать контекст:

- PrivateQueueConcurrencyType
- MainQueueConcurrencyType

Контекст с PrivateQueueConcurrencyType будем называть Background Context'ом. До этого мы использовали только контекст, который работает на главном потоке, он был для нас создан PersistantContainer'ом:

```
container.viewContext
```

Для создания background контекста у Persistant Container есть метод newBackgroundContext:

```
func newBackgroundContext() -> NSManagedObjectContext
```

Обычно стоит избегать обработки большого объема данных в главном потоке, а также данных, которые пользователь явно не запрашивал. Например, сохранение данных в Core Data при получении их из сети.

Для нашего с вами примера сделаем добавление новой организации через background context. Для этого откроем OrganizationsViewController и добавим свойство backgroundContext:

```
var backgroundContext: NSManagedObjectContext!
```

Теперь отроем AppDelegate. Создадим там background context и передадим его нашему view controller'у:

```
vc.backgroundContext = container.newBackgroundContext()
```

Вернемся к OrganizationsViewController и удалим весь код в методе addOrganization. Отлично, подготовительная часть закончена и мы готовы сделать сохранение.

Сразу напишем dispatch async и объявим weak - self:

```
DispatchQueue.global(qos: .userInitiated).async { [weak self] in

}
```

Добавим внутрь проверку, что self - не nil:

```
guard let 'self' = self else { return }
```

Создадим болванку организации, но теперь воспользуемся алтернативным методом:

```
let organization = Organization(context: self.backgroundContext)
```

Он делает то же самое что и NSEntityDescription.insertNewObject. Добавим название организации:

```
organization.name = "Yandex Music"
```

Теперь выполним сохранение. Для этого воспользуемся методом performAndWait, который есть у контекста:

```
self.backgroundContext.performAndWait {
    do {
        try self.backgroundContext.save()
    } catch {
        print(error)
    }
}
```

Запустим приложение и нажмем кнопку "Добавить": Ничего не происходит. Почему же? Сохранение произошло успешно, но мы никак не синхронизировали данные между нашими контекстами. Если сейчас мы запустим приложение еще раз, то увидим, что организация добавилась.

Для синхронизации данных мы воспользуемся методом контекста

```
mergeChanges(fromContextDidSave:)
```

Этот метод позволяет смержить данные, которые были сохранены из background context'а в контекст главного потока.

При сохранении Core Data отсылает уведомление NSManagedObjectContextDidSave. Подпишемся на него, написав в методе viewDidLoad следующее:

```
NotificationCenter.default.addObserver(self, selector:
#selector(managedObjectContextDidSave(notification:)), name:
NSNotification.Name.NSManagedObjectContextDidSave, object: nil)
```

Теперь реализуем метод managedObjectContextDidSave:

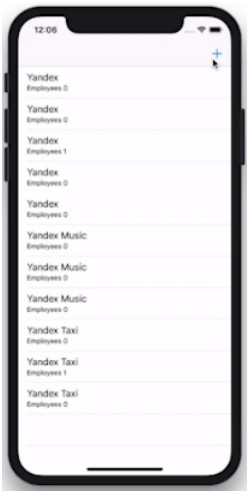
```
@objc func managedObjectContextDidSave(notification: Notification) {

}
```

Поскольку уведомление может поступать из разных потоков, то стоит вызывать мерж изменений внутри контекста главного потока. Для этого воспользуемся методом perform:

```
context.perform {
    self.context.mergeChanges(fromContextDidSave: notification)
}
```

Запустим наше приложение и попробуем добавить организацию. Теперь мы видим, что она добавляется:



Поговорим о конфликтах. Как только мы начали работать с несколькими контекстами, при сохранении могут появляться конфликты. Например, если мы поменяли одни и те же данные в двух контекстах. При сохранении Core Data проверяет, есть ли конфликты в данных, и если они есть, то она использует merge policy для их разрешения. Если вы не указывали никакую политику, то будет использована политика по умолчанию - NSErrorMergePolicy.

Есть несколько предустановленных политик, которые покрывают большинство случаев:

- NSErrorMergePolicy. Данная политика просто вызывает ошибку, при мерже данных.
- NSRollbackMergePolicy. Сохраняет измения в данных, кроме тех, в которых есть конфликты.
- NSOverwriteMergePolicy. Сохраняет все данные и перезатирает данные хранилища теми, которые есть в памяти.
- NSMergeByPropertyStoreTrumpMergePolicy. Сохраняет данные для каждого атрибута, а в случае конфликта будут взяты данные из хранилища.

- NSMergeByPropertyObjectTrumpMergePolicy. Сохраняет данные для каждого атрибута, а в случае конфликта будут взяты данные из памяти.

В случае, если вам не подходит ни одна из политик, вы можете усналедовать более подходящую вам и кастомизировать ее поведение.

Чтобы задать merge Policy, у контекста есть одноименное свойство. Например, это можно сделать вот так:

```
context.mergePolicy = NSMergeByPropertyObjectTrumpMergePolicy
```

Если хотите более подробно узнать про политики и их самостоятельную реализацию, то можно обратиться к документации Apple.

Также еще раз стоит вспомнить, что не стоит передавать NSManagedObject между контекстами напрямую: это может привести к порче данных и крешу приложения. Для передачи между контекстами стоит использовать NSManagedObjectID.

В этом разделе мы выполнили простой пример сохранения данных на background контексте, познакомились с методом синхронизации и поговорили про политики разрешения конфликтов.

Нам этом наше знакомство с Core Data подошло к концу. Core Data - очень большая тема, которая одна может лечь в основу целого курса. Мы с вами затронули только самые основы. Для более подробного изучения данного фреймворка стоит прочитать официальную документацию, а также другую литературу. Например, книгу "Core Data"Florian Kugler'a.

На этом я прощаюсь с вами, и до новых встреч!