Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования
«Московский физико-технический институт
(государственный университет)»

Факультет управления и прикладной математики
Кафедра теоретической и прикладной информатики

ИСПОЛЬЗОВАНИЕ CLOUDKIT В КАЧЕСТВЕ СРЕДСТВА СИНХРОНИЗАЦИИ ДАННЫХ COREDATA

Выпускная квалификационная работа
(бакалаврская работа)

Направление подготовки: 03.03.01 Прикладные математика и физика

Выполнил:
студент 376 группы _____ Копырин Денис Валерьевич

Научный руководитель:
асс. _____ Соболев Артемий Анатольевич

Москва, 2017

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современном мире у каждого человека есть множество электронных устройств, которые обрабатывают информацию. Передача данных между различными устройствами является нетривиальной и актуальной задачей, необходимой для работы с ними. В данной работе рассматривается один из оптимальных способов синхронизации данных между гаджетами Apple или процессами на устройстве.

Для хранения информации на отдельно взятом устройстве Apple предоставляет API для постоянного хранилища “CoreData”. Это API дает возможности работы с различными базами данных, например, с SQLite. В виду его обобщенности, CoreData можно использовать не только для постоянного хранилища, но также и для работы с in-memory базой данных.

С другой стороны, для передачи данных между устройствами используется другой API от Apple – CloudKit. Он предназначен для работы с удаленным реляционным хранилищем key-value записей iCloud, который предоставляет удобное средство для обмена данных не только между устройствами одного пользователя, но и для совместной работы с данными.

Одно из требований к решению данной задачи было сделать работу с CloudKit как можно меньше видимой для пользователя, в пределе сделать CloudKit “прозрачным” для пользователя. В виду востребованности средства синхронизации между устройствами, уже существуют подобные решения, сравнение с которыми также было произведено в этой работе.

Целью настоящей работы являлось написание прослойки между двумя API от Apple и создание удобного средства для синхронизации данных, а также оценка оптимальности полученного решения и сравнение его с конкурентными методами обмена данных.

В ходе выполнения дипломной работы были решены следующие задачи:

– изучен стек технологий CloudKit и CoreData;

– разработана формальная схема передачи информации между двумя API и написан программный продукт;

– установлено и настроено программное обеспечение, реализующее выбранные для сравнения технологии;

– разработаны тестовые программные средства, и проведены вычислительные эксперименты;

– проведен сравнительный анализ с другими аналогичными решениями для синхронизации данных.

По завершении выполнения дипломной работы получены рабочая схема синхронизации данных между устройствами и реализована в проекте Swedo.

Результаты дипломной работы изложены в пояснительной записке, состоящей из введения, трёх глав, заключения и приложения.

Обзор литературы

Статьи [1], [2] дают представление о структуре фрэймворка Core Data, [3], [4] описывает теоретическое обоснование эффективности Core Data. Статьи [5], [6] рассказывают о принципах работы CloudKit и iCloud, а также показывают их возможное применение. Сайты [7] и [8] являются официальной документацией Apple для Core Data и iCloud соответственно. Ссылка [9] является примером решения задачи, представленного в работе, с которым производилось сравнение.

ГЛАВА 1. ОПИСАНИЕ COREDATA

1.1. Введение

В апреле 2005 года, компания Apple выпустила OS X 10.4 вместе с которой появилась библиотека для работы с данными CoreData. Несмотря на то что существуют другие средства хранения информации, CoreData отлично подходит для работы с Cocoa Framework.

CoreData помогает в создании уровня приложения, отвечающему за хранение данных, а именно связи между объектами подобно обычной реляционной базе данных. Однако, CoreData позволяет не только сохранять или получать данные с диска, но и работать с данными, находящимися в памяти.

CoreData это не только ORM (Object-Relation Mapping), но и средство для создания графа связей между объектами. При этом, CoreData никак не завязан на UI и поэтому может использоваться для сохранения данных в бэкграунде. [2]

Рассмотрим простейший граф данных CoreData рис. 1.1.

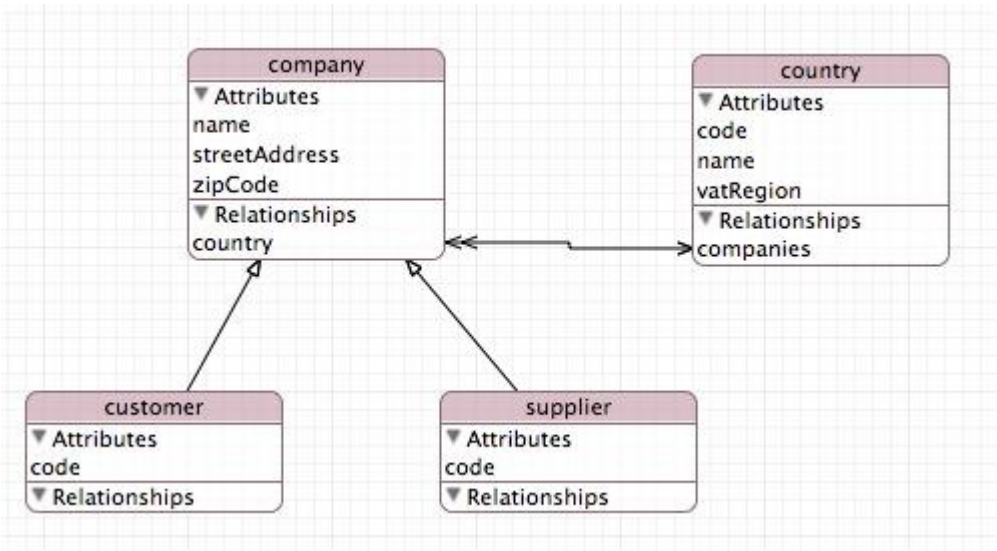

Рис. 1.1. Граф данных в CoreData

Как и в обычной реляционной базе данных, в CoreData можно создавать **сущности** (Customer, Company, Country, Supplier), **атрибуты** у каждой сущности (name, streetAddress, zipCode у Company) и **связи** (1:n между Company и Country). Такая структура данных остается в базе данных любого поддерживаемого типа.

1.2 CoreData Stack

Рассмотрим компоненты, входящие в CoreData под названием *CoreData Stack.* [3] Приведем принципиальную схему работы CoreData (Рис 1.2)

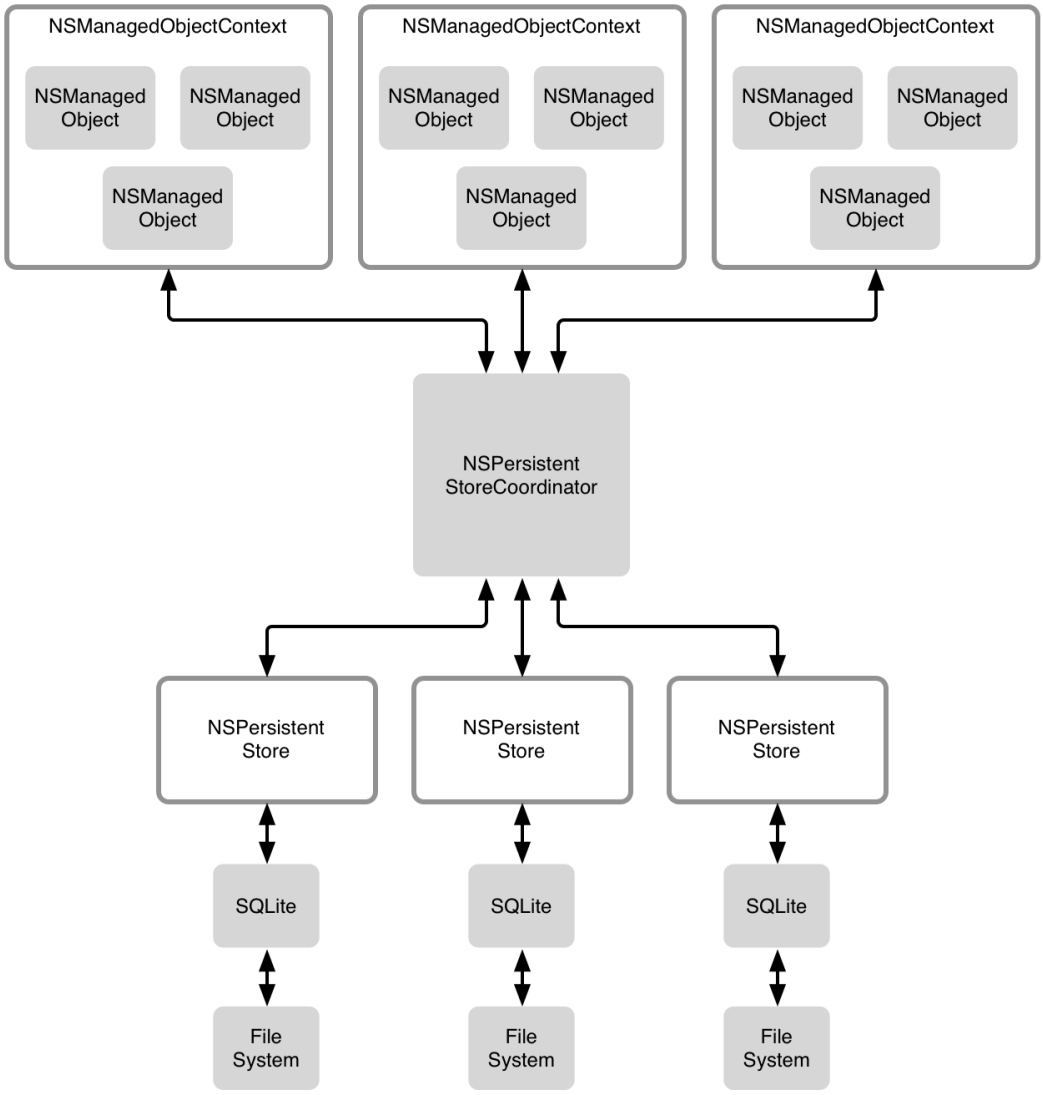

Рис. 1.2. CoreData Stack

Как видно из рисунка, в схеме можно выделить две основные части. Одна часть отвечает за представление графа объектов. Вторая часть

отвечает за хранение данных на диске. Связь между ними осуществляется при помощи NsPersistentStoreCoordinator.

1.2.1 Представление графа объектов

Представление графа объектов - это то, где будет работать логика уровня модели приложения. Объекты модельного уровня (подклассы NSManagedObject) находятся внутри контекста (NSManagedObjectContext). Обычно разрабочики используют один контекст, и все объекты находятся в этом контексте, тем не менее возможно создание нескольких контекстов, в том числе создание дочерних контекстов. Важно понимать, что каждый объект в модели привязан к какому-нибудь контексту и, наоборот, контекст знает какие объекты привязаны к нему, объекты можно передавать между контекстами.

Также контекст является очень удобным средством для перехвата и обработки событий изменения состояния объектов в базе данных. При помощи дочерних контекстов можно добавлять данные в контекст из других потоков и при помощи средства нотификаций можно получать информацию о состоянии объектов

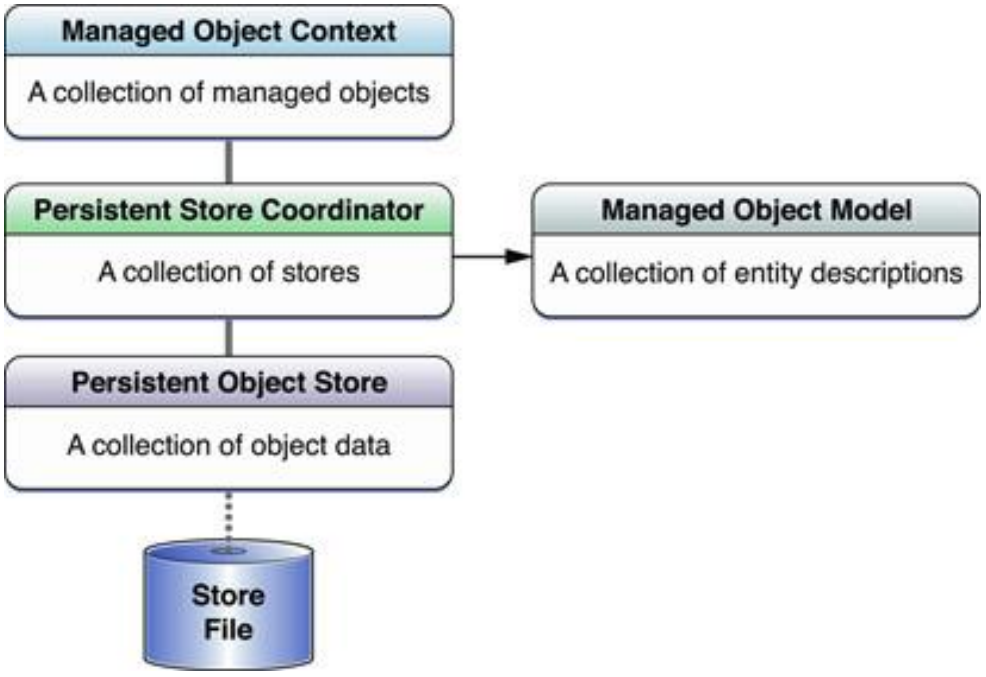

Рис 1.3: Persistent часть стека CoreData

1.2.2 CoreData Persistency

Другая часть стека – часть, которая поддерживает постоянство данных – часть, которая читает и пишет на файловую систему. Почти во всех случаях постоянный координатор хранилища (PersistentStoreCoordinator) содержит одно постоянное хранилище (PersistentStore), которое подключено к нему, и это хранилище взаимодействует с базой данных SQLite на файловой системе. Для нашей задачи потребуется работать не только с базой данных SQLite, но также и с виртуальным in-memory хранилищем данных, которое реализуется при помощи инкрементального хранилища (NSIncrementalStore).

При запуске нашей программы происходит инициализации обеих частей CoreData, подгружается SQLite база данных, оптимизируется при изменении модели и устанавливается изначальный NSPersistentStore для нее. Далее происходит обычная работа с NSPersistentStoreCoordinator через ManagedObjectContext.

1.3 Объекты в памяти

В этой части будет представлены основные методы и поля для работы с объектами в памяти (**NSManagedObject**) и контекстом (**NSManagedObjectContext**).

Начнем с рассмотрения объектов в памяти. NSManagedObject – generic class, который дает основные методы для реализации объекта модели в CoreData. Обычно, объекты, которые используются для работы наследуются от NSManagedObject и реализуют методы, необходимые для его корректной работы, а также объявляют хранимые поля. В последних версиях Xcode, эти классы создаются автоматически. На рис. 1.2 приведен пример подобного объявления.

Для данной работы достаточно использовать инициализатор в контекстом, который создает in-memory объекты для CoreData. Для его

работы необходимо знание о **NSEntityDescription** и **NSManagedObjectContext**.

Дадим описание для NSEntityDescription. NSEntityDescription – это объект, который соответствует каждому сущности в базе данных CoreData. Каждый экземпляр объекта описывает имя сущности и его свойства (атрибуты и связи). Создавать экземпляры объектов можно при помощи методов NSPersistentStoreCoordinator по имени сущности.

Самую главную роль в работе объектов в памяти несомненно играет NSManagedObjectContext, который следит за состояниями объектов и позволяет их сохранять на диск через запросы к NSPersistentStoreCoordinator. Рассмотрим основные решения, которые используются в контексте:

Life-Cycle Management. Контекст играет центральную роль в жизненном цикле объектов в памяти. Он должен позволять подтверждать корректность объектов, валидировать связи между объектами. Для пользователя видны методы получения, сохранения и изменения объектов.

Parent Store. Любой контекст должен иметь родительское хранилище, где он будет сохранять данные, полученные через in-memory объекты. Обычно, контексты имеют NSPersistentStore в качестве родительского хранилища, но начиная с OS X 10.7, родительское хранилище может быть другой контекст. Это может быть нужно для работы в другом потоке с контекстом: для данного потока создается контекст, родительское хранилище которого является наш контекст. При использовании только дочернего контекста, целостность родительского контекста не будет нарушаться. При сохранении дочернего контекста, родительский контекст обновится, не нарушая целостности in-memory объектов. В зависимости от ситуации, можно или сохранить данные в контексте через perform для родительского контекста или не вызывать функцию сохранения, что позволяет делать отложенное сохранение на диск.

Notifications. При изменении, создании или удалении объектов контекст отправляет нотификацию о событии для пост-обработки. Заметим, что так можно делать пост-обработку изменения объектов, но в виду того, что нотификации посылаются до сохранения, рекомендуется изменять только объекты в памяти. Для “долгих” операций можно обрабатывать другую нотификацию: нотификацию на сохранения контекста, которая отправляет объекты в памяти на NSStore.

Concurrency. Каждому контексту обязан соответствовать лишь один поток, поэтому для конкурентного доступа требуется создание дочернего контекста, который будет относится к родительскому контексту. При сохранении на дочернем контексте требуется понимать какие запросы будут выполняться в каком порядке. Это можно настроить в типе очереди на запросе: Confinement, Private или Main queue. Для сохранения данных в родительском контексте нужно использовать perform:

1.4 Виды Persistent Store

CoreData предоставляет различные возможности для хранения данных. Основные 4 варианта хранения данных - SQLite, Binary, XML, and In-Memory. Первые 3 из них являются средствами для постоянного хранения данных, а последний работает в памяти. За это отвечает класс NSPersistentStore, который может работать только в пределе этих 4 перечисленных типов.

Для создания нестандартных типов хранения информации использовать NSPersistentStore нельзя из-за того, что нельзя переопределять методы этого класса. Для решения подобной задачи есть два других класса: **NSAtomicStore** и **NSIncrementalStore**, которые наследуются от NSPersistentStore. Рассмотрим эти классы подробнее.

NSAtomicStore отличается от NSIncrementalStore методами, которые нужно реализовывать для их работы. AtomicStore похож по своему поведению на контекст, он должен предоставлять методы добавления узлов (**node**) и их сохранения. IncrementalStore также позволяется добавлять новые объекты, но в ответ дает значения, при помощи которых можно сохранять объекты, однако IncrementalStore не дает функции сохранения и ожидает, что сохранения происходят инкрементально. [5]

ГЛАВА 2. ОПИСАНИЕ CLOUDKIT

2.1 Введение

Для разработчиков приложений как на мобильные платформы, так и на стационарные решения, необходима возможность работы не только на локальных устройствах, но и в облачных хранилищах. Существует множество возможностей для хранения данных, Apple также предоставляется свой решение под названием *iCloud*.

iCloud – облачное хранилище и сервис по облачным вычислениям, выпущенный 21 Октября 2011. Он дает возможность хранения разнородных данных, привычных для пользователя: документы, изображения, музыку, заметки и т.п. Сервис предоставляет метод синхронизации данных для устройств, а также wireless backup данных с носимых устройств в iCloud и последующее их восстановление на другие устройства. [6]

Множество различных сервисов построено вокруг iCloud и поэтому появление API для удобной работы с ним было необходимо. Таким API стал **CloudKit**. Фрэймворк обеспечивает аутентификацию, работу с публичной и приватной базами данных и key-value хранилище с отображением в in-memory объекты.

Рассмотрим причины, по которым в данной работе был выбран именно CloudKit как основное средство для синхронизации данных.

Простота. Для работы с CloudKit не требуется какой-либо нетривиальной установки, потому что фрэймворк полностью интегрирован в Xcode и для его изначальной работы только лишь требуется аккаунт разработчика и credentials для iCloud хранилища. Всю аутентификацию CloudKit производит сам. API для разработки также является тривиальным.

Надежность. Еще одно преимущество CloudKit это надежность хранения данных, CloudKit позволяет удобно организовать хранение данных приватных данных пользователя и основывать безопасность

приложения на проверенных решениях от Apple вместо реализации своей системы.

Цена. Для каждого разработчика в программе от Apple дается относительно быстрый и дешевый контейнер. Изначально, в бесплатный лимит входит 10Gb “сырых” данных и 100Mb в базе данных, при этом со временем работы бесплатный лимит увеличивается до соответственно 1Pb и 10Tb что достаточно для оптимальной работы малых приложений.

2.2 Структура фрэмворка

Рассмотрим более подробно структуру фрэймворка CloudKit. CloudKit предоставляет разработчикам возможность хранить данные в key-value хранилище и базу данных, данными из которой пользователи могут обмениваться и делиться. В базе данных определяется записи (*CKRecord*) различных типов. Типы позволяется разделять записи в базе данных и определяют какие именно данные и какие поля имеет та или иная запись. Каждая запись — словарь ключей-значений, каждое значение в словаре дает информацию об одном поле в записи. Поля могут хранить простейшие значения как строки, числа и т.п. или более нетривиальные данные как местоположения или ссылки. Все постоянные данные в CloudKit должны быть представлены в *схеме CloudKit.*

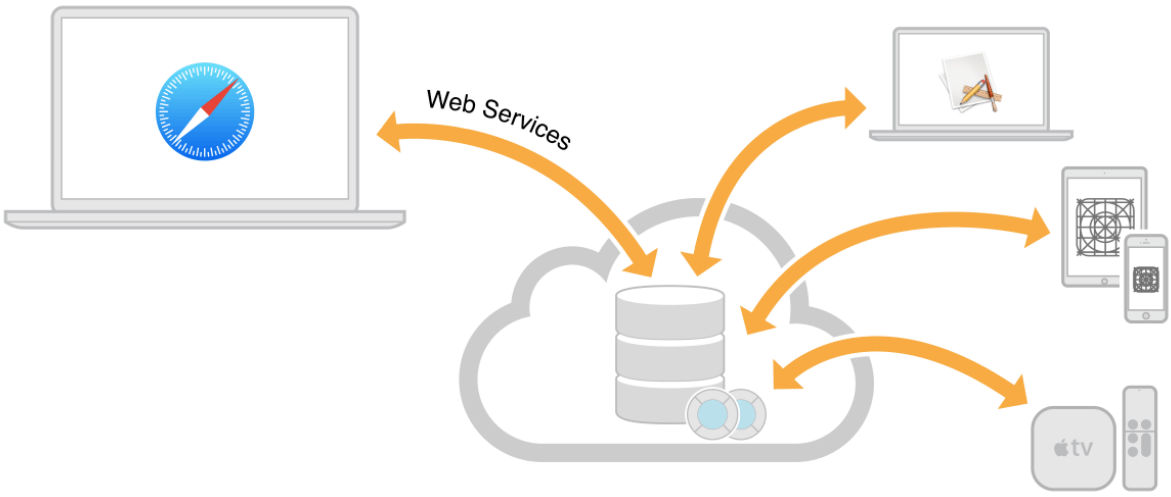

Рис 2.1: Принципиальная схема работы CoreData

Заметим, что вся работа на стороне клиента должна быть сделана через CloudKit, а именно отправка данных пользователя на сервер и их получение, обработка приходящих нотификаций об изменении данных на сервере “со стороны” и получение обновленных данных. При этом обратим внимание на то, что на клиентском устройстве формат хранения данных может быть совершенно другой и поэтому конвертацию данных разработчик должен проводить сам. [7]

2.3 Общая модель данных

Основной концепт в модели данных CloudKit – *контейнер* (CKContainer). CKContainer является классом, который инкапсулирует данные приложения и отвечает определенному приложению на сервере iCloud. Контейнер сохраняет и принимает данные, отправляется из приложения, включая контент, который доступен публично (publicDatabase), приватно (privateDatabase) и разделенно (sharedDatabase). Поэтому можно считать, что контейнер обеспечивает связь между приложением и сервером с данными.

Определим отличия между различными базами данных. По своему применению, приватная и публичная базы данных сходны друг с другом, однако разделенная база данных не позволяет создавать новые записи в отличии от предыдущих двух. Поэтому сначала будем рассматривать базы данных, в которых разрешено создание новых записей.

Приватная база данных разрешает создание дополнительные *зоны записей* (CKRecordZone). При помощи нестандартных зон можно разделять записи по различным категориям проще, инкапсулировать и разделять различные записи друг от друга. Кроме того, нестандартным зонам можно предоставить нестандартные возможности, а именно инкрементальное получение данных (fetchChanges), атомарное изменение базы данных для многих записей (atomic) и возможность разделенения (sharing).

Публичная же база данных имеет ограниченные возможности и в виду общей доступности записей внутри базы данных. Именно поэтому в публичной базе данных запрещено создание нестандартных зон с записями, что приводит к ограниченным возможностям публичной базы данных для инкрементального скачивания зоны, а также атомарной записи множества записей и распределения записей.

Для получения доступа к данным приватной базы другими пользователями используется возможность зоны с записями “sharing”. С ее помощью, пользователь может “переместить” запись в разделенную базу данных. Обратим внимание, что зоны в распределенной базе данных, как и обычные записи нельзя создавать со стороны пользователя, но они создаются сами в зависимости от пользователя, который разделил записи и от оригинальной зоны в приватной базе данных. Как уже упоминалось, разделять записи из публичной базы данных запрещено.

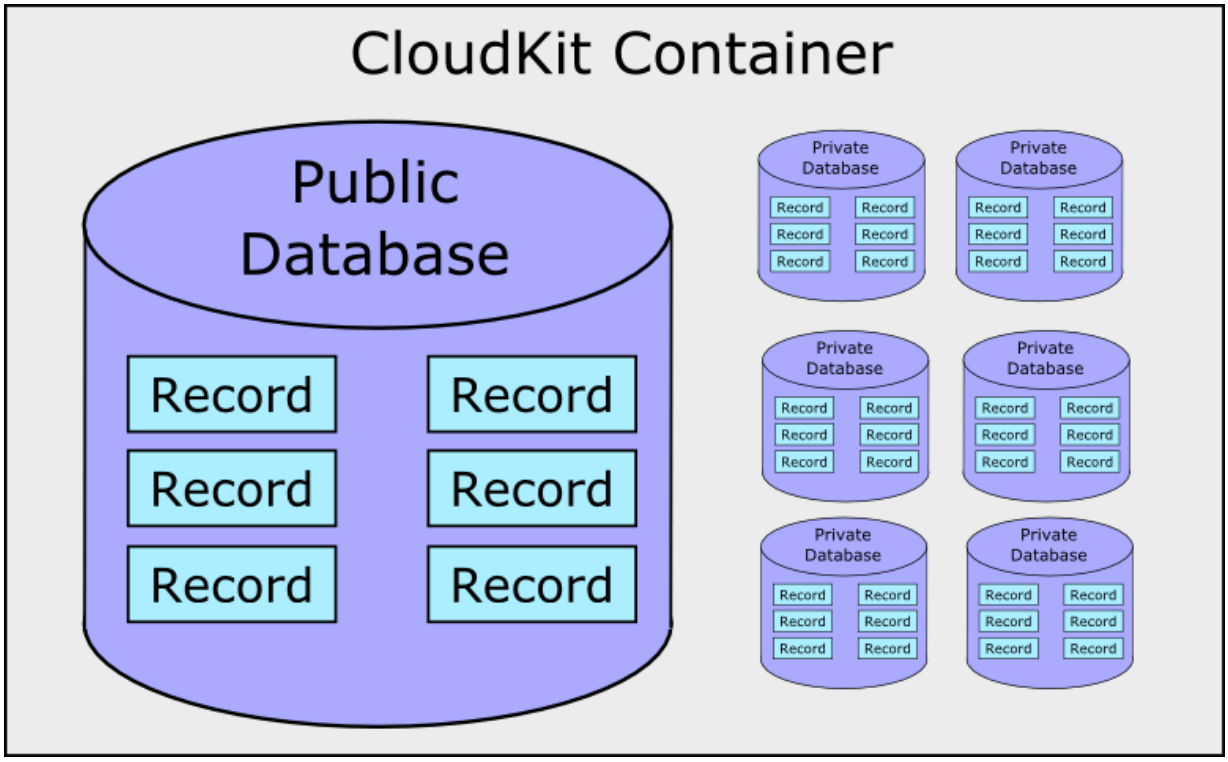

Рис 2.2: Модель данных CloudKit

ГЛАВА 3. СИНХРОНИЗАЦИЯ ОБЪЕКТОВ

3.1 Модель данных CoreData - контекст

Рассмотрим модель, используемую для синхронизации одного объекта в одной таблице базы данных CoreData. Такое рассмотрение будет является также полным для синхронизации многих объектов в пределах одной таблицы без наличия дополнительных связей между объектами. При наличии нескольких таблиц, будет необходимо также рассмотрения связей между таблицами, а именно необходима синхронизация связей между объектами в различных таблицах.

Для простоты, будем считать, что модель данных CoreData и модель данных iCloud уже настроены и готовы для работы. Наша задача на данном этапе – настроить инфраструктуру обмена информацией между двумя несвязанными структурами хранения данных. Введем для этого понятие Linker – связующее звено между конкретным устройством с базой данных CoreData и облачным хранилищем iCloud. Его задача будет поддержание структуры данных CoreData обновленной и синхронизированной с iCloud.

Для обновления данных из CoreData, будем использовать средство, описанное в Главе 1: Notifications. Для этого нам будет необходимо обрабатывать приходящие сообщения об изменении состояния контекста CoreData, изменять in-memory объекты в Linker, а на сохранении отдавать данные обратно в CoreData.

Для получения данных из CoreData, можно воспользоваться различными API, разработанными Apple. При этом, для инициализации получения данных из CoreData, существует широко известный метод push-notification, приходящий на данное приложение от серверов iCloud, при получении которых, приложение должно инициировать получение данных с iCloud. Для того, чтобы построить подобную модель, необходимо понять, как работает контекст CoreData. Рассмотрим следующие рисунки.[8]

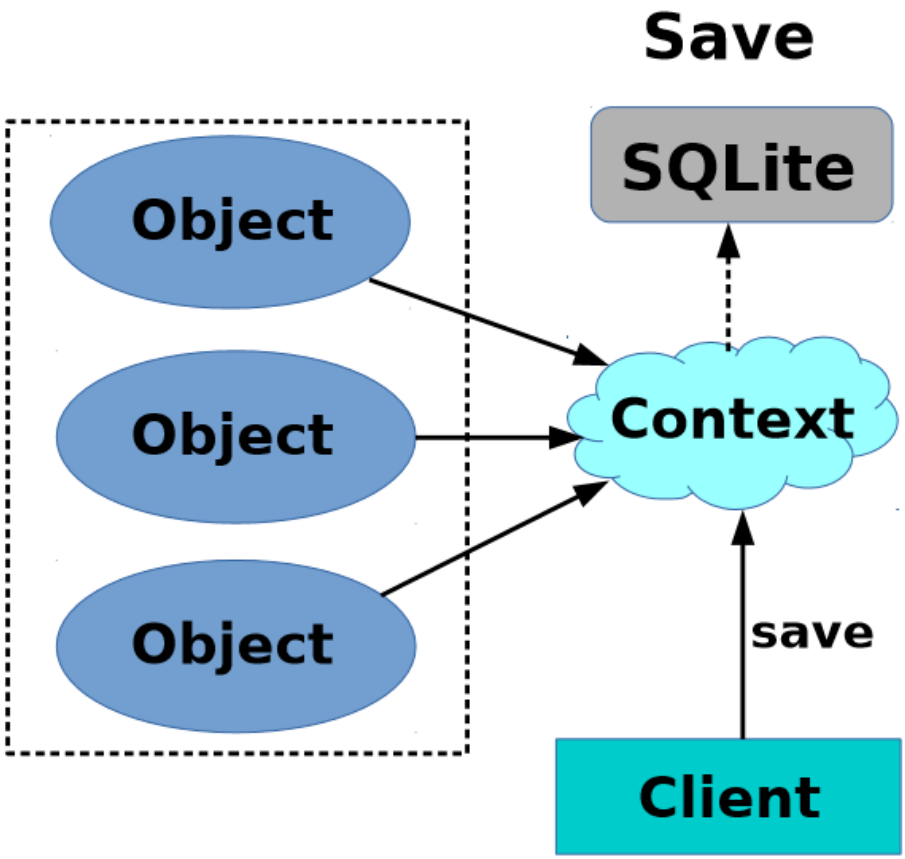

Рис 3.1: Схема сохранения данных при помощи контекста

При работе с CoreData разработчику приложения нужно уметь сохранять и получать данные из хранилища данных. Для этого каждый in-memory объект в CoreData привязан к своему контексту. Когда приложению необходимо произвести сохранение данных из памяти в постоянное хранилище, клиент вызывает у контекста метод “save”, которые производит сохранение всех привязанных к нему объектов в хранилище.

Для работы модели синхронизации, нам будет необходимо перехватывать события сохранения данных. Производить это можно двумя различными способами. В этой Главе будет рассмотрен способ перехвата событий при помощи нотификаций, которые генерируются при изменении любых in-memory объектов, а также их сохранение. Более подробно это будет рассмотрено в главе 2.3.

Теперь опишем загрузку данных из постоянного хранилища. Для этого воспользуемся рисунком 3.2.

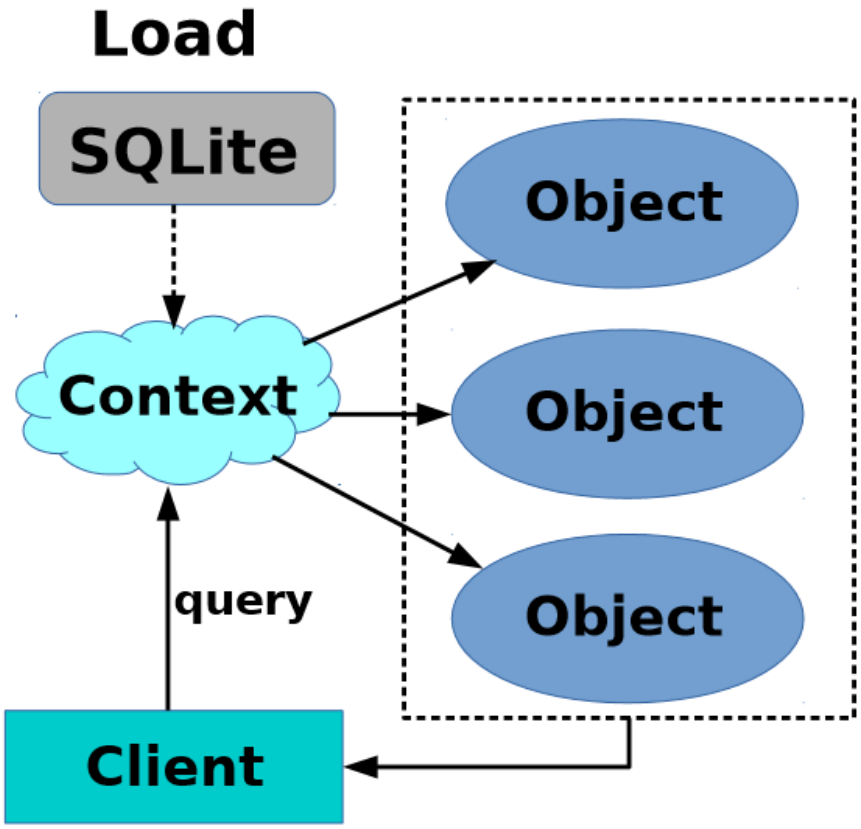

Рис 3.2: Схема загрузки данных при помощи контекста

Схема получения данных из постоянного хранилища сходна со схемой сохранения данных. Однако, теперь для инициации получения данных из CoreData требуется создавать *запрос* к базе данных по предикату. Запросы в базу данных по предикатам необходимо совершать асинхронно, при окончании запроса исполняется пользовательское замыкание, в котором разработчик получается новые in-memory объекты, привязанные к данном контексту.

Как и раньше, изменение полученных новых объектов создает нотификации, которые можно обрабатывать для решения задачи, поставленной в данной работе.

В пределах данной главы, нотификаций от изменений объектов будет достаточно для осуществления простейшей синхронизации объектов в CoreData.

Опишем действия, которые необходимо делать в случае изменения и сохранения объектов, привязанных к CoreData.

didChange: Изменение объектов в CoreData можно разделить на создание, удаление или изменение без создания. При появлении нового in-memory объекта CoreData, нам потребуется объявить соответствующий ему in-memory объект в CloudKit, а именно сгенерировать для данного объекта CoreData GUID, которые будем сохранять в данном объекте для последующей возможности его синхронизации. При удалении объекта по сохраненному ранее при создании GUID, находим in-memory объект и записываем его для дальнейшего удаления. При изменении без создания процедура примерно такая же, как и при удалении, но необходимо записать еще и поля нового in-memory объекта. В итоге, в модели Linker сохранены in-memory объекты CloudKit, которые необходимо сохранить или удалить.

didSave: Можно с хорошей вероятностью считать, чтобы операция сохранения данных выполняется сильно медленнее, чем операция создания объектов в памяти, которую можно оценить отношением скорости записи на диск к скорости записи в RAM. Поэтому в “хорошей программе”, операции сохранения будут вызываться реже, чем операции изменения объектов в памяти. Привяжем сообщения о сохранении данных в постоянное хранилище к отправке данных на сервер iCloud. Для этого требуется воспользоваться сохраненными ранее в операции didChange объектами CloudKit, создать операцию модификации базы данных iCloud и асинхронно ее исполнить. Если операция не удалась, необходимо “отложить” плохие объекты и попробовать отправить их снова позже.

Обратим внимание, что контексты для работы в фоновом режиме и при пользовательском режиме отличаются, поэтому необходимо ставить подписываться на сообщения как для фонового контекста, так и для видового контекста.

Проблема со многими контекстами усложняется при обновлении данных, пришедших с iCloud. В этой ситуации необходимо отличать режим, в котором работает приложение, что можно сделать при помощи средств CoreData, а именно необходимо получать текущий контекст из NSPersistentContainer. Поэтому для работы модели недостаточно сохранять контекст, но сохранять весь container или не обновлять данные в фоне.

Принципиальная схема работы загрузки данных в iCloud представлена на рисунке 3.3

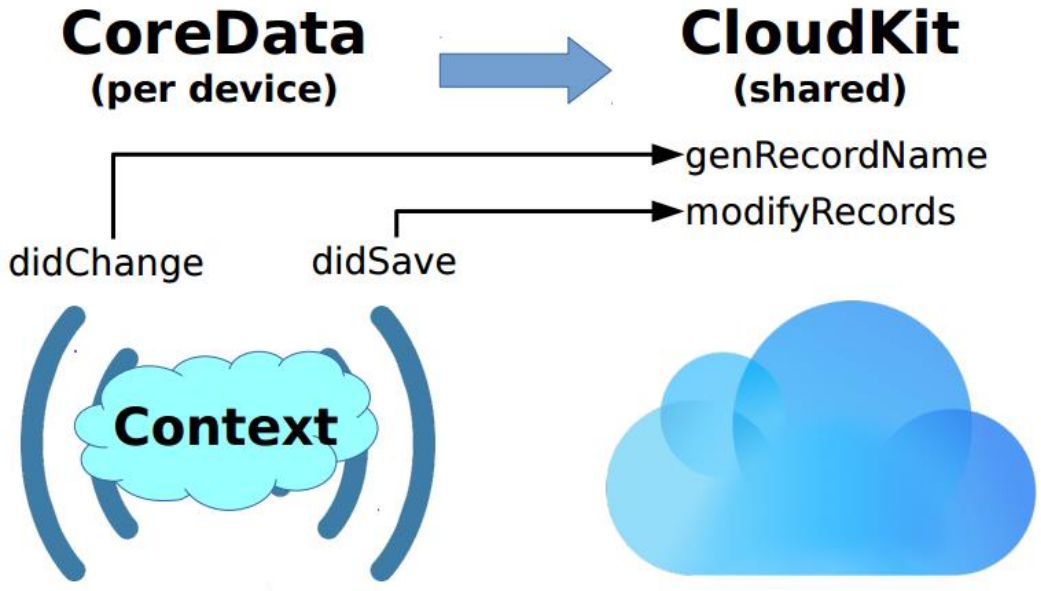

Рис 3.3: Принципиальная схема отправки данных на iCloud

3.2 Получение данных из iCloud

Как было сказано раньше, Apple предоставляет разработчикам множество различных средств для получения информации из базы данных iCloud. В данной главе будут представлены самые оптимальные для задачи синхронизации методы и представлены ограничения на их использование. [9]

Классический способ получения данных из iCloud является создание запроса на поля из записей в базе данных по их GUID. Однако, данный подход ресурсоемкий, потому что неизвестно какие именно данные были изменены и приходиться запрашивать все данные по записи. Тем не менее, данный подход является надежным и не зависит от того какая база данных используется или какие возможности имеет зона. Поэтому данный подход все равно необходимо рассматривать, так как он является универсальным.

Операция, которая осуществляет выборку записей из iCloud, называется CKFetchRecordsOperation. С ее помощью можно запрашивать любые записи из данной зоны, как по полям, так и целые записи. При окончании работы операции, вызываются соответствующие коллбэки на обработку результатов: по отдельным записям perRecordProgressBlock, по окончанию perRecordCompletionBlock и по прогрессу загрузки perRecordProgressBlock.

При таком подходе узнать какие поля из записей нужно получить невозможно из-за ограничений API. Тем не менее, для изначальной загрузки данных на устройства клиента, данной API является удобным.

Для получения обновлений с данной зоны, Apple предоставляет другой интерфейс, который является более оптимальным, по сравнению с универсальным решением, потому что на стороне сервера определяется какие поля и записи нужно послать клиенту. Рассмотрим операцию CKFetchRecordZoneChangesOperation.

Как и в случае с универсальным решением, клиенту необходимо предоставить зону или несколько зон для которой нужно получить новые записи с параметрами. Аналогично, для каждой указанной зоны, разработчик может выставить для получения желаемые поля desiredKeys, но теперь количество записей, полученных с сервера можно ограничивать при помощи resultsLimit. Также, для операции можно выставить флаг fetchAllChanges, который определяет будет ли приложение отправлять дополнительные запросы на сервер.

Для работы операции, нужно определить замыкание, которое будет изменять токен последнего изменения recordZoneChangeTokensUpdate и замыкания для обработки новых записей: изменение recordChangedBlock; удаление recordWithIDWasDeletedBlock и окончание recordZoneChangesCompletionBlock. Схема получений обновлений представлена на рисунке 3.4. Устройства 1 и 2 производят изменения в базе данных от состояний “a” до состояний “e”. Устройство 3 исполняет операцию CKFetchRecordZoneChangedOperation с токеном “a”. Операция производит два запроса к базе данных iCloud с обновлением токена и останавливается в состоянии “e”. Устройство 2 вносит дополнительные изменения до состояния “i”. Устройство 3 сохраняет последний токен по коллбэку “e”, и в следующий раз будет обращение с токеном “e” для получения оставшихся обновлений до токена “f”.

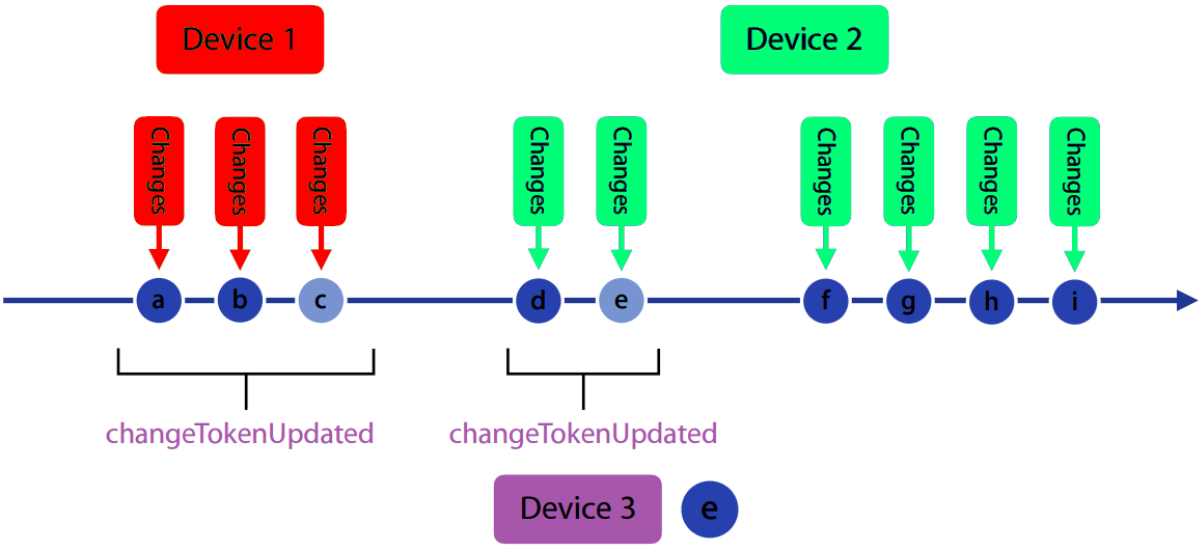

Рис 3.4: Схема получения данных

3.3 Сообщения о синхронизации

Для инициализации получения данных из iCloud Apple предоставляет обобщенное API для получения push-notification со стороны сервера. В случае CloudKit, push-notification реализованы через систему подписки на события обновления записей заданного типа в пределах базы данных или зоны.

Аналогично другим событиям в iCloud, подписки необходимо создавать при помощи операций над базой данных, а именно операцией CKModifySubscriptionsOperation. Получения подписки по id можно производить при помощи операции CKFetchSubscriptionsOperation.

При создании подписки на записи, на устройство будут приходить нотификации различного вида: CKRecordZoneNotification для изменений в зоне, CKDatabaseNotification для изменений в базе данных или CKQueryNotification для изменений объекта в данной зоне. Последнее из нотификация является самым важным из предыдущих трех и именно оно будет рассматриваться в данной работе.

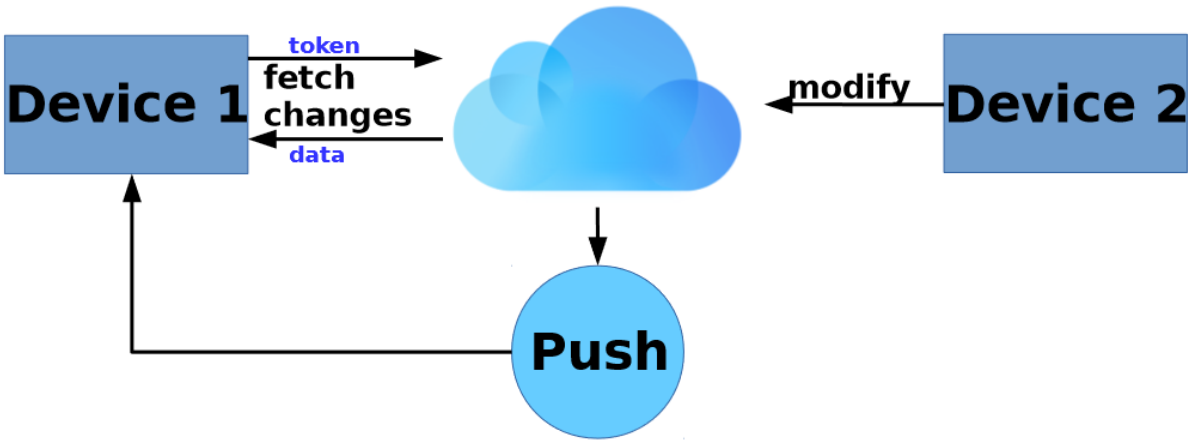

Рис 3.4: Схема получения нотификаций от сервера

3.4 Общая модель Linker

Объединим результаты двух предыдущих параграфов в общую модель Linker, которые позволяет как отправлять изменения данных. Рассмотрим рисунок 3.5.

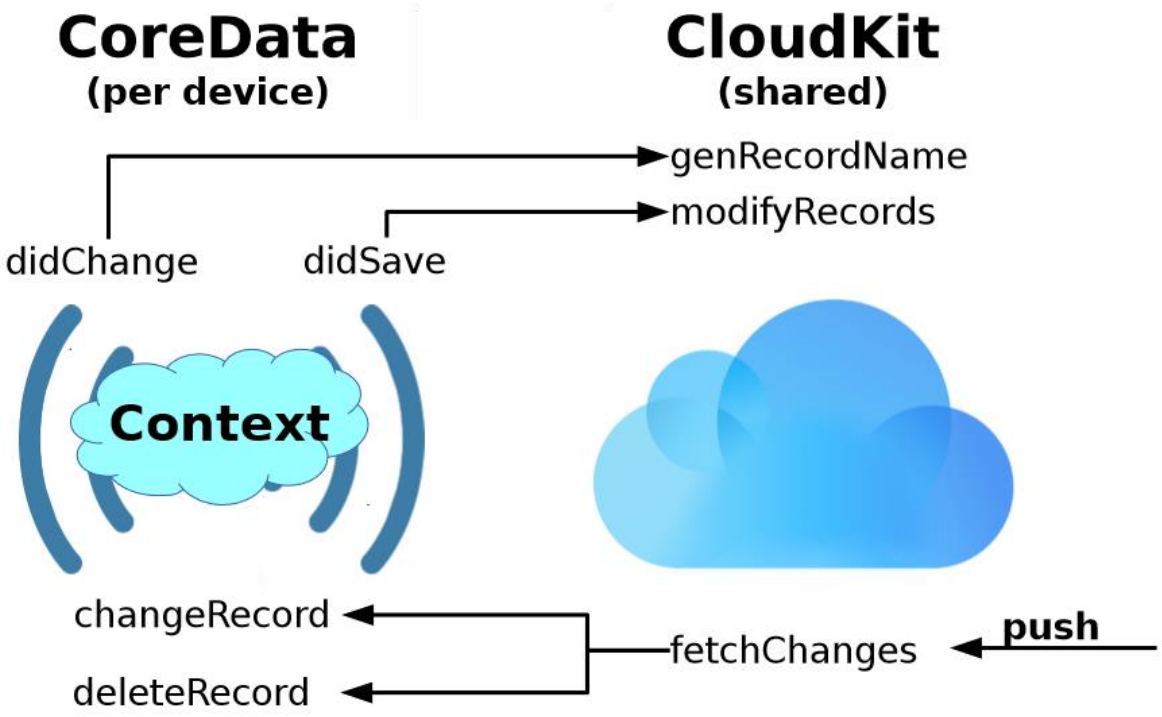

Рис 3.5: Модель Linker

Часть, которая отвечает за контекст, была рассмотрен в параграфе 3.1. Поэтому рассмотрим часть, отвечающую за получение данных из CloudKit. Для того, чтобы начать получение данных из CloudKit, iCloud отправляет на устройство пользователя push-notification. При помощи fetchChanges для зоны, которая поддерживает данную возможность, получаем все изменения зоны и аннулируем пришедшую нотификацию, если требуется. Если зона не позволяет использовать fetchChanges, обрабатываем каждую нотификацию по отдельности, объединяем их в bulk-запрос и получаем новые данные через универсальный способ. Новые записи сортируем на удаляемые и создаваемые и при помощи средств CoreData, обновляем на клиенте данные.

3.5 Оценка производительности

Модель, предложенная в данной работе, не является уникальной, но решение использует современное API для работы с CloudKit и дает хорошую совместимость с любыми продуктами Apple, не требуя сторонних API.

Анализ производительности решения будет приведен на основе сравнения с аналогичным сервисом – Parse. Parse – сервис, предоставляющий аналогичное API для хранения данных на удаленном сервере и синхронизацию данных между различными устройствами. В данной работе были произведены измерения скорости загрузки данных на сервера Apple и Parse. Измерения скорости загрузки 40к записей представлены на рисунке 3.6.

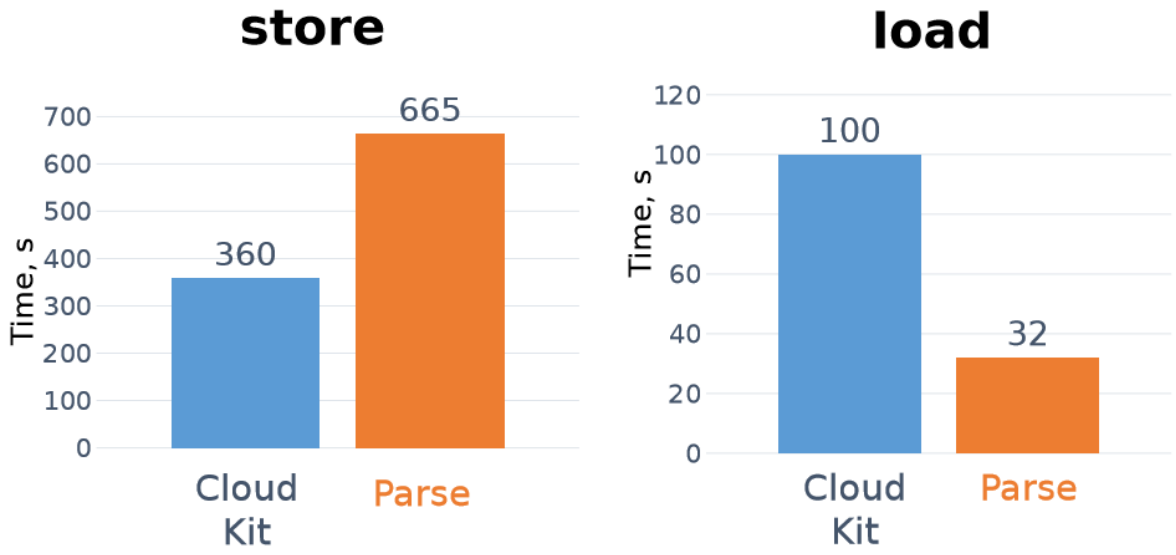

Рис 3.6: Время отправки и получения записей CloudKit и Parse (больше хуже)

Обратим внимание на то, что скорость загрузки данных на сервер CloudKit быстрее чем, для Parse, однако загрузка данных быстрее для Parse. Такое поведение говорит о том, что Parse загружает данные более оптимально, чем CloudKit, но Parse не поддерживает загрузку изменений, что делает Parse более медленным в целом.

Для оценки локального хранилища, рассмотрим размер данных, занимаемый на диске при хранении 100кБ данных. Результаты приведены на рисунке 3.7.

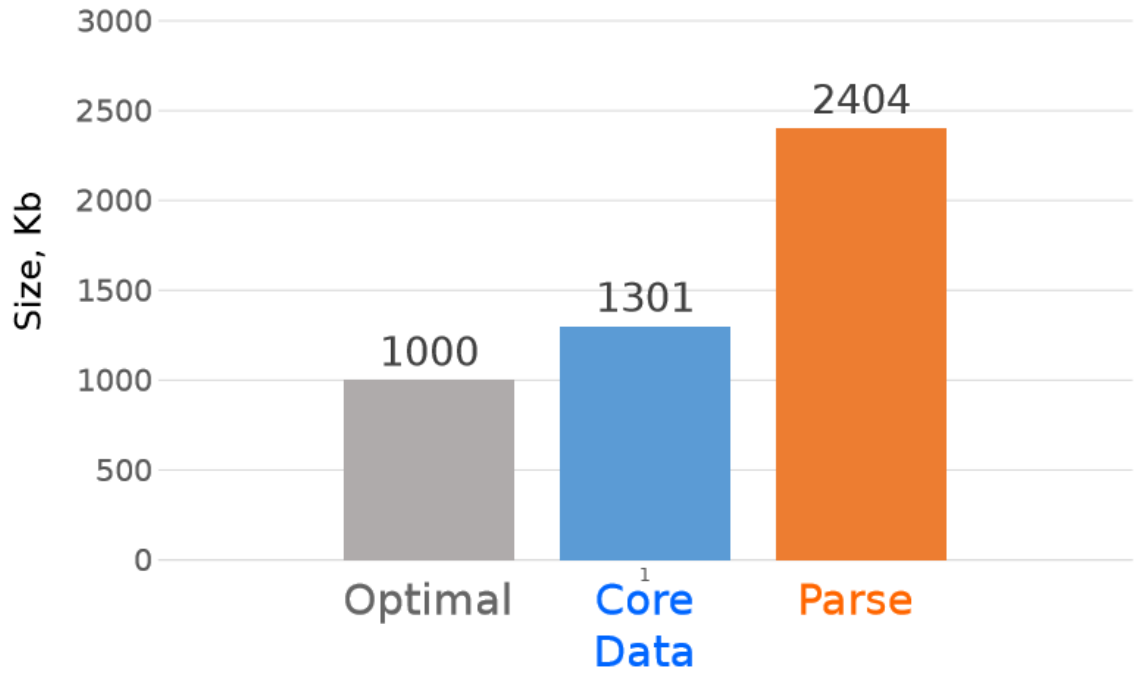

Рис 3.7: Размер хранимых данных CloudKit и Parse (больше хуже)

Видно, что CoreData хранит данные более оптимально, чем Parse, однако база данных Parse позволяет более нетривиальные вещи, чем CoreData, например, автоматическая синхронизация данных между in-memory объектами и данными на жестком диске. Однако, это приводит к худшей производительности базы данных Parse в целом.

ЗАКЛЮЧЕНИЕ

В данной работе были изучены основы работы с фрэмворками CoreData и CloudKit и найдено решение проблемы оптимальной синхронизации данных CoreData при помощи CloudKit. С учетом полученных результатов был реализован прототип Linker, который дает простое и удобное API для синхронизации данных между устройствами.

В ходе работы было произведено сравнение производительности аналогичного решения с полученным в ходе работы и определены недостатки связки CoreData – CloudKit и показаны причины низкой производительности в некоторых тестах.

Linker был успешно использован в проекте “Swedo” для обновления данных CoreData и передачи данных между пользователями обучающей платформы.

БИБЛИОГРАФИЯ

[1] Daniel Eggert. Core Data Overview. Available: https://www.objc.io/issues/4-core-data/core-data-overview/

[2] Matt Gallagher, The difference between Core Data and Database.: Available: https://www.cocoawithlove.com/2010/02/differences-between-core-data-and.html

[3] Андрей Шмиг. Core Data для iOS. Глава №1. Теоретическая часть. Available: https://habrahabr.ru/post/191334/

[4] Apple Developer Documentation, «Persistent Store Types and Behaviours» Apple, Inc. [В Интернете]. Available: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/CoreData/PersistentStoreFeatures.html.

[5] Croath Liu. CloudKit. Available: http://nshipster.com/cloudkit/

[6] Gabriel Theodoropoulos. Working with CloudKit in iOS 8. Available: http://www.appcoda.com/cloudkit-introduction-tutorial/

[7] Apple Developer Documentation. Core Data. Available: https://developer.apple.com/documentation/coredata

[8] Apple Developer Documentation. CloudKit. Available: https://developer.apple.com/icloud/

[9] Parse. http://parseplatform.org/

ПРИЛОЖЕНИЕ А: ИСХОДНЫЙ КОД LINKER

```swift
//
//  CKLinker.swift
//  CKLinker
//
//  Created by Denis Kopyrin on 11/16/16.
//  Copyright © 2016 avriy. All rights reserved.
//

import CloudKit
import CoreData
import Cocoa

enum LinkerError: Error {
    case badEntity
}

public struct LinkerNames {
    static let recordNameAttributeName = "cachedRecordID"
    static let cloudKitMetadataEntityName = "CloudKitMetadata"
    static let tokenAttributeName = "serverChangeToken"
}

/// Linker allows to sync CoreData and CloudKit, making CloudKit
transparent to Client
public class Linker: RecordChangeDelegate, FetchChangeDelegate {
    /// recordZone for CloudKitManager
    private let recordZoneID: CKRecordZoneID
    /// CloudKitManager for Linker with predefined zone and database
    private let cloudKitManager: CloudKitManager

    /// context that is observed by Linker
    private let context: NSManagedObjectContext
    /// CoreDataManager for Linker with predefined context
    private let coreDataManager: CoreDataManager

    /// Child context to perform async save
    private let childContext: NSManagedObjectContext

    private let recordType: String

    private let recordEntity: NSEntityDescription

    private let tokenEntity: NSEntityDescription

    private let predicate: NSPredicate

    private let classType: CKAutoRepresentableModel.Type

    //TODO: Implement control for isReady with LinkerManager
    public private(set) var isReady: Bool

    //MARK: - Init

    /**
     Initializes new Linker with following parameters
```

```
     - Parameters:
        - context: CoreData context that will be observed and all
changes made from it will be saved to CloudKit.
        Every entity should have attribute
CKLinker.recordNameAttribute and database should have entity
CKLinker.cloudKitMetadataEntity with attribute token in
CKLinker.tokenAttribute
        - database: DataBase to use in CloudKit
        - recordZone: Zone to use in CloudKit. Provided Zone should
capable to fetchChanges

     - Returns: Linker that user should save strong link to
     */
    public init(classType: CKAutoRepresentableModel.Type, context:
NSManagedObjectContext, container: CKContainer, database:
CKDatabase, recordZone: CKRecordZone, predicate: NSPredicate =
NSPredicate(value: true)) throws {
        self.recordZoneID = recordZone.zoneID
        self.recordType = classType.recordType

        self.context = context
        self.childContext = NSManagedObjectContext(concurrencyType:
.mainQueueConcurrencyType)
        self.childContext.parent = self.context
        self.coreDataManager = CoreDataManager(context:
self.childContext, parentContext: self.context)
        self.predicate = predicate
        self.isReady = false
        self.classType = classType

        guard let recordEntity =
NSEntityDescription.entity(forEntityName: recordType, in: context)
else {
            throw LinkerError.badEntity
        }
        self.recordEntity = recordEntity

        guard let tokenEntity =
NSEntityDescription.entity(forEntityName:
LinkerNames.cloudKitMetadataEntityName, in: context) else {
            fatalError("Failed to get entity for token")
        }
        self.tokenEntity = tokenEntity

        let token: CKServerChangeToken? =
self.coreDataManager.loadToken(recordType: recordType, predicate:
predicate)

        self.cloudKitManager = CloudKitManager(database: database,
container: container, recordZone: recordZone, token: token)
        self.cloudKitManager.recordChangeDelegate = self
        self.cloudKitManager.initZone(errorHandler:
consoleErrorHandler) { [weak self] in
            self?.isReady = true
        }
```

```
        if token == nil { //Observers will be added after token will
be added
            self.cloudKitManager.generateToken(errorHandler:
consoleErrorHandler)
            self.setupData()
        } else {
            NotificationCenter.default.addObserver(self, selector:
#selector(contextDidSaveHandler(notification:)), name:
.NSManagedObjectContextDidSave, object: context)
            NotificationCenter.default.addObserver(self, selector:
#selector(contextObjectsDidChangeHandler(notification:)), name:
.NSManagedObjectContextObjectsDidChange, object: context)
        }
    }

    deinit {
        NotificationCenter.default.removeObserver(self)
    }

    public func subscribe(completion: @escaping (_ subscriptionID:
String, _ fetchChangeDelegate: FetchChangeDelegate) -> ()) {
        cloudKitManager.createSubscription(entityName: recordType,
searchPredicate: predicate) { [weak self] (subscription, error) in
            if error != nil {
                self?.cloudKitManager.update(errorHandler:
consoleErrorHandler) { }
                //TODO: This is kinda ehh solution
                guard let error = error as? CKError else {
                    return
                }
                guard let subscriptionError =
error.partialErrorsByItemID?.values.first else {
                    return
                }
                let separatedString =
subscriptionError.localizedDescription.components(separatedBy: "'")
                guard separatedString.count > 2 else {
                    return
                }
                let subscriptionID = separatedString[1]
                if let cloudKitManager = self?.cloudKitManager {
                    completion(subscriptionID, cloudKitManager)
                }
            } else {
                if let subscriptionID =
subscription?.subscriptionID, let cloudKitManager =
self?.cloudKitManager {
                    completion(subscriptionID, cloudKitManager)
                }
            }
        }
    }

    private func setupData() {
        cloudKitManager.fetch(entityName: recordType, predicate:
predicate) { [weak self] (records, error) in
            guard let strongSelf = self else { return }
```

```
            guard let records = records else { return }
            for record in records {
                strongSelf.change(record: record, errorHandler:
consoleErrorHandler)
            }

            NotificationCenter.default.removeObserver(strongSelf)
            strongSelf.context.performAndWait {
                do {
                    try strongSelf.coreDataManager.saveContext()
                    try strongSelf.context.save()
                } catch {
                    print("Failed to save: \(error)")
                }
            }
            NotificationCenter.default.addObserver(strongSelf,
selector:
#selector(strongSelf.contextDidSaveHandler(notification:)), name:
.NSManagedObjectContextDidSave, object: strongSelf.context)
            NotificationCenter.default.addObserver(strongSelf,
selector:
#selector(strongSelf.contextObjectsDidChangeHandler(notification:)),
name: .NSManagedObjectContextObjectsDidChange, object:
strongSelf.context)
        }
    }

    public func update(errorHandler: @escaping ErrorHandler,
completionHandler: @escaping () -> ()) {
        cloudKitManager.update(errorHandler: errorHandler,
completionHandler: completionHandler)
    }

    //MARK: - Notification handlers
    @objc private func contextObjectsDidChangeHandler(notification:
Notification) {
        guard let userInfo = notification.userInfo else {
            print("Failed to get userInfo!")
            return
        }

        //If we insert new objects, let's assign a recordName to it
        if let insertedObjects = userInfo[NSInsertedObjectsKey] as?
Set<NSManagedObject> {
            for object in insertedObjects {
                if object.value(forKey:
LinkerNames.recordNameAttributeName) as? String != nil { continue }
                guard let objectRecordType = object.entity.name else
{ continue }
                guard recordType == objectRecordType else { continue
}
                let record = CKRecord(recordType: recordType,
zoneID: recordZoneID)
                object.setValue(record.recordID.recordName, forKey:
LinkerNames.recordNameAttributeName)
            }
        }
    }
```

```
    @objc func contextDidSaveHandler(notification: Notification) {
        guard let userInfo = notification.userInfo else {
            print("Failed to get userInfo!")
            return
        }


        if let insertedObjects = userInfo[NSInsertedObjectsKey] as?
Set<NSManagedObject> {
            print("Generating dict (insert)")
            let records = generateRecords(objects: insertedObjects)
            cloudKitManager.insert(records: records, entityName:
recordType)
        }

        if let updatedObjects = userInfo[NSUpdatedObjectsKey] as?
Set<NSManagedObject> {
            print("Generating dict (update)")
            let records = generateRecords(objects: updatedObjects)
            cloudKitManager.insert(records: records, entityName:
recordType)
        }

        if let deletedObjects = userInfo[NSDeletedObjectsKey] as?
Set<NSManagedObject> {
            let records = generateRecords(objects: deletedObjects)
            cloudKitManager.delete(recordIDs: records.map {
$0.recordID }, entityName: recordType)
        }
    }

    //MARK: - CloudKit object parser
    private func generateRecords(objects: Set<NSManagedObject>) ->
[CKRecord] {
        return objects.flatMap { (object) in
            guard let objectRecordType = object.entity.name else {
return nil }
            guard objectRecordType == recordType else { return nil }
            guard let objectModeled = object as?
CKAutoRepresentableModel else { return nil }
            return try? objectModeled.record()
        }
    }

    //MARK: - RecordChangeDelegate
    func change(record: CKRecord, errorHandler: @escaping
ErrorHandler) {
        guard recordType == record.recordType else { return }
        let object = coreDataManager.makeObject(recordName:
record.recordID.recordName, entity: recordEntity)
        object.setValue(record.recordID.recordName, forKey:
LinkerNames.recordNameAttributeName)
        guard let objectModel = object as? CKRepresentableModel else
{ return }

        do {
            try objectModel.updateWith(record: record)
```

```
            } catch {
                errorHandler(error)
            }
        }

        func deleteRecord(with recordID: CKRecordID) {
            _ = coreDataManager.deleteObjects(recordName:
recordID.recordName, entity: recordEntity)
        }

        func bulkEnd(token: CKServerChangeToken?, errorHandler:
@escaping ErrorHandler) {
            NotificationCenter.default.removeObserver(self)
            if let token = token {
                coreDataManager.saveToken(recordType: recordType,
predicate: predicate, token: token)
            }

            context.performAndWait { [weak self] in
                do {
                    try self?.coreDataManager.saveContext()
                    try self?.context.save()
                } catch {
                    errorHandler(error)
                }
            }
            NotificationCenter.default.addObserver(self, selector:
#selector(contextDidSaveHandler(notification:)), name:
.NSManagedObjectContextDidSave, object: context)
            NotificationCenter.default.addObserver(self, selector:
#selector(contextObjectsDidChangeHandler(notification:)), name:
.NSManagedObjectContextObjectsDidChange, object: context)
        }

        //MARK: - FetchChangesDelegate
        public func fetchChanges(for notification: CKNotification?,
errorHandler: @escaping ErrorHandler) {
            cloudKitManager.fetchChanges(for: notification,
errorHandler: errorHandler)
        }
}
```

```
//
//  CloudManager.swift
//  CKLinker
//
//  Created by Denis Kopyrin on 07.11.16.
//  Copyright © 2016 avriy. All rights reserved.
//

import Foundation
import CloudKit

fileprivate func consoleHandler<T> (value: T, error: Error?) {
    if let error = error {
        print(error)
    }
}

fileprivate extension Array {
    func split(bySize chunkSize: Int) -> [[Element]] {
        let count = self.count
        let dividers = stride(from: 0, to: count, by: chunkSize)
        return dividers.map { (rightSide) in
            let leftSide = rightSide + chunkSize < count ? rightSide
+ chunkSize: count
            return Array(self[rightSide ..< leftSide])
        }
    }
}

class CloudKitManager: FetchChangeDelegate {
    static let bulkSize = 400

    var serverChangeToken: CKServerChangeToken?
    var recordChangeDelegate: RecordChangeDelegate?

    let container: CKContainer
    let recordZone: CKRecordZone
    let database: CKDatabase

    //MARK: - Inits
    init(database: CKDatabase, container: CKContainer, recordZone:
CKRecordZone, token: CKServerChangeToken?) {
        self.container = container
        self.database = database
        self.recordZone = recordZone
        self.serverChangeToken = token
    }

    public func initZone(errorHandler: @escaping ErrorHandler,
completionHandler: @escaping () -> ()) {
        guard self.recordZone != CKRecordZone.default() else {
            return
        }

        let operation =
CKModifyRecordZonesOperation(recordZonesToSave: [recordZone],
recordZoneIDsToDelete: nil)
```

```
        operation.modifyRecordZonesCompletionBlock = { (_, _, error)
in
            if let error = error {
                errorHandler(error)
            }
            completionHandler()
        }
        database.add(operation)
    }

    func fetch(entityName: String, predicate: NSPredicate,
completion: @escaping (_ record: [CKRecord]?, _ error: Error?) ->
Void) {
        let query = CKQuery(recordType: entityName, predicate:
predicate)
        let queryOperation = CKQueryOperation(query: query)
        queryOperation.zoneID = recordZone.zoneID
        var records: [CKRecord] = []

        queryOperation.recordFetchedBlock = { (record) in
            records.append(record)
        }
        queryOperation.queryCompletionBlock = { (_, error) in //Yes,
this is bad, but for now I'll take that
            completion(records, error as Error?)
        }

        database.add(queryOperation)
    }

    func fetch(recordID: CKRecordID, _ completion: @escaping (_
record: CKRecord?, _: Error?) -> Void) {
        database.fetch(withRecordID: recordID) { (record, error) in
            completion(record, error)
        }
    }

    func delete(recordID: CKRecordID, _ completion: @escaping (_
recordID: CKRecordID?, _: Error?) -> Void) {
        database.delete(withRecordID: recordID) { (recordID, error)
in
            completion(recordID, error)
        }
    }

    /// Limit resources this function can take to ~400 records/ids
    /// This operation supports download of big amount of records!
    func modify(recordsToSave: [CKRecord], recordIDsToDelete:
[CKRecordID], entityName: String, completion: @escaping (_ records:
[CKRecord]?, _ recordIDs: [CKRecordID]?, _ error: Error?) -> Void) {
        let group = DispatchGroup()
        let queue = OperationQueue()
        queue.maxConcurrentOperationCount = 1

        var totalSavedRecords: [CKRecord] = []
        var totalDeletedRecordIDs: [CKRecordID] = []
        var totalErrors: [Error] = []
```

```
        let waitQueue = DispatchQueue(label: "Wait Queue")

        var modifyBlock: (([CKRecord]?, [CKRecordID]?, Error?) ->
Void)!
        modifyBlock = { [weak self] (savedRecords, deletedRecordIDs,
error) in
            if let error = error as? CKError {
                switch error {
                case CKError.requestRateLimited:
                    guard let timeToWait =
error.userInfo[CKErrorRetryAfterKey] as? Int else {
                        break
                    }
                    let deadlineTime = DispatchTime.now() +
.seconds(timeToWait)
                    waitQueue.asyncAfter(deadline: deadlineTime) {
                        self?.modify(recordsToSave: recordsToSave,
recordIDsToDelete: recordIDsToDelete, entityName: entityName,
completion: modifyBlock)
                    }
                default:
                    if let records = savedRecords {
totalSavedRecords.append(contentsOf: records) }
                    if let recordIDs = deletedRecordIDs {
totalDeletedRecordIDs.append(contentsOf: recordIDs) }
                    totalErrors.append(error)
                    group.leave()
                }
            } else {
                if let records = savedRecords {
totalSavedRecords.append(contentsOf: records) }
                if let recordIDs = deletedRecordIDs {
totalDeletedRecordIDs.append(contentsOf: recordIDs) }
                group.leave()
            }
        }

        recordsToSave.split(bySize:
CloudKitManager.bulkSize).forEach {
            let operation = CKModifyRecordsOperation(recordsToSave:
$0, recordIDsToDelete: nil)
            operation.qualityOfService =
QualityOfService.userInteractive
            operation.isAtomic = false
            operation.savePolicy = .allKeys
            group.enter()
            operation.modifyRecordsCompletionBlock = modifyBlock

            operation.database = database
            queue.addOperation(operation)
        }

        recordIDsToDelete.split(bySize:
CloudKitManager.bulkSize).forEach {
            let operation = CKModifyRecordsOperation(recordsToSave:
nil, recordIDsToDelete: $0)
            group.enter()
```

```
            operation.qualityOfService =
QualityOfService.userInteractive
            operation.modifyRecordsCompletionBlock = modifyBlock
            operation.database = database
            queue.addOperation(operation)
        }

        group.notify(queue: DispatchQueue.main) {
            completion(totalSavedRecords, totalDeletedRecordIDs,
totalErrors.first)
        }
    }

    func insert(records: [CKRecord], entityName: String, completion:
@escaping (_ records: [CKRecord]?, _ error: Error?) -> Void =
consoleHandler) {
        DispatchQueue.main.async {
            self.modify(recordsToSave: records, recordIDsToDelete:
[], entityName: entityName) { (records, _, error) in
                completion(records, error)
            }
        }
    }

    func delete(recordIDs: [CKRecordID], entityName: String,
completion: @escaping (_ records: [CKRecordID]?, _ error: Error?) ->
Void = consoleHandler) {
        modify(recordsToSave: [], recordIDsToDelete: recordIDs,
entityName: entityName) { (_, recordIDs, error) in
            completion(recordIDs, error)
        }
    }

    //MARK: - Subscriptions
    func createSubscription(entityName: String, searchPredicate:
NSPredicate, completion: @escaping (_ subscription: CKSubscription?,
_ error: Error?) -> Void = consoleHandler) {
        let subscription = CKQuerySubscription(recordType:
entityName, predicate: searchPredicate, options:
[.firesOnRecordUpdate, .firesOnRecordDeletion,
.firesOnRecordCreation])
        subscription.notificationInfo = createNotification()

        let operation =
CKModifySubscriptionsOperation(subscriptionsToSave: [subscription],
subscriptionIDsToDelete: nil)
        operation.modifySubscriptionsCompletionBlock = { (subscr, _,
error) in
            completion(subscr?.first, error)
        }

        database.add(operation)
    }

    func removeSubscription(entityName: String, searchPredicate:
NSPredicate) {
        let subscription = CKQuerySubscription(recordType:
entityName, predicate: searchPredicate, options:
```

```
[.firesOnRecordUpdate, .firesOnRecordDeletion,
.firesOnRecordCreation])
        let operation =
CKModifySubscriptionsOperation(subscriptionsToSave: [subscription],
subscriptionIDsToDelete: nil)

        database.add(operation)
    }

    func listSubscriptions() {
        database.fetchAllSubscriptions { (_, _) in }
    }

    func createNotification() -> CKNotificationInfo {
        let notificationInfo = CKNotificationInfo()
        notificationInfo.shouldSendContentAvailable = true

NSApplication.shared().registerForRemoteNotifications(matching:
NSRemoteNotificationType.badge)
        notificationInfo.shouldBadge = true
        return notificationInfo
    }

    //MARK: - Token
    func generateToken(errorHandler: @escaping ErrorHandler) {
        let operation =
CKFetchNotificationChangesOperation(previousServerChangeToken:
serverChangeToken)
        operation.fetchNotificationChangesCompletionBlock = { [weak
self] (token, error) in
            self?.serverChangeToken = token
            if let error = error {
                errorHandler(error)
            }
        }

        container.add(operation)
    }

    //MARK: - Notification callback
    func fetchChanges(for notification: CKNotification?,
errorHandler: @escaping ErrorHandler) {
        if let notification = notification {
            process(notification: notification)
        }

        update(errorHandler: errorHandler) { }
    }

    func update(errorHandler: @escaping ErrorHandler,
completionHandler: @escaping () -> ()) {
        let operation =
CKFetchNotificationChangesOperation(previousServerChangeToken:
serverChangeToken)

        var notifications: [CKNotificationID] = []
```

```
        operation.notificationChangedBlock = { [weak self]
(notification) in
            self?.process(notification: notification)
            notifications.append(notification.notificationID!)
        }

        operation.fetchNotificationChangesCompletionBlock = { [weak
self] (token, error) in
            self?.serverChangeToken = token
            if let error = error {
                errorHandler(error)
            }
            self?.mark(notifications: notifications, errorHandler:
consoleErrorHandler)
            self?.recordChangeDelegate?.bulkEnd(token: token,
errorHandler: consoleErrorHandler)
            completionHandler()
        }

        CKContainer.default().add(operation)
    }

    private func process(notification: CKNotification) {
        guard notification.notificationType ==
CKNotificationType.query else {
            return
        }

        let queryNotification = notification as! CKQueryNotification
        switch(queryNotification.queryNotificationReason) {
        case .recordCreated, .recordUpdated:
            guard let recordID = queryNotification.recordID else {
                return
            }

            fetch(recordID: recordID) { [weak self] (record, error)
in
                guard let record = record else {
                    return
                }
                self?.recordChangeDelegate?.change(record: record,
errorHandler: consoleErrorHandler)
            }
        case .recordDeleted:
            guard let recordID = queryNotification.recordID else {
                return
            }

            delete(recordID: recordID) { [weak self] (recordID,
error) in
                guard let recordID = recordID else {
                    return
                }
                self?.recordChangeDelegate?.deleteRecord(with:
recordID)
            }
        }
    }
```

```
    func mark(notifications: [CKNotificationID], errorHandler:
@escaping ErrorHandler) {
        let operation =
CKMarkNotificationsReadOperation(notificationIDsToMarkRead:
notifications)
        operation.markNotificationsReadCompletionBlock = { (_,
error) in
            if let error = error {
                errorHandler(error)
            }
        }
        CKContainer.default().add(operation)
    }
}
```

```
//
//  CoreDataManager.swift
//  CKLinker
//
//  Created by Denis Kopyrin on 11/17/16.
//  Copyright © 2016 avriy. All rights reserved.
//


import Cocoa
import CoreData
import CloudKit

class CoreDataManager {
    let context: NSManagedObjectContext
    let parentContext: NSManagedObjectContext

    init(context: NSManagedObjectContext, parentContext:
NSManagedObjectContext) {
        self.context = context
        self.parentContext = parentContext

        context.mergePolicy =
NSMergeByPropertyObjectTrumpMergePolicy
        parentContext.mergePolicy =
NSMergeByPropertyObjectTrumpMergePolicy
    }

    func saveContext() throws {
        try context.save()
    }

    //MARK: - General object
    func makeObject(recordName: String, entity: NSEntityDescription)
-> NSManagedObject {
        return findObject(recordName: recordName, entityName:
entity.name!) ?? NSManagedObject(entity: entity, insertInto:
context)
    }

    func findObject(recordName: String, entityName: String) ->
NSManagedObject? {
        let searchResults = try? getObjects(byPredicate:
NSPredicate(format: LinkerNames.recordNameAttributeName + " == %@",
recordName), entityName: entityName)
        return searchResults??.first
    }

    func getObjects(byPredicate predicate: NSPredicate, entityName:
String) throws -> [NSManagedObject]? {
        let fetchRequest: NSFetchRequest<NSManagedObject> =
NSFetchRequest(entityName: entityName)
        fetchRequest.predicate = predicate

        return try parentContext.fetch(fetchRequest)
    }
```

```
    func deleteObjects(byPredicate predicate: NSPredicate, entity:
NSEntityDescription) -> [NSManagedObject]? {
        guard let objectsOptional = try? getObjects(byPredicate:
predicate, entityName: entity.name!) else {
            return nil
        }
        guard let objects = objectsOptional else {
            return nil
        }
        parentContext.performAndWait {
            objects.forEach { (object) in
                self.parentContext.delete(object)
            }
        }
        return objects
    }

    func deleteObjects(recordName: String, entity:
NSEntityDescription) -> [NSManagedObject]? {
        return deleteObjects(byPredicate: NSPredicate(format:
LinkerNames.recordNameAttributeName + " == %@", recordName), entity:
entity)
    }

    //MARK: - Token
    func loadToken(recordType: String, predicate: NSPredicate) ->
CKServerChangeToken? {
        let group = DispatchGroup()
        group.enter()

        var token: CKServerChangeToken? = nil
        NSPersistentContainer.getMetadataValue(forKey:
LinkerNames.tokenAttributeName + "_" + recordType + "_" +
predicate.predicateFormat, errorHandler: consoleErrorHandler) {
(data: Data?) in
            defer { group.leave() }
            guard let data = data else {
                return
            }
            token = NSKeyedUnarchiver.unarchiveObject(with: data)
as? CKServerChangeToken
        }
        group.wait()
        return token
    }

    func saveToken(recordType: String, predicate: NSPredicate,
token: CKServerChangeToken, errorHandler: @escaping ErrorHandler =
consoleErrorHandler) {
        let data = NSKeyedArchiver.archivedData(withRootObject:
token)
        DispatchQueue.main.async {
            NSPersistentContainer.setMetadataValue(forKey:
LinkerNames.tokenAttributeName + "_" + recordType + "_" +
predicate.predicateFormat, value: data, errorHandler: errorHandler)
        }
    }
}
```

```swift
//
//  FetchChangeDelegate.swift
//  CKLinker
//
//  Created by Denis Kopyrin on 11/17/16.
//  Copyright © 2016 avriy. All rights reserved.
//

import Foundation
import CloudKit

public protocol FetchChangeDelegate {
    func fetchChanges(for notification: CKNotification?,
errorHandler: @escaping ErrorHandler)
}

//
//  RecordChangeDelegate.swift
//  CKLinker
//
//  Created by Denis Kopyrin on 10/30/16.
//  Copyright © 2016 avriy. All rights reserved.
//

import Foundation
import CloudKit

/// 2 functions on top will be called and does not expect you to
save anything on disk
protocol RecordChangeDelegate {
    /// This function should either create or update record
    func change(record: CKRecord, errorHandler: @escaping
ErrorHandler)
    func deleteRecord(with recordID: CKRecordID)

    func bulkEnd(token: CKServerChangeToken?, errorHandler:
@escaping ErrorHandler)
}
```

```
//
//  LinkerManager.swift
//  Swedo
//
//  Created by Denis Kopyrin on 2/12/17.
//  Copyright © 2017 avriy. All rights reserved.
//

import Foundation
import CloudKit

public class LinkerManager: FetchChangeDelegate {
    private var linkers: [String: FetchChangeDelegate] =
[:]//SubcriptionID -> Linkers

    private let cloudContainer: CKContainer
    private let persistentContainer: NSPersistentContainer

    public var contextForLazyLinkers: NSManagedObjectContext
    public var containerForLazyLinkers: CKContainer
    public var databaseForLazyLinkers: CKDatabase
    public var recordZoneForLazyLinkers: CKRecordZone

    public init(persistentContainer: NSPersistentContainer,
cloudContainer: CKContainer) {
        self.persistentContainer = persistentContainer
        self.cloudContainer = cloudContainer

        self.contextForLazyLinkers = persistentContainer.viewContext
        self.containerForLazyLinkers = cloudContainer
        self.databaseForLazyLinkers =
cloudContainer.publicCloudDatabase
        self.recordZoneForLazyLinkers = CKRecordZone.default()

        let notificationCenter = NotificationCenter.default
        notificationCenter.addObserver(forName:
.didReceiveRemoteNotification, object: nil, queue: nil, using:
fetchChangesNotificationListener)
    }

    public func fetchChanges(for notification: CKNotification?,
errorHandler: @escaping ErrorHandler) {
        guard let subscriptionID = notification?.subscriptionID else
{ return }
        let linker = linkers[subscriptionID]
        if let linker = linker {
            linker.fetchChanges(for: notification, errorHandler:
errorHandler)
        } else {
            let operation =
CKFetchSubscriptionsOperation(subscriptionIDs: [subscriptionID])
            var linker: Linker? = nil

            operation.fetchSubscriptionCompletionBlock = { [weak
self] (subscriptionDictionary, error) in
                if let error = error {
                    errorHandler(error)
                    return
```

```
                }
                guard let subscriptionDictionary =
subscriptionDictionary else {
                    print()
                    return
                }
                guard let subscription =
subscriptionDictionary[subscriptionID] else {
                    return
                }
                guard let querySubscription = subscription as?
CKQuerySubscription else {
                    return
                }
                let predicate = querySubscription.predicate
                let recordType = querySubscription.recordType
                guard let recordClass =
NSClassFromString(recordType) else {
                    return
                }
                guard let autoRepresentableModelType = recordClass
as? CKAutoRepresentableModel.Type else {
                    return
                }
                guard let strongSelf = self else {
                    return
                }

                DispatchQueue.main.sync {
                    do {
                        linker = try Linker(classType:
autoRepresentableModelType, context:
strongSelf.contextForLazyLinkers, container:
strongSelf.containerForLazyLinkers, database:
strongSelf.databaseForLazyLinkers, recordZone:
strongSelf.recordZoneForLazyLinkers, predicate: predicate)
                    } catch {
                        errorHandler(error)
                    }
                }
                self?.linkers[subscriptionID] = linker
                linker?.fetchChanges(for: notification,
errorHandler: errorHandler)
            }

            databaseForLazyLinkers.add(operation)
        }
    }

    public func createLinker(classType:
CKAutoRepresentableModel.Type, context: NSManagedObjectContext,
container: CKContainer, database: CKDatabase, recordZone:
CKRecordZone, predicate: NSPredicate = NSPredicate(value: true))
throws -> Linker {
        let linker = try Linker(classType: classType, context:
context, container: container, database: database, recordZone:
recordZone, predicate: predicate)
        linker.subscribe() { [weak self] (id, linker) in
```

```
            self?.linkers[id] = linker
        }
        return linker
    }

    //MARK: - Notification Listener
    public func fetchChangesNotificationListener(notification:
Notification) {
        let cknNotification =
CKNotification(fromRemoteNotificationDictionary:
notification.userInfo as! [String: NSObject])
        if cknNotification.notificationType == .query, let _ =
cknNotification as? CKQueryNotification {
            fetchChanges(for: cknNotification, errorHandler:
consoleErrorHandler)
        } else {
            fetchChanges(for: nil, errorHandler:
consoleErrorHandler)
        }
    }
}

public extension Notification.Name {
    static let didReceiveRemoteNotification =
Notification.Name("didRecieveRemoteNotification")
}
```

```
//
//  CoursesVM.swift
//  SWEDO
//
//Example of Linker Usage

import Cocoa
import SwedoKit
import CloudKit

class CoursesContext: ContainerContext {

    let persistentContainer: NSPersistentContainer
    let cloudContainer: CKContainer
    let linkerManager: LinkerManager

    let user: CourseUser

    init(persistentContainer pc: NSPersistentContainer,
cloudContainer cc: CKContainer, linkerManager lm: LinkerManager,
user: CourseUser) {
        persistentContainer = pc
        cloudContainer = cc
        linkerManager = lm
        self.user = user
    }
}

class CoursesVM: NSObject, ViewModelProtocol {

    var updatedContext: CoursesContext
    private let publicCoursePredicate = NSPredicate(format:
"courseVisibility == 0")
    private let subscriptionPredicate: NSPredicate
    let linkerCourse: Linker?
    let linkerSubscription: Linker?

    required init(context: CoursesContext) {
        updatedContext = context
        let container = CKContainer.default()
        let database = container.publicCloudDatabase
        let recordZone = CKRecordZone.default()
        do {
            self.linkerCourse = try
updatedContext.linkerManager.createLinker(classType: Course.self
,context: context.persistentContainer.viewContext, container:
container, database: database, recordZone: recordZone, predicate:
publicCoursePredicate)
        } catch {
            self.linkerCourse = nil
            print("Failed to create Linker: \(error)")
        }

        guard let cachedRecordID = context.user.cachedRecordID else
{
            fatalError("cachedRecordID of courseUser is nil")
        }
```

```
        subscriptionPredicate = NSPredicate(format:
"student.cachedRecordID == %@", cachedRecordID)
        do {
            self.linkerSubscription = try
updatedContext.linkerManager.createLinker(classType:
CourseSubscription.self, context:
context.persistentContainer.viewContext, container: container,
database: database, recordZone: recordZone, predicate:
subscriptionPredicate)
        } catch {
            self.linkerSubscription = nil
            print("Failed to create Linker: \(error)")
        }
    }

    var isProgressing: Bool  {
        return isProgressingForCourse ||
isProgressingForSubscription
    }
    dynamic var isProgressingForCourse: Bool = false
    dynamic var isProgressingForSubscription: Bool = false

    class func keyPathsForValuesAffectingIsProgressing() ->
Set<String> {
        return Set([#keyPath(isProgressingForCourse),
#keyPath(isProgressingForSubscription)])
    }

    func update(errorHandler eh: @escaping ErrorHandler) {
        isProgressingForCourse = true
        linkerCourse?.update(errorHandler: { [weak self] error in
            self?.isProgressingForCourse = false
            eh(error)
            }, completionHandler: { [weak self] in
                self?.isProgressingForCourse = false
        })
        isProgressingForSubscription = true
        linkerSubscription?.update(errorHandler: { [weak self] error
in
            self?.isProgressingForSubscription = false
            eh(error)
            }, completionHandler: { [weak self] in
                self?.isProgressingForSubscription = false
        })
    }
}
```